RECOMMENDING ADAPTIVE CHANGES FOR FRAMEWORK EVOLUTION

by
Barthélémy Dagenais

School of Computer Science
McGill University, Montréal

August 2008

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of
Master of Science

Abstract

Application frameworks provide a general design that connects together the different parts of a program and that free developers from low-value activities. In the course of a framework's evolution, changes ranging from a simple function renaming to major modifications of the design can break client programs that depend on the framework. 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 a large open-source framework and three client programs, our approach recommended relevant adaptive changes with a high level of precision, and detected non-trivial changes typically undiscovered by current changes detection techniques.

Résumé

Les cadres d'applications fournissent un design général qui relie les différentes parties d'un programme et qui libère les développeurs des tâches à faible valeur ajoutée. Durant l'évolution d'un cadre d'applications, des changements allant d'un simple renommage d'une fonction à des modifications majeures du design peuvent briser les programmes clients qui dépendent du cadre d'applications. Trouver un remplacement adéquat pour les éléments du cadre d'applications qui étaient accédés par le programme client et qui ont été supprimés durant l'évolution du cadre d'applications peut s'avérer être une tâche difficile. Nous présentons un système de recommandations, SemDiff, qui suggère des adaptations aux programmes clients en analysant comment un cadre d'applications a été adapté à ses propres modifications. Dans une étude sur l'évolution d'un grand cadre d'applications en code source libre et de trois programmes clients, notre système a recommendé des adaptations pertinentes avec un haut degré de précision. Notre système a aussi détecté des changements non triviaux qui sont typiquement ignorés par les techniques de détection de changements courantes.

Acknowledgments

This Master's project would not have been possible without the continuous and generous help of my supervisor, Martin Robillard. His rigorous approach to research was a constant source of inspiration and motivation. I would also like to thank Prof. Laurie Hendren for providing me with helpful insights and comments on this project. Finally, I thank my significant other, Geneviève, for her never failing encouragements and support. My master studies were financially supported by a NSERC Canada Graduate Scholarship and by a McGill School of Computer Science top-up award.

Barthélémy Dagenais
McGill University
June 2008

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Application frameworks support large-scale reuse by providing a general design that connects together the different parts of a program. Usually, frameworks also offer code libraries that perform common tasks and that free developers from low-value programming activities. By developing *client programs* 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 function renaming (i.e., a refactoring) to major design modifications can break client programs: framework functions that the client used to call could be replaced or deleted in the new version of the framework.

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 [25, 31], providing migration paths [19], or deprecating existing methods and indicating new replacements. Unfortunately, current tools cannot capture changes more complex than refactorings. Refactorings are code changes that preserve the application behavior (e.g., changing the name of a class) and most refactorings have the property of preserving the various characteristics of the code being changed (e.g., its structure). When multiple refactorings are applied together or when major changes modifying multiple characteristics of the code happen, current tools fail to automatically recognize the change. An alternative solution would be to manually document a framework's evolution by creating a report that maps the old functions with the new functions, but this solution is not always cost-effective, especially for fast-evolving frameworks. Although framework users are encouraged to only use public Application Programming Interfaces (API) as they usually provide a long-term contract of stability, 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 public interfaces [17].

A recent study of API evolution found that more than 80% of API-breaking changes (e.g., when a function provided by an API is removed) were caused by refactorings, and concluded that techniques aiming at documenting or detecting refactorings were desirable [24]. The authors of the 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" [24, p.105]. To detect the largest portion of API-breaking changes, i.e., refactorings, several approaches have been proposed [23, 29, 34, 35, 48, 49].

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 non-trivial changes (e.g., a composition of simple refactorings).

To help developers repair client programs that are affected by the non-trivial evolution of a framework, we propose an approach to recommend adaptive changes [21]. These changes are a form of maintenance aiming at adjusting a software to comply with its technological environment [38]. 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. Because our approach only analyzes the source code that changed during the evolution of a framework, as opposed to analyzing the complete framework, the server component uses a technique we devised to enable the use of static analysis on partial programs. The SemDiff 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.

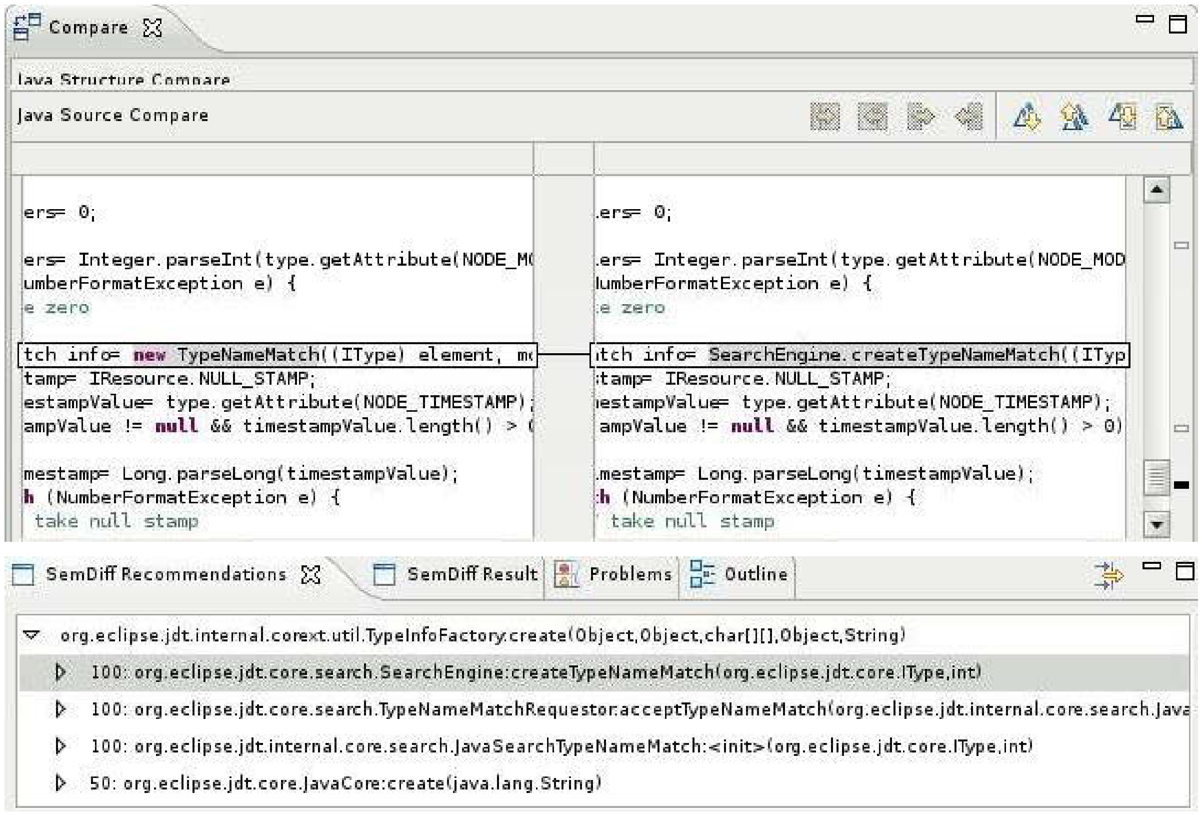
We evaluated the effectiveness and the need for our approach by using SemDiff in an historical study of one large framework, Eclipse [3], in which we recommended adaptive changes for three client programs broken as the result of API-changes. Eclipse is a large application framework mainly employed to create and extend integrated development environments, the Java Development Tools (JDT) being the most commonly used. We then compared our results with the recommendations of a research tool [23] that detects refactorings and that is representative of current changes detection techniques. Our study showed that our approach provided a relevant functionality replacement for 89% of the broken methods, detected non-trivial 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.

The contributions of this thesis include (1) an approach to automatically recommend adaptive changes in the face of non-trivial framework evolution, (2) a technique to enable the static analysis of partial Java programs, (3) the architecture of a complete system to track a framework's evolution and infer non-trivial changes, and (4) an historical study on the redesign of a framework component that occurred in Eclipse, providing evidence for the effectiveness for our system.

1.1 A sample scenario

Let us consider the case of a developer who decides to reuse internal classes of the Eclipse framework in a client program. 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 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. When Eclipse 3.3 is released, the developer loads the client project in the development environment, which automatically tries to build the developer's program against the new version of the framework. At this point, the developer discovers that the compiler generates multiple compilation errors, because the `TypeInfo` class and its methods are no longer accessible in Eclipse 3.3. The developer then starts exploring 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 that no similarly-named classes provide the required functionality, the developer moves on to see if the missing class is in the same package but under a different name. Again, no classes are found with the same functionality.



```
ers= 0;

ers= Integer.parseInt(type.getAttribute(NODE_MO
umberFormatException e) {
e zero

tch info= new TypeNameMatch((IType) element, mo
tamp= IResource.NULL_STAMP;
estampValue= type.getAttribute(NODE_TIMESTAMP);
ampValue != null && timestampValue.length() > 0

mestamp= Long.parseLong(timestampValue);
h (NumberFormatException e) {
 take null stamp
```

```
.ers= 0;

.ers= Integer.parseInt(type.getAttribute(NODE_MOD
lumberFormatException e) {
e zero

itch info= SearchEngine.createTypeNameMatch((ITyp
tamp= IResource.NULL_STAMP;
iestampValue= type.getAttribute(NODE_TIMESTAMP);
ampValue != null && timestampValue.length() > 0)

.mestamp= Long.parseLong(timestampValue);
:h (NumberFormatException e) {
' take null stamp
```

```
▽ org.eclipse.jdt.internal.corext.util.TypeInfoFactory.create(Object,Object,char[][],Object,String)
    ▷ 100: org.eclipse.jdt.core.search.SearchEngine:createTypeNameMatch(org.eclipse.jdt.core.IType,int)
    ▷ 100: org.eclipse.jdt.core.search.TypeNameMatchRequestor.acceptTypeNameMatch(org.eclipse.jdt.internal.core.search.Java
    ▷ 100: org.eclipse.jdt.internal.core.search.JavaSearchTypeNameMatch:<init>(org.eclipse.jdt.core.IType,int)
    ▷ 50: org.eclipse.jdt.core.JavaCore:create(java.lang.String)
```

Figure 1.1: 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.

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`, as `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 the changes involving the `TypeInfo` class were more than cosmetic.

As illustrated by Figure 1.1, our SemDiff approach can make the process of adapting a client program to an evolving framework more efficient by (1) providing adaptive change recommendations (bottom frame), and (2) providing a source code example that illustrates how the framework was adapted to its own changes (top frame). As opposed to current approaches that display a list of refactorings [23, 35, 48] or change patterns [34] 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.

1.2 Overview

In the remaining chapters, we describe the principles and implementation details underlying our approach (Chapter 2) and present a study on the evolution of the Eclipse Java Development Tool (JDT) framework (Chapter 3). We then provide the details of a technique we devised to statically analyze partial programs (Chapter 4), the types of programs that are analyzed by SemDiff, and present the results of the empirical evaluation we performed to assess the precision of our technique (Chapter 5). We conclude with an overview of the related work (Chapter 6) and final observations (Chapter 7).

Chapter 2
The SemDiff Approach and Tool

Figure 2.1 provides an overview of the SemDiff implementation. SemDiff consists of a set of Eclipse plug-ins forming a 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 one of the analyses performed by the server.

2.1 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 1.1), and queries the recommender for potential replacements. The recommender then formulates

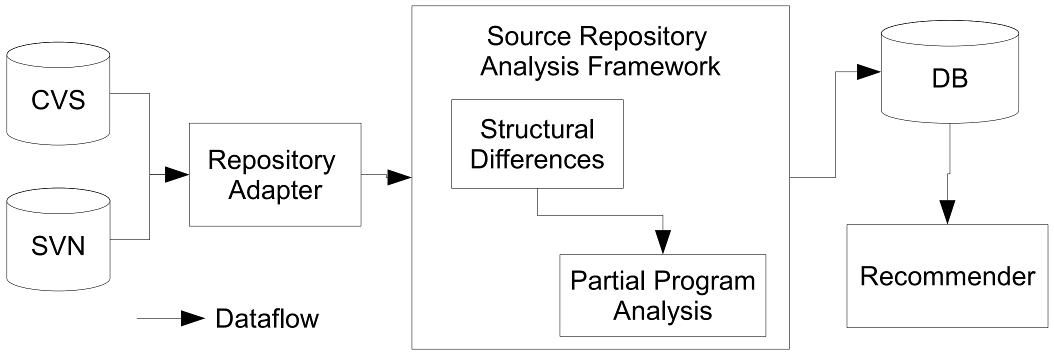


Figure 2.1: SemDiff Overview

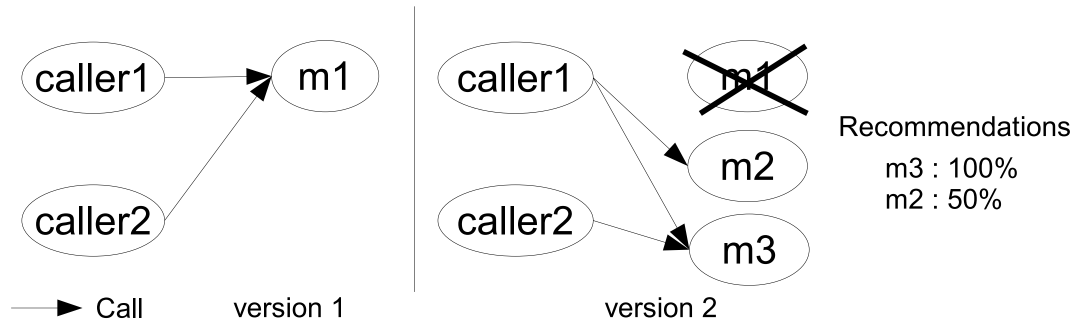


Figure 2.2: Using method call changes

recommendations by using an analysis of the high level changes inferred by the source repository analysis framework.

2.1.1 Using Call Differences

We base our recommendation strategy on the hypothesis 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. Thus, by using differences in outgoing calls for a given method during a framework's history, we can see how a framework was adapted to its own changes. For example, in Figure 2.2, 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 2.2). Then, we gather all calls that were added in these methods as part of the 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 an implicit target version v, we define the confidence value of a method n that replaces a call to method m with the following equations:

$$\textbf{Rem}(m) := \{x \mid x \text{ is a method that removed a call to } m\}$$

$$\textbf{Add}(m) := \{x \mid x \text{ is a method that added a call to } m\}$$

$$\textbf{Callees}(m) := \{x \mid m \text{ calls } x\}$$

$$\textbf{Callers}(m) := \{x \mid x \text{ calls } m\}$$

$$\textbf{Potential}(m) := \bigcup_{x \in Rem(m)} Callees(x)$$

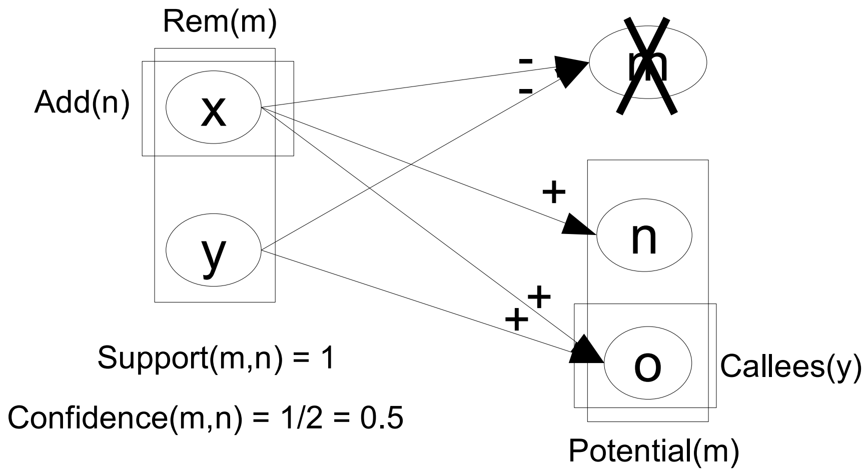


Figure 2.3: Support and confidence of a replacement

$$\mathbf{Support}(m,n) := |Rem(m) \cap Add(n)|$$

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

Figure 2.3 shows an example of each of the above 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 relevance of the recommendations. For example, if o is the recommendation with the highest support, 2, and recommendation n has half of this support, 1, n will have a confidence of 50% (0.5). Because the support is bounded by the number of callers of the removed method ($|Rem(m)| \leq |Callers(m)|$), we hypothesize that the framework will have a sufficient amount of calls to its own methods so relevant recommendations can be discriminated from irrelevant ones. Section 3.2 shows how this hypothesis held in practice.

When looking for a replacement for a method m, we search for the methods that removed a call to m and not all callers of m. Moreover, we do not care if method m still exists. This enables us to get recommendations (1) for methods that are deprecated but still accessed by a subset of the framework or (2) for methods that are replaced before they are actually deleted. This is one difference with previous approaches that use the addition and deletion of methods as the basis for detecting changes and refactoring [34, 35, 48, 49].

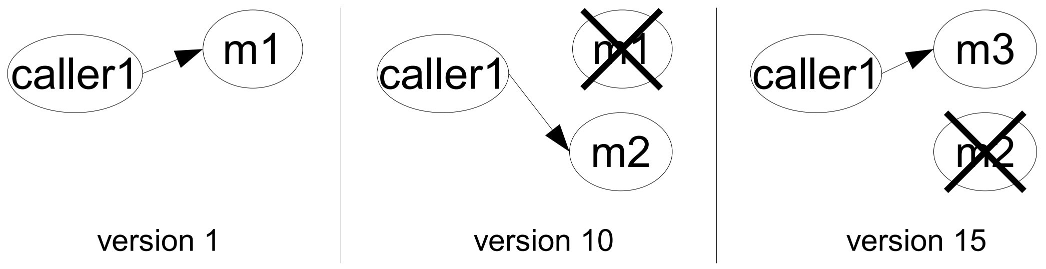


Figure 2.4: Changes chain

2.1.2 Change Chains

It is possible that during the course of a framework's evolution, a method is replaced several times, i.e., it is part of a *change chain*. As illustrated by Figure 2.4, a method might be renamed once in one version and renamed a second time in another version. Additionally, since 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 relevant. One solution would be to automatically check if the recommended call exists in the framework version used by the client program. Unfortunately, this is not an adequate solution if the recommended method is deprecated or if the method is no longer used by the framework but was not removed. 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 2.4 where our system would recommend to replace a call to `m1` by a call to `m3`.

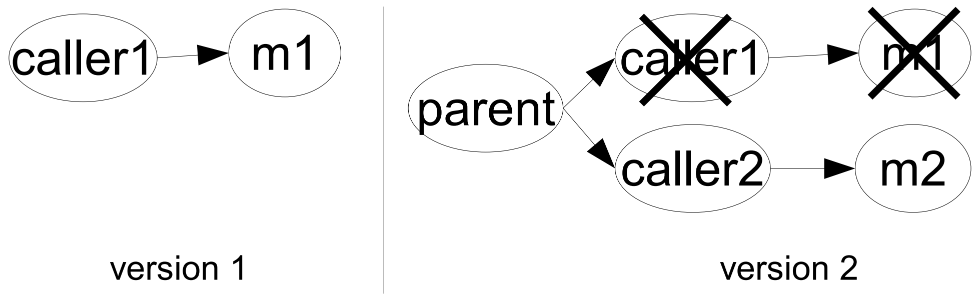


Figure 2.5: Callee and caller are deleted together

2.1.3 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 2.5 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 minus the methods that were previously called by the deleted caller. Figure 2.5 illustrates the case where `caller1` was replaced by `caller2` and `caller2` replaced a call to `m1` by a call to `m2`.

When finding a replacement for a caller method, SemDiff can generate multiple recommendations (e.g., the method was splitted, there are truly multiple relevant replacements, false positives, etc.). To reduce the impact of false positives, we first remove from the potential caller 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. We chose a threshold of 0.6 because it appeared to offer the best results during initial prototyping of the approach.

2.1.4 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, etc.). In these cases, the framework is not being adapting 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.

2.1.5 Complexity

The main factors affecting the computational complexity of SemDiff's algorithm are the number of methods that removed a call to the queried method, the number of different added calls, the maximum change chain length, and the maximum length of unstable callers.

2.1.6 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.

2.1.7 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, i.e., 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. In practice, however, we expect changes to root methods to be significant, and as such to be more likely to be documented.

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 those 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 those 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 [22]: it is possible that our approach could directly benefit from such developments.

2.2 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 those high-level changes in a database for future use by our recommendation system.

2.2.1 Source Repositories

Currently, SemDiff provides adapters to retrieve information from CVS [2] and Subversion (SVN) [13] 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. On the other hand, 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 [48,52]: 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 [51].

The merging of branches is another issue that arises when analyzing software repositories. This operation is not explicitly documented by neither CVS nor 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 [48, 52]: we ignored change sets involving more than 40 files.

2.2.2 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

```
// 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());
}
```
Figure 2.6: Outgoing call differences

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 target of a call in a given source file if the class defining the callee is not part of 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 non-trivial change into smaller and incremental changes (i.e., change sets). Combining the granularity of the change set with the quality of full 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 differ 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, provides a list of all methods that were added, removed and modified. 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 2.6, CallDiff would indicate that between version 1.1 and version 1.2, the call `PrintStream.println()` 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 2.3.

2.2.3 Persisting Changes

After the execution of the analyses for a change set, the results are stored in a PostgreSQL database [11] and made available to our recommender.

2.3 Analyzing Partial Programs

With Java, most parsers and static analysis programs fail to reconstruct the complete type hierarchy if they only receive as input a subset of the program source code without the

```
 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 2.7: Partial Program Analysis Example

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 2.7, 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 modified the Soot Java static analysis framework [47] and the Java source parser it uses, Polyglot [40]. We present in this section an overview of our approach [20] and how it relates to SemDiff; we give the details about the implementation of our approach in Chapter 4.

To recover the types in partial programs, 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 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()`.

Recovering types in partial programs is a fundamentally undecidable problem, which means that our technique can produce erroneous results. For example, our technique infers that the method `Util.method2()` at 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 inference result, `Z`, is not equals to the formal type, but it is in its hierarchy. As we found in the empirical evaluation of our technique (see Chapter 5), our technique recovered most types in a partial program:

on average, 91.2% of the recovered types were correct, 6.1% were imprecise (`unknown` or *hierarchy-related*), and 2.7% were erroneous.

SemDiff needs to take into account that the type information of a call might be incomplete and imprecise. 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 refers 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`. Indeed, 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 at least on matching call. 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 then try to find 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.

Chapter 3
Evaluating the SemDiff Approach

The main strategy underlying SemDiff relies on a number of hypotheses we made on 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. Can SemDiff recommend to a client program adaptive changes that replace a functionality deleted during a framework's evolution?
2. Is the confidence value good enough to discriminate relevant recommendations from false positives?
3. Can SemDiff detect changes more complex than refactorings?

3.1 Study Design

To answer the above questions, we performed an historical 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 or that was deprecated (as determined through warnings based on the use of the *@deprecated*

Client	**Eclipse 3.1**	**Eclipse 3.3**
Mylyn	0.5	2.0
JBoss IDE	1.1	1.5
jdt.debug.ui	3.1	3.3

Table 3.1: Client program versions

javadoc tag), we used the SemDiff recommender and a refactoring detection tool to see if we could 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, in our experience, client programs that depend on JDT always depend on at least one of those 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. We chose to study those plug-ins across three major revisions (3.1, 3.2 and 3.3) to increase the odds of finding non-trivial changes and change chains.

Finding suitable client programs was an 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 [1] 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 [5], 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.

We found three client programs that met our study criteria: **Mylyn** [33], a task-focused environment, **JBoss IDE** [7], a development environment for the JBoss web application server, and **jdt.debug.ui**, the Java debugging environment in Eclipse. Table 3.1 gives the client program versions used for Eclipse 3.1 and Eclipse 3.3.

SemDiff. We used SemDiff to analyze the source history of the Eclipse framework. SemDiff processed 10127 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 was deprecated or 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 non-trivial changes that occurred during the framework's evolution from simple refactorings. We chose RefactoringCrawler [23] 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 Chapter 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`).

Following the tool author's recommendations,[1] 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 non-trivial 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 a the first version of a client program, we tried to find a refactoring involving the called method. If we found such a 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.

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

3.2 Results

Table 3.2 shows the results of our study. For each client program, we list the number of compilation errors related to the JDT framework (Errors), the number of errors within the scope of our approach (Scope), the number of errors that could be fixed based on the top three SemDiff recommendations (SemDiff) and the number of errors that could be fixed based on the refactorings detected by RefactoringCrawler (RC).

The number of errors represents all deprecated accesses and 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 8 were within the scope of our approach. SemDiff provided relevant recommendations for 8 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 (or RefactoringCrawler). Because SemDiff takes as input a method and gives as output a list of methods, we only considered unresolved and deprecated 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 Scope column for jdt.debug.ui represent two interpretations of the scope of our approach. Although there were 19 errors that are applicable to our approach, five 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 14 as the number of adaptation problems for which there is an objectively verifiable solution.

The execution of the repository analysis framework took 16 hours on a Pentium D 3.2 Ghz with 2 Gb of RAM and running Ubuntu Server 7.04. This analysis needs only to be performed once before a user can make requests in different disconnected sessions. On average, each request took 1 second to complete. Running the three analysis with RefactoringCrawler took 13 hours.

Relevant recommendations. SemDiff found relevant recommendations for 89% 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

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 3.2: Relevant recommendations by SemDiff and RefactoringCrawler

returned an instance of the `TypeInfo` class presented in Section 1.1. Those two replacements were a call to the `JavaSearchTypeNameMatch` constructor and a call to the method `SearchEngine.createTypeNameMatch(...)`, both returning a subclass of `TypeNameMatch`.

Again in Mylyn, SemDiff was able to correctly recommend a call to `OpenTypeHistory.-remove(TypeNameMatch)` which replaced a call to `OpenTypeHistory.remove(TypeInfo)`. Although this change might appear to be a simple refactoring, it was not detected by RefactoringCrawler as such. 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.

Finally, in jdt.debug.ui, SemDiff was not able to provide the correct recommendation for four methods. These are protected methods defined in the class `TypeSelectionDialog2` and accessed solely by this class. Because `TypeSelectionDialog2` was deprecated but not removed in Eclipse 3.3, no call to those four methods were removed and SemDiff could not generate any recommendation. As an alternative strategy, we asked SemDiff to find a replacement for a public method of `TypeSelectionDialog2`, and this time, the recommender suggested a method from a new class that was used by jdt.debug.ui. Indeed, the framework was adapted to the deprecation of a public method, but not to the deprecation of the protected ones.

Confidence value. In most cases, the confidence value was necessary to discriminate relevant replacements from false positives because SemDiff produced an average of 7.1 recommendations per request. In average, there were 1.6 recommendations with a confidence value of 1.0 per request. The support of the relevant recommendations had an average of 2.2 methods. Because this low support was enough to distinguish relevant recommendations from bad ones, we consider this to be evidence that our approach can work only by analyzing the code of the framework itself and does not require a set of examples using the framework.

Non-trivial changes. Although RefactoringCrawler found 319 refactorings between JDT releases 3.1 and 3.3, only one of them was related to errors in the client programs we studied (the six errors reported in Table 3.2). This observation provides evidence that our approach

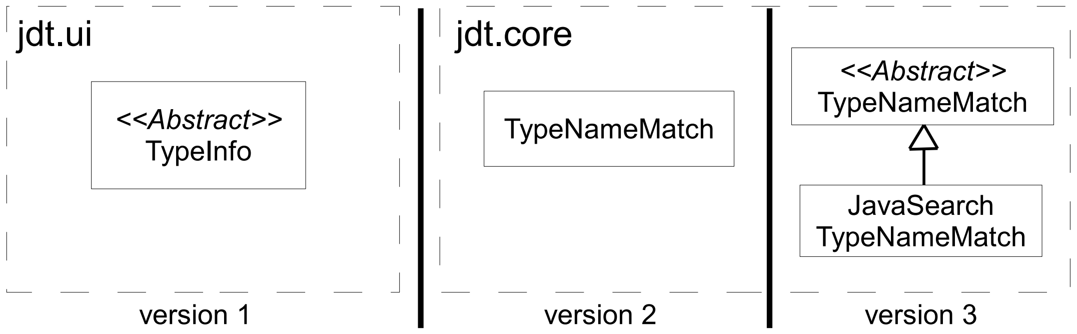


Figure 3.1: Evolution of TypeInfo into JavaSeachTypeNameMatch.

works in the face of non-trivial 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 3.1, this change spanned across the two `jdt` plug-ins and was part of a change chain.

Another interesting recommendation was provided for the JBoss IDE: SemDiff recommended to replace the constructor of `ListContentProvider` with the constructor of `Array-ContentProvider`. 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 non-trivial changes, SemDiff used the three heuristics presented in Section 2.1 several times when recommending adaptive changes to the three client programs: change chains were detected in 6% of the requests, caller replacements had to be found in 6% of the requests and change sets with spurious calls were removed 67% of the time.

Summary. SemDiff was able to recommend relevant adaptive changes in the face of framework evolution in 89% of the time. The fact that RefactoringCrawler was only able to detect a small subset of the changes that broke 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 6 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 affecting the development process.

3.3 Threats to Validity

The external validity of this study is limited by the fact that we only studied the evolution of the Eclipse JDT framework and it might not be 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, we can reasonably assume that the results we obtained were not related to a particular developer profile.

To evaluate the relevance of our recommendations, we analyzed 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. We cannot assess how the developers would have used our recommendations, although such an assessment would form a natural next step in the evaluation of our approach.

Finally, by using client programs and a refactoring detection tool, we tried to limit 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.

Chapter 4
Partial Program Analysis

Static program analysis is an important tool for software engineering research in general: techniques such as bug detection [14] and feature location [50] heavily depend on static analyses to model a program's behavior and structure.

Compiler frameworks, with which many static analyses were developed, usually assume that the complete program is available (either as source code or as a high-level binaries such as Java .class files), even if only part of that program is to be analyzed. When the complete program is available, it is straightforward for the compiler to build a correct, typed, and complete intermediate representation (IR) for the part of the program to be analyzed.

However, some methodologies used by software engineering techniques preclude the access to complete programs. Indeed, source code retrieved from software versioning systems [52], web repositories [46], or bug reports [16], is typically difficult to compile: source folders and libraries required to build a snippet of code may not be known or accessible, and the correct versions of those code artifacts may be impossible to automatically determine (e.g., which version of Log4J is needed to compile Foo.java 1.4?).

Because SemDiff only analyzes partial programs, we devised a technique that produces complete and typed intermediate representations for the source code of partial Java programs, even when only part of the program is accessible. The unavailability of many class declarations leads to two main challenges: (1) dealing with syntactic ambiguities and (2) determining the correct types for expressions such as field accesses and method calls.

Syntactic ambiguities arise when classes and members for other parts of the program are not available. For example, consider the statement `E.dothat()`. Without the declaration for `E`, it is not possible to determine if `E` is a class or a field. If `E` is a missing class, it could be a call to a static method `dothat()` which is a member of the missing class `E`. Otherwise, if `E` is a missing field, then this is a virtual method call, with receiver `E`.

```
 1  class A {
 2    String p1;
 3    void add(Object o) {}
 4  }
 5
 6  class B {
 7    void main() {
 8      A a = new A();
 9      a.p1 = "hello";
10      a.add(a.p1);
11    }
12  }
```

Figure 4.1: A complete program

Typing problems arise when a compiler or tool does not have access to the complete type hierarchy and the signatures of fields or methods in classes that are directly or indirectly referenced by the program under analysis. A Java compiler usually creates an intermediate representation (IR) such as an abstract syntax tree, annotating the IR with the appropriate types, based on type declarations. For example, given the complete Java code snippet shown in Figure 4.1, the compiler would use the declaration of class `A` at line 1 to find that the declared type of the expression `a.p1` at line 9 is `String`. The compiler would also use the declaration of class `A` to find that the method called at line 9 is the method `A.add(Object)` declared at line 3.

Tools like SemDiff do not have access to the complete program and all of its declared types. For example, assume that SemDiff had access only to the source code for class `B` (and not `A`). In this case, our tool cannot find the declared type of some expressions in the incomplete program (e.g., what is the declared type of `a.p1`?). To deal with this problem, other tools often fall back on syntactic analysis, which provides limited information. For example, if only class `B` is available, a syntactic analysis will conclude that a method named *add* with *one* parameter is called at line 10. This lack of precise type information is a problem for tools, like PARSEWeb [46] and SemDiff, that analyze partial programs to recommend to programmers method calls from arbitrary frameworks. These tools require the types of the receiver and the formal parameters in order for their recommendations to be useful (e.g., telling the user to call a method named *add* is not helpful if many classes declare such method). These tools thus need to perform partial type inference to get more information from the source code of incomplete programs. For example, if only the declaration of class `B` was accessible, SemDiff should conclude that at line 10, the method `A.add(String)` is called by looking at the assignment of field `p1`. This information, although not strictly correct (there is no method `A.add(String)`, but there is a method `A.add(Object)`), is more precise than the one provided by syntactic analysis.

Software engineering techniques usually tolerate a certain level of imprecision and errors, as measured by precision and recall, so they can benefit from information that is often more precise, but potentially incorrect. Thus, in designing our approach we traded some guarantees on correctness for increased precision. This also implies that our approach is not suitable for situations where a sound analysis is required, such as in program optimization.

We devised a technique, *Partial Program Analysis* (PPA), which builds a typed IR of incomplete Java programs' source code. Our technique recovers the declared types by performing partial type inference and resolves syntactic ambiguities inherent to incomplete programs using heuristics. Although it is impossible to guarantee that the generated IR is correct with respect to the result one would get given the complete program, we aim to generate an IR which: (1) obeys the type constraints available in the partial program under analysis, (2) does not introduce unknown program constructs, and (3) is suitable for use with other static analysis tools.

We implemented this approach in a prototype using Soot, a static analysis framework [47], and Polyglot [40], an extensible compiler framework. Currently, our prototype transforms Java source code from an incomplete program into a typed abstract syntax tree (AST) and into Jimple, a typed three-address intermediate representation, because those representations are suitable for most static analyses. Our positive experiences with SemDiff gave us some confidence that PPA will be useful for developing other software engineering analyses and tools.[1]

To validate to what extent our proposed PPA technique produces useful results, we performed a quantitative study on four open source programs, three of them from the DaCapo benchmark suite [45], for three common scenarios. We found that even for the hardest scenario, when the source code of only one class is available, partial program analysis could generate an intermediate representation that was on average 91% identical to the intermediate representation of the same class analyzed with the whole program.

Because partial program analysis can benefit various software engineering tools, the contributions of our approach goes beyond SemDiff's requirements. The contributions brought by our approach are: (1) the PPA techniques that allow us to analyze incomplete Java programs, and that deal with both syntactic ambiguities and typing problems, (2) the implementation of a tool based on PPA that produces an IR and AST representation of an incomplete program, and (3) an empirical evaluation of our approach.

[1]The PPA implementation is available at http://www.sable.mcgill.ca/ppa.

```
 1  class D {
 2    int field1;
 3    void main() {
 4      A varA = new A();
 5      String s = "hello";
 6      field1 = m1(s);
 7      varA.field3 = s;
 8      varA.field3 = B.field3;
 9      Object o = new String("hello");
10    }
11    short m1(Object o) {return 0;}
12  }
```

Figure 4.2: A partial program

In the remainder of this chapter, we introduce our problem and terminology in more detail in Section 4.1. We then describe our approach to solving the typing problems in Section 4.2 and the ambiguous syntax problems in Section 4.3. We put it all together in Section 4.4, where we describe our overall algorithm. Finally, in Chapter 5, we report on the results of the empirical evaluation of our technique.

4.1 Partial Java Programs

We consider a partial program to be *a subset of a program's source files*. This definition is suitable for current software engineering tools that can get as input complete source files and that require more precise information than what syntactic analysis can provide.[2] In a source file, we associate a *type fact* with all references to *declared types*. For example, there are six type facts associated with line 6 in Figure 4.2: the declared type of field1's container (`D`), the declared type of field1 (`int`), the declared return type of method m1 (`short`), the declared type of the method m1's target (`D`), the declared type of the method m1's formal parameter (`Object`) and the declared type of the argument (`String`). To simplify our presentation, we will use the term *type* to refer to a declared type (as opposed to a runtime type) in the remainder of this chapter. For example, at line 9, the declared type of the variable `o` is `Object`, but the type of this variable during runtime is `String`.

Given a partial program, the challenge is to recover as many correct type facts as possible without having access to the rest of the program, i.e., referenced source files, binaries, or dependencies such as libraries. For example, by analyzing only class `D`, we can

[2]This definition of partial programs could even be relaxed to include any snippet of well-formed code. Although our approach does not rely on a complete source file, the parser implementation that we currently use does.

infer two type facts at line 7: (1) the type of the container holding `field3` is `A` or one of its ancestors and (2) the type of `field3` is a `String` or one of its ancestors. Furthermore, we know at line 7 that the type of `varA` is `A` because we have access to its declaration in method `main`. In the remainder of this chapter, we will use static fields in our examples (e.g., `B.field3` at line 8) instead of instance fields to reduce the size of the examples: our analysis considers each syntactically different field access as a distinct field, even if they share the same name. For example, at line 8, PPA considers that there are two distinct fields `field3`: the first is attached to the local variable `varA` and the second is attached to the type `B`. In the text, we will also always refer to the field name without the qualifier when it is unambiguous.

Having access to only a subset of the source files forces us to make an important assumption when performing partial program analysis:

Compilable Program Assumption: The source files of a partial program compile without any error given the required dependencies.

This is a reasonable assumption for code extracted from software versioning systems or web repositories because a popular convention is to only commit source files if they compile. The absence of class declarations makes it impossible to detect type-related errors such as calling a non-existing method so we cannot assess whether the code is compilable or not. The compilable program assumption is thus necessary to infer type facts: in potentially uncompilable source code, every inference made on type usage could be wrong.

In this chapter, we will focus on Java 1.4 which does not include features such as generics and autoboxing, and we will also assume that the user has access to standard Java types (either in a binary or source format) such as `java.lang.Object`. Again, this is a reasonable assumption because those classes are required to execute any Java program.

4.2 Recovering Types in Partial Programs

When analyzing a partial Java program, it is possible to infer type facts by looking at how a type is used in the program. For example, in Figure 4.3, we see that the program assigns an instance of class `B` to `field1` at line 3. Because of the Java type system, we know that `field1` must be `B` or one of its ancestors.

We define the operator $dt(x)$ which returns the declared type t of the Java expression x. For example, at line 4, $dt($`coll`$) =$ `Collection`. We also define the *subtype* operators, $t1 <: t2$, which means that $t1$ is a subtype of $t2$, i.e., it is either $t2$ or a descendant of $t2$. The *supertype* operator, $t1 :> t2$, means that $t1$ is either $t2$ or an ancestor of $t2$. For the last two operators, we consider class extension and interface implementation and extension

```
class E {
  void main() {
    A.field1 = new B("hello");
    Collection coll = A.field2;
    A.field1.m2().m3();
  }
}
```

Figure 4.3: Inferring type facts

(through the `extends` and `implements` keywords) to be the only generators of subtypes and supertypes. We use the *related type* operator, $t1 \sim t2$ when we know that two types are related, i.e, one is a subtype of the other or they share a common ancestor or descendant.[3] Finally, we define the operator $target(x)$ which returns the target's type t of a method x or the container's type of a field x. For example, $target($`field1`$) :>$ `A`. Because partial programs are imprecise, a type t can be either precise (e.g., $dt($`coll`$) =$ `Collection`) or imprecise (e.g., $dt($`field1`$) :>$ `B`).

When we try to infer a type fact, it sometimes happens that we cannot recover any information at all. We thus define the following data structures to handle these cases:

The `unknown` type is used as a placeholder for any type that is referenced, but not explicitly named. For example, in the following call chain, `m2().m3()`, the return type of the method `m3` is `unknown`, which denotes either a Java primitive, a Java class or `void`.

To fully qualify any type which has an ambiguous Fully Qualified Name (FQN), we define the package `p-unknown`. This package is necessary to distinguish a type located in the default (empty) package from a type that is located in an unknown package. The fully qualified name of the `unknown` type is thus `p-unknown.unknown` even if for the sake of brevity, we will always use the short name `unknown`.

Finally, we define a *type fact* as being a record containing the following attributes: (1) a Java expression x (e.g., a reference to a field), (2) the type t of the expression before the inference, and (3) the type t of the expression after the inference. For example, if we just found that the unknown field `field1` was a supertype of class `B`, we would have inferred the following type fact: {`field1`, `unknown`, :> `B`}.

[3]In Java, because all reference types are a subtype of `java.lang.Object`, all reference types are related to each other. The related type operator can still be used to distinguish reference types from primitives as they are not related.

4.2.1 Type Inference Strategies

To infer type facts, we rely on several strategies based on the Java programming language type system. These inference strategies are sound in the sense that the real declared types will always respect the constraints of the inferred type facts. For example, if a strategy infers that the type of an expression is a subtype of `java.util.List`, the real type is guaranteed to have this property. Although the inference strategies can generate imprecise type facts such as {`field1`, `unknown`, <: `java.lang.Object`}, we found during our evaluation that they generally recover the real type. Table 4.1 shows the intuition behind the inference strategies that we devised. The formal inference rules for each of those strategies are presented in Appendix A.

Inference Strategy	**Example**	**Explanation**

Assignment	`B.field1 = "Hello World";`
	`C c = B.field2;`
		The type of an unknown expression on the right-hand side is the subtype of a known left-hand side expression's type and vice-versa, e.g., {`field1`, `unknown`, :> `java.lang.String`}. and {`field2`, `unknown`, <: `C`}.

Return	`int m1() {`
	`  return B.method2();`
	`}`
		The type of an unknown return expression is the subtype of the method's declared return type, e.g, {`method2`, `unknown`, <: `int`}.

Method binding	`void main() {`
	`  B.field3 = m3(B.field4);`
	`}`
	`D m3(E p1) {...}`
		If we know the exact method binding, the types of the actual parameters are subtypes of the formal parameters' types, and the expression to which the method is assigned to is a supertype of the method's return type, e.g., {`field3`, `unknown`, :> `D`} and {`field4`, `unknown`, <: `E`}.

Condition	`if (B.method4()) {`
	`  ...`
	`}`
		An expression used as a condition must resolve to a boolean, e.g., {`method4`, `unknown`, `boolean`}.

Binary and unary operators	`int i = B.field7 - 10;`	Depending on the operands' types and the expected return type of a binary or unary expression, it might be possible to infer the primitive type of an expression by taking into account implicit type promotion, e.g., {`field7`, `unknown`, <: `int`}.

Array	`B.field10 = B.field8[B.field9];`	When an unknown expression is used as an array index, we can infer that it is a subtype of `int`. For example, at line 14, we can infer that {`field9`, `unknown`, <: `int`}. When an unknown expression is accessed using an array index, we can infer that the type of the expression is an array, e.g., {`field8`, `unknown`, `unknown[]`}.
Switch	**switch**(B.field11) { ... }	Switch expressions enable us to infer that the operand is a subtype of the `int` primitive, e.g., {`field11`, `unknown`, <: `int`}.
Conditional	`B.field14 = B.field12 ? 'c' : B.field13;` **int** `i = B.field12 ? 'c' : B.field16;`	Conditional expressions, represented by the ternary operator `?` can be used in two fashions. First, if their return type is not know, we can at least infer that the two last operands' type must be related, e.g., {`field13`, `unknown`, ∼ `char`}. Indeed, if *dt*(`field14`) = `long`, `field13`'s type can be either a subtype or a supertype of `char`: we can thus only say that the two types are related. When the return type is known, we know that the last two operands must be a subtype of the return type, e.g., {`field16`, `unknown`, <: `int`}.

Table 4.1: Type Inference Strategies

4.2.2 Inferring Method Bindings

When an expression's type has been inferred (e.g., using one of our inference strategies), it is sometimes possible to determine a method binding that is ambiguous from a purely syntactic point of view. Consider the call to method `m1` at line 4 in Figure 4.4. Because there are two declarations of a method `m1` with one parameter, it is not possible to decide what method is called when looking only at the syntax of the program. On the other hand, if we perform some type inference, we know that *dt*(`field1`) :> `B` at line 3, and we are certain that we call the method `m1(B)` declared at line 10. Once we know the exact method binding, we can then use our *method binding* inference strategy. Unfortunately, there are cases like line 5 where we cannot identify the correct method binding because the method is overloaded and the declared type of the parameter is unknown.

A class can also potentially overload a method declared in a supertype. For example, in class `H`, we cannot infer that the method called at line 17 is the one declared at line 20. Indeed, we know from line 16 that *dt*(`field3`) :> `C`. Suppose that *dt*(`field3`) = `Object` (which

```
1   class G {
2     void main() {
3       A.field1 = new B();
4       A.field4 = m1(A.field1);
5       A.field5 = m1(A.field2);
6       B.m2(2);
7       B.m2(A.field2);
8     }
9
10    D m1(B b) { ... }
11    E m1(int i) { ... }
12  }
13
14  class H extends I {
15    void main() {
16       A.field3 = new C();
17       m1(A.field3);
18    }
19
20    void m1(C c) { ... }
21  }
```

Figure 4.4: Method binding

respects the type fact we inferred) and that the supertype of `I` defines a method `m1(Object)`. It follows that the method called at line 17, is `I.m1(Object)` and not `H.m1(C)`. Determining a method binding is thus an undecidable problem because of overloaded methods.

4.2.3 Inferring Type Members

Until now, we focused on inferring the type of an expression, but, at the same time, we need to infer the existence and types of members (i.e., fields and methods). For example, in Figure 4.4, we inferred at line 3 that there is a field named `field1` that is declared in the type `A` or one of its supertypes. When analyzing a complete program, we could check whether these members exist and are accessible. Because we only have access to class `G`, we must rely on our assumption that the underlying code compiles and that both members are accessible from the context of the calling method `G.main`.

More specifically, when PPA encounters a reference to a type whose declaration is not available, it creates an internal representation of the type. If a member is accessed from this type, we add the member to the generated type declaration. For example, at line 3 in Figure 4.4, PPA would generate the fact that class `A` has a field called `field1` whose type is :> `B`.

```
 1  class F {
 2    void main() {
 3      Object o1 = A.field3;
 4      String s1 = A.field3;
 5      A.field4 = new Object();
 6      String s2 = A.field4;
 7      B b = A.field5;
 8      C c = A.field5;
 9    }
10  }
```

Figure 4.5: Combining type facts

In adding missing members, fields and methods are treated differently. Since fields cannot be overloaded, we only generate a missing field once and reuse this one if another occurrence of the same field occurs. However, since methods can be overloaded, generated methods cannot be reused like generated fields. For example, at line 6, we can infer that there is a method called `m2` that is in one of the supertypes of `B` and that takes as a parameter a supertype of `int`. At line 7, we cannot safely reuse this fact to infer the type of the actual parameter `field2` because there might be another method called `m2` with a different parameter type. We thus infer that there is a method called `m2` that takes a parameter of type `unknown`.

Finally, inferred type members can be refined by type inference. For example, if we later find that *dt*(`field2`) <: `C`, we will add a method `m2(C)` to `B`.

4.2.4 Combining Type Inference

Sometimes, we can infer two type facts related to the same expression. For example, in Figure 4.5, at lines 3 and 4, we infer that *dt*(`field3`) <: `Object` and *dt*(`field3`) <: `String`. By definition of a subtype, it is clear that *dt*(`field3`) <: `String` because `String` <: `Object`. We say that the two inferred type facts are *converging* and we only keep the most precise type fact (<: `String`). On the other hand, the type facts that we infer at lines 5 and 6 are *erroneous*: *dt*(`field4`) :> `Object` and *dt*(`field4`) <: `String` cannot be true at the same time. Erroneous type facts contradict our compilable program assumption, but this is one of the few cases where we can detect a compilation error: in this case, PPA reports an error. Finally, the two last type facts at lines 7 and 8, *dt*(`field5`) <: `B` and *dt*(`field5`) <: `C`, are *conflicting*: it is not possible to decide which of the two type facts is the most precise because three type hierarchies can explain the code of lines 7 and 8[4]:

[4]The converse is also true if we have the two following type facts: *dt*(`field5`) :> `B` and *dt*(`field5`) :> `C`.

```
class Y {
  int m1() {
    A a1 = Z.field1;
    Z.field1 = Z.field2;
    A a2 = Z.field3;
    Z.field4 = Z.field3;
  }
}
```

Figure 4.6: Conflicting type direction

1. `B` <: `C`, so dt(`field5`) <: `B`
2. `C` <: `B`, so dt(`field5`) <: `C`
3. There exists a type `P` which is a common descendant of `B` and `C` (either `B` or `C` must be an interface). In that case, neither type fact is more precise.

In the case of the third possibility, even if we knew the whole type hierarchy of `B` and `C`, it would still be impossible to determine the type of `field5` because there might be more than one common descendant `P`.

When we encounter two conflicting type facts, we first try to select the safest one, where we determine that a type fact is safer than another using the total ordering: `unknown` < `missing` < `super missing` < `full`. Each member of this ordering is defined as follows:

If the type of a fact is *unknown*, it is less safe than a fact whose type is known but whose declaration is *missing* (e.g., we know that $dt(x)$ = `B`, but we do not have access to the declaration of `B`). A known type with a missing declaration is less safe than a known type whose declaration is accessible but not all of its *supertypes* (e.g., we have access to the declaration of `B`, but one of its supertype's declaration is missing). Finally, the safest type, *full*, means that we have access to its declaration and the declaration of all of its supertypes. If the two type facts are equally safe, we keep the first type fact that we inferred. The rationale behind this scheme is that we only keep types that allow us to work with safer (i.e., known) types, which is generally more precise than just keeping the first inferred type fact and ignoring further type facts.

Another alternative strategy would be to treat all related type facts as constraints and try to solve the constraints to obtain the most precise type fact. However, during our early experimentation with PPA, we observed that we generally produced either one type fact or multiple conflicting type facts (e.g., `field5` in Figure 4.5), which forced us to make an arbitrary choice. Therefore, we had no reason to think that a more complex type combination scheme involving constraint solving would produce more accurate results.

Finally, when combining type facts, the direction of the types, whether they are subtypes or supertypes, might conflict. In Figure 4.6, we can produce this *inference chain* at lines 3 and 4: $dt($`field2`$)$ <: $dt($`field1`$)$ <: `A`. It is thus clear that $dt($`field2`$)$ <: `A` by transitivity. On the other hand, the directions of the types at lines 5 and 6 conflict: $dt($`field4`$)$:> $dt($`field3`$)$ <: `A`. We can thus only say that $dt($`field4`$) \sim$ `A` or in other words, that there is a path in the type hierarchy that links `field4` with `A`.[5]

4.3 Ambiguous Syntax in Partial Programs

The programming language syntax is a source of imprecision: Table 4.2 shows the main constructs that are ambiguous in partial Java programs.

Syntax ambiguity	**Example**	**Explanation**
Fully Qualified Name (FQN)	Figure 4.7	The FQN of a type cannot always be soundly inferred in a partial program because a programmer can use the `import *` construct. For example, at line 9 in Figure 4.7, the FQN of `C` can either be: `C` (in the default package), `ca.mcgill.C` (in the package of `A`), or `ppa.C` (because of `import ppa.*`). Additionally, we cannot soundly infer the FQN of a type contained in a known package (e.g., `java.util`). For example, at line 10, the `Collection` type might be contained either in `java.util` or in `ca.mcgill`. Line 11 gives a hint that the latter FQN is the good one since the `java.util.Collection` type does not declare the method `doThis`.
Package or Class?	Line 13 in Figure 4.7	It is not always possible to discriminate the part in the FQN that relates to a type from the part that relates to the package. For example, at line 13, variable `d` might be of type `D` or of type `internal.D` (an internal class).

[5]Unfortunately, in Java, this is true for any two given reference types because every type is a subtype of `Object` so there is always a path from one type to another type that passes by `Object`. In a language like C++ which does not have this concept of a universal supertype, related types would have a more precise meaning.

Field or Class?	**class** B **extends** C { **void** main() { E.doThat(); E = **new** F(); } }	An expression such as `E` in the first line of the `main` method can be either a field or a class. In the former case, we infer that *target*(`doThat`) = `unknown` and in the latter case, we infer that *target*(`doThat`) :> `E`. It is sometimes possible to resolve this ambiguity by looking at other lines of code (such as the assignment) that provide hint that the expression is a field.
This or Container?	**class** G **extends** H { **public void** main() { I i = **new** I() { **public void** m1() { f1 = 2; ...	It is not always possible to soundly infer the container of a particular member in an internal class because a reference to `this` or to the `container` of an internal class is implicit in Java. For example, it is not clear whether *target*(`f1`) :> `I` or if *target*(`f1`) :> `H`.
Overloaded operators	**class** J **extends** K { **void** main() { **int** i = 2 − f2; String s = "Hello"+(f3+2)+ "World"; } }	Some operators are overloaded by the Java language. For example, it is not clear whether the `+` operator is the addition operator or the String concatenation operator in the `main` method. In the latter case, because a String can be concatenated with an arbitrary type instance or a primitive, it is still not possible to soundly infer the type of the field `f3`.

Table 4.2: Ambiguous syntax constructs in Java

When we encounter such ambiguous syntax constructs, we can either (1) create an unknown node in the AST representation, or (2) use an heuristic that guesses the real construct. The first strategy is sound, in the sense that it doesn't introduce a potentially wrong construct. However, it potentially introduces many unknown parts of the code, losing useful parts of the program. Furthermore, it breaks the compatibility with client tools, which assume only valid Java constructs. We relied on the use of heuristics that can produce wrong, but potentially more precise results.

Fully Qualified Name. When we encounter a reference to a simple type name (e.g., `String`), we use the following heuristic to find the FQN of the ambiguous type:

1. If the ambiguous type is fully qualified, we use that FQN (e.g., `ppa.internal.D` in Figure 4.7).
2. If there is an explicit import statement that ends with the ambiguous type name, we use the FQN specified in the import statement (e.g., `soot.Unit` in Figure 4.7).

3. If we have access to the packages imported using a wildcard import statement (e.g., `import java.util.*`) and we find a type whose name is the same as the ambiguous type name, we use that FQN (e.g., `java.util.Collection` in Figure 4.7). If later on we realize that this type is *not adequate* (e.g., we are calling a method that is not declared in this type), we rely on the last two heuristics to determine its FQN.
4. If there is no wildcard import statement, we append the name of the ambiguous type to the package of the analyzed type (e.g., `ca.mgill.C`).
5. If there is at least one wildcard import statement, we append the name of the ambiguous type to the unknown package (e.g., `p-unknown.C` in Figure 4.7).

Rules #3 and #4 can lead to wrong fully qualified names because the type might be declared in the default package. We expect most programs to avoid defining types in the default package because this practice is discouraged and often impractical.

Package or Class? We always consider the last part of a fully qualified name (after the last dot) to be the simple name of the type and the rest of the FQN to be the package. This can be a false assumption if the FQN refers to an internal type. A false assumption has no impact on the FQN (it is the same no matter if the type is internal) but it changes the type of node in the AST representation. Thus, if at a later point we find that the initialization can only refer to an internal type, we modify its AST representation.

Another strategy would be to use the Java naming convention (a type name and a package name should respectively start with an uppercase and lowercase character) to

```
package ca.mcgill;

import ppa.*;
import java.util.*;
import soot.Unit;

class A {
  void main() {
    C c = B.getC();
    Collection coll = B.getCollection();
    coll.doThis();
    Unit u = B.getUnit();
    ppa.internal.D d = B.getD();
  }
}
```

Figure 4.7: Ambiguous fully qualified name

determine which part of the FQN is the package and which part is the type. We would still need to tune this heuristic on a per project basis.

Field or Class? We consider any ambiguous reference (e.g., `E.doThat()`) to be a static method call from a class. If we find a hint contradicting that assumption (such as the instantiation of the ambiguous reference), we change the AST node accordingly. Like the previous heuristic, we could also use the Java naming convention.

This or Container? Most of the time, an ambiguous reference to `this` or to the container of internal types is impossible to resolve. We thus chose to always replace such ambiguous references by a reference to `this`.

Overloaded operators. When we encounter an overloaded operator such as `+` or `&`, we always consider that the type of the operands is `unknown`. We use the binary operator inference strategy to decide the type of the operands when it is possible.

4.4 PPA Algorithm

Figure 4.8 shows an overview of the algorithm which consists of three passes. Although the general techniques introduced in this chapter could be used in other systems, we implemented our approach using Polyglot [40] and Soot [47]. Polyglot is an extensible compiler that creates an AST representation of a source file by applying various passes such as disambiguation, type checking and exception checking. Soot uses Polyglot as a frontend to parse Java source files and then transforms the AST into a three-address intermediate representation called Jimple that can be used to perform data flow analysis. Our algorithm mainly extends Polyglot and works at the AST level. We first review the three main passes of the algorithm and then discuss the different modes in which the algorithm can be executed, its termination property, and its time complexity.

Seed pass. The seed pass is performed while Polyglot builds the AST of a source file. First, Polyglot tries to disambiguate each AST node (e.g., it determines if the expression is a field reference, a method call, a local variable reference, etc.). We modified Polyglot so it infers the missing type members as described in Section 4.2.3 and uses the heuristics described in Section 4.3 to resolve ambiguous syntax constructs.

Once the AST nodes are disambiguated, PPA visits each node and applies the inference strategy (presented in Appendix A) that corresponds to the type of the visited AST node. For example, at line 4 in Figure 4.9, PPA can use the assignment inference strategy to infer the following type fact: {`field1`, `unknown`, <: `A`}.

We visit each node of the AST in postfix order because the type of the parent node is often determined by the children's types. During the visit, we generate type facts and

```
// Seed pass
for each node in AST do
  Disambiguate node
  Infer type facts
  Put and merge type facts into worklist
end for

// Type inference pass
while worklist is not empty do
  for each node impacted by type fact do
    Make node safer
    Infer type facts
    Put and merge type facts into worklist
  end for
end while

// Method binding pass
for each ambiguous method call do
  Select the first possible call binding
  Infer type facts
  Put and merge type facts into worklist
end for
while worklist is not empty do
  for each node impacted by type fact do
    Make node safer
    Infer type facts
    Put and merge type facts into worklist
  end for
end while
```

Figure 4.8: Partial program analysis algorithm

```
class Y extends X {
  int m1() {
    System.out.println(Z.field1);
    A a1 = Z.field1;
    Z.field1 = Z.field2;
    B b1 = Z.field1;
    Z.field2 = Z.field3;
    A.m1(Z.field1);
    System.out.println(Z.field4);
    m2(Z.field5);
  }

  void m2(C param1) {...}
}
```

Figure 4.9: Sample program

append them to a worklist. When two type facts refer to the same expression (e.g., we can infer two type facts at line 4 and 6 that are related to the field `Z.field1`), we merge them according to the combination strategy we presented in Section 4.2.4.

During the seed pass, we only *infer* type facts and the `unknown` type is assigned to all unknown expressions. If a complete and correct program was available, there would be no unknown types at this point. However, partial programs often have some unknown types after the seed pass. For example, in Figure 4.9, $dt($`field1`$) =$ `unknown`.

Type inference pass. Once the AST is built, we can use the type facts that we inferred to modify the nodes of the AST (called *Make node safer* in Figure 4.8). When modifying the type of a node, we keep the complete type fact in memory, but we can only assign a simple type to a node to simplify the usage of the AST. For example, if we have the following type fact, {`field1`, `unknown`, <: `A`}, we modify the declared type of the field nodes at lines 4, 5, 6, and 8 to be equal to `A`.

Finally, when we modify a node, it is possible that we can infer a new type fact. For example, at line 5, when we modify the assignment node, we can infer that {`field2`, `unknown`, <: `A`} using the assignment inference strategy. The inferred type facts are appended and merged into the worklist.

Method binding pass. When we build the AST and infer type facts, we can encounter ambiguous method calls. For example, at line 9, we do not know which `println` method is called: it is an overloaded method. During the seed pass and type inference pass, we only select a method binding if there is no ambiguity to make sure that we do not introduce potentially erroneous or conflicting type facts.

The method binding pass basically *forces* the compiler to select the first possible declaration of method calls that remain ambiguous. Once the declaration is selected, this enables the inference of new type facts that are appended and merged into the worklist. The worklist is then processed like in the type inference pass. For example, if we executed this pass on the program listed in Figure 4.9, we would find that two method calls remain ambiguous: the call to `println` at line 9 and the call to `m2` at line 10 (because `Y` extends `X`, the call might refer to a method declared in `X`). By forcing the selection of a method declaration, we would conclude that the method `println(boolean)` is called at line 9 and `m2(C)` at line 10, which would lead to the inference of the two following type facts: {`field4`, `unknown`, `boolean`} and {`field5`, `unknown`, <: `C`}.

Results. Once the three passes are completed, PPA produces a typed abstract syntax tree. Each node in the AST contains three kinds of information: (1) a type, (2) whether the declaration of the type is available (i.e., the source is accessible) or was generated by PPA, and (3) whether the type is unsound. A type can be unsound if it was inferred using one of our syntax heuristics or in the case of Java, if it was inferred through the related type operator ($\sim$). As seen in Section 4.2.4, types can also be obtained from an inference chain: if one of the type in the chain is unsound, the rest of the chain is also considered to be unsound. Finally, the Soot framework translates the abstract syntax tree into a typed three-address intermediate representation.

Modes of execution. There are three main parameters that can be adjusted when using PPA.

The first parameter concerns the input of the analysis. Depending on the availability of source files, PPA can be performed on one Java source file at a time or on a set of source files. In the latter case, the worklist containing the inferred type facts is *shared* among all source files and the type inference and the method binding passes are only executed once the seed pass has been performed on each source file. This enables the sharing of inferred type facts which can lead to more precise inference, but it can also propagate imprecise type facts (see Appendix B for an example).

The second parameter allows the user to disable type inference, effectively preventing the execution of the type inference and the method binding pass. When type inference is disabled, all unknown expressions are assigned to the `unknown` type. PPA still performs type member inference and uses our heuristics to resolve ambiguous syntax constructs because those are needed to build the AST.

Finally, the third parameter enables the user to disable the method binding pass, preventing the selection of arbitrary method bindings.

In Chapter 5 we use these parameters to examine the effectiveness of our approach in different scenarios and to measure the added benefits of enabling the type inference and method binding passes.

Termination. Our algorithm is ensured to always terminate. First, the number of AST nodes and type facts in a given program is finite, so the first pass is always sure to complete. The second pass also always completes because the number of times a type fact related to a particular expression can be inferred is finite. As explained in Section 4.2.4, we only infer a new type fact related to an expression if (1) it converges or (2) it conflicts with a previous type fact and is safer than a previous type fact. The number of converging type facts that we can infer on an expression is bounded by the depth of the type hierarchy and the number of conflicting type facts that we can infer on a particular expression is bounded by 4 (from unknown to full). Finally, in the third pass, we select the binding of ambiguous method calls, which are of finite number.

Complexity. To analyze the time complexity of our algorithm, we consider each pass individually. The complexity of the algorithm is bounded by n, the number of AST nodes in all source files, f_a, the number of type facts in all source files, r, the maximum number of nodes referring to a type fact in all source files, f_n, the maximum number of type facts that can be inferred on a node (typically the maximum number of parameters in a method call), m, the maximum number of ambiguous method calls, k, the constant time required to perform operations on a node such as disambiguation, modification or method call binding selection, and h, the maximum depth of the program's type hierarchy. We consider that the selection of a method binding takes a constant time because we always select the first binding. We can express the complexity of each pass with the following formulas:

$$\textbf{Seed pass} = nf_nk = O(nf_n)$$

$$\textbf{Type inference pass} = h \times f_arf_nk = O(hf_arf_n)$$

$$\textbf{M. binding pass} = mf_nk + h \times f_arf_nk = O(mf_n + hf_arf_n)$$

Although the cost to process the worklist is potentially high ($O(hf_af_nr)$), several factors reduce the time complexity in practice. We found during our evaluation of partial program analysis that $h < 4$ because most type facts related to the same expression were conflicting, and when they were converging, they always converged fast. The number of nodes impacted by a type fact, r, was also small: it was on average equal to 1.57 and always below 213. Finally, the number of inferred type facts per node was low: $f_n < 4$.

Chapter 5
Evaluating Partial Program Analysis

To validate the performance of partial program analysis under various circumstances, we performed an empirical study on four open-source systems. We were mostly interested in evaluating the following criteria:

1. The quality of the results obtained by PPA as measured by the number of correct and erroneous type facts.
2. The impact of the input (i.e., size of the partial program) on PPA precision.
3. The contribution of the various inference strategies in producing more precise results.

5.1 Experimental Design

We performed partial program analysis on every single class, including anonymous and internal classes, of four open-source systems. Table 5.1 shows the target systems along with their version, the number of classes and the number of source lines of code (SLOC) they have. We selected these systems because they are relatively complex, their version history was available, they could be compiled with Java 1.4, and because the programming language and software engineering communities frequently analyze those programs. The first three systems, Lucene [10], JFreeChart [8], and Jython [9] are part of the DaCapo benchmark suite [45]. Because the three first systems are self-contained, i.e., they do not require any other library outside the Java standard library, we selected a fourth system, Spring [12], which depends on 90 external jar files to compile. This was to increase the external validity of our evaluation by analyzing various kinds of Java programs.

In general, we wanted to assess the quality of the results obtained by partial program analysis against the results obtained when the complete program is available. To perform this comparison, we executed PPA on each class separately, without any other classes in the target system, and obtained an intermediate representation of each class in the form of a Jimple file. We also transformed every class of the complete target system into Jimple. We thus obtained two Jimple representations for each class, one from PPA and the other from the complete system, that we could compare. The following example shows two Jimple statements, the first one from the partial program, the second one from the complete program.

```
i = virtualinvoke $r1.<p-unknown.unknown: int length()>();
i = virtualinvoke $r1.<java.lang.String: int length()>();
```

When comparing the type facts referenced by two statements, there are four possible outcomes:

correct The two types are the same. For example, the return type of the method `length` is `int` in both statements.

unknown The type of the partial program is `unknown`, which means that PPA could not infer anything about this type. This is the case of the method's target in the first statement.

hierarchy correct The type in the partial program is a supertype or a subtype, depending on the type direction in the type fact, of the type in the complete program. For example, if $dt(\texttt{\$r1}) = \texttt{CharSequence}$ at line 1 and $dt(\texttt{\$r1}) = \texttt{String}$ at line 2, we say that the two types are *hierarchy correct*.

erroneous All other cases. Erroneous types can be inferred when we combine conflicting type facts or when we use certain syntax heuristics.

We only compared the short name of the types. The ability to infer the fully qualified name of a type solely depends on the project coding convention: if a project such as Lucene or Jython allows the usage of wildcard import statements, most inferred types will have a `p-unknown` package. Because a short name, given the context in which it is used, is often

Target	**Version**	**# Classes**	**SLOC**
Lucene	2.2.0	371	23937
JFreeChart	1.0.9	561	81538
Jython	2.2.1	995	83763
Spring	2.5.1	2011	98938

Table 5.1: Target systems

	Outcome	**Lucene**	**JFreeChart**	**Jython**	**Spring**
baseline	% correct	89.20	89.23	81.20	87.16
	% unknown	8.23	9.02	13.88	8.01
	% h. correct	0.29	1.12	1.91	1.12
	% erroneous	2.29	0.63	3.01	3.71
inf.	% correct	93.48	94.12	87.94	90.78
no bind.	% unknown	3.52	3.89	6.63	4.30
	% h. correct	0.34	1.16	2.36	1.20
	% erroneous	2.67	0.83	3.07	3.71
inf.	% correct	93.80	94.40	88.22	90.97
bind.	% unknown	2.46	3.56	6.21	4.07
	% h. correct	0.38	1.19	2.44	1.24
	% erroneous	2.71	0.85	3.13	3.72
	Total Facts	87706	250155	312907	325641

Table 5.2: Partial program analysis results

sufficient to uniquely identify a type, we preferred to classify as correct, types with an unknown package that matched the short name of a real type.

Finally, we chose to compare the Jimple intermediate representation of the partial program and the complete program because (1) this is the typical representation used to perform data flow analysis, (2) this provides a reasonable estimate of the results we would obtain if we performed the comparison at the AST or bytecode level (the transformation from Jimple to AST or bytecode is more straightforward than the transformation from AST to bytecode), and (3) there are fewer statement types in Jimple than node types in a Polyglot AST which makes the comparison easier and more robust.

5.2 Quality

We executed our implementation of partial program analysis on one class at a time without its dependencies, for all classes in our four target systems. Table 5.2 shows the results of PPA. There are three main sections in the table corresponding to the three configurations we used to execute PPA: (1) our baseline configuration (type inference and method binding disabled) [1], (2) type inference enabled and method binding disabled, and (3) type inference and method binding enabled. For each of the configurations, the percentage of type facts in the Jimple IR that correspond to one of the four possible comparison outcomes is indicated

[1]Since our tool must build a properly constructed Polyglot AST in order to continue processing an entire class file, this is the minimal configuration we can enable. It uses the declared types that are available inside the class under analysis, plus syntax heuristics (Section 4.3) and it infers missing type members (Section 4.2.3).

below the target system. For example, with the baseline configuration in Lucene, 89.20% of the type facts recovered by PPA were correct and 2.29% of the type facts were erroneous. The last line reports the total number of type facts in each complete system. For example, there were 250155 type facts in JFreeChart.

The first observation we can make is that our baseline configuration recovered most of the type facts in the partial programs (up to 89.23% in JFreeChart). Thus, combining the declared types available for the class under analysis with the syntax heuristics and type member inference works reasonably well. However, this baseline configuration can be improved upon, and the results indicate that type inference provides most of the remaining improvement. In the best case (Jython), type inference enabled the recovery of 6.7% of correct type facts. Forcing method bindings had a much smaller impact on the precision of the results because in the best case (Lucene), it recovered only 0.32% of correct type facts.

Syntax heuristics were the largest contributor of erroneous type facts. In the worst case (Spring), 3.71% of the inferred type facts were erroneous because of the syntax heuristics. These errors are effectively unavoidable because most of the syntax construct ambiguities represent undecidable problems. Still, as future work, we could validate the assumptions behind our syntax heuristics on more systems to ensure that they are representative and minimize the potential for erroneous type facts.

The number of unknown type facts decreased significantly when we enabled type inference and method binding. Table 5.3 shows the distribution of the unknown types once we enabled these two parameters. On average, the type inference and method binding passes correctly recovered 52% of the types that were previously unknown. On average, only 1% of the unknown types were erroneously inferred by these two passes. This provides evidence that performing type inference and method binding is desirable.

Hierarchy correct type facts only accounted for a small portion of the total type facts. This suggests that even if our heuristics and type inference strategies are theoretically imprecise (i.e., we often infer that an the type of an expression is subtype or a supertype of a type `T`), in practice, they often recover the exact type. The small number of hierarchy correct and erroneous type facts introduced by type inference and method binding also indicates that conflicting type facts do not represent a serious threat to the precision of the results.

Outcome	**Lucene**	**JFreeChart**	**Jython**	**Spring**
% correct	55.92	57.33	50.61	47.62
% h. correct	1.08	0.79	3.78	1.46
% erroneous	5.17	2.37	0.86	0.15
% unknown	37.83	39.50	44.75	50.77

Table 5.3: Unknown types distribution after the type inference and method binding passes

Target	**From**	**To**	**# Revisions**	**# Classes**
Lucene	149000	616506	1017	4800
JFreeChart	1	712	185	924
Jython	1	4011	1267	20609
Spring	2003-08-01	2008-02-24	7299	31101

Table 5.4: Target systems versions

5.3 Analysis Input

Partial program analysis can be performed on one class or on a set of classes. Having access to multiple type declarations can potentially improve the precision of the analysis. Because the accessibility to source files may vary from one technique to the other, we devised three scenarios that are representative of current software engineering techniques. The first scenario assumes that the user of partial program analysis only has access to one class: this is the same scenario as the previous section. The second scenario assumes that the user has access to one class and all classes that are directly referenced by this class. Approaches that mine code from web repositories would typically have access to a subset of the direct dependencies. The third scenario assumes that the user mines version histories and has thus access to all files that were modified in the same change set.

To evaluate the second scenario, we took each class in a target system and computed their direct dependencies using the complete target system. For each class, we provided the source files containing the class and the direct dependencies to our tool, but we did not provide any dependencies that were contained in a jar file. We then compared the inferred type facts from the class in the partial program with the type facts from the class in the complete program, but we did not compare the type facts inferred in the direct dependencies.

For the third scenario, we first retrieved the change sets, i.e., files that were committed together, from the Subversion repositories of Lucene, JFreeChart and Jython and we recovered the change sets from the CVS repository of Spring using a standard change set inference technique [51]. For each change set, we computed the list of classes that (1) were changed or modified, and (2) still existed in the current version of the program. We

		Outcome	**Lucene**	**JFreeChart**	**Jython**	**Spring**
single	*baseline*	% correct	89.20	89.23	81.20	87.16
		% erroneous	2.29	0.63	3.01	3.71
	inf.	% correct	93.80	94.40	88.22	90.97
		% erroneous	2.70	0.85	3.13	3.72
dep	*baseline*	% correct	99.30	95.97	98.13	93.31
		% erroneous	0.33	0.23	0.27	2.64
	inf.	% correct	99.56	98.39	98.92	95.00
		% erroneous	0.26	0.29	0.29	2.67
cs	*baseline*	% correct	89.52	89.54	86.92	86.13
		% erroneous	2.00	0.56	2.97	4.91
	inf.	% correct	93.88	94.82	90.68	90.34
		% erroneous	2.62	0.72	3.24	4.91

Table 5.5: Partial program analysis inputs

obtained a collection of class sets taken from the *current version* of the program that we provided as input to our tool. For each change set, we compared the type facts inferred in all classes in the change set with the type facts from the same classes in the complete program. Table 5.4 shows the range of versions we mined for each target system, the number of change sets (revisions) containing Java source files related to the target system and the total number of classes that we analyzed.

For each of the three scenarios, we executed PPA with the three configurations used in Section 5.2: (1) baseline configuration, (2) type inference enabled and method binding disabled, and (3) type inference and method binding enabled. Table 5.5 shows the results of our analysis. The three main sections represent the three scenarios we evaluated: single class (*single*), one class with all direct dependencies (*dep*), and all classes in the same change set (*cs*). For each section, we report the percentage of correct and erroneous type facts for the first (*baseline*) and third configurations (*inf.*) of PPA.

In all cases, we omitted the results of the second configuration because there was no significant difference between it and the third configuration. The results for the first configuration are the same as Table 5.2.

Including the direct dependencies greatly increased the percentage of correct type facts PPA could infer. This high precision actually left little room for improvement from type inference. Indeed, on average, 96% of the type facts were correct in the baseline configuration when including all direct dependencies as opposed to 86% correct type facts in our baseline configuration of single class analysis. We obtained fewer correct type facts when analyzing Spring because a subset of the direct dependencies was contained in jar

	Strategy	**Lucene**	**JFreeChart**	**Jython**	**Spring**
single	% Assign.	45.59	61.06	36.73	38.37
inf.	% Return	13.41	6.10	52.93	31.12
no bind.	% Method	0.66	0.72	0.39	0.35
	% Condition	8.73	5.10	1.38	16.50
	% Binary	22.01	17.64	6.06	7.12
	% Unary	3.48	8.75	0.91	3.55
	Total facts	4571	6700	41521	14462
single	% Assign.	41.95	57.31	29.52	36.29
inf.	% Return	12.25	5.73	42.53	29.42
bind.	% Method	8.55	6.64	19.93	5.72
	% Condition	7.97	4.79	1.11	15.60
	% Binary	20.28	16.73	4.87	6.77
	% Unary	3.18	8.21	0.73	3.36
	Total facts	5006	7138	51675	15297

Table 5.6: Inference strategies results

files which were not supplied to the compiler. Overall, these results suggest that, when possible, retrieving a subset of the dependencies might be highly beneficial since adding the dependencies had a greater impact than type inference. Finally, because certain members were declared in an ancestor and were thus not accessible, we still inferred erroneous and unknown type facts.

Analyzing all classes in a change set did not significantly improve the precision of our results. On average, only 34% of the direct dependencies of a class were in the same change set. Usually, even if two related classes are in the same change set, the improvement might be minimal if one class only accesses a few members in the other class. Still, further analysis of the change sets results are required. For example, some files are changed more often than others: if the files that are frequently changed are also the ones that gives the best (or the worst) results when analyzed by our tool, the results will be highly biased toward these files. This could explain the decrease of precision for Spring (90.97% for single class analysis versus 90.34% for change set analysis).

5.4 Inference Strategies

Since we showed that type inference was beneficial, we were interested in analyzing the contribution of each inference strategy we devised and presented in Section 4.2.1. This information can be used to determine which inference strategies are worth implementing if PPA needs to be implemented in an existing technique. For each type fact that was processed in the worklist (see Figure 4.8), we recorded the inference strategy that caused its insertion in the worklist which provided a good estimation of the contribution of each strategy.

Table 5.6 shows the percentage of type facts that each of the six most popular inference strategies generated: the other inference strategies had a negligible contribution. Because the ordering and the proportion of the inference strategies were similar for each input scenario, we only report the results when we analyzed one class at a time. The upper part of the table shows the proportion of each inference strategy when performing type inference without method binding and the lower part shows the results when performing type inference with method binding. The last line in each part indicates the number of type facts that were processed in the worklist. For example, when performing type inference and method binding on Lucene, the assignment inference strategy generated 41.95% of the type facts.

The assignment inference strategy was the largest contributor of type facts in all target systems except Jython. The return and binary inference strategies came second in two target systems each. Those three strategies contributed to 90% of the type facts when disabling method binding and 76% when enabling method binding. Because a strategy can also trigger the use of another strategy (e.g., we find the type of a field using the assignment strategy and then we use the method binding strategy because a method uses this field as a parameter), we were interested in the inference chains produced by our approach. We found the average inference chain length to be 1.02, meaning that generally, an inference strategy *does not* trigger the use of another strategy. We also found in a manual investigation of the inference chains for each inference strategy that there is no significant correlation between any two inference strategies.

5.5 Threats to Validity

The external validity of this study is limited by the fact that we only studied four programs. Because three of these programs are self-contained, i.e., they do not require external libraries, we studied the Spring Framework which requires 90 jar files to increase the scope of our evaluation. Our four target systems have a large number of lines of code and their purposes are different enough to be representative of many Java programs. The fact that the results were also relatively stable across all four programs suggests that results obtained with different systems would be similar.

Our unit of measurement to evaluate the precision of PPA was the number of correct type facts in the Jimple intermediate representation. This unit is a good indicator for techniques that use PPA to retrieve static type information (e.g., SemDiff), but it is not sufficient to evaluate the usefulness of our approach for client static analyses such as call graph generation and points-to analysis that use the IR produced by PPA. These client static analyses typically require a higher precision than the software engineering tools that currently use PPA. Researchers in both our groups and others will be able to do these sorts of experiments now that PPA is fully implemented and publicly available.

The parser that PPA uses takes as input well-formed Java source files. Hence, it was not possible to evaluate our approach against code snippets even if PPA does not require complete source files. Although it is possible that analyzing only code snippets could decrease the precision of our approach, we found during early experimentation that type inference did not often cross the method boundaries, so the performance of PPA should not be dramatically impacted.

Finally, when we analyzed the change sets, we only used the latest version of the classes for each change set as opposed to using the version of these classes at the time of the change set. We expect the results of our analysis to be representative because the set of direct dependencies should be relatively stable during the lifetime of a class.

Chapter 6
Related Work

Supporting framework evolution and analyzing partial programs are active research areas and various techniques have been proposed with these goals.

6.1 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 [19] 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! [31] 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

Method	**Program Element Characteristics**	**Versions**
Origin Analysis [29]	name similarity, code metrics, calls	two complete versions selected manually
UMLDiff [49]	name similarity, code relationships	full versions between two versions
M. Kim et al. [34]	name similarity	two complete versions selected manually
S. Kim et al. [35]	name similarity, code metrics, calls, textual similarity	two complete versions selected manually
Dig et al. [23]	syntactic similarity (Shingle), code relationships	two complete versions selected manually
Weissgerber et al. [48]	structural and code clone differences	all change sets between two versions
SemDiff	structural and outgoing call differences	all change sets between any versions

Table 6.1: Comparison of change detection approaches

be *replayed* in a client program to repair broken method calls. Explicit support for saving and replaying refactorings was introduced in Eclipse 3.3 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 the second technique as it is not limited to explicitly applied refactorings. Moreover, author of the client program does not need to search through a list of refactorings to find the one that is relevant to a broken method.

6.2 Change Detection

Most refactoring detection techniques refer to Origin Analysis [29] as the basis of their approach. Origin Analysis is a semi-automated technique that aims at tracing back to the source of a program element in a previous version of the program to detect changes such as splitting and merging. Similarly, if a refactoring 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, refactoring detection techniques use a variety of strategies based on program element characteristics (e.g., name, location, outgoing and incoming references, textual similarity) to assess the similarity of two elements across two versions. If the similarity of the program elements is beyond a certain threshold, those techniques conclude that one is a refactored variant of the other.

More precisely, techniques such as UMLDiff [49] analyze code relationship similarity (e.g., calls, hierarchy, accesses, etc.) between complete versions of a program to detect high level changes such as refactorings. RefactoringCrawler [23] is a similar but more lightweight alternative to this approach that also analyzes some code relationships and uses Shingle, a custom syntactic similarity analysis, on two complete versions of a program to detect refactorings. A later implementation of Origin Analysis [35] fully automated the approach for Java, but also reduced the number of change types that the technique detected to consider program elements renaming and move. Mining software repositories [52] 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 [48]. Other researchers also used repository mining to predict the likelihood of a class to be refactored in the next two months [41] or to classify fine-grained changes that occurred inside method bodies [27]. Finally, M. Kim et al. proposed a technique for detecting change patterns by leveraging the similarity of program element names [34]. Table 6.1

shows a comparison between the characteristics that are used to assess the similarity of program elements and the types of programs these various approaches require...

Typically, techniques based on Origin Analysis will not be able to detect some or all of the following three kinds of changes: first, changes that disfigure a significant part of the program structure such as a rearchitecture will throw out most refactoring detection techniques because the level of change in a program element's characteristics will be too high to safely conclude that two program elements are similar. Second, small changes performed on small methods will also hinder the accuracy of most refactoring detection techniques: for instance, if one line of code is changed in a two-lines method, some technique will conclude that 50% of the method changed and that the two are not similar. Third, changes involving code external to the program under study will be missed. For example, if a feature that was orginally in the program is now imported from an external program (e.g., a library offers a better version of the feature), techniques based on origin analysis will not be able to capture the change because the destination is not in the analyzed program.

As shown with the study presented in Chapter 3, 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. Still, techniques based on origin analysis can give a certain degree of confidence about the origin of an element throughout the evolution of a program. This confidence is required by activities demanding detailed traceability information, such as bug tracking.

Finally, another approach using associative data mining on framework usage changes (such as method call changes) also outperformed an origin analysis-like technique and could even recommend more change patterns than SemDiff (e.g., a field access should be replaced by a method call) [44]. As opposed to SemDiff, this approach only compares two full versions, which can introduce more false-positives in the results and it also requires stable callers.

6.3 Framework Usage

Another family of approaches could potentially be used to support framework evolution. Strathcona [32] and FrUIT [18] 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 those 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 becomes available when a new framework version is released. On the other hand, SemDiff only uses the usage examples inside the framework itself to produce results.

6.4 Static Analysis of Partial Programs

Static analysis tools typically assume that a complete and correct representation of the program is available. Our work is different in that we assume that only part of the program is available. To the best of our knowledge, we know no other research project, except the prototype used in PARSEWeb [46], that performs type inference and resolves syntactic ambiguities with such constraints. Unfortunately, it is not possible to provide a full evaluation of the PARSEWeb's prototype because it is not publicly available and the paper only describes two inference strategies the prototype used. Those strategies are equivalent to our return and method binding inference strategies.

Still, there are several techniques that deal with the parsing of incomplete programs. Two such examples include *fuzzy parsers* [37] which extract high level structures out of incomplete or syntactically incorrect programs, and *island grammars* [39] which parse snippets of code into islands (recognizable constructs of interest) and water (remaining parts). Knapen et. al. presented an approach for parsing C++ programs when missing some header files [36]. Their motivation was quite similar to ours, since they wanted to deal with situations where not all the code was available. They developed various semantic tests and heuristics, similar to the syntax heuristics we used and described in Section 4.3, to determine the nature of an ambiguous syntactic constructs. As opposed to PPA, this C++ parser does not try to infer the declared type of an AST node (like the inference strategies presented in Section 4.2) and it creates an unknown node when it cannot soundly determine the nature of an expression.

The programming system generator (PSG) enabled the creation of development environments that could handle the parsing and analysis of incomplete programs [15]. For example, the environment could list the possible types of an undeclared variable according to its context. The computation of the possible types was encoded into a grammar written by the PSG users: the inference rules had thus to be manually defined and only simple inheritance schemes (e.g., Pascal) were supported. Another difference is that the environment generated by PSG was only *suggesting* possible types: it was not making any definitive choice like PPA.

Finally, modern Integrated Development Environments (IDE) often include tools to execute or analyze snippets of code. For example, the Java parser [4] in Eclipse is able to generate an Abstract Syntax Tree for incomplete programs, but it does not try to resolve syntax ambiguities and it does not provide any typing information when the declaration of a type is missing. The Scrapbook editor [6] tries to execute any snippet of code even if it is not included in a Java class. Again, this tool reports an error if it encounters an undeclared type.

In terms of inferring declared types for Java, Gagnon et. al. solved a related problem of finding declared types of local variables when starting from Java bytecode [28]. Although their approach also used type constraints to assign declared types, their setting is quite different since they have access to the complete program and type hierarchy, and there are no syntactic ambiguities in bytecode.

Other work, less directly related, includes static analyses techniques that aim to analyze only part of a program. These are often designed for software engineering applications, where it is too expensive to analyze the whole program. These techniques use an intermediate representation generated from the *complete program* where all type declarations are accessible. Examples of these techniques include *partial data flow analysis* [26, 30] which uses a demand-driven approach to analyze only the relevant part of a whole program and *fragment analysis* [42, 43] which does a full analysis on a given fragment of the program, using summary information for the remainder.

Chapter 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. An historical study of the Eclipse JDT framework and three of its client programs showed that our technique can detect non-trivial changes, and that it succeeded in providing correct recommendations for 89% of the cases of missing functionality between Eclipse release 3.1 and 3.3. As opposed to previous work on refactoring detection techniques, our approach can recommend methods located outside of the framework when a functionality has been imported from external libraries.

We also presented *Partial Program Analysis*, a technique that builds a typed abstract syntax tree and a typed three-address intermediate representation that software engineering tools can use to get more precise type information than what syntactic analysis traditionally provides. We covered the two main challenges when analyzing partial programs in Java, the ambiguous language constructs and the determination of declared types, and we proposed type inference strategies and heuristics to solve these problems. We performed an empirical study on four open source programs and found that, on average, Partial Program Analysis could uncover 91.2% of correct type facts when analyzing one class at a time and only produced 2.7% of erroneous type facts. This high precision suggests that partial program analysis is a viable approach to enable useful static analysis on incomplete Java programs.

We conclude that analyzing outgoing call differences in version histories is an efficient and robust technique to track a framework's evolution and repair dependent client programs.

Appendix A
Type Inference Strategies

Table A.1 presents the inference rules that PPA applies in a partial program. The first column indicates the inference rule name, the second column shows the AST node templates to which the inference rule is applicable. The third column lists the type facts that are inferred.

In an AST node template, arbitrary Java expressions are represented by `w`, `x`, `y`, and `z`, and types are represented by t, $t1$, and $t2$. The operator *baseType* returns the base type of an array (e.g., *baseType*(`int[]`) = `int`). The operator *safest* returns the safest type, or the first type if both are as safe, as defined in Section 4.2.4.

Although type facts are generated for each AST node for which we do not have access to the declared type, the type facts are only added to the worklist if they meet the safety criterion presented in Section 4.2.4. Finally, for the sake of brevity, we only present the relevant subset of the unary and binary operators.

Inference Strategies	**Code**	**Generated Type Fact**

Assignment	x = y;	{`x`, *dt*(`x`), :> *dt*(`y`)}
		{`y`, *dt*(`y`), <: *dt*(`x`)}

Return	*t* m1() {
	...
	return y;
	}
		{`y`, *dt*(`y`), <: *t*}

Method Binding	**void** m1() {
	...
	x = m2(y);
	}
	
	*t*1 m2(*t*2 param1) {
	...
	}
		{`x`, *dt*(`x`), <: *t*1}
		{`y`, *dt*(`y`), <: *t*2}
		We assume that m2 declared after m1 is the binding selected by PPA.

Condition	**for** (y; x; z)
	while (x)
	if (x)
	x ? y : z
		{`x`, dt(`x`), `boolean`}

Unary	x++
	!y
		{`x`, dt(`x`), ∼ `int`}
		{`y`, dt(`y`), `boolean`}

Binary	x + y
	w & z
	w | z
		{`x`, dt(`x`), ∼ dt(`y`)}
		{`y`, dt(`y`), ∼ dt(`x`)}
		{`w`, dt(`w`), ∼ dt(`z`)}
		{`z`, dt(`z`), ∼ dt(`w`)}
		or
		{`w`, dt(`w`), `boolean`}
		{`z`, dt(`z`), `boolean`}
		if the expected return type of `&` or `|` is a boolean.

Array	x[y]	{`y`, dt(`y`), <: `int` }
		{`x`, dt(`x`), dt(`x[y]`)[] }
		{`x[y]`, dt(`x[y]`), $baseType$(`x`) }

Switch	**switch**(x)	{`x`, dt(`x`), <: `int`}

Conditional	x ? y : z;	{`y`, dt(`y`), ∼ `z`}
		{`z`, dt(`z`), ∼ `y`}
		{`x ? y : z`, dt(`x ? y : z`), $safest$(dt(`y`), dt(`z`))}
		or
		{`y`, dt(`y`), <: t}
		{`z`, dt(`z`), <: t}
		if the expected type t of the conditional is known.

Table A.1: Type Inference Rules

Appendix B
Syntax Imprecision

When analyzing multiple source files, errors and imprecisions can be propagated. For example, suppose that the classes `A` and `B` presented in Figure B.1 are provided as input to PPA. The field `f2` is shared by the two classes. In class A, PPA finds that $dt(\texttt{f2}) \sim \texttt{String}$ because $dt(\texttt{f2})$ <: $dt(\texttt{f1})$:> `String` (inference chain produced at lines 4 and 5). PPA then concludes that at line 12, the method `C.m1(String)` is called. Unfortunately, this is not true since $dt(\texttt{f2})$ could be of any reference type (e.g., `List`) and the parameter of method `m1` could end up being “far” from the type `String`.

If the two files had been analyzed separately, PPA would have concluded that at line 12, the method `C.m1(Unknown)` was called. This is still imprecise, but not as misleading as the previous inference.

```
 1  // File A.java
 2  class A {
 3    void main() {
 4      C.f1 = "hello";
 5      C.f1 = C.f2;
 6    }
 7  }
 8
 9  // File B.java
10  class B {
11    void main() {
12      C.m1(C.f2);
13    }
14  }
```

Figure B.1: A partial program

Bibliography

[1] AspectJ Development Tool. http://www.eclipse.org/ajdt/, last accessed on June 27th 2008.

[2] CVS. http://www.nongnu.org/cvs/, last accessed on June 27th 2008.

[3] Eclipse. http://www.eclipse.org/, last accessed on June 27th 2008.

[4] Eclipse JDT parser. http://www.eclipse.org/jdt/core/index.php, last accessed on June 27th 2008.

[5] Eclipse Modeling Framework. http://www.eclipse.org/modeling/emf/, last accessed on June 27th 2008.

[6] Eclipse Scrapbook Editor. http://www.eclipsezone.com/eclipse/forums/t61137.html, last accessed on June 27th 2008.

[7] JBossIDE. http://www.jboss.org/tools/, last accessed on June 27th 2008.

[8] JFreeChart. http://www.object-refinery.com/jfreechart/, last accessed on June 27th 2008.

[9] Jython. http://www.jython.org/, last accessed on June 27th 2008.

[10] Lucene. http://lucene.apache.org/, last accessed on June 27th 2008.

[11] PostgreSQL. http://www.postgresql.org/, last accessed on June 27th 2008.

[12] The Spring Framework. http://www.springframework.org/, last accessed on June 27th 2008.

[13] Subversion. http://subversion.tigris.org/, last accessed on June 27th 2008.

[14] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 805–806, 2007.

[15] Rolf Bahlke and Gregor Snelting. The PSG system: from formal language definitions to interactive programming environments. *ACM Transactions Programming Languages and Systems*, 8(4):547–576, 1986.

[16] Nicolas Bettenburg, Rahul Premraj, and Thomas Zimmermann. Extracting structural information from bug reports. In *Proceedings of the 2008 international workshop on Mining software repositories*, pages 27–30, 2008.

[17] Jean-Sébastien Boulanger and Martin P. Robillard. Managing concern interfaces. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 14–23, 2006.

[18] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. Fruit: Ide support for framework understanding. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 55–59, 2006.

[19] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proceedings of the 1996 IEEE International Conference on Software Maintenance*, page 359, 1996.

[20] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *To appear in Proceedings of the 23rd ACM SIGPLAN conference on Object Oriented Programming Systems and Applications*, page 10 pages, 2008.

[21] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *Proceedings of the 30th ACM/IEEE international conference on Software engineering*, pages 481–490, 2008.

[22] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 185–194, 2007.

[23] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 404–428, 2006.

[24] Danny Dig and Ralph Johnson. How do APIs evolve&quest; A story of refactoring: Research articles. *Journal of Software Maintenance and Evolution*, 18(2):83–107, 2006.

[25] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, pages 427–436, 2007.

[26] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions Programming Languages and Systems*, 19(6):992–1030, 1997.

[27] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[28] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, pages 199–219, 2000.

[29] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineerings*, 31(2):166–181, 2005.

[30] Rajiv Gupta and Mary Lou Soffa. A framework for partial data flow analysis. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 4–13, 1994.

[31] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th ACM/IEEE international conference on Software engineering*, pages 274–283, 2005.

[32] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[33] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, 2006.

[34] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, pages 333–343, 2007.

[35] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, 2005.

[36] Gregory Knapen, Bruno Laguë, Michel Dagenais, and Ettore Merlo. Parsing C++ Despite Missing Declarations. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension*, page 114, 1999.

[37] Rainer Koppler. A systematic approach to fuzzy parsing. *Software – Practice and Experience*, 27(6):637–649, 1997.

[38] Bennett P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[39] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, page 13, 2001.

[40] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 138–152, 2003.

[41] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 354–363, 2007.

[42] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE Transactions on Software Engineering*, 30(6):372–387, 2004.

[43] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 235–252, 1999.

[44] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th ACM/IEEE international conference on Software engineering*, pages 471–480, 2008.

[45] Stephen M. Blackburn et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

[46] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 204–213, 2007.

[47] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13, 1999.

[48] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, 2006.

[49] Zhenchang Xing and Eleni Stroulia. Understanding the evolution and co-evolution of classes in object-oriented systems. *International Journal of Software Engineering and Knowledge Engineering*, 16(1):23–51, 2006.

[50] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology*, 15(2):195–226, 2006.

[51] Thomas Zimmermann and Peter Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, 2004.

[52] Thomas Zimmermann, Andreas Zeller, Peter Weißgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.