A History-Based Matching Approach to Identification of Framework Evolution

Sichen Meng[1,2], Xiaoyin Wang[1,2], Lu Zhang[1,2], Hong Mei[1,2]
[1]*Key Laboratory of High Confidence Software Technologies, Ministry of Education*
[2]*School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China*
{*mengsc11,wangxy06,zhanglu,meih*}*@sei.pku.edu.cn*

***Abstract*—In practice, it is common that a framework and its client programs evolve simultaneously. Thus, developers of client programs may need to migrate their programs to the new release of the framework when the framework evolves. As framework developers can hardly always guarantee backward compatibility during the evolution of a framework, migration of its client program is often time-consuming and error-prone. To facilitate this migration, researchers have proposed two categories of approaches to identification of framework evolution: *operation-based* approaches and *matching-based* approaches. To overcome the main limitations of the two categories of approaches, we propose a novel approach named *HiMa*, which is based on matching each pair of consecutive revisions recorded in the evolution history of the framework and aggregating revision-level rules to obtain framework-evolution rules. We implemented our *HiMa* approach as an Eclipse plug-in targeting at frameworks written in Java using SVN as the version-control system. We further performed an experimental study on *HiMa* together with a state-of-art approach named *AURA* using six tasks based on three subject Java frameworks. Our experimental results demonstrate that *HiMa* achieves higher precision and higher recall than *AURA* in most circumstances and is never inferior to *AURA* in terms of precision and recall in any circumstances, although *HiMa* is computationally more costly than *AURA*.**

***Keywords*-framework evolution; software migration; mining version history; natural language processing**

I. Introduction

Nowadays, many developers choose to use software frameworks to reduce the cost and/or improve the quality of application development. Just as other software products, software frameworks also need evolution to obtain new features and/or improve quality. As a result, when a software framework evolves, developers of client programs of the framework may need to migrate their client programs to the new release of the framework to take advantage of the new release. Ideally, developers of a framework should guarantee that the new release be backwardly compatible with each release before the new release, so that developers of any client program using an old release of the framework are able to immediately migrate to the new release without further modifying the client program. However, researchers (e.g., Chow and Notkin [1], Steyaert et al. [2], Balaban et al. [3], and Dig and Johnson [4]) have identified the pervasive existence of backward incompatibility between a new release and an old release in frameworks. Wu et al. [5] presented an example of backward incompatibility in a widely used framework (i.e., between JHotDraw 5.2 and 5.3). Thus, migration from an old release of a framework to a new release may imply extra modifications of the client code. These modifications may typically be tedious and require a lot of effort [5]. To alleviate the backward incompatibility problem, it is necessary to identify the evolution rules between the two releases of the framework.

In particular, researchers (e.g., Wu et al. [5]) have demonstrated the necessity of automatically identifying four types of evolution rules: *one-to-one* (i.e., one method in the old release replaced by another method in the new release), *one-to-many* (i.e., one method in the old release replaced by more than one method in the new release), *many-to-one* (more than one method in the old release replaced by one method in the new release), and *simply-deleted* (i.e., methods in the old release not appearing and having no replacements in the new release).

In fact, identification of framework evolution has become a research focus in recent years. Roughly speaking, there are mainly two categories of approaches to identification of framework evolution.

First, researchers (e.g., Henkel and Diwan [6]) have proposed *operation-based* approaches, which use Integrated Development Environments (IDEs) to record change operations related to framework evolution and replay the recorded operations to identify framework evolution. These approaches typically are able to achieve both high recall and high precision[1], but the requirement of using specialized IDEs makes them inapplicable for most existing frameworks.

Second, researchers (e.g., Wu et al. [5]) have proposed *matching-based* approaches, which compute evolution rules via matching the source code of the two releases. Due to the independence of IDEs, matching-based approaches are applicable for more existing frameworks. But matching-based approaches can hardly always achieve high precision without compromising recall due to the inability to sufficiently use finer information during framework evolution. As a sub-category of *matching-based* approaches, there are also approaches (e.g., Schäfer et al. [7]) that analyze adaptations in only instantiation code to infer framework-evolution rules.

[1]Recall refers to the ratio of correctly identified rules to all genuine rules and precision refers to the ratio of correctly identified rules to all identified rules.

The use of instantiation code[2] may help improve precision, but the reliance on only instantiation code makes these approaches difficult to obtain rules related to Application Programming Interfaces (APIs) not used in instantiation code.

In this paper, we propose a history-based matching approach (named *HiMa*) to identification of framework evolution. The basic idea of *HiMa* is to identify change rules related to framework evolution via matching each pair of consecutive revisions of the framework stored in its version-control system (e.g., SVN[3]), and aggregate the revision-level rules to obtain the evolution rules between the two releases. To achieve wide applicability, *HiMa* relies on only information stored in common software-development infrastructure. To achieve both high precision and high recall, *HiMa* analyzes all revisions between the two releases during the evolution history of the framework. Moreover, as our empirical investigation has identified that comments associated with revisions often contain information about framework evolution, we also utilize these comments in *HiMa* to further improve precision and recall.

To evaluate our *HiMa* approach, we conducted an experimental study of *HiMa* together with a recently proposed matching-based approach (i.e., *AURA* [5]) using six tasks based on three subject Java frameworks. Our experimental results demonstrate that *HiMa* achieves higher precision and higher recall than *AURA* in most circumstances and is never inferior to *AURA* in any circumstances, although the computational cost of *HiMa* is higher than that of *AURA*.

This paper makes the following main contributions:

- A novel approach, which overcomes main limitations of both categories of existing approaches, to identification of framework evolution.
- A technique based on natural language processing to identify change rules from comments associated with framework revisions.
- An experimental comparison of our approach with *AURA* to demonstrate the effectiveness of our approach.

The remaining of this paper is organized as follows. Section II reviews existing research related to ours. Section III presents our approach. Section IV reports an experimental study of our approach. Section V discusses further issues. Section VI concludes with pointers to future work.

II. Related Work

A. Identification of Framework Evolution

For the ease of presentation, we classify existing approaches to identification of framework evolution into two categories: *operation-based* approaches and *matching-based* approaches.

[2]Researchers typically refer to the code invoking framework APIs as instantiation code when investigating framework-evolution identification.

[3]subversion.apache.org, accessed in March 2012

1) Operation-Based Approaches: The basic idea of *operation-based* approaches is to record change operations performed on the framework in an IDE and replay the operations to identify framework evolution. Henkel and Diwan [6] proposed *CatchUP!*, which uses a specialized IDE to record refactoring operations performed on APIs of the framework and replay these operations to update client code. Moreover, Dig et al. [8] proposed a refactoring-aware software configuration management tool named *MolhadoRef*, which also utilizes recording and replaying of change operations to merge versions. Although *MolhadoRef* does not explicitly target at identification of framework evolution, it can be adapted for this purpose.

As API refactoring operations can accurately characterize how the framework evolves, operation-based approaches are typically very accurate. That is to say, both recall and precision of operation-based approaches can be very high. However, recording and maintaining API refactoring operations along the evolution history of the framework may be a burden for framework developers. Thus, although some widely used IDEs (e.g., *JBuilder* [9]) already provide the capability of recording change operations, many existing frameworks are developed with IDEs not providing (or switching off) the capability of recording change operations. As a result, operation-based approaches are not widely applicable for existing frameworks.

To overcome limitations of existing operation-based approaches, our approach is based on revisions instead of operations in the evolution history of the framework. Due to the wide adoption of version-control systems (e.g., SVN) in practice, there may typically exist a large number of revisions between two releases of a framework. Of course, information in revisions is still less fine than information in operations. But the availability of revisions is much higher than that of operations. Moreover, as developers often provide a summary of change in the form of plain texts associated with a revision, our approach also uses this information to further improve accuracy.

2) Matching-Based Approaches: The basic idea of *matching-based* approaches is to compare the source code of the two releases, and figure out evolution rules from the differences between the two releases using some heuristics. In particular, researchers have investigated the following types of heuristics: heuristics based on call dependency [10], [11], [12], [13], [5], [14], heuristics based on text similarity [15], [11], [12], [16], [17], [18], [5], [14], heuristics based on structure similarity [16], and heuristics based on metrics [19], [11], [12], [20], [18], [14]. Some approaches even use more than one type of heuristics. Godfrey and Zou [11] proposed to use origin analysis, which relies on heuristics based on text similarity, call dependency, and metrics, to infer evolution rules. As Godfrey and Zou's approach is semi-automatic, S. Kim et al. [12] further proposed to automate their approach. Weißgerber and Diehl [20] proposed to use metrics based on syntactical differences to infer candidate refactorings from changes between file versions and use clone analysis to rank the candidate refactorings. Xing and Stroulia [16] proposed *Diff-Catchup*, which matches two models written in the Unified Modeling Language (UML) using heuristics based on text similarity and structure similarity. M. Kim et al. [17] summarized some patterns for name changes during framework evolution and used these patterns to infer rules of framework evolution. The most advanced matching-based approach is *AURA* [5], which involves multiple rounds of iteration using heuristics based on call dependency and text similarity. According to Wu et al. [5], the *AURA* approach is able to overcome several limitations of previous approaches and vastly improve recall. However, *AURA* typically concedes some decrease in precision and in some special cases the decrease in precision may be significant.

It should be noted that some of the preceding heuristics can be extended to instantiation code to explore how instantiation code of the framework adapts to framework evolution. Schäfer et al. [7] proposed to infer rules of framework evolution from the changes in client code for migrating from one release to another. Dagenais and Robillard [21] proposed *SemDiff*, which infers rules of framework evolution via analyzing how the framework changes itself in response to its evolution. Due to the abundance of adaptation examples for framework evolution, using instantiation code can improve precision due to the capability to corroborate between adaptation examples. Moreover, *SemDiff* considers adaptations in the evolution history between the two releases. The use of adaptation information in a finer granularity further improves its accuracy. However, rules inferred from instantiation code do not include those related to APIs not used in instantiation code. That is to say, using only instantiation code may even decrease recall. Note that frameworks typically contain cold spots [22], which are APIs seldom used in instantiation code.

As matching-based approaches do not rely on recorded operations, applicability of matching-based approaches is typically higher than operation-based approaches. However, matching-based approaches can hardly always be very accurate due to the inability to sufficiently use finer evolution information. In fact, matching-based approaches typically need to balance between precision and recall. In general, the *AURA* approach makes a good trade-off between precision and recall, but some complex changes inside method bodies may mislead *AURA* [5].

To overcome limitations of existing matching-based approaches, our approach uses finer information recorded in revisions together with their comments recorded in the evolution history of the framework. Because of the abundance of information, our approach is able to use strict criteria to identify enough rules and corroborate among the information to discard incorrect rules. Note that, although Weißgerber and Diehl's approach[4] [20] also utilizes file versions in version-control systems, their approach aims to infer possible refactorings from one transaction committed to a framework but not to further aggregate refactorings inferred from multiple transactions to obtain evolution rules between two releases of the framework. Furthermore, their approach does not utilize comments associated with versions and requires human intervention to check the ranked list.

B. Natural Language Processing for Software Engineering

Sawyer et al. [23] proposed *REVERE*, which combines part-of-speech (POS) [24] tagging and semantic tagging to synthesize requirements from documents. Fantechi et al. [25] proposed to analyze use case descriptions in natural languages using natural language parsing [26]. Kof [27] proposed to use POS tagging to analyze requirement documents to obtain missing objects and actions. Shepherd et al. [28] proposed to locate and understand action-oriented concerns in programs using a combination of several Natural Language Processing (NLP) techniques. Tan et al. [29] proposed *iComment*, which combines NLP and other techniques to extract explicit program rules and detects inconsistencies between comments and source code. On top of *iComment*, Tan et al. [30] proposed *aComment*, which combines NLP with call-graph analysis to infer specifications for interruptions and detects interruption-related bugs. Zhong et al. [31] proposed *Doc2Spec*, which combines named entity recognition (NER) [32] and class-hierarchy analysis to infer implicit specifications from object-oriented API documents. Abebe and Tonella [33] proposed to use natural language parsing to extract concepts from source code.

To our knowledge, our approach, which uses specialized NLP techniques with natural language parsing [26] to infer change rules from revision comments, is the first approach that applies NLP for framework-evolution identification.

III. Our Approach

A. Basic Idea

The basic idea of our approach is to identify change rules between each pair of consecutive revisions and aggregate these change rules to obtain the evolution rules between the two releases. Note that a mainstream version control system such as SVN typically records every revision (subversion) of the entire software system under development.

Intuitively, these revisions provide more information of changes made in the evolution history of the framework. For example, Struts 1.1 and Struts 1.2.4 correspond to revisions 50638 and 51732 in its corresponding SVN repository with 1073 revisions containing code changes in between.

Furthermore, according to our empirical evidence, framework developers often summarize the main changes between a revision and its predecessor revision in the comment for

[4]Weißgerber and Diehl's approach actually targets at refactoring identification, which is slightly different from framework-evolution identification. Please refer to Schäfer et al. [7] for further discussion on the main difference between the two problems.

checking in the successor revision. When identifying change rules between two revisions, we use the comment associated with the successor revision as our initial clues.

B. Matching Two Consecutive Revisions

We use the following two main steps to identify change rules between two revisions. First, we analyze the comment of the successor revision to identify some raw change rules, and validate the raw change rules against the source code to obtain a set of initial change rules. Second, we expand the set of identified initial change rules to obtain change rules for other changes between the two revisions. Unlike *AURA*, to reduce computational cost, we do not use iteration.

1) Identifying Initial Change Rules: We use the following five sub-steps to identify initial change rules.

First, we split the entire comment into sentences. In natural language processing, delimiter punctuation marks (e.g., '?', '!', and '.') serve as separations of sentences. We also rely on these delimiters, but we need to handle the following two special cases. The first case is the dot mark serving as a connector of a class and its member (e.g., `Circle.toString()`). To deal with this case, we use a dot together with a following space instead of mere a dot as a delimiter. The second case is the hard new-line symbol. As developers may directly use a hard new-line symbol to start a new sentence, we also count hard new-line symbols as delimiters.

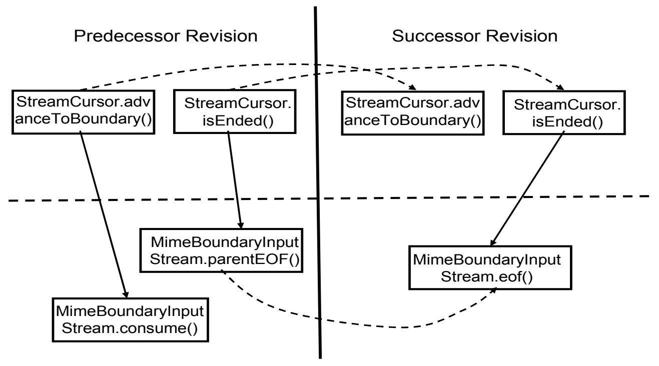
Second, we filter out irrelevant sentences as follows. Here, we are primarily interested in words and phrases meaning either addition, deletion, or replacement, which are referred to as *change identifiers* in this paper. In fact, we maintain three synonym lists (which we put on our project website to keep them always consistent with our implementation), each representing a *change type*, which could be *addition*, *deletion*, or *replacement*. Furthermore, we are also interested in sentences containing entity names (i.e., class names or method names). In particular, we deem the following cases as entity names: 1) a word after the word "class" or the word "method", 2) a pair of empty parenthesis or a pair of parenthesis inside which the text matches the syntax of method signatures, 3) a word immediately following a dot (e.g., `Resize` in `Circle.Resize`), and 4) a word composed of several words through underlines or capitalization of the following words (e.g., `ContentTransferEncodingField`). Thus, we keep only those sentences that contain at least one *change identifier* and at least one entity name. Note that the criteria for entity names considered here may not be very accurate. The aim of using these criteria is to filter out irrelevant sentences, and we further validate entity names in the final sub-step.

Third, for each remaining sentence after the filtering in the second sub-step, if all the *change identifiers* are of the same *change type* (denoted as T), we extract one raw change rule of change type T consisting of all the entity names in the sentence. For example, if a sentence contains three entity names (denoted as *Name1*, *Name2* and *Name3*) and the *change type* is *addition*, we denote the change rule as $<addition, \{Name1, Name2, Name3\}>$.

Fourth, if such a sentence contains *change identifiers* of more than one *change type*, we extract more than one change rule for this sentence. To achieve this goal, we use a natural language parser to identify the verbs in the sentence and all the words associated with each verb. If such a verb is a *change identifier*, we extract a rule for the verb considering only entity names associated with the verb in a way similar to the third sub-step. For example, from sentence "we added method A() and changed method B() to B1()", our *comment analysis* is able to identify two raw rules: $<addition, \{A\}>$ and $<replacement, \{B\}, \{B1\}>$. According to our experience, there are only a few sentences containing *change identifiers* of two or more change types. Thus, the heavy-weight natural language parsing would not be of too much a burden.

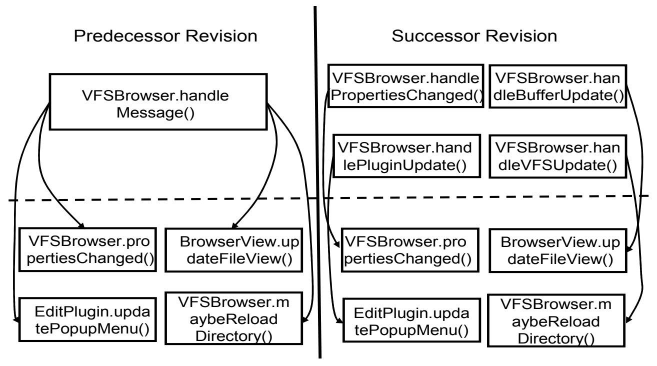
Finally, we validate and refine the raw change rules identified in the third and the fourth sub-steps against the source code of the two revisions. In particular, we use the following strategies for our validation and refinement.

- If the raw rule type is *addition*, for each entity name (denoted as n) in this rule, we check whether n exists in the successor revision and does not exist in the predecessor revision. If so, we keep n in the rule; and otherwise, we remove n from the rule. If n is a method name, we deem n as an *newly-added* method. If n is a class name, we deem all methods in class n as *newly-added* methods.
- If the raw rule type is *deletion*, for each entity name (denoted as n) in this rule, we check whether n exists in the predecessor revision and does not exist in the successor revision. If so, we keep n in the rule; and otherwise, we remove n from the rule. If n is a method name, we deem n as a *simply-deleted* method. If n is a class name, we deem all methods in class n as *simply-deleted* methods.
- If the raw rule type is *replacement*, for each entity name (denoted as n) in this rule, we check whether n exists in the predecessor revision or exists in the successor revision. Here, we demand that all the entity names in this rule are class names or all the entity names are method names; otherwise, we ignore this raw rule due to uncertainty of whether it is class replacement or method replacement. We also ignore all entity names not appearing in either the predecessor revision or the successor revision. Among the remaining entity names, we deem entity names appearing only in the successor revision as target entities, and we deem the other entity names as source entities. According to Dig et al. [13] and Schäfer et al. [7], developers may keep both the source entity and the target entity in the successor revision for backward compatibility.



Predecessor Revision
Successor Revision
StreamCursor.advanceToBoundary()
StreamCursor.isEnded()
StreamCursor.advanceToBoundary()
StreamCursor.isEnded()
MimeBoundaryInputStream.parentEOF()
MimeBoundaryInputStream.eof()
MimeBoundaryInputStream.consume()

Figure 1. Example for *Caller Analysis*



Predecessor Revision
Successor Revision
VFSBrowser.handleMessage()
VFSBrowser.handlePropertiesChanged()
VFSBrowser.handleBufferUpdate()
VFSBrowser.handlePluginUpdate()
VFSBrowser.handleVFSUpdate()
VFSBrowser.propertiesChanged()
BrowserView.updateFileView()
EditPlugin.updatePopupMenu()
VFSBrowser.maybeReloadDirectory()
VFSBrowser.propertiesChanged()
BrowserView.updateFileView()
EditPlugin.updatePopupMenu()
VFSBrowser.maybeReloadDirectory()

Figure 2. Example for *Callee Analysis*

Similarly, we transform class replacement to method replacement: a *one-to-one* rule for each pair of exactly matched methods and one replacement rule for all the remaining methods. Thus, depending on the number of the source methods and the number of the target methods, each resulting change rule can be of one of the following types: *one-to-one*, *many-to-one*, *one-to-many*, or even *many-to-many*.

2) Expanding Initial Set of Change Rules: To expand the initial set of change rules identified in the first main step, similar to existing matching-based approaches, we also rely on call-dependency analysis. However, as the differences between two consecutive revisions are typically very small, we are able to use strict criteria without thresholds for our expansion.

First, we analyze the source code of the two revisions to obtain the set of source methods (i.e., methods in the predecessor revision but not in the successor revision) and the set of target methods (i.e., methods in the successor revision but not in the predecessor revision). We also remove methods already appearing in the initial set of change rules from the two sets. We denote the resulting set of source methods as M_S and the resulting set of target methods as M_T.

Second, based on M_S, M_T and change rules identified in the first main step, we use *caller analysis* to expand change rules as follows. The intuition is that methods always invoked in the same contexts may have correspondence. Given method s in the predecessor revision and method t in the successor revision, we call t matches s through *one-to-one* relationships (denoted as $M_{\leftrightarrow}(s,t)$), if and only if the signature of t is exactly the same as that of s or there is a *one-to-one* rule identified in the first main step between s and t. For a set of methods in the predecessor revision (denoted as S) and a set of methods in the successor revision (denoted as T), we extend the definition of $M_{\leftrightarrow}(s,t)$ to define $M_{\leftrightarrow}(S,T)$ as Formula 1.

$$M_{\leftrightarrow}(S,T) = \begin{cases} true, & \forall s \in S \exists t \in T\ M_{\leftrightarrow}(s,t) \wedge \\ & \forall t \in T \exists s \in S\ M_{\leftrightarrow}(s,t); \\ false, & \text{otherwise.} \end{cases} \qquad (1)$$

Thus, given a method (denoted as m_s) in M_S and a method (denoted as m_t) in M_T, let us denote the set of callers of m_s in the predecessor revision as $Caller(m_s)$ and the set of callers of m_t in the successor revision as $Caller(m_t)$. We deem that there is a replacement relationship between m_t and m_s, if and only if $M_{\leftrightarrow}(Caller(m_s), Caller(m_t))$. Note that we rely on only identified *one-to-one* relationships between methods to guarantee precision. Due to our definition of $M_{\leftrightarrow}$, it is likely for us to identify a subset (denoted as M'_S) of M_S and a subset (denoted as M'_T) of M_T such that there is a replacement relationship between each method in M'_T and each method in M'_S. That is to say, one expanded change rule may be either a *one-to-one* rule, a *many-to-one* rule, a *one-to-many* rule, or even a *many-to-many* rule, depending on the number of methods in M'_S and the number of methods in M'_T.

Figure 1 depicts an example (from Mime4J) for expansion with *caller analysis*. In the figure, methods `advanceToBoundary()` and `isEnded()` appear in both the predecessor revision and successor revision. We need to identify the relationships between methods `parentEOF()` and `consume()` appearing in only the predecessor revision and method `eof()` appearing in only the successor revision. As both `parentEOF()` and `eof()` are called by only `isEnded()`, we identify a *one-to-one* rule: `parentEOF()` replaced by `eof()`. We identify no rule for `consume()` in this sub-step.

Third, based on M_S, M_T and change rules identified in the first main step, we also use *callee analysis* to expand change rules. The intuition is that methods sharing very similar calling structures in implementation may have correspondence. Similar to our *caller analysis*, our *callee analysis* is also based on existing *one-to-one* relationships. However, we also need to further consider unaligned matching of callees in *callee analysis*. Given a subset (denoted as M'_S) of M_S and a subset (denoted as M'_T) of M_T, we deem that there is a replacement rule between M'_S and M'_T if and only if $M_{\leftrightarrow}(\bigcup_{m_s \in M'_S} Callee(m_s), \bigcup_{m_t \in M'_T} Callee(m_t))$, where $Callee(m)$ denotes the set of callees of method m. To ensure the atomicity of the identified replacement rule, we also demand that there is no replacement rule identified in this sub-step between any subset other than M'_S itself and any subset other than M'_T itself.

Figure 2 depicts an example (from jEdit) for expansion with *callee analysis*. In this figure, the following four methods appear in both the predecessor revision and the successor revision: `propertiesChanged()`, `updateFileView()`, `updatePopupMenu()`, and `maybeReloadDirectory()`. In the predecessor revision, all the four methods are called by `handleMessage()`, but in the successor revision, each of the four method is called by only one method. Using the preceding *callee analysis*, we are able to identify the splitting of the method for handling all messages into four methods, each handling a specific type of messages.

Fourth, among the remaining methods in M_S and M_T, we further use some *mapping conditions* (which are actually inspired by Weißgerber and Diehl's conditions for local refactorings [20]) to identify *one-to-one* replacement between methods that have no calling relationships with other methods. Let us use $N(m)$, $R(m)$, and $P(m)$ to denote the name, the return type, and the parameters of m, respectively. If there exist $m_1 \in M_S$ and $m_2 \in M_T$ such that m_1 and m_2 are in the same class and satisfy one of the following conditions: 1) $N(m_1) \neq N(m_2) \wedge R(m_1) = R(m_2) \wedge P(m_1) = P(m_2)$, 2) $N(m_1) = N(m_2) \wedge R(m_1) \neq R(m_2) \wedge P(m_1) = P(m_2)$, or 3) $N(m_1) = N(m_2) \wedge R(m_1) = R(m_2) \wedge P(m_1) \neq P(m_2)$, we deem that there is a *one-to-one* rule in the form of m_1 replaced by m_2.

Finally, if there are still methods in M_S associated with no change rules, we deem them as *simply-deleted*. Similarly, if there are still methods in M_T associated with no change rules, we deem them as *newly-added*. It should be noted that, if a *simply-deleted* method m in class X actually overrides another method m' defined in a super class of X, we adjust the rule for deleting m to a replacement rule (i.e., m replaced by m'), because the semantics of deleting m in object orientation is to direct calls of m to m'.

C. Aggregating Change Rules

Supposing that we have n revisions (denoted as R_1, R_2,...R_n, where R_1 and R_n are two releases) of the framework, and we have identified a set of change rules between each pair of consecutive revisions, we aggregate these change rules as follows. We aggregate rules for changing from R_1 to R_2 with rules for changing from R_2 and R_3 to obtain rules for changing from R_1 to R_3. We continue this aggregation process until we get rules for changing from R_1 to R_n. For the ease of presentation, we focus on how we aggregate change rules between R_1 and R_2 with change rules between R_2 and R_3 below.

First, we deem either a rule for a *newly-added* method or a rule for a *simply-deleted* rule as a rule for method replacement. That is to say, a change rule is in the form of a set of methods (denoted S) in the predecessor revision replaced by a set of methods (denoted as T) in the successor revision. If S is empty, the rule is a rule for *newly-added* methods. In such a case, we break the rule into several rules, each of which is about one *newly-added* method. If T is empty, the rule is a rule for *simply-deleted* methods. In such a case, we also break the rule into several rules, each of which is about one *simply-deleted* method. Thus, any change rule r is in the form of $Source(r)$ replaced by $Target(r)$. Let us denote the set of rules for changing R_1 to R_2 as $RS_{1\to2}$=$\{a_1$,a_2,...$a_k\}$ and the set of rules for changing R_2 to R_3 as $RS_{2\to3}$=$\{b_1$,b_2,...$b_m\}$.

Second, we deem both change rules in $RS_{1\to2}$ not related to any change rules in $RS_{2\to3}$ and change rules in $RS_{2\to3}$ not related to any change rules in $RS_{1\to2}$ as rules for changing from R_1 to R_3. Change rule a in $RS_{1\to2}$ and change rule b in $RS_{2\to3}$ are related to each other (denoted as $Relate(a,b)$), if and only if $Target(a) \cap Source(b) \neq \emptyset$.

Third, among the remaining related change rules in $RS_{1\to2}$ and $RS_{2\to3}$, we rewrite these change rules as follows. For such a rule (denoted as a) in $RS_{1\to2}$, we calculate the set of rules that are transitively related to a. Two rules (denoted as x and y) in $RS_{1\to2} \cup RS_{2\to3}$ are transitively related to each other (denoted as $TranRelate(x,y)$), if and only if $Relate(x,y)$ or $\exists c_1, c_2, ...c_p \in RS_{1\to2} \cup RS_{2\to3}, (c_1 = x) \wedge (c_p = y) \wedge Relate(c_i, c_{i+1}) (1 \leq i < p)$. Supposing that the set of rules that are transitively related to a is $RS'_{1\to2} \cup RS'_{2\to3}$ (where $RS'_{1\to2} \subseteq RS_{1\to2}$ and $RS'_{2\to3} \subseteq RS_{2\to3}$), we have the following rule for changing from R_1 to R_3: $\bigcup_{c \in RS'_{1\to2}} Source(c) \cup (\bigcup_{c \in RS'_{2\to3}} Source(c) - \bigcup_{c \in RS'_{1\to2}} Target(c))$ replaced by $\bigcup_{c \in RS'_{2\to3}} Target(c) \cup (\bigcup_{c \in RS'_{1\to2}} Target(c) - \bigcup_{c \in RS'_{2\to3}} Source(c))$. We continue to rewrite the remaining rules until we have rewritten all the change rules in $RS_{1\to2} \cup RS_{2\to3}$.

Supposing that we have the rule of "A replaced by B1 and B2" for changing from revision R_1 to revision R_2 and the rule of "B2 and B3 replaced by C" for changing from revision R_2 to revision R_3, the preceding rule rewriting enables us to identify the rule of "A and B3 replaced by B1 and C" for changing from revision R_1 to revision R_3.

Fourth, we further resolve conflicts between change rules. In particular, we consider the following two cases of conflicts. The first case is a conflict between a rule for a *simply-deleted* method and a rule for *a newly-added* method. This conflict occurs when a method already deleted in R_2 is added back in R_3. For such a case, we drop both rules. The second case is a conflict between a rule for a *simply-deleted* method and a rule for method replacement. This conflict occurs when a method is replaced by another method in R_2 but still kept for backward compatibility, and the obsolete method is deleted in R_3. For such a case, we drop the rule for the *simply-deleted* method.

Finally, after we have obtained the change rules between

Table I
BASIC INFORMATION OF THE SUBJECTS

Framework	Release	Size (KLOC)	Packages	Classes	Methods
jEdit	4.1	114	28	341	3891
	4.2	141	31	394	4500
	4.3	177	38	531	5868
Struts	1.1	176	51	720	5939
	1.2.4	161	52	798	6070
	1.2.7	157	52	777	6527
Mime4J	0.3	11	17	74	431
	0.4	23	32	144	989
	0.5	24	32	156	1080

R_1 and R_n, we validate the rules against the source code of R_1 and R_n in the same way as the validation of rules in Section III-B1. Note that, as rules between some revisions may not be fully accurate, we need to check the resulting rules to get rid of invalid rules.

D. Implementation

We implemented our *HiMa* approach as an Eclipse plug-in for Java frameworks that use SVN as the version-control system. As mentioned previously, *HiMa* relies on SVN to locate each revision of the framework. Our implementation further utilizes features of SVN and Java to avoid unnecessary code analysis. First, SVN enables us to focus on only the changed files when matching two revisions. Second, as classes in Java programs correspond to files, our implementation directly derives information for addition or deletion of classes when encountering added or deleted files. Furthermore, we implemented our *comment analysis* on top of an open-source tool for NLP named *OpenNLP*[5].

IV. EXPERIMENTAL STUDY

To evaluate our *HiMa* approach, we conducted an experimental study on *HiMa*. In our experimental study, we investigated the following two research questions.

- **RQ1**: How effective is our *HiMa* approach in comparison with existing approaches?
- **RQ2**: How do the four main techniques in our *HiMa* approach contribute to the overall effectiveness?

The first research question is concerned with whether *HiMa* is competitive with existing approaches to framework-evolution identification. The second research question is concerned with how the main techniques in *HiMa* impact the effectiveness of *HiMa*.

A. Experimental Design

As there exist quite a few approaches to framework-evolution identification, we compared our *HiMa* approach with *AURA*[6] in our experimental study when investigating the effectiveness of *HiMa*. There are mainly two reasons for us to choose *AURA* for comparison. First, *AURA* is a state-of-art approach to framework-evolution identification and *AURA* provides several advanced features that previous

[5]opennlp.sourceforge.net/projects.html, accessed in March 2012

[6]www.ptidej.net/downloads/experiments/icse10b, accessed in March 2012

Table II
OVERALL EFFECTIVENESS

Framework& Task	*HiMa*			*AURA*		
	Ide.	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*
jEdit&Task1	432	416	96.3	420	317	75.5
jEdit&Task2	1057	1043	98.7	1059	881	83.2
Struts&Task1	184	183	99.5	184	179	97.3
Struts&Task2	84	84	100.0	85	69	81.2
Mime4J&Task1	208	205	98.6	206	184	88.4
Mime4J&Task2	98	98	100.0	98	87	88.8

approaches do not provide (please refer to our discussion in Section V). Second, Wu et al. [5] have empirically compared *AURA* with several existing approaches (i.e., M. Kim et al.'s approach [17], Schäfer et al.'s approach [7], and the *SemDiff* approach [21]) and demonstrated the competitiveness and even superiorness of *AURA*.

In our experimental study, we used three open-source Java frameworks as subjects: Struts[7], jEdit[8], and Mime4J[9]. Struts is a framework for developing Java web applications; jEdit is a text editor with APIs for users to develop plug-ins; and Mime4J is a framework for developing applications that analyze email messages. The reasons for choosing these three Java frameworks are as follows. First, all the three subjects are frameworks of medium sizes, and thus it is feasible for us to manually inspect whether each identified rule is correct. Second, some of the subjects (e.g., jEdit and Struts) have been used in previous studies (e.g., M. Kim et al. [17], Schäfer et al. [7], and Wu et al. [5]), and thus it is helpful to compare our experimental results with theirs.

For each subject, we applied *HiMa* and *AURA* for two tasks involving three releases. That is to say, for each subject the first task is based on the first two releases and the second task is based on the last two releases. Thus, we are able to check whether the experimented approaches performed consistently on each subject. For example, based on jEdit, we formed the following two tasks for framework-evolution identification: from release 4.1 to release 4.2 and from release 4.2 to release 4.3. Note that, to avoid basing our experiments in a scenario in favor of our approach, we did not use the task based on the first and the third releases for each subject. Table I depicts the basic information of the three releases of each subject.

For each subject and each task, either *HiMa* or *AURA* identified a set of evolution rules. As *AURA* identifies only four types of rules (i.e., *simply-deleted*, *one-to-one*, *one-to-many*, and *many-to-one*), we considered rules of only these four types in our experimental study. We adopted a way similar to Wu et al. [5] to determine the correctness of identified rules. That is to say, for each set of identified evolution rules, we manually inspected whether each rule in the set is correct with the help of framework documentation. In particular, the first and the second authors of this paper

[7]struts.apache.org, accessed in March 2012

[8]www.jedit.org, accessed in March 2012

[9]james.apache.org/mime4j, accessed in March 2012

Table III
OVERALL EFFECTIVENESS FOR DIFFERENT TYPES OF RULES

Framework& Task	*Simply-Deleted*						*One-to-One*						*One-to-Many*						*Many-to-One*					
	HiMa			*AURA*			*HiMa*			*AURA*			*HiMa*			*AURA*			*HiMa*			*AURA*		
	Ide.	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*
jEdit&T1	206	192	93.2	126	115	91.3	214	212	99.1	264	190	72.0	9	9	100.0	13	9	69.2	3	3	100.0	17	3	17.7
jEdit&T2	169	157	92.9	148	97	65.5	883	881	99.8	884	781	88.4	1	1	100.0	13	2	15.4	4	4	100.0	14	1	7.1
Struts&T1	76	76	100.0	79	77	97.5	108	107	99.1	105	102	97.1	0	0	-	0	0	-	0	0	-	0	0	-
Struts&T2	66	66	100.0	50	50	100.0	18	18	100.0	35	19	54.3	0	0	-	0	0	-	0	0	-	0	0	-
Mime4J&T1	17	16	94.1	14	6	42.9	191	189	99.0	191	178	93.2	0	0	-	0	0	-	0	0	-	1	0	0.0
Mime4J&T2	15	15	100.0	8	6	75.0	83	83	100.0	88	81	92.0	0	0	-	2	0	0.0	0	0	-	0	0	-

Table IV
DETAILED COMPARISON OF IDENTIFIED RULES

Framework& Task	*HiMa&AURA*		Only *HiMa*		Only *AURA*	
	Ide.	*Cor.*	*Ide.*	*Cor.*	*Ide.*	*Cor.*
jEdit&Task1	310	307	122	109	110	10
jEdit&Task2	870	869	187	174	189	12
Struts&Task1	177	176	7	7	7	3
Struts&Task2	68	68	16	16	17	1
Mime4J&Task1	187	184	21	21	19	0
Mime4J&Task2	87	87	11	11	11	0

together with another two students (not authors of this paper) participated in the manual inspection, and we ensured that each identified rule was inspected by at least two inspectors. In case of conflicts, all the related inspectors discussed face to face to resolve the conflicts. In our manual inspection, we payed special attention to the rules identified by only *HiMa* and the rules identified by only *AURA*, because human errors made on determining the correctness of such rules would affect *HiMa* and *AURA* differently. We put the detailed information of the identified rules on our project website[10].

As our *HiMa* approach contains several techniques, to investigate how the main techniques in *HiMa* contribute to its overall effectiveness, we further experimentally compared five combinations of the main techniques in *HiMa*. That is to say, besides *HiMa* itself, we also experimented with another four combinations, each turning off one technical component. Similar to the experimental comparison between *HiMa* and *AURA*, we manually inspected the correctness of each identified rule and put the detailed information on our project website.

We also recorded the execution time for either *HiMa* or *AURA* to perform each task on each subject. All the execution time is based on executing *HiMa* and *AURA* on Windows 7 with an Intel Core i5 2.53GHz CPU and 6GB memory. We used the execution time of *HiMa* and *AURA* as an indicator for their performance.

B. Results and Analysis

1) RQ1: Overall Effectiveness: Table II depicts the overall effectiveness of *HiMa* and *AURA*. In this table, columns 2-4 depict the number of rules identified by *HiMa*, the number of correct rules identified by *HiMa*, and the precision of *HiMa* in percentage points for each subject and each task. Similar to columns 2-4, columns 5-7 depict these numbers for *AURA*. From Table II, we have the following observation.

[10]sourceforge.net/projects/hima

For each subject and each task, *HiMa* consistently achieves higher precision than *AURA* and as *HiMa* is able to always identify more correct evolution rules, *HiMa* also consistently achieves higher recall than *AURA*, although *AURA* also achieves competitive results. For example, for the first task on Mime4J, *HiMa* identifies 21 more correct rules and achieves ten percentage points higher in precision than *AURA*. Note that, as we do not know the exact number of correct evolution rules in total for each of the six tasks, we are not able to calculate the exact recall values for *HiMa* and *AURA*.

We further compare the overall effectiveness of *HiMa* and *AURA* in terms of each rule type in Table III. Similar to Table II, columns headed by "*Ide.*" depict the numbers of identified rules, columns headed by "*Cor.*" depict the numbers of correctly identified rules, and columns headed by "*Pre.*" depict the values of precision in percentage points. From Table III, we have the following observations.

First, the trend of overall effectiveness of *HiMa* and *AURA* in terms of each rule type is similar to the trend of overall effectiveness for all types. That is to say, *HiMa* achieves both precision and recall no less than *AURA* for all circumstances. This observation demonstrates *HiMa*'s consistent superiorness over *AURA* for each rule type. Furthermore, *HiMa* never achieves poor precision for any circumstances, while there are some circumstances for *AURA* to perform unsatisfactorily (e.g., the second task for *AURA* to identify *one-to-one* rules for Struts and the first task for *AURA* to identify rules of *simply-deleted* methods for Mime4J). It should be noted that there are 10 correct rules (i.e., 7 in the second task for jEdit and 3 in the first task for Struts) that *HiMa* and *AURA* identified as different types. The reason is that *HiMa* deems the deletion of an overridden method in a class as redirecting calls to a method in a superclass.

Second, neither *HiMa* nor *AURA* identifies many rules for *one-to-many* or *many-to-one* replacement. We suspect the reason to be that frameworks are not common to evolve with *one-to-many* or *many-to-one* replacement. However, identifing rules for *one-to-many* and *many-to-one* replacement may still be important, because migration tasks concerned with APIs involved in *one-to-many* and *many-to-one* replacement may be more difficult than other migration tasks.

Typically, there are some rules that can be identified by both *HiMa* and *AURA*. To analyze the relationships

Table V
CONTRIBUTIONS OF THE MAIN TECHNIQUES TO THE OVERALL EFFECTIVENESS

Framework& Task	Without Comment			Without Caller			Without Callee			Without Mapping			*HiMa* (ALL)		
	Ide.	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*	*Ide.*	*Cor.*	*Pre.*
jEdit&Task1	432	384	88.9	436	322	73.9	432	402	93.1	432	402	93.1	432	416	96.3
jEdit&Task2	1057	1042	98.6	1057	929	87.9	1057	859	81.3	1057	995	94.1	1057	1043	98.7
Struts&Task1	184	173	94.0	184	182	98.9	184	183	99.5	184	183	99.5	184	183	99.5
Struts&Task2	84	84	100.0	84	84	100.0	84	66	78.6	84	84	100.0	84	84	100.0
Mime4J&Task1	208	202	97.1	208	193	92.8	208	189	90.9	208	205	98.6	208	205	98.6
Mime4J&Task2	98	97	99.0	98	96	98.0	98	97	99.0	98	98	100.0	98	98	100.0

between rules identified by *HiMa* and rules identified by *AURA*, we further provide a detailed comparison of rules identified by *HiMa* and *AURA* in Table IV. In this table, for each subject and each task, columns 2-3 depict the number of rules identified by both *HiMa* and *AURA*, and the number of correct rules among them; columns 4-7 depict the information for rules identified by only *HiMa* but not *AURA* and rules identified by only *AURA* but not *HiMa*. Here, we deem the 10 correct rules that *HiMa* and *AURA* classified into different types as correctly identified by both *HiMa* and *AURA*. From Table IV, we have the following observations.

First, the majority of correct rules can actually be identified by both *HiMa* and *AURA*. This observation thus confirms the effectiveness of both approaches. Second, very few correct rules are identified by *AURA* but not by *HiMa*, while there are quite some correct rules identified by *HiMa* but not by *AURA*. This observation indicates that, although *HiMa* and *AURA* can complement each other in practice, *HiMa* should be preferred if only one approach can be used.

2) RQ2: Contributions of the Four Main Techniques: To compare the five combinations of the four main techniques in our *HiMa* approach, each column in Table V depicts the results of one combination. For example, "Without Comment" denotes *HiMa* without using *comment analysis* and "*HiMa* (ALL)" denotes *HiMa* itself. From Table V, we have the following observation concerning the contributions of the main techniques in our *HiMa* approach.

HiMa without using either *comment analysis*, *callee analysis*, *caller analysis*, or *mapping conditions* would typically identify fewer evolution rules that can be correctly identified by *HiMa* using all the techniques in some circumstances. This observation indicates that all of the techniques in *HiMa* are useful. Furthermore, as turning off any of the four techniques in *HiMa* may not always result in significant decrease in the number of correctly identified evolution rules, the effects of the four techniques may somehow overlap with each other for a specific framework.

It should be noted that the need for using all the four techniques in *HiMa* actually lies in the way for *HiMa* to use them. As *HiMa* uses all the four techniques to match each pair of consecutive revisions, slight loss in accuracy induced in matching some pair of revisions may affect the overall effectiveness. Thus, it is preferable for us to employ multiple techniques to identify as many correct rules as possible for each pair of revisions. Supposing that we have one rule for changing revision R_1 to revision R_2 in the form of "method A replaced by method B" and one rule for changing revision R_2 to revision R_3 in the form of "method B replaced by method C", failure to identify either of the two rules would result in failure to identify the rule "method A replaced by method C" for changing revision R_1 to revision R_3.

Table VI
EXECUTION TIME (IN MINUTES)

Framework	Task 1			Task 2		
	#Rev.	*HiMa*	*AURA*	*#Rev.*	*HiMa*	*AURA*
jEdit	586	88.89	0.61	2672	208.06	1.81
Struts	1074	156.62	1.48	415	100.88	0.76
Mime4J	185	26.88	0.44	25	2.70	0.34

C. Performance

Table VI lists the execution time of *HiMa* and *AURA* for the six tasks. Columns headed by "*#Rev.*" depict the numbers of pairs of revisions used by *HiMa*. From this table, we can observe that the execution time of *HiMa* is always much longer than that of *AURA*. In particular, *HiMa* is typically 100-150 times more costly than *AURA* computationally. We suspect the reason to be that *HiMa* needs to analyze too much more revisions than *AURA*. For example, for the first task on jEdit, *HiMa* analyzed 586 pairs of revisions but *AURA* analyzed only one pair of revisions. As the aim of framework-evolution identification is to facilitate the migration of client code and one framework may have a large number of clients, the extra computational cost of *HiMa* would not be a big burden for its application. Furthermore, as the execution time of *HiMa* may be more sensitive to the number of revisions, *HiMa* may not be significantly less scalable than *AURA* in response to the increase of the framework size. Of course, more experiments on large frameworks are necessary to further investigate *HiMa*'s scalability.

D. Threats to Validity

1) Construct Validity: Threats to construct validity are concerned with whether the measurement in the study reflects real-world situations. The main threat to construct validity in our study is the way of validating identified rules. As we adopted a manual process to check whether each identified rule to be correct, errors made in this manual process would bias our experimental results. To reduce this threat, four human inspectors carefully determined the correctness of the identified rules ensuring each rule inspected by at least two inspectors. Note that existing studies (e.g., M. Kim et al. [17], Schäfer et al. [7], and Wu et al. [5]) typically adopt manual inspection of identified rules to evaluate approaches to framework-evolution identification.

2) Internal Validity: Threats to internal validity are concerned with the uncontrolled factors that may also be responsible for the experimental results. In our study, the main threat to internal validity is the possible faults in the implementation of our approach. To reduce this threat, we reviewed all the code before conducting the study.

3) External Validity: Threats to external validity are concerned with whether the experimental results are generalizable for other situations. In our study, the main threat to external validity lies in the representativeness of our subjects. To reduce this threat, we chose three subjects (including two subjects used in previous research [17], [5]). However, as we experimented with frameworks written in only Java using only SVN as the version-control system, our experimental results may be specific to Java and SVN. Further reduction of this threat may require evaluation of our approach on frameworks written in other languages using other version-control systems. Note that our approach requires the version-control system to support locating each framework revision.

V. Discussion

In this section, we discuss the following issues related to our *HiMa* approach: the strength and weakness of using revisions and comments, the ability to identify plentiful types of rules, the ability to avoid thresholds, and the need to assist client migration.

Use of revisions and comments. The main strength of using revisions is that the differences between two consecutive revisions are much smaller than the differences between two releases. Thus, rule identification between two consecutive revisions could be more accurate than rule identification between two releases. The main strength of using comments is that the information in comments is orthogonal to information in source code. Thus, conceding NLP's inaccuracy, we are able to use source code of revisions to validate and refine raw rules identified from comments. The main weakness of using revisions and comments is that the abundance of information may incur too much computational cost to use deep code analysis. However, our empirical results indicate that shallow analysis on abundance of information may still be superior to deep analysis on scarceness of information.

Automatic identification of plentiful types of rules. In the literature, researchers (e.g., Wu et al. [5]) have demonstrated that it is necessary to automatically identify the four types (i.e., *simply-deleted*, *one-to-one*, *one-to-many*, and *many-to-one*) of evolution rules. In fact, *AURA* is the first approach that satisfies this requirement. Our *HiMa* approach is able to not only satisfy this requirement but also outperform *AURA* for each type of rules. In fact, *HiMa* can also identify *many-to-many* rules, which *AURA* cannot identify.

Avoiding threshold tuning. According to Wu et al. [5], not using thresholds is also a key feature of approaches to framework-evolution identification, because the use of thresholds implies tedious tuning for different contexts. Among all the existing automatic matching-based approaches, *AURA* is the only approach that does not use any threshold. In particular, *AURA* employs a procedure to calculate the confidence of each identified rule and keeps only rules with 100% confidence. Unlike *AURA*, as differences between two consecutive revisions are very small, *HiMa* is able to naturally avoid using any threshold by just keeping rules that satisfy our strict criteria.

Assisting client migration. It should be noted that the final goal of framework-evolution identification is to automate the migration of client code. To achieve this goal, it is necessary to further identify framework-evolution-related usage patterns. In fact, Zhong et al. [34] and Nyuyen et al. [14] have investigated techniques that may help obtain such patterns.

VI. Conclusion and Future Work

In this paper, we have proposed a novel approach named *HiMa* to identification of framework evolution. The distinctive feature of our *HiMa* approach is to match each pair of consecutive revisions in the evolution history of the framework and aggregate revision-level rules to form evolution rules between the two releases. To evaluate our *HiMa* approach, we empirically compared *HiMa* with *AURA* using three Java frameworks as subjects. Our empirical results indicate that, for both precision and recall, *HiMa* is superior to *AURA* in most circumstances and is never inferior to *AURA* in any circumstances. Due to the use of a large number of revisions, *HiMa* is computationally more costly than *AURA*. However, the higher cost in computation would not be an inhibitive factor for the application of *HiMa*.

In future work, we plan to investigate the following issues. First, we plan to implement *HiMa* for other languages and other version-control systems, and conduct further experiments to address the main threats to the validity of our experimental study. Second, as client code provides a source of information not utilized in our approach, we plan to incorporate analysis of client code into our approach to further improve the accuracy of our approach. Finally, we plan to further investigate optimization of the matching between each pair of revisions so as to incorporate deeper analysis for higher accuracy.

Acknowledgments

This research is sponsored by the National 973 Program of China No. 2009CB320703 and the Science Fund for Creative Research Groups of China No. 61121063. We thank the authors of *AURA* for providing their implementation of *AURA* publicly. We thank Mr. Chun Zhang for his valuable suggestions for this research, and Mr. Tian Lan and Mr. Muyao Zhu for their assistance in our empirical evaluation.

REFERENCES

[1] K. Chow and D. Notkin, “Semi-automatic update of applications in response to library changes,” in *Proc. ICSM*, 1996, pp. 359–368.

[2] P. Steyaert, C. Lucas, K. Mens, and T. D’Hondt, “Reuse contracts: Managing the evolution of reusable assets,” in *Proc. OOPSLA*, 1996, pp. 268–285.

[3] I. Balaban, F. Tip, and R. Fuhrer, “Refactoring support for class library migration,” in *Proc. OOPSLA*, 2005, pp. 265–279.

[4] D. Dig and R. Johnson, “How do APIs evolve? A story of refactoring,” *JSME*, vol. 18, no. 2, pp. 83–107, March 2006.

[5] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim, “AURA: A hybrid approach to identify framework evolution,” in *Proc. ICSE*, 2010, pp. 325–334.

[6] J. Henkel and A. Diwan, “Catchup!: Capturing and replaying refactorings to support API evolution,” in *Proc. ICSE*, 2005, pp. 274–283.

[7] T. Schäfer, J. Jonas, and M. Mezini, “Mining framework usage changes from instantiation code,” in *Proc. ICSE*, 2008, pp. 471–480.

[8] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, “Refactoring-aware configuration management for object-oriented programs,” in *Proc. ICSE*, 2007, pp. 427–436.

[9] C. Kemper and C. Overbeck, “What’s new with JBuilder,” in *Proc. JavaOne*, 2005.

[10] G. Malpohl, J. J. Hunt, and W. F. Tichy, “Renaming detection,” *ASEJ*, vol. 10, no. 2, pp. 183–202, April 2003.

[11] M. W. Godfrey and L. Zou, “Using origin analysis to detect merging and splitting of source code entities,” *IEEE TSE*, vol. 31, no. 2, pp. 166–181, February 2005.

[12] S. Kim, K. Pan, and E. J. Whitehead, Jr., “When functions change their names: Automatic detection of origin relationships,” in *Proc. WCRE*, 2005, pp. 143–152.

[13] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, “Automated detection of refactorings in evolving components,” in *Proc. ECOOP*, 2006, pp. 404–428.

[14] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, “A graph-based approach to API usage adaptation,” in *Proc. OOPSLA*, 2010, pp. 302–321.

[15] G. Antoniol, M. D. Penta, and E. Merlo, “An automatic approach to identify class evolution discontinuities,” in *Proc. IWPSE*, 2004, pp. 31–40.

[16] Z. Xing and E. Stroulia, “API-evolution support with Diff-CatchUp,” *IEEE TSE*, vol. 33, no. 12, pp. 818–836, December 2007.

[17] M. Kim, D. Notkin, and D. Grossman, “Automatic inference of structural changes for matching across program versions,” in *Proc. ICSE*, 2007, pp. 333–343.

[18] K. Taneja, D. Dig, and T. Xie, “Automated detection of API refactorings in libraries,” in *Proc. ASE*, 2007, pp. 377–380.

[19] S. Demeyer, S. Ducasse, and O. Nierstrasz, “Finding refactorings via change metrics,” in *Proc. OOPSLA*, 2000, pp. 166–177.

[20] P. Weißgerber and S. Diehl, “Identifying refactorings from source-code changes,” in *Proc. ASE*, 2006, pp. 231–240.

[21] B. Dagenais and M. P. Robillard, “Recommending adaptive changes for framework evolution,” in *Proc. ICSE*, 2008, pp. 481–490.

[22] S. Thummalapenta and T. Xie, “SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web,” in *Proc. ASE*, 2008, pp. 327–336.

[23] P. Sawyer, P. Rayson, and R. Garside, “REVERE: Support for requirements synthesis from documents,” *Information Systems Frontiers*, vol. 4, no. 3, pp. 343–353, March 2002.

[24] M. Asahara and Y. Matsumoto, “Extended models and tools for high-performance part-of-speech,” in *Proc. COLING*, 2000, pp. 21–27.

[25] A. Fantechi, S. Gnesi, G. Lami, and A. Maccari, “Applications of linguistic techniques for use case analysis,” *Requirement Engineering*, vol. 8, no. 3, pp. 161–170, March 2003.

[26] M. J. Collins and T. Koo, “Discriminative reranking for natural language parsing,” *Computational Linguistics*, vol. 31, no. 1, pp. 25–70, March 2005.

[27] L. Kof, “Scenarios: Identifying missing objects and actions by means of computational linguistics,” in *Proc. RE*, 2007, pp. 121–130.

[28] D. Shepherd, Z. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, “Using natural language program analysis to locate and understand action-oriented concerns,” in *Proc. AOSD*, 2007, pp. 212–224.

[29] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, “/* iComment: Bugs or bad comments? */,” in *Proc. SOSP*, 2007, pp. 145–158.

[30] L. Tan, Y. Zhou, and Y. Padioleau, “aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs,” in *Proc. ICSE*, 2011, pp. 11–20.

[31] H. Zhong, L. Zhang, T. Xie, and H. Mei, “Inferring resource specifications from natural language API documentation,” in *Proc. ASE*, 2009, pp. 307–318.

[32] Z. Kozareva, Ó. Ferrández, A. Montoyo, R. Muñoz, A. Suárez, and J. Gómez, “Combining data-driven systems for improving named entity recognition,” *DKE*, vol. 61, no. 3, pp. 449–466, June 2007.

[33] S. L. Abebe and P. Tonella, “Natural language parsing of program element names for concept extraction,” in *Proc. ICPC*, 2010, pp. 156–159.

[34] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, “Mining API mapping for language migration,” in *Proc. ICSE*, 2010, pp. 195–204.