Université de Montréal

AURA: A Hybrid Approach to Identify Framework Evolution

par
Wei Wu

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Décembre, 2009

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

AURA: A Hybrid Approach to Identify Framework Evolution

présenté par:

Wei Wu

a été évalué par un jury composé des personnes suivantes:

Houari A. Sahraoui,	président-rapporteur
Yann-Gaël Guéhéneuc,	directeur de recherche
Marc Feeley,	membre du jury

Mémoire accepté le: ....

RÉSUMÉ

Les cadriciels et les bibliothèques sont indispensables aux systèmes logiciels d'aujourd'hui. Quand ils évoluent, il est souvent fastidieux et coûteux pour les développeurs de faire la mise à jour de leur code.

Par conséquent, des approches ont été proposées pour aider les développeurs à migrer leur code. Généralement, ces approches ne peuvent identifier automatiquement les règles de modification une-remplacée-par-plusieurs méthodes et plusieurs-remplacées-par-une méthode. De plus, elles font souvent un compromis entre rappel et précision dans leur résultats en utilisant un ou plusieurs seuils expérimentaux.

Nous présentons AURA (AUtomatic change Rule Assistant), une nouvelle approche hybride qui combine *call dependency analysis* et *text similarity analysis* pour surmonter ces limitations. Nous avons implanté AURA en Java et comparé ses résultats sur cinq cadriciels avec trois approches précédentes par Dagenais et Robillard, M. Kim *et al.*, et Schäfer *et al.*. Les résultats de cette comparaison montrent que, en moyenne, le rappel de AURA est 53,07% plus que celui des autre approches avec une précision similaire (0,10% en moins).

Mots clés: évolution de logiciel, *call dependency analysis*, *text similarity analysis*, étude empirique.

ABSTRACT

Software frameworks and libraries are indispensable to today's software systems. As they evolve, it is often time-consuming for developers to keep their code up-to-date.

Approaches have been proposed to facilitate this. Usually, these approaches cannot automatically identify change rules for one-replaced-by-many and many-replaced-by-one methods, and they trade off recall for higher precision using one or more experimentally-evaluated thresholds.

We introduce AURA (AUtomatic change Rule Assistant), a novel hybrid approach that combines call dependency and text similarity analyses to overcome these limitations. We implement it in a Java system and compare it on five frameworks with three previous approaches by Dagenais and Robillard, M. Kim *et al.*, and Schäfer *et al.* The comparison shows that, on average, the recall of AURA is 53.07% higher while its precision is similar (0.10% lower).

Keywords: software evolution, call dependency analysis, text similarity analysis, empirical study.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AST	Abstract Syntax Tree
AURA	AUtomatic change Rule Assistant
DOM	Document Object Model
FN	False Negative
FP	False Positive
JAXB	Java Architecture for Xml Binding
JDT	Java Development Tool
JFC	Java Foundation Class
LCS	Longest Common Subsequence
LD	Levenshtein Distance
SAX	Simple API for XML
UML	Unified Modeling Language
XML	eXtensible Markup Language
XSD	XML Schema Definition

NOTATION

$A \circlearrowleft B$	A should be replaced by B
$\mathbf{ALLKR}(a)$	Set of Key Replacement methods which are called by $\mathbf{NEW}(a)$
$\mathbf{ALLN}(t,a)$	How many times the Key Replacement method of target method t are called in $\mathbf{NEW}(a)$
$\mathbf{CHCS}(t)$	Set of Co-replacement methods of target method t with Highest Confidence value
$\mathbf{CRMS}(t)$	Candidate Replacement Method Set of target method t
$\mathbf{CV}(t,c)$	Confidence Value of candidate replacement method c to target method t
$\mathbf{HCS}(t)$	Candidate replacement method set of target method t with Highest Confidence value
$\mathbf{KR}(t)$	Key Replacement method of target method t
$\mathbf{KAS}(t)$	Set of Anchors which call Key replacement method of target method t
$m1 \rightarrow m2$	Method $m1$ calls method $m2$
$m1 \nrightarrow m2$	Method $m1$ does not call method $m2$
NEW(a)	Anchor method a in the NEW release of a program
OLD(a)	Anchor method a in the OLD release of a program
S_a	Set of anchors
S_{gcrm}	Set of global candidate replacement methods
S_{tm}	Set of target methods
S_{tmca}	Set of target methods which are called by at least one anchor
S_{tmuca}	Set of target methods which are not called by any anchor
$[a,b]$	A match in which method a is replaced by method b
$\lvert S\rvert$	The cardinality of a set S
$\Delta\mathbf{P}(A,B)$	The difference in precision of two approaches A and B
$\Delta\mathbf{R}(A,B)$	The difference in recall of two approaches A and B

To My Family.

ACKNOWLEDGMENTS

I would like to thank my supervisor Yann-Gaël Guéhéneuc. His expertise, encouragement and support helped me to overcome the difficulties that I met while carrying out this work. I really enjoyed working with him.

I would like to thank Giuliano Antoniol, Miryung Kim, Barthélémy Dagenais and Martin P. Robillard for their valuable advices and comments on this work. I also express my gratitude to Miryung Kim, Barthélémy Dagenais and Thorsten Schäfer for their generosities of sharing their experimental results.

I would like to thank the members of my thesis committee: Houari A. Sahraoui, Yann-Gaël Guéhéneuc and Marc Feeley. I sincerely appreciate their time, comments and advices.

I would like to thank Ptidej team, GEODES Lab., Soccer Lab., and everybody who helped me during my study and research at Université de Montréal. Without their help, I could not have such a wonderful experience here.

Finally, I want to thank my family. Their support and patience are essential to me.

This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

CHAPTER 1

INTRODUCTION: FRAMEWORK EVOLUTION IDENTIFICATION

Software frameworks[1] and libraries are widely used in software development for cost reduction. They evolve constantly to fix bugs and meet new requirements. In theory, the Application Programming Interface (API) of the new release of a framework should be *backward-compatible* with its previous releases, so that programs linked[2] to the framework continue to work with the new release. In practice, the API syntax and semantics change [3, 8, 25]. For example, from JHotDraw 5.2 to 5.3, method `CH.ifa.draw.figures.LineConnection.end()` was replaced by `LineConnection.getEndConnector()`; such change may have direct consequences on a program using the JHotDraw framework, such as compile errors, or indirect consequences, such as runtime errors if invoking a deleted method using reflection.

To prevent backward-compatibility problems, developers may delay or avoid using a new release. Yet, if they want to benefit from new features or security patches, they must evolve their programs. This *evolution process* often requires a lot of effort because developers must dig into the documents and/or source code of the new and previous releases to understand their differences and to make their programs compatible with the new release.

Consequently, many approaches have been developed to ease this evolution process and reduce the developers' effort. Some require that the framework developers do additional work, such as providing explicit change rules with annotations [4], or that they record API updates to the framework. [10, 15, 18].

To reduce the framework developers' involvement, some approaches automatically identify change rules that describe a matching between *target methods*, *i.e.*, methods existing in the old release but not in the new one, and *replacement methods* in the new release [2, 6, 7, 9, 11, 14, 17, 19, 20, 23, 24, 26, 28].

[1] Without loss of generality, we use the term "framework" to mean both frameworks and libraries.
[2] We refer readers to [13] for a discussion on the links between frameworks and programs.

However, framework developers may not be willing to build change rules manually or use specific tools. Also, some previous approaches [6, 24] cannot detect change rules for target methods not used within the previous releases of the framework and program. Some [19] cannot identify replacement methods if the names of the old and new releases are not similar enough. Still, others [6, 14, 19, 20, 24, 28] require context-dependent thresholds which are chosen through experimental evaluations and may not apply in different contexts.

Furthermore, no existing approaches can *automatically* handle one-replaced-by-many ("one-to-many" in the following) or many-replaced-by-one ("many-to-one") change rules and *explicitly* identify target methods that are simply-deleted, *i.e.*, target methods with no replacement methods in the new release. It is important to identify these one-to-many and many-to-one change rules, because they can guide developers towards new functionalities in the new release. Making simply-deleted methods explicit can release developers from searching their replacements manually.

In particular, developers should be provided with as many relevant change rules as possible to save their effort to identify appropriate rules from a potentially very large code base. Thus, an approach should have the maximum recall [5] without decreasing its precision [5]. Indeed, it is easier for a developer to discard an inappropriate change rule among a couple of hundred rules than to identify an appropriate change rule among thousands of possible method pairs.

Consequently, we propose a novel hybrid language- and context-independent approach, AURA (AUtomatic change Rule Assistant), that combines the advantages and overcomes the limitations of previous approaches:

1. It increases recall by combining call dependency and text similarity analyses in a multi-iteration algorithm.

2. It automatically adapts to different frameworks by not using any experimentally-evaluated threshold.

3. It automatically generates one-to-many, many-to-one, and simply-deleted change rules.

Using a detailed evaluation on four medium-size real-world systems, we show that the percentage of one-to-many and many-to-one change rules covers 8.08% of the total number of target methods. Moreover, the results of the evaluation show that the combination of call dependency and text similarity analyses into a multi-iteration algorithm improves, on average, recall by 53.07% in comparison to previous approaches with a slight decrease of 0.10% in precision. In our evaluation, we also apply AURA to Eclipse and compare its results with those of SemDiff developed by Dagenais and Robillard [6]. We show that the approximate precision of AURA is 92.86% while SemDiff's is up to 100.00%.

A paper based on part of this work has been accepted for publication on the 32nd International Conference on Software Engineering (ICSE 2010).

In the remainder of this paper, Chapter 2 presents motivating examples that illustrate the limitations of previous approaches. Chapter 3 discusses related work. Chapter 4 and 5 describe our approach and its implementation while Chapter 6 evaluates it on five real-world systems. Chapter 7 discusses open issues and Chapter 8 concludes this paper and discusses future work.

CHAPTER 2

MOTIVATING EXAMPLES OF FRAMEWORK EVOLUTION IDENTIFICATION

We illustrate the advantages of AURA with the following motivating examples.

2.1 Multi-iteration Algorithm

Let us assume that a developer must adapt a `Client` program from using Eclipse JDT 3.1 to its 3.3 release, as shown in Figure 2.1. The method `Indents.computeIndentLength(...)` was called in 3.1. However this method on longer exists in 3.3. By manual studying their source code, the following change rules can be identified:

(1) `Indents.getChangeIndentEdits(...)`

$\circlearrowleft$ `IndentManipulation.getChangeIndentEdits(...)`

(2) `Indents.computeIndentLength(...)`

$\circlearrowleft$ `IndentManipulation.indexOfIndent(...)`

where $\circlearrowleft$ means "should be replaced with". To save the effort of manual analysis, the developer would like to obtain them by using some automatic approach.

Previous approaches using either text similarity or call dependency analyses could provide the developer with the first change rule but would not readily suggest the second one, because the method signatures are not similar enough and the callers of the methods changed as well. AURA would report the two change rules above.

With its multi-iteration algorithm, AURA detects that `Indents.getChangeIndentEdits(...)` is replaced by `IndentManipulation.getChangeIndentEdits(...)` in the first iteration. Then, in the following iteration, using the first change rule, AURA also reports that `Indents.computeIndentLength(...)` is replaced by `IndentManipulation.indexOfIndent(...)`. Consequently,

```
// Version 3.1
package org.eclipse.jdt.internal.core.dom.rewrite;
public class SourceModifier implements ISourceModifier {
    public ReplaceEdit[ getModifications(String source){
        ...
        return Indents.getChangeIndentEdits(...);
    }
}
package org.eclipse.jdt.internal.core.dom.rewrite;
class Indents {
    void getChangeIndentEdits(...) {
        ...
        int length= Indents.computeIndentLength(...);
        ...
    }
}

// Version 3.3
package org.eclipse.jdt.internal.core.dom.rewrite;
public class SourceModifier implements ISourceModifier {
    public ReplaceEdit[ getModifications(String source){
        ...
        return IndentManipulation(.getChangeIndentEdits(...);
    }
}
package org.eclipse.jdt.core.formatter;
class IndentManipulation {
    void getChangeIndentEdits(...) {
        ...
        int length= this.indexOfIndent(...);
        ...
    }
}
```

Figure 2.1: Example of many iterations

AURA can identify more change rules than previous approaches in a situation where a set of methods are renamed and–or moved.

2.2 One-to-many Change Rules

Let us assume that a developer must adapt a program built on top of JHotDraw 5.2 to its 5.3 release. The program adds some new commands to the framework. For the sake of simplicity, let us use `CutCommand` in Figure 2.2. Syntactically, this command would not compile with the new release because the expected signature of commands has changed. Semantically, release 5.3 introduces an undo–redo mechanism that should be used by the new command if appropriate. Therefore, the developer would expect to obtain the following change rule automatically, which advices the developer to consider making the command undoable.

```
// Version 5.2
protected JMenu createEditMenu( {
    ...
    menu.add(new CutCommand(``Cut'', view()), new MenuShortcut('x'));
    ...
}

// Version 5.3
protected JMenu createEditMenu( {
    ...
    menu.add(new UndoableCommand(
        new CutCommand(``Cut'', this)), new MenuShortcut('x'));
    ...
}
```

Figure 2.2: Example of a one-to-many change rule

(1) `CutCommand.CutCommand(DrawingView...)`

$\circlearrowleft$ `CutCommand.CutCommand(Alignment, DrawingEditor)`

and `UndoableCommand.UndoableCommand(Command)`

Previous approaches, using call dependency or text similarity analyses, would only report a change rule from `CutCommand.CutCommand(DrawingView...)` to `CutCommand.CutCommand(Alignment, DrawingEditor)`, *i.e.*, a rule fixing the syntactic difference. They would not help the developer in spotting the new feature provided by the framework with its new feature of undoable commands.

AURA reports a one-to-many change rule that suggests replacing the target method `CutCommand.CutCommand(DrawingView...)` with calls to the replacement methods `CutCommand.CutCommand(Alignment,DrawingEditor)` and `UndoableCommand.UndoableCommand(Command)`. Figure 2.2 illustrates the new implementation where an `UndoableCommand` now encapsulates `CutCommand`.

2.3 Many-to-one Change Rules

Let us assume that a developer must adapt a program built on top of JEdit 4.1 to its 4.2 release and the program called methods `DirectoryMenu.DirectoryMenu(...)`, `MarkersMenu.MarkersMenu()` and `RecentDirectoriesMenu.RecentDi` `rectoriesMenu()`, which are replaced by `EnhancedMenu.EnhancedMenu(...)` in the release 4.2.

With previous approaches that generate one-to-one change rules, the developer could know that `DirectoryMenu.DirectoryMenu(...)` is replaced by `EnhancedMenu.EnhancedMenu(...)`, but she would need to find the other two methods manually. Some previous approaches might produce erroneous change rules for the other two target methods due to their high textual similarity with other irrelevant methods.

With AURA, the developer will be informed that the three methods are replaced by `EnhancedMenu.EnhancedMenu(...)`, which frees her from manually searching for replacements or relying on incorrect suggestions.

2.4 Simply-deleted Change Rules

Let us assume a developer who uses JFreeChart 0.9.11 and wants to upgrade her code to the 0.9.12 release. She will get a compile error on method `DefaultBoxAndWhiskerDataset.createNumberArray(...)` because this method does not exist anymore. Therefore, she could expect an automatic approach to warn her by providing her with the change rule (∅ is empty set):

```
(1) DefaultBoxAndWhiskerDataset
        .createNumberArray(...)
 ↺  ∅
```

Previous approaches could not help her because they do not generate simply-deleted rules explicitly: they would not inform her whether the method has been simply deleted or if the approach is just unable to find a replacement. Thus, the developers would have to spend some time and effort to find that there is *no* proper replacement method.

AURA explicitly generates the change rule above, saving the developer's efforts clearing the uncertainty of the approach performing correctly or not.

CHAPTER 3

RELATED WORK

In this chapter, we give a survey about related works first, then introduce three of them with more detailed information because we will compare with them in Chapter 6, finally we summarize the limitations of existing works.

3.1 Survey of Framework Evolution Identification Approaches

Several approaches help developers evolve their programs when the frameworks that they use change. We studied these approaches and identified eight features. Table 3.1 summarizes the different approaches according to their features and highlight the advantages of AURA. In the following, we further define and discuss the different features and approaches.

3.1.1 Capturing API Updates

Existing approaches of capturing API-level changes either require the framework developers' efforts by manually specifying the change rules or by requiring them to use a particular IDE to automatically record the refactorings performed. Chow and Notkin [4] presented a method that requires the framework developers to provide change rules with the new releases. CatchUP! [15] and JBuilder [18] record the refactoring operations in one release and replay them in another. MolhadoRef [10] also employs a record-and-replay technique for handling API-level changes in merging program versions. These approaches are able to provide accurate change rules because of the framework developers' involvement, which might not always be available.

3.1.2 Matching Techniques

Previous approaches use different code matching techniques to find change rules between old and new releases. Dagenais and Robillard developed SemDiff [6], which suggests adaptation to clients by analyzing how a framework adapts to its own changes. Schäfer *et al.* [24] mined framework-usage change rules from already ported instantiations. These two previous approaches compute support and confidence values on call dependency analysis. Godfrey and Zou [14] presented a semi-automatic hybrid approach to perform origin analysis using text similarity, metrics, and call dependency analyses. S. Kim *et al.* [20] automated Godfrey and Zou's approach. Diff-CatchUp developed by Xing and Stroulia [29] analyses textual and structural similarities of Unified Modeling Language (UML) logical design models to recognize API changes. M. Kim *et al.*'s [19] approach leveraged systematic renaming patterns to match old APIs to new APIs.

3.1.3 Many-to-one and one-to-many

Godfrey and Zou [14] detected three cases of merging (Clone Elimination, Service Consolidation, Pipeline Contraction) and three cases of splitting (Clone Introduction, Service Extraction, Pipeline Expansion). We extend merging/splitting to many-to-one/one-to-many change rules. The difference between merging/splitting and many-to-one/one-to-many change rules is that the former is limited to cases defined by Godfrey and Zou [14], while the latter includes any case, *e.g.*, new functionality. SemDiff [6] and Diff-Catch-Up [29] are able to report many-to-one and one-to-many change rules but are semi-automatic, *i.e.*, developers must manually select correct replacements from a provided candidate list. M. Kim *et al.*'s approach [19] automatically reports many-to-one rules.

3.1.4 Simply Deleted

Simply-deleted target methods have no replacement methods in the new release. Semi-automatic approaches [6, 14, 29] and those that require framework developers' involvement [4, 10, 15] are able to report simply-deleted rules. Automatic approaches [19, 20, 24] do not report this type of change rule explicitly in their results.

3.1.5 Automatic and Thresholds

All automatic approaches [19, 20, 24], except record-and-replay ones, use thresholds to keep a balance between precision and recall [5]. Typically, they use experimentally-evaluated thresholds to filter out candidate replacement methods, thus potentially increasing precision but decreasing recall.

3.1.6 Types of Changes

Schäfer *et al.* [24] classified changes between old and new releases into 12 change patterns. We summarize them into three categories of change rules: (1) method rules: all the targets and replacements of a change rule are methods; (2) field rules: all the targets and replacements of a change rule can be methods or fields; (3) inheritance rules: the inheritance relation changes. We report the types of changes found by each approach and compare the results in Section 6.

3.2 Three Benchmark Approaches

In the following section, we illustrate in detail three existing approaches [6, 19, 24], whose results will be compared against those of AURA in Chapter 6. The structure of the illustration is organized in five parts:

1. Contributions
2. Main Matching Techniques
3. Algorithm
4. Evaluation
5. Conclusion

3.2.1 SemDiff

Dagenais and Robillard developed an recommendation system SemDiff [6] based on call dependency analysis to suggest adaptations to client programs by analyzing how a program adapts to is own changes.

3.2.1.1 Contributions

SemDiff brings three contributions: (1) a technique to automatically recommend adaptive changes of non-trivial framework API evolution (users may need to manually choose the right change from the recommendations), (2) the architecture of a complete system to track framework evolution and to infer non-trivial changes, and (3) an experiment to evaluate their result on Eclipse.

3.2.1.2 Main Matching Techniques

The main matching technique of SemDiff is call dependency analysis. They differentiate the outgoing calls of the same methods of two releases of a program. First, for each deleted outgoing call, they build a change set of added outgoing calls. Then, they use *Confidence Value* to rank the methods in the change set to recommend adaptive changes. For a method *m* deleted from the old release and its potential replacement in the new release *n*: *Confidence Value* is defined as:

$$\begin{aligned}
\textbf{Rem}(m) &= \{x \mid x \text{ is a method that removed a call to } m\} \\
\textbf{Add}(m) &= \{x \mid x \text{ is a method that added a call to } m\} \\
\textbf{Callees}(m) &= \{x \mid m \text{ calls } x\} \\
\textbf{Callers}(m) &= \{x \mid x \text{ calls } m\} \\
\textbf{Potential}(m) &= \bigcup_{x \in \textbf{Rem}(m)} \textbf{Callees}(x) \\
\textbf{Support}(m,n) &= |\textbf{Rem}(m) \cap \textbf{Add}(n)| \\
\textbf{Confidence}(m,n) &= \frac{\textbf{Support}(m,n)}{Max(\bigcup_{c \in \textbf{Potential}(m)} \textbf{Support}(m,c))}
\end{aligned}$$

3.2.1.3 Algorithm

The main steps of the algorithm of SemDiff are:

Step 1: Differentiate the outgoing calls of the same methods existing in both old and new releases of a program. For each deleted outgoing call m, build a potential replacement set P_m.

Step 2: Rank the methods in P_m by their Confidence Values.

Step 3: Filter P_m. To reduce the percentage of false positives, they remove the elements in P_m whose Confidence Values are lower than an experimentally-determined threshold. They found that the value of 0.6 gives the best results.

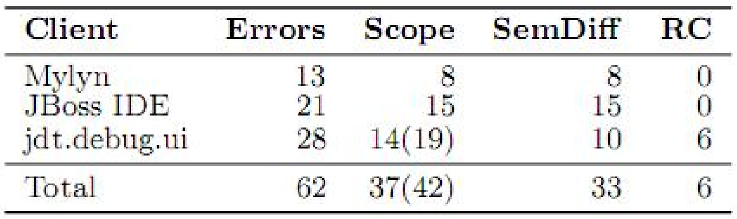
Step 4: Remove spurious calls. An outgoing call $m1$ removed from one method in the new release might be added to another method. In this case, there is no need to find a replacement for $m1$, because it is still used. They defined such kind of calls as spurious calls. They removed all methods of spurious calls in their results.

Because the input of SemDiff is the change sets of software repositories, some parts of their algorithm, such as change chain detection and partial program analysis, are only related to this special implementation. For more information on those parts, please refer to their paper [6].

3.2.1.4 Evaluation

They use the number of Errors in Scope and the number of Errors Solved by SemDiff and Refactoring Crawler (RC) [9] to evaluate their result. Errors in Scope are the unsolved and deprecated method calls while compiling client programs of the old release with the new release of a framework. Errors Solved are the compile errors that SemDiff or RC help solve by providing the correct recommendations. If the correct replacement method of a broken method call is one of the top three recommendations of SemDiff, they consider that it is a relevant recommendation.

They choose two modules of Eclipse Java Development Tool (JDT), `org.eclipse.jdt.core` and `org.eclipse.jdt.ui`, from version 3.1 to 3.3 as target framework systems; and Mylyn version 0.5 to 2.0, JBossIDE version 1.5 to 2.0[1], and `jdt.debug.ui` version 3.1 to 3.3, as client programs. The evaluation result is shown in Figure 3.1, which is Table 2 in their paper [6].



Client	Errors	Scope	SemDiff	RC
Mylyn	13	8	8	0
JBoss IDE	21	15	15	0
jdt.debug.ui	28	14(19)	10	6
Total	62	37(42)	33	6

Figure 3.1: Evaluation Result of SemDiff

In the result of `jdt.bebug.ui`, there are two numbers in Errors in Scope column, because within 19 Errors in Scope, five of them are replaced by the methods outside the target systems. The five broken methods are excluded in the evaluation result. On average, SemDiff provides relevant recommendations for 89% of broken method calls of their target systems.

3.2.1.5 Conclusion

SemDiff suggests adaptations to client programs to the new release of a framework by analyzing how the framework adapts to its own changes. The evaluation on Eclipse

[1]Confirmed by Dagenais that it was version 1.5 to 2.0, not 1.1-1.5.

JDT 3.1 to 3.3 shows that SemDiff can recommend more correct replacement to repair broken client programs than Refactoring Crawler.

3.2.2 Approach of Kim et al.

Kim *et al.* (KIM) developed an approach [19] based on text similarity analysis, which automatically detects structural changes between two releases of a program and presents them as first-order relational logic rules.

3.2.2.1 Contributions

The approach of KIM brings two contributions. First, it automatically infers possible changes at or above method level, then uses them to generate matches, *i.e.*, method level change rules. Second, it summarizes its result in a form of first-order relational logic rules. For example, if class `A` in the old release of a program is renamed to `B` and method `m()` of `A` is moved to class `C` in the new release. Their approach will generate a change rule such as:

```
for all x in A.*(*)
    except{A.m()}
    classReplace(x,A,B)
```

3.2.2.2 Main Matching Techniques

The matching technique that KIM used in their work is text similarity analysis. Given two releases of a program O and N, first, they extracted their method signature sets M_o and M_n respectively. Then, for each method m in M_o, they used tokenized Longest Common Subsequence (LCS) to select the most similar replacement method in M_n by breaking them into tokens starting with uppercase letters.

3.2.2.3 Algorithm

3.2.2.3.1 Definitions and Predicates: The algorithm of the approach of KIM is a rule-based inference process. The following definitions and predicates are used to illustrate it:

Change Rule: a quantifier and a scope to which a transformation applies.

```
for all x in chart.*Plot.get*Range()
    except {chart.MarkerPlot.getVerticalRange}
    argAppend(x, [ValueAxis]))
```

Quantifier: an element of the *scope*.

Scope: a set of source code elements, such as packages, classes, or methods. A *scope* can be presented as $scope - exceptions$, where *exceptions* is a subset of the *scope*.

Transformation: nine types of transformation were defined. Please see Table 3.2.

Candidate Rule: similar to *change rule*. The only difference between them is that a *candidate rule* may include one or more transformations, while a *change rule* only has a single transformation.

Match: a method-level change rule. If method $a()$ is replaced by method $b()$, the *match* is presented as $[a,b]$. Many *matches* can be covered by one *change rule*.

Given a domain $D = M_o - M_n$, a codomain $C = M_n$, a set of candidate rules CR, R a subset of CR, M a set of matches covered by R, an exception threshold $0 \leq \varepsilon \leq 1$ and a candidate rule $r \in CR$: for all x in S, $t_1(x) \wedge \cdots \wedge t_i(x)$, where S is a *scope* and $t_i(x)$ is a transformation applied on x:

$$\begin{aligned}
\textbf{match}(r,[a,b]) &= a \in S \ \wedge \ t_1(\cdots t_i(a)) = b \ \wedge \ r \text{ covers } [a,b] \\
\textbf{conflict}([a,b],[a',b']) &= a \equiv a' \ \wedge \ b \neq b' \\
\textbf{positive}(r,[a,b]) &= \textbf{match}(r,[a,b]) \ \wedge \ [a,b] \in \{D \times C\} \ \wedge \\
&\quad \neg\textbf{conflict}([a,b],[a',b']) \mid \forall \ [a',b'] \in M \\
\textbf{negative}(r,[a,b]) &= \neg\textbf{positive}(r,[a,b]) \\
P &= \{[a,b] \mid \textbf{positive}(r,[a,b])\} \\
E &= \{[a,b] \mid \textbf{negative}(r,[a,b])\} \\
\textbf{valid}(r) &= \frac{|P|}{|P| \ + \ |E|} > (1-\varepsilon) \\
\textbf{selected}(r) &= r \mid \forall r \in CR, \ \textbf{valid}(r) \ \wedge \ r \text{ maximizes } |M \cup P|
\end{aligned}$$

Their algorithm includes four procedures: (1) generating seed matches, (2) generating candidate rules (3) filtering candidate rules, and (4) generating change rules.

3.2.2.3.2 Generating Seed Matches: First, for each method $m \in M_o - M_n$, KIM find the most similar substitute $s \in M_n - M_o$ by computing tokenized LCS. Then, they use an empirical threshold γ to select seed matches, *i.e.*, the tokenized LCS between m and s must be greater than γ. They experimentally determined that γ in the range of 0.65-0.70 (65%-70% tokens are the same) gives good result. Seed matches are not necessarily correct matches. The Procedure 3 will filter them.

3.2.2.3.3 Generating Candidate Rules: For each seed match $[a,b]$, KIM build a set of candidate rules CR. First, they find the set of transformations $T = \{t_1, \cdots, t_i\}$ where $t_1(\cdots t_i(a)) = b$. Second, they generate the token set of the signature TK of a in which each token starts with a uppercase letter. Third, they replace all tokens in each subset of TK with wild-card operator $*$. Fourth, they create candidate scope expressions by replacing the corresponding tokens in TK with each wild-card-operatorized subset. Therefore, the set of candidate scope expressions CSE of a includes 2^n elements, where

n is the number of tokens of a. Last, they generate the set of candidate rules CR by combining each candidate scope expression with each subset of T. Each candidate rule is a generalization of a seed match.

An example is shown in Talbe 3.3.

3.2.2.3.4 Filtering Candidate Rules: In this procedure, KIM filter CR to select a subset that covers the largest number of matches. The selected candidate rules may add a limited number of exceptions to their scope.

First, R the selected candidate rule set and M the match set, that R covers, are set to $\emptyset$. Then, they iterate the following operations: $\forall\ r \mid \textbf{selected}(r)$, they update $CR := CR - \{r\}$, $R := R \cup \{(r,P,E)\}$ and $M := M \cup P$. Here, P and E are the set of positive and negative matches covered by r respectively. If no $r \in CR$ adds more matches to M, the iteration terminates.

3.2.2.3.5 Generating Change Rules: The difference between a candidate rule and a change rule is that the former may have more than one transformation while the latter only has a single one. To convert selected candidate rules to change rules, for each type of transformations t, KIM combine the scope expressions of all selected candidate rules that contain t, as the scope expression of the new change rule. They also add the negative matches of the selected candidate change rules to the exception of the new change rule, if they do not hold for t.

3.2.2.4 Evaluation

3.2.2.4.1 Indicator: KIM use recall and precision [5] to evaluate the result of their approach. Because the set *relavant rules* is a priori unknow in Equation 6.1, they build a set of correct matches C to simulate it, where C is defined as:

$$C \;=\; C_{kim} \cup C_{other}$$

where C_{kim} and C_{other} are the manually-confirmed correct match sets found by their approach and that by others.

Consequently, in the evaluation of their approach, the recall and precision are defined as:

$$\begin{aligned} Recall &= \frac{|M \cap C|}{|C|} \\ Precision &= \frac{|M \cap C|}{|M|} \end{aligned}$$

where M is the match set found by their approach.

Because they present the result of their approach in two forms (change rules and matches), they also use $M/R\ ratio$ as an indicator to present the conciseness of their change rules. It is defined as:

$$M/R\ ratio = \frac{|M|}{|Rules|}$$

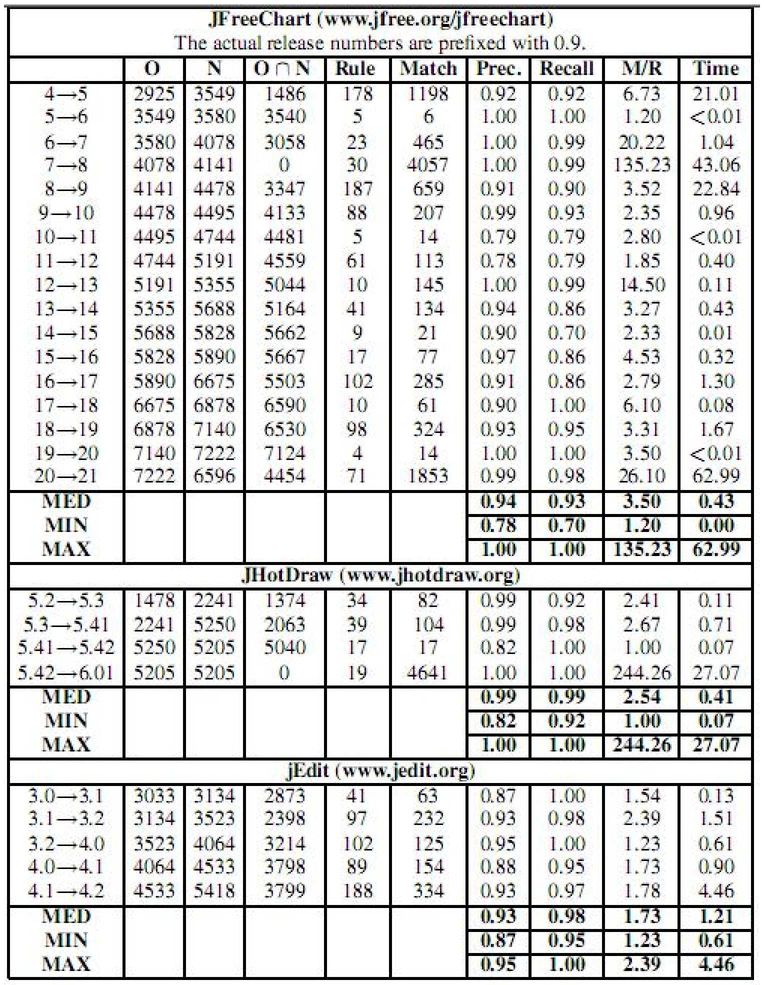
where $Rules$ is the set of change rules that their approach find. Higher $M/R\ ratio$ means that one change rule can cover more matches on average, *i.e.*, the result is more concise.

3.2.2.4.2 Result: They analyzed several versions of three open source projects: JFreeChart, JHotDraw, and JEdit with $\gamma = 0.7$ and $\varepsilon = 0.34$. The results are shown in Figure 3.2, which is Table 3 in their paper [19]:

For all the three projects, the top 20% of the change rules covers over 55% of the matches, and the top 40% of the change rules covers over 70% of the matches.

3.2.2.4.3 Comparison with Other Approaches: KIM compare their approach with the approaches of Xing and Stroulia (XS) [27], Weißgerber and Diehl (WD) [26]. They also choose an origin analysis tool developped by S. Kim *et al.* (KPW) [20] as another reference approach. KIM use $\gamma = 0.65$ and $\varepsilon = 0.34$ to analyze the target projects in the comparison.

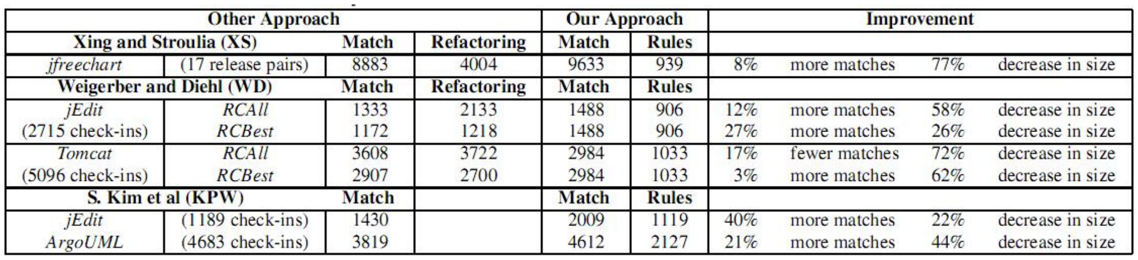
First, they compare the total numbers of matches detected by those approaches and



JFreeChart (www.jfree.org/jfreechart) The actual release numbers are prefixed with 0.9.									
	O	N	O ∩ N	Rule	Match	Prec.	Recall	M/R	Time
4→5	2925	3549	1486	178	1198	0.92	0.92	6.73	21.01
5→6	3549	3580	3540	5	6	1.00	1.00	1.20	<0.01
6→7	3580	4078	3058	23	465	1.00	0.99	20.22	1.04
7→8	4078	4141	0	30	4057	1.00	0.99	135.23	43.06
8→9	4141	4478	3347	187	659	0.91	0.90	3.52	22.84
9→10	4478	4495	4133	88	207	0.99	0.93	2.35	0.96
10→11	4495	4744	4481	5	14	0.79	0.79	2.80	<0.01
11→12	4744	5191	4559	61	113	0.78	0.79	1.85	0.40
12→13	5191	5355	5044	10	145	1.00	0.99	14.50	0.11
13→14	5355	5688	5164	41	134	0.94	0.86	3.27	0.43
14→15	5688	5828	5662	9	21	0.90	0.70	2.33	0.01
15→16	5828	5890	5667	17	77	0.97	0.86	4.53	0.32
16→17	5890	6675	5503	102	285	0.91	0.86	2.79	1.30
17→18	6675	6878	6590	10	61	0.90	1.00	6.10	0.08
18→19	6878	7140	6530	98	324	0.93	0.95	3.31	1.67
19→20	7140	7222	7124	4	14	1.00	1.00	3.50	<0.01
20→21	7222	6596	4454	71	1853	0.99	0.98	26.10	62.99
MED						**0.94**	**0.93**	**3.50**	**0.43**
MIN						**0.78**	**0.70**	**1.20**	**0.00**
MAX						**1.00**	**1.00**	**135.23**	**62.99**
JHotDraw (www.jhotdraw.org)									
5.2→5.3	1478	2241	1374	34	82	0.99	0.92	2.41	0.11
5.3→5.41	2241	5250	2063	39	104	0.99	0.98	2.67	0.71
5.41→5.42	5250	5205	5040	17	17	0.82	1.00	1.00	0.07
5.42→6.01	5205	5205	0	19	4641	1.00	1.00	244.26	27.07
MED						**0.99**	**0.99**	**2.54**	**0.41**
MIN						**0.82**	**0.92**	**1.00**	**0.07**
MAX						**1.00**	**1.00**	**244.26**	**27.07**
jEdit (www.jedit.org)									
3.0→3.1	3033	3134	2873	41	63	0.87	1.00	1.54	0.13
3.1→3.2	3134	3523	2398	97	232	0.93	0.98	2.39	1.51
3.2→4.0	3523	4064	3214	102	125	0.95	1.00	1.23	0.61
4.0→4.1	4064	4533	3798	89	154	0.88	0.95	1.73	0.90
4.1→4.2	4533	5418	3799	188	334	0.93	0.97	1.78	4.46
MED						**0.93**	**0.98**	**1.73**	**1.21**
MIN						**0.87**	**0.95**	**1.23**	**0.61**
MAX						**0.95**	**1.00**	**2.39**	**4.46**

Figure 3.2: Evaluation Result of the Approach of Kim *et al.*

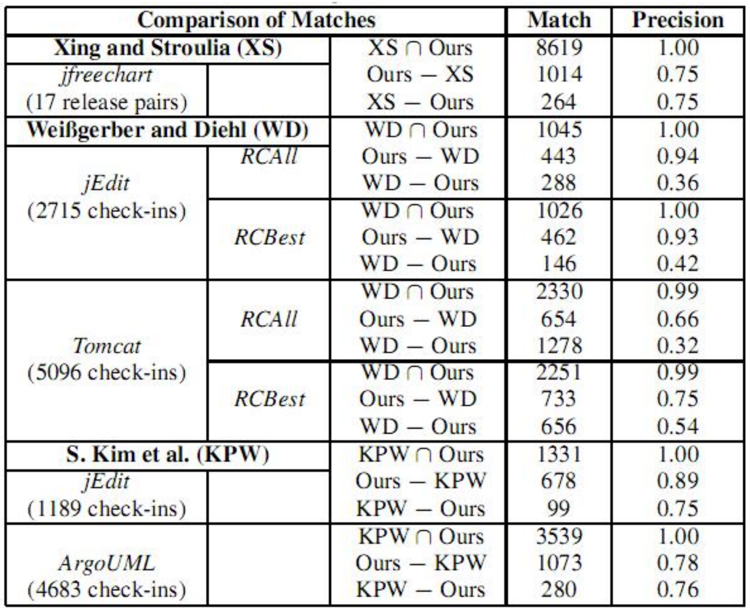
compute the improvement of the approach of KIM. Because they also present their result in first-order relational logic change rules, KIM compute the decrease in size of the presentation of change rules as well. The comparison result is shown in Figure 3.3, which is Table 4 in their paper [19].



Other Approach				Our Approach		Improvement			
Xing and Stroulia (XS)		**Match**	**Refactoring**	**Match**	**Rules**				
jfreechart	(17 release pairs)	8883	4004	9633	939	8%	more matches	77%	decrease in size
Weigerber and Diehl (WD)		**Match**	**Refactoring**	**Match**	**Rules**				
jEdit	*RCAll*	1333	2133	1488	906	12%	more matches	58%	decrease in size
(2715 check-ins)	*RCBest*	1172	1218	1488	906	27%	more matches	26%	decrease in size
Tomcat	*RCAll*	3608	3722	2984	1033	17%	fewer matches	72%	decrease in size
(5096 check-ins)	*RCBest*	2907	2700	2984	1033	3%	more matches	62%	decrease in size
S. Kim et al (KPW)		**Match**		**Match**	**Rules**				
jEdit	(1189 check-ins)	1430		2009	1119	40%	more matches	22%	decrease in size
ArgoUML	(4683 check-ins)	3819		4612	2127	21%	more matches	44%	decrease in size

Figure 3.3: Comparison of Total Number of Matches

Second, KIM compare the precison of three sets with the other approaches: (1) $KIM \cap Other$, (2) $KIM - Other$, (3) $Other - KIM$. The result is shown in Figure 3.4, which is Table 5 in their paper [19].



Comparison of Matches			Match	Precision
Xing and Stroulia (XS)		XS ∩ Ours	8619	1.00
jfreechart (17 release pairs)		Ours − XS	1014	0.75
		XS − Ours	264	0.75
Weißgerber and Diehl (WD)		WD ∩ Ours	1045	1.00
jEdit (2715 check-ins)	*RCAll*	Ours − WD	443	0.94
		WD − Ours	288	0.36
	RCBest	WD ∩ Ours	1026	1.00
		Ours − WD	462	0.93
		WD − Ours	146	0.42
Tomcat (5096 check-ins)	*RCAll*	WD ∩ Ours	2330	0.99
		Ours − WD	654	0.66
		WD − Ours	1278	0.32
	RCBest	WD ∩ Ours	2251	0.99
		Ours − WD	733	0.75
		WD − Ours	656	0.54
S. Kim et al. (KPW)		KPW ∩ Ours	1331	1.00
jEdit (1189 check-ins)		Ours − KPW	678	0.89
		KPW − Ours	99	0.75
ArgoUML (4683 check-ins)		KPW ∩ Ours	3539	1.00
		Ours − KPW	1073	0.78
		KPW − Ours	280	0.76

Figure 3.4: Comparison of the Precision of Matches

Because some matches generated by the approach of WD are redundant, KIM compare the precision in two ways: *RCAll* (with redundancy) and *RCBest* (without redundancy). For example, between two releases, class A was renamed to B, the approach of WD will generate rename A to B and move all methods of A to B. In the comparison of $RCAll$, all matches are considered, while redundant matches are excluded from the comparison of $RCBest$.

3.2.2.5 Conclusion

The approach of Kim *et al.* automatically detects structural changes of different releases of programs and represent them as a set of first-order relational logic change rules, which can reduce the size of presentation of their result. By comparing their approach with three other approaches, they found that their approach can identify more matches with higher precision.

3.2.3 Approach of Schäfer et al.

Schäfer *et al.* (SCF) developped an approach [24] based on call dependency analysis to mine framework usage change rules from already ported instantiations, *i.e.*, the code that uses the framework, such as its client programs or its test cases.

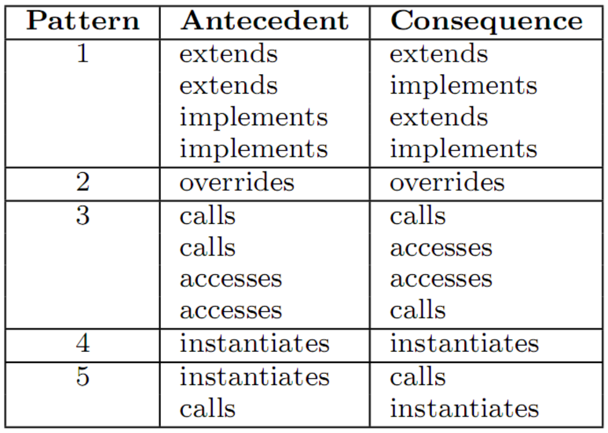
3.2.3.1 Contributions

The contribution of the approach of SCF is that it shows that analyzing instantiation code and association rule mining can find a substantial number of usage changes caused by conceptual changes of the framework rather than refactorings.

3.2.3.2 Main Matching Techniques

The matching technique used by SCF is association rule mining [1], which use *support* and *confidence* (defined later) to measure the credibility of an association of two software elements of two releases of a program. If the software elements are methods, call dependency analysis is needed to compute *support* and *confidence*. Because most of change rules that their approach generates are method level change rules (92% for Struts and 74% for JHotDraw), we can say their main matching technique is call dependency analysis.

To apply association rule mining to detect change rules of software frameworks, they define transaction as a combination of facts from same context and belonging to the same preclassified change patterns. Here, same context means same class or method declaration and the list of the change patterns is show in Figure 3.5, which is Table 2 in their paper [24].



Pattern	Antecedent	Consequence
1	extends	extends
	extends	implements
	implements	extends
	implements	implements
2	overrides	overrides
3	calls	calls
	calls	accesses
	accesses	accesses
	accesses	calls
4	instantiates	instantiates
5	instantiates	calls
	calls	instantiates

Figure 3.5: List of Change Patterns

For example, “method $A.a()$ calls method $C.m()$” cannot be in the same transaction with “method $B.b()$ calls method $C.n()$”, because they are not in the same context; “class A accesses field $C.f$” and “class A implements interface I ”are not allowed in the same transaction either, because they do not belong to the same pattern.

Given a transaction $t \in T$, T is the set of transactions with the same pattern as t’s, f_a and f_c belong to the antecedent and the consequence of t respectively, *support* and *confidence* of f_a and f_c can be computed as:

$$\begin{aligned}
\textbf{Ant}(t, f_a) &= t \text{ has } f_a \text{ as antecedent} \\
\textbf{Con}(t, f_c) &= t \text{ has } f_c \text{ as consequence} \\
\textbf{Support}(f_a, f_c) &= |\{t \mid \textbf{Ant}(t, f_a) \wedge \textbf{Con}(t, f_c)\}| \\
\textbf{Confidence}(f_a, f_c) &= \frac{\textbf{Support}(f_a, f_c)}{|\{t \mid \textbf{Ant}(t, f_a)\}|}
\end{aligned}$$

3.2.3.3 Algorithm

The algorithm of the approach of SCF includes two parts: (1) generating transaction sets, (2) mining change rules.

3.2.3.3.1 Generating transaction set: First, for each same context, *e.g.*, same class and method declarations, in both releases of a program, SCF build possible transactions according to the change patterns listed in Figure 3.5. Then, they remove the transactions whose antecedents and consequences are the same.

3.2.3.3.2 Mining change rules: Using the transaction set generated in the previous part, SCF filter out the change rule candidates $CRC(f_a, f_c)$ which is defined as:

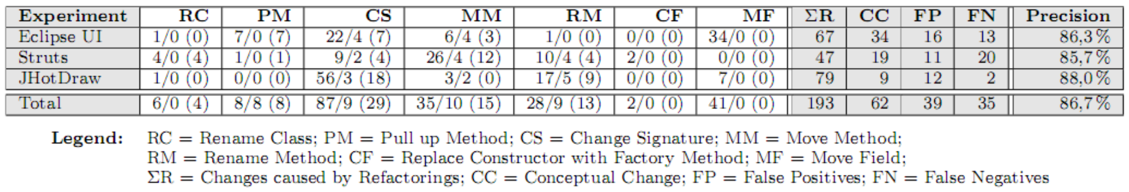
$$CRC(f_a, f_c) = \{f_a, f_c \mid \textbf{Support}(f_a, f_c) > \gamma \wedge \textbf{Confidence}(f_a, f_c) > \varepsilon\}$$

Here, γ and ε are the threshold of support and confidence respectively.

If more than one change rule candidates have the same f_a or f_c, they use tokenized Levenshtein Distance (LD) to find the most similar pair. Therefore, the change rules detected by their approach are one-to-one.

3.2.3.4 Evaluation

With $\gamma = 2$ and $\varepsilon = 0.33$, SCF evaluated their approach on Eclipse UI 2.1.3 to 3.0, JHotDraw 5.2 to 5.3 and Struts 1.1 to 1.2.4, they manually inspected the results and compared them with Refactoring Crawler (RC) [9]. An overview of the results is shown in Figure 3.6, which is the Table 7 in their paper [24]:



Experiment	RC	PM	CS	MM	RM	CF	MF	ΣR	CC	FP	FN	Precision
Eclipse UI	1/0 (0)	7/0 (7)	22/4 (7)	6/4 (3)	1/0 (0)	0/0 (0)	34/0 (0)	67	34	16	13	86,3%
Struts	4/0 (4)	1/0 (1)	9/2 (4)	26/4 (12)	10/4 (4)	2/0 (0)	0/0 (0)	47	19	11	20	85,7%
JHotDraw	1/0 (0)	0/0 (0)	56/3 (18)	3/2 (0)	17/5 (9)	0/0 (0)	7/0 (0)	79	9	12	2	88,0%
Total	6/0 (4)	8/8 (8)	87/9 (29)	35/10 (15)	28/9 (13)	2/0 (0)	41/0 (0)	193	62	39	35	86,7%

Legend: RC = Rename Class; PM = Pull up Method; CS = Change Signature; MM = Move Method;
RM = Rename Method; CF = Replace Constructor with Factory Method; MF = Move Field;
ΣR = Changes caused by Refactorings; CC = Conceptual Change; FP = False Positives; FN = False Negatives

Figure 3.6: Evaluation Result of the Approach of Schafer *et al.*

SCF classify the change rules generated by their approach into three categories: (1) changes caused by refactorings (ΣR), (2) conceptual changes (CC), and (3) false positives (FP). The false negative (FN) column in the tables shows the number of change rules found by RC, but not the approach of SCF.

The columns 2 to 8 in the table represent the change rules caused by refactorings. The first number in those cells means the number of rules caused by corresponding refactoring; the second number is the number of change rules caused by more than one refactoring operations; the third number means the number of change rules detected by RC. Because a change rules of the approach of SCF can be caused by more than one refactoring operations, the sum of the first numbers of columns 2 to 8 of each row might be greater than the corresponding ΣR.

3.2.3.5 Conclusion

In contrary to other approaches, the approach of Schäfer *et al.* analyzes instantiation code, which often exists, to learn how to use the new release of frameworks. Combining with association rule mining technique, their approach can find more change rules than refactoring detecting tools with high precision.

3.3 Summary

AURA overcomes the following limitations of existing approaches:

- Text similarity-based approaches cannot detect replacement methods that do not share similar textual names with their target methods.
- Call dependency-based approaches cannot detect replacement methods for target methods that are not used in frameworks and linked programs.
- No approach can automatically detect many-to-one and one-to-many change rules.
- No approach detects and evaluates simply-deleted methods in their results.
- All automatic approaches except record-and-replay, use thresholds set through experimental evaluations, which may not apply in all contexts.

Approaches	Features									
	FDI	**Main Matching Technique**	**One-to-many Rules**	**Many-to-one Rules**	**Simply-deleted Rules**	**Methods**	**Fields**	**Inheritance Relations**	**Auto-matic**	**Thre-sholds**
Chow *et al.* [4]	Yes	A	No	No	Yes	Yes	No	No	No	No
SemDiff [6]	No	CD	Yes	Yes	Yes	Yes	No	No	No	Yes
Godfrey *et al.* [14]	No	TS, M, and CD	Yes	Yes	Yes	Yes	No	No	No	Yes
CatchUp! [15]	Yes	N/A	No	No	No	Yes	Yes	Yes	Yes	No
M. Kim *et al.* [19]	No	TS	No	Yes	Yes	Yes	No	No	Yes	Yes
S. Kim *et al.* [20]	No	TS, M, and CD	No	No	No	Yes	No	No	Yes	Yes
Schäfer *et al.* [24]	No	CD	No	No	No	Yes	Yes	Yes	Yes	Yes
Diff-CatchUp [29]	No	TS and SS	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
AURA	No	CD and TS	Yes	Yes	Yes	Yes	No	No	Yes	No

Table 3.1: Feature Comparison. (A = Annotation, CD = Call Dependency, FDI = Framework Developer Involvement, M = Metrics, N/A = Not Applicable, TS = Text Similarity, SS = Structural Similarity)

Transformation	Definition
packageReplace(x:Method, f:Text, t:Text)	change x's package name from f to t
classReplace(x:Method, f:Text, t:Text)	change x's class name from f to t
procedureReplace(x:Method, f:Text, t:Text)	change x's procedure name from f to t
returnReplace(x:Method, f:Text, t:Text)	change x's return name from f to t
inputSignatureReplace(x:Method, f:List[Text], t:List[Text])	change x's input argument list name from f to t
argReplace(x:Method, f:Text, t:Text)	change argument type f to t in x's input argument list
argAppend(x:Method, t:List[Text])	append the argument type list t to x's input argument list
argDelete(x:Method, t:Text)	delete every occurrence of type t in x's input argument list
typeReplace(x:Method, f:Text, t:Text)	change every occurrence of type f to t in x

Table 3.2: Transformations

Seed Match	$[pOld.cOld.aMthd(),\ pNew.cNew.aMethod()]$
T	$\{packageReplace,\ classReplace\}$
TK	$\{p,\ Old,\ c,\ Old,\ a,\ Method\}$
CSE	$\{*.*.*(),\ p*.*.*(),\ \cdots,\ pOld.cOld.a*(),\ pOld.cOld.aMethod()\}$
CR	$\{$for all x in $*.*.*()$ $packageReplace(x, Old, New),\ \cdots,$ for all x in $pOld.cOld.aMethod()$ $classReplace(x, Old, New)$ $packageReplace(x, Old, New)\}$

Table 3.3: Candidate Rule Generation

CHAPTER 4

OUR APPROACH: AURA

Our approach is based on call-dependency and text-similarity analyses and a multi-iteration algorithm. We choose call dependency and text similarity as the main matching techniques of our hybrid approach for two reasons: previous approaches using these analyses have good precision [6, 19, 24]; these techniques are compatible with each other because they apply directly to source code.

According to previous approaches [6, 19, 24], we assume that a target method can be deleted or replaced by one or more replacement methods and more than one target method can be replaced by the same replacement method. All replacement methods are taken from the candidate set of all methods existing in the new release of a framework or belonging to other frameworks provided by the same vendor. We do not consider methods from the frameworks of different vendors. For example, when we analyze `org.eclipse.jdt.core`, the methods from other Eclipse plug-ins, such as `org.eclipse.jface`, belong to the candidate replacement method set, but those from Sun Java Foundation Classes (JFC) do not.

We include the methods from the frameworks provided by the same vendor only because framework developers may move methods between their frameworks. This inclusion is a good trade-off between accuracy and performance, because a large candidate set compromises performance but increases precision. After analyzing the results of the four medium-size subject systems in the evaluation, we found that less than 1% of all methods were replaced by those from the frameworks of other vendors.

4.1 Background

Call dependency analysis discovers the calls to the methods of frameworks from the programs using them. These calls reflect the behavior of frameworks more accurately than text similarity, in particular when detecting many-to-one and one-to-many change rules.

To illustrate the call-dependency analysis used in our approach, let us define an *anchor* as either (1) a pair of methods with the same signature (including return type, declaring module in which the methods are defined, name, and parameter lists) that exist in both the old and new releases of the framework or (2) a pair of methods already identified as target and replacement methods. We also define two predicates for an anchor a:

$$\begin{aligned}\mathbf{OLD}(a) &= \text{the method of } a \text{ in the old release}\\ \mathbf{NEW}(a) &= \text{the method of } a \text{ in the new release}\end{aligned}$$

In the following, we note $m1 \rightarrow m2$ if a method $m1$ calls a method $m2$ ($\nrightarrow$ if it does not). We compute the confidence value (CV) for a given target method t and its candidate replacement method c as:

$$\mathbf{CV}(t,c) = \frac{\mathbf{A}(t,c)}{\mathbf{A}(t)}, \text{ with:}$$

$$\begin{aligned}\mathbf{A}(t) &= |\{\, a \mid a \text{ is an anchor} \wedge \mathbf{OLD}(a) \rightarrow t \,\}|\\ \mathbf{A}(t,c) &= |\{\, a \mid a \in \mathbf{A}(t) \wedge \mathbf{NEW}(a) \rightarrow c \,\}|\end{aligned}$$

The confidence value represents the call-dependency similarity of a target method and its candidate replacement methods.

To compute the text similarity of two methods, we tokenize each method signature as proposed by Lawrie *et al.* [21] by splitting them at upper-case letters and other legal characters (except lower case letter and numbers), for example '_' and '$' in Java. Based on the tokenized signatures, our text similarity algorithm computes the similarity of two methods using first their signatures: (return types, declaring modules in which the methods are defined, names, and parameter list), then their Levenshtein Distance (LD) [22], and finally, their Longest Common Subsequence (LCS) [16]. When we compare the text similarities of two candidate replacement methods to a target method, we first compare their signature-level similarity. If they are different, we do not compute their

LD and LCS. We apply the same strategy to LD and LCS.

We combine LD and LCS to compare the text similarity between two methods, because LD and LCS pertain to two different aspects of string comparison: LD is concerned with the difference between strings but is not able to tell if they have something in common, while LCS focuses on their common part but is not able to tell how different they are.

We use the following example to demonstrate the text similarity comparison process. Let us assume that we want to find out which one is most similar to target method `int A.ab(int)` among `int A.a(int), int A.abc(int), int A.abcd (int)` and `int B.a(float).`

First, we rule out `int B.a(float)` because only two elements (return value and method name)of its signature are same to those of the target method, while the other three have three same elements (return value, declaring module and parameter).

Second, we compute the LDs between remaining candidate methods and the target methods and rule out `int A.abcd(int)`, because its LD is greater than the other two.

Last, we calculate the LCS between the two candidate methods and we can know the most similar one is `int A.abc(int)`, because its LCS is longer than that of `int A.a(int).`

By combining the three techniques, we can get the sole most similar method. If we used only one of them, we would get at least two candidate methods with the same similarity.

4.2 Algorithm

Using the previous call dependency and text similarity analyses, AURA generates change rules from the old to the new release of a framework in the following steps:

1. Global Data Set Generation: By differentiating the sets of method signatures in the old and new releases, we build the set of target methods, S_{tm}; the set of anchors, S_a; and, the set of global candidate replacement methods, S_{gcrm}, which includes all the methods defined in the new release. The target methods whose change rules were already detected in previous iterations are not included in S_{tm} and whose replacements are excluded from S_{gcrm}, were added to S_a after being detected.

2. Target Methods Classification: Using call-dependency analysis, we divide S_{tm} in:

$$\begin{aligned} S_{tmca} &= \{\ t \mid a \in S_a, \exists\ \mathbf{OLD}(a) \rightarrow t\ \} \\ S_{tmuca} &= \{\ t \mid a \in S_a, \nexists\ \mathbf{OLD}(a) \rightarrow t\ \} \\ \text{with:} &\quad S_{tm} = S_{tmca} \cup S_{tmuca}. \end{aligned}$$

3. Candidate Replacement Method Set Generation: Also, using call-dependency analysis, for each target method t in S_{tmca}, we build the set of corresponding candidate replacement methods in the new release, using the predicate:

$$\begin{aligned} \mathbf{CRMS}(t) = \{\ m \mid\ & m \in S_{gcrm} \land a \in S_a \\ & \land\ \mathbf{OLD}(a) \rightarrow t \\ & \land\ \mathbf{OLD}(a) \nrightarrow m \\ & \land\ \mathbf{NEW}(a) \rightarrow m\ \}. \end{aligned}$$

4. Confidence Value Computation: We compute the confidence value of each candidate replacement method c in $\mathbf{CRMS}(t)$, with respect to its corresponding target method t, with $t \in S_{tmca}$. We then generate change rules for all target methods in S_{tmca} using the confidence values and $|\mathbf{HCS}(t)|$, where $\mathbf{HCS}(t) = \{\ c \mid c \in \mathbf{CRMS}(t), \mathbf{CV}(t,c) = 100\%\ \}$, as follows:

4a. $\forall\ t \mid |\mathbf{HCS}(t)| = 1$: We build the change rule $t \circlearrowleft c$ and add it to S_a (in the form of an anchor). If S_a does not change, we stop iterating and go to the next step, or we go back to Step 1.

4b. $\forall\ t \mid |\mathbf{HCS}(t)| > 1$: The relation between t and its candidate replacement methods is one-to-one or one-to-many. We assign the proper candidate replacement methods using text similarity analysis and the number $\mathbf{N}(m,a,t)$ of times that t and its candidate replacement methods are called in their anchors, in two steps:

4b1. Key-replacement Methods Identification: We use text-similarity to identify key-replacement methods for all t . The key-replacement method $\mathbf{KR}(t)$ to t is the only method that is the most similar to t from the candidate replacement methods whose names are equal to t's or from the methods in $\mathbf{HCS}(t)$.

4b2. Co-replacement Methods Identification: The co-replacement methods to t are chosen from $\mathbf{CHCS}(t)$ using $\mathbf{N}(m,a,t)$ and the support $\mathbf{S}(t,c)$ defined below. A target method can have zero or more co-replacement methods regardless of their textual similarity. We define the $\mathbf{CHCS}(t)$ of co-candidate methods and $\mathbf{KAS}(t)$ of anchors that call $\mathbf{KR}(t)$, and two counters, such as:

$$\begin{aligned}
\mathbf{CHCS}(t) &= \{\ c \mid c \in \mathbf{HCS}(t) \wedge c \text{ is not a key-} \\
&\quad\ \text{replacement method to} \\
&\quad\ \text{any target methods}\ \} \\
\mathbf{KAS}(t) &= \{\ a \mid a \in S_a \wedge \mathbf{NEW}(a) \rightarrow \mathbf{KR}(t)\ \} \\
\mathbf{N}(m,a,t) &= |\{\mathbf{NEW}(a) \rightarrow m \mid a \in \mathbf{KAS}(t)\}| \\
\mathbf{ALLKR}(a) &= \{\ k \mid \text{k is a key-replacement} \\
&\quad\ \wedge\ \mathbf{NEW}(a) \rightarrow k\ \} \\
\mathbf{ALLN}(t,a) &= |\{\ \mathbf{NEW}(a) \rightarrow k \mid a \in \mathbf{KAS}(t) \\
&\quad\ \wedge k \in \mathbf{ALLKR}(a)\ \}|
\end{aligned}$$

From an anchor $a \in \mathbf{KAS}(t)$, we compute the call count of the key-replacement of t: $m = \mathbf{N}(\mathbf{KR}(t), a, t)$, the call count of a candidate method $c \in \mathbf{CHCS}(t)$: $p = \mathbf{N}(c, a, t)$, and the call count of all the key-replacement methods called in a: $q = \mathbf{ALLN}(t, a)$. We compare p with m and q and only keep co-candidate methods meeting the following two conditions:

- $m = p > 1$: c is called more than one time and exactly as many as the number of times that $\mathbf{KR}(t)$ is called. In this case, c has a high possibility to collaborate with $\mathbf{KR}(t)$ in the new release.
- $q <= p \wedge q > 1$: c is called as many as (or more than) the number of times that the key-replacements of all target methods in the same anchor a, and all the key-replacements are called more than once. In this case, c is likely to collaborate with all the key-replacement methods.

Then, we select the co-candidate methods left in $\mathbf{CHCS}(t)$ with the highest support $\mathbf{S}(t, c)$ as the co-replacement methods, where the support is defined as:

$$\begin{aligned} \mathbf{S}(t,c) \quad = \quad & |\{\, m \mid m \in \{\text{all the methods in the new release}\} \\ & \wedge\, m \rightarrow \mathbf{KR}(t) \wedge\, m \rightarrow c \,\}| \end{aligned}$$

For a target method whose replacement methods are detected in this step, if its co-replacement methods set is empty, AURA generates a one-to-one change rule; otherwise it generates a one-to-many rule.

4c. $\forall\, t \mid |\mathbf{HCS}(t)| = 0$: We choose the most similar candidate replacement methods to t from the methods whose name is equal to t's in $\mathbf{CRMS}(t)$ or from all the methods in $\mathbf{CRMS}(t)$. Then, we choose the candidate methods with the highest confidence value as the replacement methods. The rules detected by this step could be one-to-many rules if there is more than one candidate method with same text-similarity and confidence values. In this step, we give text-similarity analysis priority over confidence value, because a confidence value less than 100% indicates a behaviour change in one or more anchors.

5. Text Similarity Only Rule Generation: For each $t \in S_{tmuca}$, we use text similarity to find its replacement methods with the most similar signatures from S_{gcrm}. If there is more than one candidate replacement method, we select one randomly. We could generate one-to-many rules, if there is more than one candidate method with the same text-similarity to a target. But according to our evaluation, the most relevant cases are one-to-one rules.

6. Simply-deleted Method Rule Identification: Finally, we examine the target methods in S_{tmuca}. If the replacement of one of these methods also exists in the old release, we mark the target method as simply-deleted method, *i.e.*, a target method with no replacement method in the new release. We only identify simply-deleted method rules in this step because target methods in S_{tmuca} have never been used or their context of use changed between the old and new releases. Furthermore, their most similar candidate replacement methods are not methods added to the new release. These target methods are most likely to be simply-deleted.

CHAPTER 5

IMPLEMENTATION OF AURA

As described in Chapter 4, the algorithm of AURA applies to most of high-level programming languages, but it is impractical to implement it in all of them and evaluate it on all of them. Java is a popular Object-Oriented programming language and there are lots of open-source projects developed in it, so we implemented AURA in Java with the same name and evaluated on five Java open source projects. The detailed evaluation result will be presented in next chapter. Our implementation includes four modules: AURA Data Model and its Builder, AURA Rule Model and its Builder. The diagram of our implementation is show in Figure 5.1.

The development and deployment of AURA require JDK 1.6 with JAXB [1] (Java Architecture for eXtensible markup language Binding) support and Eclipse 3.5. JAXB is a Java package which allows developers to seamlessly exchange data between Java programs and eXtensible Markup Language (XML) files. With JAXB, developers can generate Java classes form XML Schema Definition (XSD) files, then use these classes to load from or to write to XML files in an easier way than Simple API for XML (SAX) or Document Object Model (DOM). In the following sections, we introduce the functionality and structure of each module.

5.1 AURA Data Model

AURA Data Model stores the information extracted from the source code of the two releases of a program extracted by AURA Data Model Builder, and the intermediate data generated by AURA Rule Builder. Its package diagram is shown in Figure 5.2

`aura.xsd` contains the XSD files used by JAXB to generate Java data classes. Packages with prefix `aura.model.jaxb` are automatically generated by JAXB. Packages with `ex` after `jaxb` define the extended classes in automatically generated pack-

[1] http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

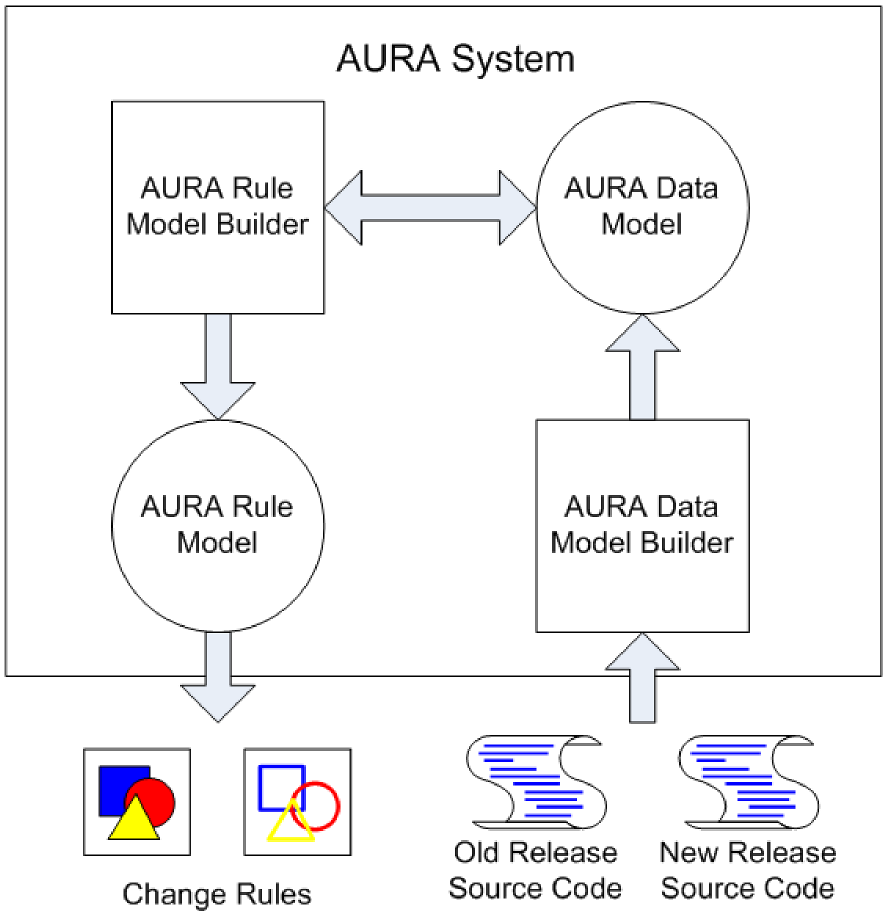


Figure 5.1: Diagram of AURA Implementation

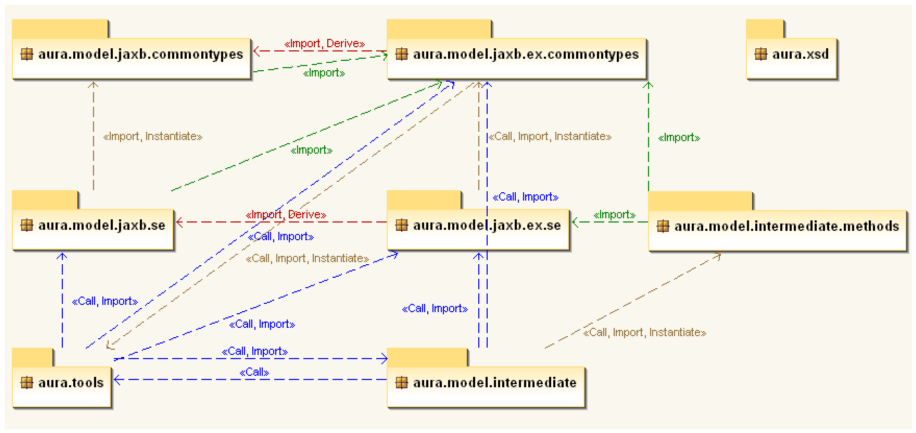


Figure 5.2: Diagram of AURA Data Model

ages. For simplicity, in the following parts of this chapter, we will elide prefix `aura.model.jaxb` or `aura.model.jaxb.ex`.

`commontypes` and `aura.xsd` are shared with AURA Rule Model. `se` stores the information extracted from the source codes of the target program. `intermediate` and `intermediate.methods` hold the temporary information during rule generating, such as anchor, target and candidate method lists. `aura.tools` includes the classes provide auxiliary functions for rule generating, for example, computing tokenized Levenshtein Distance.

5.2 AURA Data Model Builder

The current version of AURA Data Model Builder is an Eclipse plugin. It extracts the information needed by call dependency analysis from the source code of the two releases of a program by parsing the Abstract Syntax Tree (AST) generated by Eclipse JDT. Its package diagram is shown in Figure 5.3.

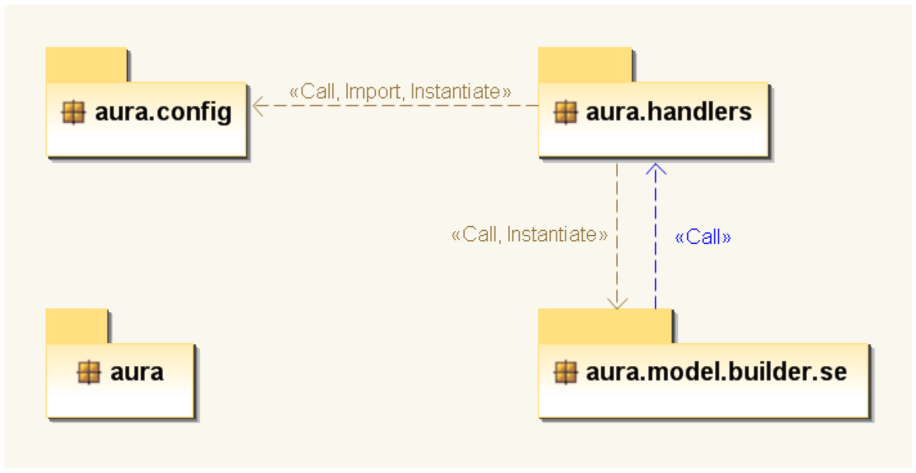


Figure 5.3: Diagram of AURA Data Model Builder

`aura` and `aura.handlers` are required parts of Eclipse plugins. `aura.config` passes the information related to a target program and the path where the change rules are saved to. The classes in `aura.model.builder.se` extract target program information to AURA Data Model. Most of extracting operations are conducted by a Eclipse

AST visitor [12].

5.3 AURA Rule Builder

AURA Rule Builder includes five visitors to IntermediateModel in AURA Data Model. Each one of the visitors implements a corresponding step in the rule generation algorithm described in previous chapter. Its class diagram is shown in Figure 5.4.

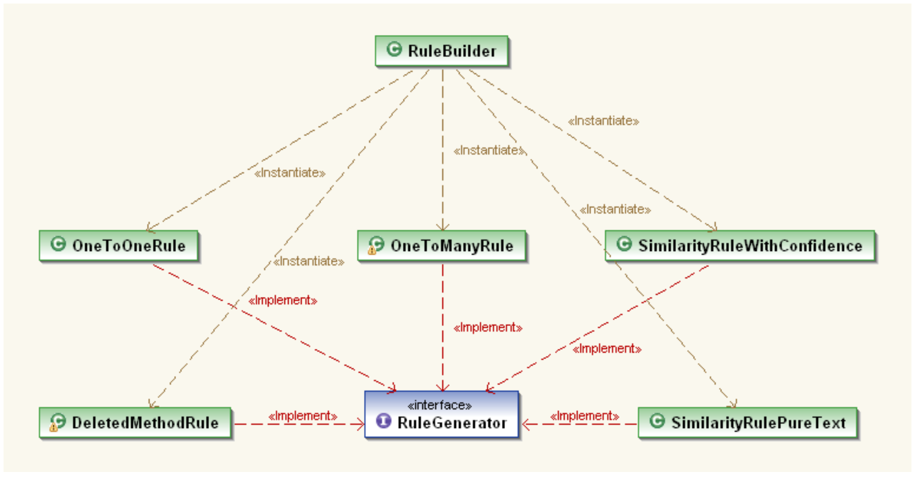


Figure 5.4: Diagram of AURA Rule Builder

5.4 AURA Rule Model

AURA Rule Model stores and presents the change rules detected by the visitors of AURA Rule Builder. Its package diagram is shown in Figure 5.5

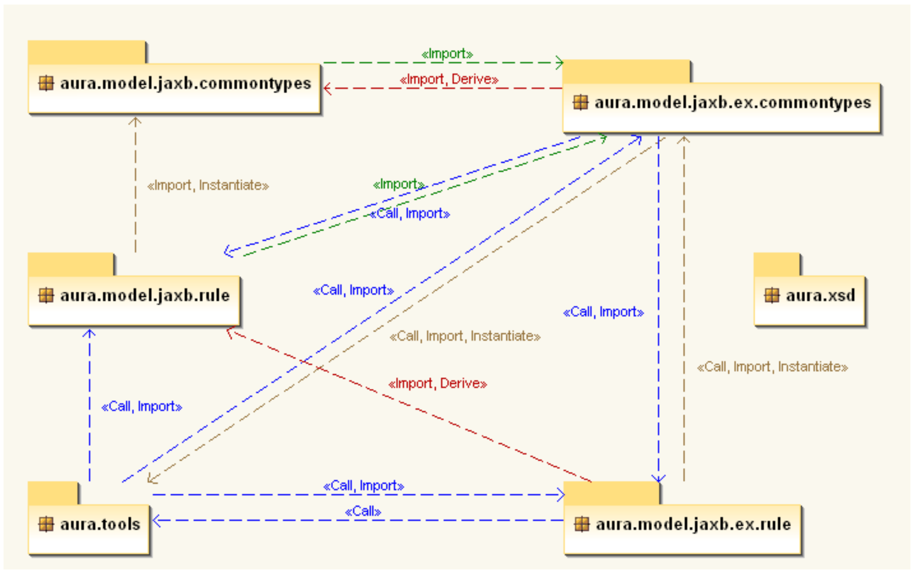


Figure 5.5: Diagram of AURA Rule Model

CHAPTER 6

EVALUATION OF AURA

We now evaluate AURA on several systems by comparing its results to those of previous approaches.

6.1 Evaluation Design

We implemented our approach as a Java program and evaluate it on five open source systems meeting the following conditions: (1) different sizes; (2) developed independently from each other; and, (3) studied in previous work. The last condition reduces the bias in the selection of the subject systems and facilitates the comparison with previous work. Table 6.1 summarizes the five subject systems.

We use the four medium size systems (JFreeChart, JHotDraw, JEdit, and Struts) to compare AURA with the approaches of M. Kim *et al.* [19] and Schäfer *et al.* [24]. We use the large system (`org.eclipse.jdt.core` and `org.eclipse.jdt.ui`) to compare AURA with SemDiff [6].

We reuse the results of the three approaches provided by their authors because it is impractical to reanalyse all the target systems and also to avoid experimenter bias.

We include one-to-many change rules by treating them as one-to-one change rules because previous approaches do not report such rules. We convert many-to-one change rules into as many one-to-one change rules as target methods.

6.2 Hypothesis and Performance Indicators

Our hypothesis is that AURA will find more relevant change rules than previous approaches with comparable precision, *i.e.*, it will have a better recall than and similar precision to those of previous approaches.

We cannot use recall and precision [5] directly to compare the performance of AURA and previous approaches because the set *relevant rules* is *a priori* unknown in:

Subject Systems	Releases	# Methods
JFreeChart	0.9.11	4,751
	0.9.12	5,197
JHotDraw	5.2	1,486
	5.3	2,265
JEdit	4.1	2,773
	4.2	3,547
Struts	1.1	5,973
	1.2.4	6,111
`org.eclipse.jdt.core`	3.1	35,439
`org.eclipse.jdt.ui`	3.3	47,237

Table 6.1: Subject Systems.

$$\begin{aligned}\textbf{Precision} &= \frac{|\{relevant\ rules\} \bigcap \{retrieved\ rules\}|}{|\{retrieved\ rules\}|} \\ \textbf{Recall} &= \frac{|\{relevant\ rules\} \bigcap \{retrieved\ rules\}|}{|\{relevant\ rules\}|}\end{aligned}$$

Therefore, to eliminate the influence of this unknown set, we define the set $\{correct\ rules\}$, which can be obtained by manually inspecting the set $\{retrieved\ rules\}$ as:

$$\begin{aligned}\{correct\ rules\} &= \{relevant\ rules\} \\ &\bigcap\ \{retrieved\ rules\}.\end{aligned}$$

We introduce the differences in precision, $\mathbf{\Delta P}$, and recall, $\mathbf{\Delta R}$, as two functions of the change rules detected by two different approaches, A and B:

$$\begin{aligned}\mathbf{\Delta P}(A,B) &= \frac{\textbf{Precision}_A - \textbf{Precision}_B}{\textbf{Precision}_B} \\ &= \frac{|\{correct\ rules\}_A| \times |\{retrieved\ rules\}_B|}{|\{retrieved\ rules\}_A| \times |\{correct\ rules\}_B|} \\ &\quad -1 \\ \mathbf{\Delta R}(A,B) &= \frac{\textbf{Recall}_A - \textbf{Recall}_B}{\textbf{Recall}_B} \\ &= \frac{|\{correct\ rules\}_A| - |\{correct\ rules\}_B|}{|\{correct\ rules\}_B|}\end{aligned}$$

Using $\mathbf{\Delta P}(A,B)$ and $\mathbf{\Delta R}(A,B)$, we can compare the precision and recall of two approaches and avoid the influence of the unknown set $\{relevant\ rules\}$. We compute $\{correct\ rules\}$ for AURA on four medium-size systems, JFreeChart, JHotDraw, JEdit, and Struts by manual inspection. For previous approaches, we use the data provided by the corresponding authors.

For the two Eclipse plug-ins, `org.eclipse.jdt.core` and `org.eclipse.jdt.ui`, from 3.1 to 3.3, AURA generates more than 4,500 change rules. Thus, it is impractical to validate all these rules manually. We follow Dagenais and Robillard's evaluation method [6]: choose the same three client programs of these plug-ins, *i.e.*, `org.eclipse.jdt.debug.ui`, Mylyn, and JBossIDE; compile them with Eclipse 3.3; use the change rules found by our approach to solve the compile errors in scope *i.e.*, compile errors caused by the methods not existing anymore in release 3.3; and, compute the precision of the change rules that cover these compile errors.

6.3 Comparison on the Medium-size Systems

In Table 6.2, we present the **ΔP** and **ΔR** on each subject system between AURA and M. Kim *et al.*'s [19] and Schäfer *et al.*'s [24] approaches, in column 5 and 6. We then report the average values for each approach in column 7 and 8. In the last three rows, we present the total average values of AURA compared to the two approaches: **ΔR** is 53.07% with a precision of 88.25%, while **ΔP** is -0.10%.

6.3.1 Comparison with M. Kim et al.'s Approach

M. Kim *et al.* [19] present their results in two formats: first-order relational logic rules, for example "alge rules of AURA. Therefore, we use the number of matches from [19] to compare their results with ours.

On average, **ΔP** is -5.66% while **ΔR** is 53.21%. We gain in recall at the small expense of precision.

On JHotDraw from 5.2 to 5.3 and JFreeChart from 0.9.11 to 0.9.12, the **ΔR**s are 19.49% and 75.86% while the **ΔP**s are -6.69% and 3.50%, respectively. These results show that the combination of call-dependency and text-similarity analyses improves re-

Systems	Indicators	AURA	M. Kim *et al.*[19]	ΔR	ΔP	Averages	
						ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	97	81	19.49%	-6.69%	53.21%	-5.66%
	Precision	92.38%	99.00%				
JEdit 4.1-4.2	# Correct rule	356	217 [3]	64.29%	-13.78%		
	Precision	80.18%	93.00%				
JFreeChart 0.9.11-0.9.12	# Correct rule	155	88	75.86%	3.50%		
	Precision	80.73%	78.00%				

Systems	Indicators	AURA	Schäfer *et al.* [24]	ΔR	ΔP	Averages	
						ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	97	88	10.23%	4.98%	52.86%	8.24%
	Precision	92.38%	88.00%				
Struts 1.1-1.2.4	# Correct rule	129	66	95.49%	11.50%		
	Precision	96.56%	85.70%				

Total Average	Precision of AURA	88.25%
	ΔR	53.07%
	ΔP	-0.10%

Table 6.2: Comparison of the results on medium-size systems (with simply-deleted change rules).

Systems		AURA	SemDiff [6]
`org.eclipse.jdt.debug.ui` 3.1 - 3.3	# Errors in Scope	4	
	# Found Rules	4	4
	# Correct Rules	4	4
Mylyn 0.5-2.0	# Errors in Scope	2	
	# Found Rules	2	2
	# Correct Rules	1	2
JBossIDE 1.5-2.0 [4]	# Errors in Scope	8	
	# Found Rules	8	8
	# Correct Rules	8	8
Precision		92.86%	≤ 100.00%

Table 6.3: Evaluation of a sample of change rules on the large system.

call with precision comparable to approaches based on text-similarity analyses. A slight decrease of precision (-6.69%) is acceptable because the recall increases satisfactorily (19.49%).

On JEdit from 4.1 to 4.2, the **ΔR** is 64.29% while **ΔP** is -13.79%. The **ΔP** decrease is twice as much as that of JHostDraw from 5.2 to 5.3. Two factors cause this decrease. First, call-dependency analysis is more sensitive to structural changes than text similarity analysis. In JEdit 4.2, the API remained quite stable but the implementation of the methods changed radically. AURA first uses call dependency analysis that generates irrelevant change rules that could be avoided if it used text similarity analysis directly. Second, AURA does not use any experimentally-evaluated thresholds that would help balancing recall and precision.

6.3.2 Comparison with Schäfer et al.'s Approach

On average, Δ**P** is 8.24% while Δ**R** is 52.86%. AURA has positive Δ**R** and Δ**P** both on JHotDraw from 5.2 to 5.3 and Struts from 1.1 to 1.2.4 in comparison to Schäfer *et al.*'s [24]. On JHotDraw from 5.2 to 5.3, the Δ**R** and Δ**P** are 10.23% and 5.00%, while they are 95.49% and 11.50% on Struts from 1.1 to 1.2.4. Text-similarity analysis is the main contributor to the improvements. In our evaluation, the change rules of 59.05% target methods (62) of JHotDraw from 5.2 to 5.3 are detected by call-dependency analysis, while the number for Struts from 1.1 to 1.2.4 is only 17.04% (23). The change rules for the rest of the target methods are generated by text-similarity analysis.

In Schäfer *et al.*'s results [24], more change rules were identified than by AURA using call-dependency analysis because they also generate other types of change rules that are not in the scope of AURA, such as change rules for fields, inheritance relations, and methods existing in both the old and new releases. AURA only generates change rules for methods that physically disappeared in the new release.

6.4 Comparison on a Large-size System

In Table 6.3, we present the results of AURA and SemDiff [6] to solve the compile errors of three Eclipse 3.1 plug-ins when compiling them against Eclipse 3.3.

In SemDiff [6], correct rules are defined as replacement methods that can be found in the top three recommendations provided by SemDiff. It is easy for developers to choose the right replacement from these three. In our approach, we provide only one recommendation per target method, possibly one-to-one, one-to-many, many-to-one, or simply-deleted change rule. Therefore, to compare the results of AURA with those of SemDiff, we must account for this discrepancy in the way correct rules are counted.

[3] AURA only analyzed the code in packages org.gjt.sp.* and compare the result with corresponding part of M. Kim *et al.*'s work [19], because this package contains the source code for the main functions of JEdit and it is already large enough for manual analysis (444 target methods).

[4] Confirmed by Dagenais, it is 1.5-2.0

If every correct rule was the first recommendation of the top three, SemDiff would have a precision of 100.00%, comparable to the precision of 92.86% of AURA. However, it is also possible that the correct rule was the second or third of the top three. Consequently, for the first recommendation, the precision of SemDiff could be actually less than 100%, thus AURA is competitive with SemDiff.

Because `org.eclipse.jdt.core` and `org.eclipse.jdt.ui` from 3.1 to 3.3 have more than 4500 target methods and SemDiff is a semi-automatic approach, it is impractical to get $\{correct\ rules\}$ manually. Thus, we do not compute **ΔR** between AURA and SemDiff.

6.5 Comparison without Simply-deleted Methods

Previous approaches, such as [19, 24], do not explicitly report simply-deleted change rules in their results. We remove the simply-deleted change rules from AURA results and compare these with the results of previous approaches to assess their influence on precision and recall.

As shown in Table 6.4, **ΔP** is stable and remains similar to that with simply-deleted method rules (0.24% vs -0.10%). **ΔR** decreases from 53.07% to 6.80%. The **ΔR**s of AURA to the two approaches [19, 24] are different. The **ΔR** to Kim *et al.*'s approach [19] decreases to 13.34%, while the **ΔR** to Schäfer *et al.*'s approach [24] drops to -3.02%.

The sharp decrease of **ΔR** has two causes. First, a large number of target methods are deleted from the new releases without replacements. Through manual inspection, we found that, on average, 31.93% of target methods in the change rules generated by AURA are simply deleted from the new releases of the four medium-size systems. For Struts from 1.1 to 1.2.4, this percentage is as high as 57% (77 methods). Second, AURA and Schäfer *et al.*'s approach do not have the same scope, so the value of **ΔR** to their approach decreases dramatically.

Even with this decrease of **ΔR** on Struts from 1.1 to 1.2.4. (-15.14%), AURA still improves recall with similar precision when not considering simply-deleted method rules.

Systems	Indicators	AURA	M. Kim *et al.* [19]	ΔR	ΔP	Averages ΔR	Averages ΔP
JHotDraw 5.2-5.3	# Correct rule	96	81	18.26%	-3.03%	13.34%	-5.01%
	Precision	96.00%	99.00%				
JEdit 4.1-4.2	# Correct rule	247	217 [3]	13.99%	-17.00%		
	Precision	77.19%	93.00%				
JFreeChart 0.9.11-0.9.12	# Correct rule	95	88	7.78%	5.00%		
	Precision	81.90%	78.00%				

Systems	Indicators	AURA	Schäfer *et al.* [24]	ΔR	ΔP	Averages ΔR	Averages ΔP
JHotDraw 5.2-5.3	# Correct rule	96	88	9.09%	9.09%	-3.02%	8.11%
	Precision	96.00%	88.00%				
Struts 1.1-1.2.4	# Correct rule	56	66	-15.14%	7.12%		
	Precision	91.08%	85.70%				

Total Average	Precision of AURA	88.58%
	ΔR	6.08%
	ΔP	0.24%

Table 6.4: Comparison of the results on medium-size systems without simply-deleted change rules

6.6 Performance

Since the analyses of AURA and of the previous approaches were conducted on different hardware and software platforms, the reported performance data are only descriptive and we will not compare them.

The analysis of the four medium-size systems takes less than three minutes on Windows XP SP3 with Intel Core Duo 1.5GHz and 4GB RAM. M. Kim *et al.* [19] report computation times of seven minutes on average while Schäfer *et al.* [24] report less than 30 minutes, but do not specify their software and hardware platforms.

The analyzing Eclipse JDT core and UI 3.1–3.3 takes seven hours on CentOS 5.3 with AMD Opteron Dual-Core 2.4GHz and 16GB RAM. SemDiff [6] took 16 hours on a Pentium D 3.2Ghz with 2GB of RAM running Ubuntu Server 7.04.

6.7 Threats to Validity

We now discuss the threats to validity of our evaluation following the guidelines provided for case study research [30].

Construct validity threats concern the relation between theory and observation; in our context, they are mainly due to errors introduced in the algorithm and the manual validation. We are aware that we could have introduced a bias during the manual validation of the change rules produced by AURA. We did our best to avoid this bias and provide all data on-line for further independent validation[5]. AURA in Step 5 uses a random selection that could also introduce variation in our results. However, these variations should occur very rarely.

Threats to *internal validity* do not affect this particular study, being a systematic comparison of AURA with previous approaches using well-defined measures, $\mathbf{\Delta P}$ and $\mathbf{\Delta R}$.

Conclusion validity threats concern the relation between the treatment and the outcome. We used un-biased systematic measures and the data provided by the authors of previous approaches without any changes other tan those discussed in Section 6. Thus, we believe that no threats to the validity of our conclusion remain.

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to re-implement AURA and replicate its evaluation and comparison. Moreover, all studied systems and data from previous approaches are publicly available or available upon request to their authors. Finally, the raw data on which our study is based are available on the Web[3].

Threats to *external validity* concern the possibility to generalize our findings. We studied five systems of different sizes, belonging to different domains and evaluated by previous approaches. However, we only analyzed Java code; therefore it is possible that AURA would perform differently on other programming languages, like C$\sharp$ or C++. Further validation on a larger set of systems and comparison with other approaches are desirable.

[5]`http://www.ptidej.net/downloads/experiments/prop-icse10b`

CHAPTER 7

DISCUSSION OF AURA

We now discuss the strengths and limitations of AURA.

7.1 Strengths

7.1.1 Higher Recall and Comparable Precision

The evaluation results show that AURA has higher recall than and comparable precision to those of previous approaches. Three factors contribute to the improvement:
1. Combination of call-dependency and text-similarity Analyses: Approaches using call-dependency analysis can only find change rules whose target and replacement methods are called by some anchors. For the five systems that we analyzed, on average, only 33.85% of the change rules are found by call-dependency analysis. Schäfer *et al.* [24] can found more rules, because they also generate other types of change rules besides target method change rules.

Approaches using text-similarity analysis find rules for all target methods but with a higher rate of false positives. In practice, they trade recall for precision using one or many thresholds.

AURA is able to find change rules for more target methods than previous approaches, but with a slight loss of precision. The evaluation results show that the Δ**R** of AURA is 53.07% with about -0.10% lower precision, on average.

2. Multi-iteration algorithm: The multi-iteration algorithm improves both recall and precision. It impacts positively the results in two cases: the first case is illustrated in Section 2; the second case occurs by removing the replacement methods of other already-detected target methods from the candidate replacement method set of a target method. For example, if the candidate set of `m()` is {`a()`,`b()`} and in a previous iteration AURA detected that `a()` is the replacement of `x()`, then AURA removes `a()` from the candidate replacement method set of `m()` and immediately identifies its replacement as `b()`. This second case does not preclude identifying many-to-one change rules in a previous iteration. On the four medium-size systems, the average precision decreases by 2.50% if we a use a one-iteration algorithm, calculated after both call-dependency and text-similarity analyses.

3. Three–Unit text similarity: AURA uses signature-level similarity, LD and LCS, to compute the text similarity of two methods. On the four medium-size systems, the average precision decreases by 3.53%, 2.41%, and 4.51% if we remove each step, respectively.

7.1.2 Many-to-one, One-to-many, Simply-deleted Rules

Previous approaches only automatically generate one-to-one change rules. Some approaches [6, 14] can semi-automatically generate many-to-one and one-to-many rules, but developers must manually analyze the rules to select the appropriate replacement methods. AURA applies a call-dependency analysis first and then uses a text-similarity analysis to overcome this limitation of previous approaches.

None of the previous automatic approaches explicitly reports simply-deleted method change rules. We manually identified that, in the four medium-size systems, 31.93% of target methods in the change rules that AURA generated are simply-deleted. We argue that simply-deleted method rules are as important as other types of change rules because they are a part of the total change rules of a program. They should be identified, evaluated and counted in the precision and recall computation.

7.1.3 Threshold

Existing automatic approaches [19, 20, 24], which do not require framework developers' involvement, depend on experimentally-evaluated thresholds. These thresholds cannot be predicted for a new framework without analyzing it and evaluating the result. We could use the values of the tuned thresholds for some frameworks already analyzed but they might not be applicable.

AURA completely eliminates thresholds and adapts naturally to different frameworks. It could therefore be used immediately by developers without any settings.

7.2 Limitations

AURA cannot detect one-to-many and many-to-one change rules for target methods that no anchor calls. However, it can still find one-to-one rule using text similarity analysis.

Major changes to the internal implementation of anchors compromise the precision of AURA. For example, the precision of AURA for JEdit from 4.1 to 4.2 decreases by 13.78% wrt. M. Kim *et al.*'s [19] because, between the two releases, the API remained quite stable but the implementation of the methods changed radically, thus confusing the first steps of our approach based on call-dependency analysis. This limitation is shared by all call-dependency-based approaches.

AURA only generates change rules for methods. During the evaluation of AURA, we found that some getters are replaced by direct field accesses. Future work includes modifying the definition of change rules to take into account field and type-related changes by analyzing inheritance relations and polymorphism.

CHAPTER 8

CONCLUSION AND FUTURE WORK

We presented AURA, a hybrid approach that combines call-dependency and text-similarity analyses to provide developers with change rules when adapting their programs from one release of a framework to the next.

Our approach offers the following contributions:

1. It increases recall by combining call dependency and text similarity analyses in a multi-iteration algorithm;

2. It automatically adapts to different frameworks by not using any experimentally-evaluated threshold;

3. It reduces developers' efforts by automatically generating one-to-many and many-to-one change rules.

The results of the evaluation of AURA on four medium-size systems and in comparison to previous work showed that the combination of call-dependency and text-similarity analyses into a multi-iteration algorithm improves recall on average by 53.07% with a slight decrease of 0.10% in precision. We also applied AURA on Eclipse and compared its results with those of SemDiff [6] and showed that the approximated precision of AURA is 92.86% while SemDiff's is up to 100.00%.

In future work, we plan to extend our approach in several directions: analyze target systems in other programming languages than Java; add heuristics that generate change rules for types and fields by analyzing inheritance relations and polymorphism; combine AURA with approaches that use other matching techniques; present AURA results in first-order relational logic rules, as introduced by M. Kim *et al.* [19]; perform usability studies to determine the efficacy of AURA.

BIBLIOGRAPHY

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM. ISBN 0-89791-592-5. doi: http://doi.acm.org/10.1145/170035.170072.

[2] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE '04: Proceedings of the Principles of Software Evolution, 7th International Workshop*, pages 31–40. IEEE Computer Society, 2004.

[3] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 265–279, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094832.

[4] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7677-9.

[5] J. Cohen. *Statistical power analysis for the behavioral sciences*. L. Earlbaum Associates, 1988.

[6] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368154.

[7] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X. doi: http://doi.acm.org/10.1145/353171.353183.

[8] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, 2006. ISSN 1532-060X. doi: http://dx.doi.org/10.1002/smr.v18:2.

[9] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer Berlin / Heidelberg, July 2006. ISBN 978-3-540-35726-1.

[10] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.71.

[11] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2601-2. doi: http://dx.doi.org/10.1109/ICPC.2006.16.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, 1995. ISBN 0201633612. URL `http://www.amazon.co.uk/exec/obidos/ASIN/0201633612/citeulike-21`.

[13] Daniel M. German and Ahmed E. Hassan. License integration patterns: Addressing license mismatches in component-based development. In *ICSE '09: Proceedings*

of the 2009 IEEE 31st International Conference on Software Engineering, pages 188–198, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: http://dx.doi.org/10.1109/ICSE.2009.5070520.

[14] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2005.28.

[15] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: http://doi.acm.org/10.1145/1062455.1062512.

[16] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Commun. ACM*, 20(5):350–353, 1977. ISSN 0001-0782.

[17] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 736, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1189-9.

[18] Christian Kemper and Charles Overbeck. What's new with jbuilder. In *JavaOne Sun's 2005 Worldwide Java Developer Conference*, 2005.

[19] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Washington, DC, USA, Not Available 2007. IEEE Computer Society. ISBN 0-7695-2828-7. doi: http://dx.doi.org/10.1109/ICSE.2007.20.

[20] Sunghun Kim, Kai Pan, and E. James Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Wash-

ington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2474-5. doi: http://dx.doi.org/10.1109/WCRE.2005.33.

[21] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *Sixth IEEE International Workshop on Source Code Analysis and Manipulation.*, pages 139–148, Sept. 2006.

[22] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. Technical Report 8, 1966.

[23] Guido Malpohl, James J. Hunt, and Walter E Tichy. Renaming detection. page 73, 2000.

[24] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 471–480, New York, NY, USA, May 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368153.

[25] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: managing the evolution of reusable assets. *SIGPLAN Not.*, 31(10):268–285, 1996. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/236338.236363.

[26] Peter Weißgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2. doi: http://dx.doi.org/10.1109/ASE.2006.41.

[27] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-59593-993-4. doi: http://doi.acm.org/10.1145/1101908.1101919.

[28] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on umldiff change-facts queries. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 263–274, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1. doi: http://dx.doi.org/10.1109/WCRE.2006.48.

[29] Zhenchang Xing and Eleni Stroulia. API-evolution support with diff-CatchUp. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 33(12):818 – 836, December 2007. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2007.70747.

[30] Robert K. Yin. *Case Study Research: Design and Methods - Third Edition*. SAGE Publications, 3 edition, 2002.