EPM–RT–2013-05

HALO : A MULTI-FEATURE TWO-PASS ANALYSIS TO IDENTIFY FRAMEWORK API EVOLUTION

Wei Wu, Yann-Gaël Ghéhéneuc, Giuliano Antoniol,
Miryung Kim
Département de Génie informatique et génie logiciel
École Polytechnique de Montréal

Mai 2013

Poly

ÉCOLE POLYTECHNIQUE MONTRÉAL

EPM-RT-2013-05

HALO: A MULTI-FEATURE TWO-PASS ANALYSIS TO IDENTIFY FRAMEWORK API EVOLUTION

Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Miryung Kim*
Département de génie informatique et génie logiciel
École Polytechnique de Montréal
The University of Texas at Austin*

Mai 2013

Dépôt légal :
Bibliothèque nationale du Québec, 2010
Bibliothèque nationale du Canada, 2010

EPM-RT-2013-5
HALO : A Multi-Feature Two-Pass Analysis to Identify Framework API Evolution
par : Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, Miryung Kim
Département de génie informatique et génie logiciel
École Polytechnique de Montréal

HALO: A Multi-Feature Two-Pass Analysis to Identify Framework API Evolution

Wei Wu, *Member, IEEE,* Yann-Gaël Guéhéneuc, *Member, IEEE,* Giuliano Antoniol, *Member, IEEE,* and Miryung Kim, *Member, IEEE*

Abstract—Software frameworks and libraries are indispensable to today's software systems. Because of the fast development of open-source software in recent years, frameworks and libraries have became much versatile as any open-source system or part thereof can be used as a framework (or a library). Developer can reuse frameworks in innovative ways that are not expected by the providers of frameworks. Many frameworks are not well documented and very few owners provide specific documents to describe the changes between different releases of their frameworks. When they evolve, it is often time-consuming for developers to keep their dependent code up-to-date. Approaches have been proposed to lessen the impact of framework evolution on developers by identifying API evolution or change rules between two releases of a framework. However, the precision and recall of the change rules generated by these approaches depend on the features that they use, such as call-dependency relations or text similarity. If these features do not provide enough information, the approaches can miss correct change rules and compromise the precision and recall. For example, if a method in the old release of a framework is not called by other methods, we cannot find its change rule using call-dependency relations alone. Considering more features can overcome this limitation. Yet, because many features may also give contradictory information, integrating them is not straightforward. We thus introduce Halo, a novel hybrid approach that uses multiple features, including call dependency relations, method documentations, inheritance relations, and text similarity. Halo implements a two-pass analysis inspired by pattern classification problem. We implement Halo in Java and compare it with four state-of-the-art approaches. The comparison shows that, on average, the recall and the precision of Halo are 43% and 5% higher than that of other approaches.

Index Terms—Software evolution, call dependency analysis, text similarity analysis, inheritance relation analysis, empirical study

1 Introduction

Software frameworks and libraries are widely used for cost reduction. Without loss of generality, we use the term "framework" to represent frameworks, libraries in this article. When these framework evolve to fix bugs, patch security vulnerabilities, 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 to the framework continue to work with the new release [1]. In practice, the API syntax and semantics change [2], [3], [4]. 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, including compiling errors, runtime errors if the deleted method is invoked using reflection.

In theory, changes to private or internal APIs should not impact legit client programs; but they do in practice for three reasons. First, visibility mechanisms of most programming languages are limited and force developers to promote the visibility of methods to "public", which should otherwise be hidden. This limit is typically encountered when developing frameworks in Java. Classes in the "internal" packages of the framework must be "public" so that they are visible to other framework classes. Being "public" make them also wrongly visible to clients of the framework. A means to work around such limitation is to use frameworks with more fine-grain visibility mechanisms, such as OSGi. Second, the official public API can change, with some public methods becoming hidden and some private methods becoming public. Third, with open-source frameworks, developers can promote any method to the status of public API by changing its visibility and recompiling the necessary classes. Consequently, we consider that any method *could* be part of the API of a framework and, in the following, will use the term API to designate any method. (Considering any method as part of the potential API of a framework also allows us to discover more interesting change rules, which can be subsequently filtered to show only rules pertaining to public methods.)

To prevent backward-compatibility problems, developers may delay or avoid using a new release. Yet,

- *Wei Wu, Yann-Gaël Guéhéneuc, and Giuliano Antoniol are with the Department of Computer engineering and Software engineering at École Polytechnique de Montréal, Canada. E-mail: {wei.wu,yann-gael.gueheneuc}@polymtl.ca,antoniol@ieee.org*
- *Miryung Kim is with the department of Electrical and Computer Engineering at University of Texas at Austin. E-mail: miryung@ece.utexas.edu*

if they want to benefit from new features or security patches, they must evolve their programs. However, many frameworks are not sufficiently documented, especially when it comes to the changes between two releases and the rules to adapt programs from an older release to a new one. For example, Google releases Android API difference reports[1] regularly, but the Android API difference reports are rarely sufficient, as we describe in Section 5.1. Generally speaking, upgrading to new releases of frameworks requires significant developers' effort due to changes in APIs. Developers must dig into the documents and the source code of the new and previous releases of the frameworks to understand their differences and to make their programs compatible with the new releases. The longer they delay the upgrading, the more time consuming it is, because there will be more changes to adapt.

Furthermore, different from the code-scavenging described by Krueger [5], which is the practice of coping/pasting/understanding/modifying small pieces of code fragments, many frameworks, such as those built by the Eclipse Foundation[2] and Apache Software Foundation[3], are large and complex. Developers may use them without fully understanding them. Also, because these frameworks are open source, developers can reuse these frameworks in innovative ways that are not anticipated by their providers. A recent example is Amazon, which developed Kindle Fire based on Android to compete with Google and is free to modify Android as it sees fit. These two practices of using frameworks make upgrading to the new releases of frameworks difficult.

Consequently, many approaches have been developed to ease the framework API evolution process and reduce the developers' effort of adapting their programs to new releases of frameworks. Some approaches require that the framework developers do additional work, such as providing explicit change rules with annotations [6], or that they record API updates to the framework [7], [8], [9]. However, framework developers may not be able or willing to build change rules manually or use specific tools.

To avoid the extra work for framework developers, some approaches automatically identify API evolution or *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, such as [10], [11], [12], [13]. Target and replacement methods are described in terms of their signatures: return type, declaring module, method name, and formal parameters.

Besides changes to the public API, these approaches also report change rules pertaining to internal APIs that are often used by both maintainers and external users of frameworks. Maintainers can face internal API evolution problem with their own frameworks. For example, when maintainers are newcomers or must maintain code written by others. Internal APIs are used by the external users of frameworks as well. For example, Businge *et al.* investigated 1,873 versions of 512 Eclipse third-party plug-ins and discovered that 44% of them use internal APIs [14]. In their survey of the use of Eclipse internal APIs, 70% of the developers declared using Eclipse internal APIs and only 3.3% of the developers always followed Eclipse Provisional API Guidelines [15]. Moreover, Robillard and DeLine [16] showed that understanding internal APIs help developers to know better how to use the public APIs. Furthermore, internal APIs usually are less or non-documented. Thus, including internal API change rules can help developers.

However, the precision and recall of the change rules generated by these approaches depend on the features that they use, such as call-dependency relations or text similarity. If these features do not provide enough information, the approaches can miss correct change rules and report wrong ones, which compromises their precision and recall. For example, if a method in the old release of a framework is not called by other methods, we cannot find the related change rule using call-dependency relations. Intuitively, considering more features can overcome this limitation, but it is complicated to integrate them because different features may give contradictory information. For one target method, call-dependency analysis and text similarity analysis could suggest different replacement methods. This contradictory information might offset the possible improvement brought by integrating multiple features.

While studying the state of the art on framework API evolution, we observed that this problem is an instance of binary pattern classification problem [17]. Indeed, solving this problem requires classifying the methods in the new release as the replacement of a target method in the old release, according to the patterns described by various features of the target methods. Therefore, we can use existing pattern classification approaches to improve the precision and recall of generated change rules.

In this paper, we propose a novel approach Halo that uses call dependency relations, method documentations, inheritance relations, and text similarities as features. Halo also automatically generates feedback data to further improve the precision and recall. Similar techniques are used to solve general pattern classification problems [18], [19] and other software engineering problems [20], [21].

Thus, the main contributions of this work are:

1) Bring pattern classification vocabulary/techniques to framework API evolution.
2) Show that consideration of multiple features

1. http://developer.android.com/sdk/api_diff/8/changes.html
2. http://www.eclipse.org/
3. http://www.apache.org/

yields higher precision and recall.
3) Analyze the impact of different sub-components of Halo on its overall precision.

The differences between Halo and our previous approach AURA [13] are that:

1) Halo considers four features: call dependency relations, method signature text-similarities, method documentations, and inheritance relations, while AURA uses the first two only.
2) Halo implements a two-pass analysis approach using feedback to improve precision, while AURA has only one pass.
3) We also expand our original evaluation with the Android SDK data set in this paper.

Using a detailed manual evaluation on five open-source systems written in Java: JEdit, JHotDraw, JFreeChart, Struts, and Android SDK, we find that Halo improves, on average, recall by 43% and precision by 5% in comparison to three previous approaches: M. Kim *et al.*'s [11], Schäfer *et al.*'s [12] approaches, and AURA [13].

We present the effect sizes of the improvements of Halo in precision and recall in comparison to the three previous approaches with Cliff's d [22]. The results show that, for recall, the improvements of Halo are large in comparison to M. Kim *et al.*'s, Schäfer *et al.*'s approaches and AURA; for precision, Halo has a large improvement in comparison to Schäfer *et al.*'s approach and AURA, and a small decrease in comparison to M. Kim *et al.*'s approach. Cliff's d values confirm that the average recall and precision of Halo are better than those of the previous approaches.

We also study the influences of the new components of Halo on the precision and recall of the generated change rules. The detailed results are presented and discussed in Section 5.5.

To verify the scalability of Halo, we apply Halo on two Eclipse subsystems (JDT core and JDT UI) between releases 3.1 and 3.3. The numbers of methods of the two releases of JDT core and JDT UI are 35,439 and 47,237 respectively, larger than the number of all the methods in the five medium size systems that we evaluated. The analysis takes five hours on CentOS 5.5 with Intel Xeon 16-Core 2.4GHz and 48GB RAM.

We also compare the results on Eclipse JDT core and JDT UI with those of SemDiff [10] and AURA [13]. Because of the large sizes of the two Eclipse subsystems, it is impractical to manually evaluate all the change rules of the two systems. We compare Halo with SemDiff and AURA on the sample set of target methods used by [10]. We describe the comparison results in Section 4. The results show that the approximated precision of Halo is 100% same as SemDiff and that of AURA is 92.86%. However, SemDiff is, unlike Halo, semi-automatic and thus developers must choose the correct replacement methods from a list that SemDiff generates.

In the remainder of this paper, Section 2 presents a motivating scenario. Section 3 describes our approach. Section 4 evaluates it on five real-world systems. Section 5 discusses open issues. Section 6 presents related work, and Section 7 concludes this paper with future work.

2 Halo – Usage Scenario

A typical scenario of using Halo to help framework API evolution identification is as follows. Steve is a Java developer and he uses two frameworks, `Summer` 1.5 and, `Autumn` 2.0, in one of his program, `Season`. He finds out that there is an important bug fix in `Summer` 1.6 and a new feature that he likes in `Autumn` 3.0. He thus decides to upgrade `Season` to use the new releases of the two frameworks. When he compiles `Season` with the two new releases, he observes that there are 18 compilation errors. All of them are due to some methods in the old releases not existing in the new one, eight in `Summer` and ten in `Autumn`.

On the one hand, Steve checks the web site of the providers of the two frameworks and notices that the development team of `Summer` generated change rules between v1.5 and v1.6 with Halo, verified them, and put the list of the change rules on their web site. Steve then searches the list and obtains their replacements in a couple minutes.

On the other hand, the team of `Autumn` provides neither change rules between v2.0 and v3.0 nor documentation. Steve knows Halo and analyses the source code of the two releases with it. Then, he checks the replacement methods suggested by the change rules related to the compilation errors and observes that eight change rules are correct and two are wrong. Yet, he can fix the eight compilation errors using the correct change rules right away and explores the two remaining compilation errors in the source code of v2.0 and v3.0.

Consequently, it takes more time for Steve to upgrade `Autumn` than `Summer` with Halo, yet he does not have to solve all compilation errors by digging into the source code manually. Thus, Halo saves his time. In the following, we describe the inner-working of Halo and concrete examples from six open-source systems that demonstrate the advantage of Halo in Section 5.

3 Halo – Approach

We observed that the framework API evolution identification problem is a binary pattern classification problem. Given a target method in the old release of a framework, we want to classify the methods in the new release as being its replacement or not. We use the features of the target method, such as call-dependency relations, text similarities, etc., for the classification. In the scope of Halo, target methods are the methods existing in the old release but not

in the new one in terms of their signatures: return type, declaring module, method name, and formal parameters.

Based on the assumption made by Dagenais and Robilliard [10]: that a target method is simply deleted or replaced by one or more methods in the new release, we describe the framework API evolution identification problem formally as follows.

3.1 Formulation

Let $V_1, \ldots V_N$ be the evolution history of a framework where V_i is its i-th release. Given two framework releases V_i and V_j with $j > i$, the goal of our approach is to model and resolve the problem of identifying API change rules between releases i and j.

A release V_i is modeled as a set of methods $V_i = \{m_{1,i}, \ ... \ , \ m_{n,i}\}$ where $m_{r,i}$ represents the method m_r in release V_i and n is the number of methods in that release. A method $m_{r,i}$ is defined as $m_{r,i} =< ID_{m_{r,i}}, F(m_{r,i}) >$, where $ID_{m_{r,i}}$ is the method identity (a unique identifier) of the method $m_{r,i}$ and $F(m_{r,i})$ is a set of features describing $m_{r,i}$. In Java programs, the method identity $ID_{m_{r,i}}$ can be the fully qualified name of a method. The set of features $F(m_{r,i})$ may contain call dependencies, inheritance relations, method signatures, method documentations, and so on.

Between V_i and V_j, methods are added, deleted, and modified. We must identify replacement methods for those methods in V_i but no longer in V_j in terms of their signatures including return value, declaring module, method name, and formal parameters. Let us assume that the function $S(m_{r,i})$ returns the signature of method $m_{r,i}$ and we choose $S(m_{r,i})$ as the $ID_{m_{r,i}}$ in our work. Then, the target methods set T is:

$$T = \{m_{r,i} | \nexists m_{s,j} \in V_j : S(m_{r,i}) = S(m_{s,j})\} \quad (1)$$

By definition, methods in T do not have any counterpart in V_j with identical feature sets. They must have different signatures at least.

The problem can thus be presented as binary classification of the elements in $T \times V_j$ into subsets. Each method t in T is replaced by the methods in a subset of V_j or ϕ. Indeed, it is possible that some methods in V_j do not replace any method in T.

To classify the elements in $T \times V_j$ into subsets, we compare the similarity of the features between methods in V_j and methods in T. We define $F(m) \sim F(t)$ to mean that m is similar to t considering the feature set. For any given method t in T, we compute the replacement subset $R(t)$:

$$R(t) = \{m_{s,j} | m_{s,j} \in V_j : F(m_{s,j}) \sim F(t)\} \quad (2)$$

It is also possible that there is no method to replace t, therefore $|R(t)| \geq 0$

Besides the feature set $F(m_{s,j})$, we also use some feedback data, FD, to make the similarity computation more accurate. For example, if Class B is already known as the replacement of Class A, this information can be provided as feedback to calculate the similarities between the methods in A and B. The feedback data are optional and can be generated automatically or provided by developers.

Based on the formulation above, we extended our previous work AURA [13] to further improve the precision and recall of the generated change rules. We name the new approach Halo. In the following presentation of each component (Figure 2 and Figure 3), we first give a general introduction of the components and then describe its details formally.

3.2 Software Artifacts and Features

Halo takes the source code of two releases of a framework as inputs. We do not use binary code (such as Java bytecode) because we also leverage method documentations in source code. If method documentations are not available, our approach still can generate change rules but, possibly, with reduced precision and recall. As with previous work by Dagenais and Robillard [10] and Schäfer et al. [12], our approach naturally considers instantiation code (test cases or client programs) if they are provided with some release of the framework source code.

To balance accuracy and performance, we assume that all replacement methods are taken from all the methods existing in the new release of the framework to be analyzed 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 the Java Foundation Classes (JFC) do not. We include the methods from the frameworks provided by the same vendor only, because developers may move methods between these frameworks.

Besides call-dependency relations and method signature text-similarities, which are used in AURA, we also use method documentations and inheritance relations.

The set $F(t)$ of Halo is made by four features:

- call dependencies or $CD(t)$
- method signatures or $S(t)$
- method documentations or $C(t)$
- inheritance relations or $I(t)$

In a nutshell, we model methods with a 4-dimensional tuple, where each element is a structured feature (e.g., an inheritance tree).

3.3 Method Similarity

To compute the similarities between a method in the new release and a target method, we first compute

their call-dependency similarity, method documentation similarity, inheritance similarity, and method signature similarity and combine method signature, inheritance, and method documentation similarity into method definition text similarity (defined later in this section). Then, we use the algorithm presented in Section 3.4 to rank all the methods in the new release according to the similarities.

We now formally explain how the features $F(x)$ of a method x in a new release V_j extracted from the framework source code are used to compute the similarity between x and a target method t in T. To instantiate the concept of $\sim$, we define $\sigma(x,t)$ to denote the similarity between method x and t. Thus:

$$F(x) \sim F(t) | x \in V_j, \nexists y \in V_j, \sigma(y,t) > \sigma(x,t)$$

With respect to the four features considered in Halo, we define the specific similarities $\sigma_{CD}(x,t)$, $\sigma_D(x,t)$, $\sigma_C(x,t)$, and $\sigma_I(x,t)$. Method similarity $\sigma(x,t)$ is a function of the four specific similarities:

$$\begin{aligned}\sigma(x,t) &= \psi(x,t) \\ &= \psi(\sigma_{CD}(x,t), \sigma_D(x,t), \sigma_C(x,t), \sigma_I(x,t))\end{aligned}$$

Strict Anchor Set A_s: Before presenting our concrete implementations of the elements of $\sigma(x,t)$, we introduce the concept of Strict Anchor Set A_s. A_s consists of the known method pairs between two releases of a framework V_i and V_j, *i.e.*, the stable parts when the framework evolves from V_i to V_j.

A_s is important to our algorithm to compute the call-dependency similarity $\sigma_{CD}(x,t)$. If A_s does not change, Halo completes the call-dependency analysis and starts using other features to detect the other change rules. Our algorithm is presented in detail in Section 3.4.

To compute A_s, we first define an *anchor* a as either (1) a pair of methods with the same signature (including return type, declaring module, name, and parameter lists) that exist in both the old and new releases or (2) a pair of methods already identified as target and replacement methods. The set of anchors A is defined as:

$$\begin{aligned}a &= < a_i,\ a_j > \wedge\ j > i \\ A &= \{a \mid (a_i \in V_i \\ &\quad \wedge\ \ a_j \in V_j \wedge S(a_i) = S(a_j))\ \cup\ known\ < a_i, a_j >\}\end{aligned}$$

A_s is initialized by A, as in AURA. Then, we use a stricter criteria to refine A to A_s in Halo. If the implementations of two methods in an anchor are dramatically changed, regular call-dependency analysis will more likely generate incorrect change rules because the call relations are churned in the new release. To avoid such incorrect change rules, we compute the difference of the implementations between the two methods in an anchor as the number of methods that they call. Thus, num_{a_i} and num_{a_j} are the numbers of methods called by the old (a_i) and the new releases (a_j) of Anchor a respectively. Only if the minimum value of num_{a_i} and num_{a_j} is greater than the difference between num_{a_i} and num_{a_j}, we add a to A_s. The set of strict anchors A_s is defined as:

$$\begin{aligned}A_s &= \{a \mid a \in A \\ &\quad \wedge\ \ |num_{a_i} - num_{a_j}| < min(num_{a_i}, num_{a_j})\}\end{aligned}$$

Although simplistic, this condition on anchors works well in practice as discussed in Section 5. Then, in a way similar to the fixed point algorithm [23], Halo incrementally updates A_s while discovering new change rules, until the A_s set does not change.

Call Dependency Similarity $\sigma_{CD}(x,t)$: Call dependency analysis discovers the calls between framework APIs and the methods using them. These calls reflect the behavior of frameworks more accurately than text similarity.

There are different implementations of call-dependency analyses and ours is based on association rule mining [24] which uses Confidence Value (CV) to measure the connections between the target methods and their possible replacements. SemDiff [10] and Schäfer *et al.*'s approach [12] are also based on a similar idea.

Based on A_s, we compute the CV for a given target method *t* and its candidate replacement method *x* as:

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

$$\begin{aligned}\mathbf{A}(t) &= |\{\ a \mid a\ \in\ A_s\ \wedge a_i \rightarrow t\ \}| \\ \mathbf{A}(t,x) &= |\{\ a \mid a\ \in\ A_s\ \wedge\ a_i \rightarrow t\ \wedge\ a_j \rightarrow x\ \}|\end{aligned}$$

where $a_i \rightarrow t$ represents method a_i calls method t. Then, we use CV to represent the call-dependency similarity between x and t:

$$\sigma_{CD}(x,t) = CV(x,t) \tag{3}$$

Method Documentation Similarity $\sigma_C(x,t)$: We also measure the similarity of the method documentations of two methods. First, we tokenize the method documentations as proposed by Lawrie *et al.* [25] by splitting them at upper-case letters and other characters, such underscore, space, punctuation. Then, we use the Longest Common Subsequence (LCS) to measure their similarity. Let $|S|$ represent the tokenized length of a string S, our implementation of method documentation similarity is defined as:

$$\sigma_C(x,t) = \frac{LCS(C(x),\ C(t))}{|C(t)|} \tag{4}$$

Inheritance Relation Similarity $\sigma_I(x,t)$: To compute the similarity of the inheritance trees, we first convert the whole inheritance tree of each method into a string by traversing both inheritance trees and implementation trees in lexicographic order. Using an Java example in Figure 1, if a method m belongs to a class C, the parent class tree of C is

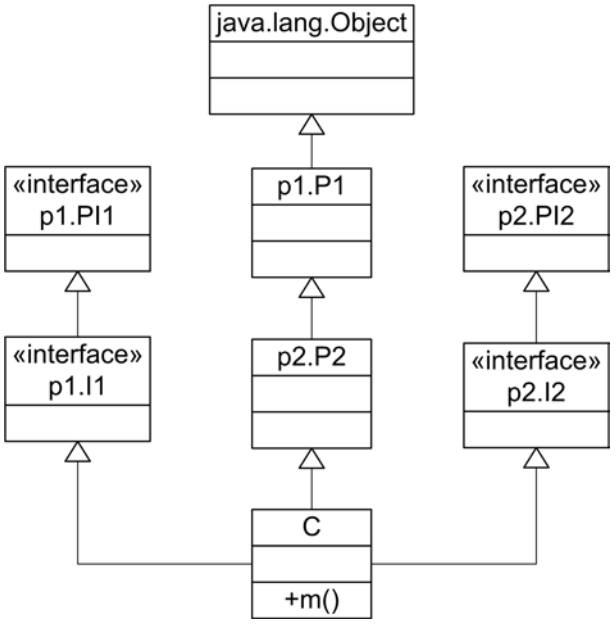

Figure 1. Inheritance Tree Example

$java.lang.Object$, $p1.P1$, and $p2.P2$. C also implements two interfaces $p1.I1$ and $p2.I2$ whose parent interfaces are $p1.PI1$ and $p2.PI2$ respectively. Then the string representing the inheritance tree of C is $p1.P1.p2.P2.p1.PI1.p1.I1.p2.PI2.p2.I2$. (In this example, we do not take $java.lang.Object$ into account, because it is the root of all Java classes and adding it has no influence on the result). Then, we use LCS between the two strings representing the inheritance trees of the classes in which the two methods are defined to measure the similarity of their inheritance relations:

$$\sigma_I(x,t) = \frac{LCS(I(x),\ I(t))}{|I(t)|} \tag{5}$$

Method Definition Text Similarity $\sigma_D(x,t)$: Besides method documentation and inheritance relation similarities, we also use three additional measurements to describe method definition similarity $\sigma_D(x,t)$:

1. Method-level similarity $MLS(x,t)$: we define functions $R(x), D(x), N(x), P(x)$ to get the signatures of the return type, declaring module, name, and formal parameters of method x respectively. The method level similarity between x and a target method t is:

$$\begin{aligned} E_x &= \{R(x), D(x), N(x), P(x)\} \\ d_i(x,t) &= \begin{cases} 0\ if\ E_x(i) \neq E_t(i) \\ 1\ if\ E_x(i) = E_t(i) \end{cases} \\ MLS(x,t) &= \frac{1}{4}\sum_{i=1}^{4} d_i(x,t) \end{aligned}$$

where $E_x(i)$ return $R(x), D(x), N(x), P(x)$ respectively while $i = 1, ..., 4$. $MLS(x,t)$ returns how many of the four elements (return types, declaring modules, names, and parameter list) of the signature of x are identical to those of t. Method-level similarity reflects coarse-grain similarity between two methods' signatures.

2. Levenshtein Distance $LD(S(x),\ S(t))$ and Longest Common Subsequence $LCS(S(x),\ S(t))$: we combine LD and LCS to compare the text-similarity between two method signatures because LD and LCS pertain to two different aspects of string comparison. LD [26] is concerned with the difference between strings but not with what they have in common, while LCS [27] focuses on their common parts but cannot 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. So, if we only use one of these measures, we obtain two methods with the same similarity value. Thus, first using LD, we narrow the candidate set to `a` and `abc`, then comparing them with LCS, we can identify that `abc` is most similar to `ab`.

Before computing the LD and the LCS of method documentations, inheritance relations, and method signatures, we also tokenize them as proposed by Lawrie *et al.* [25].

Using $MLS(x,t)$, $LD(S(x),\ S(t))$, $LCS(S(x),\ S(t))$, $\sigma_I(x,t)$, and $\sigma_C(x,t)$, our multi-stage text-similarity comparison algorithm (presented in the next paragraph) sorts the methods in V_j in descending similarity order. Thus, the implementation of method definition text similarity in Halo is:

$$\sigma_D(x,t) = 1 - \frac{POS_D(x,t)}{|V_j|} \tag{6}$$

Here, $POS_D(x,t)$ is the position of method x in the sorted V_j using our multi-stage text similarity comparison algorithm regarding to t.

Multi-stage Text-similarity Comparison: Our multi-stage text-similarity comparison algorithm decides which one between two methods x and y is more similar to a target method t using the following steps:

1) If their method level similarities to t are different ($MLS(x,t) \neq MLS(y,t)$), the one with $max(MLS(x,t), MLS(y,t))$ is more similar to t.
2) If $MLS(x,t) = MLS(y,t))$ and only one method is in the same declaring modules of t ($D(x) = D(t)$ or $D(y) = D(t)$), that one is more similar to t.
3) If $D(x) = D(y) = D(t)$ or $D(x) \neq D(t) \wedge D(y) \neq D(t)$ and there is only one method with the same method name as t ($N(x) = N(t)$ or $N(y) = N(t)$)), then Halo choose this method as being more similar to t.
4) If $N(x) = N(y) = N(t)$ or $N(x) \neq N(t) \wedge N(y) \neq N(t)$ and there is only one method with the same parameter list as t ($P(x) = P(t)$ or $P(y) = P(t)$),

then Halo choose this method as being more similar to t.

5) If $P(x) = P(y) = P(t)$ or $P(x) \neq P(t) \wedge P(y) \neq P(t)$ and the inheritance relation similarities of x and y in comparison to that of t are different ($\sigma_I(x,t) \neq \sigma_I(y,t)$), the one with $max(\sigma_I(x,t), \sigma_I(y,t))$ is more similar to t.
6) If $\sigma_I(x,t) = \sigma_I(x,t)$ and the LDs of the signatures of x and y in comparison to that of t are different ($LD(S(x),S(t)) \neq LD(S(y),S(t))$), the one with $min(LD(S(x),S(t)), LD(S(y),S(t)))$ is more similar to t.
7) If $LD(S(x),S(t)) = LD(S(y),S(t))$ and the LCS of the signatures of x and y in comparison to that of t are different ($LCS(S(x),S(t)) \neq LCS(S(y),S(t))$), the one with $max(LCS(S(x),S(t)), LCS(S(y),S(t)))$ is more similar to t.
8) If $LCS(S(x),S(t)) = LCS(S(y),S(t))$ and the method documentation similarities of x and y in comparison to that of t are different ($\sigma_C(x,t) \neq \sigma_C(y,t)$, the one with $max(\sigma_C(x,t), \sigma_C(y,t))$ is more similar to t.
9) If $\sigma_C(x,t) = \sigma_C(y,t)$, choose y is more similar to t arbitrarily.

Method Similarity $\sigma(x,t)$**:** We use the call-dependency, method-definition, method documentation, and inheritance-relation similarities defined above to compute the method similarity $\sigma(x,t)$ that gives the final classification results. Given a target method t, we sort the methods in release V_j in descending similarity order. The method similarity between method x in V_j and a target method t is defined as:

$$\sigma(x,t) = \psi(x,t) = 1 - \frac{POS(x,t)}{|V_j|}$$

$POS(x,t)$ is the position of method x in the sorted V_j regarding to t.

Halo implements the sorting algorithm in a binary classifier (presented in Section 3.4) to decide if $F(x) \sim F(t)$ holds true or if it is $F(x) \not\sim F(t)$ for a target method t in T and for any method x of V_j.

3.4 Classifier

To compute $\sigma(x,t)$ as described above, we build a two-pass algorithm. The processes of the two passes are the same. The second pass uses the feedback data FD generated by the first pass as additional input to improve the precision and recall of the change rules as shown in Figure 3.

In any pattern classification problem, it is possible that some features are missing [17]. In our approach, we can always obtain signature similarity and inheritance relations for any method, but we may not have call-dependency relations and method documentations. Thus, these two features can be missing for some methods.

To handle missing features, the classification algorithm works in several steps. We first classify the target methods into four categories: (1) called by anchors and with method documentations, (2) called by anchors and without method documentations, (3) with method documentations but not called by anchors, and (4) neither called by anchors nor with method documentations. Then, it processes these categories separately. In a similar way to the fix-point algorithm used by data-flow analyses [23], if the change rules of the target methods in categories 1 to 3 satisfy the condition for strict anchor, we add them to A_s. In the next iteration, we recompute $\sigma_{CD}(x,t)$ for the remaining target methods to detect more change rules by call-dependency analysis. We now present the algorithm of a single pass and discuss later the use of feedback data FD in the second pass. The flowchart diagram of the algorithm of a single pass is shown in Figure 2. The components with dark background are the classification steps.

Step 1 - Called by Anchors: We measure the call-dependency relations between a method x in V_j and a target method t with $\sigma_{CD}(x,t)$ that is implemented by confidence value $CV(x,t)$.

Step 1.1. For each target method t, which has only one method x in V_j with 100% confidence value or $|\{x|\sigma_{CD}(x,t) = 100\% \wedge x \in V_j\}| = 1$, we choose that the method as the replacement method of t and put it to $R(t)$. After processing all ts that have only one method with 100% confidence values and updated A_s, we recompute the confidence values of the methods in V_j to the remaining target methods in T to see if there are new target methods that have only one method in V_j with 100% confidence values. If yes, we repeat this step. Otherwise, we go to Step 1.2.

Step 1.2. For each target method t that has a set of method X with 100% confidence values, *i.e.*, $X = \{x|\sigma_{CD}(x,t) = 100\% \wedge x \in V_j\}$ and $|X| > 1$. First, we check if there is a method in X with an identical method documentation as that of the target method, *i.e.*, $\sigma_C(x,t) = 1$. If there is only one method x with $\sigma_C(x,t) = 1$, x is the replacement method of t. If there are more than one or none, we choose the most similar method using the multi-stage text-similarity described in Section 3.3 as the replacement method. In this step, we also generate one-replaced-by-many rules using the same algorithm as AURA.

Step 1.3. For each target method t that has a set of method X with not 100% confidence values, *i.e.*, $X = \{x|\sigma_{CD}(x,t) < 100\% \wedge x \in V_j\}$, we choose the replacement method using our multi-stage text-similarity as the replacement method.

Step 2 - With Method Documentations but Not Called by Anchors: For each target method t not called by anchors, we check if there is a method set X in V_j with the method documentation identical to that of t, *i.e.*, $X = \{x|\sigma_C(x,t) = 100\% \wedge x \in V_j\}$. If $|X| = 1$, we choose the method as the replacement

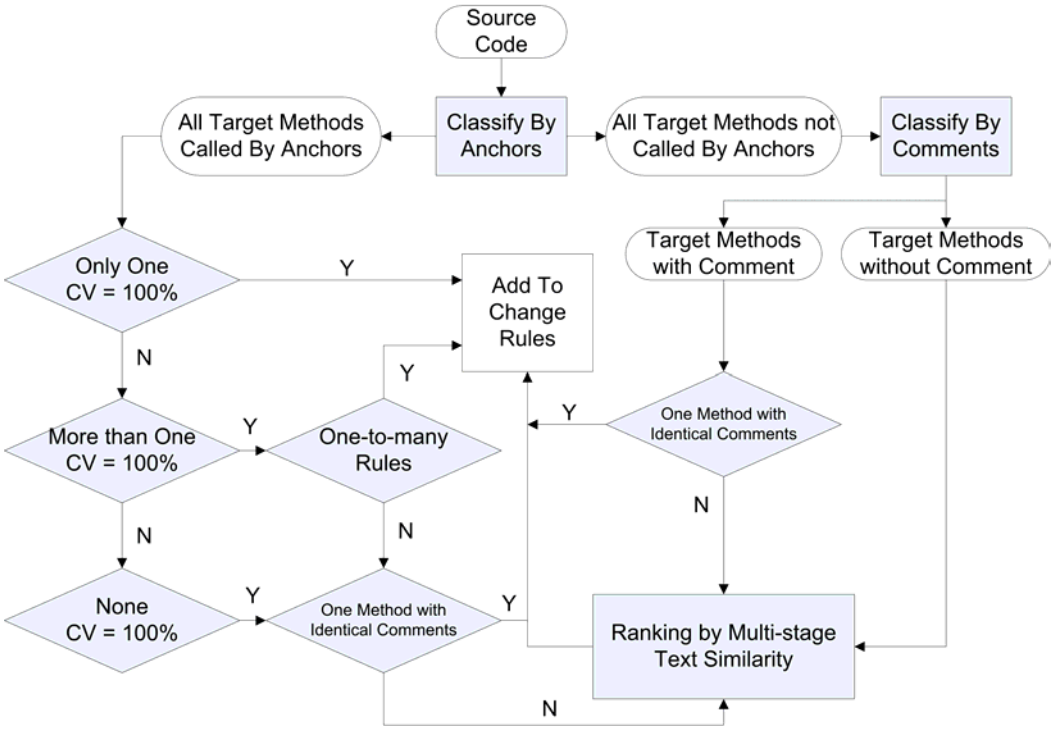

Figure 2. Halo Classifier Single Pass

method of t. If $|X| > 1$, we choose the most similar method in X using our multi-stage text-similarity as the replacement method. If this target–replacement pair satisfies the condition for strict anchor, we add them to A_s. After this step, if A_s changed, we recompute the confidence values for the remaining target methods and go back to Step 1. If the anchor set is not changed after this step, we continue to step 3.

Step 3 - Neither Called by Anchors nor With Method Documentations: For the rest of the target methods, we use our multi-stage text similarity comparison algorithm to find their replacement methods with the highest method definition similarity from V_j. Because not all the target methods have replacement methods, we examine the change rules generated in this step to detect simply-deleted methods. If a replacement method z for a target method t satisfy this condition:

$$\exists z \in V_j \mid \forall x \in V_j \ : \ \sigma(z,t) > \sigma(x,t)$$
$$\wedge\ S(z) \neq S(x) \wedge\ \exists y \in V_i \ : \ S(z) = S(y)$$

then our algorithm considers t a simply-deleted method: a target method with no replacement method in the new release. We only identify simply-deleted method rules in the last step by this heuristic: if a target methods left for this step have never been called by anchors, they do not have identical method documentations of another methods, their most similar methods by their definitions are methods existing in the old release, we classify these target methods as simply-deleted.

3.5 Feedback Data

Halo uses a two-pass analysis algorithm, which uses feedback data FD generated in the first pass as an input of the second pass. The observation behind this two-pass analysis algorithm is that methods in object-oriented program are contained in classes, which are themselves contained in packages. The intuition is that knowing that certain methods in a class (respectively, certain classes in a package) are more likely to be replaced by the methods in another class (respectively, the classes in another package), we can promote those methods in the already matched classes to possibly improve the precision of the change rules.

Before generating FD, we filter out the change rules with lower signature similarity in the output of the first pass. We apply a threshold γ to filter set of the change rules $R(t)$. We keep only the change rules with the LCS of the signatures between the replacement and target methods above γ, the filtered replacement sets $R(t)$s is denoted as $R_\gamma(t)$s:

$$\begin{aligned} R_\gamma(t) \quad &= \quad \{m | m \in R(t) : t \in T \\ &\wedge \quad \frac{LCS(S(m), S(t))}{|S(t)|} > \gamma\} \end{aligned}$$

where T is the target method set.

Previous approaches [10], [11], [12], [28] showed that using thresholds can improve precision but compromise recall. Because we only use a threshold to generate the feedback data FD, this threshold does not have a negative impact on the recall of change rules generated in the second pass.

We abstract the $R_\gamma(t)$ following M. Kim *et al.*'s algorithm [11] to summarize the method-level change rules into first-order relation logical rules, like "all the methods in class A are moved to class B except method $m1$ and $m2$". This format describes the high-level changes between two releases of a framework and we use it as our feedback data. FD is defined formally as:

$$\begin{aligned} T_{C_i} &= \{t | t \in T : t \in C_i\} \\ R_{\gamma C_i C_j} &= \{m | m \in R_\gamma(t) : t \in T_{C_i} \wedge m \in C_j\} \\ FD &= \{< C_i, C_j > | C_i \in V_i, C_j \in V_j : \\ &\nexists \; C_k \in V_j, |R_{\gamma C_i C_k}| > |R_{\gamma C_i C_j}|\} \end{aligned}$$

where C_i is a class from V_i and C_j, C_k are classes in V_j, $m \in C$ describes that method m belongs to class C.

For example, after the first pass, we can have change rules stating that six methods in class A are replaced by five methods in class B and one is replaced by a method in class C in the $R_\gamma(t)$. The high-level changes in the feedback data will be $< A, B >$, which means that "methods in A are more likely to be replaced by methods in B". We do not treat the change rule involving A and C as a wrong rule, because it is possible that methods in one class are replaced by methods in two or more classes in the new release. The feedback data just tells the classifier that the methods in one class are more likely to be replaced by the methods in another class in the new release and thus to promote change rules in which these methods are involved. When Halo computes the method definition text similarity $\sigma_D(x, t)$, it treats the two classes as the same class instead of computing the real text similarity between the names of the two classes. Therefore, the methods in the classes included in FD are promoted to the same level of the methods in the class of the target method in our multi-stage text similarity comparisons but final matching also depends on the other features used by the comparison algorithm.

Figure 3 shows the flowchart diagram of the two-pass analysis. The two passes have the same classifier. The differences between them are: (1) the first pass uses a threshold to keep the change rules with high LCS and derives the training data FD from them and (2) the second pass uses the high-precision change rules and the feedback data FD as other inputs besides the feature sets to improve the precision of the method similarity $\sigma(x, t)$ and then generate the final change rules incrementally. More specifically, the high-precision change rules will be a part of the final change rules and some of them can be added to the strict anchor set A_s if they meet the selection criteria. The feedback data are used while computing method definition text similarity $\sigma_D(x, t)$.

3.6 Implementation

We implement our approach as a command-line Java program. This program takes the source code of two releases of a framework as input and outputs a set of change rules in plain text and XML format (see Figure 4). The former is readable by developers while the latter is useful for automated post-processing.

The feature extractor is based on Eclipse JDT parser, which we use to extract the information related to the features used in our approach, *i.e.*, call-dependency relations, inheritance relations, method documentations, and method signatures.

Call-dependency relations include all the methods defined in the two releases of the framework to be analyzed and the methods called by them. As discussed in Section 3.2, to balance accuracy and performance, we do not extract the methods from third-party libraries.

Inheritance relations are the full inheritance trees of the classes of the methods in call-dependency relations. For frameworks written in Java, they include both `extends` trees and `implements` trees.

Method documentations extracted from the source code are linked the the methods containing them. In the current implementation, we only extract the textual part (excluding annotations) of Javadoc because they are connected with methods by the Java compiler. Extending to other formats of method documentations is future work.

Before computing the similarities, the method signatures, method documentations, class, and package names are tokenized in the way proposed by Lawrie *et al.* [25]. They are split at upper-case letters and other legal characters, except lower case letters and numbers.

For a target method t, Halo first computes $\sigma(x, t)$ for each method x in the newer release V_j . Then, Halo generates the replacement set $R(t)$ using the classification algorithm described in Section 3.4 to decide if a method x in V_j replaces a target method t in T t or not according to $\sigma(x, t)$.

In our implementation, we set the threshold $\gamma = 0.75$ to generate the feedback data in the first pass. It means that, in the feedback data, the target and replacement methods must have, at least, 75% tokens in common. We choose this value because M. Kim *et al.* [11] showed that a threshold between 0.65-0.70 gives good balance between precision and recall and the average precision is above 90%. We use a higher value (75%) to have more precise feedback data. Our implementation of Halo is available on-line[4].

4 Evaluation

We now evaluate and compare Halo with other previous approaches on several target systems.

4. http://www.ptidej.net/downloads/experiments/tse13a

Figure 3. Halo Two-pass Classifier

```
// Text format
void CH.ifa.draw.standard.StandardDrawingView.selectionChanged()
    Replacement
     --- void
CH.ifa.draw.standard.StandardDrawingView.fireSelectionChanged()

// XML format
        <targetMethod>
            <owner>
                <qualifingName>
                    <headToken>C</headToken>
                    <tailToken>H</tailToken>
                </qualifingName>
                <qualifingName>
                    <headToken>ifa</headToken>
                </qualifingName>
                <qualifingName>
                    <headToken>draw</headToken>
                </qualifingName>
                <qualifingName>
                    <headToken>standard</headToken>
                </qualifingName>
                <elementName>
                    <headToken>Standard</headToken>
                    <tailToken>Drawing</tailToken>
                    <tailToken>View</tailToken>
                </elementName>
                <isArray>false</isArray>
            </owner>
            <returnType>
                <elementName>
                    <headToken>void</headToken>
                </elementName>
                <isArray>false</isArray>
            </returnType>
            <methodName>
                <headToken>selection</headToken>
                <tailToken>Changed</tailToken>
            </methodName>
        </targetMethod>
        <replacementMethods>
            <verified>false</verified>
            <replacementMethod>
                <owner>
                    ...
                </owner>
                <returnType>
                    <elementName>
                        <headToken>void</headToken>
                    </elementName>
                    <isArray>false</isArray>
                </returnType>
                <methodName>
                    <headToken>fire</headToken>
                    <tailToken>Selection</tailToken>
                    <tailToken>Changed</tailToken>
                </methodName>
            </replacementMethod>
        </replacementMethods>
```

Figure 4. Example of Halo Outputs

4.1 Design

We compared the precision and recall values on Android SDK, JFreeChart, JHotDraw, JEdit, and Struts to compare Halo with AURA, the approaches of M. Kim *et al.* [11] and Schäfer *et al.* [12]. We use the large system (`org.eclipse.jdt.core` and `org.eclipse.jdt.ui`) to compare Halo with AURA and SemDiff [10]. Table 1 summarizes the six subject systems.

Android is written in several programming languages but its SDK is in Java. We analyzed Android SDK releases 2.1_r2.1p2 and 2.2.3_r2 with Halo. They are the latest versions of two releases of Android: Eclair (API Level 7) and Froyo (API Level 8). First, we downloaded all the source code of the two versions of Android from its software repository. Then, we compiled the two versions to obtain their complete source code, because some parts of Android SDK are written in AIDL (Android Interface Definition Language)[5], which is used to generate Java code.

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

To verify the change rules generated by Halo, we read the source code of the target systems and related documents to decide if the change rules are correct or not. For each target system, at least two of the authors inspected the output of Halo and discussed their assessment before agreeing on the evaluation of the results. A third author was called when the two authors could not agree.

Among the five subject systems that we manually analyzed, the results of three of them took more time to evaluate than the others. The first is Android SDK because it is the largest one. The other two are JEdit and JFreeChart. JEdit changed dramatically between releases 4.1 and 4.2, especially the implementations of some anchor methods. The changes lowered the precision of call-dependency analysis and also impeded the inspection. In JFreeChart, there are many methods with similar signatures. We spent a lot time to distinguish them to make sure that the change rules were correct or not.

4.2 Hypothesis and Performance Indicators

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

We can use precision but not recall [29], to directly compare the performance of Halo and that of the previous approaches because the set $relevant\ rules$ is *a prior* unknown in:

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

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

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

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

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

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 Halo on five medium-size systems, Android SDK, JFreeChart, JHotDraw, JEdit, and Struts by manual inspection. For the previous approaches, we use the data provided by the corresponding authors.

Besides $\Delta\mathbf{P}$ and $\Delta\mathbf{R}$, we also use Cliff's d [22] to show the effect sizes between the precision and recall of Halo and those of previous approaches. Cliff's d is a non-parametric effect size that does not require any pre-knowledge of the distribution of the data. The difference is trivial when $0.0 \leq |d| < 0.147$, small when $0.147 \leq |d| < 0.33$, moderate when $0.33 \leq |d| < 0.474$, and large when $0.474 \leq |d|$. $|d|$ is the absolute value of d. A positive (negative) d means an increase (decrease) of the precision or the number of correct rules between Halo and those of previous approaches. The Cliff's d values of precision (d_p) and of the approximate recall ($d_{r'}$) defined below are computed according to the following equations:

$$d = \frac{\#(x_i > y_j) - \#(x_i < y_j)}{n_x \times n_y}$$

where $\#(x_i > y_j)$ is the number of times that $x_i > y_j$ is true and n_x and n_y are the sizes of the two data sets respectively.

We compute the Cliff's d of an approximate recall $d_{r'}$ instead of the Cliff's d of the true recall because the

5. http://developer.android.com/guide/developing/tools/aidl.html

Table 1
Subject Systems

Subject Systems	Releases	# Methods	Analysed By
Android SDK	2.1_r2.1p2	20516	Kim et al. [11] AURA [13] Halo
	2.2.3_r2	21214	
JFreeChart	0.9.11	4,751	Kim et al. [11] AURA [13] Halo
	0.9.12	5,197	
JHotDraw	5.2	1,486	Kim et al. [11] Schäfer et al. [12] AURA [13] Halo
	5.3	2,265	
JEdit	4.1	2,773	Kim et al. [11] AURA [13] Halo
	4.2	3,547	
Struts	1.1	5,973	Schäfer et al. [12] AURA [13] Halo
	1.2.4	6,111	
`org.eclipse.jdt.core`	3.1	35,439	SemDiff [10] AURA [13] Halo
`org.eclipse.jdt.ui`	3.3	47,237	

unknown set $\{relevant\ rules\}$ prevents us to compute the latter, but we use the total number of methods that do not exist, called $M_{deleted}$, in the new release of a framework as $|\{relevant\ rules\}|$ to compute an approximation of the recall r'. We believe that using $M_{deleted}$ to represent the number of relevant rules is reasonable for two reasons: first, this number is actually greater or equal to the number of the change rules generated by previous approaches and Halo; second, the use of this number impacts the computation of the recall of all approaches equally: if it favors one, it favors all the others.

For the two Eclipse plug-ins, `org.eclipse.jdt.core` and `org.eclipse.jdt.ui`, from 3.1 to 3.3, Halo 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 [10]: we 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 methods not existing anymore in release 3.3; and, compute the precision of the change rules that cover these compile errors.

4.3 Comparison on the Medium-size Systems

In Table 2, we present the $\Delta\mathbf{P}$ and $\Delta\mathbf{R}$ in column 5 and 6 of each subject system between Halo and M. Kim *et al.*'s [11], Schäfer *et al.*'s [12] approaches, and AURA [13]. In the last three rows, we present the total average values of Halo compared to the three approaches: $\Delta\mathbf{R}$ is 38% with a precision of 92%, while $\Delta\mathbf{P}$ is 5%. In Table 7, we also report the complete

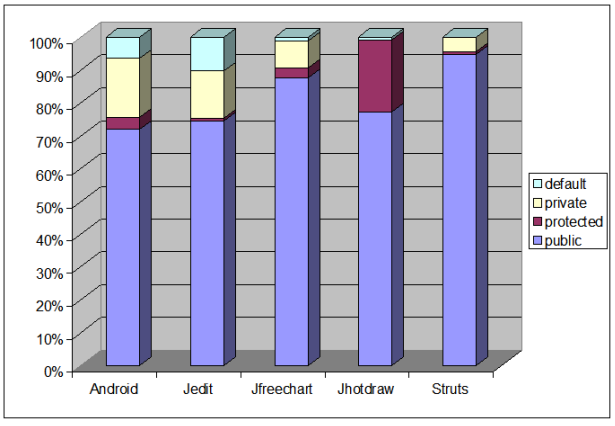

Figure 5. Visibility distribution of the results on medium-size systems with simply-deleted change rules

comparison data, including the average values of $\Delta\mathbf{P}$, $\Delta\mathbf{R}$, $d_{r'}$, and d_p for each approach in columns 7 to 10. Figure 5 shows that most of the change rules are on public methods.

Comparison with M. Kim et al.'s approach: M. Kim *et al.* [11] 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 Halo. Therefore, we use the number of matches from [11] to compare their results with ours.

On average, $\Delta\mathbf{P}$ is -0.3% while $\Delta\mathbf{R}$ is 80%. We gain in recall and also catch up in precision. The values of $d_{r'}$ and d_p also confirm that the improvement of the

Table 2
Summary of the results on medium-size systems with simply-deleted change rules

Systems	Indicators	Halo	M. Kim *et al.* [11]	ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	102	81	26%	-3%
	Precision	96%	99%		
JEdit 4.1-4.2	# Correct rule	386	217 [4]	78%	-9%
	Precision	85%	93%		
JFreeChart 0.9.11-0.9.12	# Correct rule	169	88	92%	12%
	Precision	88%	78%		
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	1739	772	125%	-2%
	Precision	91%	93%		

Systems	Indicators	Halo	Schäfer *et al.* [12]	ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	102	88	16%	9%
	Precision	96%	88%		
Struts 1.1-1.2.4	# Correct rule	133	66	102%	15%
	Precision	99%	86%		

Systems	Indicators	Halo	AURA [13]	ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	102	97	5%	4%
	Precision	96%	92%		
JEdit 4.1-4.2	# Correct rule	386	360	8%	6%
	Precision	85%	80%		
JFreeChart 0.9.11-0.9.12	# Correct rule	169	155	9%	8%
	Precision	88%	81%		
Struts 1.1-1.2.4	# Correct rule	133	129	4%	5%
	Precision	99%	96%		
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	1739	1608	8%	8%
	Precision	91%	85%		

Total Average		
	Precision of Halo	92%
	ΔR	43%
	ΔP	5%

number of correct rules is large and the decrease of precision is small.

On JHotDraw from 5.2 to 5.3, JFreeChart from 0.9.11 to 0.9.12, and Android SDK from 2.1_r2.1p2 to 2.2.3_r2, the Δ**R**s are 26%, 92% and 125% while the Δ**P**s are -3%, 12% and -2%, respectively. A slight decrease of precision (-3% and -2%) is acceptable because the recall increases satisfactorily (26% and 125%).

On JEdit from 4.1 to 4.2, the Δ**R** is 78% while Δ**P** is -9%. Two aspects 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. The feedback data set is not effective when call-dependency analysis gives the wrong methods to choose.

Comparison with Schäfer et al.'s approach: On average, Δ**P** is 12% while Δ**R** is 59%. Both $d_{r'}$ and d_p indicate large improvement in precision and recall. Halo has positive Δ**R** and Δ**P** both on JHotDraw from 5.2 to 5.3 and Struts from 1.1 to 1.2.4 in comparison to Schäfer *et al.*'s [12]. On JHotDraw from 5.2 to 5.3, the Δ**R** and Δ**P** are 16% and 9%, while they are 102% and 15% on Struts from 1.1 to 1.2.4.

Comparison with AURA: As an extension of AURA, the results of Halo are also better than those of AURA in our experiments, especially in precision. On average, Δ**P** is 6% while Δ**R** is 7%. The values of $d_{r'}$ and d_p showed large improvement in recall and moderate increase in precision.

Halo has positive Δ**R** and Δ**P** on all the four systems. These results show that considering more features and two-pass analysis improve both recall and precision. We also analyze the contributions of each features to the precision of the change rules generated by Halo. More detailed discussions are in Section 5.

4.4 Comparison on a Large-size System

In Table 3, we present the results of Halo, AURA and SemDiff [10] to solve the compile errors of three Eclipse 3.1 plug-ins when compiling them against Eclipse 3.3.

The precision of Halo is 100% and that of AURA is 92.86% only with one wrong change rules in Mylyn from V0.5 to V2.0.

In SemDiff [10], 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 Halo with those of SemDiff, we

3. Halo only analyzed the packages org.gjt.sp.* and compared its results with those of M. Kim *et al.* [11]. These packages contain the code for JEdit main functions and are large enough for manual analysis (456 target methods).

4. Confirmed by Dagenais, it is 1.5-2.0

Table 3
Evaluation of a sample of change rules on the large system

Systems		Halo	AURA	SemDiff [10]
org.eclipse.jdt.debug.ui 3.1 - 3.3	# Errors in Scope	4		
	# Found Rules	4	4	4
	# Correct Rules	4	4	4
Mylyn 0.5-2.0	# Errors in Scope	2		
	# Found Rules	2	2	2
	# Correct Rules	2	1	2
JBossIDE 1.5-2.0 [5]	# Errors in Scope	8		
	# Found Rules	8	8	8
	# Correct Rules	8	8	8
Precision		100%	92.86%	$\leq$ 100.00%

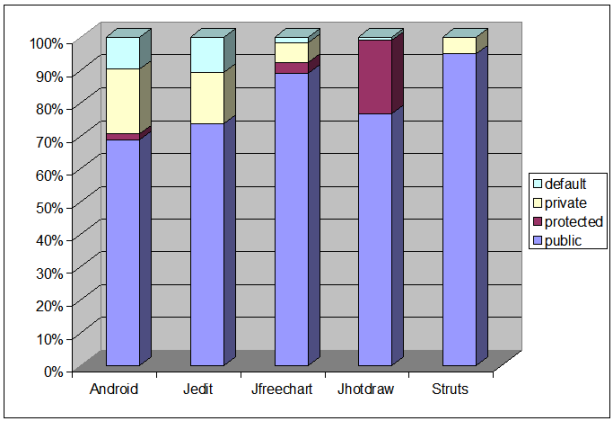

Figure 6. Visibility distribution of the results on medium-size systems without simply-deleted change rules

must account for this discrepancy in the way correct rules are counted.

If every correct rule was the first recommendation of the top three rules reported by SemDiff, then SemDiff would have a precision of 100.00%. However, it is also possible that the correct rule was the second or third recommendation, SemDiff would have less than 100% precision, thus Halo is competitive with SemDiff.

4.5 Comparison without Simply-deleted Methods

Some previous approaches, such as [11], [12], do not explicitly report simply-deleted change rules in their results while AURA and Halo report them. We remove the simply-deleted change rules from the results of Halo on the four medium-size systems and compare them again with the results of the previous approaches to assess their influence on precision and recall.

As shown in Table 4, $\Delta\mathbf{P}$ and $\Delta\mathbf{R}$ are equal or lower than those with simply-deleted method rules (4% vs. 5% and 14% vs. 43%, respectively). So, the results of Halo with and without simply-deleted rules have similar precision and the former has lower recall because many target methods are actually simply deleted. Table 8, reports the complete comparison data, including the average values of $\Delta\mathbf{P}$, $\Delta\mathbf{R}$, $d_{r'}$, and d_p, for each approach without simply-deleted methods in columns 7 to 10. Most of the non-simply-deleted change rules are also on public methods as showed in Figure 6.

The results shows that Halo also performs better than previous approaches on no-simply-deleted rules.

4.6 Performance

The analyses of Halo 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 with Halo takes less than ten minutes on Windows XP SP3 with Intel Core Duo 1.5GHz and 4GB RAM. Analyzing Android SDK 2.1 and 2.2, Eclipse JDT core and UI 3.1–3.3 with Halo takes two and five hours respectively on CentOS 5.5 with Intel Xeon 16-Core 2.4GHz and 48GB RAM.

4.7 Threats to Validity

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

Construct validity threats concern the relation between theory and observation. First, we made sure that we rigorously implement the approach described in previous sections, through careful peer-reviews of the code and the results. In our context, we want to see if Halo generate more accurate change rules than previous approaches. We use different approaches as treatments and we observe the precision and recall of their outputs. In Halo, we assume that all replacement methods are taken from all the methods existing in the new release of the framework to be analyzed or belonging to other frameworks provided by the same vendor. However, it is possible that developers replace target methods with methods from the frameworks of other vendors. After analyzing the results of the subject systems, we found that such replacement occurs in less than 1% of all replacement methods. Thus, we believe that this threat does not compromise the construct validity.

Table 4
Summary of the results on medium-size systems without simply-deleted change rules

Systems	Indicators	Halo	M. Kim *et al.* [11]	ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	99	81	22%	-3%
	Precision	96%	99%		
JEdit 4.1-4.2	# Correct rule	289	217 [4]	33%	-13%
	Precision	81%	93%		
JFreeChart 0.9.11-0.9.12	# Correct rule	113	88	28%	15%
	Precision	90%	78%		
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	914	772	18%	-6%
	Precision	87%	93%		

Systems	Indicators	Halo	Schäfer *et al.* [12]	ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	99	88	13%	9%
	Precision	96%	88%		
Struts 1.1-1.2.4	# Correct rule	58	66	-12%	13%
	Precision	97%	86%		

Systems	Indicators	Halo	AURA [13]	ΔR	ΔP
JHotDraw 5.2-5.3	# Correct rule	99	96	3%	0%
	Precision	96%	96%		
JEdit 4.1-4.2	# Correct rule	289	247	17%	5%
	Precision	81%	77%		
JFreeChart 0.9.11-0.9.12	# Correct rule	113	95	19%	10%
	Precision	90%	82%		
Struts 1.1-1.2.4	# Correct rule	58	56	4%	5%
	Precision	97%	92%		
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	914	826	11%	8%
	Precision	87%	81%		

Total Average	Precision of Halo	90%
	ΔR	14%
	ΔP	4%

Internal validity verifies if the outcome is really caused by the treatment. In the multi-stage text similarity comparison algorithm of Halo, there is an arbitrary selection of candidates in the last step that could introduce randomness in the results. However, only 0.71% of the comparisons between the candidates in our experiments are decided by this arbitrary selection. Thus, the influence of this arbitrary selection is negligible. In our implementation, we only extract Javadoc, which do not cover all the method documentations. In the subject systems that we analyzed, more than 90% of method documentations are in Javadoc. We believe that the influence of Javadoc is not significant. We inspected the change rules generated by Halo manually. We cannot rule out human error in validating results. Because we read the source code of the subject systems carefully and at least two of the authors evaluated and agreed on the results of each system, we believe that this threat does not compromise the internal validity.

Conclusion validity threats concern the relation between the treatment and the outcome. We used unbiased systematic measures and the data provided by the authors of previous approaches without any changes. We did provide all data on-line for further independent validation[2].

Reliability validity threats concern the possibility of replicating this study. We attempted here to provide all the necessary details to re-implement Halo 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-line[2].

Threats to *external validity* concern the possibility to generalize our findings. We studied five systems of different sizes, belonging to different domains, and evaluated by the previous approaches. However, we only analyzed Java code; therefore it is possible that Halo 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. Future work also includes an empirical study to investigate the usability of change rules generated by the tools on framework API evolution.

5 Discussion

This section discusses the scope of API changes and the differences between Halo and previous approaches.

5.1 Android API Difference Reports

The meta-data of the Web pages of Android API difference reports show that they are generated by JDiff[6]. JDiff reports packages, classes, methods, and

6. http://www.jdiff.org

fields that are removed, added, or changed (including changes to their documentation), at configurable API level. JDiff does not report API change rules. For added and removed methods, JDiff does not report if an added method replaces a method in the old release or if a removed method is replaced by a method in the new release. The changes of methods reported by JDiff include deprecation and changes in thrown exceptions but not on method signatures.

Therefore, the Android API difference reports really describe API differences, not the relations among the changed APIs, such as the mapping between removed methods and their replacements. The usefulness of such reports is limited to developers because a list of new added methods without additional information does not help them to know how to adapt their client code. A list of removed methods cannot help developers find their replacements. Developers can still use methods annotated as @deprecated without any error. Changes in thrown exceptions can be fixed in Eclipse by one mouse click without consulting the reports.

Contrary to JDiff, Halo, and other similar approaches, reports change rules consisting of the replacement methods of target methods that existed in the old release of a framework, e.g., Android SDK, but not in its new release (in terms of their signatures). The missing methods in the new release would cause compilation errors. Halo shows developers what are the replacements of the target methods and saves them the time to search in the new release to find their replacements. Such change rules are not in Android API difference reports currently.

For example, the Android API difference report between v2.1 and v2.2 includes 246 methods: 206 methods are marked as added, three methods are marked as removed, and 37 methods are marked as changed. The 206 added methods are listed in the report without any further information, neither if they replace some old methods nor if they provide new features. Among the 40 removed and changed methods, two are JDK methods (`java.net.DatagramSocketImpl.getOption(int)` and `setOption(int, Object)`), which are not defined in the Android SDK source code. JDiff reports another 20 methods because their documentation changed by adding the @deprecated annotation. The remaining 18 are changed in the thrown exceptions.

None of these 40 methods reported in the Android API difference report between v2.1 and v2.2 are in the scope of Halo, because either they are not a part of Android SDK v2.2 source code or their definitions still exist in v2.2. Many changes, like class `android.pim.vcard.ContactStruct` in Android API v2.1 that was replaced by class `android.pim.vcard.VCardEntry` in v2.2, are absent from the Android API difference report, but are detected by Halo. In total, Halo detected 1,739 correct change rules between Android SDK v2.1 and V2.2, of which 914 are not simply-deleted rules.

5.2 Multi-Features

In AURA [13], we use call-dependency relations and text-similarities as features, which boost recall with similar precision compared with previous approaches that only use one of the two features. According to Duda *et al.* [17], considering multi-feature is a way to improve pattern classification. Therefore, we added another two features: inheritance relations and method documentations. Our experimental results (discussed in Section 5.5) show that they contribute to the improvement of precision. Here are two examples to show the contributions of the two features:

Method Documentation: The replacement method of `boolean org.eclipse.jdt.internal.ui.text.correction.ConvertIterableLoopProposal.isApplicable()` from Eclipse JDT UI 3.1 is `org.eclipse.core.runtime.IStatus org.eclipse.jdt.internal.corext.fix.ConvertIterableLoopOperation.satisfiesPreconditions()` in its 3.3 release. Because of the dramatic changes in their return values, package names, class names, and method names, this change rule cannot be detected by text similarity analysis alone. These two methods are not called by any anchor, so call dependency analysis does not apply either.

Using method documentations, Halo detects the correct change rule because the target and the replacement methods have an identical method documentation (shown in Figure 7). The benefit of considering method documentations is not only to detect change rules, but also to improve the computation time. Halo indexes all methods by their method documentations. Thus, it can detect change rules using method documentation analysis much more efficiently than those using text-similarity analysis.

Inheritance: To find the replacement method of `org.jfree.data.DefaultBoxAndWhiskerDataset.getMedianValue(int, int)` from JFreeChart 0.9.11 in its 0.9.12 release, the methods `DefaultBoxAndWhiskerCategoryDataset.getMedianValue(int, int)` and `DefaultBoxAndWhiskerXYDataset.getMedianValue(int,int)` can be identified using text-similarity as two candidates for the replacement method. It is difficult to decide which one is correct because of the high similarity between them. Halo uses inheritance relations to clearly distinguish `DefaultBoxAndWhiskerXYDataset.getMedianValue(int, int)` from the other because the class of the former implements the same interface `XYDataset` as `DefaultBoxAndWhiskerDataset`, while the class of the latter implements another interface `CategoryDataset` (shown in Figure 8).

We could use other measurements of inheritance tree similarity, such as [31], [32]. Research on choosing

```
// Version 3.1
/**
 * Is this proposal applicable
 */
boolean org.eclipse.jdt.internal.ui.text.correction.ConvertIterableLoopProposal.isApplicable()

// Version 3.3
/**
 * Is this proposal applicable
 */
org.eclipse.core.runtime.IStatus
    org.eclipse.jdt.internal.corext.fix.ConvertIterableLoopOperation.satisfiesPreconditions()
```

Figure 7. Change rule detected by method documentations

```
// Version 0.9.11
class DefaultBoxAndWhiskerDataset implements BoxAndWhiskerDataset {...}
interface BoxAndWhiskerDataset extends XYDataset {...}

// Version 0.9.12
class DefaultBoxAndWhiskerXYDataset implements BoxAndWhiskerXYDataset {...}
interface BoxAndWhiskerXYDataset extends XYDataset {...}

class DefaultBoxAndWhiskerCategoryDataset implements BoxAndWhiskerCategoryDataset, RangeInfo {...}
interface BoxAndWhiskerCategoryDataset extends CategoryDataset {...}
```

Figure 8. Change rule detected by inheritance relations

different features or different measurements of the same features to develop better approaches is one of our future approaches.

5.3 Multi-pass Analysis and Feedback Data

Halo is a two-pass analysis approach. The first pass generates feedback data and derives the high-level change rules, such as "methods in class A will be replaced by methods in class B". The second pass promotes the rankings of the methods covered by the high-level change rules. Using feedback data, we can give priority to certain methods without ruling out the possibility that the target methods can also be replaced by other methods.

The replacement method of `void android.pim.vcard.ContactStruct.setPosition(java.lang.String)` from Android SDK 2.1 is `void android.pim.vcard.VCardEntry.handleTitleValue(java.lang.String)` in its 2.2 release. In v2.1, method `ContactStruct.setPosition` handles the hierarchical position of an individual in an organization. In v2.2, this function is moved to `VCardEntry.handleTitleValue`. Because both their class names and method names are completely changed, this change rule cannot be detected by AURA (shown in Figure 9).

With Halo, we find that 19 other methods in `android.pim.vcard.ContactStruct` are replaced by the methods in `android.pim.vcard.VCardEntry` in the feedback data generated by the first pass. Thus, we promote `android.pim.vcard.VCardEntry` as being the same class as `android.pim.vcard.ContactStruct` when we compute the text similarity between the signatures of the two methods and correctly discover this change rule.

5.4 Strict Anchor

If the signatures of two methods in an anchor are the same but their implementations are dramatically changed, the call-dependency analyses used by previous approaches will most likely generate wrong change rules because the call relations are churned in the new release of the framework. If num_o and num_n are the numbers of methods called by the old and the new releases in a method pair respectively, when $|num_o - num_n| < min(num_o, num_n)$, Halo accepts the method pair as an anchor; otherwise, it does not. This criteria for anchors excludes method pairs with too different implementations. The intuition in using strict anchor is to reduce the number of wrong change rules generated by the call dependency analysis and to give our approach a chance to detect the change rules using other features correctly.

For example, the method `org.eclipse.jdt.ui.JavaElementSorter.getClassPathIndex(...)` is called by the method `org.eclipse.jdt.ui.JavaElementSorter.compare(...)` in Eclipse JDT UI 3.1 as well as by 12 other methods. In Eclipse JDT UI 3.3, the method `org.eclipse.jdt.ui.JavaElementSorter.compare(...)` only calls one method `org.eclipse.jdt.ui.JavaElementComparator.compare(...)`. Without strict anchor, our previous approach would treat `org.eclipse.jdt.ui.JavaElementComparator.compare(...)` as the replacement of `org.eclipse.jdt.ui.JavaElementSorter.getClassPathIndex(...)` because the former has 100% confidence value to the latter.

Because of the different numbers of methods called by the two versions of `org.eclipse.jdt.ui.JavaElementSorter.compare(...)` (13:1), it is not a strict anchor. So, the correct replacement method `org.eclipse.jdt.ui.JavaElementComparator.`

```
// Version 2.1
void android.pim.vcard.ContactStruct.setPosition(java.lang.String)

// Version 2.2
void android.pim.vcard.VCardEntry.handleTitleValue(java.lang.String)
```

Figure 9. Change rule detected with the help of feedback data

`getClassPathIndex(...)` can be detected later by text similarity analysis in Halo.

Using other techniques, such as clone detection techniques (summarized in [33]) to discover accurate anchors, is an interesting topic for future work.

5.5 Contributions of the New Components of Halo

Halo uses four new components to improve its results with respect to previous approaches: a feedback data set, method documentations, inheritance relations, and stricter anchors. We conduct a series of experiments to analyze the influences of each component on the overall results. In each experiment, we disable one component in Halo and called it Halo$^-$. Then, we compare the precision of Halo$^-$ with that of Halo and report the differences. Because Halo$^-$ and Halo generate the same number of change rules, the trend between their differences in recall will be the same as that in precision. So, we only report the differences in precision. Table 5 and Table 6 show the results of these experiments. The data in the former table are the results on all change rules; the latter only for no-simply deleted change rules.

First, we see that the influences of the components are different from system to system. For JHotDraw, there is not much differences on precision, if we remove one component from Halo. jEdit is the contrary: it is sensitive to all the components. The other systems are between them. We explain these observations as follows: JHotDraw is a system developed by Gamma *et al.* to demonstrate the application of design patterns [34]. It is elegantly designed and consistently coded. When removing one component, the other components can provide enough information to allow Halo to correctly detect the relevant change rules. The changes between jEdit 4.1 an 4.2 are dramatic. Especially, there are many methods in the two releases with the same signatures but different implementations. It confuses call-dependency analysis and compromises its precision. Thus, Halo needs all the components to work together to have accurate results for jEdit.

Second, the use of feedback data has more influence on the systems with lower precision while using AURA. The lower precision on these systems is mainly caused by the dramatic changes between the two releases. High-quality feedback data helps to improve the precision. If the precision is already relatively high, the improvement is less obvious.

5.6 Thresholds

Using thresholds is a common tools in software engineering to trade recall for precision [35], [11], [12]. Our previous work, AURA [13], completely eliminates thresholds to boost recall with precision similar to that previous approaches. In Halo, we use a threshold $\gamma = 75\%$ to automatically select feedback data to increase precision without compromising recall. Future work includes using different values of this threshold.

5.7 Limitations

In Halo, our classifier prioritizes different features: call dependency similarity is the highest and method definition text similarity is the lowest. The irrelevant change rules generated by high-priority features in the early steps can be propagated and amplified in later steps. It is difficult to correct them even if low-priority features tell something different, because we do not know which feature we should trust more. As future work, we will investigate a new methodology to overcome this limitation.

Halo only generates change rules for methods. During the evaluation of Halo, we found that some getters are replaced by direct field accesses. Future work includes modifying the definition of change rules to take fields into account.

6 Related Work

We now briefly introduce previous approaches solving on framework API evolution identification and other relevant work that shares matching techniques with them.

6.1 Approaches Solving the Framework API Evolution Identification

Several approaches help developers evolve their programs when the frameworks that they use change. We now discuss the differences between these approaches and ours according to the common aspects in the formulation presented in this paper. We do not discuss other aspects that do not apply to previous approaches, such as feedback loop.

Inputs: Existing approaches of capturing API-level changes require the framework developers to manually enter the change rules or to use a particular IDE to automatically record the changes. Chow and Notkin [6] presented a method that requires the framework developers to provide change rules with the new releases. CatchUP! [8] and JBuilder [9] record the

Table 5
The Influences of the components on Halo precision

Difference To Original Halo	Android	JEdit	JFreeChart	JHotDraw	Struts	Average
No Feedback Data	-1.8%	-4.8%	-1.6%	0.0%	-1.5%	-1.9%
No Method Documentation	-1.0%	-5.3%	-1.6%	-0.9%	-2.2%	-2.2%
No Inheritance	-1.0%	-5.5%	-3.1%	-1.9%	-2.2%	-2.7%
No Strict Anchor	-0.5%	-7.7%	-2.1%	-1.9%	-4.4%	-3.3%

Table 6
The Influences of the components on Halo precision of no-simply deleted change rules

Difference To Original Halo	Android	JEdit	JFreeChart	JHotDraw	Struts	Average
No Feedback Data	-1.2%	-8.9%	-8.5%	0.0%	-3.1%	-4.4%
No Method Documentation	0.0%	-1.8%	-1.5%	-0.0%	-0.0%	-0.1%
No Inheritance	-1.5%	-1.6%	-5.9%	-1.0%	-3.1%	-2.7%
No Strict Anchor	-0.8%	-3.0%	-0.7%	-1.9%	-4.6%	-1.9%

refactoring operations in one release and replay them in another. MolhadoRef [7] also employs a record-and-replay technique for handling API-level changes in merging program releases. These approaches can provide accurate change rules because of the framework developers' involvement, which might not always be available.

Dagenais and Robillard's SemDiff [10] and HiMa of Meng *et al.* [36] use software repository commits. Diff-CatchUp, developed by Xing and Stroulia [28], uses the models of logical design of two releases of a system as input. Others approaches [37], [35], [11], [12], [13] and Halo take the source code of evolved frameworks as input. Schäfer *et al.* [12]'s approach also uses the client code as a part of its input.

Features: The features used by the approaches of capturing API-level changes [6], [8], [9], [7] are different presentations of the change rules manually added or automatically captured. These approaches have a specific model for each target method and its replacement.

Godfrey and Zou's [37] and S. Kim *et al.*'s [35] approaches use text similarity, software metrics, and call dependency relations to describe target methods and their replacements. Xing and Stroulia [28] extract the differences between two releases of the logical design models using lexical and structural similarity, including text similarity, inheritance relations, usage dependencies, and association relations. M. Kim *et al.*'s [11] compute LCS of the target methods and its replacement to measure the difference between them. SemDiff [10], Schäfer *et al.* [12], and AURA [13] use call-dependency relations measured by confidence value and various presentations of text similarity. HiMa [36] uses call-dependency relations and natural-language-analysis-filtered method documentations of consecutive commits. Halo considers call-dependency relations, text similarity, method documentations and inheritance relations as its features.

Classifiers: The approaches of capturing API-level changes only need simple classifiers to match the change rules that they have to the target methods, but the developers' involvement that they require might not be available.

The classifiers of Godfrey and Zou's [37] and S. Kim *et al.*'s [35] approaches are based on origin analysis techniques. The former is semi-automatic while the latter is automatic. Diff-CatchUp [28] defines three sets of heuristics for class, method, and fields, respectively to order the possible possible replacement methods. SemDiff [10] and Schäfer *et al.* [12] first use confidence values to preselect the possible replacement methods, then use text similarity to rank them. The difference between them is that the former focuses on how frameworks adapt to their own changes while the latter discovers the usage changes from client code. M. Kim *et al.*'s [11] classifier leverages systematic renaming patterns using text similarity to match old APIs to new APIs. AURA [13] uses an ad hoc algorithm, similar to a binary classifier. HiMa's [36] classifier generates initial change rules using processed revision method comments, then expands and refines them using call-dependency analysis. Halo implements a two-pass analysis classifier. First, it generates feedback data automatically, then feeds them to the second pass to improve precision.

6.2 Other Relevant Work

Kim and Notkin surveyed program element matching techniques for multi-version program analyses [38]. They grouped matching techniques into eight categories: entity name matching; string matching; syntax-tree matching; control-flow graph matching; program-dependence graph matching; binary-code matching; clone detection; and origin analysis. These techniques can be applied to solve the framework API evolution identification and to many other tasks in software engineering as follows:

Framework Evolution between Different Frameworks: Nita and Notkin used Twinning [39] to adapt different Java frameworks with similar functionalities. Twinning can describe two kinds of mapping between APIs A and B that perform resembling tasks: (1) directly mapping A to B or vice versa, and (2)

generating an abstract encapsulation C of A and B, then user can use C uniformly by Twinning. Zhong *et al.* presented MAN [40] to map APIs between Java and C#. They used a graph-based technique to analyze the API usages of some programs in Java and C# and derived the mapping between the APIs of the two languages.

API Analysis: Exemplar developed by Grechanik *et al.* [41] analyzed API calls to improve the precision of searching programs with similar functions over the Internet. Kawrykow and Robillard's approach [42] detects the reimplementation of APIs of existing libraries in client programs. McMillan *et al.* created a code-search system, Portfolio [43], to search and visualize relevant functions and their usages from an internal database. Nguyen *et al.* [44] presented LibSync to help developers learn complex API usage change patterns from the clients that have been already upgraded to new releases of frameworks. Cossette and Walker [45] manually analysed the public API incompatibilities of several versions of Struts, Log4j, and jDom. They classified public API incompatibilities into 14 categories and discovered that the current framework API evolution approaches only cover 20% of these cases because many API changes, such as replacing methods by configuration files or third-party libraries, are not in the scope of current framework API evolution approaches. Their work pointed directions of future framework API evolution research.

Software Evolution Comprehension: Fluri *et al.* presented Change Distilling [46], a tree differencing algorithm to rebuild change road-maps between two releases of some program elements. LSdiff developed by Kim and Notkin [47] summarizes the changes in method signatures, method bodies and fields between two releases of frameworks into systematic structural differences and presents anomalies in them. Ref-Finder of Kim *et al.* [48] automatically detects the 63 types of refactoring classified by Fowler et *et al.* [49] using template logic queries. Kpodjedo *et al.* [50] developed an approach that uses Error Tolerant Graph Matching algorithm to match evolving program elements and identify their stable parts, *i.e.*, the classes with stable relations (association, inheritance, and aggregation) across releases of an object-oriented program.

7 Conclusions

We presented Halo, a two-pass analysis approach considering call-dependency, text-similarity, method documentations, and inheritance relations to provide developers with change rules when adapting their systems from one release of a framework to the next.

The main contributions of Halo are:

1) Bring pattern classification vocabulary/techniques to framework API evolution.
2) Show that consideration of multiple features yields higher precision and recall.
3) Analyze the impact of different sub-components of Halo on its overall precision.

The results of the evaluation of Halo on five medium-size systems and its comparison to previous approaches showed that (1) the two-pass analysis inspired by binary pattern classification and (2) the use of call-dependency, text-similarity, method documentation, and inheritance-relation similarities do improve recall by 43% and precision by 5% on average. We also applied Halo on Eclipse and compared its results with those of SemDiff [10] and we showed that the approximated precision of both Halo and SemDiff is 100%. However, SemDiff needs users' inputs and is not fully automated.

In future work, we plan to extend our approach in several directions: investigate a new methodology to resolve the conflicting results between different features; analyze target systems in other programming languages than Java; consider other types of method documentations besides Javadoc; add heuristics that generate change rules for types and fields; consider other name splitting approaches; present Halo results in first-order relational logic rules, as introduced by M. Kim *et al.* [11]; applying more sophisticated pattern classification techniques.

Acknowledgment

We thank Barthélémy Dagenais and Martin P. Robillard for providing advice and their data and conducting analysis with the latest version of their approach. We are also grateful to Thorsten Schäfer for his experimental results. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

References

[1] 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 188–198.

[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*. New York, NY, USA: ACM, 2005, pp. 265–279.

[3] D. Dig and R. Johnson, "How do apis evolve - a story of refactoring: Research articles," *J. Softw. Maint. Evol.*, vol. 18, no. 2, pp. 83–107, 2006.

[4] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt, "Reuse contracts: managing the evolution of reusable assets," *SIGPLAN Not.*, vol. 31, no. 10, pp. 268–285, 1996.

[5] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, Jun. 1992.

[6] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *: Proceedings of the 1996 International Conference on Software Maintenance*, ser. ICSM 1996. Washington, DC, USA: IEEE Computer Society, 1996, p. 359.

[7] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 427–436.

[8] 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*. New York, NY, USA: ACM, 2005, pp. 274–283.

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

[10] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, pp. 19:1–19:35, Sep. 2011.

[11] 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*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 333–343.

[12] 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*. New York, NY, USA: ACM, May 2008, pp. 471–480.

[13] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 325–334. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806848

[14] J. Businge, A. Serebrenik, and M. van den Brand, "Eclipse api usage: the good and the bad," in *SQM*, 2012, pp. 54–62.

[15] ——, "Analyzing the eclipse api usage: Putting the developer in the loop," in *CSMR*, 2013, pp. 37–46.

[16] M. P. Robillard and R. DeLine, "A field study of api learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[17] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification 2nd Edition*. Wiley-Interscience, 2004.

[18] T. Joachims, "A probabilistic analysis of the rocchio algorithm with tfidf for text categorization," in *Proceedings of the Fourteenth International Conference on Machine Learning*, ser. ICML '97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, pp. 143–151.

[19] A. Maiga, N. Ali, N. Bhattacharya, A. Saban, Y.-G. Guéhéneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *WCRE '12: Proceedings of the 19th Working Conference on Reverse Engineering*, 2012.

[20] J. H. Hayes, A. Dekhtyar, S. K. Sundaram, and S. Howard, "Helping analysts trace requirements: An objective look," in *Proceedings of the Requirements Engineering Conference, 12th IEEE International*, ser. RE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 249–259.

[21] Y. Shin and J. Cleland-Huang, "A comparative evaluation of two user feedback techniques for requirements trace retrieval," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: ACM, 2012, pp. 1069–1074.

[22] R. Grissom and J. Kim, *Effect sizes for research: a broad practical approach*. Lawrence Erlbaum Associates, 2005.

[23] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools 2nd Edition*. Addison Wesley, 2007.

[24] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1993, pp. 207–216.

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

[26] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet Physics Doklady*, vol. 10, p. 707, 1966.

[27] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[28] Z. Xing and E. Stroulia, "API-evolution support with diff-CatchUp," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 818 – 836, December 2007.

[29] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, 1988.

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

[31] H. Bunke, "On a relation between graph edit distance and maximum common subgraph," *Pattern Recogn. Lett.*, vol. 18, no. 9, pp. 689–694, Aug. 1997.

[32] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998.

[33] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, Sep. 2007.

[34] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[35] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 143–152.

[36] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *Proceedings of 34th International Conference on Software Engineering*, ser. ICSE 2012, 2012, pp. 353–363.

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

[38] M. Kim and D. Notkin, "Program element matching for multi-version program analyses," in *Proceedings of the 2006 international workshop on Mining software repositories*, ser. MSR '06. New York, NY, USA: ACM, 2006, pp. 58–64.

[39] M. Nita and D. Notkin, "Using twinning to adapt programs to alternative apis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 205–214.

[40] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining api mapping for language migration," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 195–204.

[41] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 475–484.

[42] D. Kawrykow and M. P. Robillard, "Improving api usage through automatic detection of redundant code," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 111–122.

[43] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceeding of the 33rd international conference on Software engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 111–120.

[44] 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 *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 302–321.

[45] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 55:1–55:11.

[46] B. Fluri, M. Wuersch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change

extraction," *IEEE Trans. Softw. Eng.*, vol. 33, pp. 725–743, November 2007.
[47] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 309–319.
[48] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-finder: a refactoring reconstruction tool based on logic query templates," in *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 371–372. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882353
[49] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[50] S. Kpodjedo, F. Ricca, P. Galinier, Y.-G. Guéhéneuc, and G. Antoniol, "Design evolution metrics for defect prediction in object oriented systems," *Empirical Software Engineering*, vol. 16, no. 1, pp. 141–175, February 2011.

APPENDIX: COMPLETE COMPARISON DATA

Table 7
Comparison of the results on medium-size systems with simply-deleted change rules

Systems	Indicators	Halo	M. Kim *et al.* [11]	ΔR	ΔP	Averages ΔR	Averages ΔP	Averages $d_{r'}$	Averages d_p
JHotDraw 5.2-5.3	# Correct rule	102	81	26%	-3%	80%	-0.3%	Large 1.00	Small -0.25
	Precision	96%	99%						
JEdit 4.1-4.2	# Correct rule	386	217 [4]	78%	-9%				
	Precision	85%	93%						
JFreeChart 0.9.11-0.9.12	# Correct rule	169	88	92%	12%				
	Precision	88%	78%						
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	1739	772	125%	-2%				
	Precision	91%	93%						

Systems	Indicators	Halo	Schäfer *et al.* [12]	ΔR	ΔP	Averages ΔR	Averages ΔP	Averages $d_{r'}$	Averages d_p
JHotDraw 5.2-5.3	# Correct rule	102	88	16%	9%	59%	12%	Large 1.00	Large 1.00
	Precision	96%	88%						
Struts 1.1-1.2.4	# Correct rule	133	66	102%	15%				
	Precision	99%	86%						

Systems	Indicators	Halo	AURA [13]	ΔR	ΔP	Averages ΔR	Averages ΔP	Averages $d_{r'}$	Averages d_p
JHotDraw 5.2-5.3	# Correct rule	102	97	5%	4%	7%	6%	Large 0.48	Large 0.48
	Precision	96%	92%						
JEdit 4.1-4.2	# Correct rule	386	360	8%	6%				
	Precision	85%	80%						
JFreeChart 0.9.11-0.9.12	# Correct rule	169	155	9%	8%				
	Precision	88%	81%						
Struts 1.1-1.2.4	# Correct rule	133	129	4%	5%				
	Precision	99%	96%						
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	1739	1608	8%	8%				
	Precision	91%	85%						

Total Average		
	Precision of Halo	92%
	ΔR	43%
	ΔP	5%

Table 8
Comparison of the results on medium-size systems without simply-deleted change rules

Systems	Indicators	Halo	M. Kim *et al.* [11]	ΔR	ΔP	Averages			
						ΔR	ΔP	$d_{r'}$	d_p
JHotDraw 5.2-5.3	# Correct rule	99	81	22%	-3%	25%	-1.7%	Large 0.75	Small -0.25
	Precision	96%	99%						
JEdit 4.1-4.2	# Correct rule	289	217 [4]	33%	-13%				
	Precision	81%	93%						
JFreeChart 0.9.11-0.9.12	# Correct rule	113	88	28%	15%				
	Precision	90%	78%						
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	914	772	18%	-6%				
	Precision	87%	93%						

Systems	Indicators	Halo	Schäfer *et al.* [12]	ΔR	ΔP	Averages			
						ΔR	ΔP	$d_{r'}$	d_p
JHotDraw 5.2-5.3	# Correct rule	99	88	13%	9%	0.2%	11%	None 0.00	Large 1.0
	Precision	96%	88%						
Struts 1.1-1.2.4	# Correct rule	58	66	-12%	13%				
	Precision	97%	86%						

Systems	Indicators	Halo	AURA [13]	ΔR	ΔP	Averages			
						ΔR	ΔP	$d_{r'}$	d_p
JHotDraw 5.2-5.3	# Correct rule	99	96	3%	0%	7%	6%	Moderate 0.36	Moderate 0.36
	Precision	96%	96%						
JEdit 4.1-4.2	# Correct rule	289	247	17%	5%				
	Precision	81%	77%						
JFreeChart 0.9.11-0.9.12	# Correct rule	113	95	19%	10%				
	Precision	90%	82%						
Struts 1.1-1.2.4	# Correct rule	58	56	4%	5%				
	Precision	97%	92%						
Android SDK 2.1_r2.1p2 - 2.2.3_r2	# Correct rule	914	826	11%	8%				
	Precision	87%	81%						

Total Average	Precision of Halo	90%
	ΔR	14%
	ΔP	4%

L'École Polytechnique se spécialise dans la formation d'ingénieurs et la recherche en ingénierie depuis 1873

École Polytechnique de Montréal

École affiliée à l'Université de Montréal

Campus de l'Université de Montréal
C.P. 6079, succ. Centre-ville
Montréal (Québec)
Canada H3C 3A7

www.polymtl.ca

ÉCOLE
POLYTECHNIQUE
MONTRÉAL