Recommending Adaptive Changes for Framework Evolution

Barthelemy Dagenais and Martin P. Robillard

School of Computer Science
McGill University

Presentation by:
Hemayet Ahmed Chowdhury

Introduction

Frameworks can provide large scale reuse of tasks for developers.

- However, as the frameworks evolve, changes ranging from a simple refactoring to a complete rearchitecture can break client programs.
- For example, removal of methods from the framework and poor documentation can lead to developers putting in a lot of effort and time to figure out which methods replaced which.
- Simple Refactoring tools that just try to detect deleted methods don't really help in slightly more complex or non-trivial situations.
- SemDiff proposes a technique to automatically recommend adaptive changes in the face of non-trivial framework evolution.

SemDiff Architecture

SemDiff consists of two parts : the recommender and a server component

- Developers send request to the server for a method call that does not exist in the new version of the framework anymore
- The server then retrieves data of the framework's version history from it's source repositories
- It then uses a Call Difference mechanism to formulate a list of methods with same functionality
- Call Difference Hypothesis: calls to deleted methods will be replaced in the same change set by one or more calls to methods that provide a similar functionality
- Recommender uses confidence metrics to rank its suggestions.

Adaptive Change Recommendation Data Flow

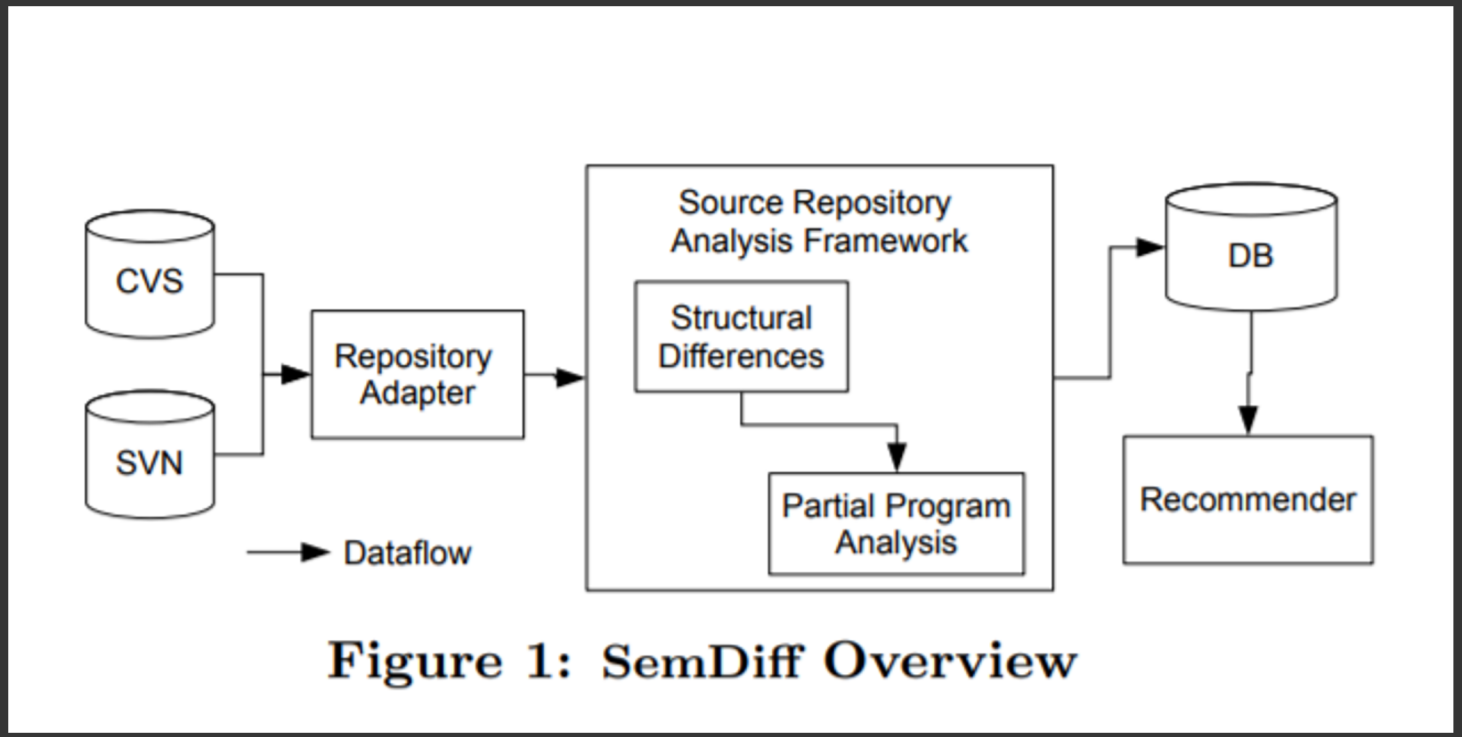

Figure 1: SemDiff Overview

Confidence Metrics

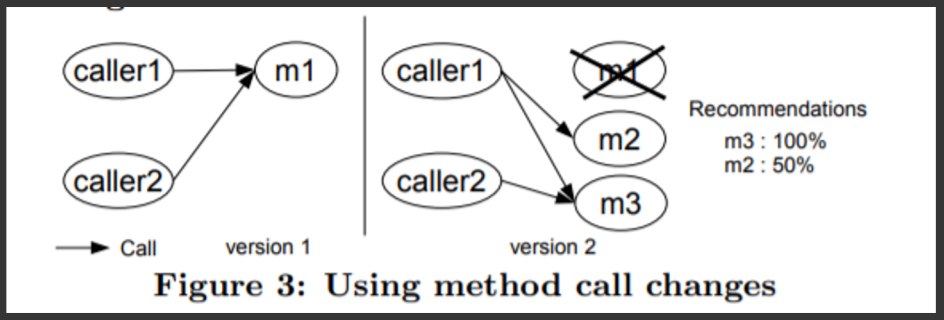

Figure 3: Using method call changes

Support of m2 : 1 (since it replaces m1 once)
Support of m3 : 2 (replaces m1 on 2 occassions)

Confidence of m2 = ½ = 50%
Confidence of m3 = 2/2 = 100%

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

Change Chains and Caller Unstability

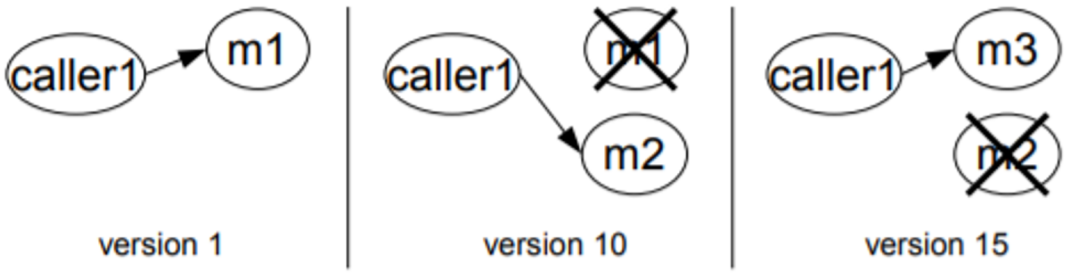

Figure 5: Changes chain

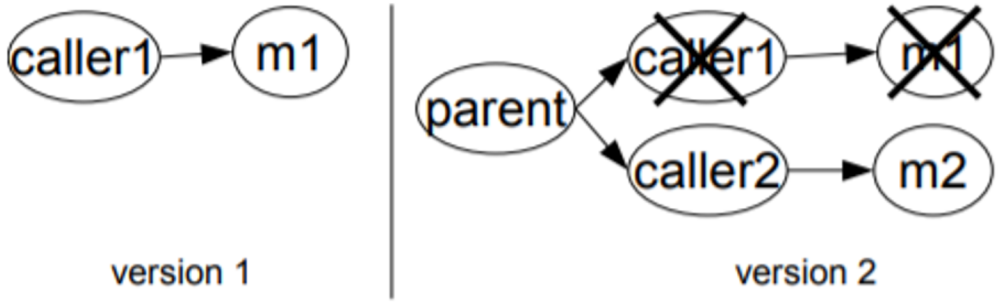

Figure 6: Callee and caller are deleted together

Limitations of the current approach

- Doens't handle removal of root methods (methods that are not called in the framework, but reside in separate libraries)
- Does not group recommendations. Recommends replaced methods separately, even though they may come in conjunction.
- Does not take program semantics/context into consideration. Recommendations cannot guarantee all features will run perfectly.

Retrieving from source repositories

SemDiff provides adapters to retrieve information from

- CVS repositories
- SVN repositories
- Can handle merging of branches.

Change Analysis

SemDiff performs two analyses on every change set in the framework's version history.

- StructDiff : provides a list of all methods that were added, removed and modified
- CallDiff : finds the calls that were added or removed between two versions of each method identified by StructDiff.

Partial Program Analysis

- Since SemDiff only works on a subset of the source code (the change sets), it has to employ partial program analysis.
- Partial program analysis comes with it's own problems since the parsers don't have enough information about all the classes.

Partial Program Analysis - Inferring Types

```
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: }
```

Figure 8: Partial Program Analysis Example

Return type of method 2 in line 9 is inferred as Z, even though it may return an unknown subtype of Z

Partial Program Analysis - Polymorphism

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

Seeking replacement for common methods such as ‘add’ can result in SemDiff giving out false positives, since many other methods may have their own add methods removed which have nothing to do with this specific one.

Evaluation Methodology

- 3 Client programs were used against versions of 1 framework
- 2 versions of each client program selected (c1,c2)
- c1 uses an old framework (f1), c2 uses a new one (f2)
- Compile c1 on f2
- For each method that fails to compile, use SemDiff
- Compare with C2 code to see if the recommendations are implemented.
- A typical refactoring tool (Refactoring Crawler) was also used for comparison.

Evaluation Results

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

Table 2: Number of relevant recommendations by SemDiff and RefactoringCrawler

Threats To Validity

- External : authors studied the evolution of the Eclipse JDT framework and it might not be representative of the code and evolution patterns of other frameworks.
- Authors also didn't analyse how real life developers would handle their recommendations.
- The choice of client programs is subject to investigator bias

Related Work

- CatchUp : a tool that captures refactorings by developers and replays them for a client program
- UMLDiff analyzes code relationship similarity (e.g., calls, hierarchy, accesses, etc.) between complete versions of a program to detect high level changes such as refactorings.
- Strathcona and FrUIT 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.

Thank you