AURA: A Hybrid Approach to Identify Framework Evolution

Wei Wu[1], Yann-Gaël Guéhéneuc[1], Giuliano Antoniol[2], and Miryung Kim[3]

[1]Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada
[2] SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
[3] ECED, The University of Texas at Austin, USA

E-mail: {wuwei,guehene}@iro.umontreal.ca,
giuliano.antoniol@polymtl.ca, miryung@ece.utexas.edu

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, so 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, 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, *e.g.*, 0.10% lower.

1. INTRODUCTION

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 [2, 8, 21]. For example, from JHotDraw 5.2 to 5.3, method `CH.ifa.draw.figures.LineConnection.end()` was replaced by `LineConnection.getEndConnector()`; such a change may have direct consequences on a program using the JHotDraw framework, such as compile errors, or indirect ones, 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' efforts. Some require that the framework developers do additional work, such as providing explicit change rules with annotations [3], or that they record API updates to the framework. [9, 13, 15]. 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 [1, 5, 6, 7, 10, 12, 14, 17, 16, 19, 20, 22, 23].

However, framework developers may not be willing to build change rules manually or use specific tools. Also, some previous approaches [5, 20] cannot detect change rules for target methods not used within the previous releases of the framework and program. Others, such as [16], cannot identify replacement methods if the names of the old and new releases are not similar enough. Still others [5, 12, 17, 16, 20, 23] 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, as illustrated in the following section. 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.

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

---

[1]Without loss of generality, we use the term "framework" to mean both frameworks and libraries.

[2]We refer readers to [11] for a discussion on the links between frameworks and programs.

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 reduces developers' efforts by automatically generating one-to-many and many-to-one 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 [5]. We show that the approximated precision of AURA is 92.86% while SemDiff's is up to 100.00%.

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

2. MOTIVATING EXAMPLES

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

Multi-iteration Algorithm.

Let us assume that a developer must adapt her `Client` program from using Eclipse JDT 3.1 to its 3.3 release, as shown in Figure 1. The method `Indents.computeIndentLength(...)` was called in 3.1. However this method on longer exists in 3.3. Using an automatic approach, the developer would expect to obtain the following change rules:

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

↻ `IndentManipulation.getChangeIndentEdits(...)`

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

↻ `IndentManipulation.indexOfIndent(...)`

where ↻ means "should be replaced with".

Previous approaches, using either text similarity [16] or call dependency [5, 20] 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.computeInd-`

```
// 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 1: Example of many iterations

```
// 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: Example of a one-to-many change rule.

`entLength(...)` is replaced by `IndentManipulation.indexOfIndent(...)`.

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. 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.

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

↻ `CutCommand.CutCommand(Alignment, DrawingEditor)`

    and `UndoableCommand.UndoableCommand(Command)`

Previous approaches, using either text similarity [16] or

call dependency [5, 20] analyses, would only report a change rule from `CutCommand.CutCommand(DrawingView...)` to `Cut-Command.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(Drawing-View...)` with calls to the replacement methods `CutCommand.CutCommand(Alignment,DrawingEditor)` and `UndoableCom-mand.UndoableCommand(Command)`. Figure 2 illustrates the new implementation where an `UndoableCommand` now encapsulates `CutCommand`.

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(...)`, `Mar-kersMenu.MarkersMenu()`, and `RecentDirectoriesMenu.Re-centDirectoriesMenu()`, which are replaced by `EnhancedMe-nu.EnhancedMenu(...)`in the release 4.2.

With previous approaches [16, 20] that generate one-to-one change rules, the developer could know that `Directory-Menu.DirectoryMenu(...)` is replaced by `EnhancedMenu.-EnhancedMenu(...)`, but would need to find the other two methods manually. Some previous approaches, such as [16], 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.

3. RELATED WORK

Several approaches help developers evolve their programs when the frameworks that they use change. We studied these approaches and identified eight features. Table 1 summarizes the different approaches according to their features and highlight the advantages of AURA. We now further define and discuss the different features and approaches.

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 [3] presented a method that requires the framework developers to provide change rules with the new releases. CatchUP! [13] and JBuilder [15] record the refactoring operations in one release and replay them in another. MolhadoRef [9] 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.

Matching Techniques.

Previous approaches use different code matching techniques to find change rules between old and new releases. Dagenais and Robillard developed SemDiff [5], which suggests adaptation to clients by analyzing how a framework adapts to

its own changes. Schäfer *et al.* [20] mined framework-usage change rules from already ported instantiations. These two previous approaches compute support and confidence value on call dependency analysis. Godfrey and Zou [12] presented a semi-automatic hybrid approach to perform origin analysis using text similarity, metrics, and call dependency analyses. S. Kim *et al.* [17] automated Godfrey and Zou's approach. Diff-CatchUp, developed by Xing and Stroulia [24], analyses textual and structural similarities of UML logical design models to recognise API changes. M. Kim *et al.*'s [16] approach leveraged systematic renaming patterns to match old APIs to new APIs.

Many-to-one and one-to-many.

Godfrey and Zou [12] 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 [12], while the latter includes any case, *e.g.*, new functionality. SemDiff [5] and Diff-Catch-Up [24] 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 [16] automatically reports many-to-one rules.

Simply Deleted.

Simply-deleted target methods have no replacement methods in the new release. Semi-automatic approaches [5, 12, 24] and those that require framework developers' involvement [3, 9, 13] are able to report simply-deleted rules. Automatic approaches [17, 16, 20] do not report this type of change rule explicitly in their result.

Automatic and Thresholds.

All automatic approaches [17, 16, 20], except record-and-replay ones, use thresholds to keep a balance between precision and recall. Typically, they use experimentally-evaluated thresholds to filter out candidate replacement methods, thus potentially increasing precision but decreasing recall.

Types of Changes.

Schäfer *et al.* [20] 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 approaches and compare the results in Section 5.

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 re-

Approaches	Features									
	FDI	Main Matching Technique	One-to-many Rules	Many-to-one Rules	Simply-deleted Rules	Methods	Fields	Inheritance Relations	Auto-matic	Thres-holds
Chow *et al.* **[3]**	Yes	A	No	No	Yes	Yes	No	No	No	No
SemDiff [5]	No	CD	Yes	Yes	Yes	Yes	No	No	No	Yes
Godfrey *et al.* **[12]**	No	TS, M, and CD	Yes	Yes	Yes	Yes	No	No	No	Yes
CatchUp! [13]	Yes	N/A	No	No	No	Yes	Yes	Yes	Yes	No
M. Kim *et al.* **[16]**	No	TS	No	Yes	Yes	Yes	No	No	Yes	Yes
S. Kim *et al.* **[17]**	No	TS, M, and CD	No	No	No	Yes	No	No	Yes	Yes
Schäfer *et al.* **[20]**	No	CD	No	No	No	Yes	Yes	Yes	Yes	Yes
Diff-CatchUp [24]	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 1: Feature comparison. (A = Annotation, CD = Call Dependency, FDI = Framework Developer Involvement, M = Metrics, N/A = Not Applicable, TS = Text Similarity, SS = Structural Similarity)

placement 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.

- All automatic approaches except record-and-replay use thresholds set through experimental evaluations, which may not apply in any context.

4. OUR APPROACH

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 [5, 16, 20]; these techniques are compatible with each other because they apply directly to source code.

The assumption of our approach is that a target method is 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 Preliminary

Call dependency analysis discovers the calls between the methods of frameworks and 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 meth-

ods with the same signature (including return type, declaring module, 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:

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

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:

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

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

where $||S||$ represents the cardinality of S. 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.* [18] 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, names, and parameter list), then their Levenshtein Distance (LD), and finally, their Longest Common Subsequence (LCS). 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. For example, let us assume that we want to identify the string most similar to `ab` between `a`, `abc`, and `abcd`. Both `a` and `abc` have the same LD and both `abc` and `abcd` have the same LCS. Thus, by combining LD and LCS, we can identify that `abc` is most similar to `ab`.

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:

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

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:

$$\mathbf{CRMS}(t) = \{ m \mid m \in S_{gcrm} \wedge a \in S_a$$
$$\wedge \mathbf{OLD}(a) \to t$$
$$\wedge \mathbf{OLD}(a) \nrightarrow m$$
$$\wedge \mathbf{NEW}(a) \to m \}.$$

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 $t \circlearrowright c$ 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 meth-

ods 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:

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

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 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-replacements.

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:

$$\mathbf{S}(t,c) = ||\{ m \mid$$
$$m \in \{\text{all the methods in the new release}\}$$
$$\wedge m \to \mathbf{KR}(t) \wedge m \to c \}||$$

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 behavior 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 rule, if there is more than one candidate method with the same

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 2: Subject Systems.

text-similarity to a target. But according to our evaluation, 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.

4.3 Implementation

We implemented our approach as a Java program that includes two components: Model Builder and Rule Generator. The former component converts the source code of the old version and of the new version of a program into the language-independent AURA Model. The current version of Model Builder is an Eclipse plugin operating on the abstract syntax tree generated by the Eclipse Java parser. The latter component generates change rules using the AURA Model. It can be used both as an Eclipse plugin or as a standalone Java program. The executable and source code of AURA can be found on our Web site[5].

5. EVALUATION

We now evaluate and compare AURA on several systems.

5.1 Design

We evaluated AURA 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 2 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.* [16] and Schäfer *et al.* [20]. We use the large system (`org.eclipse.jdt.core` and `org.eclipse.jdt.ui`) to compare AURA with SemDiff [5].

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 the previous approaches do

---

[5] `www.ptidej.net/downloads/experiments/icse10b`

not report such rules. We convert many-to-one change rules into as many one-to-one change rules as target methods.

5.2 Hypothesis and Performance Indicators

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

We cannot use recall and precision [4] directly to compare the performance of AURA and the previous approaches because the set *relevant rules* is *a priori* unknown in:

$$\text{Precision} = \frac{||\{relevant\ rules\} \bigcap \{retrieved\ rules\}||}{||\{retrieved\ rules\}||}$$

$$\text{Recall} = \frac{||\{relevant\ rules\} \bigcap \{retrieved\ rules\}||}{||\{relevant\ rules\}||}$$

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:

$$\{correct\ rules\} = \{relevant\ rules\}$$
$$\bigcap \{retrieved\ rules\}.$$

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

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

Using $\Delta \mathbf{P}(A, B)$ and $\Delta \mathbf{R}(A, B)$, 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 the 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 [5]: 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.

5.3 Comparison on the Medium-size Systems

In Table 3, we present the $\Delta \mathbf{P}$ and $\Delta \mathbf{R}$ on each subject system between AURA and M. Kim *et al.*'s [16] and Schäfer *et al.*'s [20] 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

Systems	Indicators	AURA	M. Kim *et al.* [16]	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.* [20]	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 3: Comparison of the results on medium-size systems with simply-deleted change rules.

Systems		AURA	SemDiff [5]
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 4: Evaluation of a sample of change rules on the large system.

AURA compared to the two approaches: **ΔR** is 53.07% with a precision of 88.25%, while **ΔP** is -0.10%.

Comparison with M. Kim et al.'s [16] approach.

M. Kim *et al.* [16] present their results in two formats: first-order relational logic rules, for example "all methods in class `A`, replaced by the same name methods in class `B`, except methods `a()` and `b()`", and matches, for example `A.c() ↻ B.d()`. The latter format corresponds to the change rules of AURA. Therefore, we use the number of matches from [16] 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 recall 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 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.

Comparison with Schäfer et al.'s [20] approach.

On average, **ΔP** is 8.24% while **ΔR** is 52.86%. AURA has positive **ΔR** and **ΔP** 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 [20]. On JHotDraw from 5.2 to 5.3, the **ΔR** and **ΔP** 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). Text-similarity analysis generates the change rules for the other target methods.

In Schäfer *et al.*'s results [20], 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.

5.4 Comparison on A Large-size System

In Table 4, we present the results of AURA and SemDiff [5] to solve the compile errors of three Eclipse 3.1 plug-ins when compiling them against Eclipse 3.3.

In SemDiff [5], 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. Therefore, to compare the results of AURA with those of SemDiff, we must account for this discrepancy in the way correct rules are counted.

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.

---

[3]AURA only analyzed the packages org.gjt.sp.* and compared its results with those of M. Kim *et al.* [16]. These packages contain the code for JEdit main functions and are large enough for manual analysis (444 target methods).

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

5.5 Comparison w/o Simply-deleted Methods

Previous approaches, such as [16, 20], 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 the previous approaches to assess their influence on precision and recall.

As shown in Table 5, $\Delta\mathbf{P}$ is stable and remains similar to that with simply-deleted method rules (0.24% vs -0.10%). $\Delta\mathbf{R}$ decreases from 53.07% to 6.80%. The $\Delta\mathbf{R}$s of AURA to the two approaches [16, 20] are different. The $\Delta\mathbf{R}$ to Kim *et al.*'s approach [16] decreases to 13.34%, while the $\Delta\mathbf{R}$ to Schäfer *et al.*'s approach [20] drops to -3.02%.

The sharp decrease of $\Delta\mathbf{R}$ has two causes. First, 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 have different scopes, so $\Delta\mathbf{R}$ decreases dramatically.

Even with this decrease of $\Delta\mathbf{R}$ 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.

5.6 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.* [16] report computation times of seven minutes on average while Schäfer *et al.* [20] report less than 30 minutes, but do not specify their software and hardware platforms.

Analysing 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 [5] took 16 hours on a Pentium D 3.2Ghz with 2GB of RAM running Ubuntu Server 7.04.

5.7 Threats to Validity

We now discuss the threats to validity of our evaluation following the guidelines provided for case study research [25].

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 study, being a systematic comparison of AURA with the previous approaches using well-defined measures, $\Delta\mathbf{P}$ and $\Delta\mathbf{R}$.

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 the previous approaches without any changes other than those discussed in Section 5. 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 the previous approaches are publicly available or available upon request to their authors. The raw data on which our study is based are available on the Web[5].

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

6. DISCUSSION

We now discuss the strengths and limitations of AURA.

6.1 Strengths

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.* [20] can find 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 off recall for precision using 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 $\Delta\mathbf{R}$ 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.

Systems	Indicators	AURA	M. Kim *et al.* [16]	ΔR	ΔP	Averages	
						ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	96	81	18.26%	-3.03%		
	Precision	96.00%	99.00%				
JEdit 4.1-4.2	# Correct rule	247	217 [3]	13.99%	-17.00%	13.34%	-5.01%
	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.* [20]	ΔR	ΔP	Averages	
						ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	96	88	9.09%	9.09%		
	Precision	96.00%	88.00%			-3.02%	8.11%
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 5: Comparison of the results on medium-size systems without simply-deleted change rules

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

Previous approaches only automatically generate one-to-one change rules. Some approaches [5, 12] 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.

Threshold.

Existing automatic approaches [17, 16, 20], 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.

6.2 Limitations

AURA cannot detect one-to-many and many-to-one change rules for target methods that are not called by any anchor. However, it can still find one-to-one rules 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 [16] 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.

7. CONCLUSION AND FUTURE WORK

We presented AURA, an 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 [5] 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.* [16]; perform usability studies to determine the efficacy of AURA.

8. ACKNOWLEDGMENTS

Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

9. REFERENCES

[1] G. Antoniol, M. D. Penta, and E. 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.

[2] I. Balaban, F. Tip, and R. 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.

[3] K. Chow and D. 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.

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

[5] B. Dagenais and M. 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.

[6] S. Demeyer, S. Ducasse, and O. 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.

[7] D. Dig, C. Comertoglu, D. Marinov, and R. 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.

[8] D. Dig and R. Johnson. How do apis evolve? a story of refactoring: Research articles. *J. Softw. Maint. Evol.*, 18(2):83–107, 2006.

[9] D. Dig, K. Manzoor, R. Johnson, and T. 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.

[10] B. Fluri and H. 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.

[11] D. M. German and A. 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.

[12] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

[13] J. Henkel and A. 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.

[14] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. 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.

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

[16] M. Kim, D. Notkin, and D. 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.

[17] S. Kim, K. Pan, and E. J. 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, Washington, DC, USA, 2005. IEEE Computer Society.

[18] 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.

[19] G. Malpohl, J. J. Hunt, and W. E. Tichy. Renaming detection. page 73, 2000.

[20] T. Schäfer, J. Jonas, and M. 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.

[21] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: managing the evolution of reusable assets. *SIGPLAN Not.*, 31(10):268–285, 1996.

[22] P. Weißgerber and S. 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.

[23] Z. Xing and E. 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.

[24] Z. Xing and E. Stroulia. API-evolution support with diff-CatchUp. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 33(12):818 – 836, December 2007.

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