2013

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

Kai-Shin Lu
Iowa State University

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

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

LIST OF TABLES

ACKNOWLEDGEMENTS

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 (**T**esting and **AI**-Planning Based **R**efactoring **P**ath 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

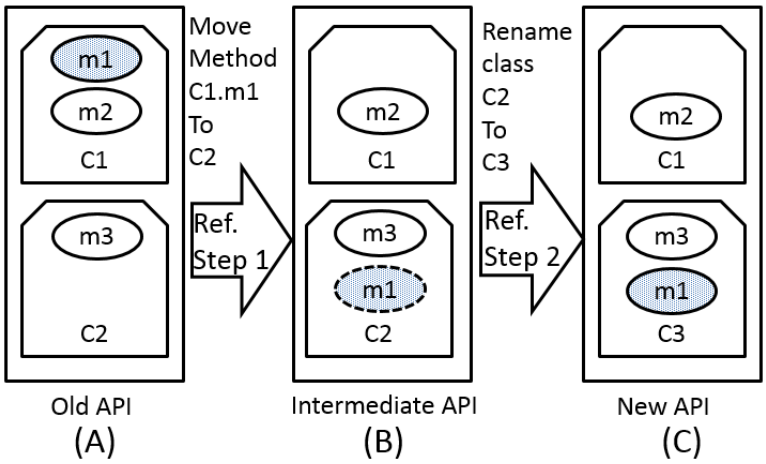

Figure 1.1 An example of Temporal-Dependent Refactoring Steps (TDRS): (A) shows the old API of this component, (B) shows the intermediate API, and (C) shows the new API. Method *m1* in class *C1* (denoted as *C1.m1*) was moved to class *C2* and then C2 was renamed to *C3*. These two refactoring steps, "move method *C1.m1* to *C2*" and "rename class *C2* to *C3*", are TDRS.

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 **a**utomatic **l**oad-**t**ime **a**daptation framework for refactoring-based evolution of software component. ALTA is an Aspect-Oriented-Programming (AOP) [21] 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.

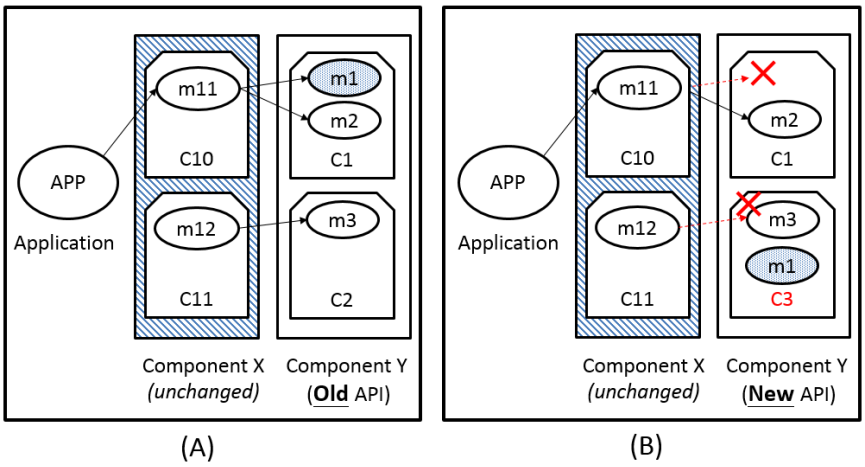

Figure 1.2 Example of a third-party API-caller. (A) Before upgrading component Y. (B) After upgrading component Y. After upgrading Y, X and Y became incompatible.

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.

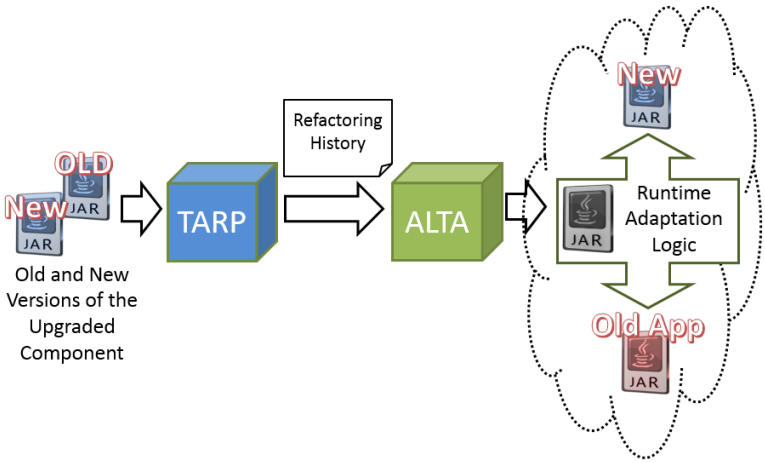

Figure 1.3 Overview of the proposed solution. The white arrows (with solid lines) represent data flows.

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.

4. Can support Temporal-Dependent Refactoring Steps (TDRS).

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]

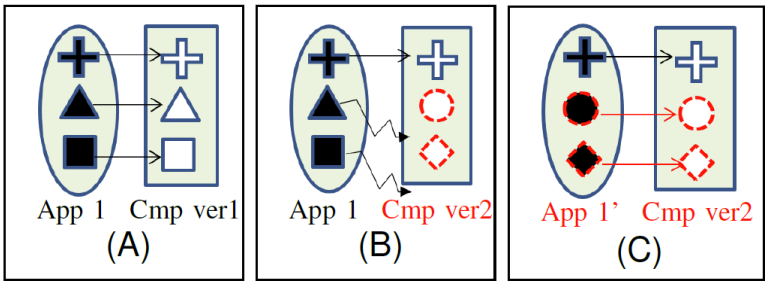

Figure 2.1 The main idea of fixing incompatibility problems by migrating application. (A) System before evolution. (B) System after evolution. Application 1 (App 1) and the upgraded Component (Cmp ver 2) are not compatible. (C) App 1 has been migrated to App 1'; therefore compatibility problems were fixed.

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 **a**utomatic **l**oad-**t**ime **a**daptation 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

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

Item	Feature	CatchUp!	ReBA	Comeback!	ALTA
1	Static modification target	*Application*	*Component*	*Component*	None
2	Can work without source code?	*No*	Yes	Yes	Yes
3	Support load-time modification?	*No*	*No*	*No*	Yes
4	Support conflict-making refactorings?	Yes	*No*	*No*	Yes

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.

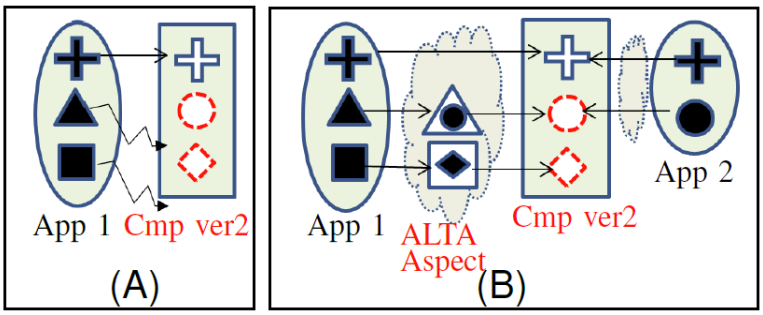

Figure 2.2 The main idea of ALTA's load-time adaptation. (A) System after evolution. Application 1 (App 1) and the upgraded Component (Cmp ver 2) are not compatible. (B) With ALTA Aspect, App 1 can run with Cmp ver 2 correctly. App 2 also runs under ALTA, but because it was designed with Cmp ver2, ALTA 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 refactorings 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:

```
1 public void printCode(int code){
2   System.out.println("Code="+code);
3 }
```

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 at. proposed a framework to support COTS composition [29], Sanchez et al. used AOP to adapt synchronization policies [27]. However, ALTA is the first

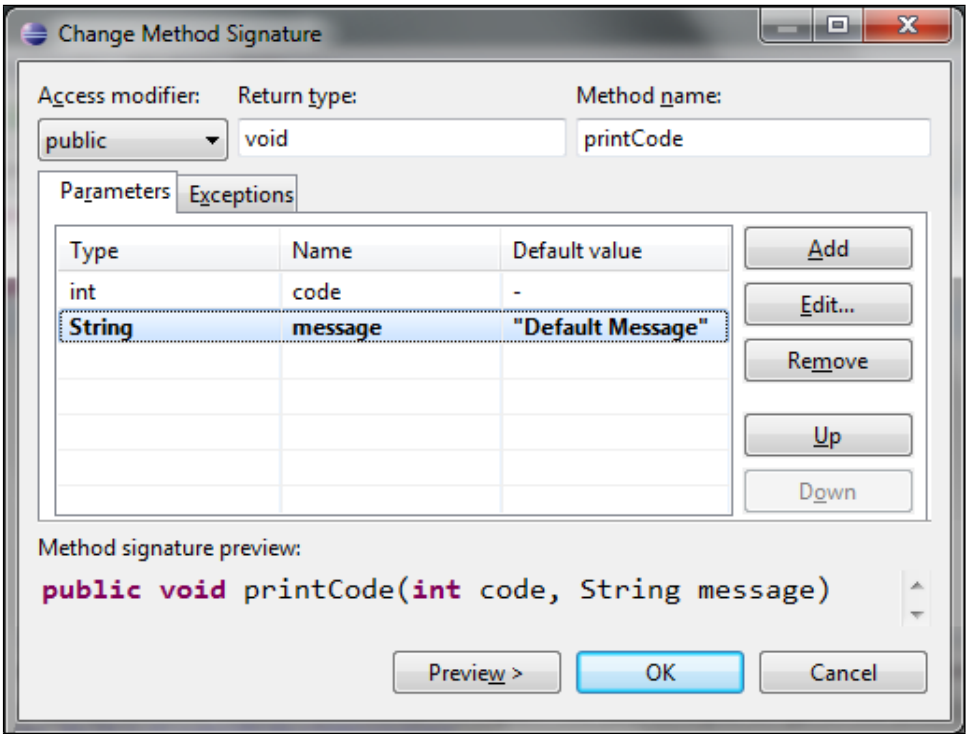

Figure 2.3 Refactoring wizard in Eclipse.

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! [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 refactor 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 refactor action one by one to the **application**; therefore CatchUp! will upgrade the

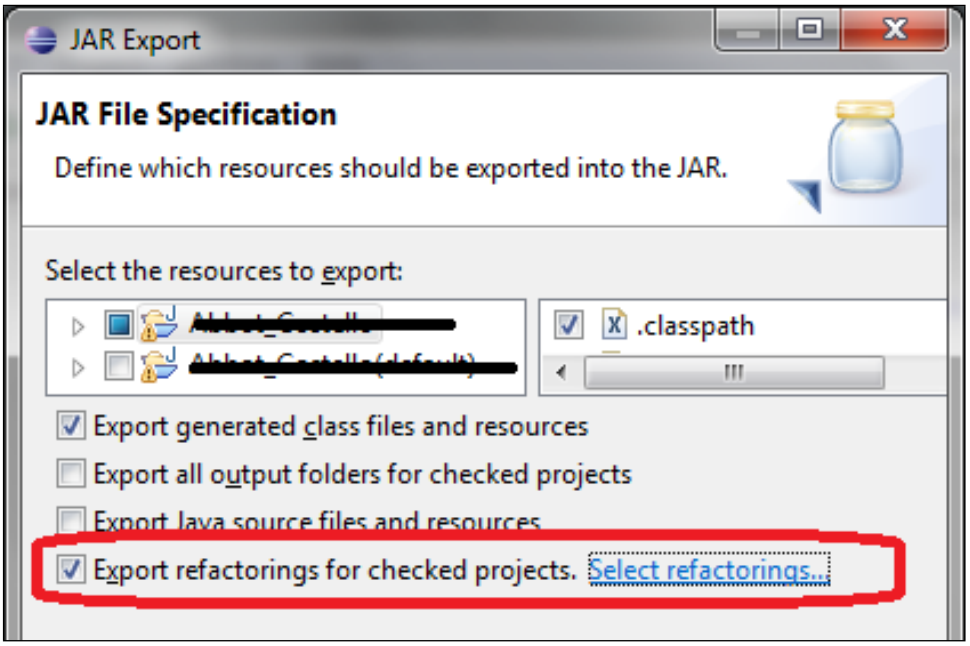

Figure 2.4 Exporting refactoring information in Eclipse.

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 refactor action to create a backward-compatible layer.

```
<?xml version="1.0" encoding="UTF-8"?>
- <session version="1.0">
    <refactoring version="1.0" references="true" project="ALTA_LibAfterChange"
      name="getFormatter_REN" input="/src<org.apache.commons.lang3.text
      {CompositeFormat.java[CompositeFormat~getFormatter"
      id="org.eclipse.jdt.ui.rename.method" flags="589830"
      description="Rename method 'getFormatter'" deprecate="false"
      delegate="false" comment="Rename method
      'org.apache.commons.lang3.text.CompositeFormat.getFormatter()' to
      'getFormatter_REN' - Original project: 'ALTA_LibAfterChange' - Original
      element:
      'org.apache.commons.lang3.text.CompositeFormat.getFormatter()' -
      Renamed element:
      'org.apache.commons.lang3.text.CompositeFormat.getFormatter_REN()'
      - Update references to refactored element"/>
    <refactoring version="1.0" references="true" project="ALTA_LibAfterChange"
      name="EqualsBuilder_REN"
      input="/src<org.apache.commons.lang3.builder{EqualsBuilder.java
      [EqualsBuilder" id="org.eclipse.jdt.ui.rename.type" flags="589830"
      description="Rename type 'EqualsBuilder'" comment="Rename type
      'org.apache.commons.lang3.builder.EqualsBuilder' to
      'EqualsBuilder_REN' - Original project: 'ALTA_LibAfterChange' - Original
      element: 'org.apache.commons.lang3.builder.EqualsBuilder' - Renamed
      element: 'org.apache.commons.lang3.builder.EqualsBuilder_REN' -
      Update references to refactored element - Update textual occurrences in
      comments and strings" textual="false" similarDeclarations="false"
      qualified="false" matchStrategy="1"/>
  </session>
```

Figure 2.5 Sample refactoring history file (in XML format).

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.

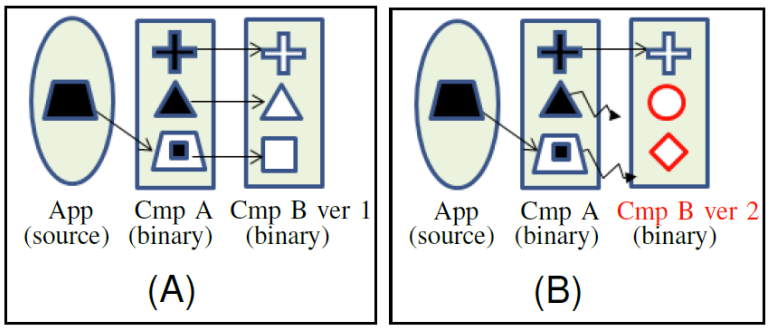

Figure 2.6 Example of a third-party API-caller. (A) Before upgrading Cmp (i.e., component) B . (B) After upgrading Cmp B. Cmp B and A became incompatible, while only the source of App (i.e., the application) is available.

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.

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

Figure 2.9 The stub generated by Comeback!

2.4 Method

2.4.1 Framework and Process

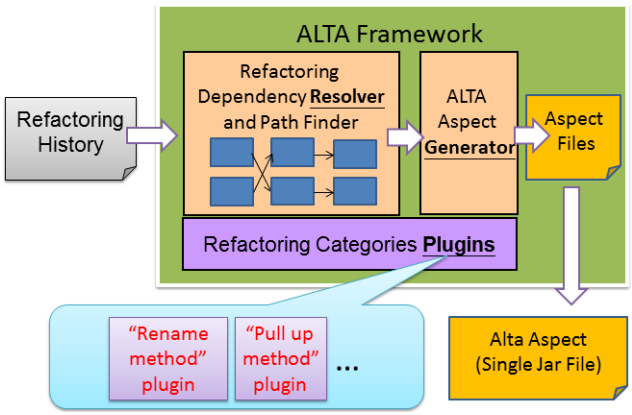

Figure 2.10 Architecture diagram of ALTA.

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 **identify-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)).

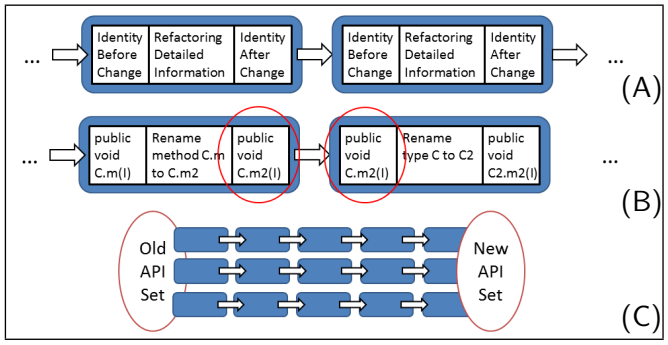

Figure 2.11 Refactoring paths which contains many linked refactoring nodes.

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

```
1  for each refactoring R in the refactoring history{
2    Get R's refactoring type T.
3    Use T to get R's corresponding plugin P.
4    Create one refactoring node N which represents R by P.
5    if(N can be appended into an existing path H){
6      Append N to the end of H.
7    }
8    else{
9      Create a new (empty) path H.
10     if(R is not about changing package){
11       Generate compensative ancient nodes.
12       Add these compensative ancient nodes to H.
13     }
14     Add N to H.
15   }
16   if(R is not about changing method){
17     Generate impacted nodes.
18     Append these impacted nodes into all related lists.
19   }
20 }
```

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 refactor actions in the following history file:

```
1 rename ClassA to ClassB;
2 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., `ClassB.methodX()`) does not exist in the old component. The reason for this nonexistence is that the refactoring in line 1 above already renamed `ClassA` to `ClassB`. Therefore, Resolver will create a special node called **compensative ancient node**, and its identity-before-change is `ClassA.methodX()` and identity-before-change is `ClassB.methodX()` (see line 11 and 12 in Figure 2.12). It is important to note that the identity-before-change of the created **compensative ancient node** exists in the old component. Finally, Resolver will add the **compensative ancient node** and the `nodeOfLine2` into the new path respectively.

Similarly, if we rename a method `ClassA.methodX()` to `ClassA.methodY()` first and then rename `ClassA` to `ClassB`, Resolver will generate an additional node given that `ClassA.methodY()` cannot be found in the new component. The identity-before-change of this new node is `ClassA.methodY()` and its identity-after-change is `ClassB.methodY()`. Later, Resolver will append it after the node which represents the `rename method` refactoring. We call this additional node an **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.

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.

2.4.4.1 If there is a missing method

Generator will use AspectJ's `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.

2.4.4.2 If there are conflict method signatures

Generator will use pointcuts with the **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[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

[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:

```
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 totoal 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.

Exp No.	**Refactoring Information**	**Test Result and Branch Coverage**
CRT 1	Rename type 'b.ClassA' to 'ClassA_ren'; Rename method 'b.ClassA_ren.methodB(...)' to 'methodB_REN'; Rename package 'b' to 'b_ren'	Tests run: 4299, Failures: 0, Errors: 0, Time elapsed: 3.747 sec, 100%
CRT 2	Rename method 'kslu.libA.LibClassA.methodD(...)' to 'methodD_REN'; Rename package 'kslu.libA' to 'kslu.libA_REN' ; Rename method 'kslu.libA_REN.LibClassA.methodD_REN(...)' to 'methodD_REN2'; Rename method 'kslu.libA_REN.LibClassB.methodC(...)' to 'methodC_REN'; Rename type 'kslu.libA_REN.LibClassB' to 'LibClassB_REN'; Rename method 'kslu.libA_REN.LibClassB_REN.methodB(...)' to 'methodB_REN'	Tests run: 280, Failures: 0, Errors: 0, Time elapsed: 0.929 sec, 100%
CRT 3	Change method 'public int A.ClassA.methodA(int a)' to 'private int methodA(int a)	Tests run: 562, Failures: 0, Errors: 0, Time elapsed: 0.706 sec, 100%
CRT 4	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)'	Tests run: 3883, Failures: 0, Errors: 0, Time elapsed: 2.224 sec , 100%
CRT 5	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';	Tests run: 1610, Failures: 0, Errors: 0, Time elapsed: 2.634 sec, 100%
CRT 6	Rename method 'util.Mathematics.divide(...)' to 'div'; Rename type 'util.Mathematics' to 'Math'; Change method 'public int util.Math.div(int i, int j)' to 'public int div(int i, int j, String msg)'; Change method 'public int util.Math.div(int i, int j, String msg)' to 'public int div(int j, int i, String msg)'	Tests run: 3442, Failures: 0, Errors: 0, Time elapsed: 4.146 sec , 100%
OSLT 1	Rename method 'org.apache.commons.lang3.text .CompositeFormat.getFormatter()' to 'getFormatter_REN'; Rename type 'org.apache.commons.lang3.builder.EqualsBuilder' to 'EqualsBuilder_REN'	Tests run: 2039, Failures: 0, Errors: 0, Time elapsed: 21.432 sec (no coverage data)
OSLT 2	Rename method 'org.apache.commons.lang3.text .CompositeFormat.reformat(...)' to 'reformat_REN' ; Rename type 'org.apache.commons.lang3.text .CompositeFormat' to 'CompositeFormat_REN'	Tests run: 2039, Failures: 0, Errors: 0, Time elapsed: 22.647 sec (no coverage data)

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 `sum+=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 `sum+=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 $\geq$ 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.

```
1  import java.lang.reflect.*;
2  privileged aspect SampleAspect {
3
4    //inter-type declaration: define a hidden field.
5    public packageA.ClassA_REN
6        packageA.ClassA.hiddenObj=null;
7
8    //inter-type declaration: define a method.
9    public static int packageA.ClassC.methodInClassC
10       (java.lang.String var1) {
11     return  packageA.ClassC.methodInClassC_REN(var1);
12   }
13
14   //pointcut and advice: handle all methods in ClassA.
15   Object around () : call(* packageA.ClassA.*(..)){
16     ... //skip
17   }
18
19   //pointcut and advice: handle conflict-APIs
20   int around(int var1) throws IOException:
21     call(int packageA.ClassA.go(int) throws IOException)
22     && args(var1) && !within(packageA.*) {
23       ... //skip
24   }
25
26   //pointcut and advice: handle all constructors of ClassA
27   packageA.ClassA around () :
28       call(packageA.ClassA.new(..)){
29     ... //skip
30   }
31
32   //pointcut and advice: handle the 'set' actions
33   //of all fields in ClassA
34   void around(Object input,  packageA.ClassA targ)
35       : set(* packageA.ClassA.*)  && args(input)
36       && target(targ)
37       && !set(* packageA.ClassA.hiddenObj){
38     ... //skip
39   }
40
41   //pointcut and advice: handle the 'get' actions
42   //of all fields in ClassA
43   Object around(packageA.ClassA targ)
44       : get(* packageA.ClassA.*)
45       && target(targ)
46       && !get(* packageA.ClassA.hiddenObj) {
47     ... //skip
48   }
49 }
```

Figure 2.13 Sample aspect generated by Generator.

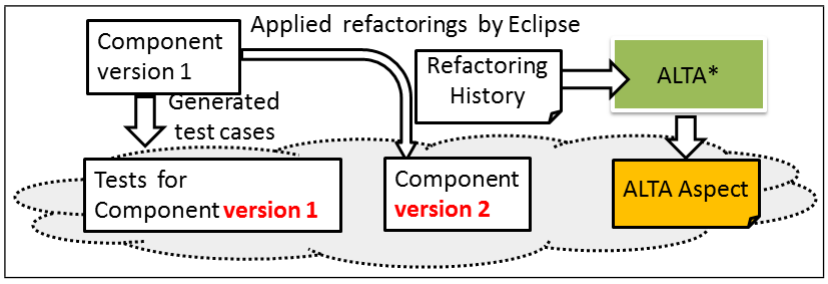

Figure 2.14 The process of compound refactoring tests.

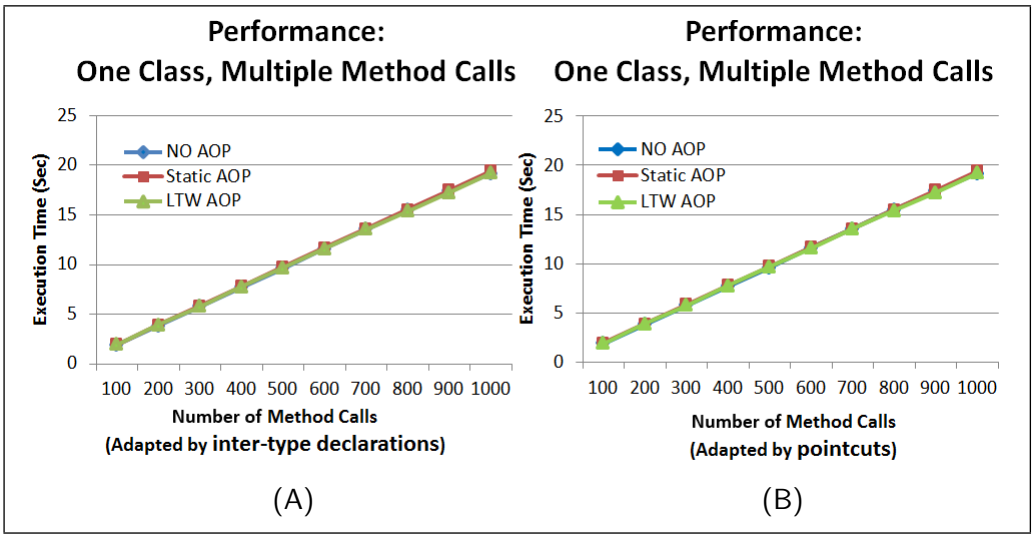

Figure 2.15 Performance report. X-axis: number of method adapted in one class; y-axis: performance.

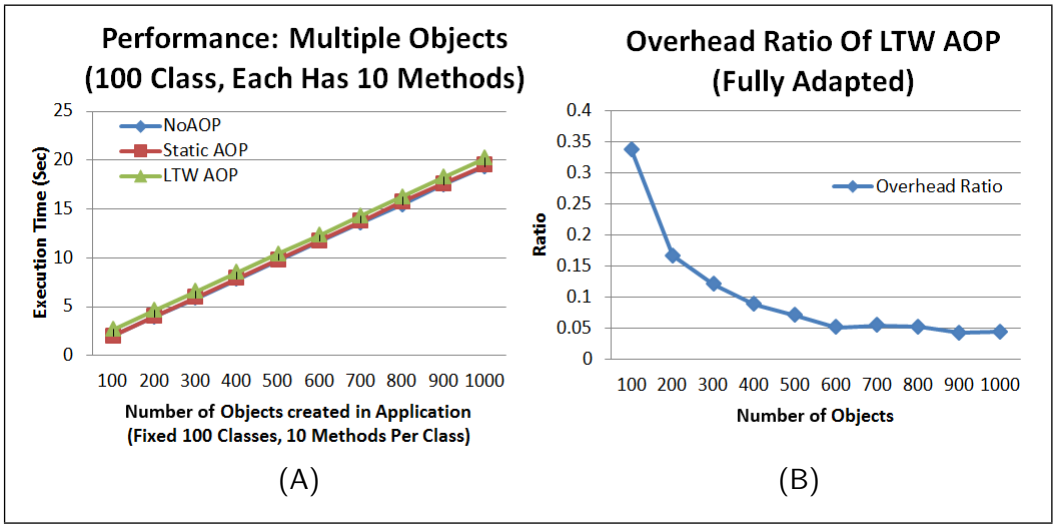

Figure 2.16 Performance report. X-axis: number of object created; y-axis: performance. 100% of the methods in each class were adapted.

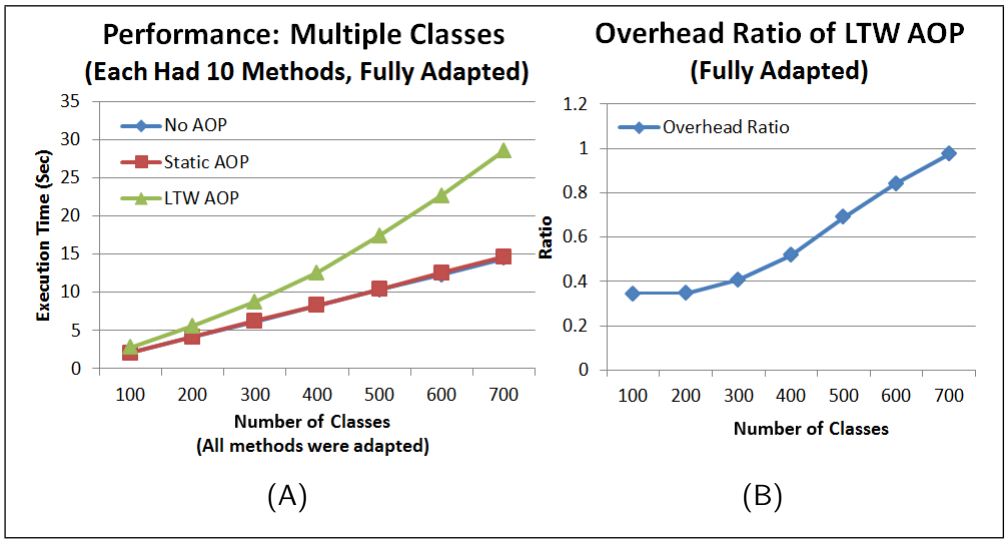

Figure 2.17 Performance report. X-axis: the number of class adapted; y-axis: performance. 100% methods in each class were adapted.

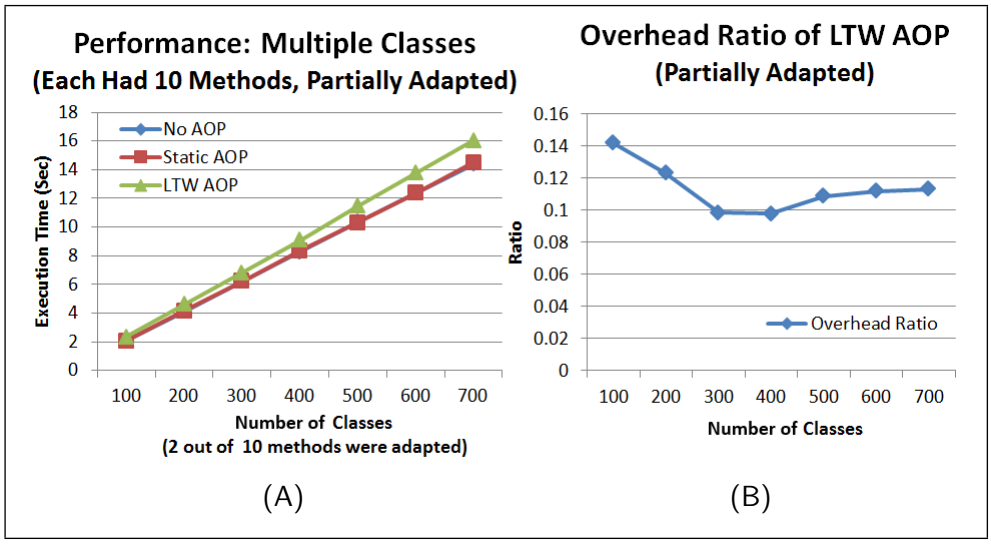

Figure 2.18 Performance report. X-axis: adapted class count; y-axis: performance. 20% of the methods in each class were adapted.

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 exists either in the old or new API.**

   Figure 3.1 illustrates this problem. Suppose when upgrading a component, you move a

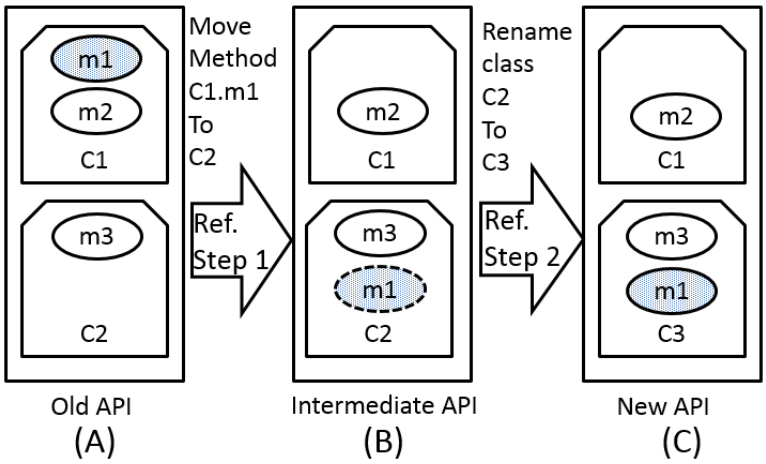

Figure 3.1 An example of Temporal-Dependent Refactoring Steps (TDRS): (A) shows the old API of this component, (B) shows the intermediate API, and (C) shows the new API. Method *m1* in class *C1* (denoted as *C1.m1*) was moved to class *C2* and then C2 was renamed to *C3*. These two refactoring steps, "move method *C1.m1* to *C2*" and "rename class *C2* to *C3*", are TDRS.

method *m1* from class *C1* to class *C2*, then rename class *C2* to *C3* (see Figure 3.1). Since *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. **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 X in the middle of Figure 3.2 (A) and (B)), all existing solutions cannot work.

3. **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.

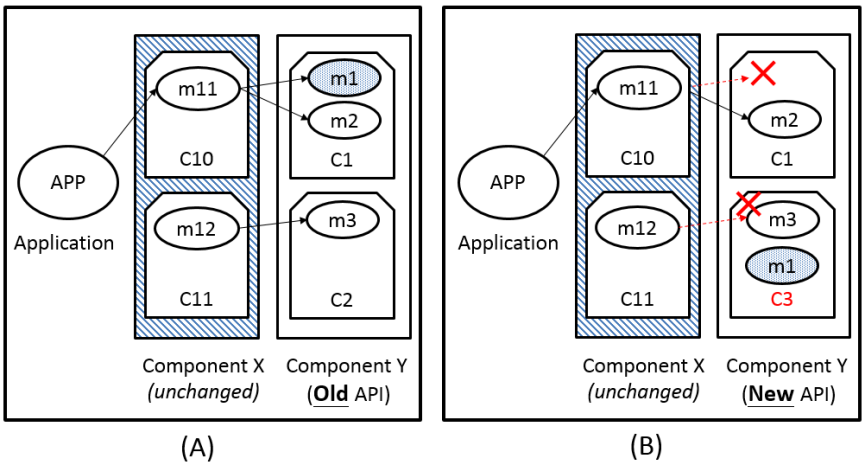

Figure 3.2 Example of a third-party API-caller. (A) Before upgrading component Y. (B) After upgrading component Y. After upgrading Y, X and Y became incompatible.

3.1.1 The Proposed Framework

In this study, we introduce TARP (**T**esting and **AI**-Planning Based **R**efactoring **P**ath 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 planer 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

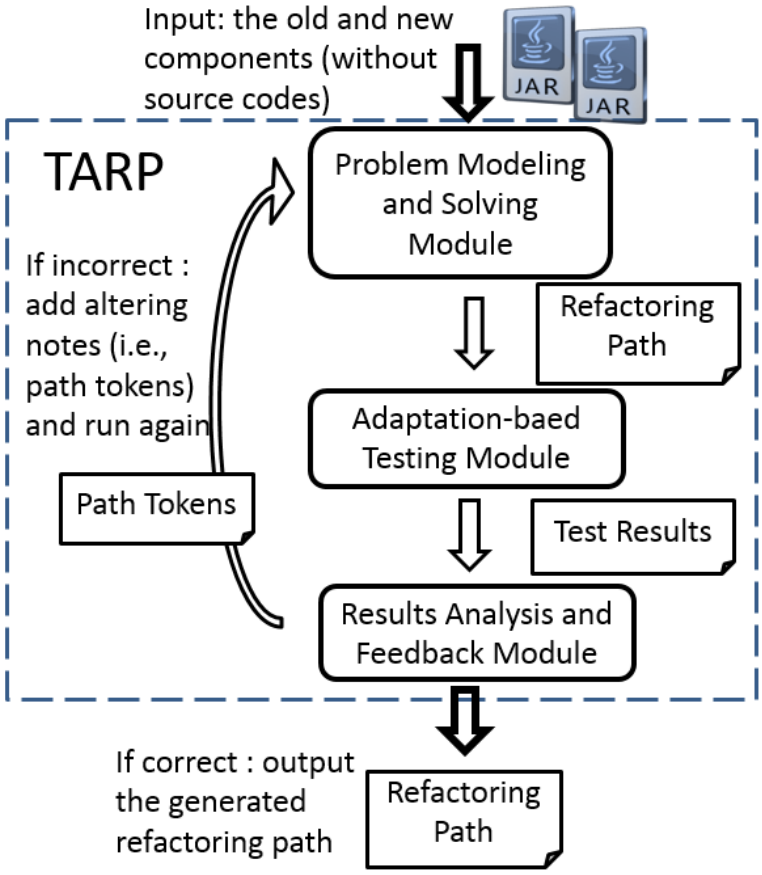

Figure 3.3 Modules and conceptual data flows of TARP. There are three modules: Problem Modeling and Solving Module (PMSM), Adaptation-based Testing Module (ATM), and Result Analysis and Feedback Module (RAFM).

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

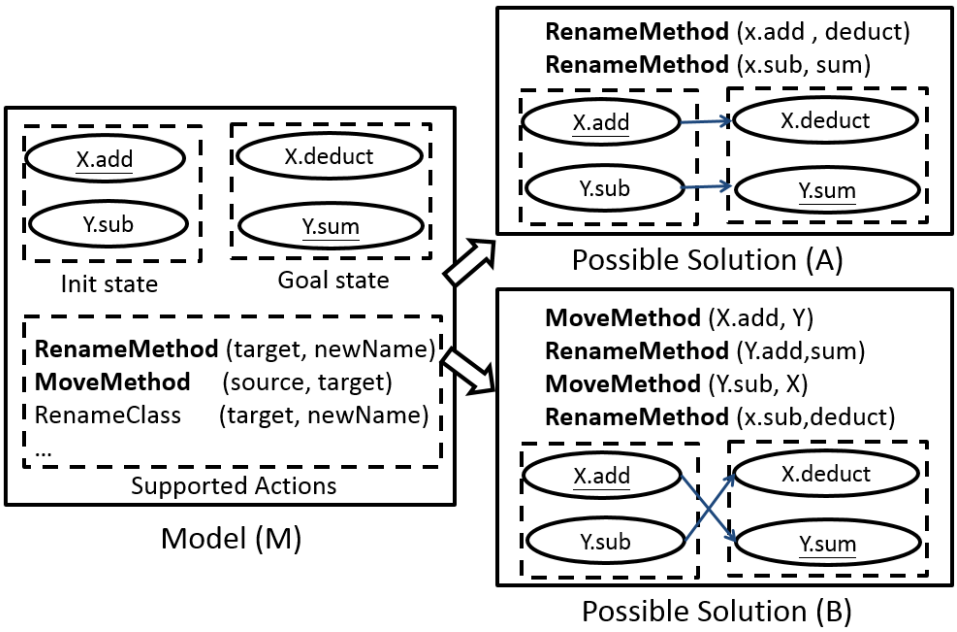

Figure 3.4 A model with multiple solutions: (M) is the model, and solution (A) and (B) are two possible solutions. While changing method *X.add* to *Y.sum* is more reasonable, they are both seen as "AI-Planning correct" solutions given that (A) and (B) have exactly the same initial state and goal state.

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,

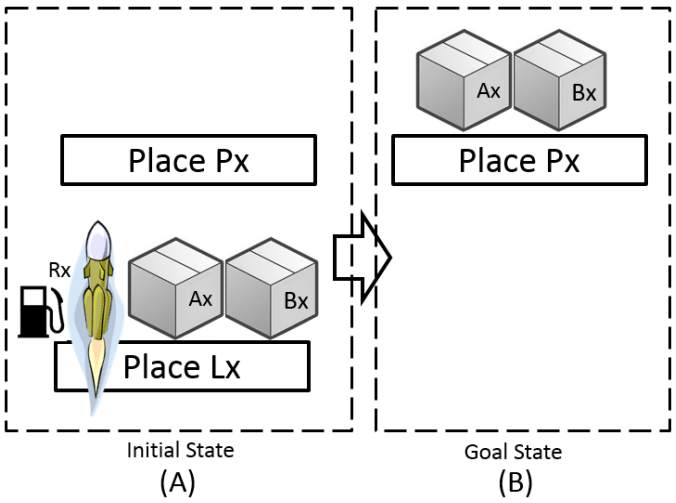

Figure 3.5 Example of an AI planning problem

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
5  (:predicates
6    (At ?o - Object ?p - Place)
7    (Hasfuel ?r - Rocket)
8    (In ?c - Cargo ?r - Rocket)
9  )
10
11 (:objects
12   Ax Bx - Cargo
```

```
13    Rx - Rocket
14    Lx Px - Place
15 )
```

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:

```
1 (:init
2    (At Ax Lx)
3    (At Bx Lx)
4    (At Rx Lx)
5    (HasFuel Rx)
6 )
```

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:

```
1 (:goal
2    (and
3       (At Ax Px)
4       (At Bx Px)
5    )
6 )
```

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:

```
1 (:action load
2    :parameters
```

```
3     (?r - Rocket ?p - Place ?c - Cargo)
4   :precondition (and
5     (At ?r ?p)
6     (At ?c ?p)
7   )
8   :effect (and
9     (In ?c ?r)
10    (not (At ?c ?p))
11  )
12 )
```

```
1  (:action unload
2    :parameters
3      (?r - Rocket ?p - Place ?c - Cargo)
4    :precondition (and
5      (At ?r ?p)
6      (In ?c ?r)
7    )
8    :effect (and
9      (At ?c ?p)
10     (not (In ?c ?r))
11   )
12 )
```

```
1  (:action move
2    :parameters
3      (?r - Rocket
4       ?from - Place ?to - Place)
5    :precondition (and
6      (not (=  ?from ?to))
7      (At ?r ?from)
8      (Hasfuel ?r)
9    )
10   :effect (and
11     (At ?r ?to)
12     (not (At ?r ?from))
```

```
13      (not (Hasfuel ?r))
14    )
15  )
```

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}.

```
1 (load Rx Lx Ax)
2 (load Rx Lx Bx)
3 (move Rx Lx Px)
4 (unload Rx Px Ax)
5 (unload Rx Px Bx)
```

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  —> C3.m1
2 C2.m3  —> 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.

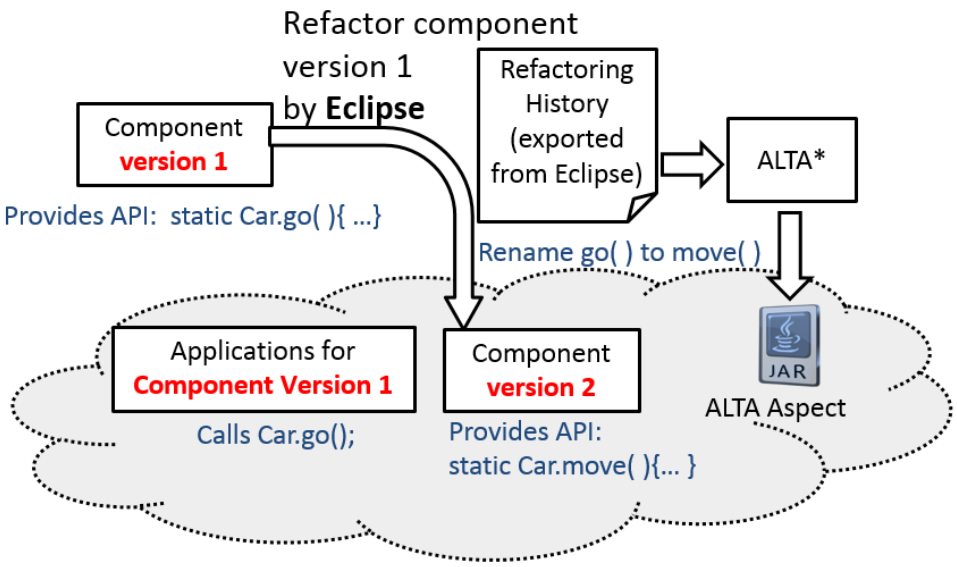

Figure 3.6 The concept of ALTA's runtime adaptation. ALTA will redirect all Car.go() calls to Car.move(), and redirect the return value back to the caller on the fly.

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:

```
1  (:types
2     APIObject - object
3     Class - APIObject
4     Method - APIObject
5  )
6
7  (:predicates
8     (Contains
9        ?c - Class
10       ?m - Method
11    )
12 )
13
14 (:action MoveMethod
15    :parameters (?m1 - Method
16          ?cFrom ?cTo - Class )
17      :precondition (and
18         (Contains ?cFrom ?m1)
19       )
20      :effect (and
21         (Contains ?cTo ?m1)
22   (not (Contains ?cFrom ?m1))
23      )
24 )
```

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).

```
1 (:objects
2   C1 C2 - Class
3   m1 m2 m3 - Method
4 )
5
6 (:init
7   (Contains C1  m1 )
8   (Contains C1  m2 )
9   (Contains C2  m3 )
```

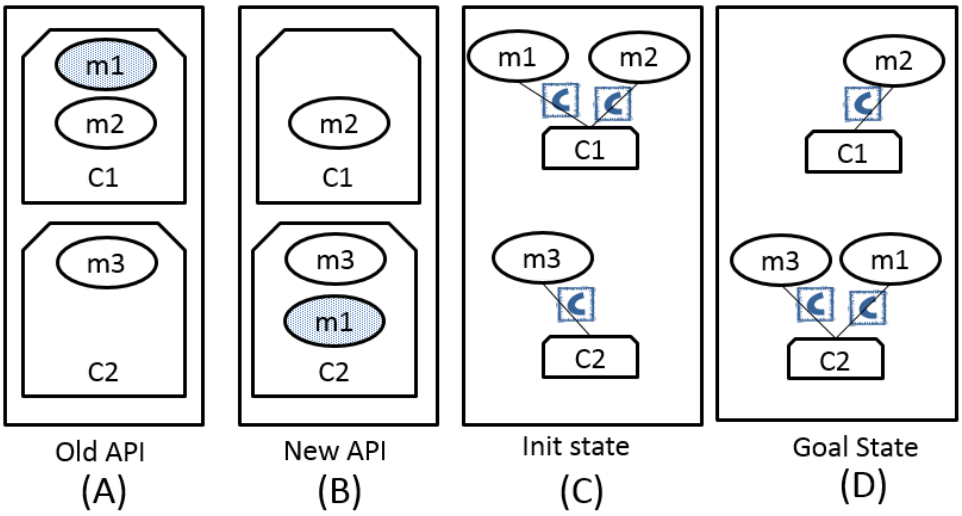

Figure 3.7 An example of the preliminary modeling strategy. (A) and (B) show a simplified example from Figure 3.1. Note that Figure 3.1 (C) was removed from this example so Figure 3.1 (B) became the new API. (C) and (D) are the model of this example created by the preliminary modeling strategy, where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

```
10 )
11
12 (:goal
13   (and
14     (Contains C1 m2 )
15     (Contains C2 m3 )
16     (Contains C2 m1 )
17   )
18 )
```

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:

```
1 { (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.

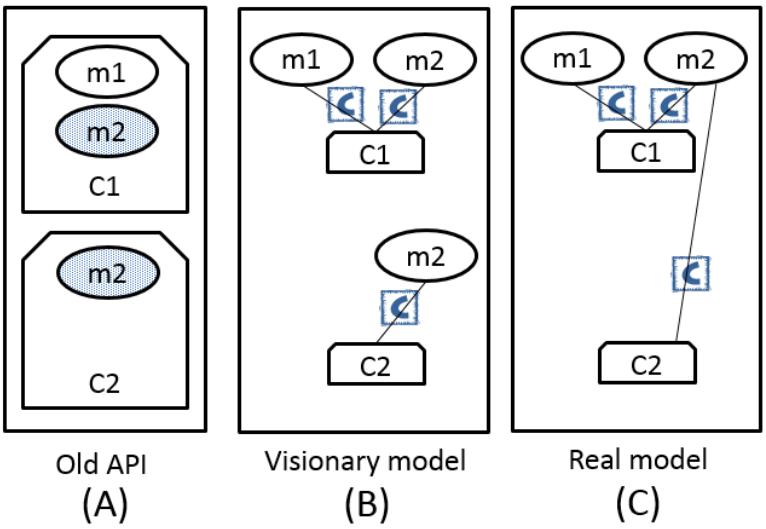

Figure 3.8 Example of the same-name problem. (A) shows that each class contains a method named *m2*. (B) is what we want to define by our preliminary modeling strategy but (C) is what we actually defined.

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 *C1* and *C2* have a method *m2*. 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 *m2* methods in line 3 will only create ONE object in AI planner.

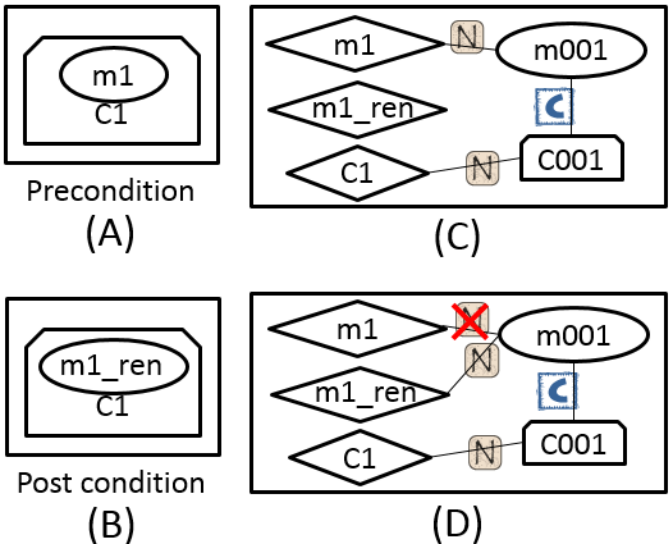

Figure 3.9 Example of modeling a rename action. (A) is the precondition and (B) is the post condition. Because all object's identity cannot be changed in an AI planning model, we model rename actions by using "HasName" predicates. In (C), the method *m001* is related to a name object called *m1*, and 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 predefine a dummy

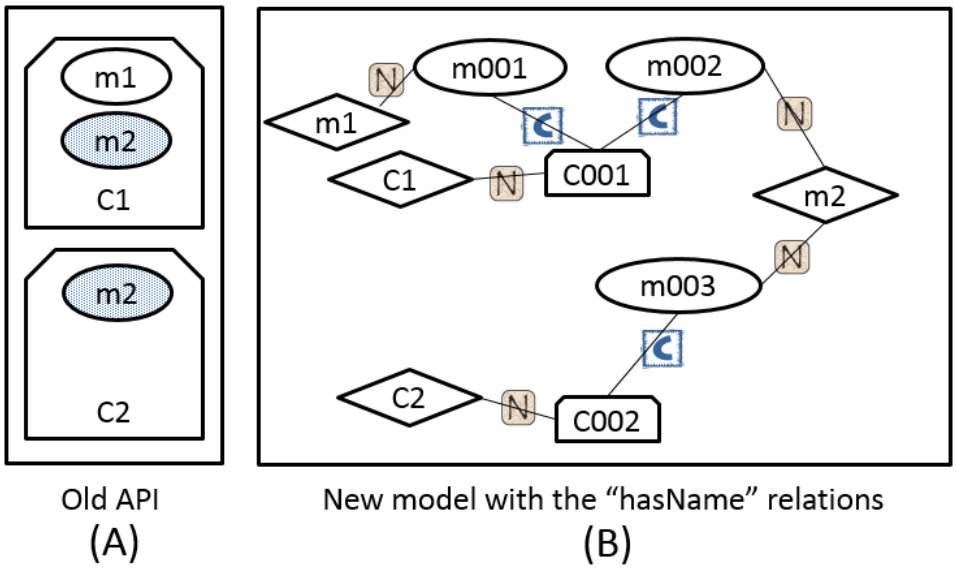

Figure 3.10 The solution of the same-name problem. (A) shows the API and (B) shows the model. Each object has a unique serial number, and each icon "N" shows a "HasName" relation, and diamond-shaped objects are name-typed objects. Note that *m0002* and *m0003* share *m2* because they have the same method 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*).

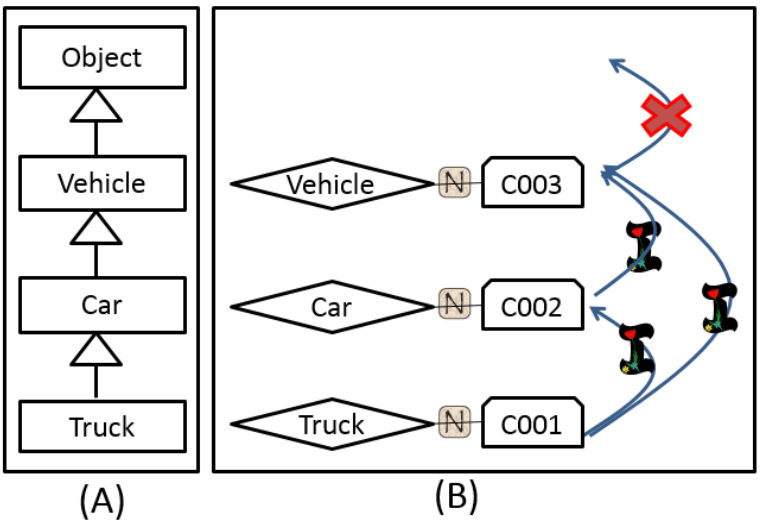

Figure 3.11 An example of modeling inheritance relations. (A) shows the API and (B) shows the model. The icon "I" represents the "Inherit" predicate.

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

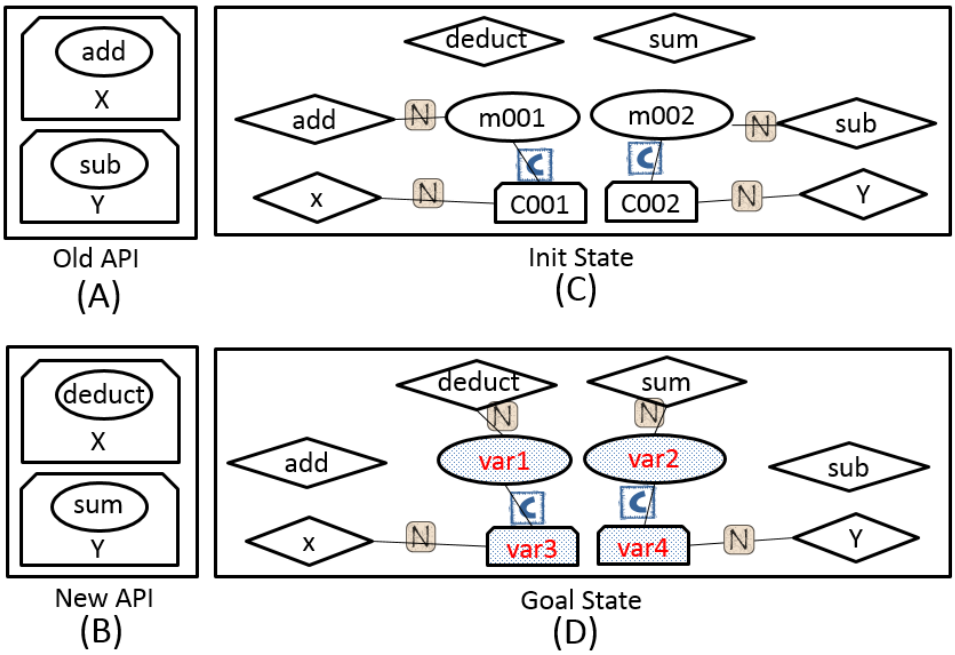

Figure 3.12 An example of the uncertain identities problem. (A) is the old API, (B) is the new API, (C) is the modeled initial state from (A), and (D) is the modeled goal state from (B).

the result. In Figure 3.12 (C), the API object *C001* has a name *X* and it contains another API object *m001* whose name is *add*, which tells us that "there is a class named *X* who contains a method called *add*". Similarly, this model also says that there is another class called *Y* who contains a method named *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 *deduct*, *sum*, *X* and *Y*. Moreover, we know that the API object related to name object *X* (denoted as a variable *var3*) will contain another API object which is related to the name object *deduct* (*var1*). Similarly, we know that *var3* has a name *Y*, *var3* contains *var2*, and *var3* has a name *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 **when we reuse an ID, we are actually binding all objects**

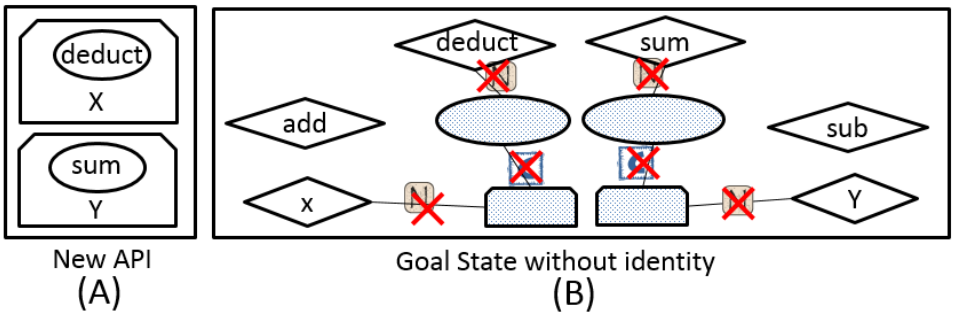

Figure 3.13 Goal state without API-object identities: all the relations cannot be built.

who share this ID. In other words, we make the planner biased. For example, if we assign a value *m001* to the variable *var1* in Figure 3.12, we are telling the planner "the method named *add* in the initial state is actually the method named *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 *m777* to *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 **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 "**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

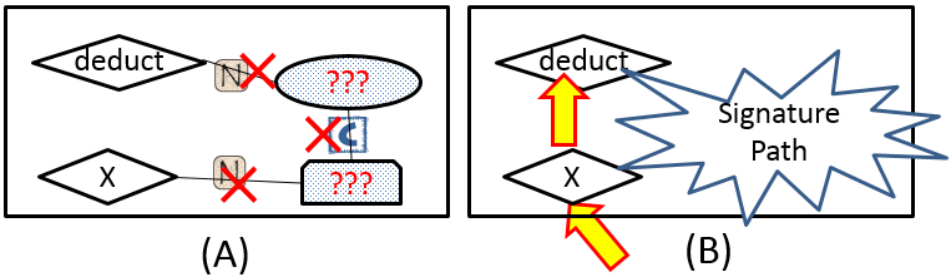

Figure 3.14 The concept of a signature path. (A) shows a goal state without any identity of API-objects; therefore we cannot create "HasName" or "Contains" predicates. (B) shows a signature path that describes a path which includes a class name and a method name.

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 "SignaturePathTill**XYZ**". "XYZ" represents the end point of this path. For example, the predicate "(SignaturePathTill-Method *?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

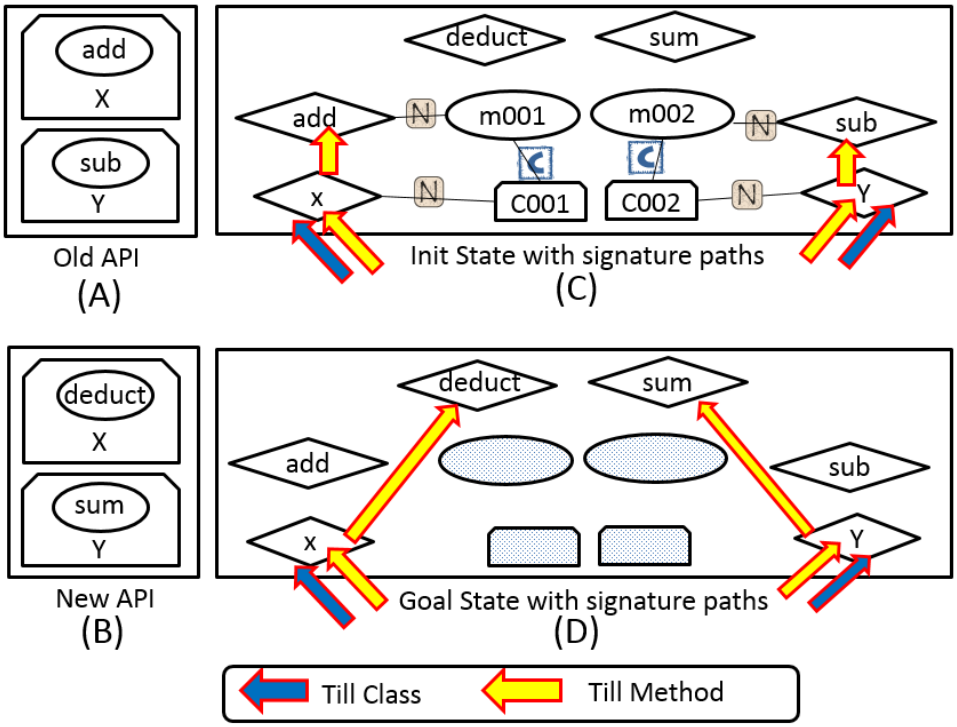

Figure 3.15 Goal state without API-object identities but with signature paths: all the important concepts have been correctly captured.

follows:

```
(: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)
```

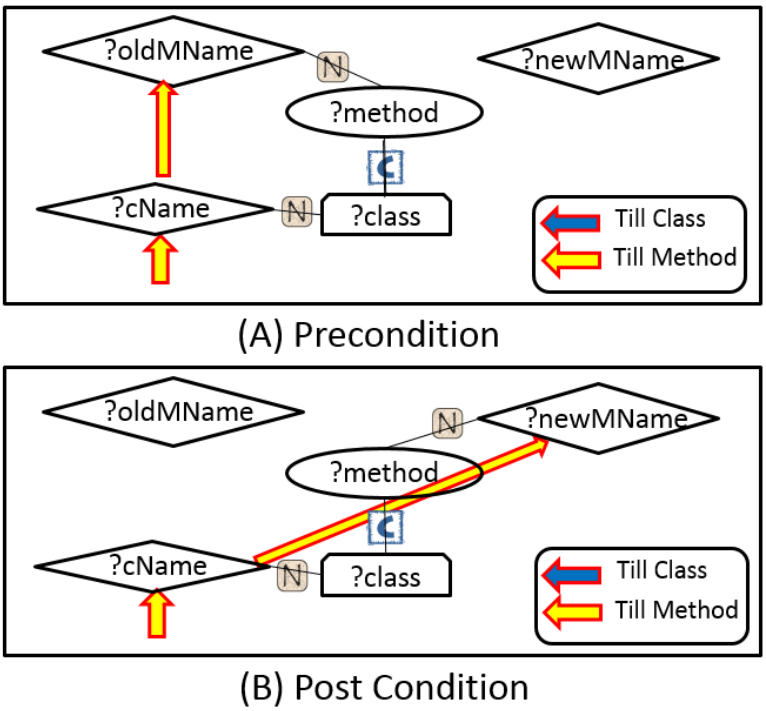

Figure 3.16 A sample "rename method" actions that supports name path. (A) is the precondtion and (B) is the post condition.

```
16      (HasName ?method ?oldMName)
17      (not (HasName ?method ?newMName))
18
19      ;; signature paths
20      (SignaturePathTillMethod
21          ?cName ?oldMName)
22      (not (SignaturePathTillMethod
23          ?cName ?newMName))
24    )
25
26    :effect (and
27      ;; object structures
28
29      ;; name relations
30      (not (HasName ?method ?oldMName))
31      (HasName ?method ?newMName)
32
```

```
33      ;; change signature paths
34      (not (SignaturePathTillMethod
35          ?cName ?oldMName))
36      (SignaturePathTillMethod
37          ?cName ?newMName)
38    )
39  )
```

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 therefor 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

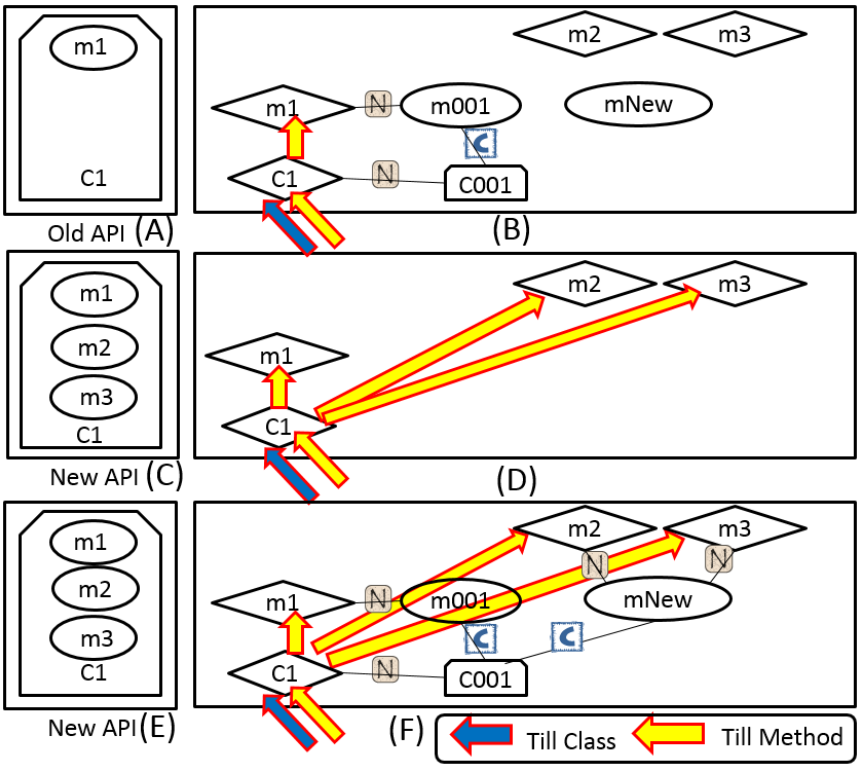

Figure 3.17 An example of supporting new methods.

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.

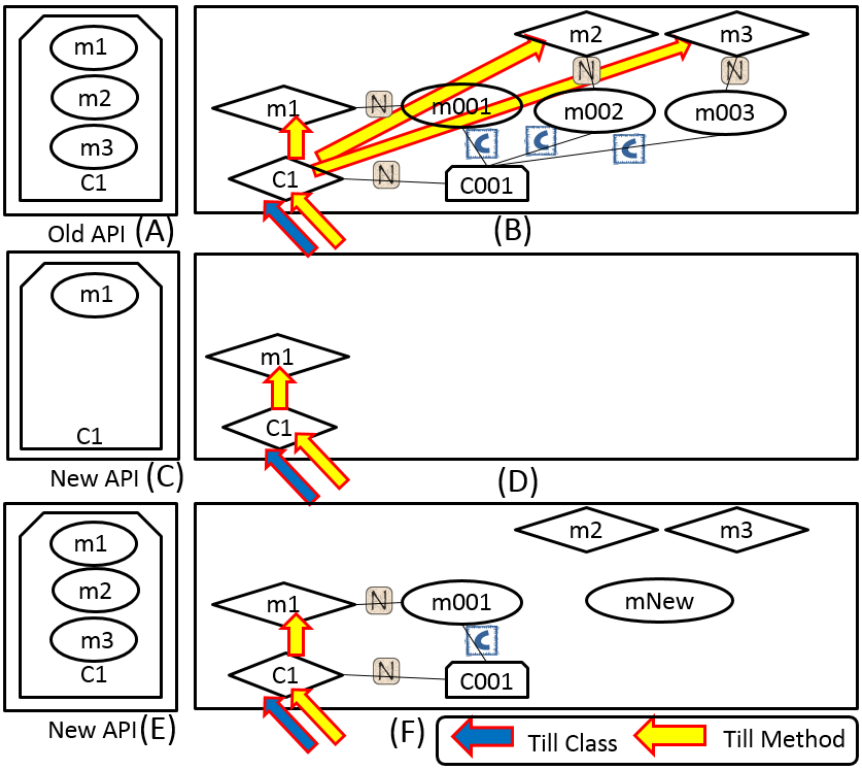

Figure 3.18 An example of supporting method deletions.

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(*?class*, *?className*, *?method*, *?methodName*)". In this action, the predicate (Contains ?class, ?method), (HasName ?method ?methodName) and (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.

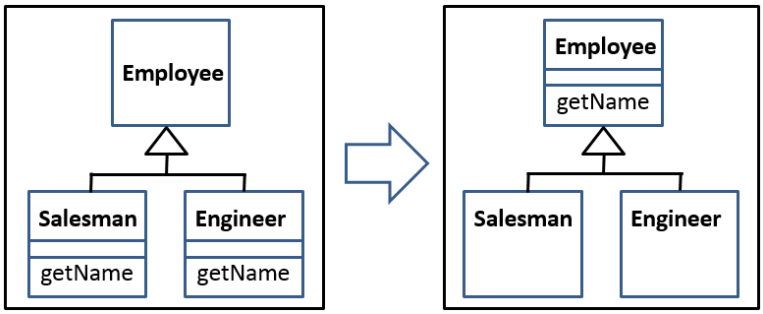

Figure 3.19 The concept of Pull-up methods.

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):

```
1 action pullupMethodFrom1Child:
2   (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj
3    ?cParentName ?cParentObj)
```

```
4
5  action pullupMethodFrom2Children:
6    (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj
7                ?c2m2Obj ?c2Name ?c2Obj
8     ?cParentName ?cParentObj)
9
10 action pullupMethodFrom3Children:
11   (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj
12               ?c2m2Obj ?c2Name ?c2Obj
13               ?c3m3Obj ?c3Name ?c3Obj
14    ?cParentName ?cParentObj)
15
16 action pullupMethodFrom4Children:
17   (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj
18               ?c2m2Obj ?c2Name ?c2Obj
19               ?c3m3Obj ?c3Name ?c3Obj
20               ?c4m4Obj ?c4Name ?c4Obj
21    ?cParentName ?cParentObj)
```

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:

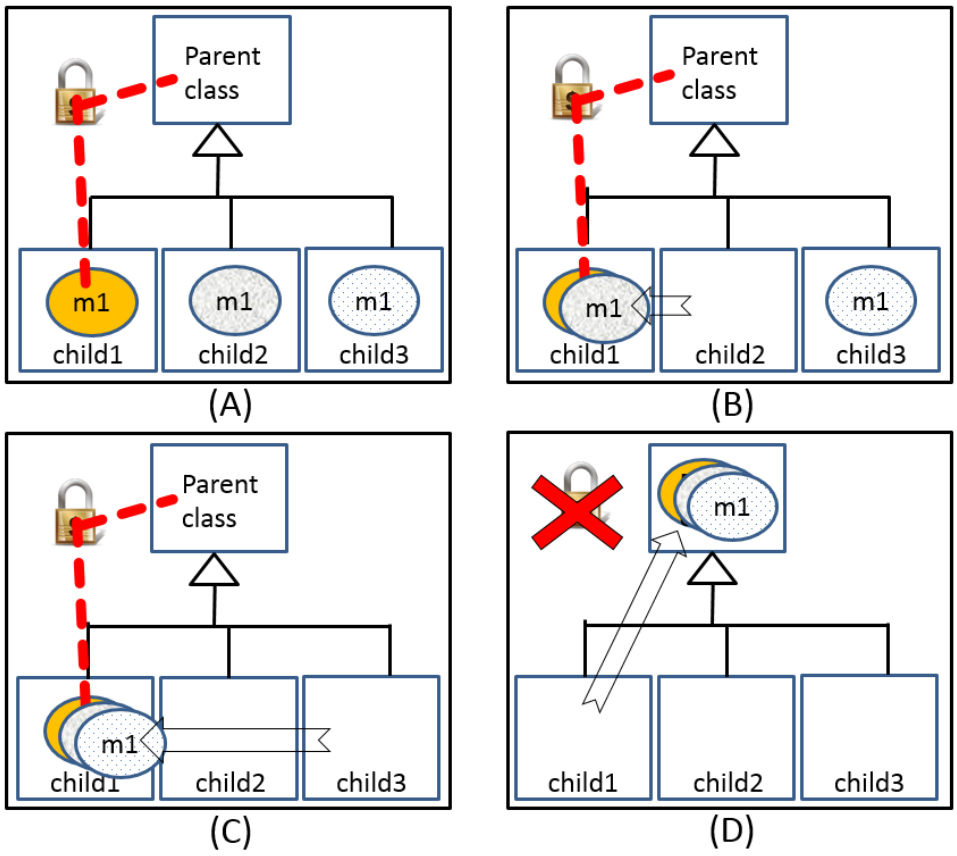

Figure 3.20 Model the pull-up method pattern by refactoring transaction and parameter reducing.

```
1  (pullupMethod_start
2      getName m001 Salesman c001
3                   Employee c003)
4  (pullUpMethods_mergingSiblings
5      getName m001 Salesman c001
6              m002 Engineer c002
7                   Employee c003)
8  (pullupMethod_end
9      getName m001 Salesman c001
10                  Employee c003)
```

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 pullupMethod_start, **2** pullUpMethods_mergingSiblings, and 1 pullupMethod_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:

```
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      Else{
7        For each class C in P{
8          If (exists Class C' in P' and C.name == C'.name ){
9            If(C.content == C'.content){
10             delete C and C';
11           }
12           Else{
13             For each method M in C{
14               If (exists method M' in C' and M.signature == M'.signature ){
15                 delete M and M';
16               }
17             }
18             For each field F in C{
19               If (exists method F' in C' and F.signature == F'.signature ){
```

```
20                  delete F and F';
21 } } } } } } } }
```

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-

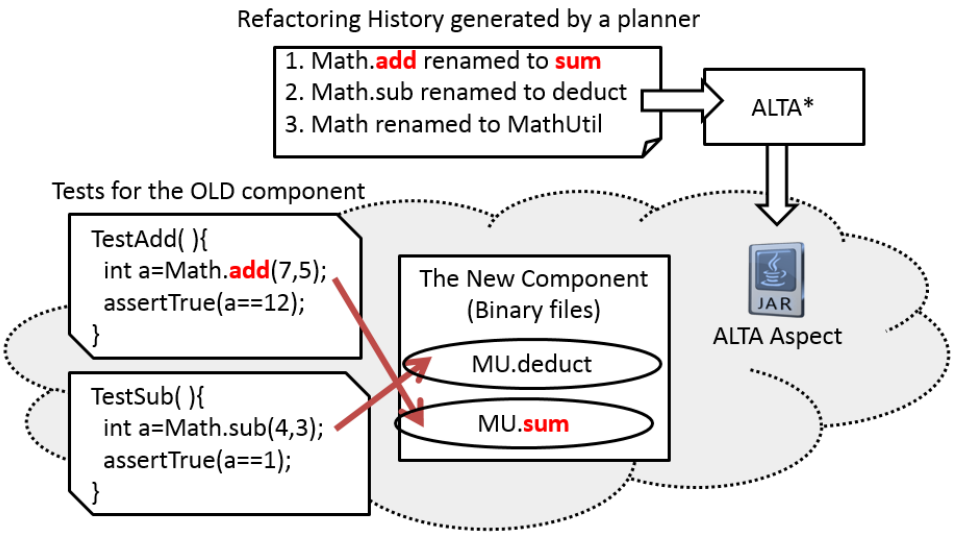

Figure 3.21 Example of an adaptation-base testing with a correct input refactoring path. The path says the "add" method was renamed to the "sum" method, and the "sub" method was renamed to the "deduct" method; therefore there will be no errors or fails in the test report.

toring path is correct, so ALTA* will do a correct adaptation to bridge the method "add" to the method "sum". When a test calls add(7,2), it will expect to get 9 as the result, and because ALTA* forward the call to "add", the caller will get 9 as the result; therefore this test will pass. Similarly, the tests for deduct will pass, too. On the other hand, Figure 3.22 shows a counter example. With the incorrect input path, ALTA* will do an incorrect adaptation to bridge the method "add" to the method "deduct". When a test calls add(7,2), it will expect to get 9 as the result, but because ALTA* forward the call to "deduct", the caller will get 5 as the result; therefore this test will fail.

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

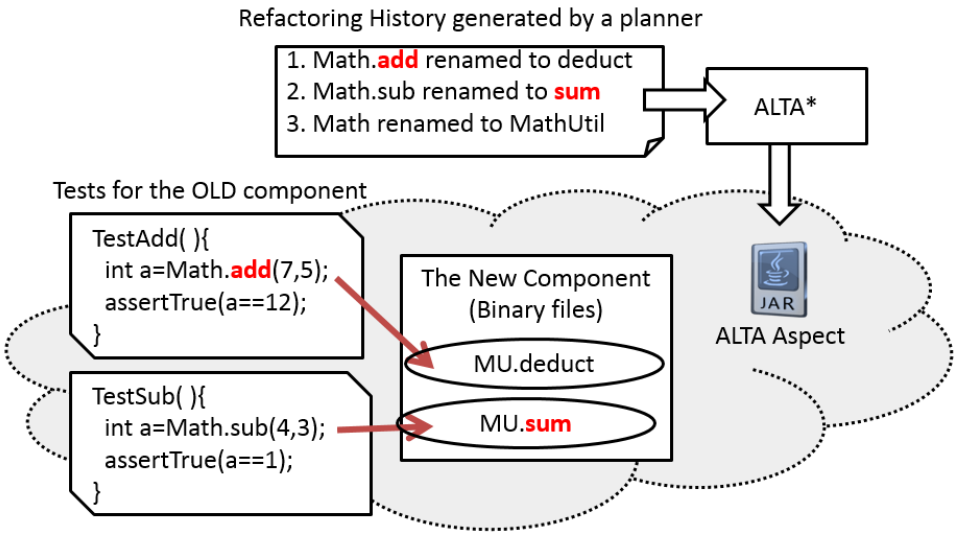

Figure 3.22 Example of an adaptation-base testing. The input refactoring is incorrect (it says the "add" method was renamed to the "deduct" method"); therefore there must be some errors or failures in the test report.

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 ($Src \rightarrow Dest$) 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 *Src*'s hand. Then we claim that *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 *Src* to anywhere but *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

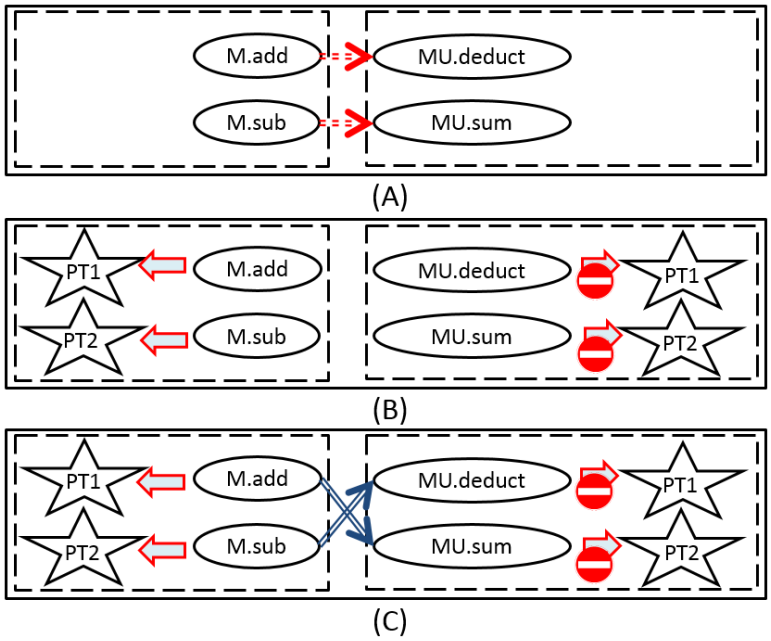

Figure 3.23 Example of retrieving a better solution by adding path tokens. Dashed (red) arrows are incorrect mappings and others (blue) are correct mappings.

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 "SignaturePathTillPath-Token" 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

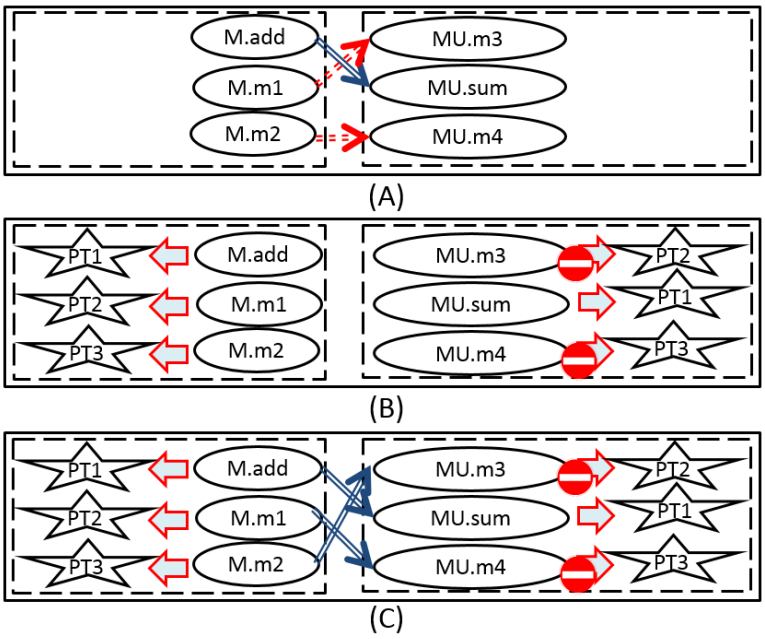

Figure 3.24 Example of using a positive path token assertion with 2 negative path token assertions. Dashed (red) arrows are incorrect mappings and others (blue) are correct mappings.

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), *M.add* maps to *MU.sum* because this mapping is required in the goal state. Regarding *M.m1*, because the planner cannot map it to *MU.m3* (since there is already a NPTA for this mapping) or MU.sum (since *M.add* needs to be mapped to *MU.sum*), it will be mapped to a new target *M.m1.* Similarly, a planner will map *M.m1* to a new target *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: NxN

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

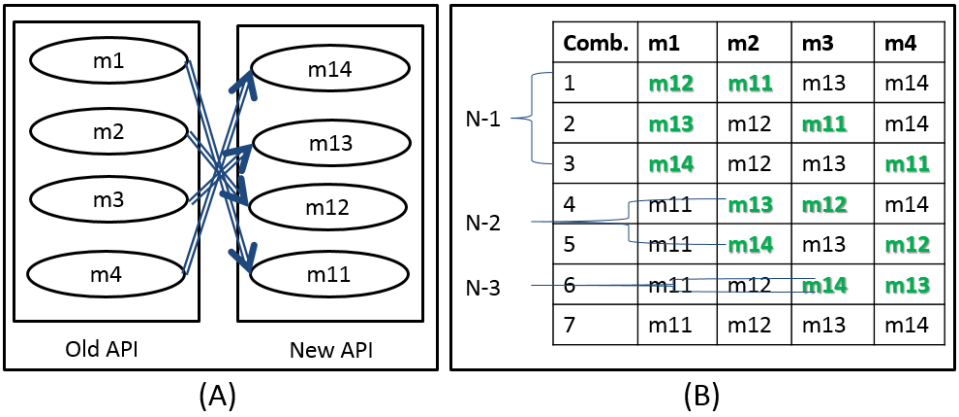

Figure 3.25 A 4x4 mapping problem. Arrows in (A) indicate the correct mappings. By using negative path token assertions, there are at most $N-1=3$ combinations which contains a wrong target of *m1*, and $N-2=2$ combinations for *m2*, $N-3=1$ combination for *m3*, and 0 combination for *m4*. So the total number of combinations in the worst case is $3+2+1+1=7$. Bold-typed fonts such as *m12* and *m11* in the first data row of (B) indicate the swapping of correct answers.

same-parameter-types targets in the new API, then there will be ${}_NP_N = N!$ combinations.

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 (*m1* $\rightarrow$ *m11*), (*m2* $\rightarrow$ *m12*), (*m3* $\rightarrow$ *m13*) and (*m4* $\rightarrow$ *m14*). For *m1*, a planner can map it to a wrong target for $N-1=3$ times (see (B)'s combination 1

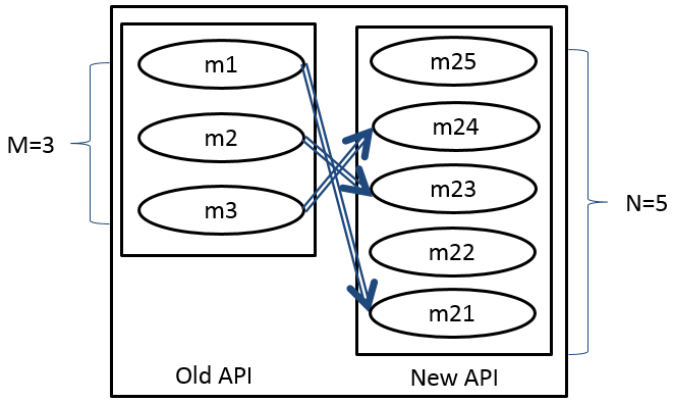

Figure 3.26 A MxN mapping problem where M=3 and N=5. Arrows in this graph indicate the correct mappings.

to 3). However, for *m2*, it only has $N-2=2$ incorrect targets because after the combination 1 TARP already eliminated the ($m2 \rightarrow m11$) mapping. Similarly, for *m3*, it only has $N-3=1$ wrong target, and *m4* 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 *m3* needs to be mapped to *m13*. Similarly, the planner will also skip #3 and #5 because *m4* needs to be mapped to *m14*. 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

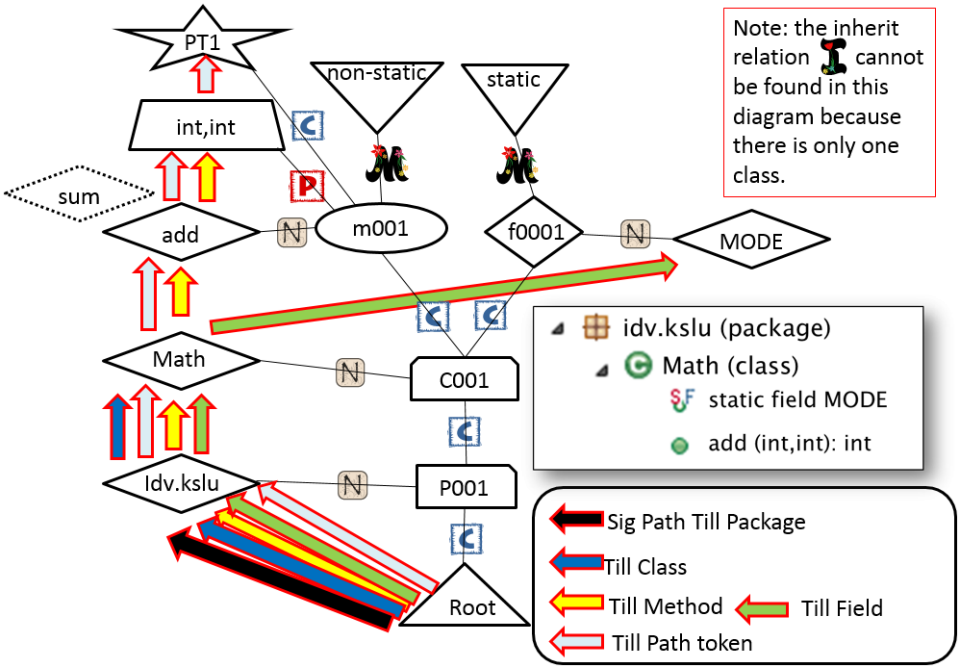

Figure 3.27 A sample model: the initial state

(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 $O(M_j N_j)$ (when using PPTA) or $O(N_j)$ (when using PPTA).

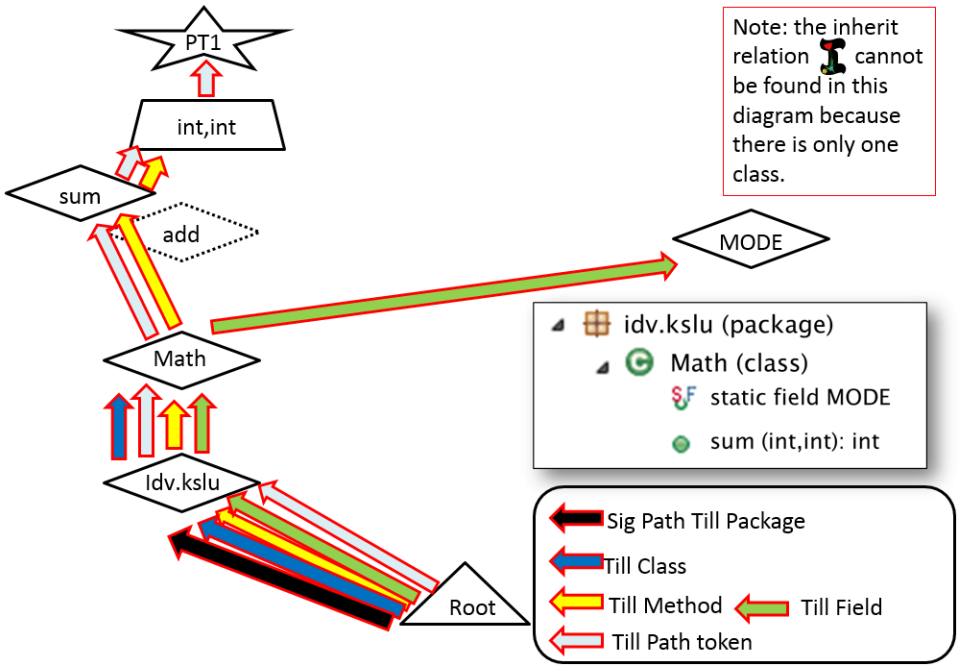

Figure 3.28 A sample model: the goal state

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

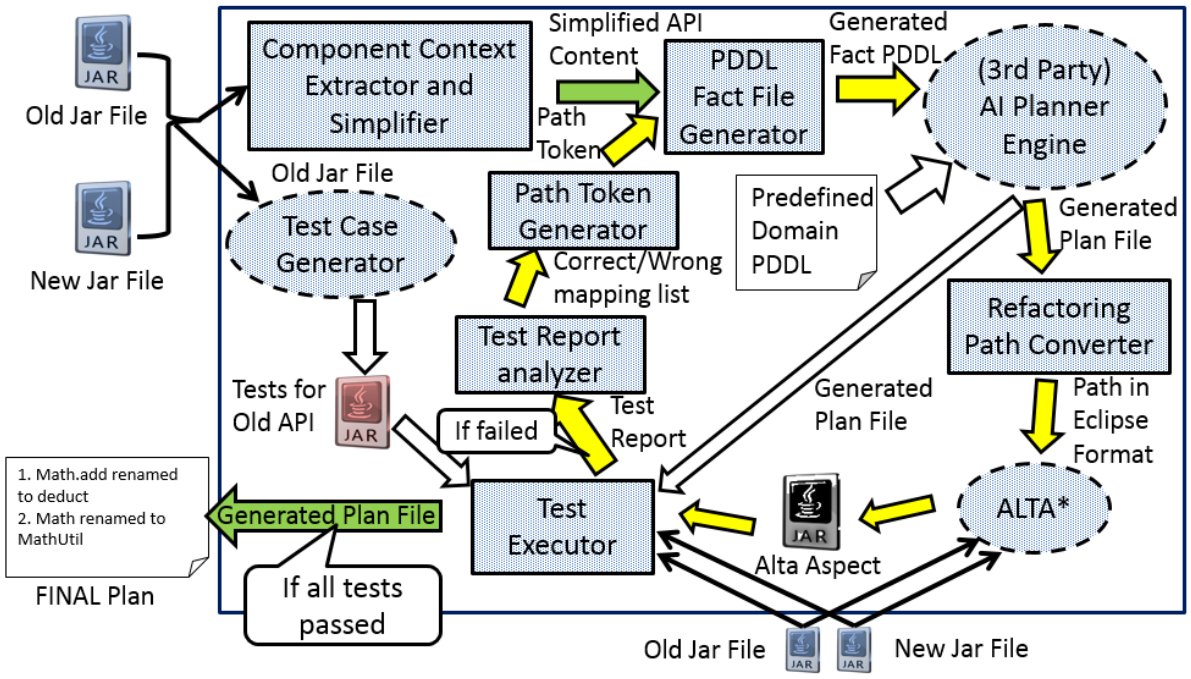

Figure 3.29 System Architecture of TARP*. There are six internal modules and three third–party tools (i.e., the dashed bubbles).

(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 → 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”. Similary, 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
  MethodModifer - object
  FieldModifier  - object

  ;; path tokens
  PackagePathToken - Object
  ClassPathToken - Object
  MethodPathToken - Object
  FieldPathToken - Object
```

```
29
30 )
31
32 (:predicates
33
34   ;;tree structures
35   (ContainsPackage ?r - APIRoot ?p - Package)
36   (ContainsClass ?p - Package  ?c - Class)
37   (ContainsMethod ?c - Class ?m -  Method)
38   (ContainsField ?c - Class ?f -  Field)
39
40   ;;path token
41   (ContainsPackagePathToken ?p - Package ?t - PackagePathToken)
42   (ContainsClassPathToken ?c - Class ?t - ClassPathToken)
43   (ContainsMethodPathToken ?m - Method ?t - MethodPathToken)
44   (ContainsFieldPathToken ?f - Field   ?t - FieldPathToken)
45
46   (HasPackageName ?p - Package ?pName - PackageName)
47   (HasClassName ?c - Class ?cName - ClassName)
48   (HasMethodName ?m - Method ?mName - MethodName)
49   (HasFieldName ?f - Field   ?fName - FieldName)
50
51   ;;types
52   (HasMethodParamTypes ?m - Method ?types - MethodParamTypes)
53
54   ;; modifiers
55   (HasMethodModifier ?m - Method ?mod - MethodModifier)
56   (HasFieldModifier ?f - Field ?mod - FieldModifier)
57
58   ;; signature paths
59   ;; package
60   (SignaturePathTillPackage ?r - APIRoot ?pName - PackageName)
61   (SignaturePathTillPackagePathToken ?r - APIRoot ?pName - PackageName ?
         pToken - PackagePathToken)
62
```

```
63    ;; class
64    (SignaturePathTillClass ?r - APIRoot ?pName - PackageName ?cName -
          ClassName)
65    (SignaturePathTillClassPathToken ?r - APIRoot ?pName - PackageName ?cName
          - ClassName  ?cToken - ClassPathToken)
66
67    ;; method -- remember to include types
68    (SignaturePathTillMethod ?r - APIRoot ?pName - PackageName ?cName -
          ClassName ?mName - MethodName ?types - MethodParamTypes)
69    (SignaturePathTillMethodPathToken ?r - APIRoot ?pName - PackageName ?cName
           - ClassName ?mName - MethodName ?mParamTypes - MethodParamTypes ?
          mToken - MethodPathToken)
70
71    ;; field
72    (SignaturePathTillField ?r - APIRoot ?pName - PackageName ?cName -
          ClassName ?fName - FieldName)
73    (SignaturePathTillFieldPathToken ?r - APIRoot ?pName - PackageName ?cName
          - ClassName ?fName - FieldName ?fToken - FieldPathToken)
74
75    ;; inherit
76    (Inherit ?classChild - Class ?classParent - Class )
77
78    ;; pull-up transactions
79    (notPullingUpMethods)
80
81    ;; note: all objects, not names
82    (MethodDuringPullingUp ?m1 - Method ?m1ParamTypes - MethodParamTypes ?
          cFrom - Class ?cTo - Class)
83 )
```

```
1  ;; rename method
2  (:action renameMethod
3
4    :parameters (
5      ?root - APIRoot
6      ?p1 - Package
```

```
7       ?p1Name - PackageName
8       ?c1 - Class
9       ?c1Name - ClassName
10      ?m1 - Method
11      ?m1Name - MethodName
12      ?m1NewName - MethodName
13      ?m1ParamTypes - MethodParamTypes
14    )
15
16    :precondition (and
17
18      (notPullingUpMethods)
19      ;; Object structure check
20      (ContainsPackage ?root ?p1)
21      (ContainsClass ?p1 ?c1)
22      (ContainsMethod ?c1 ?m1)
23
24      ;; name check
25      (HasPackageName ?p1 ?p1Name)
26      (HasClassName ?c1 ?c1Name)
27      (HasMethodName ?m1 ?m1Name)
28      (not (HasMethodName ?m1 ?m1NewName))
29      (HasMethodParamTypes ?m1 ?m1ParamTypes)
30
31      ;; signature path
32      (SignaturePathTillPackage ?root ?p1Name)
33      (SignaturePathTillClass ?root ?p1Name ?c1Name)
34      (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes)
35      (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1NewName ?
            m1ParamTypes))
36
37    )
38
39    :effect (and
40
```

```
41      ;; change unique id object structure
42      ;; nothing needs to be changed in this case
43
44      ;; change name part
45      (not (HasMethodName ?m1 ?m1Name))
46      (HasMethodName ?m1 ?m1NewName)
47
48      ;; change related signature paths
49      ;; 1. sigpathTillmethod m1, via c1
50      (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?
            m1ParamTypes))
51      (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1NewName ?m1ParamTypes)
52
53      ;; 2. sigpathTillMethodPathtoken: m1's token
54      (forall (?oneMethodPathToken - MethodPathToken)
55        (when (and
56          (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
57          )
58          (and
59            ;; remove the sig path
60            (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?
                  m1Name ?m1ParamTypes ?oneMethodPathToken))
61            ;; adding the path
62            (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?m1NewName
                  ?m1ParamTypes ?oneMethodPathToken)
63          )
64        )
65      )
66    )
67  )
```

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

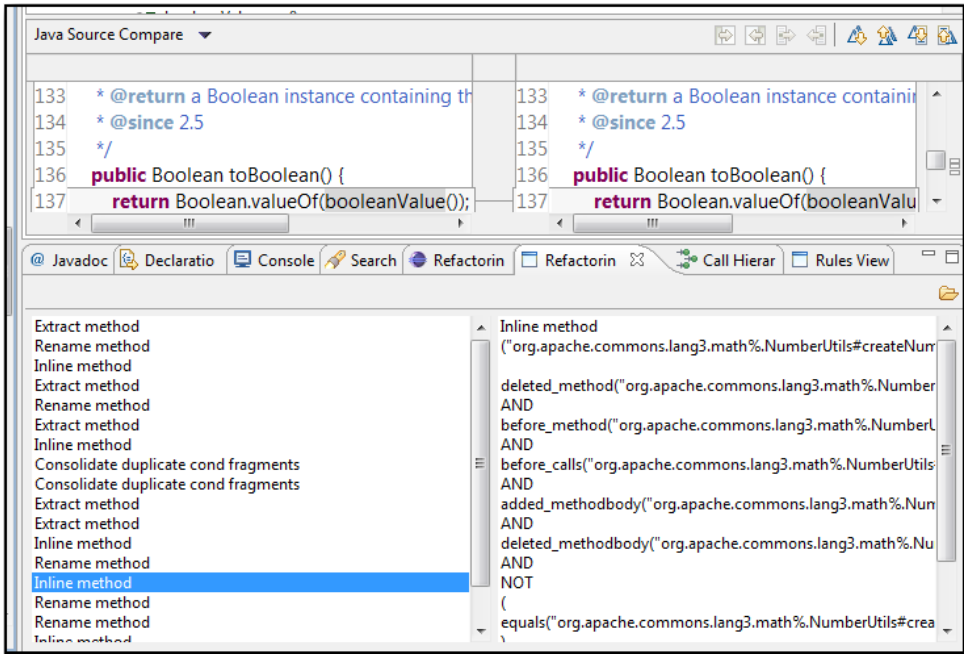

Figure 3.30 The result screen of the Exp. 3 from LSdiff.

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

Exp. ID	Component (LOC)	Applied Refactorings	Has TDRS?	Refactoring Crawler		LSdiff		TARP*		
				Result Correct?	Comput. Time (Sec.)	Result Correct?	Comput. Time (Sec.)	Result Correct?	Comput. Time (Sec.)	Adaptation-based Testing Result
1	Apache POI 3.1 (136K)	Renamed 2 independent methods in a class.	**No**	**Yes**	50.110	**Partially.** Found all but also found 1 *Inline Method* and 1 *Extract Method*	49.610	**Yes**	66.102	618 test, 100% passed
2	Google Collection 1.0 (32K)	Renamed 1 package and renamed 1 irrelevant class.	**No**	**Partially.** Found all but also found 2 *Change Method Signature*	30.544	**No** – Only found 11 *Move Method*.	25.725	**Yes**	116.031	207 test, 100% passed
3	Apache Commons Lang 3.0.1 (55K)	Renamed 2 classes and renamed 4 methods (all independent)	**No**	**Partially.** Correctly found 2 *Rename Method*, but also found 3 *Move Method* inside the renamed class.	17.975	**Partially.** Correctly found 1 *Renamed Method*, but also found 13 *Move Method*, 6 *Move Field*, 1 *Inline Method*, and 1 *Extract Method*.	11.197	**Yes**	55.196	559 test, 100% passed
4	Apache POI 3.1 (136K)	Renamed a static method and moved it to another class.	**Yes**	**No** – Found nothing.	51.300	**No** – Found nothing.	329.958	**Yes**	49.111	618 test, 100% passed
5	Apache POI 3.1 (136K)	Renamed a class and renamed one method insides this class.	**Yes**	**No** – Found nothing.	91.526	**No** – Only found 2 *Move Field*.	268.054	**Yes**	42.253	618 test, 100% passed

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

Exp. ID	Tool	Detected Refactoring History
1	Real History	Rename method 'org.apache.poi.hpsf.Section.getOffset()' to 'getOffset_REN' Rename method 'org.apache.poi.hpsf.Section.getCodepage()' to 'getCodepage_REN'
	Refactoring Crawler	Rename method 'org.apache.poi.hpsf.Section.getOffset()' to 'getOffset_REN' Rename method 'org.apache.poi.hpsf.Section.getCodepage()' to 'getCodepage_REN'
	LSdiff	Consolidate duplicate cond fragments (invalid refactoring step) Inline method 'org.apache.poi.hpsf.Section.toString()' Rename method 'org.apache.poi.hpsf.Section.getOffset()' to 'getOffset_REN' Extract method 'org.apache.poi.hpsf.Section.toString()' Rename method 'org.apache.poi.hpsf.Section.getCodepage()' to 'getCodepage_REN'
	TARP*	Rename method 'org.apache.poi.hpsf.Section.getOffset()' to 'getOffset_REN' Rename method 'org.apache.poi.hpsf.Section.getCodepage()' to 'getCodepage_REN'
2	Real History	Rename package 'com.google.common.annotations' to 'com.google.common.annotations_REN' Rename class 'com.google.common.collect.ForwardingList' to 'ForwardingList_REN'
	Refactoring Crawler	Rename package 'com.google.common.annotations' to 'com.google.common.annotations_REN' Rename class 'com.google.common.collect.ForwardingList' to 'ForwardingList_REN' Change method signature 'com.google.common.collect.ForwardingList.listIterator()' to 'ForwardingList_REN.listIterator(int)' Change method signature 'com.google.common.collect.ForwardingList.listIterator(int)' to 'ForwardingList_REN.listIterator()'
	LSdiff	Move method 'com.google.common.collect.ForwardingList.listIterator()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.subList()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.indexOf()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.lastIndexOf()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.hashCode()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.add()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.get()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.addAll()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.set()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.remove()' to 'ForwardingList_REN' Move method 'com.google.common.collect.ForwardingList.equals()' to 'ForwardingList_REN'
	TARP*	Rename package 'com.google.common.annotations' to 'com.google.common.annotations_REN' Rename class 'com.google.common.collect.ForwardingList' to 'ForwardingList_REN'

tests (if any) until all tests passed. We called the new set of tests "clean the no-error test". We took this step because we want to have "applications" which work without any problem before component upgrades. If not, when we check the test reports, we cannot tell if an error or failure resulted from the original application or ALTA*.

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 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 3.2 (Continued)

Open Source Component Refactoring Path Reconstruction Test Report

Exp. ID	Tool	Detected Refactoring History
3	Real History	Rename class 'org.apache.commons.lang3.builder.IDKey' to 'IDKey_REN' Rename method 'org.apache.commons.lang3.math.NumberUtils.max(long;long;long)' to 'max_ren' Rename class 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer' to 'MultiBackgroundInitializer_REN' Rename method 'org.apache.commons.lang3.math.NumberUtils.toByte(java.lang.String;byte)' to 'toByte_ren' Rename method 'org.apache.commons.lang3.math.NumberUtils.createBigDecimal(java.lang.String)' to 'createBigDecimal_ren' Rename method 'org.apache.commons.lang3.math.NumberUtils.min(double[])' to 'min_ren'
	Refactoring Crawler	Rename method 'org.apache.commons.lang3.math.NumberUtils.toByte(java.lang.String;byte)' to 'toByte_ren' Rename method 'org.apache.commons.lang3.math.NumberUtils.createBigDecimal(java.lang.String)' to 'createBigDecimal_ren' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.getTaskCount()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.builder.IDKey.hashCode()' to 'org.apache.commons.lang3.builder.IDKey_REN' Move method: 'org.apache.commons.lang3.builder.IDKey.equals' to 'org.apache.commons.lang3.builder.IDKey_REN.equals'
	LSdiff	Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.getResultObject()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move field: 'org.apache.commons.lang3.builder.IDKey.id' to 'org.apache.commons.lang3.builder.IDKey_REN' Move field: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.childInitializers' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.isException' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.getTaskCount' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move field: 'org.apache.commons.lang3.builder.IDKey.value' to 'org.apache.commons.lang3.builder.IDKey_REN' Move field: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.initializers' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Consolidate duplicate cond fragments (invalid refactoring step) Rename method 'org.apache.commons.lang3.math.NumberUtils.createBigDecimal()' to 'createBigDecimal_ren' Move method: 'org.apache.commons.lang3.builder.IDKey.hashCode()' to 'org.apache.commons.lang3.builder.IDKey_REN' Move method: 'org.apache.commons.lang3.builder.IDKey.equals()' to 'org.apache.commons.lang3.builder.IDKey_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.isSuccessful()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move field: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.exceptions' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move field: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.resultObjects' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Inline method: 'org.apache.commons.lang3.math.NumberUtils.createNumber()' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.getException()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.initializerNames()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.checkName()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.getInitializer()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Extract method: 'org.apache.commons.lang3.math.NumberUtils.createNumber()' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.addInitializer()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' Move method: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.initialize()' to 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN'
	TARP*	Rename class 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer' to 'MultiBackgroundInitializer_REN' Rename class 'org.apache.commons.lang3.builder.IDKey' to 'IDKey_REN' Rename method 'org.apache.commons.lang3.math.NumberUtils.toByte(java.lang.String;byte)' to 'toByte_ren' Rename method 'org.apache.commons.lang3.math.NumberUtils.min(double[])' to 'min_ren' Rename method 'org.apache.commons.lang3.math.NumberUtils.max(long;long;long)' to 'max_ren' Rename method 'org.apache.commons.lang3.math.NumberUtils.createBigDecimal(java.lang.String)' to 'createBigDecimal_ren'

Table 3.2 (Continued)

Open Source Component Refactoring Path Reconstruction Test Report

Exp. ID	Tool	Detected Refactoring History
4	Real History	Rename method 'org.apache.poi.util.StringUtil.hasMultibyte(java.lang.String)' to 'hasMultibyte_ren' Move method 'org.apache.poi.util.StringUtil.hasMultibyte_ren(java.lang.String)' to 'org.apache.poi.util.IOUtils'
	Refactoring Crawler	Found Nothing
	LSdiff	Found Nothing
	TARP*	Rename method 'org.apache.poi.util.StringUtil.hasMultibyte(java.lang.String)' to 'hasMultibyte_ren' Move method'org.apache.poi.util.StringUtil.hasMultibyte_ren(java.lang.String)' to 'org.apache.poi.util.IOUtils'
5	Real History	Rename class 'org.apache.poi.util.TempFile' to 'TempFile_REN' Rename method 'org.apache.poi.util.TempFile_REN.createTempFile(java.lang.String, java.lang.String)' to 'createTempFile_REN'
	Refactoring Crawler	Found Nothing
	LSdiff	Move Field 'org.apache.poi.util.TempFile.rnd' to 'TempFile_REN.rnd' Move Field 'org.apache.poi.util.TempFile.dir' to 'TempFile_REN'
	TARP*	Rename class 'org.apache.poi.util.TempFile' to 'TempFile_REN' Rename method 'org.apache.poi.util.TempFile_REN.createTempFile(java.lang.String, java.lang.String)' to 'createTempFile_REN'

Table 3.3 Open Source Component Automatic Adaptation Results

Exp. ID	Component (LOC)	Applied Refactorings	Has TDRS?	Official Tests (as Applications) Execution Results		
				Run with the Old Component	Run with the New Component **without** ALTA*	Run with the New Component **with** ALTA*
1	Apache POI 3.1 (136K)	Renamed 2 independent methods in a class.	**No**	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.583 (Sec)	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.593 (Sec)	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.682 (Sec)
2	Google Collection 1.0 (32K)	Renamed 1 package and renamed 1 irrelevant class.	**No**	220 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 0.412 (Sec)	220 Tests, 80% Pass, **44 Errors**, 0 Failures. Time: 0.036 (Sec)	220 Tests, 100% Pass, **0 Errors**, 0 Failures. Time: 0.429 (Sec)
3	Apache Commons Lang 3.0.1 (55K)	Renamed 2 classes and renamed 4 methods (all independent)	**No**	2013 Tests, 100% Pass, **0 Errors**, 0 Failures. Time: 7.596 (Sec)	2013 Tests, 99.7% Pass, **7 Errors**, 0 Failures. Time: 7.555 (Sec)	2013 Tests, 100% Pass, **0 Errors**, 0 Failures. Time: 8.213 (Sec)
4	Apache POI 3.1 (136K)	Moved 1 static method from one class to another, then renamed that method.	**Yes**	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.510 (Sec)	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.489 (Sec)	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.677 (Sec)
5	Apache POI 3.1 (136K)	Renamed 1 method in a class, and renamed that class.	**Yes**	927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.596 (Sec)	927 Tests, 99.7% Pass, **3 Errors**, 0 Failures. Time: 1.571 (Sec)	927 Tests, 100% Pass, **0 Errors**, 0 Failures. Time: 2.456 (Sec)

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="89_ApacheCommonsLang3.0.1-manyChanges_new" name="IDKey_REN" matchStrategy="1"
      input="/src<org.apache.commons.lang3.builder{IDKey.java[IDKey" id="org.eclipse.jdt.ui.rename.type"
      flags="589830" description="Rename type 'IDKey'" comment="Rename type
      'org.apache.commons.lang3.builder.IDKey' to 'IDKey_REN' - Original project:
      '89_ApacheCommonsLang3.0.1-manyChanges_new' - Original element:
      'org.apache.commons.lang3.builder.IDKey' - Renamed element:
      'org.apache.commons.lang3.builder.IDKey_REN' - Update references to refactored element - Update
      textual occurrences in comments and strings"/>
    <refactoring version="1.0" references="true" project="89_ApacheCommonsLang3.0.1-manyChanges_new"
      name="max_ren" input="/src<org.apache.commons.lang3.math{NumberUtils.java
      [NumberUtils~max~J~J~J" id="org.eclipse.jdt.ui.rename.method" flags="589830" description="Rename
      method 'max'" comment="Rename method 'org.apache.commons.lang3.math.NumberUtils.max(...)' to
      'max_ren' - Original project: '89_ApacheCommonsLang3.0.1-manyChanges_new' - Original element:
      'org.apache.commons.lang3.math.NumberUtils.max(...)' - Renamed element:
      'org.apache.commons.lang3.math.NumberUtils.max_ren(...)' - Update references to refactored element"
      deprecate="true" delegate="false"/>
    <refactoring version="1.0" textual="false" similarDeclarations="false" references="true" qualified="false"
      project="89_ApacheCommonsLang3.0.1-manyChanges_new" name="MultiBackgroundInitializer_REN"
      matchStrategy="1" 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_ApacheCommonsLang3.0.1-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"/>
    <refactoring version="1.0" references="true" project="89_ApacheCommonsLang3.0.1-manyChanges_new"
      name="toByte_ren" input="/src<org.apache.commons.lang3.math{NumberUtils.java
      [NumberUtils~toByte~QString;~B" id="org.eclipse.jdt.ui.rename.method" flags="589830"
      description="Rename method 'toByte'" comment="Rename method
      'org.apache.commons.lang3.math.NumberUtils.toByte(...)' to 'toByte_ren' - Original project:
      '89_ApacheCommonsLang3.0.1-manyChanges_new' - Original element:
      'org.apache.commons.lang3.math.NumberUtils.toByte(...)' - Renamed element:
      'org.apache.commons.lang3.math.NumberUtils.toByte_ren(...)' - Update references to refactored
      element" deprecate="true" delegate="false"/>
    <refactoring version="1.0" references="true" project="89_ApacheCommonsLang3.0.1-manyChanges_new"
      name="createBigDecimal_ren" input="/src<org.apache.commons.lang3.math{NumberUtils.java
      [NumberUtils~createBigDecimal~QString;" id="org.eclipse.jdt.ui.rename.method" flags="589830"
      description="Rename method 'createBigDecimal'" comment="Rename method
      'org.apache.commons.lang3.math.NumberUtils.createBigDecimal(...)' to 'createBigDecimal_ren' -
      Original project: '89_ApacheCommonsLang3.0.1-manyChanges_new' - Original element:
      'org.apache.commons.lang3.math.NumberUtils.createBigDecimal(...)' - Renamed element:
      'org.apache.commons.lang3.math.NumberUtils.createBigDecimal_ren(...)' - Update references to
      refactored element" deprecate="true" delegate="false"/>
    <refactoring version="1.0" references="true" project="89_ApacheCommonsLang3.0.1-manyChanges_new"
      name="min_ren" input="/src<org.apache.commons.lang3.math{NumberUtils.java
      [NumberUtils~min~\[D" id="org.eclipse.jdt.ui.rename.method" flags="589830" description="Rename
      method 'min'" comment="Rename method 'org.apache.commons.lang3.math.NumberUtils.min(...)' to
      'min_ren' - Original project: '89_ApacheCommonsLang3.0.1-manyChanges_new' - Original element:
      'org.apache.commons.lang3.math.NumberUtils.min(...)' - Renamed element:
      'org.apache.commons.lang3.math.NumberUtils.min_ren(...)' - Update references to refactored element"
      deprecate="true" delegate="false"/>
  </session>
```

APPENDIX B. TARP* Domain File Ver. 1.2

```
(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
```

```
29        MethodModifer - object
30        FieldModifier  - object
31
32        ;; path tokens
33        MethodPathToken - Object
34
35     )
36
37    (:predicates
38          ;;tree structures
39          (ContainsPackage    ?r - APIRoot  ?p - Package)
40          (ContainsClass      ?p - Package     ?c - Class)
41          (ContainsMethod     ?c - Class     ?m -  Method)
42          (ContainsField    ?c - Class     ?f -  Field)
43
44          ;;path token
45          (ContainsMethodPathToken    ?m - Method    ?t - MethodPathToken)
46
47          (HasPackageName     ?p - Package  ?pName - PackageName)
48          (HasClassName     ?c - Class     ?cName - ClassName)
49          (HasMethodName      ?m - Method    ?mName - MethodName)
50          (HasFieldName     ?f - Field     ?fName - FieldName)
51
52          ;;types
53          (HasMethodParamTypes ?m - Method    ?types - MethodParamTypes)
54
55          ;; modifiers
56          (HasMethodModifier    ?m - Method    ?mod - MethodModifier)
57          (HasFieldModifier  ?f - Field     ?mod - FieldModifier)
58
59          ;; signature paths
60          ;; package
61          (SignaturePathTillPackage ?r - APIRoot ?pName - PackageName)
62
63          ;; class
```

```
64          (SignaturePathTillClass ?r - APIRoot ?pName - PackageName ?cName -
                ClassName)
65
66          ;; method -- remember to include types
67          (SignaturePathTillMethod ?r - APIRoot ?pName - PackageName ?cName -
                ClassName
68                  ?mName - MethodName ?types - MethodParamTypes)
69          (SignaturePathTillMethodPathToken ?r - APIRoot ?pName - PackageName ?
                cName - ClassName
70                  ?mName - MethodName ?mParamTypes - MethodParamTypes ?mToken -
                        MethodPathToken)
71
72        ;; field
73          (SignaturePathTillField ?r - APIRoot ?pName - PackageName ?cName -
                ClassName
74                  ?fName - FieldName)
75
76          ;; inherit
77          (Inherit ?classChild - Class ?classParent - Class )
78
79          ;; pull-up transactions
80          (notPullingUpMethods)
81
82          ;; note: all objects, not names
83          (MethodDuringPullingUp ?m1 - Method ?m1ParamTypes - MethodParamTypes
84             ?cFrom - Class ?cTo - Class)
85
86  )
87
88
89  ;;
90  ;; ============================ actions ============================
91  ;;
92
93  ;; rename field
```

```
94  (:action renameField
95       :parameters (
96        ?root - APIRoot
97        ?p1 - Package
98        ?p1Name - PackageName
99        ?c1 - Class
100       ?c1Name - ClassName
101       ?f1 - Field
102       ?f1Name - FieldName
103       ?f1NewName - FieldName
104
105       )
106      :precondition (and
107
108       (notPullingUpMethods)
109
110       ;;uuid structure check
111       (ContainsPackage ?root ?p1)
112       (ContainsClass ?p1 ?c1)
113         (ContainsField ?c1 ?f1)
114
115         ;; name check
116         (HasPackageName ?p1 ?p1Name)
117         (HasClassName ?c1 ?c1Name)
118         (HasFieldName ?f1 ?f1Name)
119         (not (HasFieldName ?f1 ?f1NewName))
120
121         ;; signature path
122         (SignaturePathTillPackage ?root ?p1Name)
123         (SignaturePathTillClass ?root ?p1Name ?c1Name)
124     (SignaturePathTillField ?root ?p1Name ?c1Name ?f1Name)
125     (not (SignaturePathTillField ?root ?p1Name ?c1Name ?f1NewName))
126
127       )
128      :effect (and
```

```
129
130         ;;  change unique id object structure
131         ;; nothing needs to be changed
132
133         ;; change name part
134         (not (HasFieldName ?f1 ?f1Name))
135           (HasFieldName ?f1 ?f1NewName)
136
137       ;; change related signature paths
138         ;; 1. sigpathTillmethod m1, via c1
139         (not (SignaturePathTillField ?root ?p1Name ?c1Name ?f1Name))
140         (SignaturePathTillField ?root ?p1Name ?c1Name ?f1NewName)
141
142
143        )
144   )
145
146    ;; we only allow to move static field right now.
147   (:action moveField
148        :parameters (
149         ?root - APIRoot
150         ?p1 - Package
151         ?p1Name - PackageName
152         ?CFrom - Class
153         ?cFromName - ClassName
154         ?f1 - Field
155         ?f1Name - FieldName
156         ?f1Modifier - FieldModifier
157
158         ?p2 - Package
159         ?p2Name - PackageName
160         ?cTo - Class
161         ?cToName - ClassName
162         )
163        :precondition (and
```

```
164
165         (notPullingUpMethods)
166
167         ;;uuid structure check
168         (ContainsPackage ?root ?p1)
169         (ContainsPackage ?root ?p2)
170           (ContainsClass ?p1 ?cFrom)
171           (ContainsClass ?p2 ?cTo)
172           (ContainsField ?cFrom ?f1)
173           (not (ContainsField ?cTo ?f1))
174
175           ;; name check
176           (HasPackageName ?p1 ?p1Name)
177           (HasPackageName ?p2 ?p2Name)
178           (HasClassName ?cFrom ?cFromName)
179           (HasClassName ?cTo ?cToName)
180           (HasFieldName ?f1 ?f1Name)
181           (HasFieldModifier ?f1 ?f1Modifier)
182           ;; critical part
183           (= ?f1Modifier Static_Field)
184
185           ;; signature path
186           (SignaturePathTillPackage ?root ?p1Name)
187           (SignaturePathTillPackage ?root ?p2Name)
188       (SignaturePathTillClass ?root ?p1Name ?cFromName)
189       (SignaturePathTillClass ?root ?p2Name ?cToName)
190       (SignaturePathTillField ?root ?p1Name ?cFromName ?f1Name)
191       (not (SignaturePathTillField ?root ?p2Name ?cToName ?f1Name))
192
193         )
194        :effect (and
195
196         ;;  change unique id object structure
197         (not (ContainsField ?cFrom ?f1))
198           (ContainsField ?cTo ?f1)
```

```
199
200         ;; change related signature paths
201           ;; 1. sigpathTillField f1, via cFrom
202           (not (SignaturePathTillField ?root ?p1Name ?cFromName ?f1Name))
203         ;; 2. 1. sigpathTillField f1, via cTo
204         (SignaturePathTillField ?root ?p2Name ?cToName ?f1Name)
205
206          )
207   )
208
209
210   ;; we only allow to move static method right now.
211   (:action moveMethod
212         :parameters (
213          ?root - APIRoot
214          ?p1 - Package
215          ?p1Name - PackageName
216          ?CFrom - Class
217          ?cFromName - ClassName
218          ?m1 - Method
219          ?m1Name - MethodName
220          ?m1ParamTypes - MethodParamTypes
221          ?m1Modifier - MethodModifier
222
223          ?p2 - Package
224          ?p2Name - PackageName
225          ?cTo - Class
226          ?cToName - ClassName
227          )
228         :precondition (and
229
230          (notPullingUpMethods)
231
232          ;;uuid structure check
233          (ContainsPackage ?root ?p1)
```

```
234         (ContainsPackage ?root ?p2)
235           (ContainsClass ?p1 ?cFrom)
236           (ContainsClass ?p2 ?cTo)
237           (ContainsMethod ?cFrom ?m1)
238           (not (ContainsMethod ?cTo ?m1))
239
240           ;; name check
241           (HasPackageName ?p1 ?p1Name)
242           (HasPackageName ?p2 ?p2Name)
243           (HasClassName ?cFrom ?cFromName)
244           (HasClassName ?cTo ?cToName)
245           (HasMethodName ?m1 ?m1Name)
246           (HasMethodParamTypes ?m1 ?m1ParamTypes)
247           (HasMethodModifier ?m1 ?m1Modifier)
248           ;; critical part
249          (= ?m1Modifier Static_Method)
250
251           ;; signature path
252           (SignaturePathTillPackage ?root ?p1Name)
253           (SignaturePathTillPackage ?root ?p2Name)
254       (SignaturePathTillClass ?root ?p1Name ?cFromName)
255       (SignaturePathTillClass ?root ?p2Name ?cToName)
256       (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
257       (not (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)
              )
258
259         )
260        :effect (and
261
262         ;;  change unique id object structure
263         (not (ContainsMethod ?cFrom ?m1))
264           (ContainsMethod ?cTo ?m1)
265
266       ;; change related signature paths
267         ;; 1. sigpathTillmethod m1, via cFrom
```

```
268         (not (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?
                m1ParamTypes))
269       ;; 2. 1. sigpathTillmethod m1, via cTo
270       (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)
271
272         ;; 3. sigpathTillMethodPathtoken: m1's token, remove all via cFrom and
                add all via cTo
273         (forall (?oneMethodPathToken - MethodPathToken)
274             (when (and
275               (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
276               )
277               (and
278                 ;; remove the sig path via cFrom
279                 (not (SignaturePathTillMethodPathToken ?root ?p1Name ?cFromName ?
                        m1Name ?m1ParamTypes ?oneMethodPathToken))
280                 ;; adding the path via cTo
281               (SignaturePathTillMethodPathToken ?root ?p2Name ?cToName ?m1Name ?
                      m1ParamTypes ?oneMethodPathToken)
282               )
283             )
284         )
285       )
286   )
287
288   ;; rename method
289   (:action renameMethod
290         :parameters (
291          ?root - APIRoot
292          ?p1 - Package
293          ?p1Name - PackageName
294          ?c1 - Class
295          ?c1Name - ClassName
296          ?m1 - Method
297          ?m1Name - MethodName
298          ?m1NewName - MethodName
```

```
299          ?m1ParamTypes - MethodParamTypes
300          )
301         :precondition (and
302
303          (notPullingUpMethods)
304
305          ;;uuid structure check
306          (ContainsPackage ?root ?p1)
307          (ContainsClass ?p1 ?c1)
308            (ContainsMethod ?c1 ?m1)
309
310            ;; name check
311            (HasPackageName ?p1 ?p1Name)
312            (HasClassName ?c1 ?c1Name)
313            (HasMethodName ?m1 ?m1Name)
314            (not (HasMethodName ?m1 ?m1NewName))
315            (HasMethodParamTypes ?m1 ?m1ParamTypes)
316
317            ;; signature path
318            (SignaturePathTillPackage ?root ?p1Name)
319            (SignaturePathTillClass ?root ?p1Name ?c1Name)
320        (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes)
321        (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1NewName ?
               m1ParamTypes))
322
323          )
324         :effect (and
325
326          ;;  change unique id object structure
327          ;; nothing needs to be changed
328
329          ;; change name part
330          (not (HasMethodName ?m1 ?m1Name))
331            (HasMethodName ?m1 ?m1NewName)
332
```

```
333       ;; change related signature paths
334         ;; 1. sigpathTillmethod m1, via c1
335         (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes
                ))
336         (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1NewName ?m1ParamTypes)
337
338         ;; 3. sigpathTillMethodPathtoken: m1's token
339         (forall (?oneMethodPathToken - MethodPathToken)
340             (when (and
341               (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
342               )
343               (and
344                 ;; remove the sig path
345                 (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?
                        m1Name ?m1ParamTypes ?oneMethodPathToken))
346                 ;; adding the path
347               (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?m1NewName
                    ?m1ParamTypes ?oneMethodPathToken)
348               )
349             )
350         )
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
```

```
365          ?p2 - Package
366          ?p2Name - PackageName
367          ?cTo - Class
368          ?cToName - ClassName
369          )
370         :precondition (and
371
372          ;; not running pulling up method
373          (notPullingUpMethods)
374          (not (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo))
375
376          ;;uuid structure check
377          (ContainsPackage ?root ?p1)
378          (ContainsPackage ?root ?p2)
379            (ContainsClass ?p1 ?cFrom)
380            (ContainsClass ?p2 ?cTo)
381            (ContainsMethod ?cFrom ?m1)
382            (not (ContainsMethod ?cTo ?m1))
383
384            ;; name check
385            (HasPackageName ?p1 ?p1Name)
386            (HasPackageName ?p2 ?p2Name)
387            (HasClassName ?cFrom ?cFromName)
388            (HasClassName ?cTo ?cToName)
389            (HasMethodName ?m1 ?m1Name)
390            (HasMethodParamTypes ?m1 ?m1ParamTypes)
391
392            ;; cFrom extends cTo
393          (Inherit ?cFrom ?cTo)
394
395            ;; signature path
396            (SignaturePathTillPackage ?root ?p1Name)
397            (SignaturePathTillPackage ?root ?p2Name)
398        (SignaturePathTillClass ?root ?p1Name ?cFromName)
399        (SignaturePathTillClass ?root ?p2Name ?cToName)
```

```
400         (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
401         (not (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)
                )
402
403          )
404         :effect (and
405            ;; just start up
406            (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo)
407          (not (notPullingUpMethods))
408         )
409    )
410
411    (:action pullUpMethods_mergingSiblings
412         :parameters (
413          ?root - APIRoot
414
415          ?p1 - Package
416          ?p1Name - PackageName
417          ?CFrom - Class
418          ?cFromName - ClassName
419          ?m1 - Method
420          ?m1Name - MethodName
421          ?m1ParamTypes - MethodParamTypes
422          ?m1Modifer - MethodModifier
423
424          ?p2 - Package
425          ?p2Name - PackageName
426          ?cSibling - Class
427          ?cSiblingName - ClassName
428          ?m2 - Method
429          ;;name note: m2 should be in m1Name!
430          ;;type note: m2 should has the same param types!
431
432          ?cTo - Class
433          )
```

```
434         :precondition (and
435
436          ;;  must during pulling up process
437          (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo)
438          (not (notPullingUpMethods))
439
440          ;;uuid structure check
441          (ContainsPackage ?root ?p1)
442          (ContainsPackage ?root ?p2)
443            (ContainsClass ?p1 ?cFrom)
444            (ContainsClass ?p2 ?cSibling)
445            (ContainsMethod ?cFrom ?m1)
446            (ContainsMethod ?cSibling ?m2)
447
448            ;; name check
449            (HasPackageName ?p1 ?p1Name)
450            (HasPackageName ?p2 ?p2Name)
451            (HasClassName ?cFrom ?cFromName)
452            (HasClassName ?cSibling ?cSiblingName)
453            (HasMethodName ?m1 ?m1Name)
454            (HasMethodName ?m2 ?m1Name)  ;; this part is very critical
455
456            ;; types
457            (HasMethodParamTypes ?m1 ?m1ParamTypes)
458            (HasMethodParamTypes ?m2 ?m1ParamTypes)  ;; this part is very critical
459
460          ;; modifier
461          ;; we don't need to check the static part.
462          ;; this is okay because pull up can be mixed.
463          ;; anyway, I think that eclipse has a bug in this issue
464          ;; so I am going to forbid it.
465          (HasMethodModifier ?m1 ?m1Modifer)
466          (HasMethodModifier ?m2 ?m1Modifer)
467
468          ;; inherit
```

```
469           ;; cFrom extends cTo
470         (Inherit ?cFrom ?cTo)
471         (Inherit ?cSibling ?cTo)
472
473           ;; signature path
474           (SignaturePathTillPackage ?root ?p1Name)
475           (SignaturePathTillPackage ?root ?p2Name)
476       (SignaturePathTillClass ?root ?p1Name ?cFromName)
477       (SignaturePathTillClass ?root ?p2Name ?cSiblingName)
478       (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
479       ;; this part is very critical
480       (SignaturePathTillMethod ?root ?p2Name ?cSiblingName ?m1Name ?m1ParamTypes)
481
482         )
483        :effect (and
484           ;; move all its path token
485         (forall (?oneMethodPathToken - MethodPathToken)
486           (when (and
487               (ContainsMethodPathToken ?m2 ?oneMethodPathToken)
488               )
489               (and
490                 ;; 1. move path token to m1
491                 (not (ContainsMethodPathToken ?m2 ?oneMethodPathToken))
492                 (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
493
494               ;; remove all method pathtoken signature paths through m2
495               (not (SignaturePathTillMethodPathToken ?root ?p2Name ?cSiblingName
                       ?m1Name ?m1ParamTypes ?oneMethodPathToken))
496
497               ;; add new method pathtoken signature paths through m1
498               (SignaturePathTillMethodPathToken ?root ?p1Name ?cFromName ?m1Name
                       ?m1ParamTypes ?oneMethodPathToken )
499
500                )
501             )
```

```
502           )
503
504         ;;representative
505         ;; remove all rest relations wih m2
506         ;; with container
507         (not (ContainsMethod ?cSibling ?m2))
508         ;; with name
509         (not (HasMethodName ?m2 ?m1Name))
510         ;; with types
511         (not (HasMethodParamTypes ?m2 ?m1ParamTypes))
512         ;; with modifier
513         (not (HasMethodModifier  ?m2 ?m1Modifer))
514
515         ;; remove signature path till m2
516       (not (SignaturePathTillMethod ?root ?p2Name ?cSiblingName ?m1Name ?
              m1ParamTypes))
517        )
518   )
519
520   (:action pullupMethods_end
521         :parameters (
522          ?root - APIRoot
523          ?p1 - Package
524          ?p1Name - PackageName
525          ?CFrom - Class
526          ?cFromName - ClassName
527          ?m1 - Method
528          ?m1Name - MethodName
529          ?m1ParamTypes - MethodParamTypes
530
531          ?p2 - Package
532          ?p2Name - PackageName
533          ?cTo - Class
534          ?cToName - ClassName
535          )
```

```
536         :precondition (and
537
538
539           ;;  must during pulling up process
540           (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo)
541           (not (notPullingUpMethods))
542
543           ;;uuid structure check
544           (ContainsPackage ?root ?p1)
545           (ContainsPackage ?root ?p2)
546             (ContainsClass ?p1 ?cFrom)
547             (ContainsClass ?p2 ?cTo)
548             (ContainsMethod ?cFrom ?m1)
549             (not (ContainsMethod ?cTo ?m1))
550
551             ;; name check
552             (HasPackageName ?p1 ?p1Name)
553             (HasPackageName ?p2 ?p2Name)
554             (HasClassName ?cFrom ?cFromName)
555             (HasClassName ?cTo ?cToName)
556             (HasMethodName ?m1 ?m1Name)
557             (HasMethodParamTypes ?m1 ?m1ParamTypes)
558
559             ;; cFrom extends cTo
560           (Inherit ?cFrom ?cTo)
561
562             ;; signature path
563             (SignaturePathTillPackage ?root ?p1Name)
564             (SignaturePathTillPackage ?root ?p2Name)
565         (SignaturePathTillClass ?root ?p1Name ?cFromName)
566         (SignaturePathTillClass ?root ?p2Name ?cToName)
567         (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
568         (not (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)
              )
569
```

```
570          )
571         :effect (and
572
573          ;;  change unique id object structure
574          (not (ContainsMethod ?cFrom ?m1))
575            (ContainsMethod ?cTo ?m1)
576
577        ;; change related signature paths
578          ;; 1. sigpathTillmethod m1, via cFrom
579          (not (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?
                 m1ParamTypes))
580        ;; 2. 1. sigpathTillmethod m1, via cTo
581        (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)
582
583          ;; 3. sigpathTillMethodPathtoken: m1's token, remove all via cFrom and
                 add all via cTo
584          (forall (?oneMethodPathToken - MethodPathToken)
585              (when (and
586                (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
587                )
588                (and
589                  ;; remove the sig path via cFrom
590                  (not (SignaturePathTillMethodPathToken ?root ?p1Name ?cFromName ?
                         m1Name ?m1ParamTypes ?oneMethodPathToken))
591                  ;; adding the path via cTo
592                (SignaturePathTillMethodPathToken ?root ?p2Name ?cToName ?m1Name ?
                       m1ParamTypes ?oneMethodPathToken)
593                )
594              )
595          )
596
597            ;; end this process
598            (not (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo))
599          (notPullingUpMethods)
600         )
```

```
601   )
602
603
604   (:action renameClass
605         :parameters (
606          ?root - APIRoot
607          ?p1 - Package
608          ?p1Name - PackageName
609          ?C1 - Class
610          ?c1Name - ClassName
611          ?c1NewName - ClassName
612          )
613         :precondition (and
614
615          (notPullingUpMethods)
616
617          ;; object structure
618          (ContainsPackage ?root ?p1)
619            (ContainsClass ?p1 ?c1)
620
621            ;; name
622            (HasPackageName ?p1 ?p1Name)
623            (HasClassName ?c1 ?c1Name)
624            (not (HasClassName ?c1 ?c1NewName))
625
626            ;; sig path
627            (SignaturePathTillPackage ?root ?p1Name)
628       (SignaturePathTillClass ?root ?p1Name ?c1Name)
629       (not (SignaturePathTillClass ?root ?p1Name ?c1NewName))
630          )
631         :effect (and
632          ;; the object structure didn't change.
633          ;; change object c1's name relation
634            (not (HasClassName ?c1 ?c1Name))
635            (HasClassName ?c1 ?c1NewName)
```

```
636
637             ;; change sig path till class
638             (not (SignaturePathTillClass ?root ?p1Name ?c1Name))
639         (SignaturePathTillClass ?root ?p1Name ?c1NewName)
640
641         ;; change sig path till method
642           (forall (?oneMethod - Method
643                 ?oneMethodName - MethodName
644                 ?oneMethodParamTypes - MethodParamTypes)
645             (when (and
646                 (ContainsMethod ?c1 ?oneMethod)
647                 (HasMethodName ?oneMethod ?oneMethodName)
648                 (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
649                 )
650                 (and
651                 (SignaturePathTillMethod ?root ?p1Name ?c1NewName ?oneMethodName ?
                        oneMethodParamTypes)
652                 (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?oneMethodName
                        ?oneMethodParamTypes))
653                 )
654               )
655           )
656
657           ;; change sig path till method's path token
658
659           (forall (?oneMethod - Method
660                 ?oneMethodName - MethodName
661                 ?oneMethodParamTypes - MethodParamTypes
662                 ?oneMethodPathToken - MethodPathToken)
663             (when (and
664                 (ContainsMethod ?c1 ?oneMethod)
665                 (HasMethodName ?oneMethod ?oneMethodName)
666                 (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
667                 (ContainsMethodPathToken ?oneMethod ?oneMethodPathToken)
668                 )
```

```
669                (and
670                (SignaturePathTillMethodPathToken ?root ?p1Name ?c1NewName ?
                       oneMethodName ?oneMethodParamTypes ?oneMethodPathToken )
671                (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?
                       oneMethodName ?oneMethodParamTypes ?oneMethodPathToken))
672                 )
673              )
674            )
675
676            ;; change sig path till field
677          (forall (?oneField - Field
678                ?oneFieldName - FieldName )
679            (when (and
680                (ContainsField ?c1 ?oneField)
681                (HasFieldName ?oneField ?oneFieldName)
682                )
683                (and
684                (SignaturePathTillField ?root ?p1Name ?c1NewName ?oneFieldName)
685                (not (SignaturePathTillField ?root ?p1Name ?c1Name ?oneFieldName))
686                )
687              )
688          )
689        )
690   )
691
692   (:action moveClass
693         :parameters (
694          ?root - APIRoot
695          ?p1 - Package
696          ?p1Name - PackageName
697          ?C1 - Class
698          ?c1Name - ClassName
699          ;; target
700          ?p2 - Package
701          ?p2Name - PackageName
```

```
702         )
703        :precondition (and
704
705         (notPullingUpMethods)
706
707         ;;uuid structure check
708         (ContainsPackage ?root ?p1)
709         (ContainsPackage ?root ?p2)
710           (ContainsClass ?p1 ?c1)
711           (not (ContainsClass ?p2 ?c1))
712           ;; name check
713           (HasPackageName ?p1 ?p1Name)
714           (HasPackageName ?p2 ?p2Name)
715           (HasClassName ?c1 ?c1Name)
716
717           ;; signature path
718           (SignaturePathTillPackage ?root ?p1Name)
719           (SignaturePathTillPackage ?root ?p2Name)
720       (SignaturePathTillClass ?root ?p1Name ?c1Name)
721       (not (SignaturePathTillClass ?root ?p2Name ?c1Name)) ;; this prevents name-
              conflict after moving
722                              ;; without a forall !!!!
723         )
724        :effect (and
725           ;;uuid structure change
726         (not (ContainsClass ?p1 ?c1))
727           (ContainsClass ?p2 ?c1)
728            ;; change sig path till class
729           (not (SignaturePathTillClass ?root ?p1Name ?c1Name))
730       (SignaturePathTillClass ?root ?p2Name ?c1Name)
731
732       ;; change sig path till method
733         (forall (?oneMethod - Method
734                ?oneMethodName - MethodName
735                ?oneMethodParamTypes - MethodParamTypes)
```

```
736             (when (and
737                 (ContainsMethod ?c1 ?oneMethod)
738                 (HasMethodName ?oneMethod ?oneMethodName)
739                 (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
740                 )
741                 (and
742                 (SignaturePathTillMethod ?root ?p2Name ?c1Name ?oneMethodName ?
                        oneMethodParamTypes)
743                 (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?oneMethodName
                        ?oneMethodParamTypes))
744                 )
745               )
746           )
747
748           ;; change sig path till method's path token
749
750           (forall (?oneMethod - Method
751                 ?oneMethodName - MethodName
752                 ?oneMethodParamTypes - MethodParamTypes
753                 ?oneMethodPathToken - MethodPathToken)
754             (when (and
755                 (ContainsMethod ?c1 ?oneMethod)
756                 (HasMethodName ?oneMethod ?oneMethodName)
757                 (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
758                 (ContainsMethodPathToken ?oneMethod ?oneMethodPathToken)
759                 )
760                 (and
761                 (SignaturePathTillMethodPathToken ?root ?p2Name ?c1Name ?
                        oneMethodName ?oneMethodParamTypes ?oneMethodPathToken )
762                 (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?
                        oneMethodName ?oneMethodParamTypes ?oneMethodPathToken))
763                  )
764               )
765             )
766
```

```
767           ;; change sig path till field
768         (forall (?oneField - Field
769               ?oneFieldName - FieldName )
770           (when (and
771               (ContainsField ?c1 ?oneField)
772               (HasFieldName ?oneField ?oneFieldName)
773               )
774               (and
775               (SignaturePathTillField ?root ?p2Name ?c1Name ?oneFieldName)
776               (not (SignaturePathTillField ?root ?p1Name ?c1Name ?oneFieldName))
777               )
778             )
779         )
780       )
781   )
782
783   (:action renamePackage
784         :parameters (
785          ?root - APIRoot
786          ?p1 - Package
787          ?p1Name - PackageName
788          ?p1NewName - PackageName
789          )
790         :precondition (and
791
792          (notPullingUpMethods)
793
794          (ContainsPackage ?root ?p1)
795            (HasPackageName ?p1 ?p1Name)
796            (not (HasPackageName ?p1 ?p1NewName))
797
798            (SignaturePathTillPackage ?root ?p1Name)
799        (not (SignaturePathTillPackage ?root ?p1NewName ))
800          )
801         :effect (and
```

```
802         ;; change uuid object structure
803           (not (HasPackageName ?p1 ?p1Name))
804           (HasPackageName ?p1 ?p1NewName)
805
806           ;; change sig path till package
807           (SignaturePathTillPackage ?root ?p1NewName)
808       (not (SignaturePathTillPackage ?root ?p1Name ))
809
810            ;; change sig path till class
811           (forall (?oneClass - Class
812               ?oneClassName - ClassName
813           )
814           (when (and
815               (ContainsClass ?p1 ?oneClass)
816               (HasClassName ?oneClass ?oneClassName)
817               )
818               (and
819               (SignaturePathTillClass ?root ?p1NewName ?oneClassName)
820               (not (SignaturePathTillClass ?root ?p1Name ?oneClassName ))
821               )
822             )
823         )
824
825       ;; change sig path till method
826         (forall (
827               ?oneClass - Class
828               ?oneClassName - ClassName
829               ?oneMethod - Method
830               ?oneMethodName - MethodName
831               ?oneMethodParamTypes - MethodParamTypes)
832           (when (and
833               (ContainsClass ?p1 ?oneClass)
834               (HasClassName ?oneClass ?oneClassName)
835               (ContainsMethod ?oneClass ?oneMethod)
836               (HasMethodName ?oneMethod ?oneMethodName)
```

```
837             (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
838             )
839             (and
840             (SignaturePathTillMethod ?root ?p1NewName ?oneClassName ?
                    oneMethodName ?oneMethodParamTypes)
841             (not (SignaturePathTillMethod ?root ?p1Name ?oneClassName ?
                    oneMethodName ?oneMethodParamTypes))
842             )
843         )
844     )
845
846     ;; change sig path till method's path token
847
848     (forall (
849           ?oneClass - Class
850           ?oneClassName - ClassName
851           ?oneMethod - Method
852           ?oneMethodName - MethodName
853           ?oneMethodParamTypes - MethodParamTypes
854           ?oneMethodPathToken - MethodPathToken)
855       (when (and
856           (ContainsClass ?p1 ?oneClass)
857           (HasClassName ?oneClass ?oneClassName)
858           (ContainsMethod ?oneClass ?oneMethod)
859           (HasMethodName ?oneMethod ?oneMethodName)
860           (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
861           (ContainsMethodPathToken ?oneMethod ?oneMethodPathToken)
862           )
863           (and
864           (SignaturePathTillMethodPathToken ?root ?p1NewName ?oneClassName ?
                    oneMethodName ?oneMethodParamTypes ?oneMethodPathToken )
865           (not (SignaturePathTillMethodPathToken ?root ?p1Name ?oneClassName
                    ?oneMethodName ?oneMethodParamTypes ?oneMethodPathToken))
866            )
867         )
```

```
868           )
869
870           ;; change sig path till field
871         (forall (
872               ?oneClass - Class
873               ?oneClassName - ClassName
874               ?oneField - Field
875               ?oneFieldName - FieldName )
876           (when (and
877               (ContainsClass ?p1 ?oneClass)
878               (HasClassName ?oneClass ?oneClassName)
879               (ContainsField ?oneClass ?oneField)
880               (HasFieldName ?oneField ?oneFieldName)
881               )
882               (and
883               (SignaturePathTillField ?root ?p1NewName ?oneClassName ?
                      oneFieldName)
884               (not (SignaturePathTillField ?root ?p1Name ?oneClassName ?
                      oneFieldName))
885               )
886             )
887         )
888        )
889   )
890
891 )
```

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
    dummyFielObject - 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_MTDTYPESclassjavalangobject - MethodParamTypes
    RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals - Method
    MTD_MTDequals - MethodName

    MethodParamTypes_MTDTYPESvoid - 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_MTDTYPESclassjavalangstringclassorgapachecommonslang3concurrent<LineWrapMark>
        backgroundinitializer - MethodParamTypes
    RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer<LineWrapMark>
        _MTD_MTDaddinitializer - Method
    MTD_MTDaddinitializer - MethodName

    ;; (already declared!) MethodParamTypes_MTDTYPESvoid - MethodParamTypes
    RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer<LineWrapMark>
        _MTD_MTDgettaskcount - Method
    MTD_MTDgettaskcount - MethodName
```

```
45
46      ;; (already declared!) MethodParamTypes_MTDTYPESvoid  - MethodParamTypes
47      RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer_<LineWrapMark>
48          MTD_MTDinitialize  - Method
49      MTD_MTDinitialize     - MethodName
50
51      RT_root_PKG_PKGorgapachecommonslang3math  - Package
52      PKG_PKGorgapachecommonslang3math      - PackageName
53
54      RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils  - Class
55      CLS_CLSnumberutils      - ClassName
56
57      MethodParamTypes_MTDTYPESclassjavalangstring  - MethodParamTypes
58      RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal  - Method
59      MTD_MTDcreatebigdecimal     - MethodName
60
61      MethodParamTypes_MTDTYPESlonglonglong  - MethodParamTypes
62      RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmax  - Method
63      MTD_MTDmax      - MethodName
64
65      MethodParamTypes_MTDTYPESclassd  - MethodParamTypes
66      RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmin  - Method
67      MTD_MTDmin      - MethodName
68
69      MethodParamTypes_MTDTYPESclassjavalangstringbyte  - MethodParamTypes
70      RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte  - Method
71      MTD_MTDtobyte     - MethodName
72
73
74      ;; names used in New API (we don't care object in the goal)
75      ;; (already declared!) RT_root  - APIRoot
76
77      ;; (already declared!) PKG_PKGorgapachecommonslang3builder      - PackageName
78
79      CLS_CLSidkeyren     - ClassName
80
81      ;; (already declared!) MTD_MTDequals      - MethodName
82
83      ;; (already declared!) MTD_MTDhashcode      - MethodName
84
85      ;; (already declared!) PKG_PKGorgapachecommonslang3concurrent     - PackageName
86
87      CLS_CLSmultibackgroundinitializerren      - ClassName
88
89      ;; (already declared!) MTD_MTDaddinitializer      - MethodName
90
91      ;; (already declared!) MTD_MTDgettaskcount      - MethodName
92
93      ;; (already declared!) MTD_MTDinitialize      - MethodName
94
95      ;; (already declared!) PKG_PKGorgapachecommonslang3math     - PackageName
96
97      ;; (already declared!) CLS_CLSnumberutils     - ClassName
98
99      MTD_MTDcreatebigdecimalren      - MethodName
```

```
100
101     MTD_MTDmaxren      - MethodName
102
103     MTD_MTDminren      - MethodName
104
105     MTD_MTDtobyteren       - MethodName
106
107   )
108
109   (:init
110     (notPullingUpMethods)
111     (ContainsPackage RT_root RT_root_PKG_PKGorgapachecommonslang3builder)
112     (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3builder)
113     (ContainsPackage RT_root RT_root_PKG_PKGorgapachecommonslang3concurrent)
114     (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3concurrent)
115     (ContainsPackage RT_root RT_root_PKG_PKGorgapachecommonslang3math)
116     (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3math)
117     ;; package name
118     (HasPackageName      RT_root_PKG_PKGorgapachecommonslang3builder
            PKG_PKGorgapachecommonslang3builder)
119     (ContainsClass RT_root_PKG_PKGorgapachecommonslang3builder
            RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey)
120     (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkey)
121     ;; class name
122     (HasClassName      RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey       CLS_CLSidkey)
123     (ContainsMethod RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
            RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals)
124     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkey MTD_MTDequals
            MethodParamTypes_MTDTYPESclassjavalangobject)
125     (ContainsMethod RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
            RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDhashcode)
126     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkey MTD_MTDhashcode
            MethodParamTypes_MTDTYPESvoid)
127     ;; method name
128     (HasMethodName       RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals
            MTD_MTDequals)
129     (HasMethodParamTypes      RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals
              MethodParamTypes_MTDTYPESclassjavalangobject)
130     (HasMethodModifier      RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals
             Instance_Method)
131     ;; method name
132     (HasMethodName       RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDhashcode
            MTD_MTDhashcode)
133     (HasMethodParamTypes      RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDhashcode
               MethodParamTypes_MTDTYPESvoid)
134     (HasMethodModifier      RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDhashcode
              Instance_Method)
135     ;; package name
136     (HasPackageName      RT_root_PKG_PKGorgapachecommonslang3concurrent
            PKG_PKGorgapachecommonslang3concurrent)
137     (ContainsClass RT_root_PKG_PKGorgapachecommonslang3concurrent
            RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer)
138     (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_cLSmultibackgroundinitializer)
139     ;; class name
```

```
(HasClassName RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer
    CLS_CLSmultibackgroundinitializer)

(ContainsMethod RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer
    RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDaddinitializer)
(SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
    CLS_CLSmultibackgroundinitializer MTD_MTDaddinitializer MethodParamTypes_<LineWrapMark>
    MTDTYPESclassjavalangstringclassorgapachecommonslang3concurrentbackgroundinitializer)
(ContainsMethod RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer
    RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    cLSmultibackgroundinitializer_MTD_MTDgettaskcount)
(SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
    CLS_CLSmultibackgroundinitializer MTD_MTDgettaskcount MethodParamTypes_MTDTYPESvoid)
(ContainsMethod RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer
    RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDinitialize)
(SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
    CLS_CLSmultibackgroundinitializer MTD_MTDinitialize MethodParamTypes_MTDTYPESvoid)
;; method name
(HasMethodName RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDaddinitializer MTD_MTDaddinitializer)
(HasMethodParamTypes RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDaddinitializer MethodParamTypes_<LineWrapMark>
    MTDTYPESclassjavalangstringclassorgapachecommonslang3concurrentbackgroundinitializer)
(HasMethodModifier RT_root_PKG_PKGorgapachecommonslang3concurrent_<LineWrapMark>
    CLS_CLSmultibackgroundinitializer_MTD_MTDaddinitializer Instance_Method)
;; method name
(HasMethodName RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDgettaskcount MTD_MTDgettaskcount)
(HasMethodParamTypes RT_root_PKG_PKGorgapachecommonslang3concurrent_<LineWrapMark>
    CLS_CLSmultibackgroundinitializer_MTD_MTDgettaskcount MethodParamTypes_MTDTYPESvoid)
(HasMethodModifier RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDgettaskcount Instance_Method)
;; method name
(HasMethodName RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDinitialize MTD_MTDinitialize)
(HasMethodParamTypes RT_root_PKG_PKGorgapachecommonslang3concurrent_<LineWrapMark>
    CLS_CLSmultibackgroundinitializer_MTD_MTDinitialize MethodParamTypes_MTDTYPESvoid)
(HasMethodModifier RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_<LineWrapMark>
    CLSmultibackgroundinitializer_MTD_MTDinitialize Instance_Method)
;; package name
(HasPackageName RT_root_PKG_PKGorgapachecommonslang3math PKG_PKGorgapachecommonslang3math
    )
(ContainsClass RT_root_PKG_PKGorgapachecommonslang3math
    RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils)
(SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils)
;; class name
(HasClassName RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils
    CLS_CLSnumberutils)
(ContainsMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils
    RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal)
(SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
    MTD_MTDcreatebigdecimal MethodParamTypes_MTDTYPESclassjavalangstring)
```

```
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_MTDTYPESclassjavalangstringbyte)
188     ;; method name
189     (HasMethodName
            RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal
            MTD_MTDcreatebigdecimal)
190     (HasMethodParamTypes
            RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal
            MethodParamTypes_MTDTYPESclassjavalangstring)
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
            MTD_MTDmax)
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
            MTD_MTDmin)
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
            MTD_MTDtobyte)
202     (HasMethodParamTypes     RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte
               MethodParamTypes_MTDTYPESclassjavalangstringbyte)
203     (HasMethodModifier     RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte
              Static_Method)
204   )
205
206   (:goal (and
207     (notPullingUpMethods)
208     (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3builder)
209     (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3concurrent)
210     (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3math)
211     (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkeyren)
212     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkeyren MTD_MTDequals
             MethodParamTypes_MTDTYPESclassjavalangobject)
213     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkeyren
            MTD_MTDhashcode MethodParamTypes_MTDTYPESvoid)
```

```
214     (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializerren)
215     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializerren MTD_MTDaddinitializer MethodParamTypes_<LineWrapMark>
216         MTDTYPESclassjavalangstringclassorgapachecommonslang3concurrentbackgroundinitializer)
217     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializerren MTD_MTDgettaskcount MethodParamTypes_MTDTYPESvoid)
218     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializerren MTD_MTDinitialize MethodParamTypes_MTDTYPESvoid)
219     (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils)
220     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDcreatebigdecimalren MethodParamTypes_MTDTYPESclassjavalangstring)
221     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils MTD_MTDmaxren
             MethodParamTypes_MTDTYPESlonglonglong)
222     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils MTD_MTDminren
             MethodParamTypes_MTDTYPESclassd)
223     (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDtobyteren MethodParamTypes_MTDTYPESclassjavalangstringbyte)
224     ;;Missing paths (should be removed)
225     (not (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkey))
226     (not (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializer))
227     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDcreatebigdecimal MethodParamTypes_MTDTYPESclassjavalangstring))
228     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDmax MethodParamTypes_MTDTYPESlonglonglong))
229     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDmin MethodParamTypes_MTDTYPESclassd))
230     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDtobyte MethodParamTypes_MTDTYPESclassjavalangstringbyte))
231     ;;Missing paths (should be removed): PART 2
232     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkey
            MTD_MTDequals MethodParamTypes_MTDTYPESclassjavalangobject))
233     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder CLS_CLSidkey
            MTD_MTDhashcode MethodParamTypes_MTDTYPESvoid))
234     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializer MTD_MTDaddinitializer MethodParamTypes_<LineWrapMark>
235         MTDTYPESclassjavalangstringclassorgapachecommonslang3concurrentbackgroundinitializer))
236     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializer MTD_MTDgettaskcount MethodParamTypes_MTDTYPESvoid))
237     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3concurrent
            CLS_CLSmultibackgroundinitializer MTD_MTDinitialize MethodParamTypes_MTDTYPESvoid))
238     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDcreatebigdecimal MethodParamTypes_MTDTYPESclassjavalangstring))
239     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDmax MethodParamTypes_MTDTYPESlonglonglong))
240     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDmin MethodParamTypes_MTDTYPESclassd))
241     (not (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3math CLS_CLSnumberutils
            MTD_MTDtobyte MethodParamTypes_MTDTYPESclassjavalangstringbyte))
242
243     );; end of and
244   )
245 )
```

APPENDIX D. Planning Results of Exp. 3

```
1 Time 155530
2 (RENAMECLASS RT_ROOT RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3CONCURRENT
      PKG_PKGORGAPACHECOMMONSLANG3CONCURRENT
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3CONCURRENT_CLS_CLSMULTIBACKGROUNDINITIALIZER
      CLS_CLSMULTIBACKGROUNDINITIALIZER CLS_CLSMULTIBACKGROUNDINITIALIZERREN)
3 (RENAMECLASS RT_ROOT RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3BUILDER PKG_PKGORGAPACHECOMMONSLANG3BUILDER
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3BUILDER_CLS_CLSIDKEY CLS_CLSIDKEY CLS_CLSIDKEYREN)
4 (RENAMEMETHOD RT_ROOT RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH PKG_PKGORGAPACHECOMMONSLANG3MATH
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS CLS_CLSNUMBERUTILS
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS_MTD_MTDTOBYTE MTD_MTDTOBYTE
      MTD_MTDTOBYTEREN METHODPARAMTYPES_MTDTYPESCLASSJAVALANGSTRINGBYTE)
5 (RENAMEMETHOD RT_ROOT RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH PKG_PKGORGAPACHECOMMONSLANG3MATH
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS CLS_CLSNUMBERUTILS
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS_MTD_MTDMIN MTD_MTDMIN MTD_MTDMINREN
      METHODPARAMTYPES_MTDTYPESCLASSD)
6 (RENAMEMETHOD RT_ROOT RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH PKG_PKGORGAPACHECOMMONSLANG3MATH
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS CLS_CLSNUMBERUTILS
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS_MTD_MTDMAX MTD_MTDMAX MTD_MTDMAXREN
      METHODPARAMTYPES_MTDTYPESLONGLONGLONG)
7 (RENAMEMETHOD RT_ROOT RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH PKG_PKGORGAPACHECOMMONSLANG3MATH
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS CLS_CLSNUMBERUTILS
      RT_ROOT_PKG_PKGORGAPACHECOMMONSLANG3MATH_CLS_CLSNUMBERUTILS_MTD_MTDCREATEBIGDECIMAL
      MTD_MTDCREATEBIGDECIMAL MTD_MTDCREATEBIGDECIMALREN METHODPARAMTYPES_MTDTYPESCLASSJAVALANGSTRING)
```

APPENDIX E. Readable Planning Results of Exp. 3

```
1 rename class : org.apache.commons.lang3.concurrent.MultiBackgroundInitializer --> org.
    apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN
2 rename class : org.apache.commons.lang3.builder.IDKey --> org.apache.commons.lang3.
    builder.IDKey_REN
3 rename method : org.apache.commons.lang3.math.NumberUtils.toByte (class_java.lang.String
    ;byte) --> org.apache.commons.lang3.math.NumberUtils.toByte_ren (class_java.lang.
    String;byte)
4 rename method : org.apache.commons.lang3.math.NumberUtils.min (double[]) --> org.
    apache.commons.lang3.math.NumberUtils.min_ren (double[])
5 rename method : org.apache.commons.lang3.math.NumberUtils.max (long;long;long) -->
    org.apache.commons.lang3.math.NumberUtils.max_ren (long;long;long)
6 rename method : org.apache.commons.lang3.math.NumberUtils.createBigDecimal (class_java.
    lang.String) --> org.apache.commons.lang3.math.NumberUtils.createBigDecimal_ren (
    class_java.lang.String)
```

REFERENCES

[1] R. Keller and U. Hlzle, "Binary component adaptation," in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 1445, pp. 307–329. Springer, 1998.

[2] S. Roock and A. Havenstein, "Refactoring tags for automatic refactoring of framework dependent applications," in *Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering (XP)*, 2002.

[3] I. Forman, M. Conner, S. Danforth, and L. Raper, "Release-to-Release Binary Compatibility in SOM," in *ACM SIGPLAN Notices*, ser. SIGPLAN Notices, pp. 426–438. ACM Press, 1995.

[4] I. Balaban, F. Tip, and R. M. Fuhrer, "Refactoring support for class library migration," in *OOPSLA*, pp. 265–279. ACM, 2005.

[5] A. Warth, M. Stanojevic, and T. D. Millstein, "Statically scoped object adaptation with expanders," in *OOPSLA*, pp. 37–56. ACM, 2006.

[6] W. Yu and C. Pu, "A dynamic two-phase commit protocol for adaptive composite services," *Int. J. Web Service Res.*, vol. 4, no. 1, pp. 80–100, 2007.

[7] J. Henkel and A. Diwan, "Catchup!: capturing and replaying refactorings to support api evolution," in *ICSE*, pp. 274–283. ACM, 2005.

[8] D. Dig, S. Negara, V. Mohindra, and R. E. Johnson, "Reba: refactoring-aware binary adaptation of evolving libraries," in *ICSE*, pp. 441–450. ACM, 2008.

[9] I. Savga, M. Rudolf, and S. Goetz, "Comeback!: a refactoring-based tool for binary-compatible framework upgrade," in *ICSE Companion*, pp. 941–942. ACM, 2008.

[10] G. Antoniol, M. D. Penta, and E. Merlo, "An automatic approach to identify class evolution discontinuities," in *IWPSE*, pp. 31–40. IEEE Computer Society, 2004.

[11] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *ACM SIGPLAN Notices*, ser. SIGPLAN Notices, vol. 35, pp. 166–177. ACM Press, October 2000.

[12] Z. Xing and E. Stroulia, "Refactoring detection based on UMLDiff change-facts queries," in *Working Conference on Reverse Engineering (WCRE)*, pp. 263–274. IEEE Press, 2003.

[13] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Software Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[14] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson, "Automated detection of refactorings in evolving components," in *ECOOP*, ser. Lecture Notes in Computer Science, vol. 4067, pp. 404–428. Springer, 2006.

[15] P. Weigerber and S. Diehl, "Identifying refactorings from source-code changes." in *ASE*, pp. 231–240. IEEE Computer Society, 2006.

[16] S. Kim, K. Pan, and E. J. W. Jr., "When functions change their names: Automatic detection of origin relationships," in *WCRE*, pp. 143–152. IEEE Computer Society, 2005.

[17] A. Loh and M. Kim, "Lsdiff: a program differencing tool to identify systematic structural differences," in *ICSE*, pp. 263–266. ACM, 2010.

[18] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *ICSM*, pp. 1–10. IEEE Computer Society, 2010.

[19] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *SIGSOFT FSE*, pp. 371–372. ACM, 2010.

[20] E. R. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 5–18, 2012.

[21] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys*, vol. 28, no. 4, pp. 154–154, December 1996.

[22] M. Fox and D. Long, "Pddl2. 1: An extension to pddl for expressing temporal planning domains," *J. Artif. Intell. Res.(JAIR)*, vol. 20, pp. 61–124, 2003.

[23] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *OOPSLA Companion*, pp. 815–816. ACM, 2007.

[24] K. Bennett and V. Rajlich, "Software maintenance and evolution: a roadmap," in *ICSE - Future of SE Track*, pp. 73–87, 2000.

[25] M. M. Lehman, "Software's future: Managing evolution," *IEEE Software*, vol. 15, no. 1, pp. 40–44, January/February 1998.

[26] M. Aksit, B. Tekinerdogan, and L. Bergmans, "Achieving adaptability through separation and composition of concerns," *Special Issues in Object-Oriented Programming, dpunkt*, vol. 1, pp. 12–23, 1996.

[27] F. Snchez, J. H. Nez, J. M. Murillo, and E. Pedraza, "Run-time adaptability of synchronization policies in concurrent object oriented languages," in *ECOOP Workshops*, ser. Lecture Notes in Computer Science, vol. 1543, p. 443. Springer, 1998.

[28] C. Canal, J. M. Murillo, and P. Poizat, "Software adaptation," *L'OBJET*, vol. 12, no. 1, pp. 9–31, 2006.

[29] J. Cmara, C. Canal, J. Cubo, and J. M. Murillo, "An aspect-oriented adaptation framework for dynamic component evolution," *Electr. Notes Theor. Comput. Sci.*, vol. 189, pp. 21–34, 2007.

[30] K. S. Lu and C. K. Chang, "Alta: Automatic load-time adaptation technique for refactoring-based evolution of software component," in *COMPSAC*, pp. 203–212. IEEE Computer Society, 2012.

[31] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education Inc., 2003.

[32] J. Hoffmann, "Ff: The fast-forward planning system," *AI Magazine*, vol. 22, no. 3, pp. 57–62, 2001.

[33] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for guis," in *SIGSOFT FSE*, pp. 30–39, 2000.

[34] A. M. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: Reverse engineering of graphical user interfaces for testing," in *WCRE*, pp. 260–269. IEEE Computer Society, 2003.

[35] J. Peer, "Web service composition as ai planning-a survey," *Technical report, University of St. Gallen, , Switzerland*, 2005.

[36] J. Rao and X. Su, "A survey of automated web service composition methods," in *SWSWPC*, ser. Lecture Notes in Computer Science, vol. 3387, pp. 43–54. Springer, 2004.

[37] E. Sirin, B. Parsia, D. Wu, J. A. Hendler, and D. S. Nau, "Htn planning for web service composition using shop2," *J. Web Sem.*, vol. 1, no. 4, pp. 377–396, 2004.

[38] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services," in *WWW*, pp. 77–88, 2002.

[39] S.-C. Oh, D. Lee, and S. R. T. Kumara, "A comparative illustration of ai planning-based web services composition," *SIGecom Exchanges*, vol. 5, no. 5, pp. 1–10, 2006.

[40] R. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," *Artif. Intell.*, vol. 2, no. 3/4, pp. 189–208, 1971.

[41] E. P. D. Pednault, "Adl and the state-transition model of action," *J. Log. Comput.*, vol. 4, no. 5, pp. 467–512, 1994.

[42] H. Jaygarl, K. S. Lu, and C. K. Chang, "Genred: A tool for generating and reducing object-oriented test cases," in *COMPSAC*, pp. 127–136. IEEE, 2010.

[43] J. Hoffmann, "Fast-forward planner homepage." [Online]. Available: http://fai.cs.uni-saarland.de/hoffmann/ff.html. 2001.

[44] C. K. Chang, H. Jiang, H. Ming, and K. Oyama, "Situ: A situation-theoretic approach to context-aware service evolution," *IEEE T. Services Computing*, vol. 2, no. 3, pp. 261–275, 2009.