# Object Histories in Java

by

Aakarsh Nair

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2010

© Aakarsh Nair 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

## Abstract

Developers are often faced with the task of implementing new features or diagnosing problems in large software systems. Convoluted control and data flows in large object-oriented software systems, however, make even simple tasks extremely difficult, time-consuming, and frustrating. Specifically, Java programs manipulate objects by adding and removing them from collections and by putting and getting them from other objects' fields. Complex object histories hinder program understanding by forcing software maintainers to track the provenance of objects through their past histories when diagnosing software faults.

In this thesis, we present a novel approach which answers queries about the evolution of objects throughout their lifetime in a program. On-demand answers to object history queries aids the maintenance of large software systems by allowing developers to pinpoint relevant details quickly.

We describe an event-based, flow-insensitive, interprocedural program analysis technique for computing object histories and answering history queries. Our analysis technique identifies all relevant events affecting an object and uses pointer analysis to filter out irrelevant events. It uses prior knowledge of the meanings of methods in the Java collection classes to improve the quality of the histories.

We present the details of our technique and experimental results that highlight the utility of object histories in common programming tasks.

**Acknowledgments**

I would like to express my sincere thanks and gratitude to my advisor Dr. Patrick Lam. He is truly exceptional and I am thankful I had the opportunity to learn from him. His breadth and depth of knowledge, instinct, guidance, and motivation were critical to the completion of my research.

I would also like to thank my thesis committee members Dr Lhoták and Dr. Tan for taking the time and effort to read my thesis despite their extremely busy schedules. I also appreciate their insightful comments and suggestions.

I am also thankful to my colleague Jon Eyolfson, who helped me with the many problems I faced in completion of this degree. I also appreciate Xavier Nombassi and Hang Chu for taking the time to discuss my obstacles and brainstorm solutions.

## Dedication

This thesis is dedicated to my parents. I could not have completed it without their unconditional support. I truly appreciate their encouragement, guidance, and the sacrifices they made that enabled me to pursue my studies.

# Contents

**List of Tables** ix

**List of Figures** xi

**1 Introduction** 1

1.1 Approach . . . . . . . . . . 2

1.2 Results . . . . . . . . . . 4

1.3 Limitations . . . . . . . . . . 4

1.4 Thesis Contributions . . . . . . . . . . 5

1.5 Thesis Organization . . . . . . . . . . 6

**2 Object Histories** 7

2.1 Motivating Example . . . . . . . . . . 8

2.2 Preprocessing Phase . . . . . . . . . . 10

2.2.1 Intraprocedural History Analysis . . . . . . . . . . 10

2.2.2 Method Summaries . . . . . . . . . . 12

2.2.3 Aggregating Field Events . . . . . . . . . . 14

2.2.4 Handling Java Collections . . . . . . . . . . 17

2.3 Answering Queries . . . . . . . . . . 20

2.3.1 Comprehensive Example . . . . . . . . . . 21

2.4 Filtering . . . . . . . . . . 25

2.4.1 Filtering Context . . . . . . . . . . 25

**3 Realizing Object Histories** **29**
3.1 Background . . . 29
3.1.1 Static Analysis . . . 30
3.1.2 Pointer Analysis and SPARK . . . 30
3.1.3 Soot . . . 31
3.1.4 Jimple . . . 31
3.2 Preprocessing . . . 33
3.2.1 Intraprocedural History Analysis . . . 33
3.2.2 Method Summaries . . . 37
3.2.3 Field Events . . . 38
3.2.4 The Events Database . . . 40
3.3 Answering Queries . . . 40
3.3.1 Handling Collections . . . 43
3.4 Filtering . . . 46
3.5 Graphical User Interface . . . 48

**4 Experimental Results** **49**
4.1 Case Studies . . . 49
4.1.1 Finding Erroneous State . . . 50
4.1.2 Change Support . . . 53
4.2 Bug Fix . . . 57
4.3 Benchmarks . . . 60
4.4 Object History Size Distributions . . . 61
4.5 Effect of Filtering . . . 62

**5 Related Work** **66**
5.1 Points-to Analysis . . . 66
5.2 Program Slicing . . . 66
5.2.1 Thin Slicing . . . 68
5.3 Program Exploration Approaches . . . 68

**6 Conclusions and Future Work** **70**
6.1 Future Work . . . . . 71
**References** **72**

# List of Tables


2.1 Intraprocedural History Analysis for method in Figure 2.3. . . . . . . . . . 11

2.2 Method Summaries for program in Figure 2.4. . . . . . . . . . . . . . . . . . 13

2.3 Aggregated field events for program in Figure 2.5. . . . . . . . . . . . . . . 16

2.4 Method Summaries for program in Figure 2.6. . . . . . . . . . . . . . . . . . 19

2.5 Method summary for method Main: Object foo(M) shown in Figure 2.7. . 23

2.6 Aggregated field events for `M.list` for program in Figure 2.7. . . . . . . . . 24

3.1 Jimple representation of the origins of variables from Figure 3.3. . . . . . . 35

3.2 Data flow set organization in the intraprocedural history flow analysis. . . 36

3.3 Jimple representation for static and instance field events. . . . . . . . . . . 39

4.1 Benchmark characteristics. . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

4.2 Time required for preprocessing and size of events database. . . . . . . . . 61

4.3 Average Filtered and Unfiltered Object History Sizes. . . . . . . . . . . . . 65

# List of Figures

2.1 Architecture for computing object histories. . . . . . . . . . . . . . . . . . . 7
2.2 Example Java Program. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Example Method. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.4 Example Java program for demonstrating method summaries. . . . . . . . 12
2.5 Example Program with Field Events. . . . . . . . . . . . . . . . . . . . . . . 15
2.6 Example Program highlighting collection events. . . . . . . . . . . . . . . . 18
2.7 Comprehensive example used to illustrate object histories. . . . . . . . . . 22
2.8 Java program highlighting the use of filtering. . . . . . . . . . . . . . . . . 27

3.1 Java method in original form. . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2 Jimple representation for Java method in Figure 3.1. . . . . . . . . . . . . 32
3.3 Example method showing different intrapocedural origins. . . . . . . . . . 34
3.4 Graphical front-end for extracting object histories. . . . . . . . . . . . . . . 48

4.1 Java program containing a bug found by using object histories. . . . . . . . 51
4.2 Object history for `item` in method `checkout()` from Figure 4.1. . . . . . . 52
4.3 Object history for field `item.price` in method `checkout()` from Figure 4.1. 53
4.4 Code Change Example. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.5 Object history for fields `A.a1` and `A.a2` from Figure 4.4. . . . . . . . . . . 56
4.6 Object history for `key` in method `GanttLanguage.getText()`. . . . . . . . . 58
4.7 Object History Size Distribution for Benchmarks. . . . . . . . . . . . . . . . 62
4.8 Filtered vs unfiltered object history sizes for variables assigned from instance fields in jGraphX. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

4.9 Filtered vs unfiltered object history sizes for variables originating from instance method invocations in jGraphX. . . . . . . . . . . . . . . . . . . . . . . 64

# Chapter 1

# Introduction

To implement new features or diagnose problems in large software systems, developers must first understand them. Given the drastic increase in complexity of modern software systems, understanding is often difficult, time-consuming, and frustrating. To perform even simple operations, developers must understand complex call chains and class hierarchies. Understanding large software systems is difficult since these systems typically contain masses of implementation details. Finding a particular implementation detail needed for a task at hand can be extremely challenging, time consuming and frustrating. More specifically, programs often store objects on the heap and later retrieve and use them. Worse yet, Java programs often put objects in the Java collection classes such as `ArrayList` and `HashMap` and then later get objects from them. To successfully understand the program, a developer must be able to trace the origin of any object in the program, including the original creation site and all changes to the object while it is stored on the heap. Currently, however, the only way of doing this is to manually trace an object through the call chain, while accounting for adds and removes from collections all over the program.

This thesis presents a novel approach for extracting the evolution of objects from Java programs as they pass through the heap using static analysis. *Object histories* provide a succinct description of the events an object undergoes throughout the program's execution. Events include instantiation, reads from other objects' fields or collections, writes to object fields or collections, being passed to methods, as well as being returned from a method call. These events incorporate the important changes an object undergoes as it moves through the heap.

By using object histories, a developer can find the origin of any object and changes made to it, anywhere in the program. This approach enables the developer to focus on details relevant to the problem at hand. For example, suppose an object in a program has an incorrect state causing a fault. A developer can simply obtain the object's history to find the buggy statement and rectify the problem quickly and efficiently.

## 1.1 Approach

The basic approach is to use static analysis techniques to build a database of key information necessary to compute object histories. The database supports object history queries to obtain the history of any object in the program. The database contains information about each method that appears in the application as well as information about changes to fields in the program. Individual object histories can be computed on-demand once a developer identifies an object by a local variable name and containing method using a database query.

The querying algorithm essentially works backward, starting with the current method and variable name, and recursively extracts additional events from the database. The algorithm progresses based on the history of the object found so far (e.g. field, parameter, etc) and continues until no further information can be extracted. Pointer analysis is also used during this process to filter out irrelevant events. Once the algorithm terminates, the developer is presented with a trace of the object's provenance through a graphical interface.

The information about each method contains a list of callers of the method, a list of methods invoked within the method body, information about each argument used in the method invocations, and an intraprocedural history analysis of the method. The list of callers is used to traverse the call chain if the object of interest is a parameter or a copy of a parameter to the method. The list of invoked methods and information about arguments are useful for answering object history queries about variables that were passed as arguments to other methods.

The intraprocedural history analysis of the method is a data flow analysis that gathers the history of each object-typed local variable within the method. Variables are classified as formal parameters, reads from fields, return values from a method call, new object instantiations, or exceptions. Given the histories for all variables in the method body, our system can answer queries efficiently. As mentioned earlier, the intraprocedural history analysis for each method enables it to quickly find the immediate history of any object in any method. History information also helps identify the steps required to gather additional history details for the object of interest outside the method body such as traversing the call chain or looking for field writes.

The other required component of our approach is information on fields in the program. All events that happen to a field of a class are captured and stored. A field change includes information on the kind of change (e.g. write, collection add), a reference to the abstract object being mutated (the subject of the change), a reference to the abstract object being added or removed (the indirect object of the change), and any relevant method arguments.

The approach described above is field based: we group events that happen to all fields `f` of a particular class `c`. When querying the changes to field `f` of object `o`, an instance of

class c, we use pointer analysis to filter out changes to all other fields f of known different instances o' of class c. This approach is especially effective in Java program since most field accesses happen on the this object due to setter and getter methods. We also found that this filtering technique provides most of the advantages of a field-sensitive approach, where we would store information about each instance separately.

Suppose we wish to obtain the history for object z in the method main() for the simple Java program below.

```
1 class C {
2  Object f, g;
3
4  public static void main(String[] argv) {
5    C o = new C();
6    C m = new C();
7    Object y = new Object();
8    ....
9    m.f = new Object();
10   o.f = y;
11   ....
12   Object x = o.f;
13   ....
14   p.g = x;
15   ....
16
17   // Get object history for z
18   Object z = p.g;
19
20  }
21 }
```

We compute the history of z in main() as follows. We observe that z is assigned from a read of field g of an object of class C. Our field-based analysis tells us that the field write at statement 14 is related to the read of C.g at statement 18, so the value for z comes from the local variable x. The same local variable x is read from field C.f (line 12), which was written at line 10 from variable y. This branch of the history terminates at the instantiation of y at line 7. We filter out the other write to C.f at line 9 because its base object, m, may not alias the base object o.

Object histories report all of the events that may happen to an object. Histories start from a given program point, and include all related program events (both in the past and the future, for objects that are stored on the heap at some point). In particular, object

histories include field reads and writes as well as collection manipulations involving the object, and reach back to the original instantiation points of the object.

## 1.2 Results

In order to validate the feasibility and utility of object histories in real-world software systems, we present two case studies, a bug fix in a medium-sized, open source Java program, as well as some experimental results.

Specifically, we show the utility of using object histories for finding bugs related to a variable `v` holding an erroneous state. To fix these types of bugs, we can simply extract `v`'s object history to find the event that assigned `v` the erroneous state.

We also show how object histories can be used to better understand code change tasks. For example, if we are interested in changing class `C`, we begin by simply extracting the histories for all fields `f` of `C`. The combined object histories help us to identify 2 key pieces of information: 1) classes that are dependent on `C`, and 2) an estimate of the size of the task. We can identify dependencies on class `A` by simply looking at the program points where values for the fields of `A` originate. Thus, we know that when we change `A`, we must also change dependent classes accordingly. We can get an estimate of the size of the task by counting all the program points that contain events on fields of `A`.

Next, we present a sample bug fix from Gantt Project [2] as an example to demonstrate the usefulness of object histories in real-world software systems. Gantt Project is an open-source project management and scheduling tool that is maintained by an online community of developers. It contains roughly 64,000 lines of code in 482 classes. We show how to fix a typical user interface bug in Gantt Project with minimal understanding of the source code by using object histories.

Finally, we present some quantitative results from studying three open source Java programs. Specifically, we discuss the distribution of object history sizes and the effectiveness of our filtering algorithm.

## 1.3 Limitations

The research on object histories outlined in this thesis has some limitations due to design decisions and available frameworks. This section briefly describes the limitations in the implementation of the object histories.

Our implementation accounts for only methods that appear in the application classes. Aside from Java collection methods, library methods are not accounted for while computing

method summaries. The main reasons for this decision are twofold: 1) in our experience, we found that the Java collection library classes are the most relevant libraries that affect object histories. While other library classes could also have an impact, we believe the effect to be minimal because library classes generally do not manipulate fields of application objects. 2) The computational overhead of including all library classes is significant, so the memory requirements and time for running the algorithm would rise sharply. There is, however, no conceptual barrier to including library classes.

Our approach for computing object histories is context-insensitive. This means that we consider all callers of a method $m$ when extracting the history of a parameter $p$ to method $m$. Context-sensitivity would allow us to eliminate callers that are not included in the execution path leading the program point of the query.

Our approach is also flow-insensitive at the interprocedural level. In particular, we do not account for the order of events that occur in a variable $v$'s object history. We simply report all events on $v$. However, our intraprocedural history analysis is flow-sensitive at the method level, which enables us to omit some irrelevant events within any method $m$.

## 1.4 Thesis Contributions

The major contribution of this thesis is to provide a fast, efficient, and easy to understand solution for computing object histories enabling developers to quickly understand relevant portions of large software systems. This thesis presents a novel approach which answers queries about the evolution of objects throughout their lifetime in a program. These on-demand object histories greatly aid in the maintenance of large software systems by allowing developers to pinpoint relevant details quickly.

This thesis makes the following contributions.

- **Notion of Object Histories:** This thesis presents the idea of object histories. Object histories are collections of relevant events which affect an object throughout its lifetime in a software system. Developers can use the histories of important objects in order to quickly understand key details essential for the task at hand.
- **Filtering Histories:** Object histories may contain a long list of events that may not be relevant. In this thesis, an approach for filtering out irrelevant details (where possible) is also presented. Filtering is accomplished by employing pointer analysis and comparing points to sets for appropriate objects that appear in a given object history.

- **Algorithm for Computing Object Histories:** Computing object histories can be a difficult and time consuming task. This thesis presents a quick and efficient algorithm for computing object histories without the need for building complex interprocedural program dependence graphs. Initially, the algorithm builds a core set of information for the entire program, which can later be queried repeatedly for extracting the object histories for any object that appears in the program.

- **Implementation of Algorithm and User Interface:** The algorithm described in the thesis is implemented in Java along with a graphical user interface. The implementation allows developers to generate the core database and query it order to obtain the history for any object in the program.

- **Experimental Results:** Finally, this thesis contains experimental results from using the implementation on a number of realistic benchmark programs. The open source programs used for experimental results vary in size and complexity, but confirm the applicability and utility of our approach.

## 1.5 Thesis Organization

The remainder of this thesis is organized as follows.

We describe the notion of object histories in Chapter 2. We present each phase in computing object histories by using easy to understand examples.

In Chapter 3, we present the details of computing object histories. We begin the chapter with relevant background information and provide all details (including algorithms) about our implementation of object histories.

In Chapter 4, we present our experimental results. We present two case studies to highlight the usefulness of object histories for finding erroneous state and code change tasks. Next, we demonstrate the utility of object histories for finding a bug in a real-world, medium-sized, open source Java program. We conclude the chapter with some quantitative analysis showing where we discuss the distribution of object history sizes along with the effectiveness of our filtering algorithm.

We present research related to our work on object histories in Chapter 5. Finally, we present our conclusions and directions for future work in Chapter 6.

# Chapter 2

# Object Histories

An object history summarizes the key events that happen to an object during its lifetime in a program. In particular, an object history for object $o$ is a set of events $e$, beginning with $o$'s instantiation, that happen to it over the course of program execution. Events in an object's history include its instantiation, writes of $o$ into a field, reads of fields, being passed to a method as an argument (including being added to or removed from collections), and being returned from a method as a return value.

Object histories are computed in two phases as shown in the boxes in Figure 2.1. The first phase consists of preprocessing where our system first builds a database of necessary information for the program of interest. Specifically, the database only contains information about the application methods and field events. The second phase consists of an algorithm for answering queries that queries the database on-demand to construct object histories for any object in the program.

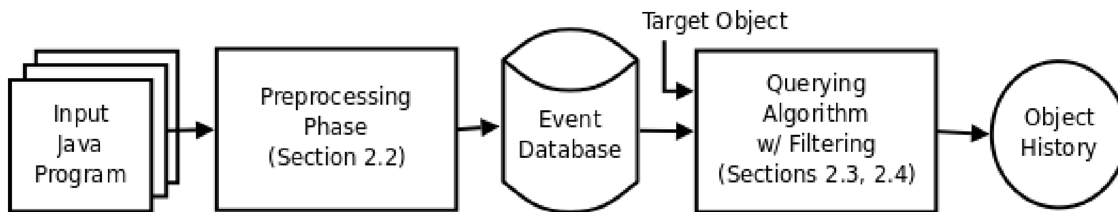


Figure 2.1: Architecture for computing object histories.

The next section provides a simple example to explain the concept of object histories and demonstrate the high level process for computing them. Next, we present details about the preprocessing phase followed by the querying algorithm which actually computes object histories. Finally, we present the details of filtering object histories to eliminate irrelevant events in an object's history.

## 2.1 Motivating Example

This section provides a high level explanation of our approach to building a database of events and computing individual object histories. Figure 2.2 shows a simple Java program that will be used to highlight our approach.

```
1  class X {
2    Object f;
3  }
4
5  public class Main {
6    public static void main(String[] args) {
7      X x1 = new X();
8      X x2 = new X();
9
10     foo(x1);
11     foo(x2);
12
13     bar(x1);
14   }
15
16   public static void foo(X x) {
17     x.f = new Object();
18   }
19
20   public static void bar(X x) {
21     Object t = new Object();
22     ...
23     t = x.f;
24     ...
25     // Get history for t
26   }
27 }
```

Figure 2.2: Example Java Program.

As mentioned previously, computing object histories consists of 2 stages. The first stage preprocesses the program to compute a database of key events for all objects in the program. The second stage consists of querying the database repeatedly to obtain the final object history for a given object.

The first stage performs an intraprocedural history analysis for each method in the program. This analysis computes the history of all variables at the method level. For the example in Figure 2.2, the intraprocedural history analysis for method `main()` would mark `x1` and `x2` as local instantiations due to the "`new() X`" statements within the method. The variable `args` is marked as formal parameter 0 to the method. The intraprocedural history analysis properly handles multiple definitions of a local variable at different points in the method. For example, in method `bar()`, the variable `t` refers to a local instantiation until line 22, but a field copy after line 23. Our analysis properly distinguishes the two different `t` variables.

Given intraprocedural histories for all application methods, the next step is to compute method summaries. The method summaries organize information necessary to compute object histories for efficient look-up by our querying algorithm (Section 3.2.2). They include a list of callers of the method, a list of invocations of other methods along with the list of arguments passed to them, and the return values, if any. For method `main()` in the program in Figure 2.2, the method summary would include 2 invocations of method `foo()` at lines 10 and 11 with parameters `x1` and `x2` and 1 invocation of method `bar()` at line 13 with parameter `x1`. Since the method is `void` and has no callers from the call graph, nothing is recorded for its return value and its list of callers is empty. Method summaries only need to be computed once for each program.

The final part of the preprocessing stage is to capture field events. We capture field events separately from method summaries to quickly identify all changes to a field `f` of class C. Field events are field-based, meaning that we group events relevant to field `f` of all instances of a particular class, `C`. Of course, such an approach records many potentially irrelevant events. We introduce filtering to eliminate many irrelevant events during the querying stage. Filtering is a technique that compares relevant points-to-sets to determine whether an event should be included in an object's history.

In Figure 2.2, field `f` of class `X` would be marked with 2 WRITE events in method `foo()` at line 17 (1 for each call to method `foo()`) as well as a READ event in method `bar()` at line 23.

Once the preprocessing stage has been completed, we have a database of field and method events that can be queried to obtain object histories. Consider a query for the history of object `t` after line 23 in Figure 2.2. It can be computed as follows:

1. Use method `bar()`'s intraprocedural history analysis to obtain history of `t` within method `bar()` after line 23. The history will contain an assignment of field `x.f` to `t` at line 23.

2. Query list of field events for all events that happened to field `f` of class `X`. The list will contain 1 field READ at line 23, 1 field WRITE at line 17 from call to `foo()` at

line 10, and 1 field WRITE at line 17 from call to `foo()` at line 11. Note that the second WRITE is actually irrelevant to the history of object `t` since the parameter used is `x2`. Step 3 therefore trims the set of events.

3. Filter irrelevant events by using the points-to-set for the instance of `x` at line 23 (which is `x1`) and intersecting it with each of the points-to-sets for the events in the field history of `X.f`. Only keep events that have a non-empty intersection. This will eliminate the irrelevant field WRITE caused by the method invocation of `foo()` at line 11 where `x2` was passed.

The process described in this section is a high level overview of computing object histories. The next sections will cover each stage of the process in greater detail.

## 2.2 Preprocessing Phase

We next discuss preprocessing in more detail. Recall that the preprocessing phase in computing object histories consists of building a database of method and field events that can be queried to compute object histories, and that this stage only needs to be run once for a given program. The preprocessing stage collects intraprocedural history analysis results, method events and field events, as well as some information necessary for the querying algorithm. We next present the details of each of these sub-stages.

### 2.2.1 Intraprocedural History Analysis

The intraprocedural history analysis captures the method-level history of all variables within a method body. Only changes to an object's history within the scope of the method are considered in this phase. More specifically, for each method $m$ and each local variable $v_m$, we compute the possible sources of $v_m$. There are five possible sources: 1) $m$'s formal parameters, 2) reads from fields, 3) return values from method calls, 4) new object instantiations, and 5) `catch` exception handling blocks.

Figure 2.3 shows a simple Java method and Table 2.1 shows the results of its intraprocedural history analysis.

```
1  public Object foo(Object p1) {
2    Object a = new Object();
3
4    Object b = null;
5    try {
6      b = bar();
7    } catch (Exception e) {
8      System.out.println(e.getMessage());
9    }
10
11   Object c = null;
12
13   if (b != null)
14     c = p1;
15   else
16     c = a;
17
18   return c;
19 }
```

Figure 2.3: Example Method.

Table 2.1: Intraprocedural History Analysis for method in Figure 2.3.

| **Variable Name** | **Origin** | **Additional Information** |
|---|---|---|
| p1 | PARAMETER_0 | Line 1, Object p1 |
| a | LOCAL_INSTANTIATION | Line 2, Object a = new Object() |
| b | NULL_CONSTANT | Line 4, Object b = null |
| | METHOD_INVOCATION | Line 6, b = bar() |
| e | EXCEPTION | Line 7, catch (Exception e) |
| c | NULL_CONSTANT | Line 11, Object c = null |
| | PARAMETER_0 | Line 14, c = p1 |
| | LOCAL_INSTANTIATION | Line 16, c = a |

A variable `v` in a given method's body may have multiple origins. This can occur due to multiple assignments to `v`. For instance, variable `c` has multiple possible sources depending on whether `b` is null. In case a method has multiple exit points (due to multiple

return statements), the intraprocedural history analysis will be a union of all the individual history analyses since it is necessary to capture all possible origins of local variables.

### 2.2.2 Method Summaries

We compute method summaries after completing the intraprocedural history analysis for all application methods. Method summaries contain information which the querying algorithm will use to compute an object history.

Method summaries for object histories contain relevant events and information required for computing object histories. Specifically, they include 4 components: 1) the intraprocedural history analysis for the method, 2) a list of callers of the method, 3) a list of methods invoked (along with their arguments), and 4) the return values, if any.

Figure 2.4 shows a simple Java program and Table 2.2 shows the associated method summaries.

```
1  Main {
2    public void main() {
3      Object a = new Object();
4      ...
5      Object b = foo(a);
6    }
7
8    public Object foo(Object x) {
9      Object o = new Object();
10     ...
11     Object x1 = x;
12     ...
13     return bar(x1);
14   }
15
16   public Object bar(Object b) {
17     ...
18     if (b != null)
19       return new Object();
20     return null;
21   }
22 }
```

Figure 2.4: Example Java program for demonstrating method summaries.

Table 2.2: Method Summaries for program in Figure 2.4.

| Method Name | Summary Component | Information |
|---|---|---|
| void main() | Intraprocedural History Analysis | a: LOCAL_INSTANTIATION<br>b: METHOD_INVOCATION |
| | List of Callers | - |
| | Invoked Methods | Object() constructor<br>foo(): argument_0: a |
| | Return Values | - |
| Object foo() | Intraprocedural History Analysis | x: PARAMETER_0<br>o: LOCAL_INSTANTIATION<br>x1: PARAMETER_0 |
| | List of Callers | main() |
| | Invoked Methods | Object() constructor<br>bar(): argument_0: x1 |
| | Return Values | METHOD_INVOCATION: bar() |
| Object bar() | Intraprocedural History Analysis | b: PARAMETER_0 |
| | List of Callers | foo() |
| | Invoked Methods | Object() constructor |
| | Return Values | LOCAL_INSTANTIATION<br>NULL_CONSTANT |

There are a few points to note:

- **Constructor Calls:** The method summaries described above do not distinguish constructor calls from regular method calls. This is because an object's instantiation may originate from a constructor call.
- **Recursive Methods:** Recursive methods are marked off with a flag to enable the querying algorithm to identify them easily.
- **Points-to-Sets:** Points-to-sets for base objects in instance method invocations and instance field reads or writes are also recorded in method summaries. This is important because we use a filtering technique based on pointer analysis to eliminate irrelevant events from object histories.

- **Return Values:** A method `m` may have multiple return values based on the control flow graph. For example, method `bar()` in Table 2.2 returns the `null` constant or a local object depending on the value of `b`.

### 2.2.3 Aggregating Field Events

Field events are recorded separately from method summaries in our approach. Trying to compute all possible field events in a program by simply using method summaries is difficult. The difficulty arises since a field `f` of class `C` can be modified in several different methods. By using method summaries alone, we cannot identify exactly which methods modify `f`. The problem is further complicated because classes often use Java collections like `HashMap` and `ArrayList` as fields and frequently manipulate them by adding and removing objects to them. We use a field-based approach where events for all fields `f` of class `C` are grouped. This allows us to find changes to a field throughout the program efficiently, but adds potentially irrelevant events to object histories. As mentioned previously, we use a filtering technique based on comparing relevant points-to-sets to omit many irrelevant events.

Field events are defined as writes to fields (`o.f = x`), reads from fields (`x = o.f`), and method invocations where the field is the base of the invocation (`x.f.add(o)`). Figure 2.5 shows a simple Java program and Table 2.3 shows the results of aggregating field events on the program.

```
1  class X {
2    public Object f;
3    private ArrayList l;
4
5    public X() {
6      l = new ArrayList();
7      f = new Object();
8    }
9
10   public void add(Object a) {
11     l.add(a)
12   }
13 }
14
15 class Main {
16
17   void main() {
18     X x = new X();
19     Object o = new Object();
20     x.add(o);
21     ...
22     Object c = x.f:
23     ...
24     int h = c.hashCode();
25   }
26
27   void bar() {
28     X x1 = new X();
29     Object a = new Object();
30     x1.f = a;
31   }
32 }
```

Figure 2.5: Example Program with Field Events.

Table 2.3: Aggregated field events for program in Figure 2.5.

| Field | Event | Location | Additional Information |
|---|---|---|---|
| X.f | WRITE | Line 7, X: X() | f = new Object() |
| | READ | Line 22, Main: main() | Object c = x.f |
| | METHOD_INVOKE | Line 24, Main: main() | c.hashCode() |
| | WRITE | Line 30, Main: bar() | x1.f = a |
| X.l | WRITE | Line 6, X: X() | l = new ArrayList() |
| | METHOD_INVOKE | Line 11, X: add(Object) | l.add(), argument_0: a |

We aggregate field events to also capture events on variables that alias fields. In the example shown in Figure 2.5, the method `hashCode()` is invoked on object `c`, which aliases field `x.f` as a result of the assignment on line 22. Thus, we infer that the method `hashcode()` is invoked on field `x.f`.

There can also be a case where one field is assigned the value of another field (`x.f = y.g`). In this case, we record a field read event for `y.g` and a field write event for `x.f`. This ensures that we account for events that may occur from aliasing. For instance, if we wish to extract the history for `x.f`, our querying algorithm will also extract the history of `y.g` because they are aliased. Note that aliasing occurs through temporary variables in our implementation.

Our approach also accounts for changes to `private` fields from getter and setter methods. For example, in Figure 2.5 above, suppose that class `X` contains getter (`getF()`) and setter (`setF()`) methods for field `f`. In this case, we would record a field read from `X.f` in method `getF()` and a field write to `X.f` in method `setF()` because we identify `this` object with class `X`. Our approach is object-insensitive, however our filtering algorithm described in Section 2.4 has some advantages of object sensitivity.

The major problem with a field-based approach is that all field events on all instances of class `X` are grouped. In Figure 2.5, we cannot distinguish the base objects in field write `x1.f = a` (line 30) and `f = new Object()` (line 7) because our algorithm simply treats both events as a WRITE to `X.f`. If computing the history of `x.f` on line 22, both events would be reported, even though `x1.f = a` on line 30 is clearly irrelevant because it does not affect `x.f` in method `main()`. We present a more detailed description of the problem and a technique for filtering out irrelevant events from object histories in Section 2.4.

### 2.2.4 Handling Java Collections

Java collection classes like `HashMap` and `ArrayList` are frequently used by Java programs to store and retrieve objects so we must handle them. This section presents our approach for handling collections.

Java provides collection interfaces and implementations of common collection data structures. The fundamental Java collection interfaces are `java.util.Map` and `java.util.Collection`. All Java collection classes directly or indirectly implement one of these interfaces. For example, `java.util.ArrayList` implements `java.util.List`, which extends `java.util.Collection`. Therefore, we would mark `java.util.ArrayList` as a collection class.

Aside from the Java collection classes, developers can also create custom collections. We assume that developers mark these collections by directly or indirectly implementing one of the fundamental collection interfaces. Thus, we identify collection classes by simply checking whether a class directly or indirectly implements one of the fundamental collection interfaces.

To account for additions and removals from collections, we track relevant collection interface method invocations such as `add()`, `put()`, `remove()`, etc along with the arguments passed to them and the target collection. These method invocations are marked with a special COLLECTION_READ or COLLECTION_WRITE flag. The information on collection method invocation and arguments is used to track what objects were added to and removed from a particular collection.

Collection classes are typically used as fields of classes, parameters to a method, or as a short-term data structure having scope within a method. Our approach for handling collections is therefore structured accordingly:

- **Collections as Fields:** If a collection is a field of a class, all method invocations on it will be recorded as part of the field events approach described in Section 2.2.3. In particular, method invocations along with information about the arguments passed and the base collection will be recorded. Additionally, if the invoked method adds to or removes from a collection, it is marked with a special COLLECTION_READ or COLLECTION_WRITE flag.
- **Collections as Parameters:** If a collection `l` is passed to a method `m` as a parameter, all method invocations on `l` will be recorded as part of `m`'s method summary. Recall that `m`'s method summary contains all method invocations within `m` along with a list of arguments passed to the invoked method. Again, if the method invocation adds or removes from a collection, it is marked with COLLECTION_READ or COLLECTION_WRITE flag.

- **Collections with Method Level Scope:** Collections that have method level scope will also be recorded as part of the method summary. Specifically, the method summary will show the instantiation point of the collection and hold all invocations on the collection with information about the arguments passed.

Figure 2.6 shows a sample Java program that contains various uses of collections. The method `addToList()` simply adds an object to the `list` field of class `M`. Method `operate()` uses a local collection, `tempList`, to store some objects it gets from parameter collection `p1`.

```
1  import java.util.ArrayList;
2
3  class M {
4    private ArrayList list;
5
6    public void addToList(Object o) {
7      list.add(o);
8    }
9  }
10
11 class N {
12   public void operate(List p1, List p2) {
13     ArrayList tempList= new ArrayList();
14         ...
15     Object a = p1.get(0);
16     Object b = p2.get(0);
17     ...
18     tempList.add(a);
19     ...
20     M m = new M();
21     m.addToList(b);
22     ...
23   }
24 }
```

Figure 2.6: Example Program highlighting collection events.

Table 2.4: Method Summaries for program in Figure 2.6.

| Summary | Information |
|---|---|
| **M:void addToList(Object)** | |
| Intraprocedural History Analysis | o: PARAMETER_0 |
| | list: FIELD: M.list |
| List of Callers | N: void operate(List, List) |
| Invoked Methods | list.add(): arg_0: o, COLLECTION_WRITE |
| Return Values | - |
| **N: void operate(List, List)** | |
| Intraprocedural History Analysis | p1: PARAMETER_0 |
| | p2: PARAMETER_1 |
| | tempList: LOCAL_INSTANTIATION |
| | a: METHOD_INVOCATION, p1.get() |
| | b: METHOD_INVOCATION, p2.get() |
| | m: LOCAL_INSTANTIATION |
| List of Callers | - |
| Invoked Methods | List: p1.get(): arg_0: 0, COLLECTION_READ |
| | List: p2.get(): arg_0: 0, COLLECTION_READ |
| | ArrayList: tempList.add(): |
| | arg_0: a, COLLECTION_WRITE |
| | M: m.addToList(): arg_0: b |
| Return Values | - |

The collection `list` in method `addToList()` is a field of class `M`. Field event aggregations capture a method invocation of `add()` with argument `a` and base `list` at line 7.

Table 2.4 shows the method summaries for the program in Figure 2.6. The summary for `addToList()` identifies `list` as an `ArrayList` and thus marks the invocation of `list.add()` as a COLLECTION_WRITE.

The summary for method `operate()` identifies parameters `p1` and `p2` as collections and thus marks invocations of `p1.get()` and `p2.get()` as COLLECTION_READs. The local collection `tempList` is also marked off as an `ArrayList` and thus the invocation to `tempList.add()` on line 18 is marked as a COLLECTION_WRITE.

The example discussed above shows that adds and removes from Java collection classes are recorded within the method summaries or field events along with special collection

flags. These flags are later used to distinguish collection events and are used by the querying algorithm which handles collections specially.

Our approach also accounts for iterators that read from collections. We map each iterator to the collection that it traverses. For instance, in the statement `Iterator i = list.iterator()`, we associate `i` with the collection `list`. We mark an invocation of `i.next()` as a COLLECTION_READ of `list`. Thus, when extracting the history of an object obtained from an iterator (`x = i.next()`), we look for COLLECTION_WRITEs to `list`.

The `java.util.ListIterator` interface allows programmers to add to a list object using the `add()` and `set()` methods of `ListIterator`. In this case, we mark each invocation of `add()` and `set()` as a COLLECTION_WRITE to the collection being traversed.

## 2.3 Answering Queries

By combining intraprocedural object histories from several methods and field events, our system provides useful information to developers about the provenance of objects of interest in a program. This section presents an overview of our querying algorithm along with a detailed example.

An object history query consists of a pair $\langle m, v \rangle$ where $m$ is a method and $v$ is a local variable belonging to $m$. Our system answers a query with a sequence of history events. Recall that events are instantiation, writes, reads, being passed to methods as an arguments (including being added to or removed from collections), and being returned from a method as a return value.

We combine local object histories to answer queries. Starting at method $m$, we report events on variable $v$. If $v$ is a return value from method $m'$, we retrieve the method summary for $m'$ and recursively report events for the return value of $m'$ using its history. Similarly, if $v$ is parameter $n$ of method $m$, we use the call graph to fetch a list $C$ of callers to $m$ and recursively report events for actual parameter $n$ of each call $c \in C$ to $m$. The first two types of events are *local* effects, where the parameter is directly passed as a parameter or return value from one method to another.

Finally, if $v$ was read from field `f` of class `K`, we report object histories for each write to field `f` of `K`. If $v$ was retrieved from a collection $\ell$, we report histories for each addition to collection $\ell$. (Our filtering algorithm helps eliminate irrelevant events.) These events are *non-local* effects, where the object travels across the heap from one method to another. We collect most of our history information from local effects but we also account for non-local effects.

### 2.3.1 Comprehensive Example

Consider the example shown in Figure 2.7. It presents interesting heap and collection-based behaviour. More specifically, it creates a pair of `Objects`, stores them in collections, and retrieves them again. To make the example more interesting, the program uses helper methods to store and retrieve the objects from an underlying `ArrayList` field. This highlights the interprocedural techniques used in our approach.

```java
import java.util.ArrayList;
class M {
  private ArrayList list;

  public M() {
    list = new ArrayList();
  }
  public boolean addToL(Object a) {
    return list.add(a);
  }
  public ArrayList getList() { return list; }
}

public class Main {
  public static void main(String[] args) {
    M m = new M();
    M n = new M();

    Object a = new Object();
    Object b = new Object();

    m.addToL(a);
    n.addToL(b);

    Main main = new Main();
    Object x = main.bar(m);
    // x and a aliased after bar()

    System.out.println(x);
  }
  public Object foo(M m) {
    return m.getList().get(0);
  }
  public Object bar(M m) {
    return foo(m);
  }
}
```

Figure 2.7: Comprehensive example used to illustrate object histories.

Suppose that we are interested in computing the object history of `x` in method `main()` (line 26). Recall that the first step in answering queries is to build a database of core events. For conciseness, we assume that the intraprocedural histories, method summaries, and field events have already been computed as described in Section 2.2.

The object history of `x` is computed as follows:

1. The first step is to obtain the intraprocedural history of `x` within `main()`. The intraprocedural history analysis of `main()` would report that `x` was assigned from invoking method `bar()` at line 26. Thus, we must find the origin of the return value of `bar()`.

2. The summary for `bar()`'s will show that `bar()` returns a value from an invocation of `foo()` at line 35. Thus, we must find the origin of the return value of `foo()`.

3. The summary for `foo()` is interesting and shown in Table 2.5. Notice that intermediate variable, `temp$0` is introduced to hold the `ArrayList` returned by `m.getList()`. The return value for `foo()` comes from an invocation of method `temp$0.get()`. Because `temp$0` has declared type `ArrayList`, the invocation of `temp$0.get()` is marked off as a COLLECTION_READ. Thus, the return value of `foo()` comes from collection `temp$0` and we must find: 1) the original collection, 2) all COLLECTION_WRITE events that added objects to the collection, and 3) the origins of each object that was added to the collection.

Table 2.5: Method summary for method Main: Object foo(M) shown in Figure 2.7.

| **Summary Component** | **Information** |
|---|---|
| Intraprocedural History Analysis | m: PARAMETER_0<br>temp$0: METHOD_INVOCATION:<br>M: ArrayList m.getList() |
| List of Callers | Main: Object bar(M) |
| Invoked Methods | m.getList()<br>temp$0.get(), argument_0: 0 |
| Return Values | METHOD_INVOCATION:<br>temp$0.get(), COLLECTION_READ |

4. To find the original collection, we need a stable name for the collection pointed to by `temp$0`. We note that `temp$0` comes from the return value of `M: getList()` using the intraprocedural history analysis of `foo()`. Thus, we next look at the return value of `getList()`.

5. The intraprocedural history analysis for `M: getList()` will show that the return value is the field `list` of class `M` at line 11. Hence, `temp$0` in `foo()` points to `M.list`. We must now find all COLLECTION_WRITE events to the field `M.list`.

6. To find COLLECTION_WRITE events on field `M.list`, we use the field events information for `M.list` shown in Table 2.6. The only COLLECTION_WRITE event occurs in method `addToL()` on line 9 with argument `a`. Thus, we must now find the origin of `a`.

Table 2.6: Aggregated field events for `M.list` for program in Figure 2.7.

| **Event** | **Location** | **Additional Information** |
|---|---|---|
| WRITE | Line 6, M: M() | list = new ArrayList() |
| METHOD_INVOKE | Line 9, M: boolean addToL() | list.add(): argument_0: a, COLLECTION_WRITE |
| READ | Line 11, M: ArrayList getList() | return list |

7. Using the intraprocedural history analysis of `addToL()`, we find that `a` is parameter 0 to the method. Thus, the true origin of `a` could be any argument that was passed to `addToL()`. We must now find all callers of `addToL()` and obtain the history of argument 0 in the invocations. Note that our approach does not account for the order of invocations of `addToL()`.

8. Using the method summary for `addToL()`, we find that it was invoked twice in method `M: main()` at lines 22 and 23 with arguments `a` and `b` respectively. The bases in the invocations are `m` and `n`, respectively. We must now find the history of arguments `a` and `b` in method `M: main()`.

9. Method `M: main()`'s method summary immediately informs us that `a` and `b` are local instantiations at lines 19 and 20.

10. Because there are no further paths that can be expanded, the algorithm terminates by concluding that `x` originates from `a` or `b` in method `M: main()` at lines 19 and 20 respectively.

By looking more closely at the code, we note that `x` is really the variable `a` from method `M: main()`. Only the call to `addToL()` at line 24 is relevant because the base, `m`, was passed to method `bar()` at line 29. Therefore, the invocation `n.addToL()` at line 25 is irrelevant in the object history of `x`. The next section on filtering explains our approach to eliminate the call to `addToL()` at line 25 from the history of `x`.

## 2.4 Filtering

Our approach for handling field events is field based. While this method is fast, it causes irrelevant events to be included when computing object histories. Irrelevant events in an object history can also occur by simply using method summaries since different instances of the receiver object cannot be distinguished in a context-insensitive analysis of the method body. This section describes our approach for filtering out irrelevant events in object histories.

Consider the instance field READ `v = o.f` of field `f` belonging to class `C`. Our raw algorithm considers all WRITEs to field `C.f`, which would include several irrelevant WRITEs. Filtering reduces this set of WRITEs by running a points-to analysis query on every write to `p.f` checking that `o` may alias `p`.

Object history queries encounter instance invokes `n.foo()` when traversing the intraprocedural history information, say for a method `m()`. But the intraprocedural history information also contains information about all of the callers to `m()`, including points-to information about the base and parameters to `m()` at each of the sites calling it. Our filtering information rejects the call to `n.foo()` if it is inconsistent with the points-to information it has collected.

Filtering is therefore a postprocessing phase which filters the analysis results based on pointer analysis information before presenting the results to the user. To implement filtering, we augment our abstraction with all relevant points-to sets, and record them as we compute our intraprocedural histories, summaries, and aggregate field events. In response to a query, we filter method invocation events which correspond to calls to instance invoke statements as well as READ field events corresponding to reads of instance fields. (Our histories also account for static methods and fields.) The next section presents details on identifying filtering criteria.

### 2.4.1 Filtering Context

In order to filter successfully, it is important to know what points-to-sets to compare at different points in the querying algorithm. To address this problem, we introduce the

notion of a filtering context. The filtering context provides the relevant points-to-sets for every point in our filtering algorithm. More specifically, a filtering context contains 1) the points-to-set for the base of instance method invocations and instance field events, 2) the points-to-sets for each of the parameters to the method being analyzed, and 3) the points-to-set for `this`.

We determine the filtering context as the first step when extracting the object history of a variable. As the querying algorithm moves through different method and field events, we update the filtering context to always contain relevant points-to-sets for the base of instance methods and fields, the parameters, and `this`.

Consider the Java program shown in Figure 2.8. Suppose we wish to compute the object history of `x` on line 19. We would first look up the return value of `foo()` and find that it returns the field `C.f`. Next, we simply look up all WRITE events to `C.f` and find 2 WRITE events on lines 14 and 15. However, we cannot eliminate the WRITE event on line 15 without using filtering. More specifically, we compare the points-to-set for the instance `c` on line 25 with the points-to-set for `c1` and `c2`. These comparisons yield that `c` may alias `c1`, but does not alias `c2`, hence we can eliminate the field WRITE on line 15.

As mentioned before, tracking the filtering context throughout the querying algorithm helps to correctly identify which points-to-sets to use. The process of computing the object history of `x` using filtering is described as follows:

1. The object `x` was assigned from the invocation of instance method `N: foo(C)`. The filtering context points-to-sets at this point are: (base=n), (param0=c1), (this=n).
2. The return value of `N: foo(C)` originates from an instance field READ of `C.f`. The filtering context points-to-sets at this point are: (base=c1), (param0=c1), (this=n).
3. There are 2 field WRITE events on field `C.f` on lines 14 and 15 with bases `c1` and `c2` respectively. The intersection of the points-to-set for the base, `c1`, in the filtering context in Step 2 with the points-to-set for `c1` is non-empty. Therefore, field WRITE on line 14 should be included in the object history of `x`. However, the intersection of the points-to-set for the `c1` and `c2` will be empty and thus the field WRITE on line 15 will be eliminated because it is irrelevant.

```
1  class C {
2    Object f;
3  }
4
5  public class N {
6
7    void main() {
8      C c1 = new C();
9      C c2 = new C();
10
11     Object a = new Object();
12     Object b = new Object();
13
14     c1.f = a;
15     c2.f = b;
16
17     N n = new N();
18
19     Object x = n.foo(c1);
20
21   }
22
23   Object foo(C c) {
24     return c.f;
25   }
26 }
```

Figure 2.8: Java program highlighting the use of filtering.

The approach described above is also used to filter out irrelevant instance method invocations. In this case, we simply look up the callers of a method `m` and reject each caller `c` where its base points-to-set has an empty intersection with the `this` points-to-set of `m`. Intuitively, this type of filtering eliminates other callers of `m`; such callers are inconsistent with the current context of `m`. We next revisit the example shown in Figure 2.7 to highlight this approach.

Recall that in Section 2.3.1, we described the process for computing the object history of `x` at line 26 without filtering. In particular, our raw algorithm was unable to eliminate the call to `addToL()` at line 23. However, we can use the filtering context to eliminate it as follows:

1. At the query statement (line 26), the filtering context points-to-sets are: (base=null), (param0=m), (this=main). The filtering context remains the same after our algorithm inspects `bar()`.

2. In method `foo()`, we must change the filtering context before analyzing `M.getList()`. In particular, we must map the points-to-set for `m` to the points-to-set for `this` in `M`. The filtering context points-to-sets are now: (base=null), (param0=null), (this=m).

3. Recall that to find the history of `a` in `addToL()`, we must analyze each invocation of `addToL()`. However, we only include invocations where the base object in the invocation may alias `this`, in the context of the original query. In particular, we intersect the points-to-set for each base object `b` in all invocations `b.addToL()` with the points-to-set for `this` in the filtering context. We only include events where there is a non-empty intersection. In our example, we intersect the points-to-set for `this` in the filtering context with the points-to-sets for `m` and `n` at lines 22 and 23 respectively. Because the points-to-set for `this` in the filtering context is the same as the points-to-set for `m`, we keep the invocation `m.addToL()` and discard `n.addToL()`.

# Chapter 3

# Realizing Object Histories

In Chapter 2, we described the notion of object histories. We also showed how we compute object histories in two phases using various examples. In this chapter, we provide detailed descriptions of computing object histories as well as details on our implementation.

The first phase of building object histories consists of preprocessing a target Java program and building a database of key events. The database contains events that happen to fields, summaries for methods, and relevant points to sets. The second phase responds to user queries by repeatedly querying the database to extract the target object's history as it moves through the heap during the program's execution.

Our approach for computing object histories employs static analysis techniques. To build the database of events, we analyze and organize all program statements, objects, and methods in the target program. Because we analyze Java programs, we wrote our implementation using the Soot [24] bytecode analysis and transformation framework.

This chapter describes our implementation of object histories along with relevant background information. Section 3.1 will provide background information on static analysis and some implementation details. Next, we present the details of the algorithms for querying the events database along with our approach for filtering irrelevant events. We conclude the chapter with a brief look at the user interface of our implementation.

## 3.1 Background

In this section, we present relevant background on techniques we used to compute object histories. In particular, we perform static analysis on the target program for building our events database and employ a filtering technique based on pointer analysis to discard irrelevant events in object histories. Our implementation of object histories uses the Soot

framework with SPARK for points-to information. Thus, we present some background information on static analysis, pointer analysis, SPARK, and Soot in the next few sections.

### 3.1.1 Static Analysis

Static analysis is a technique for analyzing the source code of a program to estimate a program's runtime behaviour without actually executing it. Because the program is never executed, fully-precise static analysis is generally undecidable and must use approximations [10, 1]. Most practical difficulties arise from estimating the effects of conditional statements and values of pointers. For example, consider the following snippet of code:

```
if (...)
  y = 1
else
  y = 0
```

During program execution, `y` is either 0 or 1 because only one branch in the conditional statement is executed. However, which branch is executed cannot be determined statically, most importantly due to the Halting Problem [10]. To tackle this problem, static analysis assumes that both branches are executed. Generally speaking, static analysis assumes that all paths in a program are executable [1]. While there has been some work in identifying branches that cannot be executed (for example [25, 3, 18]), these approaches can only approximate the runtime behaviour of software.

### 3.1.2 Pointer Analysis and SPARK

Static analysis of languages with pointers and dynamic storage such as Java and C is hard: finding the exact runtime values of variables in static analysis is generally uncomputable [10]. In particular, two or more variables could point to the same memory location. The concept of *aliasing* is used to describe this situation. We say that variables `x` and `y` *alias* each other if they point to the same memory location at some point in the execution of a program. In that case, a write to `x` will also change the value of `y`, implying that the object pointed to by `y` might change unexpectedly. Finding these aliases statically is challenging [12, 11].

There are two types of aliasing: may alias and must alias. If variables `x` and `y` alias in *some* execution of the program, we say that `x` and `y` may alias. If `x` and `y` alias on all executions of the program, we say that `x` and `y` must alias. We can extract may alias and must alias information for a program by using pointer analysis.

Pointer analysis (points-to analysis) [7] is a technique for determining the set of memory locations to which a variable may point. Pointer analysis essentially divides the heap into discrete abstract locations. Each abstract location corresponds to memory locations in concrete executions of the program. A points-to set for a variable `x` represents a set of heap locations that `x` may point to. By using points-to sets, we can determine whether a pair of variables `x` and `y` alias by simply intersecting their points-to sets. An empty intersection means that `x` and `y` do not alias.

In the context of object histories, we use points-to sets to filter out irrelevant events when computing object histories (Sections 2.4, 3.4). Specifically, we intersect points-to sets relevant to our query with points-to sets for an event `e`. We only report `e` if there is a non-empty intersection. Our methodology ensures that event `e` is relevant to our query.

To obtain points-to-sets for variables in a target program, we use the SPARK [13] Java points-to analysis framework. We chose SPARK because it is easy to use and it is integrated into the Soot Java bytecode analysis framework, enabling static analysis on Java programs using a single analysis framework. In the next section, we present some background information on Soot.

### 3.1.3 Soot

Soot [24, 23] is a Java bytecode analysis framework. Its purpose is to enable the analysis and transformation of Java bytecode. Soot can be used as a tool to inspect class files or as a framework to develop software that operates on Java bytecode.

Soot transforms Java bytecode into intermediate representations to perform various optimizations. We implement our preprocessing stage (Sections 2.2, 3.2) as a bytecode transformation module that operates on the Jimple intermediate representation of a target Java program. Specifically, our object history module performs the intraprocedural object history analysis, builds method summaries, and aggregates field events. We present details of the Jimple intermediate representation in the next section.

### 3.1.4 Jimple

Jimple [24, 23] is an intermediate representation of Java bytecode used by Soot. Jimple was introduced because Java bytecode is difficult to manipulate and optimize. The difficulty arises because Java bytecode is stack-based, which introduces implicit dependencies between statements.

Jimple is a typed, 3-address code representation of Java bytecode. Figure 3.2 shows the Jimple representation for the example program shown in Figure 3.1.

```java
public static long fibonacci(int n) {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n-1) + fibonacci(n-2);
}
```

Figure 3.1: Java method in original form.

```
public static long fibonacci(int)
{
    int n, temp$1, temp$4;
    long temp$0, temp$2, temp$3, temp$5, temp$6;

    n := @parameter0: int;
    if n <= 1 goto label0;

    goto label1;

 label0:
    nop;
    temp$0 = (long) n;
    return temp$0;

 label1:
    nop;
    temp$1 = n - 1;
    temp$2 = staticinvoke <Example: long fibonacci(int)>(temp$1);
    temp$3 = temp$2;
    temp$4 = n - 2;
    temp$5 = staticinvoke <Example: long fibonacci(int)>(temp$4);
    temp$6 = temp$3 + temp$5;
    return temp$6;
}
```

Figure 3.2: Jimple representation for Java method in Figure 3.1.

Because Jimple is a 3-address code, it introduces temporary stack variables such as `temp$1` and `temp$2` to hold intermediate values in complex statements. Also, note that all variables are given explicit types. Thus, the Jimple representation of a program greatly simplifies analyses and transformations. We describe the relevant parts of Jimple representation for computing object histories, as needed, in the next few sections.

## 3.2 Preprocessing

Having described relevant background information, we now present details of our implementation of object histories. We start by describing the preprocessing phase. The goal of this phase is to build a database of events that can be queried to extract object histories. The preprocessing phase is written in Java using the Soot Java bytecode analysis and transformation framework and operates on the Jimple intermediate representation for the target program. Preprocessing consists of performing an intraprocedural history analysis on each application method, computing method summaries, and aggregating field events. The next 3 sections describe each of these steps in detail.

### 3.2.1 Intraprocedural History Analysis

The intraprocedural history analysis is a fundamental component of the events database. It holds information about the origin of all variables within a method body. The intraprocedural history analysis executes on every application method and the results are recorded as part of the method's summary.

The intraprocedural history analysis is a flow-sensitive data flow analysis. The analysis processes each statement and identifies the origin of each variable encountered. Origins of variables are propagated forward until the end of the method body.

Variable values in a method may originate from a local instantiation (`x = new Foo()`), an exception handling block (`catch e`), a return value from a method invocation (`return x`), a field read (`x = o.f`), or a formal parameter to the method. Our data flow analysis simply looks for definition statements and uses the type of the right operand to determine a variable's intraprocedural origin.

Figure 3.3 shows a sample Java program and Table 3.1 shows the origin of each of the variables along with the relevant Jimple representation.

```
1  public class Main {
2
3    static Object f;
4
5    public static void main(String[] args) {
6
7      Object o = new Object();
8
9      try {
10       exceptionThrowingMethod();
11     } catch (Exception e) {
12
13     }
14     Object x = m();
15     Object y = f;
16   }
17 }
```

Figure 3.3: Example method showing different intrapocedural origins.

Table 3.1: Jimple representation of the origins of variables from Figure 3.3.

| Name | Origin | Right Operand | Jimple Representation |
|---|---|---|---|
| o | Local Instantiation | New Expression | temp$0 = new java.lang.Object;<br>specialinvoke<br>temp$0.<init>()>;<br>o = temp$0; |
| e | Exception Handling Block | Exception Reference | e := @caughtexception; |
| x | Method Invocation | Invoke Expression | temp$0 =<br>staticinvoke<br><Main:java.lang.Object m()>();<br>x = temp$1; |
| y | Field Read | Field Reference | y = <Main: java.lang.Object f>; |
| args | Formal Parameter | Parameter Reference | args:=@parameter0:java.lang.String[]; |

From Table 3.1, we can see that all necessary information about the origin is found within the Jimple representation. For instance, consider the Jimple representation for x = m() in the Table 3.1. The Jimple representation informs us that m() is a static method from class Main that accepts no parameters and returns a java.lang.Object.

Variables may have different origins at different points in the method. For example, the conditional assignment x=foo() below means that x's intraprocedural origin may be from a local instantiation, from the return value of foo() or the return value of bar().

```
Object x = new Object();
...
if (..)
  x = foo();
else
  x = bar();
```

In order to account for multiple assignments, we store all possible origins of a variable. At control-flow merges, we take a union of the data flow sets (representing variable origins) from each conditional branch.

We define the data flow set used in the flow analysis as a map of key-value pairs. The key is the variable name and the value is a list of possible origins along with relevant information. Table 3.2 shows what information is kept depending on the *origin* of a variable.

Table 3.2: Data flow set organization in the intraprocedural history flow analysis.

| **Variable Name** | **Origin** | **Additional Information** |
|---|---|---|
| name | LOCAL_INSTANTIATION | [Line Number]<br>[Points-to-set]<br>[Program Statement] |
| | EXCEPTION | [Line Number]<br>[Points-to-set]<br>[Program Statement] |
| | METHOD_INVOCATION | [Line Number]<br>[Points-to-set]<br>[Method Signature]<br>[Base Name] if instance invocation<br>[Base Points-to-set] |
| | FIELD_READ | [Line Number]<br>[Points-to-set]<br>[Field Name]<br>[Base Points-to-set] if instance field |
| | PARAMETER | [Line Number]<br>[Points-to-set]<br>[Parameter Number] |

The data flow analysis populates the data flow set by extracting the necessary information from the Jimple representation of assignment statements, as shown in Table 3.1. As mentioned previously, the intraprocedural history analysis is performed for each ap-

plication method and the results are stored in the method's summary. The next section describes the implementation details of method summaries.

### 3.2.2 Method Summaries

Once we have executed the intraprocedural history analysis on a method $m$, we combine the results with additional information about $m$ to form method summaries. Method summaries are crucial in answering queries about the history of a particular object. They contain information about events that occur in a method body. Method summaries by themselves are enough to answer queries for objects which are never read from fields nor written to them.

Our implementation for computing method summaries is simple. We analyze the Jimple [24] representation for each program statement and extract appropriate information. Recall that method summaries are composed of 4 parts: 1) the intraprocedural history analysis of the method, 2) a list of callers of the method, 3) a list of methods invoked with a list of arguments passed to them, and 4) the return values, if any. A method summary also contains the relevant points-to-sets for objects in its body.

The information in the method summaries is used by our querying algorithm to extract object histories on-demand. The intraprocedural history analysis of the method informs us about the method level origin of each variable in the method. A list of callers is used to extract the history of a parameter to the method by finding its history in each of the callers. The list of invoked methods allows us to find the call sites for invoked methods and the arguments that were passed. Finally, if a variable was assigned from the return value of method $m$, the return values in $m$'s method summary allow us to quickly find all origins for each of $m$'s return values.

The information for each component of the method summary is obtained as follows:

1. **Intraprocedural History Analysis:** The intraprocedural history analysis is computed as described in Section 3.2.1 for each application method and the results are stored once the analysis is completed.

2. **List of callers:** The list of callers to a method is extracted from the call graph of the program. This list contains each caller `c` of method `m` only once even if there are multiple invocations of `m`. We do, however, hold different invocations of `m` in the caller's method summary.

3. **List of invoked methods and arguments:** If the method contains a method invoke expression, we include information about the invoke in our method summary.

Specifically, we include the signature of the invoked method, the names of the arguments used in the invocation, their points-to-sets, and the name and points-to-set for the base object, if it is an instance method invocation. This information is then added to the list of invoked methods.

4. **Return Values:** Extracting return values is simple using the Jimple representation. If the method is not `void`, we identify `return` statements and record the return value. Multiple return values are recorded if there are multiple return statements based on the control flow graph.

Once the method summary is completed for a given method, the summary is stored in a global method summary map of key value pairs. The key is the method's signature and the value is its method summary. This map is part of the events database that is built during the preprocessing phase.

### 3.2.3 Field Events

To build a complete events database, we also build information about field events. Fields in classes can be modified by several methods within the class body as well as methods from other classes. Simply using method summaries to extract the history of a field would be extremely inefficient. Therefore, we aggregate field events to quickly extract the history of events on a field.

Field events are the second and final component of the events database. During preprocessing, we store information about events that occur on fields. Field events are defined as READS, WRITES, or method invocations on fields throughout a program. Recall that our approach for aggregating field events is field-based, meaning that we group all events on field `f` of all instances of class `C`. Of course, this approach will lead to several events in the object history which may not be relevant to a particular instance of `C`. To eliminate irrelevant events in the history of an object, we apply pointer analysis based filtering to identify relevant field events and discard the rest.

Field events are easily identified in the Jimple representation of program statements. Table 3.3 shows the Jimple representation of a field READ, WRITE, and method invocation on static (STAT) and instance (INST) fields.

Table 3.3: Jimple representation for static and instance field events.

| Java Statement | Event Type | Jimple Representation |
|---|---|---|
| X.f = a; | STAT_FIELD_WRITE | <X: java.lang.Object f> = a; |
| a = X.f; | STAT_FIELD_READ | temp$0=<X:java.lang.Object f>;<br>a = temp$0; |
| X.f.toString() | STAT_FIELD_INVOKE | temp$0=<X:java.lang.Object f>;<br>temp$1 = virtualinvoke<br>temp$0.<java.lang.Object:<br>java.lang.String toString()>(); |
| x.f = a; | INST_FIELD_WRITE | x.<X:java.lang.Object f> = a; |
| a = x.f; | INST_FIELD_READ | temp$0=x.<X:java.lang.Object f>;<br>a = temp$0; |
| x.f.toString() | INST_FIELD_INVOKE | temp$0=x.<X: java.lang.Object f>;<br>temp$1 = virtualinvoke<br>temp$0.<java.lang.Object:<br>java.lang.String toString()>(); |

There are a few points to note:

- **Static vs. Instance fields:** Instance fields are identified if the Jimple representation contains a base for the field reference. For instance, the Jimple statement for `x.f = a` contains a `x.` before the class name, identifying `x` as an instance of class `X`. Static fields on the other hand, have no base specified. This is clear by looking at the Jimple representation for `X.f = a` that contains only the class name, `X`, and the field name, `f`.

- **Field Copy variables:** To properly handle field events, we must also account for events that happen to variables that originate from a field read. Note that the Jimple representation for static or instance FIELD_READS or method invocations introduces intermediate variables. In Table 3.3, the intermediate variable `temp$0` is introduced in both static and instance field reads.

  To account for these intermediate variables, we introduce a FIELD_COPY label to represent a value that originates from a field READ. The original field is stored with the FIELD_COPY variable name for easy access later. In Table 3.3, `temp$0` would be marked as a FIELD_COPY because the right operand in its definition is a field reference. When our analysis encounters the Jimple statement `a = temp$0`, we immediately mark the event as a static FIELD_READ on `X.f` because we know

that `temp$0` is a static FIELD_COPY of `X.f`. A similar approach is used to identify method invocations on fields by using FIELD_COPY variables.

- **Points-to-Sets:** Once a field event is identified, relevant points-to-sets are also stored to be used by our filtering algorithm. For instance field events and method invocations, the points-to-set for the base is recorded along with the points-to-sets for the arguments passed, if any. Our filtering algorithm compares points-to-sets to determine if events are relevant to a target query. We intersect the target points-to-set with the points-to-sets for a given event and if there is an empty intersection, we discard the event. We will present details about our technique for filtering in Section 3.4.

Field events are stored in a global field events map composed of key-value pairs. The key is the fully qualified field name, and the value is the list of field events. The field events map is simply written to disk as part of the events database. Once field events have been aggregated, the events database is complete and several queries can be answered on-demand.

### 3.2.4 The Events Database

As mentioned previously, we store information about events on all variables and methods in an events database to be used by our querying algorithm later. The events database is composed of a method summaries map and a field events map. The method summaries map contains key-value pairs where the key is a method signature and the value is the corresponding method summary. The field events map also contains key-value pairs with the key being the fully qualified field name and the value is a list of events that happen to the field along with other useful information about the event.

## 3.3 Answering Queries

By combining method summaries and aggregated field events, our system can compute the history of any object in a program. To answer object history queries, we recursively query the events database until no further events relevant to the target object can be extracted. This section presents the algorithm in detail.

Recall that an object history query consists of a pair $\langle m, v \rangle$ where $m$ is a method and $v$ is a local variable belonging to $m$. Our system answers a query with a sequence of history events. Events are instantiation, writes, reads, being passed to a method as an argument

(including being added to or removed from collections), and being returned from a method as a return value.

The querying algorithm to obtain the history of variable `v` in method `m` is shown in Algorithm 3.3.1. Algorithm 3.3.1 produces a list of events that describe the object history for variable $v$ found in method $m$. This history will report all possible origins of $v$ along with details about each origin (e.g. passed as parameter, field read, etc.) and the location of each of the events (e.g. line number, method name, etc).

The algorithm accepts the target object's name $v$ and the method name $m$ as parameters

and recursively reports $v$'s history.

**Algorithm 3.3.1:** GETHISTORY($v, m$)

```
summary ← GetMethodSummary(m)
originList ← GetOriginList(summary, v)
for each origin ∈ originList
  do
    if origin is local instantiation or origin is exception
      then report line number and instantiation statement
    else if origin is parameter_n
      then
        report origin is parameter n and line number
        callerList ← GetListOfCallers(m)
        for each caller ∈ callerList
          do
            callerSummary ← GetMethodSummary(caller)
            arg ← GetArgNameInInvoke(n, m, callerSummary)
            GetHistory(arg, caller)
    else if origin is field
      then
        report origin as a field and line number
        fieldName ← GetFieldName(origin)
        GetFieldHistory(fieldName)
    else if origin is method invocation
      then
        invokedMethod ← GetInvokedMethod(origin)
        if invokedMethod is a collection method
          then
            report origin is from a collection
            base ← GetCollectionBase(origin)
            GetCollectionHistory(base, m)
          else
            report origin is from method invocation
            iSummary ← GetMethodSummary(invokedMethod)
            returnValList ← GetReturnValues(iSummary)
            for each returnVal ∈ returnValList
              do GetHistory(returnVal, invokedMethod)
return
```

In the outside loop, we explore each origin for $v$ to extract its history. If *origin* is a local instantiation or from an exception block, the algorithm terminates.

If *origin* is parameter $n$ to method $m$, we obtain the list of callers of $m$, *callerList*, from $m$'s method summary. For each caller, *caller*, we get the caller's method summary, *callerSummary*, and the argument $n$ used in the invocation of $m$. Finally, we extract the object history of *argument* in method *caller*.

If *origin* is a field, we simply obtain the history of all events on the target field, $fieldName$. The algorithm for obtaining the field history simply reports all field events on $fieldName$.

Finally, if *origin* is returned from a method invocation, we check whether the invoked method has a COLLECTION_READ or COLLECTION_WRITE flag attached to it. If a collection flag is present, we extract the history of the collection, *base*, in method $m$. We present the details of obtaining the collection history in Section 3.3.1.

If the invoked method is not a collection method, we simply obtain a list of its return values, if any, from its method summary. Next, for each return value, $returnVal$, we obtain its object history.

### 3.3.1 Handling Collections

In Section 2.2.4, we described our approach to handling events on a collection. This section presents details on identifying collections and the algorithm for computing collection events.

To properly extract the histories of objects that get stored in collections, we must find additions and removals from collections. Recall that we identify a variable $v$ as a collection by checking whether $v$'s type directly or indirectly implements one of the fundamental collection interfaces, `java.util.Map` or `java.util.Collection`. Once $v$ is identified as a collection, we compare the signature of each instance method invocation with $v$ as the base with known collection method signatures that add or extract objects from collections. For instance, suppose `c` is an instance of `ArrayList`. Then, we mark the invocation, `c.get(0)` as a COLLECTION_READ because the invoked method signature, `ArrayList: Object get(int)` belongs to a known collection method.

Once we identify a variable $v$ as a collection, we must find all collection events on $v$. Collection events are method invocations that add to or remove from collection $v$. To find the history of an object that was retrieved from a collection, we simply look for all additions to the collection and obtain the history for each addition.

The first step in finding collection events is to look for them in the current method $m$. The algorithm for computing collection events for a collection *base* found in method $m$ is shown in Algorithm 3.3.2. The algorithm looks for collection methods invoked in $m$ and

obtains the history of the arguments used in the invocations.

**Algorithm 3.3.2:** GETCOLLECTIONHISTORYINMETHOD($base, m$)

$summary \leftarrow$ GETMETHODSUMMARY($m$)
$invokedMethodList \leftarrow$ GETINVOKEDMETHODS($summary$)
**for each** $invokedMethod \in invokedMethodList$
**do** {
  **if** invokedMethod is a collection method
  **then** {
    report addition to collection
    $argument \leftarrow$ GETARGUMENTFORMETHOD($invokedMethod$)
    GETHISTORY($argument, m$)
  }
}
**return**

To find all events on a collection, we must also consider the origin of the collection itself. For example, if an object is obtained from collection $c$ in method $m$ and $c$ is a parameter to $m$, we must get callers of $m$ and obtain the history of the collection $c$ to report all collection reads and writes on $c$. For each origin for a collection $c$, we must get the history of all objects that were added to $c$.

Algorithm 3.3.3 presents our methodology for finding the origin of a collection *base* in method $m$ and extracting the history of all additions to it. Note that it is very similar to the algorithm for answering queries shown in Algorithm 3.3.1 because the process for finding the origin of a variable, regardless of whether it is a collection or not, remains the same. We choose to implement a collection customized algorithm to handle COLLECTION_READ and COLLECTION_WRITE events and find the histories for objects that were added to

these collections.

**Algorithm 3.3.3:** GetCollectionHistory($base, m$)

```
GetCollectionEventsInMethod(base, m)
summary ← GetMethodSummary(m)
originList ← GetOriginList(summary, base)
for each origin ∈ originList
  do
    if origin is parameter_n
      then
        report collection is parameter n and line number
        callerList ← GetListOfCallers(m)
        for each caller ∈ callerList
          do
            callerSummary ← GetMethodSummary(caller)
            arg ← GetArgNameInInvoke(n, m, callerSummary)
            GetCollectionHistory(arg, caller)
    else if origin is field
      then
        report origin as a field and line number
        fieldName ← GetFieldName(origin)
        GetFieldCollectionHistory(fieldName)
    else if origin is method invocation
      then
        invokedSummary ← GetMethodSummary(invokedMethod)
        returnValList ← GetReturnValues(invokedSummary)
        for each returnVal ∈ returnValList
          do GetCollectionHistory(returnVal, invokedMethod)
return
```

The first step in the algorithm above is to extract the history of all collection events found within $m$. This is accomplished by invoking Algorithm 3.3.2.

Next, we use the method summary for $m$, *summary*, to obtain a list of origins for *base*. Note that if the base is a local instantiation, the algorithm terminates because all collection events would be found by Algorithm 3.3.2.

If *base* is parameter $n$ to $m$, for each caller *caller* of $m$, we obtain argument $n$ in the invocation of $m$, and recursively extract collection events.

If *base* is a field, we find the original field, $fieldName$ and extract all field events on it. We only report COLLECTION_READ and COLLECTION_WRITE events in this case.

Finally, if *base* is the return value from invocation of a method, $invokedMethod$, we extract a list of return values for $invokedMethod$ and extract the history for each return value.

Note that the algorithms presented in this section do not filter any results. Specifically, the algorithms shown above will report events that may be irrelevant to a given query because we do not filter the results. We describe our approach to filtering out irrelevant events in object histories in detail in the next section.

## 3.4 Filtering

Filtering enables us to discard irrelevant events in an object's history. Our raw querying algorithms report irrelevant field events because our approach is field-based, meaning that we group all events on field `f` of class C. Additionally, if the origin of a variable $v$ is a parameter to method $m$, events from all callers of method $m$ are reported even if some invocations of $m$ may be irrelevant. That is, we do not check if the base in the invocation to $m$ aliases `this` for our query.

In this section, we present details on our approach to filtering. Recall that we filter instance field and method invocations by intersecting points-to-sets for the base with the points-to-sets relevant to the query. Intersecting points-to-sets informs us whether two objects may alias or not. If we find an empty intersection, we know that the two objects are distinct. We obtain and store points-to-sets for variables during the preprocessing phase.

While answering queries, we must ensure that events we report are relevant to our target object. To keep track of what is relevant, we use the notion of a filtering context. The filtering context holds relevant points-to-sets used in filtering instance method and instance field events. Specifically, the filtering context contains the points-to-sets for the relevant base in instance fields or instance method invocations, arguments passed to an invoked method, parameters, and `this`. Each time the querying algorithm searches for history events outside a method, the filtering context is updated and passed on as a parameter to the recursive query. This ensures that the context in the original query is preserved as the algorithm recursively extracts an object's history.

Our filtering technique filters instance field events and instance method invocations by intersecting the appropriate points-to-sets in the filtering context with the base points-to-set in instance fields or instance method invocations.

Specifically, the filtering context is computed depending on whether the origin of a target is a parameter, field, or method invocation. We update the filtering context throughout the querying algorithm and always pass it as a parameter at each querying step that leaves the current method. At any given point in the algorithm, we simply intersect the base in the filtering context with the base of instance method invocations or instance fields. An empty intersection means that the two objects do not alias. We discard events if there

is an empty intersection. Because bases in instance field and method invocations may be parameters to a method, we track points-to-sets for parameter or arguments to keep track of the original context of the query.

Algorithm 3.4.1 shows the filtering context for an origin, $origin_v$ for query variable $v$ in method $m$. The filtering context is the set of points-to-sets (PTS) given by $\langle basePTS, argPTS, paramPTS, thisPTS\rangle$.

**Algorithm 3.4.1:** GetFilteringContext($origin_v, m$)

**if** $origin_v$ is parameter
  **then** $\begin{cases} basePTS \leftarrow \text{GetThisPTS}(m) \\ argPTS \leftarrow \text{GetParamPTS}(m) \\ filteringContext \leftarrow \langle basePTSet, argPTSets, \texttt{null}, \texttt{null}\rangle \end{cases}$
  **else if** $origin_v$ is instance field
  **then** $\begin{cases} basePTSet \leftarrow \text{GetBasePTS}(origin, m) \\ filteringContext \leftarrow \langle basePTSet, \texttt{null}, \texttt{null}, \texttt{null}\rangle \end{cases}$
  **else if** $origin_v$ is instance method invocation
  **then** $\begin{cases} invokedMethod \leftarrow \text{GetInvokedMethod}(origin_v) \\ paramPTS \leftarrow \text{GetArgPTSForInvoke}(invokedMethod, m) \\ thisPTS \leftarrow \text{GetBasePTSForInvoke}(invokedMethod, m) \\ filteringContext \leftarrow \langle \texttt{null}, \texttt{null}, paramPTS, thisPTS\rangle \end{cases}$
**return** (*filteringContext*)

Recall that if the origin of query $v$, $origin_v$, is parameter $n$ to method $m$, the querying algorithm looks up callers of $m$, and for each caller obtains the history of argument $n$ in the invocations to $m$. To filter out irrelevant invocations of $m$, we intersect $m$'s `this` points-to-set with base points-to-sets in invocations of $m$ within each caller's body. We only extract the history of argument $n$ if the intersection is not empty.

If $origin_v$ is an instance field, we simply obtain the points-to-set for the base and set it as the filtering context base points-to-set. When extracting field events, our algorithm will simply discard instance field events where the base points-to-set has an empty intersection with the filtering context base points-to-set. This will eliminate all instance field events, $e$, where the base class of $e$ is not potentially the same object as the base in the filtering context.

Finally, if $origin_v$ is an instance method invocation, the querying algorithm gets the history of the return value of the method. In this case, we filter events in the invoked method, $invokedMethod$, by comparing the base points-to-set in its invocation in $m$ with events on `this` within $invokedMethod$'s body. To accomplish this, we simply set the base in the invocation to the `this` points-to-set in the filtering context. We also set the

parameter points-to-sets in the filtering context to the points-to-sets for arguments used in the invocation.

## 3.5 Graphical User Interface

We implemented a simple graphical user interface to enable queries to our events database. A screenshot of our implementation is shown in Figure 3.4.

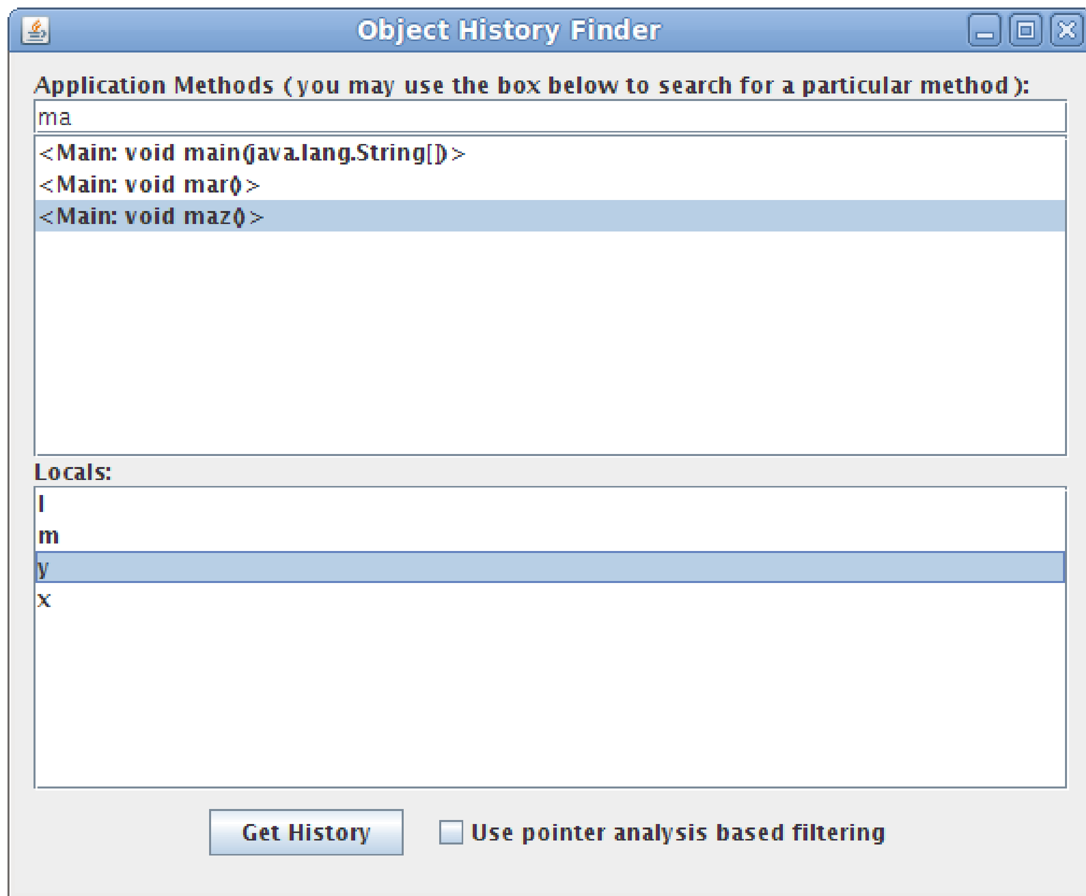


Figure 3.4: Graphical front-end for extracting object histories.

The interface presents the user with a list of application methods and variables defined within each method. Users can search for a particular method of interest. Once a user selects a method $m$ and a variable within its body $v$, the interface simply invokes the querying algorithm with the pair $\langle m, v\rangle$ and displays $v$'s object history. The interface also allows the user to enable and disable filtering on-demand.

# Chapter 4

# Experimental Results

We have implemented our object histories analysis and examined its effectiveness on a number of benchmarks. Our hypothesis is that object histories can be used to directly fix bugs where an object $o$ is assigned a faulty value. By extracting $o$'s object history, a developer can quickly identify erroneous statements that change $o$'s state and fix the bug. Developers are often required to maintain and update their code or make small changes in code behaviour. Making these changes without properly understanding the implications of the change on dependent code can lead to the creation of faults. Object histories can help identify dependent code and allow developers to make changes with confidence.

To support our hypothesis about object histories, we explore their usability in various applications in this chapter. Section 4.1 presents two sample use cases where object histories enhance program understanding. In Section 4.2, we present a bug we fixed in an open source project using our implementation of object histories. In Section 4.3, we describe our experience with computing object histories for three open source Java benchmarks. We explore object history size distributions in our benchmarks in Section 4.4. Finally, in Section 4.5 we demonstrate the effectiveness of our filtering technique to eliminate irrelevant events from object histories.

## 4.1 Case Studies

We believe object histories can significantly contribute to developers' understanding of large software systems: by extracting the object history for key objects, our system leads developers to key parts of a program that are relevant to the task at hand. To highlight the utility of object histories in program understanding, we present two use cases where object histories help developers understand programs better. In Section 4.1.1, we demonstrate

the use of object histories in bug fixing. Next, in Section 4.1.2, we use object histories to help us in changing code.

### 4.1.1 Finding Erroneous State

In large programs, finding the source of bugs can be prohibitively difficult because variables can be read and modified at several program points. The standard fault/failure model states that for a fault to be observed, the location of the fault must be reached, the state of the program must be infected, and finally, the infected state must propagate to cause some erroneous output [4, 17]. Thus, an obvious application for using object histories is to find the reachable point where the program state was infected. Object histories enable developers to examine the propagation path of the infected state from the fault to the output.

Figure 4.1 shows parts of a plausible online shopping system we wrote to illustrate the usefulness of object histories in finding erroneous state. Note that we omit details for brevity. Each item is represented by an instance of the `Item` class. The `ShoppingCart` class contains various items and provides features to manipulate them. Once a customer is ready to check out, the system invokes method `checkout()` to finalize the purchases.

The program in Figure 4.1 contains a bug in method `checkout()`, where occasionally, the total cost for all items in a shopping cart is not computed correctly. The bug arises because the method `addSelectedItem()` does not check whether the price returned by `getPrice()` is positive.

```java
import java.util.ArrayList;
class Item {
  int productID;
  int price;
  ...
}
class ShoppingCart {
  private ArrayList<Item> items;

  public void addItem(Item item) {
    items.add(item);
  }
  public ArrayList<Item> getItems() {return items;}
  ...
}
public class Actions {
  public void checkout(ShoppingCart cart) {
    ArrayList<Item> items = cart.getItems();
    int total = 0;
    for (int i = 0; i < items.size(); i++) {
      Item item = items.get(i);
      total += item.price;
      ...
    }
  }
  public void addSelectedItem(int productId,
                              ShoppingCart cart){
    Item item = new Item();
    ...
    item.price = getPrice(id);
    cart.addItem(item);
  }
  public int getPrice(int productId) {
    if (!validProductID(productId) < 0)
      return -1;
    return getPriceFromDB(productID);
  }
}
```

Figure 4.1: Java program containing a bug found by using object histories.

Even in this simple example, it is not immediately obvious where the bug lies. These types of bugs are often extremely time consuming to find in large code bases. However, by using object histories, the bug can be solved extremely quickly.

Because the code in method `checkout()` seems correct, we can infer that there must be an item in the shopping cart with an incorrect price. Finding all additions to the shopping cart is not immediately obvious even in this small example. Specifically, we cannot simply perform textual search (grep) to determine the erroneous statement due to collections used in the program. However, by using object histories, we can simply extract the object history of `item` in method `checkout()` to find all program points where items were added to a shopping cart. Figure 4.2 shows the (lightly formatted) output of our object history tool for the history of `item` in method `checkout()`.

```
Object history for method: checkout() and variable: item

item was assigned from invocation of method: <ArrayList: get()> (ln 23)
...variable came from collection: items (ln: 20)
...Looking up events on collection items
...Collection items was returned from method <ShoppingCart: getItems()> (ln 20)
...Looking up return value of <ShoppingCart: getItems()>
......Variable is copy of field <ShoppingCart: ArrayList items>
......Looking up all additions to collection <ShoppingCart: ArrayList items>
......Found add() call to collection (ln 11)
......Getting History of argument to add(): item
.........item was parameter number: 0 to method: <ShoppingCart: addItem()>
.........<ShoppingCart: addItem()> was invoked in <Actions: addSelectedItem()> (ln 32)
.........item was argument 0 in invocation
.........Looking up argument 0 's history
............Variable is local (ln 29)
```

Figure 4.2: Object history for `item` in method `checkout()` from Figure 4.1.

The object history for `item` in method `checkout()` immediately leads us to method `addSelectedItem()` which adds items to a shopping cart at line 32. Upon inspection of this method, we immediately find that the `price` field of an item is assigned from an invocation of method `getPrice()` at line 31. Next, we note that method `getPrice()` may return a negative price and at this point, we have found our bug. Now, we can simply fix the bug by adding a check for a negative price returned by `getPrice()` at line 31.

Note that we could have also extracted the history of field `item.price` in method `checkout()` to find the program points where an item's price was set. The output for the object history of `item.price` in method `checkout()` is shown in Figure 4.3.

```
Object history for method: checkOut() and variable: <Item: price>

<Item: int price> is a field object. Information on all changes to this field:
...Field was changed in method: <Actions: addSelectedItem()>
...Field was assigned from an invocation of method <Actions: getPrice()> (ln 31)
...Looking up history of return values
......Variable is local: -1  (ln 36)
......Variable was returned from method <Actions: getPriceFromDB()>

...
```

Figure 4.3: Object history for field `item.price` in method `checkout()` from Figure 4.1.

The field history of `item.price` immediately informs us that a negative price was set at line 36. Therefore, we can apply the fix in method `addSelectedItem()` to solve the bug.

By obtaining object histories for relevant objects in the program, we were able to quickly pinpoint the source of the bug. In this simple example, we could have fixed the bug by inspection. However, in large object-oriented software systems, following complex call hierarchies and finding all points where a variable is changed to an erroneous state can be extremely difficult and time consuming. We show the application of object histories to finding bugs in a large open source program in Section 4.2.

### 4.1.2 Change Support

Developers of large software systems must constantly maintain and update their code. A common kind of change is refactoring, which should not change the semantics of the code. However, developers also often need to slightly change the behaviour of code. Ad-hoc changes of program behaviour can trigger faults in code which depends on the modified code. Object histories can help identify dependent code and enable developers to confidently carry out needed changes.

Object histories can help developers identify dependencies between classes and understand how the each class is used. For instance, suppose a developer must change class $C$. To understand what effect the change may have, the developer can simply extract the object history for each field $f$ of class $C$. The object histories will show the program points where $f$ is modified. The combined object histories for all fields of $C$ can help the developer to understand the dependencies between $C$ and the rest of the program. The object histories will also contribute to the developer's understanding of how $C$ is used by the rest of the code. Finally, object histories can also be used to better understand the scope of a refactoring task by studying the dependencies between classes.

We present a simple example shown in Figure 4.4 to highlight this approach. Suppose we are interested in changing class `A` in the example shown in Figure 4.4. Class `A` is used as a field in `B` and also used in method `foo()` in class `C` for some operations.

```
1  class A {
2    private Object a1;
3    private Object a2;
4
5    public void setA1(Object a) {
6      a1 = a;
7    }
8    public void setA2(Object a) {
9      a2 = a;
10   }
11   ...
12 }
13 class B {
14   private A a;
15
16   public B(Object a1, Object a2) {
17     A a = new A();
18     a.setA1(a1);
19     a.setA2(a2);
20     this.a = a;
21   }
22 }
23
24 class C {
25   public void foo(Object o) {
26     B b = new B(o, new Object());
27     ...
28     A a = new A();
29     a.setA1(o);
30     ...
31   }
32 }
```

Figure 4.4: Code Change Example.

To properly modify `A`, we must first understand how to make changes to `A` without

causing bugs in other parts of the program. By using object histories, we can simply extract the object history for fields `A.a1` and `A.a2` to easily identify program points where `A` is used and modify them accordingly. The field histories of `A.a1` and `A.a2` using our object history implementation are shown in Figure 4.5.

```
Object history for method: setA1() and variable: <A: a1>

<A: a1> is a field object. Information on all changes to this field:
Field was changed in method: <A: setA1()>
...Field was assigned parameter 0 in method <A: setA1()>
...a was parameter number: 0 to method: <A: setA1()>
...Looking up history of a
......Method: <A: setA1()> was invoked in method: <C: foo()> (ln 29)
......a was argument 0 in invocation.
......Looking up argument 0 's history
......o was parameter number: 0 to method: <C: foo()>
......Looking up history of o

...

......Method: <A: setA1()> was invoked in method: <B: <init>()> (ln 18)
......a was argument 0 in invocation.
......Looking up argument 0 's history
......a1 was parameter number: 0 to method: <B: <init>()>
......Method: <B: <init>()> was invoked in method: <C: void foo()> (ln 32)
......a1 was argument 0 in invocation.
......Looking up argument 0 's history
......o was parameter number: 0 to method: <C: foo()>
...

Object history for method: setA2() and variable: <A: a2>

<A: a2> is a field object. Information on all changes to this field:
...Field was changed in method: <A: setA2()>
...Field was assigned parameter 0 in method <A: setA2()>
...Getting history of parameter 0
...a was parameter number: 0 to method: <A: setA2()>
......Method: <A: setA2()> was invoked in method: <B: <init>()> (ln 19)
......a was argument 0 in invocation.
......Looking up argument 0 's history
......a2 was parameter number: 1 to method: <B: <init>()>
......Method: <B: <init>()> was invoked in method: <C: void foo()> (ln 26)
......a2 was argument 1 in invocation.
.........Looking up argument 1 's history
...........Variable is local (ln 26)
```

Figure 4.5: Object history for fields `A.a1` and `A.a2` from Figure 4.4.

To obtain the object histories of fields `A.a1` and `A.a2`, we simply query our tool in methods `setA1()` and `setA2()`, respectively. As expected, the tool recursively reports the history of events to these fields. Specifically, the tool immediately informs us that `A.a1` is

changed in methods `C: foo()` at line 29 and in the constructor of `B` at line 18. The tool also tells us that `o` was assigned to `A.a1` in method `foo()`. We choose to omit the history for `o` for conciseness.

We get a similar history for field `a2`. Specifically, we are informed that `a2` is changed in the constructor of `B` and the constructor is invoked method `foo()` in class `C`. We are also told that `A.a1` is assigned a `new Object()` in method `foo()` at line 26.

By inspecting the object histories for fields `A.a1` and `A.a2`, we know that classes `B` and `C` are dependent on `A`. Specifically, we must change code in `B`'s constructor and `foo()` in `C` to be consistent with changes we make to `A`. Thus, the scope of changing `A` is limited to modifying two methods in two classes.

Through this simple example, we have shown the utility of object histories in modifying code. In larger programs with hundreds of classes and complex call chains, finding all uses of a class can be tricky, time consuming, and lead to the creation of bugs. By using object histories, however, developers can quickly identify all the program points where a given class is used to identify dependencies between classes. The developer can then change all dependent classes accordingly. Additionally, object histories also provide a quick estimate of the size of the task by identifying program points that must be inspected to ensure that no additional bugs have been introduced.

## 4.2 Bug Fix

In this section we present the utility of object histories to fix bugs in real-world software systems. We use Gantt Project as an example to show how object histories can be used to fix bugs quickly and with minimal understanding of the source code.

As with most open source software, bugs in Gantt Project are reported by users via a public bug tracking tool. These bugs are then verified and fixed by several contributors with varying degrees of familiarity and understanding of the source code.

We choose a typical bug in Gantt Project's user interface to highlight our approach. The bug report states that when a user tries to open a Gantt project from a server, the window title is "Save to a server" instead of "Open from a server".

We begin by inspecting the main class of Gantt, `GanttProject`. We observe that it contains a field `GanttLanguage.language`, which (based on the comments) is used for internationalization (i18n). That is, the `GanttLanguage` class provides strings in the local language to be used in various parts of the program.

By looking at methods of class `GanttLanguage`, we find the method `getText()` returns the text in a local language for a given `key`. Thus, we know that this method must be

invoked to obtain text “Save to a server”. To find the program points where text was extracted, we extract the object history of `key` (shown in Figure 4.6). On inspecting the object history, we note two interesting key strings, “saveToServer” and “openFromServer”, which may lead us to the underlying problem. We show the relevant parts of the output below.

```
1:  Object history for method: getText() and variable: key
2:
3:  key was parameter number: 0 to method: getText()
4:
5:  Method: getText()> was invoked in:<...gui.GanttURLChooser:void <init>()> (ln 66)
6:  key was argument 0 in invocation.
7:  Looking up argument 0's history
8 :  ...Variable is local : (”openFromServer”) (ln 66))
9:
10:  Method: getText() was invoked in method:
11:          <...gui.GanttURLChooser: void <init>()> (ln 67)
12:  key was argument 0 in invocation.
13:  Looking up argument 0's history
14 : ...Variable is local : (”saveToServer”) (ln 67))
15:
16: Method: getText() was invoked in method:
17:          <...gui.server.ConnectionPanel: void <init>()> (ln 44)
18: key was argument 0 in invocation.
19: Looking up argument 0 's history
20 : ...Variable is local : (”openFromServer”) (ln 44))
21:
22: Method: getText() was invoked in method:<...action.GPAction: getI18n()> (ln 89)
23: key was argument 0 in invocation.
24: Looking up argument 0's history
25: ...argument 0 was parameter number: 0 to method: <...action.GPAction: getI18n()>
26: ...Looking up callers of getI18n()
27: ......Method: <...action.GPAction: getI18n()> was invoked in method:
28:                     <...action.project.SaveURLAction: getLocalizedName()> (ln 19)
29: ......key was argument 0 in invocation.
30: ......Looking up argument 0's history
31 : .........Variable is local : (”saveToServer”) (ln 19))
32: ......Method: <...action.GPAction: getI18n()> was invoked in method:
33:                     <...OpenURLAction: getLocalizedName()> (ln 28)
34: ......key was argument 0 in invocation.
35: ......Looking up argument 0's history
36 : .........Variable is local : (”openFromServer”) (ln 28))
...
```

Figure 4.6: Object history for `key` in method `GanttLanguage.getText()`.

From Figure 4.6, we note that there are two calls to `getText()` from the constructor of class `GanttURLChooser` in the `gui` sub-package. This seems to be a promising lead because we have a user interface issue, so logcially it may lie within the `gui` package. We can also see that the origin of the `key` in the constructor is either "openFromServer" on line 4 or "saveToServer" on line 10. Next, we inspect the code for `GanttURLChooser`'s constructor. The relevant portion of code in class `GanttURLChooser` is shown below.

```
1  public class GanttURLChooser extends JDialog {
2
3    public GanttURLChooser(Frame parent, boolean opening,
4      String currentURL,
5      String currentUser,
6      String currentPass) {
7
8      super(parent, opening ? (GanttProject.correctLabel(
9        GanttLanguage.getInstance().getText(
10         "openFromServer"))) : (GanttProject.correctLabel(
11         GanttLanguage.getInstance().getText(
12         "saveToServer"))), true);
13 ...
14 ...
```

From the code above, we can see that title of the super class, `JDialog`, is set to "openFromServer" or "saveToServer" depending on the value of `opening`. This is the correct behaviour, so the problem must lie in an incorrect assignment of `opening`. Thus, we extract `opening`'s object history (shown below).

```
Object history for method: void <init>()> and variable: opening

opening was parameter number: 1 to method: <init>()
Method: <init>() was invoked in method:
    <...gui.ProjectUIFacadeImpl: showURLDialog()> (ln 241)
opening was argument 1 in invocation.
Looking up argument 1 's history
...Variable is local (boolean: false)(ln 241)
```

From the last line in the output, we find that `opening` is always set to `false`, causing the open Gantt project from server window title to always be "Save to a server". We have now located the bug!

We next look at the code for the origin of `opening` in the `showURLDialog()` method (shown below). We can see `false` being passed as parameter 1 to `GanttURLChooser`'s constructor at line 7.

```
1  public class ProjectUIFacadeImpl implements ProjectUIFacade {
2
3    private Document showURLDialog(IGanttProject project) {
4      Document document = project.getDocument();
5      GanttURLChooser uc = new GanttURLChooser(
6        myWorkbenchFacade.getMainFrame(),
7        false,
8        ...
9    }
10 ...
```

To apply the fix, we can extract the object history of the parameter `project` to find that it originates from methods `openRemoteProject()` and `saveProjectRemotely()` in the same `ProjectUIFacadeImpl` class. Thus we conclude that the root cause of the bug is that the same title string is used regardless of whether a project is being opened from a server or being saved remotely. A simple fix is to change the `showURLDialog()` method to accept a boolean `opening` and pass it onto `GanttURLChooser`'s constructor. Of course, we must also change methods `openRemoteProject()` and `saveRemoteProjectRemotely()` to pass `true` and `false` respectively.

We have demonstrated the utility of object histories in finding a bug in a medium-sized, open source program. By using object histories, we were able to fix the bug with minimal understanding of the code base.

## 4.3 Benchmarks

In this section, we show feasibility of computing object histories for three open source Java projects using our implementation. The Gantt project [2] is an open source project scheduling and management tool, jGraphX [14] is a graph drawing component, and Cobertura [6] is a tool for calculating the code coverage of tests. Table 4.1 shows information about these benchmarks.

Table 4.1: Benchmark characteristics.

| | Gantt Project | jGraphX | Cobertura |
|---|---|---|---|
| Lines of code | 64,000 | 45,500 | 62,000 |
| Number of Classes | 481 | 82 | 283 |
| Number of Methods | 4,443 | 1,722 | 3,302 |

Table 4.2 shows the time required by our preprocessing phase for each of the benchmarks as well as the size of the events database. The tests were run on a desktop machine with a 2.66Ghz core 2 duo E6700 processor with 4GB RAM. We ran our analysis on a new Sun JDK, but analyzed the benchmarks as if they were running Sun JDK 1.4.2 using at most 2.5 GB of heap space. Note that times for querying are omitted because our querying algorithms run almost instantaneously.

Table 4.2: Time required for preprocessing and size of events database.

| | Gantt Project | jGraphX | Cobertura |
|---|---|---|---|
| Soot + SPARK Time (s) | 207 | 218 | 253 |
| Object History Preprocessing Time (s) | 3.6 | 1.8 | 4.2 |
| Total Time (s) | 210.6 | 219.8 | 257.2 |
| Size of Events Database (kb) | 6,126 | 1,920 | 3,663 |

From Table 4.2, we can see that computing object histories takes only 3-4 minutes on our benchmarks and the events database is only a few megabytes. Once the events database is constructed, no additional computational overhead is required to answer object history queries.

## 4.4 Object History Size Distributions

In this section we present details about the distribution of object history sizes in each of the benchmarks described in Section 4.3. We define the size of an object history as the number of events reported to the user given an object history query. We randomly selected 2,000-5,000 objects from each of our benchmarks and computed their object histories as shown in Figure 4.7.

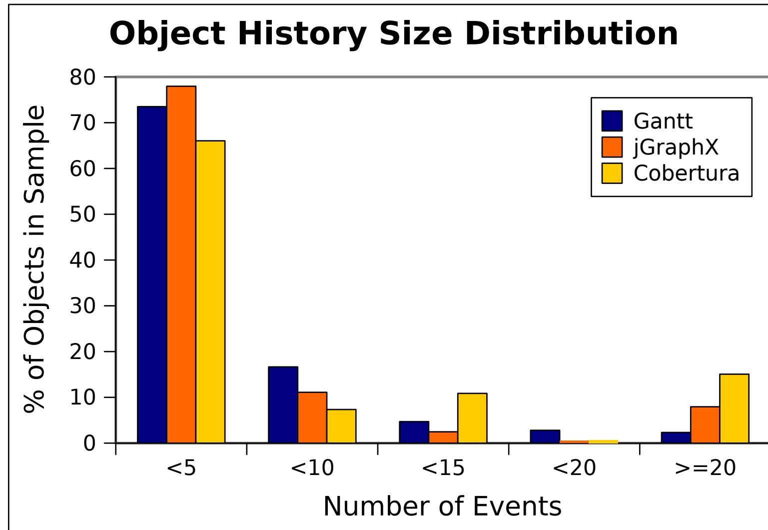


Figure 4.7: Object History Size Distribution for Benchmarks.

The majority of objects in all the projects have between 5 and 10 events in their object histories. This result is intuitive because developers often create local objects for computations and discard them as soon as they obtain the results leading to short object histories.

Interestingly, there are a small subset of objects with large histories (more than 20 events) in each project. Upon closer inspection, we found that these objects are typically parameters to methods that are called frequently in the program. User interface methods or logging methods are good examples. For instance, in Gantt, the parameter `text` in the user interface method `GanttStatusBar.setFirstText()` has a lengthy object history because `setFirstText()` is called each time Gantt's status bar text has to be updated.

## 4.5 Effect of Filtering

Recall that we group events on instance field `f` for all instances of class `C` and later apply pointer analysis based filtering to discard events that are not relevant to the query. Additionally, when extracting the history of objects assigned from an instance method invocation `a.m()`, our raw algorithm considers all paths starting with the return value of `m()`. In some cases, however, we can prune some of the paths using our filtering algorithm (one such example is shown in Figure 2.7).

In this section we show the effectiveness of our filtering technique to significantly reduce the number of events reported to the user. Figure 4.8 shows filtered vs unfiltered object

history sizes for roughly 700 randomly selected objects that were assigned from instance fields (`a = x.f`) in jGraphX. In particular, we found objects that were directly assigned from a field read for each method and then randomly decided whether to include them in our sample set.

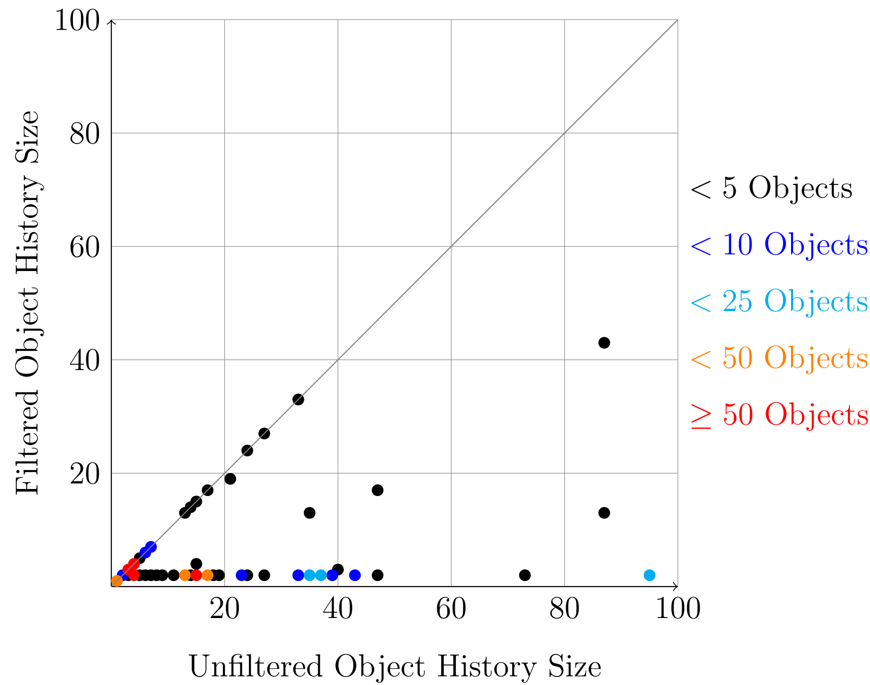


Figure 4.8: Filtered vs unfiltered object history sizes for variables assigned from instance fields in jGraphX.

Each point in Figure 4.8 represents the number of events in a target object's history. A point's `x` coordinate shows the number of events in its unfiltered object history and the `y` coordinate shows the number of events in its filtered history. The color of the point shows the number of objects that share a particular history size.

We can clearly see that filtering significantly reduces the number of events in an object's history in many cases (71.6% to be exact). The mean of the object history size drops from 47.5 events without filtering to 10.3 events with filtering. Filtering reduces the number of events reported by a factor of 4.6 in jGraphX. This result is expected because we are filtering out irrelevant events that we grouped earlier to simplify our handling of field events.

We next consider the effect of filtering in cases where an object is assigned from an instance method invocation. Figure 4.9 shows filtered vs unfiltered object history sizes for

roughly 1000 randomly selected objects that originated from instance method invocations (`a = x.foo()`) in jGraphX. As with our approach before, we simply found objects that were directly assigned from a method invocation and randomly decided whether to include them in our sample set.

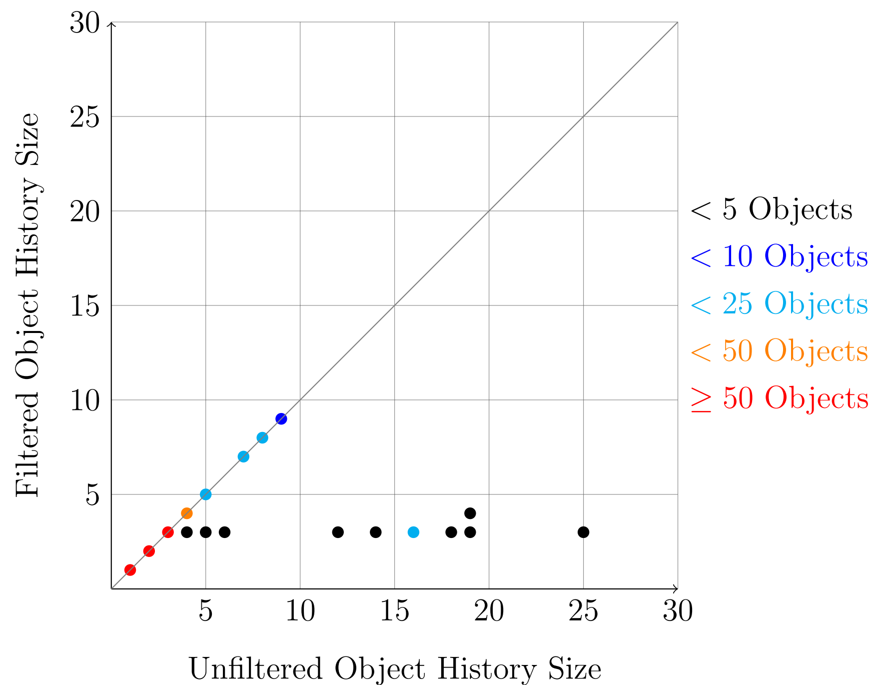


Figure 4.9: Filtered vs unfiltered object history sizes for variables originating from instance method invocations in jGraphX.

From Figure 4.9 we see that filtering reduces the size of an object's history in only some cases (9.1% of object history queries in our sample). The object history size for many objects is the same regardless of filtering. There are, however, a few objects with larger histories, where filtering makes a difference. We can see these objects towards the bottom of Figure 4.9. The mean history size without filtering is 5.9 events. By using our filtering technique, the mean drops to 2.9 evens. Filtering reduced the number of events reported by a factor of 2.03 in this case.

Table 4.3 shows the average filtered and unfiltered history sizes for objects originating from instance fields and instance method invocations in our benchmarks.

Table 4.3: Average Filtered and Unfiltered Object History Sizes.

| | **Gantt Project** | **jGraphX** | **Cobertura** |
|---|---|---|---|
| **Instance Fields** | | | |
| Average Unfiltered History Size | 34.7 | 47.5 | 14.2 |
| Average Filtered History Size | 18.9 | 10.3 | 3.6 |
| Reduction Factor | 1.8 | 4.6 | 3.9 |
| **Instance Method Invocations** | | | |
| Average Unfiltered History Size | 3.4 | 5.9 | 3.6 |
| Average Filtered History Size | 3.1 | 2.9 | 2.9 |
| Reduction Factor | 1.1 | 2.0 | 1.2 |

Based on Table 4.3, we can see that filtering makes a significant difference when objects originate from instance fields. Filtering also helps eliminate some irrelevant events for objects originating from instance method invocations in some cases, but the reduction is not as drastic.

# Chapter 5

# Related Work

We discuss three main areas of related work in this section: points-to analysis (and its applications to program understanding), program slicing, and program exploration approaches.

## 5.1 Points-to Analysis

Our approach for filtering out irrelevant events in an object history relies on good points-to analysis information. We use the default Spark implementation of points-to analysis [13], which computes call graph and context-insensitive field-sensitive points-to information for Java programs. The pointer analysis is flow-insensitive, but incorporates some benefits of flow-sensitivity.

Points-to analysis is useful for program understanding; for instance, Ghiya [8] describes how displaying the read and write sets associated with particular program points can help developers with program understanding. Object histories help make points-to information more helpful to developers by showing the evolution of abstract memory locations (points-to sets) over time.

## 5.2 Program Slicing

Object histories can answer queries about a program variable by identifying a set of program events potentially relevant to the history of that variable. Program slicing [26, 22] attempts to identify a minimal *executable* subset of the program which affects a variable at a given program point.

A key difference between our approach and program slicing is that our approach is flow-insensitive at an interprocedural level. Typical slicing approaches compute a Program Dependence Graph, or, interprocedurally, a System Dependence Graph [9] and identify all statements which exert control or data dependencies on the variable being queried. Such dependencies are paths through the PDG or SDG. Our approach instead computes a set of events in the program, organized by method, and recursively queries this set of events to build object histories. We have chosen to define an event $e$ to be relevant to a query on variable $v$ if $v$ is transitively data-dependent on $e$. Our flow-insensitive, event-based approach preserves much less state and hence makes queries significantly easier to answer: instead of needing to traverse the entire program, we can construct a list of relevant events by querying the suitably-indexed set of events from our events database.

Furthermore, because object histories focus on summarizing the behaviour of reference-typed variables, and need not deal with scalar variables (which program slicing typically devotes significant effort to understanding), we could use object-specific events such as instantiation and field reads in our summaries. Pointer analysis information is thus particularly meaningful for summarizing events. Our experience was that object histories, in conjunction with the flow-insensitive, context-insensitive pointer information, provided useful information about program behaviour.

Program slicing includes information about the base object in instance field queries. For instance, the program slice for the seed statement `x.f = a` will include information about `x` as well as `x.f`. This typically causes the slice to become unmanageably large without yielding any significant insight into statements affecting `x.f` [21]. Our approach, however, records events on `x.f` separately from `x` thereby drastically reducing the number of events reported to the user.

If a seed statement lies within a conditional block of code, program slicing also includes the conditional statement in the slice. This leads to the addition of several program statements that are not directly related to the original query. Our approach, however, ignores conditional statements and in our experience we found that this is sufficient for most practical applications.

Kaveri [20] is an Eclipse plugin providing a front-end to the Indus slicer for Java programs. Our filtering by pointer information, in particular, is similar to Indus's use of share entities [19]. Our results, however, indicate that detailed pointer analysis information eliminates much of the need for considering complex dependence graphs which summarize the whole program; instead, our object-insensitive, field-based approach combined with field-sensitive pointer analysis successfully identifies a useful set of events for understanding the behaviour of heap-manipulating Java programs.

Another important difference between our approach and typical program slicing is our special treatment of collection classes. Because Java programs use collections extensively,

encoding the semantics of collection manipulations into our algorithms enables our histories to express what happens to objects over their lifetime more succinctly than it could otherwise.

### 5.2.1 Thin Slicing

Thin slicing [21] is an improvement over traditional slicing. This approach significantly reduces the number of statements in a program slice by only considering producer statements relevant to a seed statement. Thin slicing omits slices for base objects in instance fields as well as conditional statements. Similar to our approach, context insensitive thin slicing employs pointer analysis based filtering for instance fields. Statements where the base of an instance field assignment does not alias the base of the instance field in the seed statement are omitted. The paper also proposes context-sensitive slicing, but the authors note that it was extremely expensive and did not provide much benefit.

While thin slicing is similar to our approach, there are a few important distinctions. Thin slices are demand driven and accept a collection of seed statements as input. Once the thin slice is computed, the user can specify additional statements for slicing. Using thin slicing to debug problems in a large program can be time consuming because each query takes up to 5 minutes [21]. Our two stage approach, however, preprocesses the program once and allows unlimited queries without requiring additional time.

Thin slicing only uses object sensitivity [16] to distinguish between different instances of container classes to omit statements irrelevant to the seed statement. However, by using filtering contexts we obviate the need for object sensitivity for collections without losing precision in our results. Additionally, unlike thin slicing, our filtering algorithm also provides some benefits of object sensitivity outside container classes.

Like traditional slicing, thin slicing also does not include support for handling Java collection classes and iterators. Our approach identifies read and write events on Java collections directly as well as through iterators.

## 5.3 Program Exploration Approaches

Demsky and Rinard present a technique for program exploration and understanding where object states depend on their dynamic membership in collections, or roles [5]. In their approach, a role represents a set of referencing relationships from and to a particular heap object; membership in collections is represented by inbound references to that object. They compute enhanced method interfaces by analyzing program execution traces. These

enhanced method interfaces summarize the read and write effects of methods, similar to our intraprocedural object histories.

Another event-based program exploration and verification approach is the Program Query Language [15]. As with object histories, PQL also abstracts programs into a set of events and proposes a query language over this set of events and uses pointer analysis to improve the relevance of its query results. However, PQL was designed to support queries which find application errors and security flaws, and it therefore focuses on finding all points in a program which satisfy a query (i.e. execute a certain set of events), rather than displaying the events leading up to a query consisting of a specific program point.

# Chapter 6

# Conclusions and Future Work

In this thesis, we presented the idea of object histories, how to compute them, an implementation, and experimental results. An object history for object $o$ is a set of events $e$, beginning with $o$'s instantiation, that happen to it over the course of program execution. Events in an object history include its instantiation, writes of $o$ onto a field, reads of fields, being passed to a method as an argument (including being added to removed from collections), and being returned from a method as a return value.

We compute object histories in two phases. In the first phase, we preprocess a Java program and build a database of events. The second phase queries the events database repeatedly to extract the object history of a variable $v$.

The database of events consists of two parts: 1) method summaries for all application methods, and 2) field events that are aggregated throughout the program. Method summaries are composed of an intraprocedural history analysis of the method, a list of callers, a list of invoked methods (with arguments), and the return values, if any. The intraprocedural history analysis simply finds the method level history of all variables found within a method body. Our approach for aggregating field events is field-based, meaning that we group events on field `f` of all instances of class `C` throughout the program. Of course, this leads to irrelevant events when extracting the object history of a field in a particular instance of `C`. To eliminate irrelevant events, we use a pointer analysis based filtering strategy.

To extract an object history for variable $v$ in method $m$, we start with $m$'s intraprocedural history analysis and recursively report events leading to the origin of $v$. We only report an event $e$ if it is relevant to our query for $v$. We determine relevance by intersecting points to sets (for the base in instance field or instance method invocations) in the filtering context for $v$ with the points to sets found in an event $e$. We only include an event $e$ if there is a non-empty intersection implying that the objects may alias. We determine the filtering

context to be the set of points to sets for the base in instance fields or instance method invocations, parameters to the method, arguments used in an invocation, and `this`. We update the filtering context constantly and pass it as a parameter to the recursive query to ensure that we hold onto the original context in the query $\langle v, m \rangle$.

We implemented our analysis using the Soot [24] Java bytecode analysis framework and used SPARK [13] for computing points-to sets.

We also reported experimental results for object histories. Specifically, we presented the utility of object histories in bug fixing and code change tasks. We also showed how to fix a sample bug in a medium-sized, open source Java program, Gantt Project [2]. Through this example, we demonstrated how to use object histories to fix bugs quickly with minimal understanding of the source code. We concluded the chapter with some quantitative results and discussed the distribution of object history sizes as well as the effectiveness of our filtering algorithm.

## 6.1 Future Work

Our analysis for constructing object histories is context-insensitive. Thus, when looking up the history of a parameter $p$ in a method $m$, we consider all callers of $m$. A future direction is to explore context-sensitive object histories, where we look up a caller of $m$ only if $m$ was invoked in the execution path leading to the queried program point. We believe that context-sensitive object histories will yield more specific object histories.

While our interprocedural history analysis is flow-sensitive, our interprocedural approach for computing object histories is flow-insensitive. This means that we cannot determine the order of events in an object's history and we simply report all of them. We can explore methods to compute flow-sensitive object histories which would eliminate events that happen to a variable $v$ *after* the program point where the query was made.

Another application of computing flow-sensitive object histories is to answer queries about events that happen to $v$ in the future. This set of future events would allow developers to understand how a given variable $v$ is used after a given program point. For instance, a future events query at the instantiation of $v$ could inform us about how $v$ is used throughout the rest of the program.

Finally, our implementation of a graphical user interface is stand-alone and text-based. Specifically, it is cumbersome because output must be interpreted and developers must follow program points and call chains manually. We can easily improve the user experience of our tool by integrating it into common IDEs such as Eclipse or NetBeans via a plug-in. A plug-in would make it significantly easier to query object histories directly from the

source code. We could also provide a visualization of the object history of a variable along with direct links to program statements.

# References


[1] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: principles, techniques, and tools. *Reading, MA,*, 1986. 30

[2] D. Barashev and A. Thomas. The gantt project. `http://www.ganttproject.biz`, February 2010. 4, 60, 71

[3] P.Y. Chang, E. Hao, T.Y. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 22–31. ACM, 1994. 30

[4] RA DeMilli and AJ Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991. 50

[5] Brian Demsky and Martin Rinard. Role-based exploration of object-oriented programs. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 313–324, New York, NY, USA, 2002. ACM. 68

[6] M. Doliner and SAS. Cobertura project. `http://cobertura.sourceforge.net/`, February 2010. 60

[7] M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256. ACM, 1994. 31

[8] Rakesh Ghiya. *Putting Pointer Analysis to Work*. PhD thesis, McGill University, May 1998. 66

[9] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 35–47, June 1988. 67

[10] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):337, 1992. 30

[11] W. Landi and B.G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, 1991. 30

[12] W.A. Landi. *Interprocedural aliasing in the presence of pointers*. PhD thesis, Rutgers, The State University of New Jersey, 1992. 30

[13] Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002. 31, 66, 71

[14] JGraph Ltd. jgraphx. `http://www.jgraph.com/jgraphx.html`, February 2010. 60

[15] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 365–383, 2005. 69

[16] A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(1):1–41, 2005. 68

[17] LJ Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, pages 844–857, 1990. 50

[18] S.T. Pan, K. So, and J.T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. *ACM SIGPLAN Notices*, 27(9):84, 1992. 30

[19] Venkatesh Prasad Ranganath and John Hatcliff. Pruning interference and ready dependence for slicing concurrent java programs. In *Proceedings of Compiler Construction (CC'04)*, pages 39–56, 2004. 67

[20] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using Indus and Kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9(5):489–504, 2007. 67

[21] M. Sridharan, S.J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 112–122. ACM, 2007. 67, 68

[22] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995. 66

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999. 31

[24] Raja Vallée-Rai. Soot a Java optimization framework. Master's thesis, McGill University, July 2000. 29, 31, 37, 71

[25] M.N. Wegman and F.K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991. 30

[26] Mark Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984. 66