Recommending Adaptive Changes for Framework Evolution

BARTHÉLÉMY DAGENAIS and MARTIN P. ROBILLARD, McGill University

In the course of a framework's evolution, changes ranging from a simple refactoring to a complete rearchitecture can break client programs. Finding suitable replacements for framework elements that were accessed by a client program and deleted as part of the framework's evolution can be a challenging task. We present a recommendation system, SemDiff, that suggests adaptations to client programs by analyzing how a framework was adapted to its own changes. In a study of the evolution of one open source framework and three client programs, our approach recommended relevant adaptive changes with a high level of precision. In a second study of the evolution of two frameworks, we found that related change detection approaches were better at discovering systematic changes and that SemDiff was complementary to these approaches by detecting non-trivial changes such as when a functionality is imported from an external library.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Documentation, Experimentation

Additional Key Words and Phrases: Adaptive changes, framework, legacy study, mining software repositories, origin analysis, partial program analysis, recommendation system, software evolution

ACM Reference Format:
Dagenais, B. and Robillard, M. P. 2011. Recommending adaptive changes for framework evolution. ACM Trans. Softw. Eng. Methodol. 20, 4, Article 19 (September 2011), 35 pages.
DOI = 10.1145/2000799.2000805 http://doi.acm.org/10.1145/2000799.2000805

1. INTRODUCTION

Application frameworks support large-scale reuse and free developers from low-value programming tasks. By developing *clients* that integrate with the framework code, developers are able to customize and enhance the framework to suit their specific needs. However, as the framework evolves, changes ranging from a simple refactoring to a complete rearchitecture can break client programs. To lower the cost of adapting client programs to changes in the framework, framework developers rely on a variety of techniques such as automatically capturing and documenting some of their changes [Dig et al. 2007; Henkel and Diwan 2005], providing migration paths [Chow and Notkin 1996], or deprecating existing methods and indicating new replacements [des Rivières 2007]. Current tools, however, cannot capture changes more complex than refactorings, and manually documenting a framework's evolution is not always cost-effective, especially for fast-evolving frameworks. Although framework users are encouraged to use only published Application Programming

---

This article is a revised and extended version of a paper presented at ICSE 2008.
Authors' address: B. Dagenais and M. P. Robillard, School of Computer Science, McGill University, 3480 University Street, McConnell Engineering Building no. 318, Montreal, Quebec, Canada, H3A 2A7; email: {bart, martin}@cs.mcgill.ca.

DOI 10.1145/2000799.2000805 http://doi.acm.org/10.1145/2000799.2000805

Interfaces (API) because they usually provide a stable interface around evolving features [Larman 2001], developers often use internal and undocumented parts of frameworks for a variety of good reasons, such as accessing functionality that goes beyond the ones available in the published interfaces [Boulanger and Robillard 2006]. These internal parts are accessible because they are declared as *public* in their programming language, but they are not officially *published* by framework developers [Fowler 2002].

A previous study of API evolution found that more than 80% of API-breaking changes were caused by refactorings and concluded that techniques aiming at documenting or detecting refactorings were desirable [Dig and Johnson 2006]. The authors of that study also mentioned that "Application developers will have to carry only a small fraction [less than 20%] of the remaining changes. These are changes that require human expertise" [Dig and Johnson 2006, p.105]. To detect the largest portion of API-breaking changes, that is, refactorings, several approaches have been proposed [Dig et al. 2006; Godfrey and Zou 2005; Kim et al. 2007, 2005; Weissgerber and Diehl 2006; Xing and Stroulia 2006].

Although refactoring detection techniques partially automate the tedious task of identifying and repairing small changes such as a renamed method, refactorings tend to be minor changes easily identified through a manual inspection. Indeed, refactorings usually involve only one change dimension: name or location. For example, if a method is no longer accessible in the new version of a framework, a developer can often simply perform a lexical search ("grep") to find similarly named methods (name dimension) or can look in the same module to find potential replacements (location dimension). However, it will generally be harder to repair a client program if the framework went through major modifications that led to nontrivial changes (e.g., a composition of simple refactorings).

To help developers repair client programs that are affected by the nontrivial evolution of a framework, we propose an approach to recommend adaptive changes, a form of maintenance aiming at adjusting a software system to comply with its technological environment [Lientz and Swanson 1980]. Our idea is to automatically analyze how the framework was adapted to its own changes, and to recommend similar adaptations. Basically, if a method `m1` is removed from the framework code, we can identify all of the callers of `m1` within the framework and analyze how they were adapted to the removal of `m1`.

We implemented this approach for Java in a client-server application called SemDiff. The SemDiff server component is responsible for analyzing the source code repository of a framework and for inferring high level changes such as method additions and deletions. The client component takes as input calls to methods that no longer exist in a framework and produces recommendations in the form of recommended replacement methods, accompanied by a confidence value.

We evaluated the effectiveness and the need for our approach by using SemDiff in a legacy-based study [Zelkowitz and Wallace 1998] of one framework in which we recommended adaptive changes for three broken client programs. We then compared our results with the recommendations of a typical refactoring detection tool. Our study showed that our approach provided a relevant functionality replacement for 97% of the broken methods, detected nontrivial changes that were more complex than refactorings, and could recommend methods from external libraries that replaced a framework's functionality, a change that is typically not detected by other techniques.

To better understand the strengths and limitations of SemDiff, we inspected 100 methods removed as part of the evolution of two other frameworks, and we compared the recommendations of SemDiff with the recommendations of two change detection

techniques. We found that SemDiff was often complementary to the other approaches and that only 10% of the methods removed could not be replaced by any of the three approaches.

The contributions of this article include (1) a technique to automatically recommend adaptive changes in the face of nontrivial framework evolution, (2) the architecture of a complete system to track a framework's evolution and infer nontrivial changes, and (3) a body of empirical evidence detailing the strengths and limitations of current change detection techniques.

In the remainder of this article, we present a sample scenario that illustrates the difficulties associated with nontrivial framework evolution (Section 2). We then describe the principles and implementation details underlying our approach (Section 3). We present a legacy-based study on the evolution of the Eclipse Java Development Tool (JDT) framework (Section 4) and a qualitative study on the evolution of two other frameworks and three change detection techniques (Section 5). We conclude with an overview of the related work (Section 6) and a summary of our conclusions (Section 7).

2. SAMPLE SCENARIO

Let us now consider the case of a developer who decides to reuse internal classes of the Eclipse framework in a client program. This scenario is based on the actual evolution of the Eclipse code-base. In Eclipse, classes that are in a package containing the word *internal* are, by convention, not part of the supported API. It is generally understood that internal classes can change from one version of the framework to the other and that no documentation (e.g., javadoc) or migration path is provided.

One of the classes the developer considers for reuse, `org.eclipse.jdt.internal.corext.util.TypeInfo`, is contained in the `org.eclipse.jdt.ui` plug-in in Eclipse release 3.2. This class, along with several others, such as `TypeInfoFactory`, `TypeInfoUtil` and `OpenTypeHistory`, provides services for searching and displaying Java type information (e.g., a type name, package name, or access modifiers).

When Eclipse 3.3 is released, the developer loads the client project in the development environment, which automatically tries to build the client program against the new version of the framework. At this point, the compiler generates multiple compilation errors, because the `TypeInfo` class is no longer accessible in Eclipse 3.3. The developer then explores the source code of the new version of Eclipse in the hopes of finding a suitable replacement for these missing methods, searching for a class with a name similar to `TypeInfo`. Seeing a few classes named `TypeInfo`, the developer realizes that they are defined in external libraries and that no similarly named classes provide the required functionality. The developer then moves on to see if the missing class is in the same package but under a different name. Again, no class is found with the same functionality. Moreover, the `TypeInfoFactory` and `TypeInfoUtil` classes also have disappeared from this package.

Having ruled out a simple refactoring, the developer then looks at other classes that used to depend on `TypeInfo` and sees that some of them now refer to the `org.eclipse.jdt.core.search.TypeNameMatch` class. Unfortunately, the developer finds that this class is not a perfect replacement for `TypeInfo`, because `TypeNameMatch` resides in another plug-in (`org.eclipse.jdt.core`), and its interface is much smaller (8 methods declared in `TypeNameMatch` versus 23 methods in `TypeInfo`). At this point, the developer still does not know how to replace one of the missing classes and it becomes clear that reverse-engineering part of the framework will be necessary.

As illustrated by Figure 1, our approach, SemDiff, can make the process of adapting a client program to an evolving framework more efficient by (1) providing adaptive

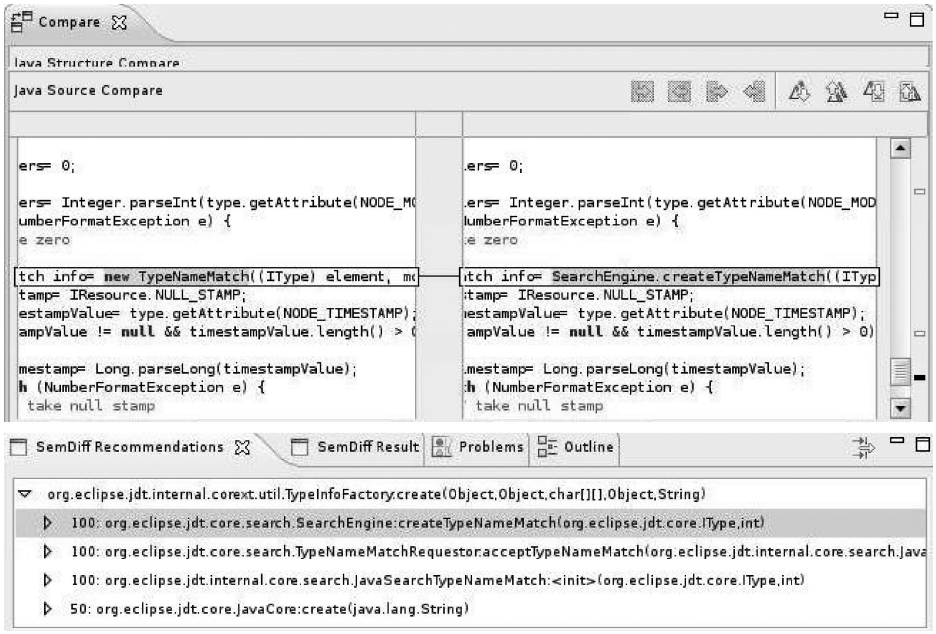

Fig. 1. The SemDiff Recommendations View (bottom) displays the recommendations to replace the TypeInfoFactory.create() method and the Compare Editor (top) shows how the framework was adapted to its own changes: the create() method was replaced by the TypeNameMatch constructor, which, in turn, was replaced by the createTypeNameMatch() method.

change recommendations (bottom frame), and (2) providing a source code example that illustrates how the framework was adapted to its own changes (top frame). In contrast to current approaches that display a list of refactorings [Dig et al. 2006; Kim et al. 2005; Weissgerber and Diehl 2006] or name transformation rules [Kim et al. 2007] and that require the user to figure out what the relevant refactorings are in a given situation, SemDiff starts with a request to repair a broken call and returns a list of potential replacements ordered by a confidence value. By providing examples extracted from the framework's source code, SemDiff can also help developers validate the recommendations and choose among alternatives.

3. SEMDIFF

To recommend adaptive changes when a framework method is removed, we hypothesize that, generally, calls to deleted methods will be replaced in the same *change set* by one or more calls to methods that provide a similar functionality. A change set, also called commit or transaction, contains all the changes that were performed by the framework developer and committed to a source control system like CVS.[1] Usually, change sets are associated with only one or a few maintenance tasks, so we assume that methods that are removed and added in the same change set are closely related.

To find replacement methods, we must analyze the history of the framework. Figure 2 provides an overview of the SemDiff implementation. SemDiff consists of a

[1]www.nongnu.org/cvs/

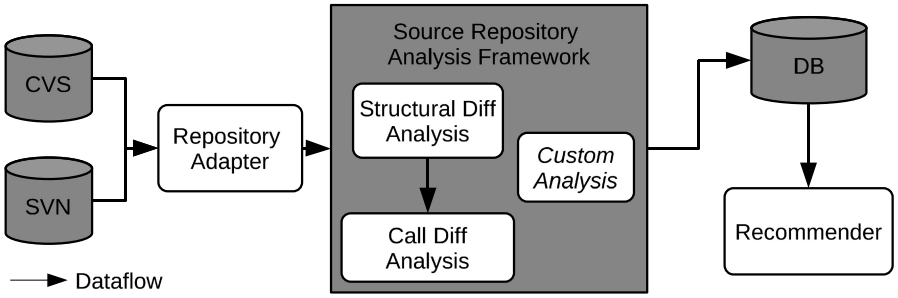

Fig. 2. SemDiff overview.

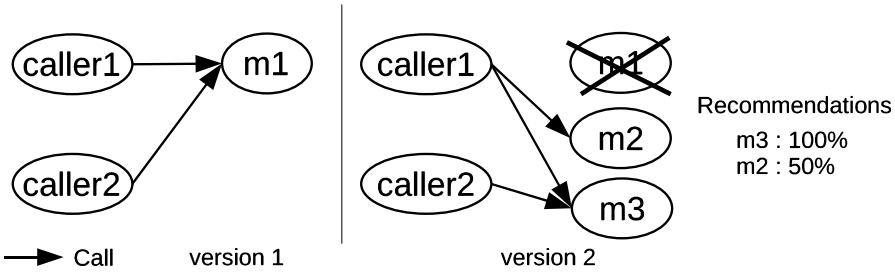

Fig. 3. Using method call changes.

client component (the recommender) and a server component (represented by the source repository analysis framework). We first present the main strategies underlying the recommender, describe how the server infers high level changes from the source repository, and cover in detail the analyses performed by the server.

3.1 Adaptive Change Recommendations

Developers can send requests to the recommender to receive suggestions of adaptive changes. With SemDiff, a developer selects a call that can no longer be resolved with the new version of a framework (e.g., a call to a method of the `TypeInfo` class presented in Section 2), and queries the recommender for potential replacements. The recommender then formulates recommendations by analyzing the high-level changes inferred by the source repository analysis framework.

3.1.1 Using Call Differences. We use differences in the outgoing calls of a given method during a framework's history to find out how the framework was adapted when a method was removed. For example, in Figure 3, method `m1` is removed between two versions. If we want to find a suitable replacement for `m1`, we first find all of the methods where a call to `m1` was deleted (e.g., methods `caller1` and `caller2` in Figure 3). Then, we gather all calls that were added in these methods as part of the same change set (e.g., `m2` and `m3`). Since we expect that methods might be adapted along with additional changes, we sort added calls by a *confidence metric* to provide a ranking of the potential replacements. Assuming that we only look for change sets made prior to a target version v, we define the confidence value of a method n that replaces a call to method m with the following equations (v is an implicit parameter of all equations).

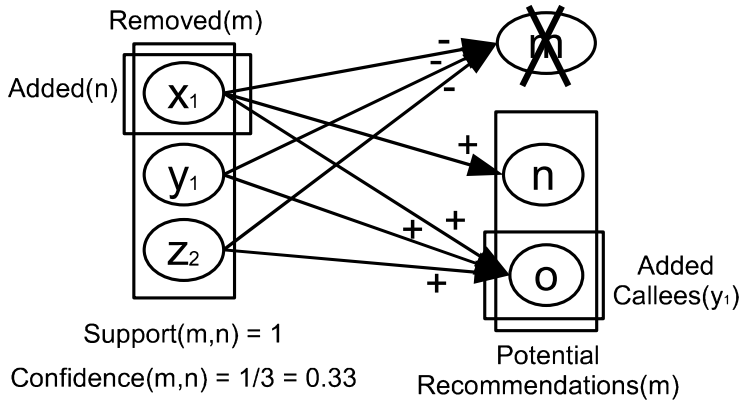

Fig. 4. Support and confidence of a replacement. The subscript numbers refer to corresponding change sets.

Removed(m) := $\{x_i \mid x$ is a method that removed a call to m at version $i\}$

Added(m) := $\{x_i \mid x$ is a method that added a call to m at version $i\}$

Added Callees(m_i) := $\{c \mid m$ added a call to c at version $i\}$

Callers(m) := $\{c \mid c$ calls $m\}$

$$\textbf{Potential Recommendations}(m) := \bigcup_{x_i \in Rem(m)} Added\ Callees(x_i)$$

$$\textbf{Support}(m, n) := |Removed(m) \cap Added(n)|$$

$$\textbf{Confidence}(m, n) := \frac{Support(m, n)}{Max\left(\bigcup_{c \in Potential(m)} Support(m, c)\right)}$$

Figure 4 shows an example of each of these definitions in the context of requesting a replacement for method m. The confidence metric of a recommendation n to replace a method m is the ratio of the recommendation's support to the maximum support for all potential recommendations. The confidence metric is therefore a normalized value with a range of $]0, 1]$ that is used to compare the recommendations.[2] For example, if o is the recommendation with the highest support, 3, and recommendation n has a third of this support, 1, n will have a confidence of 0.33. Because the support is bounded by the number of callers of the removed method ($|Removed(m)| \leq |Callers(m)|$), we hypothesize that the framework will have a sufficient amount of calls to its own methods so pertinent recommendations can be discriminated from spurious ones. Section 4.3 shows how this hypothesis held in practice.

Recommendations that have the same confidence value are further sorted by their similarity with the name of the queried method: the recommendation that has a name with the *longest subsequence* common with the name of the queried method is presented first. The rationale is that methods with similar names probably exhibit a similar behavior: this hypothesis forms the basis of Kim et al.'s approach [Kim et al. 2007].

[2]To be recommended, a replacement method must have a support of at least one, so the confidence value of a recommendation is always strictly greater than zero.

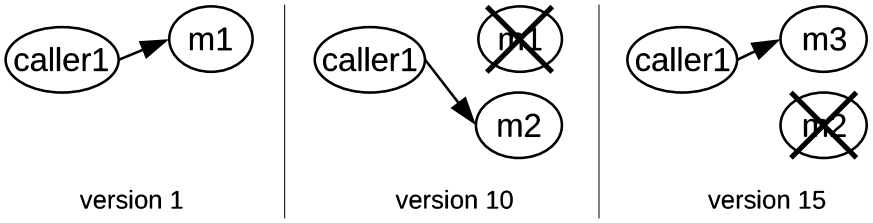

Fig. 5. Change chain.

When looking for a replacement for a method m, we search for the methods that removed a call to m as opposed to all callers of m. Moreover, we do not need to take into account whether method m still exists. This enables us to get recommendations for methods that are replaced but that are not deleted yet (e.g., deprecated methods). This is one difference with previous approaches that use the addition and deletion of methods as the basis for detecting changes and refactoring [Godfrey and Zou 2005; Kim et al. 2007, 2005; Weissgerber and Diehl 2006; Xing and Stroulia 2006].

As shown in Figure 4, SemDiff can compute recommendations from multiple change sets: a call to method o was added by methods x and y in version 1 and by method z in version 2.

In practice, our hypothesis that calls to a removed method are replaced by the callers in the same change set might not hold for all framework evolution scenarios. We now discuss the strategies that we have designed to take into account these possible variations to our hypothesis and we also review the impact of these strategies on the computation of the confidence value.

3.1.2 Change Chains. It is possible that during the course of a framework's evolution, a method is replaced several times, that is, it is part of a *change chain*. As illustrated by Figure 5, a method might be renamed once in one version and renamed a second time in another version. Additionally, because we study the evolution of a framework at the change set level, it is probable that we will come across small changes that were never accessible to client programs (e.g., a method name was misspelled and corrected in the next change set, a developer reverted to the old version of a class, etc.).

To account for these situations, we must slightly modify the strategy defined above to detect whether a method is part of a change chain. Indeed, we do not want to recommend a method that changed subsequently and that is no longer accessible or appropriate. One solution would be to automatically check if the recommended method exists in the framework version used by the client program. Unfortunately, this is not an adequate solution if the method is no longer used by the framework but was not removed (e.g., the method was deprecated). We thus rely on a different heuristic to determine whether a recommendation is part of a change chain. If we find that *some* methods calling the recommended method removed a call to the recommended method later, we conclude that our recommendation is *probably* in a change chain and the initial recommendation might still be valid. If we find that *all* methods calling the recommended method removed the call to the recommended method, we conclude that this recommendation is part of a change chain and we discard the recommendation because it is no longer relevant.

Once we have identified a recommendation as being part of a change chain, we reapply the call difference analysis described above to find a more relevant recommendation. This is illustrated in Figure 5, where our system would recommend replacing a call to `m1` by a call to `m3`. To improve the performance of our approach, SemDiff does not attempt to find change chains for recommendations with a confidence value below

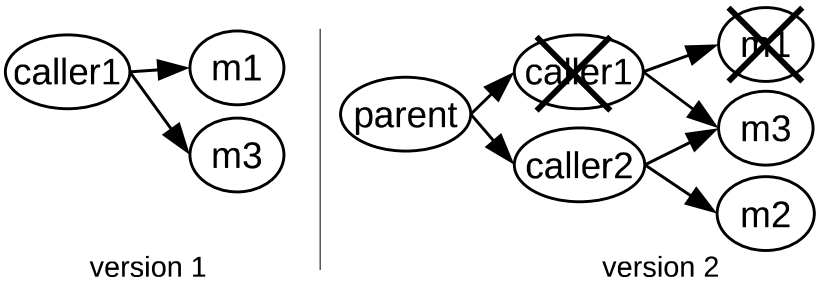

Fig. 6. Callee and caller are deleted together.

a certain threshold (0.6): these recommendations are likely to be less useful to the developer in the first place. We chose this threshold after trying values from 0.5 to 1.0 during early experimentation on the approach: we found that a value 0.6 offered the best compromise between accuracy and performance.

Impact on the confidence value. During the computation of a recommendation, if SemDiff finds that a method removed a call to the recommendation (e.g., `caller1` removed a call to the recommendation `m2` in Figure 5), SemDiff decreases the support of the recommendation by one effectively decreasing its confidence value; once the support drops to zero, we discard the recommendation. When a recommendation is part of a change chain, the confidence value of the subsequent recommendations (e.g., `m3` is recommended to replace `m2` in Figure 5) is computed as usual, that is, the support of a subsequent recommendation is divided by the maximum support of all recommendations for the query.

3.1.3 Caller Stability. Because our strategy only relies on the outgoing call relationship, it is sensitive to the stability of callers throughout the framework's evolution. For example, Figure 6 shows a situation where both the requested method, `m1`, and the caller, `caller1`, are deleted in the same change set. In this situation, the caller cannot be used to find a replacement for the requested method. To cope with this issue, we first need to find a replacement for the deleted caller and then, we can recommend the methods that are called by the caller replacement (we remove the methods that were previously called by the deleted caller). Figure 6 illustrates the case where `caller1` was replaced by `caller2` and `caller2` replaced a call to `m1` by a call to `m2`. Method `m3` is not recommended because `caller1` was already calling it. The ability to deal with unstable callers is a significant difference with Schäfer et al.'s approach that also analyzes callers to find replacements [Schäfer et al. 2008].

When finding a replacement for a caller method, SemDiff can generate multiple recommendations (e.g., the method was split, there are truly multiple relevant replacements, false positives, etc.). To reduce the impact of false positives, we first remove from the potential replacements all recommendations that have a confidence value below a certain threshold (0.6) and then, we perform the call difference analysis on each of the remaining recommendations.

Impact on the confidence value. When computing the confidence value of a recommendation, stable and unstable callers are considered equal. For example, in Figure 6, the support of `m2` would be one, even if the original caller, `caller1` was unstable and the replacement caller, `caller2`, came from a recommendation with a confidence value potentially below 100%. The advantage of this strategy is that the computation of the support is simpler and easier to understand: it represents the number of callers. The

removed := $max(|Removed(m)|)$

acal := $max(|Added\ Callees(m_i)|)$

rec := $removed \times acal$

chain := maximum length of a change chain

unstable := maximum length of unstable caller chain

$$removed \times acal = O(removed \times acal) = O(rec) \quad (1)$$

$$rec + removed = O(rec) \quad (2)$$

$$\sum_{i=0}^{chain} rec^{i+1} = O(rec^{chain}) \quad (3)$$

$$removed \times (\sum_{i=0}^{unstable} rec^{i} \times acal) = O(rec^{unstable}) \quad (4)$$

Fig. 7. Components of the complexity of SemDiff's algorithm.

disadvantage is that wrong replacement caller with a low confidence value might boost the support of spurious recommendations. This problem is limited to a certain extent by the use of the threshold discussed previously.

3.1.4 Spurious Call Removal. When finding a replacement for a method `m1`, SemDiff looks for change sets where a call to `m1` was removed because this is typically where the framework will be adapted to a change concerning `m1`. It is possible though that a call to `m1` is removed in one place and added in another place in the same change set (e.g., the caller was refactored, the caller was made more cohesive and the call to `m1` was moved elsewhere). In these cases, the framework is not being adapted to the loss of `m1` since it is still calling it elsewhere. To make sure our analysis does not take into account these spurious call removals, SemDiff will ignore all change sets where the requested method call (e.g., `m1`) was removed from one caller and added in another caller.

Impact on the confidence value. When we ignore a change set where spurious call removals happened, we potentially reduce the number of recommendations and decrease the support of recommendations. While the former consequence reduces the number of false positives, the latter consequence makes the confidence value more meaningful by only considering callers where a relevant replacement happened.

3.1.5 Complexity. As shown in Figure 7, the main factors affecting the computational complexity of SemDiff's algorithm are the number of methods that removed a call to the queried method (*removed*), the number of different added callees (*acal*), the maximum change chain length (*chain*), and the maximum length of an unstable caller chain (*unstable*).

The maximum number of recommendations (*rec*), can be approximated by multiplying the number of methods that removed a call to the queried method by the number of added callees. Assuming that searching for these methods is efficient, the number of

recommendations represents the basic complexity of SemDiff's algorithm and is given by Equation (1).

The effect of the spurious call removal on the complexity is given by Equation (2): SemDiff needs to analyze the transaction of each method that removed a call to the queried method (*removed*) to ensure that the call was not also added elsewhere in the transaction. This strategy has a negligible impact on the complexity.

The strategy to detect change chains has a significant impact on the complexity of SemDiff's algorithm. As shown in Equation (3), the number of recommendations inspected is superpolynomial because we need to find recommendations for each recommendation in a change chain. For example, in Figure 5, SemDiff had to find a recommendation to replace the recommendation *m2* because it was part of a change chain. In our experience with SemDiff, the length of a chain between two major releases of a software system never exceeded two. In practice, we expect change chain lengths to be small (e.g., below 10), so the algorithm should be polynomial. Moreover, to minimize the impact of this strategy on the performance of SemDiff, we only consider recommendations in a chain that have a confidence value superior to a given threshold (0.6).

Finally, the strategy to detect unstable callers, without change chain detection, also makes the complexity of SemDiff's algorithm superpolynomial. This is because we need to compute a set of recommendations for each unstable caller which potentially has an unstable caller too. Once we have determined a list of potential replacements for the unstable callers, we can consider the added callees. Because the strategy to detect unstable callers can significantly impact the performance of SemDiff, especially when combined with the change chains detection strategy, we limit the length of unstable caller chains to two. If the length of the chain exceeds that limit (e.g., the parent of the parent of the unstable caller is unstable), SemDiff aborts the search and issues no recommendation. This threshold has the additional advantage of limiting the imprecision brought on by the unstable caller strategy.

In practice, we never encountered a recommendation for which SemDiff had to apply both the unstable caller strategy and the change chain strategy. Because the complexity of each strategy is superpolynomial, the combination of the two strategies would also be superpolynomial. Considering the length of call chains in practice and the thresholds we use to limit the computation of unstable callers, we conclude that the algorithmic complexity of SemDiff's algorithm is polynomial in practice, but superpolynomial in theory.

3.1.6 Viewing Recommendations. The recommendations produced by SemDiff are presented to the user in the Eclipse development environment and take the form of a list of methods, ranked by their confidence value. The user can also double-click on a recommendation to open the Eclipse compare editor with one example where the recommended method replaced the broken call. This allows the user to understand why this particular recommendation was provided and see how the framework was adapted to this change.

3.1.7 Current Limitations. A number of factors constrain the applicability of our approach. The most important limitation is that the framework cannot issue recommendations for root methods, that is, methods that are never called within the framework. Classes that are called back by other libraries or protected methods that are called by a parent class not in the framework will hinder the ability of our approach to make adaptive change recommendations. One solution to the problem of root methods is to include in our analysis example programs that depend on the framework and that were adapted to its various versions: this is the strategy taken by Schäfer et al. [2008] We show in Section 5.2 that root methods are the main limitation of SemDiff.

Although our approach can make multiple recommendations per request, it does not group the recommendations together. For example, a call to method `m1` might have been replaced by a call to methods `m2` and `m3`. Our approach will present these two calls as two separate recommendations with, possibly, a different confidence value. To help a user identify relationship between recommendations, SemDiff displays the code where the changes happened: automatically inferring these relationships remains an area for future work.

Finally, adaptive changes proposed by SemDiff might not be semantically equivalent to features that need to be replaced, and blindly applying the recommended changes without proper testing could lead to serious flaws. Other research projects currently aim at generating test cases to ensure that the semantic of a program is preserved when applying a refactoring [Daniel et al. 2007; Jagannath et al. 2009]: it is possible that our approach could directly benefit from such developments.

3.2 Analyzing Source Code History

To provide adaptive change recommendations, SemDiff must first analyze a framework's source code repository by (1) retrieving the files and change data for each version of the framework, (2) running several analyses to infer high level changes such as structural and method call differences, and (3) storing these high level changes in a database for future use by our recommendation system.

Source Repositories. Currently, SemDiff provides adapters to retrieve information from CVS and Subversion (SVN)[3] repositories. SVN has the concept of change set embedded in its protocol, which makes it easy to retrieve and group files that were changed together. In contrast, CVS does not group file changes and some preprocessing of the repository log file is needed to infer change sets. We employed a technique previously used to mine CVS repositories to retrieve the change sets [Weissgerber and Diehl 2006; Zimmermann et al. 2005]: we group all log entries that occurred within a certain time window (300 seconds) and that shared the same user and log message. This concept of time window is essential to capture transactions that could span across multiple seconds [Zimmermann and Weissgerber 2004].

The merging of branches is another issue that arises when analyzing software repositories. This operation is not explicitly documented by either CVS or SVN. Detecting merges is important because we do not want to analyze the same change twice: one in the branched version and one in the merged version. To detect merges, we employed a simple heuristic used in previous work on source repository mining [Weissgerber and Diehl 2006; Zimmermann et al. 2005]: we ignored change sets involving more than 40 files.

Change Analysis. For each change set, SemDiff can run custom analyses to infer high level information such as the removal or addition of methods from the raw line difference data provided by the repository. This high-level information is then used by our recommender. Because we only retrieve the files that were added, removed or modified in each change set, we always perform analyses on a subset of the program, which limits the types of analysis that we can perform. For example, it might be impossible to fully resolve the return type of a method called by a class in a change set, but declared in a class outside the change set. Analyzing each change set (as opposed to analyzing major revisions) potentially makes the analysis of the program evolution easier because we can break down a nontrivial change into smaller and incremental changes (i.e., change sets). Combining the granularity of the change set with the quality of full

[3]subversion.tigris.org

```
// Bar.java, version 1.1
import java.util.List;
class Bar extends Foo {
  private void m1 {
    List l = new List();
    l.add(new Object());
  }
  private void m2 {
    super.m3();
    m4();
  }
}


// Bar.java, version 1.2
import semdiff.List;
class Bar extends NewFoo {
  private void m1 {
    List l = new List();
    l.add(new Object());
  }
  private void m2 {
    super.m3();
    m4();
  }
}
```

Fig. 8. Class changes impacting method bodies.

program analysis by building the whole framework after having retrieved each change set would be possible but not practical: project configuration (e.g., how to build the project) can evolve over time and differs from one project to the others and performing full program analysis on thousands of change sets would take too much time to be valuable.

We perform two analyses on every change set in the framework's version history. The first analysis, StructDiff, produces a list of all methods, fields, and classes that were added, removed, and modified. To determine whether a method has been modified, StructDiff compares the abstract syntax tree of the two versions of the method: if the two subtrees are not equal, StructDiff concludes that the method was modified. A simple prefix tree walk is used to determine the equality of the subtrees: to be considered equal, two nodes must be at the same position in the tree, they must be equal (e.g., same method call) and they must have the same children in the same order. Tree matching is more precise than simple textual comparison because it considers identifiers, but ignores changes in comments and source format. Compared to the first version of SemDiff [Dagenais and Robillard 2008], StructDiff now conservatively detects that a change outside of a method body might have impacted the method. For example, in Figure 8, the structure and the text of methods `m1` and `m2` did not change, but the semantics are different because of the modified import statement and superclass. StructDiff would detect that these two methods changed: in the case of `m1`, the body refers to a modified import, `List`, and in the case of `m2`, the body refers to a super method and a method that is not declared in `Bar`.

The second analysis, CallDiff, finds the calls that were added or removed between two versions of each method identified by StructDiff. For example, in Figure 9, CallDiff would indicate that between version 1.1 and version 1.2, the call `PrintStream.println()`

```
// method m1 in Bar.java, version 1.1
private void m1 {
  System.out.println();
}

// method m1 in Bar.java, version 1.2
private void m1 {
  System.out.print(Math.random());
}
```

Fig. 9. Outgoing call differences.

was removed and the calls `Math.random()` and `PrintStream. print(double)` were added. Because we perform this static analysis on a subset of the program source, we must rely on a custom parser and analyzer presented in Section 3.3.

Persisting changes. After the execution of the analyses for a change set, the results are stored in a PostgreSQL database[4] or an embedded HSQLDB database [5] and made available to our recommender. Using PostgreSQL offers better performance, while HSQLDB is easier to configure.

Performance and Scalability. The change analysis is only performed once on each change set. When a new change set is committed, the change analysis can be performed on this new change set without having to go through the previous change sets. In practice, we found that half of the analysis time was spent on retrieving the source file versions over the Internet. The number of files to download is limited to 40, the threshold we use to detect branch merges.

Most of the remaining analysis time is spent executing partial program analysis to perform the call difference analysis. Typically change sets with large files such as generated parser classes take the most time: large files could potentially be explicitly excluded to speed up the analysis. The total size of the program (e.g., lines of code or number of classes) only impacts the performance of the analysis if we assume that large programs have more change sets than small programs. The whole program is never analyzed completely so its size is not a limiting factor. We provide the performance data that we measured during our evaluation study in the next section.

3.3 Partial Program Analysis

With Java, most parsers and static analysis programs cannot reconstruct the complete type hierarchy if they only receive as input a subset of the program source code, without the dependencies and the rest of the program (in the form of source or binary). A few parsers, like the one provided by the Eclipse Java Development Tool framework, can construct abstract syntax trees (AST) from a subset of the program source code, but the information they provide is incomplete. For example, in Figure 10, the Eclipse parser would be able to recognize that in method `doSomething()`, there is a call to a method named `x` at line 7, but it would not indicate its target, an object of type Y, because it is an unknown class. To enable the analysis of partial programs, we extended the Eclipse Java compiler [Dagenais and Hendren 2008].

Our implementation of *Partial Program Analysis* first replaces every occurrence of unknown types by a placeholder type: `UNKNOWN`. Then, it tries to infer the actual type of the placeholder types by analyzing how the various unknown types are

[4]www.postgresql.org
[5]hsqldb.org

```
1:  import package1.*;
2:  import package2.*;
3:  import package3.Y;
4:
5:  class Foo {
6:    void doSomething(Y obj) {
7:      obj.x();
8:      obj.a = 2.2;
9:      doThis(Util.method2(obj,obj.a));
10:      Util.method3().method4();
11: }
12:
13:   void doThis(Z z) {
14:     System.out.println(z);
15:   }
16: }
```

Fig. 10. Partial program analysis example.

used. For example, our analyzer would infer that the following methods are called in `doSomething(Y)`: `Y.x()`, `Util.method2(Y,double)`, `Foo.doThis(Z)`, `Util.method3()`, and `UNKNOWN.method4()`.

Although analyzing partial Java programs is inherently unsound (e.g., without the whole program, it is sometimes impossible at compile time to determine whether an expression is a package, a class, an internal class, or a field), we found that on average, our technique correctly resolved 91.2% of the types when analyzing one class at a time and only produced 2.7% of erroneous types [Dagenais and Hendren 2008]. In practice, this level of precision means that most of the time, the inferred type is in the hierarchy of the declared type, that is, the inferred type is a subtype, a supertype or the same type as the declared type. For example, our analyzer infers that the method `Util.method2()` on line 9 returns an object of type Z, even if this method might be defined to return a *subtype* of Z. It follows that the inferred result, Z, is not equal to the declared type, but it is in its hierarchy. Inferring a *hierarchy-related* type is still more precise than inferring that the return type is `UNKNOWN` or `Object`.

The recommender also needs to take into account that the type information of a call might be incomplete. In the worst case, it might be impossible to know the target and the parameter types of a call. This is the case of method `m1` in the following example, if we do not have the definition of `myObj`'s type.

```
myObj.myMethod().m1(myObj.myOtherMethod())
```

Polymorphism can also be an issue. In the next example, the calls at line 2 and 3 refer to the same method, but our partial program analysis would treat them as two different calls, `ArrayList.add(Object)` and `ArrayList.add(String)`, if we do not have the definition of `ArrayList`.

```
1: List list = new ArrayList();
2: list.add(new Object());
3: list.add(new String());
```

This lack of accurate type information can be a serious problem for common method names, such as `add` and `remove`. If we want to find a replacement for a method `add(Object)` that is no longer accessible, we cannot search for all methods that removed a call to a method named `add` with one parameter: we would probably retrieve a lot of false positives coming from other irrelevant classes that defined a similar method.

Currently, we try three matching criteria, starting from the strictest one, until we can find calls. We first try to find methods that removed a call sharing the same name, number of parameters, and target type as the call we want to replace. If we do not find such methods, we try to match calls that share the same name, number of parameters, and parameter types. Finally, if this still does not return any results, we then try to find calls only by their name and number of parameters.

4. LEGACY-BASED EVALUATION STUDY

The main strategy underlying SemDiff relies on a number of hypotheses we made about framework evolution. We designed a study to assess the validity of these hypotheses and to evaluate the effectiveness of our approach. This study helped us answer the following questions.

(1) Is the confidence metric we use able to discriminate relevant recommendations from false positives?
(2) In addition to simple refactorings such as *rename method*, can SemDiff detect other types of changes?

4.1 Study Design

To answer these questions, we performed a legacy study of one framework and three client programs. We used SemDiff to adapt an old version of a client program to the new version of the framework. We then compared our adaptation recommendations to the historical (real) adaptation of the client program. To evaluate the complexity of the changes that occurred during the framework's evolution, we also used a refactoring detection tool to analyze the framework's history and provide recommendations to client programs.

In summary, for each client program, we selected two versions (`c1`,`c2`): one that was using an old version of the framework (`f1`) and one that was using the most recent version (`f2`). We then tried to compile the `c1` version of each client program with the `f2` version of the framework. For each method call in the client program that could not be resolved, we used the SemDiff recommender and a refactoring detection tool to find a suitable replacement for the broken method call. We then analyzed the `c2` version of the client program to see if the recommended methods were called.

Target systems. We chose the Eclipse Java Development Tool (JDT) platform as the framework to analyze in our study. This framework is large enough to provide evidence that our approach scales, its source history is publicly available, it is actively maintained and has a large ecosystem of client programs. We chose to study two modules of this framework, the `org.eclipse.jdt.core` and `org.eclipse.jdt.ui` plug-ins from version 3.1 to 3.3. These plug-ins are mainly responsible for the Java compiler and Java editor in the Eclipse development environment and they went through a nonnegligible evolution between these two releases. In our experience, client programs that depend on JDT always depend on at least one of these two plug-ins. From release 3.1 to release 3.3, the `jdt.core` and `jdt.ui` plug-ins grew respectively from 222 to 261 kLOC and from 256 to 311 kLOC. Additionally, 2,835 out of the 28,671 public and protected methods of the `jdt.core` and `jdt.ui` plug-ins were removed between versions 3.1 and 3.3 and 7548 methods were added. Many features were implemented between these two versions such as support for Java 6, inline refactoring in the editor, improved Java parser, and new compiler warnings. We chose to study these plug-ins across three major revisions (3.1, 3.2, and 3.3) to increase the odds of finding nontrivial changes and change chains.

Table I. Client Program Versions

Client	Eclipse 3.1	Eclipse 3.3
Mylyn	0.5	2.0
JBoss IDE	1.5	2.0
jdt.debug.ui	3.1	3.3

Finding suitable client programs was a harder task. We needed client programs that (1) depended on JDT, (2) had been adapted to the two versions of the framework we studied (3.1 and 3.3), and (3) replaced a functionality that disappeared during the framework's evolution by a functionality provided by the last release of the framework. We ran into several cases where the last condition was not met. For example, the AspectJ Development Tool[6] client program copied entire classes from JDT release 3.1 into its own code base instead of calling a new JDT functionality. Another JDT client program, the Eclipse Modeling Framework,[7] replaced a deprecated framework functionality with its own implementation. We could not use such client programs because they did not provide an oracle for the quality of the recommendations. However, such dramatic adaptation strategies further motivate our work by providing anecdotal evidence that adapting client code to new versions of a framework is a challenging and costly endeavor: it is possible that the developers of these client programs would have used the new JDT functionality if an easier migration path had been provided.

We found three client programs that met our study criteria: **Mylyn** [Kersten and Murphy 2006], a task-focused environment, **JBoss IDE**, a development environment for the JBoss web application server, and **jdt.debug.ui**, the Java debugging environment in Eclipse. Table I gives the client program versions used for Eclipse 3.1 and Eclipse 3.3.[8]

SemDiff. We used SemDiff to analyze the source history of the Eclipse framework. SemDiff processed 10,408 change sets for the two jdt plug-ins in the Eclipse CVS repository from January 2005 to July 2007, as Eclipse 3.1 and 3.3 were respectively released on June 27, 2005, and June 25, 2007. Because work on Eclipse 3.2 might have begun before the release of Eclipse 3.1 (e.g., in a branch), we started to study the framework in January 2005 instead of June 2005.

Once we analyzed the framework's source history, we tried to compile the first version of the client programs with Eclipse 3.3. For each call to a framework method that could not be resolved by the compiler, we ran the SemDiff recommender and noted its recommendations. We then looked at the version of the client program that had been adapted to Eclipse 3.3: if the client program called one of the top three recommendations for each broken call, we considered it to be a relevant recommendation.

RefactoringCrawler. We also used a typical refactoring detection tool to discriminate nontrivial changes that occurred during the framework's evolution from simple refactorings. We chose RefactoringCrawler [Dig et al. 2006], as it was easy to use, configurable, readily available, and representative of several refactoring detection techniques; a more detailed comparison of such techniques is given in Section 6. In essence, RefactoringCrawler takes two complete versions of a project as input and gives a list of refactoring pairs (e.g., method `m1` was renamed to method `m2`).

[6]www.eclipse.org/ajdt/

[7]www.eclipse.org/modeling/emf/

[8]In a previous study [Dagenais and Robillard 2008], we reported having analyzed JBoss IDE 1.1 and 1.5: this was an error and it should have read 1.5 and 2.0.

Following the tool author's recommendations,[9] we configured RefactoringCrawler to raise the number of detected refactorings at the expense of a higher number of false positives by lowering several threshold values. Indeed, we did not want to assess the accuracy of the tool, but use it as a baseline to differentiate refactorings from nontrivial changes. In addition to the `jdt.core` and `jdt.ui` plugins, we added the `jdt.ui.tests` and `jdt.ui.tests.refactoring` to the set of plug-ins analyzed by RefactoringCrawler to increase the incoming calls to the `jdt.ui` plug-in, which was required by this approach to increase the odds of detecting refactorings. We combined in one result set the detected refactorings from the following three version pairs: 3.1 to 3.2, 3.1 to 3.3, and 3.2 to 3.3.

We then followed the same procedure as we did for SemDiff: for each broken call in the first version of a client program, we tried to find a refactoring involving the called method. If we found such refactoring and the refactored element was used by the second version of the client program, we considered that the refactoring detection tool succeeded in providing a relevant adaptive change and that this change was a refactoring.

4.2 Comparison with Prior Study

The methodology described in this section was used to *replicate* the study based on code history described in a previous paper [Dagenais and Robillard 2008]. We replicated this study because we have made important changes to SemDiff and Partial Program Analysis that can impact the results. First, we reimplemented both the user interface and the server component of SemDiff to improve the accuracy of the analyses and to make it more usable for software engineers and researchers [Dagenais and Robillard 2009]. Second, we also reimplemented Partial Program Analysis so that it uses the Eclipse compiler as the backend instead of Polyglot [Nystrom et al. 2003], effectively providing support for new Java 5 constructs (e.g., generics, enhanced for loops). Specifically, the study described in this article replicated the previous study with the following differences.

(1) The new version of Partial Program Analysis has more type inference rules and handles fully qualified expressions more accurately. We expect this change to decrease the number of unknown types present in the recommendations.
(2) As explained in Section 3.2, the new version of StructDiff detects when an import statement or a superclass has been modified and conservatively reports as changed any method that could be impacted by these modifications. We expect to detect more call differences than in the previous version and the number of requests for which SemDiff can provide a recommendation should increase.
(3) Although we included deprecated methods in our input set in the original study, a close inspection revealed that all of these methods had associated comments providing a recommendation of the new method to use. In this case SemDiff does not provide any added value (except in the unlikely case where the comments would be wrong), so we excluded these methods from our analysis sample. These methods were the only ones for which RefactoringCrawler produced a relevant recommendation: consequently, the total number of correct recommendations from RefactoringCrawler went from 6 to 0 in Table II.
(4) While replicating the original study, we found three new compilation errors that we previously missed. Because the Java editor we used to review the client code does not highlight all errors (e.g., when a type is not found, subsequent method calls from this type are not highlighted as erroneous), we have to manually identify all erroneous method calls, a process that is error-prone. This change increased our analysis sample.

[9]D. Dig. Personal communication, 25 August 2007.

Table II. Number of Relevant Recommendations by SemDiff and Number of Refactorings Detected by RefactoringCrawler

Client	Compilation Errors	Errors in Scope	Fixes found by SemDiff	Refactorings found by RC
Mylyn	13	8	7	0
JBoss IDE	24	18	18	0
jdt.debug.ui	29	7(11)	7	0
Total	66	33(37)	32	0

4.3 Results

Table II shows the results of our study. For each client program, we list the number of compilation errors related to the JDT framework (Compilation Errors), the number of errors caused by unresolved method calls (Errors in Scope), the number of errors that could be fixed based on the top three SemDiff recommendations (Fixes found by SemDiff) and the the number of errors associated with refactorings as detected by RefactoringCrawler (Refactorings found by RC). The number of compilation errors represents all compilation errors (e.g., import statement referring to an unknown class, unknown parameter type when declaring a method, unknown method call, etc.). For example, in Mylyn, there were 13 errors related to the JDT framework of which eight were caused by unresolved method calls. SemDiff provided relevant recommendations for seven of them while RefactoringCrawler detected no refactoring relevant to these errors.

We consider an error to be within the scope of our approach if the type of the error is in the input domain of SemDiff (and RefactoringCrawler). Because SemDiff takes as input a method and gives as output a list of methods, we only considered unresolved method calls to be within the scope of our approach. Errors such as unknown import statements cannot be provided as input to SemDiff so we did not try to fix them. Even if method recommendations can be indirectly used to fix these kinds of errors, there was not always an objective way to measure the success of our recommendations.

The two numbers in the Errors in Scope column for jdt.debug.ui represent two interpretations of the scope of our approach. Although there were 11 errors that are applicable to our approach, four could not be validated by following our experimental methodology because the client program replaced the missing functionality by its own implementation instead of using methods in the new version of the framework. In this case, even if our approach (or RefactoringCrawler) provided the correct recommendations, we would not be able to assert this fact using the client's history as an oracle. We thus include seven as the number of adaptation problems for which there is an objectively verifiable solution.

The repository analysis took approximately 11.5 hours on a Xeon 3.0 Ghz with 16 Gb of RAM and running Fedora 10 operating system. This analysis needs to be performed only once before a user can make requests in different disconnected sessions. On average, each request took 1 second to complete. Running the three analyses with RefactoringCrawler took 4.2 hours.

Relevant recommendations. SemDiff found relevant recommendations for 97% of the problematic calls in the client programs. For example, in Mylyn, SemDiff suggested two relevant replacements with a confidence value of 1.0 for `TypeInfoFactory.create(...)` which returned an instance of the `TypeInfo` class presented in Section 2. These two replacements were a call to the `JavaSearchTypeNameMatch` constructor and a call to the method `SearchEngine.createTypeNameMatch(...)`, both returning a subclass of `TypeNameMatch`.

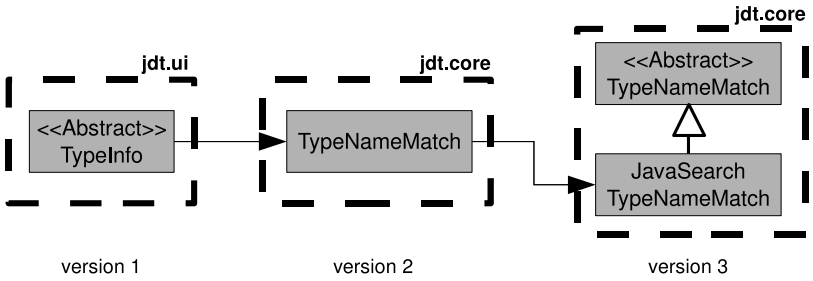

Fig. 11. Evolution of TypeInfo into JavaSeachTypeNameMatch.

Finally, in jdt.debug.ui, SemDiff was not able to provide the correct recommendation for one method in Mylyn. This is a public method, `OpenTypeHistory.contains(TypeInfo)`, that was never called in JDT and that was likely provided to give a complete API (other methods in this class include `add` and `remove`). As it happens, Mylyn also called `OpenTypeHistory.remove(TypeInfo)`, and this time, SemDiff could recommend a relevant replacement: `OpenTypeHistory.remove(TypeNameMatch)`. Changing the parameter type of the `contains` method from `TypeInfo` to `TypeNameMatch` effectively fixed the broken method in Mylyn. This example shows that SemDiff might not be able to recommend replacements for *all* the methods in a class, but recommendations for the other methods might be used by developers to indirectly fix broken methods. Additionally, the detection of this change also indicates that the heuristics introduced to cope with the inevitable inaccuracy of *Partial Program Analysis* succeeded to find a replacement to a method with a common name.

Confidence value. In most cases, the confidence value was necessary to discriminate relevant replacements from false positives because SemDiff produced an average of 13.7 recommendations per request. On average, there were 2.2 recommendations with a confidence value of 1.0 per request. The support of the relevant recommendations had an average of 2.3 methods. Because this low support was enough to distinguish relevant recommendations from bad ones, we consider this to be evidence that, for frameworks like JDT, our approach can work only by analyzing the code of the framework itself and does not require a set of examples using the framework.

Nontrivial changes. RefactoringCrawler found 14,207 refactorings between JDT releases 3.1 and 3.3. Although a subset of these reported refactorings were true positives, none were related to errors in the client programs we studied, which means that the changes that broke the client programs were likely more complex than simple refactorings (e.g., a method was both renamed and moved).[10] This observation provides evidence that our approach works in the face of nontrivial changes. For example, in Mylyn, the suggestion to replace a factory method involving the `TypeInfo` class with methods related to `TypeNameMatch` was far from trivial: as illustrated by Figure 11, this change spanned across the two `jdt` plug-ins and was part of a change chain. A developer would not have been able to locate the relevant replacement by searching for a method with a similar name or by looking at the package where the original method resided. Because this change was not mentioned in the Eclipse 3.3 release notes, it is

[10]RefactoringCrawler found many false positives, that is, refactorings that did not happen, because we configured the tool to detect as many refactorings as possible. We did not count the number of false positives because we only wanted to objectively find if a refactoring caused the compilation errors in the client programs.

likely that finding a relevant method replacement would have required a significant investigation effort.

Another interesting recommendation was provided for the JBoss IDE: SemDiff recommended to replace the constructor of `ListContentProvider` with the constructor of `ArrayContentProvider`. Although the former is located in the `jdt.ui` plug-in, the latter is located in the `org.eclipse.ui` plug-in, which was not even analyzed by SemDiff. This shows that SemDiff can provide recommendations when a framework feature is replaced by an external functionality. Being able to track changes that are outside the analyzed framework could also enable us to recommend adaptive changes related to a framework, but only by analyzing a subset of its client programs. This would make our approach usable even if the framework's source code and source history were not publicly available.

Finally, to detect nontrivial changes, SemDiff used the three heuristics presented in Section 3.1 several times when recommending adaptive changes to the three client programs: change chains were detected in 18% (6 out of 33) of the requests, caller replacements had to be found in 12% (4 out of 33) of the requests and change sets with spurious calls were removed 79% (26 out of 33) of the time.

Summary. SemDiff was able to recommend relevant adaptive changes for 97% (32 out of 33) compilation errors that were caused by framework evolution. The fact that RefactoringCrawler did not detect any refactoring that caused the compilation errors in the client programs indicates that a developer would probably have struggled to find a suitable replacement for most of the broken calls. Arguably, even if a developer had found a replacement, the low cost of SemDiff on the client side (each request took an average of 1 second) makes our approach more effective in most cases. On the server side, SemDiff took less than four seconds to process each change set, which is typically faster than the compilation of the framework or the execution of test suites in a continuous build environment. SemDiff could then be integrated with such environment without significantly affecting the development process Alternatively, the repository analyses could be performed after each minor releases.

4.4 Threats to Validity

The external validity of this study is limited by the fact that we only studied the evolution of a subset of the Eclipse JDT framework and it is probably not representative of the code and evolution patterns of other frameworks. Multiple factors such as change set granularity, method cohesion, and programming idioms vary between software projects and can affect our approach. For example, the two first factors will introduce noise while some programming idioms such as using long call chains (e.g., `m1().m2().m3().m4()`) are likely to decrease the precision of our call difference analysis. Still, the impact of these factors on the results is mitigated by the strategies we devised to prevent them (e.g., confidence value). Moreover, because 21 developers contributed to jdt.core and jdt.ui, we can reasonably assume that the results we obtained were not related to a particular developer profile.

We chose to study only two components of the JDT framework because these components implement most of the features of the framework and account for 84% of the source code in version 3.1 (478 KLOC out of 571 KLOC). Moreover, we wanted to use one component of the JDT framework as a client so we could not include the whole framework in our analysis. It is possible that we would have obtained different results by considering all components of JDT or even all components of Eclipse. Typically, when a method is deleted in one component, the other components depending on that method are adapted in a later version. Because SemDiff computes recommendations by analyzing the adaptations in any version, SemDiff could have captured these

adaptations. It is possible though that many methods could have been adapted at the same time resulting in additional noise. Because finding callers of a removed method, as opposed to noise, was the main problem encountered by SemDiff, we believe that analyzing only two components of JDT was an additional challenge to our approach and not a major threat to the validity of this study.

We evaluated the relevance of our recommendations by analyzing the evolution of client programs. Since we used historical data, we can only speculate on why the methods we recommended were called by the client programs and we cannot assess how the developers would have used our recommendations.

Finally, by using client programs and a refactoring detection tool, we limited the need for personal judgment when assessing the relevance of a recommendation and the complexity of a change. The choice of client programs is still subject to investigator bias, but this risk is mitigated by the fact that the investigators had no control and were not involved in the evolution of the selected client programs.

5. QUALITATIVE STUDY OF CHANGES

We pursued the evaluation of our approach by performing a detailed study of the changes that occurred between two versions of two other frameworks. This study provides an answer to these additional research questions.

(1) What types of changes cause methods to be removed?
(2) How do SemDiff's recommendations differ from other change detection techniques?
(3) What types of changes are typically missed by SemDiff?

To answer these questions, we studied the evolution of two other systems that exhibit different development styles and life cycles. We selected two consecutive major releases of these systems and we computed a list of the public and protected methods that had been removed between the two releases. Then, for each system, we randomly selected 50 methods that we inspected to find the cause of the removal. We also looked at these methods from the perspective of framework users who would need to adapt their client program and we tried to find a suitable replacement. Finally, we compared our findings with the results of SemDiff and two other change detection tools.

Inspecting 50 random method removals per framework, as opposed to the method removals that broke a client program as in the previous study, has several implications. The main advantage of this strategy is that it increases our analysis sample and the chance to study different removal scenarios. For example, in the previous study, we analyzed only 33 method removals because each client program was using a subset of the JDT API and only 10% of the framework's API broke between the two versions of the framework. Conversely, the risk of this strategy is to study the removal of methods whose impact on clients is not established.

5.1 Study Design

We selected two open source systems, jEdit and Nutch. These systems have different development styles and life cycles. jEdit is a mature system with an extensive source history and Nutch is a rapidly growing application that exhibits major changes between releases. Even if these programs can be used as stand-alone applications, they are also used as frameworks to respectively develop text editor plug-ins and Web crawling applications.

For each framework, we selected two consecutive major revisions where more than 50 public and protected methods had been removed. Public and protected methods can be used by client programs and each of these removed methods could be a potential

Table III. Target Program Versions

Framework	Version 1	Version 2	R. Methods
jEdit	4.1	4.2	448
Nutch	0.9	1.0	223

request for SemDiff. Table III shows the versions we studied for each framework, and the number of public and protected methods removed between the two versions.

We randomly selected 50 removed methods from each framework.[11] The sample is small enough so that we can manually assess the cause of the removal for each method, but it is also large enough to offer a wide variety of changes.

For each removed method in our sample, we tried to identify the cause of the change and locate a suitable replacement from the perspective of a framework user. We used the following methodology to analyze each removed method.

(1) Using the HistoryExplorer tool provided by SemDiff,[12] we located all the change sets where the method had been deleted and where calls to the method had been deleted.
(2) For each change set, we inspected the commit comment and compared the source files where the method had been removed.
(3) Using this information, we came up with a change rationale, that is, a small explanation of the change.
(4) By manually analyzing the change sets and by looking at the source code of the two releases of the framework, we tried to identify relevant replacements for the removed methods.
(5) At the end of our analysis of both frameworks, we grouped the change rationales into general categories of changes.

The inspection of the removed methods was a lengthy process that required on average 12 minutes per method. Although simple refactorings (e.g., method renaming) were easy to identify using the Eclipse compare editor, many changes involved numerous classes. Methods for which there was no replacement took the most time because we wanted to be highly confident that there was truly no replacement. HistoryExplorer provided us with the commit comments, the ability to compare two relevant versions of a source file, and the list of transactions where methods and calls were added, removed and modified.

Following this inspection, we analyzed the whole source history of both frameworks using SemDiff. For each removed method, we made a request to SemDiff and compared the recommendations with the list of replacements we manually computed.

Comparing with other change detection techniques. To answer our second research question, we applied two other change detection techniques on both frameworks: RefactoringCrawler [Dig et al. 2006] and Kim et al.'s approach to detect structural changes (referred to as KEA) [Kim et al. 2007]. RefactoringCrawler and KEA are techniques that significantly differ from SemDiff in their strategy to detect changes

[11]The jEdit repository also contains the source code of an external project, BeanShell, and the jEdit developers occasionally dump the source code of a new release of BeanShell in the repository. We excluded the methods removed in BeanShell from our random sample: a study of the BeanShell repository would have been necessary to accurately recommend replacements of BeanShell methods.
[12]www.cs.mcgill.ca/~swevo/semdiff

and both techniques exhibited a high accuracy[13] in their respective evaluation study. Evaluating change detection approaches by systematically analyzing the removal of program elements between two versions of a program is an evaluation strategy that has been successfully used in the past [Kim et al. 2007].

RefactoringCrawler was presented in Section 4.1 and detects refactorings by comparing the structural and textual similarity of program elements. RefactoringCrawler is representative of a family of change detection approaches derived from Origin Analysis [Godfrey and Zou 2005] that rely on these program characteristics to find method replacements. Because a study from the authors of RefactoringCrawler found that 80% of API-breaking changes such as a method removal were caused by refactorings [Dig and Johnson 2006], we expected RefactoringCrawler to detect a significant subset of the changes that caused the method removals.

Kim et al.'s approach is based on the observation that changes can be expressed with general transformation rules applied to the signature of program elements. One of the advantages of this technique is that a single rule can explain the change of many elements. For example, one rule might stipulate that all methods in the Class `ABC` beginning with the token `paint` were renamed to methods beginning with the token `draw`. We provide a more detailed explanation and comparison of various change detection techniques in Section 6.

We executed RefactoringCrawler with two sets of parameters: the default parameters and a set of parameters promoting the detection of many refactorings at the cost of many false positives. These latter parameters were the same as the ones used in our legacy-based evaluation study (Section 4.1). We executed the KEA tool with the parameters found to be optimal in their evaluation study, that is, a seed threshold of 0.7 and an exception threshold of 0.34 [Kim et al. 2007].

Even if we had performed our inspection of method removals before executing these approaches, we carefully reviewed the recommendations of the three approaches to understand why each suggestion had been recommended and to ensure that we did not miss something in our inspection. As it turned out, we revised the conclusion of our inspection for six methods. The settings of this study were similar to the evaluation studies of both KEA and RefactoringCrawler.

Finally, we did not use KEA in our previous study (Section 4) because the tool would have required significant changes to replicate our experimental setup (e.g., analysis of a subset of Eclipse plug-ins instead of the entire Eclipse platform). Additionally, in the first study, we wanted to find if a refactoring caused a compilation error and our goal was not to compare the recommendations of different approaches.

5.2 Results

The results of our study are presented in Table IV for jEdit and Table V for Nutch. The first column gives the name of the recommender. The second column (Match) gives the number of recommendations given by the recommender that matched our inspection. The third column (Match None) gives the number of method removals for which neither the recommender nor our inspection found a replacement. The fourth column (False Positive 1) gives the number of methods for which the recommendations given by the recommender differed from our inspection. The fifth column (False Positive 2) gives the number of method removals for which the recommender recommended a replacement but for which our inspection did not find a replacement. The sixth column (False Negative) gives the number of method removals for which the recommender did

[13]For this section, we consider that accuracy represents the tolerance of an approach toward false positives, that is, recommending a wrong replacement, and false negatives, that is, not recommending a replacement when there is one.

not find a replacement but for which our inspection found one. The seventh column (Incomplete Match) gives the number of recommendations that partially matched our inspection (there was no incomplete match in Nutch).

For example, the first row of Table IV indicates that for 20 method removals in jEdit, SemDiff recommended the same replacement that we found during our inspection. For 14 method removals, SemDiff did not recommend a replacement and we did not find a replacement during our inspection. Overall, 34/50 = 68% of SemDiff's recommendations or absence of recommendations matched our inspection. For one method removal, SemDiff recommended a replacement, but we found during our inspection another replacement (false positive 1). For ten method removals, SemDiff did not find a method replacement, but we found one (false negative). Overall, 11/50 = 22% of SemDiff's recommendations did not match our inspection. Finally, SemDiff recommended five replacements that we judged incomplete. For example, this was the case when SemDiff recommended only one method, but we found that a method had been replaced by many methods.

Because we had two recommendations from RefactoringCrawler for each method removal, one from the default set of parameters and one from the custom set, we merged the results and reported only the best one. For example, if one set of results recommended a relevant replacement and not the other, we reported that RefactoringCrawler had found a relevant replacement. The last row, Combined, reports the results if we combined the three approaches together and only kept the most accurate results of each approach (Match, Match None, or Match Incomplete). This assumes that we could automatically determine the relevant recommendation if the recommenders disagreed, a nontrivial task that is an area for future work.

The results for Nutch report on only 40 method removals. We found in our sample that ten methods were callback methods that Nutch implemented to use an external framework. These methods override the external framework's methods and are never called by Nutch itself. SemDiff cannot produce any recommendations for these methods because they do not have any caller. Moreover, we believe that client programs based on Nutch would not call these methods because they are expected to be executed in a specific control flow, within the external framework Nutch uses. KEA and RefactoringCrawler found a replacement that matched our inspection in seven and four cases respectively and one of the ten callbacks was never replaced. Finally, Table V also reports on methods without any caller in Nutch, but these methods are not part of the external framework used by Nutch and could legitimately be called by client programs. Such methods can be alternative constructors or getters and setters that are provided for API completion, but that are never used in Nutch.

Table IV shows that SemDiff and KEA performed equally well for jEdit with 78% (20+14+5/50) of recommendations that matched completely or partially our inspection. Although the numbers are the same for the two approaches, we show in Section 5.2.2 that both approaches were complementary by providing recommendations for different method removals. RefactoringCrawler did not perform as well with 52% of recommendations matching our inspection. For Nutch, SemDiff performed slightly better (15+13/40=70%) than KEA (65%) and RefactoringCrawler (48%). If we combine Nutch's normal methods with callback methods, KEA performed better (66%) than SemDiff (58%) and Refactoring Crawler (46%). The next sections answer our three research questions.

5.2.1 Change Types. While manually inspecting the method removals in our sample, we investigated the cause of the removal and we came up with a *change rationale*. At the end of our inspection, we grouped the change rationales into change categories. Table VI shows for each change category the number of recommendations from the

Table IV. jEdit: Comparison of Manual Investigation with Recommenders' Results

Inspection Recommender	Match	Match None	False Positive 1	False Positive 2	False Negative	Incomplete Match
SemDiff	20	14	1	0	10	5
KEA	20	14	1	0	10	5
RC	14	12	1	0	23	0
Combined	26	14	0	0	5	5

Table V. Nutch: Comparison of Manual Investigation with Recommenders' Results

Inspection Recommender	Match	Match None	False Positive 1	False Positive 2	False Negative
SemDiff	15	13	0	1	11
KEA	12	14	0	0	14
RC	8	11	0	3	18
Combined	23	14	0	0	3

Table VI. Results by Types of Changes

	jEdit				Nutch			
Type	Total	S	RC	KEA	Total	S	RC	KEA
not replaced	14	14	12	14	7	6	5	7
generics	0	0	0	0	7	7	6	7
refactoring	21	16	13	19	10	6	8	8
import	0	0	0	0	7	7	0	0
complex	13	9	1	6	6	2	0	4
field	2	0	0	0	0	0	0	0
wicked	0	0	0	0	3	0	0	0

three recommender systems that matched our inspection. For example, we see that out of 50 method removals in jEdit, 21 removals were caused by refactorings. SemDiff (S) recommended a replacement that matched our inspection for 16 of these removals, RefactoringCrawler (RC) produced 13 matching recommendations, and KEA produced 19 matching recommendations. The callback methods are not included in the Nutch section, but nine callback method removals were caused by refactorings and one removal was caused by a "wicked" change, a type of change we explain below. The numbers of each row were taken from the columns "Match" and "Match Incomplete" in Tables IV and V. The numbers in the rows "not repl." and "generics" were taken from the column "Match None." This categorization helped us understand which type of changes were typically detected by each approach.

Not Replaced. Methods for which we found no equivalent replacement were classified as being not replaced. For example, in Nutch, a setter method was removed because the field it accessed had been removed. Because we might have missed a non-obvious replacement, we considered with great care the suggestions of the three recommendation systems, when they produced one. As it can be seen in Table VI though, the number of false positives is very low for the two frameworks and in general, the

recommender systems did not produce any recommendations for this type of change type. This also shows that the three recommenders do not produce a lot of noise.

Generics. Some methods signature were changed to add generic types (e.g., from `m1(List)` to `m1(List<String>)`. When only generic types were added and the approaches did not recommend other replacements, we reported this case under the column "None None."

Refactorings. Some methods were removed and replaced as part of the following refactorings: parameter change (including return type), rename method, rename class, rename package, move method, move class. Refactoring were the most frequent change type accounting for 42% and 36% of the changes in jEdit and Nutch respectively.

Function Import. In Nutch, the class `org.apache.nutch.util.mime.MimeType` was removed and replaced by a class from another library, `org.apache.tika.mime.MimeType`. Seven method removals were related to this change and SemDiff pointed to methods in the new imported class. Because RefactoringCrawler and KEA do not recommend code outside of the framework analyzed, they did not recommend any replacements.

Complex. We classified as complex all the changes that involved many program elements or that required a significant adaptation from the client program. For example, in jEdit, the internal class `DistributedSearch.Client` was part of a redesign that resulted in the creation of three classes that partially replaced the original internal class: `DistributedSearchBean`, `DistributedSegmentBean`, and `LuceneSearchBean`. SemDiff recommended to replace a call to the constructor of `DistributedSearch.Client` by methods from the three classes. KEA did not recommend anything for the constructor, but other methods of `DistributedSearch.Client` were removed and KEA correctly recommended replacement methods from `DistributedSearch.Bean`. SemDiff did not find any replacement for these methods because they were not called by Nutch. RefactoringCrawler did not recommend anything for these method removal cases. Another example of complex changes happened in jEdit: a class was removed and replaced by a property file and several calls to `System.getProperty()`. This change was not detected by any of the three recommenders.

Method to Field. In jEdit, two methods were replaced by an access to a public field. Because the three recommenders only recommend methods, they did not recommend any replacement.

Wicked Changes. This series of changes were very difficult to understand and all recommenders failed to locate a replacement. In Nutch 0.9, the classes `Fetcher` and `Fetcher2` were two similar application classes that could be invoked from the command line (i.e., they had a main method) and that were not referenced in the rest of the project. In version 1.0 of Nutch, the class `Fetcher` was removed and the class `Fetcher2` was renamed to `Fetcher` with some minor modifications. To the recommenders, it appeared that `Fetcher2` had not been replaced and that `Fetcher` had never been removed.

By analyzing more frameworks, we would probably discover more types of changes and further refine what we called "complex changes." These seven types of changes still enabled us to evaluate the accuracy of the three change detection approaches at a finer level of granularity than when changes are aggregated at the system level.

5.2.2 Comparison between Approaches. We found SemDiff and KEA to be complementary because they provided recommendations for different method removals.

For example, SemDiff was the only approach able to recommend a replacement for the method imports in Nutch (`MimeType`). It was also the only approach to point to the various relevant replacements when the class `DistributedSearch.Client` was removed in Nutch.

KEA was the most successful approach in finding replacements for the callback methods and for most refactorings. The callback methods and the refactored methods missed by SemDiff all had in common one problem: they did not have a stable caller. Either they did not have any caller or the calling chain was modified so heavily that SemDiff's unstable caller strategy did not work.

SemDiff and KEA also provided different recommendations for the same high-level change. For example, in jEdit, a hierarchy of classes (e.g., `DirectoryMenu`) was replaced by a general class, `EnhancedMenu` and several calls to BeanShell, a dynamic scripting language used to configure the general class based on scripting files. SemDiff recommended to replace the hierarchy of classes by calls to BeanShell while KEA recommended to replace the hierarchy by `EnhancedMenu`. We judged that these recommendations, taken separately, were incomplete. In another case, SemDiff recommended to replace a call to a deleted constructor, `SearchDialog()`, by an existing factory method, `EnhancedDialog.showSearchDialog` while KEA recommended to replace the constructor by a new private constructor of the `SearchDialog` class. We judged that SemDiff's recommendation was the correct one because the private constructor recommended by KEA could not be called by jEdit client programs and the callers of the deleted constructor clearly called the factory method in the new version of jEdit. In both cases, KEA could not have suggested SemDiff's recommendations, because the recommended methods already existed in the previous version of jEdit, and KEA can recommend only new methods.

There was only one method removal for which RefactoringCrawler recommended a valid replacement missed by SemDiff and KEA. The method `DefaultInputHandler.keyPressed()` was renamed to `DefaultInputHandler.handleKey()`: RefactoringCrawler detected this change because the textual representation of the two methods' body was highly similar. The calling chain of the original method changed significantly, so SemDiff could not detect a replacement, and the name similarity of the two methods was not high enough for KEA to recommend a replacement.

For recommendations of highly structured changes like refactorings, we found that KEA and RefactoringCrawler provided a rationale that was more effective than SemDiff's rationale to understand and validate the recommendation. For example, to explain a method moved because of the renaming of a package, KEA would provide a name transformation rule like "all methods from classes whose name contains the token `Menu` changed their package from `org.gjt.sp.jedit.gui` to `org.gjt.sp.jedit.menu` ," RefactoringCrawler would say that the queried method was moved from one class to the other, and SemDiff would show that a caller replaced the removed method by the recommended method. As opposed to SemDiff's rationale, the name transformation rule of KEA and some refactorings of RefactoringCrawler enabled us to clearly identify changes that impacted more than one method (e.g., a class or package renaming).

For complex changes, we found that SemDiff's rationale had the added benefit of giving an example of how to use the recommendation. For example, Figure 12 shows the rationale given by SemDiff when requesting a replacement for the constructor `SearchDialog`. At the bottom of the screenshot, SemDiff indicates on the third line that a call to the `SearchDialog` constructor has been removed in transaction 787 and replaced by a call to `showSearchDialog()`. By double-clicking on this line, SemDiff opens the compare editor which shows two versions of a jEdit source file where the call to `SearchDialog` was replaced, making it obvious for the user that this is a valid replacement. Although the old constructor accepted only two parameters, the recommended

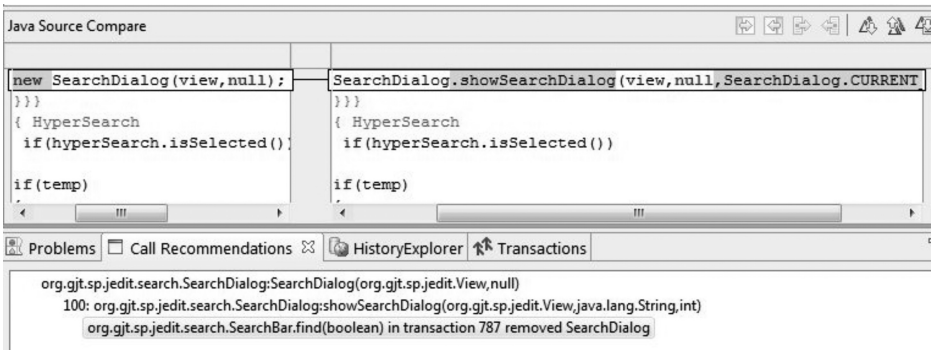

Fig. 12. SemDiff explanation for a recommendation helps developers use the recommended method.

method takes three parameters: the example given by SemDiff provides a default value that should be suitable for the user. We believe that in similar cases, showing an example in a compare editor provides a stronger justification than giving a refactoring name because it allows the user to see the change in its context.

5.2.3 Limitations. As explained in Section 3.1.7, we can analytically determine the two types of limitations of SemDiff: either there is no stable caller for the removed method (recommendation search limitation) or the replaced element is not a method call added by the same caller in the same change set (replacement strategy limitation). Empirical evidence shows that only one of these limitations is important in practice.

SemDiff could not find a relevant recommendation for 30 method removals out of 100 (false negatives); 27 of these false negatives were caused by the recommendation search limitation, that is, the absence of a caller in the framework (e.g., a callback method) or by a significant change in the calling chain. SemDiff could have recommended a relevant recommendation for five of these 27 false negatives by making a query to an alternative method: the overridden method in the parent class or the constructor.

SemDiff could not find a recommendation for the remaining three false negatives because the relevant replacements were either a field or a method call added in a new caller. These two replacement strategies are currently not supported by SemDiff. On the other hand, a quick search in HistoryExplorer for the file versions that removed calls to these methods provided the relevant replacement.

Overall, the success of SemDiff depends on the number of methods without stable callers. in the JDT study, we found that only one such method existed. In jEdit and Nutch, 27 removed methods that we manually analyzed did not have a stable caller, but 12 of these methods would not have been useful for framework users because they were callback methods or part of a stand-alone application class (wicked method removals). The low number of false negatives (27-12=15) and false positives (2) provide evidence that SemDiff is a good approach to recommend relevant replacements for framework users, but it might not be suitable to systematically find a replacement for all methods that were removed between two major releases of a framework.

As demonstrated by the "Combined" rows of Tables IV and V, combining the three approaches could significantly improve the results, but it is clear that human interaction is still necessary because there are changes (10% in jEdit and Nutch) for which none of the approaches could recommend a relevant replacement. Based on our classification of the changes, we also believe that there are changes like the wicked method removals that will always elude general change detection approaches.

5.3 Threats to Validity

The threats to the validity of our study are investigator bias during the inspection of the changes and limited external validity.

A manual inspection of the changes was necessary to find replacements that were not in the scope of current change detection approaches, to classify the types of changes and to evaluate the differences in the recommendations and rationales given by three change detection approaches. Although such inspection is inherently subjective, several factors contributed to the robustness of our inspection. First, the combined results of the three recommenders matched 91% (45+37 out of 90) of our classification and we provided strong evidence to explain why the remaining 9% (5+3 out of 90) could not be fixed by the approaches (e.g., method replaced by a field). Second, the results of the recommenders are reported as matching or not our inspection (as opposed to correct and incorrect results), hence limiting the potential bias of our evaluation. Third, our comparison of the three change detection approaches went beyond an evaluation of their accuracy and we provided examples for each observation we made. Finally, the list of inspected changes with our classification and the results of the three recommenders will be available upon request for independent analysis.

This study considered only two open source systems: these systems are standalone programs that are also used as frameworks, as opposed to pure libraries or frameworks that cannot be executed alone. Except refactorings that roughly represented 40% of the changes in both systems, we found that the types of changes, and the ability of SemDiff to recommend a relevant replacement, varied greatly, especially if we take into consideration the results of the study in Section 4. We did find three trends that we believe would generalize to many systems. First, SemDiff produces very few false positives. Second, most false negatives are caused by the absence of a stable caller, a metric that could be conservatively computed on any given system. Third, recommendations and rationales given by SemDiff differ from other change detection techniques and complement them. As we found in a previous study [Robillard and Dagenais 2008], open source systems vary widely in the way they evolve so we believe that the trends we identified are more important than attempting to generalize the accuracy of SemDiff over entire systems.

6. RELATED WORK

Supporting framework evolution is an active research area and various techniques have been proposed with this goal.

Migration Path. A number of approaches have been proposed to document framework changes and allow client programs to be automatically repaired. For example, transformation rules [Chow and Notkin 1996] can accompany a framework to indicate how calls to functions in the old framework version should be modified in order to work with the new version. CatchUp! [Henkel and Diwan 2005] is a tool integrated in the Eclipse development environment that captures refactorings explicitly applied (i.e., using Eclipse refactoring tools) by developers. The captured refactorings can then be *replayed* in a client program to repair broken method calls. Projects such as Eclipse also developed rules and guidelines to evolve APIs [des Rivières 2007]. For example, these guidelines explain how to preserve contracts (e.g., postconditions) and binary compatibility between old and new APIs. Finally, new versions of development environments provide enhanced support for API evolution. For example, Eclipse 3.3 introduced explicit support for saving and replaying refactorings in the form of Refactoring Scripts.

As opposed to transformation rules, our approach is fully automated and does not require the framework authors to explicitly describe how the client program

should adapt to the framework. Our approach also captures more changes (e.g., non-refactorings, imported functionality) than refactoring capturing and replaying techniques as it is not limited to explicitly applied refactorings. Moreover, authors of the client program do not need to search through a list of refactorings to find the one that is relevant to a broken method.

Change Detection. Most change detection techniques refer to Origin Analysis [Godfrey and Zou 2005] as the basis of their approach. Origin Analysis is a semi-automated technique that aims at tracing back the source of a program element in a previous version of the program to detect changes such as splitting and merging. Similarly, if a change detection technique finds that a method `e` in version `n-1` is the origin of method `f` in version `n`, it will conclude that method `e` was refactored to method `f`.

To identify the origin of a program element, change detection techniques use a variety of strategies based on program element characteristics (e.g., name, callers) to assess the similarity of two elements across two versions. If the similarity of the program elements is beyond a certain threshold, these techniques conclude that one is an updated variant of the other.

More precisely, techniques such as UMLDiff [Xing and Stroulia 2006] analyze code structure similarity (e.g., calls, hierarchy, accesses, etc.) between complete versions of a program to detect high level changes such as refactorings. RefactoringCrawler [Dig et al. 2006] is a similar but more lightweight alternative to this approach that also analyzes the code structure similarity and uses Shingle, a custom textual similarity analysis, on two complete versions of a program to detect refactorings. A later implementation of Origin Analysis [Kim et al. 2005] fully automated the approach for Java, but also reduced the number of change types that the technique detected to only consider program elements renaming and move. Mining software repositories [Zimmermann et al. 2005] is another technique that can be used to track changes: by analyzing a program's evolution and detecting code clone patterns (textual similarity), researchers were able to detect refactorings [Weissgerber and Diehl 2006]. Other researchers also used repository mining to predict the likelihood of a class to be refactored in the next two months [Ratzinger et al. 2007] or to classify fine-grained changes that occurred inside method bodies [Fluri et al. 2007]. Kim et al. [2007] proposed a technique for detecting change patterns by leveraging the similarity of program element names. This approach is used by LSdiff, a tool that represents systematic structural differences between two versions of a program as logic rules [Kim and Notkin 2009]. LSdiff can also be used to detect potential inconsistencies in groups of related changes.

Schäfer et al. [2008] devised, in parallel with our work on SemDiff, an approach that uses associative data mining on framework usage changes (such as method call changes) and that can recommend more change patterns than SemDiff (e.g., a field access should be replaced by a method call.) As opposed to SemDiff, this approach only compares two full versions of a program instead of analyzing change sets. The coarser granularity of the analyzed units of changes introduces more false positives in the results so Schäfer et al. must use filtering heuristics or look for specific change patterns to reduce the noise in the results. Additionally, this approach requires stable callers, that is, methods that use the framework but that keep the same signature between two versions of the framework. In the presence of unstable callers, Schäfer et al.'s approach will not be able to recommend a replacement. The absence of stable callers is not a significant problem for SemDiff because changes are analyzed at the change set level so, in practice, only a few methods change at the same time. In the case when a caller and a callee are changed together, SemDiff can use the unstable caller strategy described in Section 3.1.3. Schäfer et al. evaluated their approach on three frameworks and they also used RefactoringCrawler to determine when a change had been caused by a

Table VII. Comparison of Change Detection Approaches

Method	Program Element Characteristics	Versions
Origin Analysis	name and structural similarity	two complete versions selected manually
UMLDiff	name and structural similarity	full versions between two versions
S. Kim et al.	name, structural, and textual similarity	two complete versions selected manually
RefactoringCrawler	structural and textual similarity	two complete versions selected manually
Weissgerber et al.	code clone differences	all change sets between two versions
M. Kim et al.	name similarity	two complete versions selected manually
Schäfer et al.	usage difference	two complete versions selected manually
SemDiff	call difference	all change sets between any versions

refactoring. Although the precision of the results was high (average of 86.7%), RefactoringCrawler detected 35 refactorings out of 255 changes (14%) that were missed by Schäfer et al.'s approach. Because the frameworks that we used to evaluate SemDiff were different and because Schäfer et al.'s tool is not publicly available, we cannot directly compare our results with theirs. We found that in the three frameworks we analyzed, SemDiff had a precision of 98% and RefactoringCrawler only detected five refactorings out of 133 changes (4%) that SemDiff missed. In summary, Schäfer et al.'s approach and SemDiff are similar in their strategy of using usage changes to recommend replacements, but we believe that analyzing change sets as opposed to full versions explains why SemDiff seems to produce more accurate results.

Table VII shows a comparison between the program element characteristics and the program versions used by change detection approaches. The approaches are presented in the chronological order in which a conference or journal publication first mentioned the approach. The table only presents a high level view of the approaches and the details vary significantly. For example, although both UMLDiff and M. Kim et al. approaches compute the name similarity of two program elements, UMLDiff computes only the Longest Common Subsequence (LCS) while M. Kim et al. computes complex transformation rules. Starting from origin analysis, Table VII shows that the trend in change detection techniques has been to decrease the number of program element characteristics used to link two elements, but empirical studies performed on these approaches showed that the accuracy increased. Still, techniques based on many program element characteristics can give an additional degree of confidence about the origin of an element: if two elements in two versions of a program have a similar name, a similar structure and a similar textual representation, it is highly probably that they are the same element. This confidence is required by activities demanding detailed traceability information, such as bug tracking.

Table VIII shows a comparison of the rationales given by the various approaches and their limitations. The rationale summarizes how a user can understand the result of the various approaches if they determine that a method `m1` in version `n-1` is the same as method `m2` in version `n`. Although some of the change detection approaches presented in the tables can also detect changes in classes and packages, the rationale only focuses on method changes. As shown in the table, rationales are often based on well-defined

Table VIII. Rationale and Limitations of Change Detection Approaches

Method	Rationale	Limitations
Origin Analysis	m1 has a name and a structure similar to m2. This similarity corresponds to a well-defined splitting or merging pattern.	threshold, deletion based, codebase, function only
UMLDiff	m1 has a name and a structure similar to m2.	threshold, deletion based, codebase
S. Kim et al.	m1 has a name, a structure and a textual representation similar to m2.	threshold, deletion based, codebase, one-to-one
RefactoringCrawler	m1 has a structure and a textual representation similar to m2. This similarity corresponds to a well-defined refactoring.	threshold, codebase, one-to-one
Weissgerber et al.	m1 has a textual representation similar to m2. This similarity corresponds to a well-defined refactoring. This change occurred in a change set.	threshold, change set based, deletion based, codebase, one-to-one
M. Kim et al.	m1 has a signature similar to m2. There is a name transformation rule that explains the change in the two signatures.	threshold, deletion based, codebase, method only
Schäfer et al.	methods that used to call m1 are now calling m2.	locality, stable callers
SemDiff	methods that used to call m1 are now calling m2. This change occurred in a change set.	change set based, locality, stable callers, method only

change patterns (e.g., rename refactoring) or temporality (e.g., the change happened in a small change unit). The limitations column indicates the main factors that limit the accuracy (false positives and false negatives) of the approaches in practice.

(1) *Threshold.* The approach relies on a user defined threshold to determine if a similarity metric holds between two elements, typically missing large changes that disfigure the elements (the elements are the same, but they do not look alike).
(2) *Deletion based.* The approach only links elements that were deleted in one version and added in another version, ignoring elements that are replaced by existing elements or deprecated, but not deleted, elements.
(3) *Codebase.* The approach only considers elements that are in the same code-base, ignoring newly imported functions.
(4) *Function/Method only.* The approach only considers methods or functions, but not classes, packages or fields.
(5) *one-to-one.* The approach only detects one-to-one changes and cannot be used to detect mergings and splittings.
(6) *Change set based.* The approach only considers elements changed in the same change set, ignoring replacements in other change sets.

(7) *Locality.* The approach only considers *usages* that are replaced locally (e.g., in the same caller or in the same class), ignoring usage that is removed in one element and replaced in another element.
(8) *Stable Callers.* The approach only finds a replacement if the users (i.e., callers) of the original program element did not change.

The table does not include limitations that do not significantly impact an approach. For example, SemDiff uses thresholds to improve the performance of two strategies, but since these strategies are rarely used, they generally do not affect SemDiff's results. Although M. Kim et al.'s approach does not explicitly find methods that were split, the approach can generate multiple name transformation rules showing that one method is related to many methods. All of these techniques provide some workarounds or heuristics to reduce the negative effect of these limitations. For example, through empirical studies, most approaches limited by thresholds suggested default values that yielded the best false positive and false negative ratio [Kim et al. 2007]. Another example is that SemDiff uses a strategy to find a replacement for unstable callers.

Typically, techniques based on Origin Analysis suffer from three main limitations: they are based on thresholds, they only detect changes between deleted and added elements, and they can only recommend replacements in the codebase they analyzed. As shown with the study presented in Sections 4 and 5, our approach does not suffer from these three limitations because we do not try to assess the similarity of methods that changed: we only analyze what happens to methods that were *referring to* the changed methods. This leads to another family of limitations: locality and stable callers.

Framework Usage. Another family of approaches could potentially be used to support framework evolution. Strathcona [Holmes et al. 2006] and FrUIT [Bruch et al. 2006] are systems that mine a set of framework usage examples and recommend program elements of potential interest for framework users based on the local programming context. For example, if a developer is using class `C` and calling methods `m1` and `m2` from a certain framework, framework usage tools will typically recommend other program elements that are used along these classes and methods in the mined examples.

We could potentially use framework usage tools to recommend adaptive changes by mining usage examples of the new framework version and running the tools on each method in the client program that has a broken method call. The recommendations would probably contain the correct replacements. One of the issues with these approaches is that the data mining techniques they use usually need a fair number of usage examples in order to produce accurate results: unfortunately, it takes some time before an adequate number of usage examples become available when a new framework version is released. In contrast, SemDiff uses only the usage examples inside the framework itself to produce results.

7. CONCLUSION

We presented a technique to recommend adaptive changes for clients of framework code that has evolved in a way that is not backward-compatible. Our approach involves analyzing how the framework adapts to its own changes, and recommending similar adaptations. Specifically, our technique extracts the differences in the outgoing calls in each change set and recommends a set of method replacements accompanied by a confidence value. A historical study of the Eclipse JDT framework and three of its client programs showed that our technique can detect nontrivial changes, and that it succeeded in providing correct recommendations for 97% (32 out of 33) of the cases of missing functionality between Eclipse release 3.1 and 3.3. A qualitative study on two other frameworks helped us refine our understanding of the types of changes detected

and undetected by SemDiff and the difference between SemDiff and two other change detection approaches. For example, as opposed to previous work on change detection techniques, our approach can recommend methods located outside of the framework when a functionality has been imported from external libraries. SemDiff does not suffer from the limitations of previous change detection techniques, but it has its own type of weaknesses. We conclude that analyzing outgoing call differences is an efficient technique to track a framework's evolution and repair dependent client programs.

ACKNOWLEDGMENTS

The authors are grateful to Danny Dig for his suggestions on using RefactoringCrawler and to Miryung Kim for making her tool available. We also thank Ekwa Duala-Ekoko for his valuable comments on the article. This project was supported by NSERC.

REFERENCES

BOULANGER, J.-S. AND ROBILLARD, M. P. 2006. Managing concern interfaces. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. 14–23.

BRUCH, M., SCHÄFER, T., AND MEZINI, M. 2006. FrUiT: IDE support for framework understanding. In *Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange*. 55–59.

CHOW, K. AND NOTKIN, D. 1996. Semi-automatic update of applications in response to library changes. In *Proceedings of the International Conference on Software Maintenance*. 359–369.

DAGENAIS, B. AND HENDREN, L. 2008. Enabling static analysis for partial java programs. In *Proceedings of the Conference on Object Oriented Programming Systems and Applications*. 313–328.

DAGENAIS, B. AND ROBILLARD, M. P. 2008. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th International Conference on Software Engineering*. 481–490.

DAGENAIS, B. AND ROBILLARD, M. P. 2009. SemDiff: Analysis and recommendation support for api evolution. In *Proceedings of the 31st International Conference on Software Engineering* (Formal Demonstration). 599–602.

DANIEL, B., DIG, D., GARCIA, K., AND MARINOV, D. 2007. Automated testing of refactoring engines. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*. 185–194.

DES RIVIÈRES, J. 2007. Evolving Java-based APIs. `http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs`. (accessed 7/09).

DIG, D. AND JOHNSON, R. 2006. How do APIs evolve? A story of refactoring. *J. Softw. Maint. Evol.: Resear. Prac. 18,* 2, 83–107.

DIG, D., COMERTOGLU, C., MARINOV, D., AND JOHNSON, R. 2006. Automated detection of refactorings in evolving components. In *Proceedings of the European Conference on Object-Oriented Programming*. 404–428.

DIG, D., MANZOOR, K., JOHNSON, R., AND NGUYEN, T. N. 2007. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of the 29th International Conference on Software Engineering*. 427–436.

FLURI, B., WÜRSCH, M., PINZGER, M., AND GALL, H. C. 2007. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Engin. 33,* 11, 725–743.

FOWLER, M. 2002. Public versus published interfaces. *IEEE Softw. 19,* 2, 18–19.

GODFREY, M. W. AND ZOU, L. 2005. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Engin. 31,* 2, 166–181.

HENKEL, J. AND DIWAN, A. 2005. Catchup!: Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering*. 274–283.

HOLMES, R., WALKER, R. J., AND MURPHY, G. C. 2006. Approximate structural context matching: An approach for recommending relevant examples. *IEEE Trans. Softw. Engin. 32,* 12, 952–970.

JAGANNATH, V., LEE, Y. Y., DANIEL, B., AND MARINOV, D. 2009. Reducing the costs of bounded-exhaustive testing. In *Fundamental Approaches to Software Engineering*. 171–185.

KERSTEN, M. AND MURPHY, G. C. 2006. Using task context to improve programmer productivity. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering*. 1–11.

KIM, M. AND NOTKIN, D. 2009. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*. 309–319.

KIM, M., NOTKIN, D., AND GROSSMAN, D. 2007. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th International Conference on Software Engineering*. 333–343.

KIM, S., PAN, K., AND E. JAMES WHITEHEAD, J. 2005. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*. 143–152.

LARMAN, C. 2001. Protected variation: The importance of being closed. *IEEE Softw.*, 98–90.

LIENTZ, B. P. AND SWANSON, E. B. 1980. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

NYSTROM, N., CLARKSON, M. R., AND MYERS, A. C. 2003. Polyglot: An extensible compiler framework for java. In *Proceedings of the 12th International Conference on Compiler Construction*. 138–152.

RATZINGER, J., SIGMUND, T., VORBURGER, P., AND GALL, H. C. 2007. Mining software evolution to predict refactoring. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*.

ROBILLARD, M. P. AND DAGENAIS, B. 2008. Retrieving task-related clusters from change history. In *Proceedings of the 15th Working Conference on Reverse Engineering*. 17–26.

SCHÄFER, T., JONAS, J., AND MEZINI, M. 2008. Mining framework usage changes from instantiation code. In *Proceedings of the 30th International Conference on Software Engineering*.

WEISSGERBER, P. AND DIEHL, S. 2006. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering*. 231–240.

XING, Z. AND STROULIA, E. 2006. Understanding the evolution and co-evolution of classes in object-oriented systems. *Int. J. Softw. Engin. Knowl. Engin. 16,* 1, 23–51.

ZELKOWITZ, M. V. AND WALLACE, D. R. 1998. Experimental models for validating technology. *IEEE Comput. 31,* 5, 23–31.

ZIMMERMANN, T. AND WEISSGERBER, P. 2004. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the International Workshop on Mining Software Repositories*.

ZIMMERMANN, T., ZELLER, A., WEISSGERBER, P., AND DIEHL, S. 2005. Mining version histories to guide software changes. *IEEE Trans. Softw. Engin. 31,* 6, 429–445.

Received July 2009; revised October 2009; accepted November 2009