OBJECT NAMING IN REVERSE ENGINEERING OF UML SEQUENCE DIAGRAMS

A Thesis

Presented in Partial Fulfillment of the Requirements for

the Degree Master of Science in the

Graduate School of The Ohio State University

By

Beth Harkness Connell, B.A.

* * * * *

The Ohio State University

2004

Master's Examination Committee:

Atanas Rountev, Adviser

Neelam Soundarajan

Approved by

Adviser

Graduate Program in
Computer and Information
Science

ABSTRACT

UML sequence diagrams are essential for modeling object-oriented behavior by showing in sequential order the messages exchanged among a set of objects. Reverse engineering of sequence diagrams automatically extracts the diagrams from existing code and is useful in iterative development, software maintenance, and program testing. The challenging issues involved in creating these diagrams statically are not adequately addressed by existing approaches. This thesis presents a technique for precisely modeling the run-time receiver objects for a specific category of call sites in Java programs for the purposes of reverse engineering of sequence diagrams. Our approach is based on classic data-flow analysis techniques and uses an interprocedural flow- and context-sensitive analysis algorithm. Extensive empirical studies indicate that the cost of the analysis is practical and its solution is highly precise.

ACKNOWLEDGMENTS

I would like to thank my adviser Dr. Nasko Rountev for the many things he has taught me about software engineering and working on a large research project. I would also like to thank him for giving me the opportunity to work with him on this project.

I also want to extend my thanks to the many other teachers who have brought me to this point and helped me reach my goals.

Finally, I would like to thank my family and friends for their never ending support. In particular, I would like to thank Chris and my other running friends for helping me maintain my sanity during times of high stress.

VITA

July 3, 1979	Born - Decatur, Illinois
June 2002	B.A. Mathematics and Computer Science, Kalamazoo College, Kalamazoo, Michigan
September 2002 - September 2004	Graduate Teaching Assistant, The Ohio State University, Columbus, Ohio
September 2003 - September 2004	Course Coordinator: Data Structures for Information Systems, The Ohio State University, Columbus, Ohio

FIELDS OF STUDY

Major Field - Computer and Information Science

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

LIST OF TABLES

Table **Page**

CHAPTER 1

INTRODUCTION

Iterative development is a widely adopted approach to building software systems. When developing a system iteratively, a detailed complete system design is not prepared in advance; rather, the program evolves in short iterations where analysis, design, implementation, and testing are part of each iteration. Developers design the next iteration based on their understanding of the existing implementation. In an object-oriented program, the code needing to be analyzed to gain this understanding is typically spread out over methods in multiple classes. This makes it difficult to comprehend object interactions and the overall flow of control from manually examining the source code. It is possible, due to challenges encountered during the implementation in the previous iteration, that the design artifacts constructed in the beginning of that iteration do not match the existing code. Obtaining the necessary understanding of object interactions can be simplified by automatically creating various diagrams that represent parts of the design of the existing code. Since reverse engineering builds the diagrams from the existing code, they are guaranteed to reflect the up-to-date implementation. Diagrams, easy to understand due to their visual nature, can be used as the starting point for the next designing phase.

The Unified Modeling Language (UML) is a group of graphical notations providing a visual description of software systems [5]. UML *sequence diagrams* are essential for modeling object-oriented software by showing in sequential order the messages exchanged among a set of objects [16, 9]. Sequence diagrams also assist by showing the flow of control for a scenario within a system, by using interaction frames [9] for displaying if-then conditions, looping blocks, and other control flow information. Figure 2.2 shows two examples of UML sequence diagrams, each with an interaction frame labeled `opt` representing that the messages sent within the frame are only executed if some condition is true.

Reverse engineering of sequence diagrams automatically extracts the diagrams from existing code. Not only is this necessary for iterative development, but it is beneficial for software maintenance. As object-oriented languages grow in popularity, the number of software systems implemented in these languages greatly increases. Considerable resources are invested into such systems making them essential and worth maintaining. Due to employee turnover it may not be possible to gain the assistance of the original designers and programmers in performing maintenance, but an understanding of the existing software implementation is required for modifications. Significant periods of time can easily be wasted attempting to gain insight from investigating the source code. Alternatively, understanding of object interactions and control flow for software maintenance can be gained by reverse-engineered sequence diagrams.

Reverse engineering of sequence diagrams is also beneficial for program testing. UML models provide a rich source of test design information [2]. The first step in

designing tests for a system is to identify, model, and analyze the system's responsibilities. Reverse engineering of UML diagrams automates part of this step by modeling the behavior of the system being tested. Object interactions are complex, and it is likely that defects relating to such interactions will occur in a system. For example, messages can be passed to the wrong objects or to objects that have already been destroyed. Sequence diagrams model object interactions, helping to highlight various aspects of these interactions that should be the target of testing.

Existing tools that provide support for reverse engineering of UML do so mainly for class diagrams. Class diagrams show the static structure of a software system. The information for constructing these diagrams is easily extracted from the code. The challenge of extracting the interactions displayed in sequence diagrams has resulted in thus far inadequate tools for static reverse engineering of sequence diagrams. The need is important enough that a request to tool vendors for tool support for reverse engineering of sequence diagrams is included in one popular software development book [7].

1.1 Object Naming in Sequence Diagrams

To reverse engineer UML sequence diagrams, the following important question must be addressed: how should the object(s) referred to by the variable `x` at the call site `x.m()` be represented in the diagram? An answer to this question defines an *object naming scheme* for representing potential run-time receiver objects, by creating a mapping between these objects and the objects represented in a sequence diagram.

1.1.1 Naming Based on Variable Names

A possible naming scheme is to represent the object by the variable name used in the invocation expression. The scheme used to name objects in reverse-engineered sequence diagrams generated by the commercial Borland Together ControlCenter modeling tool is not published, but this simple approach appears to be the one in use in the tool. Figure 2.2(a) was constructed from the code in Figure 2.1 using the reverse engineering functionality of the ControlCenter tool. To simplify the presentation of the diagram, the visual representation was modified without changing its meaning. This naming scheme is misleading in two major ways. The first, demonstrated by the comparison of the two diagrams in Figure 2.2, is that one run-time object can be represented by multiple diagram objects. For example, the objects labeled `a`, `c`, `f`, and `g` in the ControlCenter diagram actually represent a single run-time object, labeled `a` in the second diagram. The redundant objects result in an unnaturally large sequence diagrams and incorrectly show that messages are sent to different objects when at run time the messages are actually sent to only one object. The other misleading result of this naming scheme occurs when a single variable name is used for multiple run-time objects. For example, if a variable `x` of class `X` is assigned a newly instantiated object after a call `x.m()`, and then method `m` is again invoked upon variable `x`, the sequence diagram constructed with the simple scheme will show method `m` being invoked twice upon the same object, when in actuality it is not possible for both calls to have been made upon the same object. Similarly, the same diagram object can sometimes be used to represent objects that are possibly the same at run time, but not guaranteed to be the same. This is exemplified in the ControlCenter diagram with the calls to `p7` and `p8`. These calls have the same run-time receiver object only if the condition `b > 0`

in the running example is false. The deficiencies of sequence diagrams produced with this naming scheme prompted us to consider some of the challenging issues raised in defining a better naming scheme. The result of this work is the object naming analysis described in this thesis.

1.1.2 Singleton Call Sites and Equivalence Relations

In attempting to create a scheme for representing the run-time receiver objects at a given call site, we recognized the need to distinguish between sites that have multiple possible run-time receiver objects and sites that have only one. Informally, a call site that has only one run-time receiver object every time the site is executed is a *singleton call site.* The advantage of this distinction is that representation of receiver objects at singleton sites in reverse-engineered diagrams is straightforward, because a single diagram object is guaranteed to represent possible run-time objects precisely. If it is possible for a call site to have more than one receiver object at run time, then the site is non-singleton and there is no guarantee of straightforward representation. Examples that illustrate this distinction and a more formal definition of singleton call sites are provided in Chapter 2.

In addition to the analysis identifying singleton sites, equivalence relations are found between these sites. The sites are singleton so there is only one possible run-time receiver object for each. If the run-time receiver at one call site is guaranteed to be the same as the object at another, then the sites are equivalent. Determining these equivalence classes is important because if call sites are equivalent, they are invoked on the same run-time object and in the reverse-engineered diagram the messages should be represented as being sent to the same diagram object.

1.2 Analysis and Evaluation

The algorithm presented in this thesis provides a way to represent run-time receiver objects at singleton call sites by identifying such sites and grouping them into equivalence classes. To design the algorithm, we started by defining a data-flow problem similar to the constant propagation problem. Chapter 3 presents the lattice and transfer functions for this problem along with a brief overview of the flow- and context-sensitive data-flow algorithm designed to find a precise solution to the problem.

The rest of the thesis is organized as follows. Several enhancements to the analysis are introduced in Chapter 4; these enhancements include a generalization for handling fields, as well as consideration of some non-singleton call sites. Empirical studies are presented in Chapter 5, in order to evaluate the precision of the analysis, the effect of the enhancements, and the running time of the analysis algorithm. These results indicate that the analysis is practical and achieves high precision. Related work is discussed in Chapter 6. Chapter 7 proposes future work and concludes the thesis.

The contributions of this thesis include:

- A naming scheme for singleton call sites
- A set of refinements for the naming scheme
- Experimental evaluation of the cost and precision of the object naming analysis
- Experimental evaluation of different parameters of the analysis

CHAPTER 2

PROBLEM DEFINITION

The analysis described in this work is part of the RED tool for reverse engineering of sequence diagrams. The goal of this project is to provide a quality tool for reverse engineering of UML 2.0 sequence diagrams from Java code. RED takes as input from the user a set of Java classes that form the *component under analysis* (CUA). All of the classes that are transitively referred to by classes in the CUA are automatically added to the input. The tool first builds a call graph for the component using the Andersen-style points-to analysis from [13]. This analysis requires a complete program so if a `main` method is not included in the CUA, the fragment analysis approach from [14] is used to adapt the whole-program points-to analysis. Additional static analyses performed by the tool include call chain analysis [12] and control flow analysis [15].

As input, the user of RED also selects a method `m` from the component. We will refer to `m` as the *start method.* The user selects this start method indicating his/her desire for a sequence diagram representing the possible sequences of run-time events when the method is invoked. The construction of this diagram raises the question about how to represent objects invoked at call sites. Suppose in Figure 2.1 that method `m` in class `A` is selected as the start method. Assume that the CUA includes class `X` and class `A`. For simplicity also assume that methods `p1` through `p8` in class

```
class X { ...}
class A {
   public void m(X a, int b){
      a.p1();
      X c = this.m2(a);
      c.p4();
      X d = this.m4();
      d.p6();
      X e = d;
      if (b > 0){ e = new X(); e.p7(); }
      e.p8();
   }
   public X m2(X f){ f.p2(); X q = this.m3(f); return q; }
   public X m3(X g){ g.p3(); return g; }
   public X m4() { this.fld.p5(); return this.fld; }
   private X fld = new X();
}
```

Figure 2.1: Running example

X do not make any calls. Figure 2.2 shows two possible sequence diagrams for this method: the first using the naming scheme based on variable names (as discussed in Section 1.1.1), and the second using the naming scheme proposed in our work.

Clearly representing objects by the variable name used in the invocation expression can be confusing and imprecise. Use of such a scheme requires the programmer or tester examining the diagram to invest considerable time attempting to recover the true nature of the object interactions by examining the source code. Since the incentive for reverse engineering of sequence diagrams was to ease understanding of object interactions, a diagram that does not clarify object interactions is inutile. Instead we propose using a naming scheme based on an *interprocedural data-flow analysis* which tracks the flow of object references.

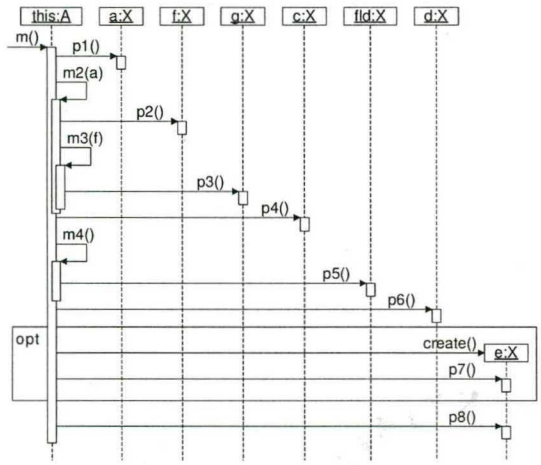
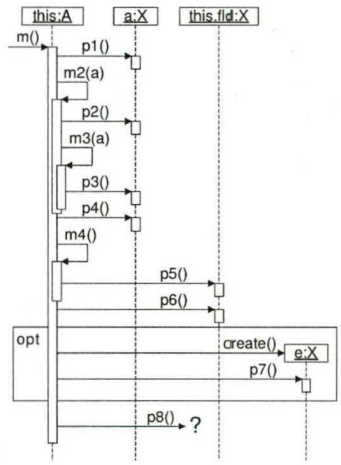


Figure 2.2: (a) Naming in ControlCenter



(b) Naming in our approach

2.1 Interesting Call Sites

Consider again the code in Figure 2.1 with `m` as the start method. The sequence diagram representing the potential sequences of run-time events that can result from invoking `m` must show the messages exchanged within the CUA along all call chains starting at the start method. An example of such a chain is "`m` calls `m2` which calls `m3` which in turn calls `p3`". Each of these calls must be represented in the diagram so it is at these call sites that we are interested in determining how to represent the run-time receiver objects. These *interesting call sites* only belong to call chains that remain within the CUA because method calls that take place in methods outside the CUA are not of interest to the user and therefore do not need to be represented in the diagram. The last method in the call chain may be outside the component, since a call to it would take place in the component. All the call sites in the running example

are interesting according to this definition because both class `A` and class `X` are in the CUA.

2.2 Singleton Allocation Sites

When a start method `m` is invoked, it is done so with respect to an initial set of objects O_i. If `m` is non-static, the receiver object is part of the set. All other objects in existence at the method invocation, including the objects passed to the start method and the objects their fields refer to, are also in the set of objects O_i. An invocation of `m` results in a sequence s_k of run-time events from from the moment `m` is invoked until it returns to its caller. Let $S_i = \{s_1, s_2, \ldots, s_n\}$ be the set of possible sequences of run-time events from invoking the start method `m` with respect to O_i. Even though the objects initially passed to the start method are set, it is possible for more than one sequence to belong to S_i. For example, a start method that takes primitive parameters affecting the flow of control clearly has multiple possible sequences for a given set of initial objects. This is the case in the running example in Figure 2.1, where there are two possible sequences of events for some O_i. Which sequence is executed depends on whether the integer passed to the start method `m` is positive.

A receiver object at a call sites may be in the set O_i of initial objects. The alternative is that the object is created at an allocation site in the sequence of run-time events leading up to the call site. Allocation sites can create multiple objects if they are in a loop or otherwise occur multiple times along a sequence s_k. We start by distinguishing between such allocation sites and allocation sites that represent the creation of an unique object in the execution of the start method and its transitive callees.

Definition 1 *Let $S_i = \{s_1, s_2, \ldots, s_n\}$ be the set of possible sequences of run-time events from invoking the start method* `m` *with respect to O_i.* ***An allocation site*** `a` ***is singleton with respect to an initial set of objects*** *O_i for the start method if for every $s_k \in S_i$,* `a` *occurs in s_k at most once.*

Statement `e = new X()` in Figure 2.1 satisfies this property for start method `m`. The statement is executed once in the sequence of events that occur when `b>0` and is not executed in the sequence when `b <= 0`.

2.3 Singleton vs. Non-Singleton Call Sites

Some call sites can have multiple possible run-time receiver objects. We distinguish between these call sites and sites for which there is only one possible run-time receiver object, because the representation is straightforward if there is only one possible receiver object.

Definition 2 *Let $S_i = \{s_1, s_2, \ldots, s_n\}$ be the set of possible sequences of run-time events from invoking the start method* `m` *with respect to O_i. A call site* `c` *is* ***singleton with respect to an initial set of objects*** *O_i for the start method if:*

1. *there does not exists a sequence $s_k \in S_i$ such that* `c` *has more than one run-time receiver object $o_{k,c}$ in s_k, even if c is executed multiple times in s_k*
2. *and if for all pairs of distinct sequences s_k, $s_l \in S_i$ and their respective run-time receiver objects $o_{k,c}$ and $o_{l,c}$ for the call site* `c`*, one of the following holds:*
   - *$o_{l,c}$ is the same object as $o_{k,c}$ and this object is in O_i*
   - *$o_{k,c} \notin O_i$ and $o_{l,c} \notin O_i$ are objects created by the same allocation site* `a` *such that* `a` *is a singleton allocation site for O_i*

For example, in Figure 2.3, the call site `b.p()` in method `n` is singleton for start method `method1` with respect to an initial set of objects where the object passed to

```
class W {
   public void method1(B b){
      this.n(b);
      this.n(b.p1());
   }
   public void method2(B b, int x){
      if (x>0) this.n(b.p2());
      else this.n(b.p3(x));
   }
   public void n(B b){ b.p(); }
}
```

```
class B {
   public B p1(){ return p2(); }

   public C p2(){ return new C(); }

   public C p3(int x){ return p2(); }

   public void p(){...}
}
```

```
class C extends B {
   public B p1(){ return this;}

   public C p3(int x){
      if(x==0) return p2();
      else return new C();
   }
}
```

Figure 2.3: Non-singleton call site `b.p()`

`method1` through formal `b` is an instance of class `C`. There is only one possible sequence of run-time events and for that sequence the only possible receiver object at the call site both times the site is executed is the instance of class `C` initially passed to the start method. On the other hand, if the object is an instance of class `B`, then the site where method `p` is called is not singleton because the first condition of the definition is not met. Both the object passed to the start method and the object allocated by the `new` expression in method `p2` are possible run-time receiver objects for the call to `p`.

Consider another example from Figure 2.3. Let `method2` be the start method, and examine again call site `b.p()`. If the object passed to `method2` is an instance of class `C` in the initial set of objects, then the first condition is met but the second is not. For sequences where integer `x` is greater than or equal to zero, the receiver object is created by the allocation expression in method `p2`. For all other sequences the receiver object is created by the allocation expression in method `p3` cf class `C`. Both allocation expressions create a new instance of class `C` upon which the method at the call site is invoked, but in order for the site `b.p()` to be singleton, the receiver objects in all sequences must be created by the *same* allocation expression. This is the case for start method `method2` with respect to object sets O_i where the parameter object is an instance of class `B`. For all possible sequences of run-time events the receiver object of `b.p()` is the object created by the allocation statement in method `p2`, which is a singleton allocation site. Since the call site is only reached once in any possible sequence, the first condition is also met and the site is singleton with respect to the object sets described.

> **Definition 3** *A call site `c` is singleton for a given start method if it is singleton with respect to all potential initial sets of objects for that start method.*

The call site `b.p()` in Figure 2.3 is clearly not singleton for either `method1` or `method2` as start methods because for neither is the site singleton with respect to all initial sets of objects. The call site is not singleton for start method `method1` with respect to sets where the object passed to the start method is an instance of class `B`. The site is not singleton for start method `method2` where the object passed to the method is an instance of class `C`.

Returning to the running example found in Figure 2.1: all call sites are singleton with `m` as the start method except `e.p8()`. This site is not singleton because the second condition of Definition 2 is not met. For any set of initial objects, the run-time receiver object is created by the singleton allocation expression `e = new X()` for the sequence of events that occur when `b>0`, but for the sequence of events when `b<=0` the receiver object is a member of the initial set of objects.

We distinguish between singleton and non-singleton call sites because in reverse-engineered diagrams receiver objects at singleton call sites can be represented precisely, whereas the representation at non-singleton call sites is not as straightforward. As shown in Figure 2.2, if a site is singleton, it is simple to represent the receiver object as the only possible run-time object. Since there is more than one possible run-time receiver object for a non-singleton site, diagrams accurately reflecting all possible object interactions would need to show at such a site that more than one receiver object exists, and possibly even show which objects are potential receiver objects.

2.3.1 Representation of Non-Singleton Call Sites

For modeling non-singleton call sites we considered several possible representations based on notations used in UML. The first representation is simple: show a message to only one of the possible receivers. This is what is done in Figure 2.2(a). Although the call to `p8()` has two possible receiver objects, only one object is shown in the diagram to be a receiver. While simple, this representation does not accurately reflect all the interactions of the objects. Information about the existence of other objects is lost so a person examining the diagram has no indication that there are possibly

different run-time objects exchanging messages. The person would have to discover this from investigating the source code. Since the representation of non-singleton sites is not differentiated in from the representation of singleton sites, the user would have to waste time examining the code to understand the behavior not only at the non-singleton sites, but also at the singleton sites to verify that they are singleton and accurately represented in the diagram.

A second idea for modeling the behavior at a non-singleton call site is to introduce a "helper" object that represents all of the possible receivers. This approach is an improvement over the first approach because no information is lost. The person examining the diagram can see there are multiple possible run-time receiver objects and which objects they are from looking at the diagram. The drawback to this approach is that it is not visually intuitive for the user. To see all of the messages exchanged by a single object `this.fld:X` the user cannot look simply at the lifeline of that object in the diagram, but must also look at all of the "helper" objects to see if `this.fld:X` is one of the possible receivers being represented.

Another approach is to show multiple messages: one for each possible run-time receiver object. In Figure 2.2(b), for the non-singleton call site `e.p8()` in the running example, one message would be shown sent to the object labeled `this.fld:X` and another shown sent to the object labeled `e:X`. This approach better models the behavior of each object by clearly showing the user all the message exchanges in which an object is involved. The problem is that non-singleton call sites can have many possible receiver objects so the diagrams resulting from using this approach could be verbose, making them hard to understand.

A fourth approach is to create a separate diagram for each alternative. A major advantage of this approach is that each diagram will accurately represent one possible execution of the system. Unfortunately, the number of diagrams generated using this approach could be quite large. For example, if there are two non-singleton call sites, one with four possible run-time receiver objects and the other with two, eight diagrams would be generated to model each alternative using this approach.

There is no straightforward approach for modeling the behavior at non-singleton call sites. Each of the representations discussed here has potential advantages and disadvantages. Due to the complexity of handling non-singleton sites, we decided to consider the question of how to represent run-time receiver objects at a call site in two stages. Once singleton sites are identified, their representation is straightforward; thus, we started by defining techniques for singleton sites. This includes identifying singleton sites and creating appropriate diagram objects for them. This work is described in this thesis. The second stage is to deal with non-singleton sites. This work is currently in progress, and except for a few special cases of non-singleton sites handled by enhancements in Chapter 4, it is not discussed further.

2.4 Equivalence Classes

Each singleton call site has exactly one run-time receiver object. Since only one diagram object should be used to represent each run-time object, we need to know all call sites for which a given object is the receiver object. We do this by first defining what it means for singleton call sites to be equivalent.

> **Definition 4** *Two singleton call sites* `c1` *and* `c2` *are equivalent with respect to a given initial set of objects for start method* `m` *if, for each possible sequence* $s_k \in S_i$, `c1` *and* `c2` *are always executed with the same run-time receiver object in* s_k.

Consider the example in Figure 2.3. For the invocation of start method `m1` where the object passed in is an instance of `C`, singleton call sites `b.p1()` and `b.p()` are equivalent.

> **Definition 5** *Two singleton call sites `c1` and `c2` are equivalent for a given start method if they are equivalent for all possible initial object sets for that method.*

While `b.p1()` and `b.p()` are equivalent for one invocation context of `m`, they are not equivalent because they are not equivalent for all possible object sets. The call site `b.p()` is not even singleton for all object sets. Call sites that are equivalent can be grouped together to form equivalence classes of singleton call sites. For start method `m` in the example in Figure 2.1, there are four equivalence classes:

`{this.m2(), this.m3(f), this.m4()}`

`{a.p1(), f.p2(), g.p3(), c.p4()}`

`{this.fld.p5(), d.p6()}` and `{X(), e.p7()}`

Each equivalence class corresponds to a diagram object in Figure 2.2(b). In general, a static analysis that identifies singleton call sites and partitions them into equivalence classes provides a naming scheme for singleton call sites, where each equivalence class is mapped to a different object in the reverse-engineered diagram.

CHAPTER 3

DATA-FLOW PROBLEM

Data-flow analysis is a classic form of static analysis of program properties. Equations are used to relate the collected information for different program points. Data-flow analysis has frequently been used for code optimization. There are many well known data-flow problems including finding reaching definitions, determining available expressions, and calculating the propagation of constant values. The goal of our data-flow problem, which is defined in this chapter, is to find the *sources* of receiver objects at call sites. If there is only one possible source for the receiver object at a site, the call site is singleton. The problem of finding the sources of receiver objects was inspired by the traditional constant propagation problem due to the similarities between the two. Constant propagation is used to determine expressions that definitely have the same value, a constant value, along all execution paths. Similarly, our analysis determines variables that definitely refer to the same unique object along all execution paths. Specifically, the variables that we are interested in are the variables through which calls are made.

As previously stated, our approach to this problem is to identify the sources of receiver objects at calls sites. The two common categories of sources discussed in this chapter are the formal parameters of the start method and singleton allocation

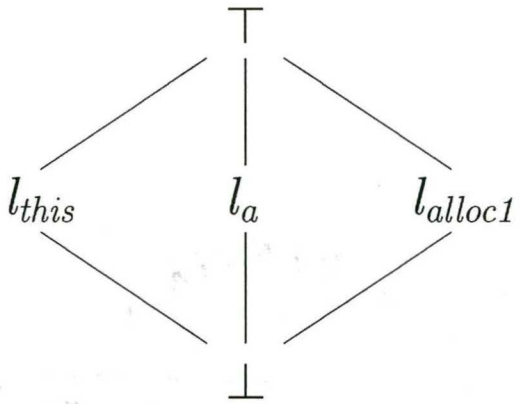


Figure 3.1: Lattice

sites. For example, for start method `m` in Figure 2.1, there are two formal parameter sources: the parameter `a` and the implicit formal `this`. If the only source of a receiver object at a call site `c` is the formal f_j, then `c` is a singleton site at which the receiver object is guaranteed to be the same object that f_j refers to at the invocation of the start method. There is also a singleton allocation site source in Figure 2.1: statement `e = new X()`. A discussion of other types of sources is presented in Chapter 4.

3.1 Data-Flow Lattice

We start defining the data-flow problem by defining a lattice of values. The possible sources of receiver objects are included as lattice elements, where each singleton allocation site and each reference formal of the start method is represented by a distinct lattice element. Additionally, the lattice contains a top element $\top$ and a bottom element $\bot$. The lattice for our running example is shown in Figure 3.1, in which the lattice element l_{this} corresponds to the implicit formal `this`, l_a corresponds to the formal `a`, and l_{alloc1} corresponds to the singleton allocation site `e = new X()`.

The goal of the analysis is for lattice elements to be associated with program variables. At a call site where the call is through a variable `v`, if the lattice element l_f corresponding to the formal f is associated with `v`, then the call site is singleton and variable `v` points to the object that `f` refers to when the start method is invoked. If instead $\perp$ is associated with `v`, then `v` can refer to more than on object, meaning the call site is not singleton. The element $\top$ associated with a variable means that the variable has not been initialized and therefore has no value. In Java, since a call through a variable `v` can only be made after the variable has been initialized, at no call sites through `v` will $\top$ be the associated lattice element in the final solution of the data-flow problem.

Consider again the running example in Figure 2.1. The final solution computed associates l_a with `a`, `c`, `f`, and `g` for each call through these variables. This means that the only possible run-time receiver object at any of these call sites is the object the formal `a` refers to at the invocation of the start method. This indicates that these call sites are singletons and form an equivalence class.

In the same example, at the call site `e.p7()` the lattice element l_{alloc1} is associated with variable `e`. This means that this particular call site is singleton, but it does not indicate that all call sites through the variable `e` are singleton. In fact, the lattice element $\perp$ is associated with the same variable `e` at the call site `e.p8()`, indicating that this site is non-singleton.

As shown in the lattice in Figure 3.1, the partial order in the lattice is $\perp \leq l_i \leq \top$. Let x and y be distinct lattice elements. The meet operation $\wedge$ is defined as follows:

$$x \wedge \perp = \perp, \qquad x \wedge \top = x, \qquad x \wedge x = x,$$

$$\text{and if } x \neq \top \text{ and } y \neq \top \text{ then } x \wedge y = \perp.$$

The meet operation is used in the analysis to merge the information gathered along the paths corresponding to potential paths of execution. For instance, in the running example, there are two paths that reach the statement `e.p8()`, so the information about which lattice elements are associated with the program variables is merged immediately preceding the statement.

Consider a location in the program where two paths merge. If a variable `v` refers to one object along one execution path and another object along the other path, then `v` will be associated with two distinct lattice elements before the merge and the last rule ($x \wedge y = \bot$) for the meet operation applies. It is logical in this situation that after the merge $\bot$ be the lattice element associated with `v`, because at run-time `v` could refer to either object.

Thus far in this discussion, the meet operation has been used to merge two lattice elements associated with a single program variable. In practice, at any program point a set of variables V exists where each $v \in V$ has a lattice element associated with it. In other words, at each program point we associate a map $S_n : V \to L$ where L is the lattice. For each variable $v_i \in V$, $S_n(v_i)$ is the lattice element associated with v_i in S_n. To merge the information gathered along two paths where S_a is the map from one path and S_b is the map of the other, the meet of the two maps $S_a \wedge S_b$ needs to be defined. The meet of S_a and S_b is defined as

$$(S_a \wedge S_b)(v_i) = S_a(v_i) \wedge S_b(v_i) \text{ for all } v_i \in V$$

3.2 Control-Flow Graphs

A control-flow graph (CFG) is used to represent the flow of control between different parts of the code. CFG nodes represent program statements, and CFG edges

represent the flow of control between statements. Consider the control-flow graphs for all methods that are reachable in the call graph from the start method. Let V be the set of all reference-typed formal parameters and reference-typed local variables. We consider the following categories of nodes:

- $v_1 = v_2$, where $v_1, v_2 \in V$ and $v_1 \neq v_2$
- $v_1 = v_2.fld$, where $v_1, v_2 \in V$ and fld is an instance field
- $v_1.fld = v_2$, where $v_1, v_2 \in V$ and fld is an instance field
- $v = X.fld$, where $v \in V$, X is a class, and fld is a static field
- $X.fld = v$, where $v \in V$, X is a class, and fld is a static field
- $v_1 = v_2[i]$, where $v_1, v_2 \in V$ and i is an integer
- $v_1[i] = v_2$, where $v_1, v_2 \in V$ and i is an integer
- $v = new\ X$:, where $v \in V$ and X is a class. We assume that such a statement represents *only* the allocation of heap memory, but not the invocation of the corresponding constructor. The constructor call is treated as a separate statement $v.X(\ldots)$.
- c or $v = c$, where $v \in V$ and c is a call expression
- *return* v, where $v \in V$
- branch node: e.g., the condition of an `if`, `switch`, or `while` statement. We assume that the condition does not have side effects — i.e., no values are changed when the condition is evaluated. Only a branch node can have multiple CFG successors.
- declaration node: e.g., $X\ v$ where $v \in V$ and X is a type
- irrelevant node: e.g., $i = 5$ where i is a variable of primitive type

It is only necessary to consider these categories of nodes, because a more complicated statement can be broken down to a sequence of simpler statements where each fits into one of the categories. This can be done by introducing temporary variables. For example, the complex statement `if(m().fld)` where the call to `m` returns an object of class `X` can be rewritten as the sequence of statements fitting into the stated categories:

`X v;`	declaration node
`v = m();`	$v = c$, where c is a call expression
`boolean b;`	irrelevant node
`b = v.fld`	irrelevant node
`if(b)`	branch node

An instance call expression is of the form $v_0.m(v_1, \ldots, v_n)$, where $v_i \in V$ for all i such that $0 \leq i \leq n$. A static call expression is $m(v_1, \ldots, v_2)$, where $v_i \in V$ and m is a static method.

3.3 Transfer Functions

We associate a map $S_n : V \to L$ with each CFG node n where, once again, V is the set of all reference-typed formal parameters and local variables and L is the lattice described in section 3.1. If $S_n(v)$ is some lattice element other than $\top$ and $\bot$, then v is guaranteed to refer to the unique object corresponding to that element immediately before the execution of n. If $S_n(v) = \bot$ then the analysis was unable to determine that v refers *only* to a particular object represented by a single lattice element. In the beginning of the analysis $S_n(v) = \top$ for all n and v, indicating that no information is currently known.

We represent the effects of program statements on maps of variables to lattice elements with *data-flow transfer functions*. For each CFG node n, the analysis associates a function $f_n : (V \to L) \to (V \to L)$. The map S_n provides information about the values of variables immediately preceding the node n, so the map $f_n(S_n)$ shows the values immediately after n. For any map $S : V \to L$, we will use the notation $S[v \mapsto l]$ to denote a new map where the only change is to the value associated with $v \in V$, which is changed to $l \in L$. The transfer functions are as follows:

- for $v_1 = v_2$: $f_n(S_n) = S[v_1 \mapsto S_n(v_2)]$

- for $v_1 = v_2.fld$: $f_n(S_n) = S_n[v_1 \mapsto \bot]$
- for $v_1.fld = v_2$: $f_n(S_n) = S_n$
- for $v = X.fld$: $f_n(S_n) = S_n[v \mapsto \bot]$
- for $X.fld = v$: $f_n(S_n) = S_n$
- for $v_1 = v_2[i]$: $f_n(S_n) = S_n[v_1 \mapsto \bot]$
- for $v_1[i] = v_2$: $f_n(S_n) = S_n$
- for $v = new\ X$:
  - $f_n(S_n) = S_n[v \mapsto l_{alloc}]$ if this is a singleton allocation site with a corresponding lattice element l_{alloc}
  - $f_n(S_n) = S_n[v \mapsto \bot]$ otherwise

For an assignment $v_1 = v_2$, the analysis propagates the current value of v_2 to v_1. If v_1 is assigned the value of a field or a value stored in an array, $\bot$ is propagated because a conservative assumption is made that *any* object reference could be assigned to v_1. In Chapter 4 we discuss an approach for more precise handling of fields.

3.4 Interprocedural Control-Flow Graphs

The handling of calls requires the introduction of an *interprocedural CFG* (ICFG) [18], in which the method-level CFGs are linked through interprocedural edges. Entry and exit nodes are added to each method's CFG, and within CFGs each node representing a call site is split into two nodes: a call node and a return node. These two nodes are connected by an intraprocedural edge. For each method `m` that could be called by call site `c`, an interprocedural edge connects the call node for `c` with the entry node of the method's CFG. Likewise, another edge connects the exit node of method `m` with `c`'s return node. Transfer functions are associated with the edges to entry nodes to represent the effects of parameter passing and with the edges from exit nodes to represent the effects of return values. There are also transfer functions

for the edges connecting a call site's call node to its return node for the handling of local variables that are not modified by the method calls.

Thus far, all of transfer functions have been for nodes with one incoming edge and one outgoing edge. Our notation has been that S_n provides the information about the values of variables immediately preceding the node n, so $f_n(S_n)$ is the map showing the values immediately after n. A call node has at least two outgoing edges: one to the return node and another to the entry node of the method being called. There are even more outgoing edges if there are multiple possible target methods. Some nodes also have more than one incoming edge. Like a call node, a return node has at least two incoming edges: one from the call node and another from the exit node of the called method. These nodes with multiple incoming and outgoing edges require more detailed notation to explicitly describe the changes to the values of variables. The function $f_{n,m} : (V \to L) \to (V \to L)$ is associated with call/exit node n's outgoing edge to node m. This means if S_n provides the information about the values of variables immediately preceding the node n, then the map $f_{n,m}(S_n)$ shows the values immediately after node n along the edge to node m.

Method calls can be made to static or non-static methods. Consider first a method call $v_0.m(v_1, \ldots, v_n)$ to a non-static method m with either a void return type or a primitive return type. Since we are not interested in the values of primitives, a call to a method returning a primitive is treated the same as a method not returning anything. Suppose there is only one possible target method m for this call and that the formal parameters of m are $w_0, \ldots, w_n$, where w_0 is the formal `this`. Let node c be the call site's call node, node r be the site's return node, node *entry* be the entry node of the target method, and node *exit* be the exit node of the target method. The

map $\top_{map} : V \to L$ maps every variable in V to $\top$. The notation $\top_{map}[v \mapsto l]$ is used to denote that variable v is mapped to l and all other variables in V are mapped to $\top$. The transfer functions for the nodes involved in this call are as follows:

- $f_{c,entry}(S_c) = \top_{map}[w_0 \mapsto S_c(v_0)] \ldots [w_n \mapsto S_c(v_n)]$
- $f_{c,r}(S_c) = S_c$
- $f_{exit,r}(S_{exit}) = \top_{map}$

As shown in the first of these transfer functions, the effect of passing parameters is similar to assignments, where the formal parameter is assigned the value of the actual parameter.

No object reference is returned by the call to method m, so the mapping of variables to lattice elements immediately following the return node of the call site r should be no different than the map of values before the call. To obtain the mapping of variables to lattice elements for the return node, the meet operation is used. The map resulting from $f_{c,r}(S_c) \wedge f_{exit,n}(S_{exit})$ provides the values of variables immediately following the return node. In this case, the map for the call node along the edge to the return node is the same as the map for the node preceding the call node, and the exit node of the called method maps every variable to $\top$. Since any element x met with $\top$ is x, the meet results in every variable being mapped to the same value it was mapped to before the method call.

Consider now a call to a static method with a primitive or void return type. The only difference is in the $f_{c,entry}$ transfer function. Since the method is not called on an object the method call is of the form $m(v_1, \ldots, v_n)$ and in the called method there is no formal w_0 for `this`. The transfer function for the static method call is as follows:

- $f_{c,entry}(S_c) = \top_{map}[w_1 \mapsto S_c(v_1)] \ldots [w_n \mapsto S_c(v_n)]$

Let us now consider more complicated situations where the methods return object references. For each method m_i with a reference return type, a "fake" return variable $ret_i \in V$ is created to represent the return value of the method. In the case of a call to a non-static method returning an object reference, the method call to m_i is of the form $r = v_0.m(v_1, \ldots, v_n)$. The $f_{c,entry}$ function is the same as the transfer function for the non-static method call with a void return type, but the other two functions are different:

- $f_{c,r}(S_c) = S_c[r \mapsto \top]$
- $f_{exit,r}(S_{exit}) = \top_{map}[r \mapsto S_{exit}(ret_i)]$

These two transfer functions are also used for static method calls where an object reference is returned in combination with the same function $f_{c,entry}$ that is used for static method calls with void return types. The notation $\top_{map}[v \mapsto l]$ in the second function denotes a new map where all variables in V are associated with $\top$ except v which is associated with $l \in L$.

The first of these transfer function shows that the intraprocedural flow of information does not tell us anything about the value of r after the call since it is assigned the value returned by the method call. The variable r is assigned the value $\top$ so that when the meet operation is performed, the value associated with r in the map for the return node will end up being the same value v is mapped to in the return node. The method call cannot effect any other values in intraprocedural flow of information, so in the second transfer function all variables except r are associated with $\top$. After the meet, these variables will be mapped to the same values that they were mapped to in S_c before the method call.

3.5 Meet-Over-All-Paths Solution

A path in an ICFG is a sequence of edges such that each edge starts at the node in which the previous edge ended. Not all paths represent possible paths of execution. For example, if a method `m` is called by both method `a` and method `b`, there are edges connecting the entry and exit nodes of `m` to both the call site in `a` and the call site in `b`. There exists a path from the call site in method `a` to the entry node of method `m`, through the method, and from the exit node of `m` to the call site in method `b`, but this is not a possible path of execution. At run time method `m` must return to the call site in the calling method `a`. Only paths that return to the appropriate call sites are *valid* paths.

The transfer function f_p for a path p is the composition of the functions for the nodes in the path. Let $P = \{p_1, \ldots, p_n\}$ be the set of valid ICFG paths from the node *start* representing the entry node of the start method to node n not including n itself. Recall that S_n is the map associated with node n that provides information about the values of variables immediately preceding the node n. Thus, the map S_{start} is the information about the values of variables immediately before the invocation of the start method. For any reference-typed formal v of the start method with corresponding lattice element l_v, $S_{start}(v) = l_v$. For all other $v \in V$, $S_{start}(v) = \top$. The *meet-over-all-valid-paths solution* S_n^* for a CFG node n is defined by the following equation:

$$S_n^* = f_{p_1}(S_{start}) \wedge \ldots \wedge f_{p_n}(S_{start}), \text{ where } p_1, \ldots, p_n \in P.$$

3.6 Analysis Algorithm

The data-flow problem defined earlier belongs to the category of *interprocedural distributive environment* (IDE) problems. In an IDE problem the information at a program point is represented by a map from some finite set of symbols to some set of values. Also, in an IDE problem, the transfer functions must distribute over the meet operation. Our data-flow problem is an example of an IDE problem, where the set V of variables in the problem is the set of symbols and the lattice elements L are the values to which the symbols are mapped. Proof that the data-flow problem is distributive can be found in the appendix of the thesis. A general approach for solving IDE problems precisely is defined by Sagiv et al. [17]. We adapted their approach to obtain a flow- and context-sensitive algorithm. The algorithm is provably precise, meaning it computes the meet-over-all-valid-paths solution for each node. Detailed information about the algorithm can be found in [11].

CHAPTER 4

ANALYSIS ENHANCEMENTS

Several refinements can be made to the data-flow problem and analysis described in the previous chapter. These refinements, described in this chapter, result in the identification of more singleton call sites. In Chapter 5, we provide experimental evaluation of the analysis with and without these enhancements. The evaluations were performed on a set of Java components. In this chapter, accompanying the discussion of the refinements are segments of code from these components showing situations where the enhancements are used to obtain better results.

4.1 Static Fields

The analysis takes a conservative approach in the handling of *static fields*. Frequently though, the values of static fields do not change. This is the case if a static fields is not modified by any method reachable from the start method. Such read-only static fields are easy to identify and they provide an additional source for receiver objects. For each read-only static field sf we add a corresponding lattice element l_{sf}. The transfer function corresponding to an assignment from a read-only static fields is

- for $v = X.sf$: $f_n(S_n) = S_n[v \mapsto l_{sf}]$.

```
public class DecimalFormat extends NumberFormat {
   private static Hashtable cachedLocaleData = new Hashtable(3);

1    public DecimalFormat() {
2       Locale def = Locale.getDefault();
3       String pattern = (String) cachedLocaleData.get(def);
4       if (pattern == null) { /*cache miss */
5          //Get the pattern for the default locale.
6          ...
7          cachedLocaleData.put(def, pattern);
8       }
9       ...
10   }
  }
```

Figure 4.1: Handling of static fields

Figure 4.1 shows an example from the `DecimalFormat` class in one of the components; this class is part of the standard library package `java.text`. It provides functionality for formatting decimal numbers and contains a static field `cachedLocaleData` which is read-only. Let the `DecimalFormat()` constructor be the start method. In this example, note that without this refinement in the handling of static fields, the call sites at lines 3 and 7 would not be identified as singleton. Clearly though, once `cachedLocaleData` is identified to be read-only, these sites are singleton.

4.2 Instance Fields

Like with static fields, the analysis takes a conservative approach in the handling of *instance fields*. Recall that when a statement such as $v_1 = v_2.fld$ is encountered, the assumption is made that *any* object reference could be assigned to v_1, so $\perp$ is propagated as the lattice element associated with v_1. This conservative approach

in dealing with instance fields results in singleton call sites not being identified as singleton. In the running example found in Figure 2.1, consider the calls to `p5` and `p6`. These singleton call sites are not identified as singleton using the conservative approach. However, there is only one possible receiver object at these sites: the object to which `this.fld` refers at the invocation of the start method `m`.

The analysis can be modified in its handling of instance fields to identify more singleton call sites. The first step is to determine with regard to a program statement $v_1 = v_2.fld$ if there is only one object to which v_2 can refer. One run through the conservative analysis associates lattice elements with all program variables at every program point, so this step is done by running the analysis once and then examining statements of the form $v_1 = v_2.fld$. If the lattice element associated with v_2 at this point is something other than $\perp$, then there is exactly one object to which v_2 can refer: the object o_i corresponding to that lattice element. The next step is to determine if the field fld of o_i is not modified by any method reachable from the starting method. If this is the case, then the value of fld in o_i does not change and the expression $v2.fld$ is guaranteed to refer to a unique object. For each such object identified, a new corresponding element $l_{o_i.fld}$ is added to the lattice. The analysis is then executed again with the new lattice and with appropriately modified transfer functions for statements of the form $v_1 = v_2.fld$.

Let us consider again the running example. In method `m4` lattice element l_{this} is associated with `this`. Since `this.fld` is not modified by any method reachable from the start method `m`, an element $l_{this.fld}$ is added to the lattice. At the call site `this.fld.p5()` element $l_{this.fld}$ is associated with `this.fld`, so the site is identified as singleton. Likewise, at the statement `return this.fld`, element $l_{this.fld}$ is associated

```
  public class BigDecimal extends Number{
     private BigInteger intVal;

1    public BigDecimal negate() {
2       return new BigDecimal((this.intVal).negate(), scale);
3    }
  }
```

Figure 4.2: Handling of instance fields

with `this.fld` and $l_{this.fld}$ is propagated to variable `d` along the interprocedural edge from the exit node to return node. As a result the call site `d.p6()` is identified as singleton.

Figure 4.2 shows a straightforward example from the `BigDecimal` class of one of the subject components, based on standard library package `java.math`. In this example `negate` is the start method. In line 2 of the code a call to the `negate` method of the `BigInteger` class is made on the `intVal` instance field of `this`. Clearly, the only object `this.intVal` can refer to is the object it refers to at the invocation of the start method. The steps described earlier identify this call site as singleton.

It is possible for the objects to which fields refer to also have fields. The process of identifying the objects to which these fields refer and introducing the corresponding new lattice elements continues until the solution stabilizes. This approach may introduce lattice elements that represent *chains of field accesses*: e.g., elements of the form $l_{x.fld1.fld2.fld3}$. As shown by the experimental evaluation of the analysis in Chapter 5, the analysis typically stabilizes with a chain length between 3 and 5.

```
class X {...}
class G {
   public void m(int b) {
      X y = n();
      if (b>0)
         y = n();
      y.k2();
   }
   public X n() {
      X x = new X();
      x.k1();
      return x;
   }
}
```

Figure 4.3: Example of a non-singleton allocation site

4.3 Non-Singleton Allocation Sites

The third enhancement considers non-singleton allocation sites. Recall that a non-singleton allocation site is a site that can be executed more than once. If the source of a receiver object is a singleton allocation site then there is only one possible receiver object, but if the source is a non-singleton allocation site, then in general the receiver object could be, for example, the object created at the first execution of the site or the object created at the fifth execution. Recall also that non-singleton allocation sites occur when some segment of code containing the allocation statement is executed more than once in some sequence of events from the start method. Consider the situation in Figure 4.3 where method `m` is the start method. The non-singleton allocation site `x = new X()` is the source of the receiver object for the call site `x.k1()` following the allocation site in the same segment of code. The receiver object `x` refers to at this call site is the object created by the expression `x = new X()` in the same

execution of the segment. The call site `x.k1()` is not singleton, but for any execution of the call it is possible to identify the receiver object source as a *particular execution of the allocation site* so we will call such sites "resolvable".

We would like the sequence diagrams to show that the sources for resolvable call sites such as `x.k1()` in Figure 4.3 are the corresponding allocation sites in the same execution of the method. To achieve this, we introduce special lattice elements l_{alloc_j} corresponding to each non-singleton allocation site s_j.

Consider the result of lattice elements corresponding to non-singleton allocation sites being propagated in the same manner as all other lattice elements. Return to the example in Figure 4.3 and let l_{alloc_n} be the lattice element associated with the allocation site in method `n`. The element would be propagated back and associated with `y` at both returns from method `n`. Due to the branching caused by the `if` statement immediately preceding the call to `k2`, a meet of the lattice elements associated with variables would be performed. The element l_{alloc_n} would be associated with `y` along both branches, so at the call to `k2` the element l_{alloc_n} would be associated with `y`. While it is true that the call to `k2` can only be made on an object created at the allocation site in method `n`, there is no way of determining during which execution of method `n` the receiver object was created. As a result, we would like $\perp$ to be associated with variable `y` at this call site and the lattice elements corresponding to non-singleton allocation sites must be handled slightly differently from other lattice elements. If a lattice element l_{alloc_j} corresponding to a non-singleton allocation site is associated with a variable at a return statement, $\perp$ is propagated back to the calling method instead of l_{alloc_j}. This guarantees that the value is confined within the method in which the allocation site is present, and the methods that it transitively calls.

Figure 4.3 shows a straightforward example where refining the handling of non-singleton allocation sites results in a sequence diagram that better reflects the behavior of the code. Due to the handling of memory allocation and constructor calls by our analysis, there is another type of situation that also benefits from this refinement. Recall from Section 3.2 that a statement such as `x = new X()` is divided into two nodes in the CFG: the first node corresponds to the allocation of memory and the second corresponds to the constructor call. As a result of this division, without the refinement, for each non-singleton allocation site, its artificially separated constructor call is determined to be a non-singleton allocation site. For example, in Figure 4.3 the statement `x = new X()` is divided by the analysis into statements `x = new X;` and `x.X()`. Without the refinement, the constructor call `x.X()` is not resolved.

4.4 Limited Propagation of Lattice Elements

The sequence diagram generated by RED can be limited by the user in two ways. First, the user can choose that calls made by some methods are not displayed in the diagram. For example, if calls made by certain library methods are not of interest to the user, the user can choose to have the calls by these methods omitted from display.

Secondly, when using RED, the user can restrict the depth of calling relationships to make the diagram easier to understand. Relationships of the form "`m` calls `m2` which calls `m3` which in turn calls `m4`" are call chains. The *depth* of a call chain is the number of methods in the chain. For example, a call chain depth of two indicates that the user desires only to have the start method and the calls this method makes displayed in the diagram. The default depth used in RED is 5.

The analysis determines from these user defined constraints the set of call graph edges that will be represented in the diagram. The values of actual parameters are then only propagated along the call-entry edges that are in this set. The transfer function $f_{c,entry}(S_c) = \top_{map}$, rather than $f_{c,entry}(S_c) = \top_{map}[w_o \mapsto S_c(v_0)] \ldots [w_n \mapsto S_c(v_n)]$, is used for edges that are not in the set of call graph edges to be represented in the diagram.

Without this enhancement, if a method `m` in the CUA transitively calls itself by calling a method outside of the CUA, values from the non-CUA method are propagated to `m`. Since these values are not necessarily the same values that were previously propagated to `m`, this can cause some call sites that would otherwise be resolved to become associated with $\perp$. The goal of RED is to represent the behavior of methods in the CUA. Understanding the behavior of a method `m` in the CUA when called by some method not the CUA is gained by making `m` the start method of a diagram, so it is not relevant to propagate values along edges from non-CUA methods, or other call graph edges that are not intended to be represented in the diagram.

CHAPTER 5

EMPIRICAL STUDY

This chapter presents the experimental results evaluating the object naming analysis. The experiments were performed on a set of 21 subject components that come from various domains and typically are parts of reusable libraries. The analysis was implemented using the Soot framework [21] and were run on a 900 MHz Sun Fire 280-R machine.

5.1 Empirical Study

Table 5.1 shows the experimental results from running the analysis with all of the enhancements described in Chapter 4 and limiting the call chain depth to the default of 5. The column labeled "Number of sequence diagrams" shows the number of component methods that contain at least one interesting call site. An interesting call site is a site that (1) belongs to call chains that remain within the CUA, and (2) is not an invocation of methods in classes such as `java.lang.String`, `java.lang.Integer`, and other numeric types that are essentially primitive types. Each method with an interesting call site was considered as a start method of a sequence diagram. The analysis was executed on each of these start methods.

Component name	Number of sequence diagrams	Time (in seconds)	Number of call sites	Percent of sites resolved
`bigdecimal`	30	1.1 (0.09)	322	55.59%
`boundaries`	39	0.8 (0.17)	367	83.38%
`bytecode`	450	32.1 (0.13)	11012	78.00%
`calendar`	101	12.0 (0.13)	918	87.47%
`checked`	10	0.2 (0.48)	10	100%
`collator`	62	2.2 (0.12)	1154	79.29%
`date`	56	9.3 (0.08)	2173	90.47%
`decimal`	48	3.6 (0.09)	1361	96.33%
`gzip`	32	1.1 (0.11)	255	93.33%
`html`	214	25.7 (0.11)	4910	84.05%
`io`	46	8.7 (0.14)	300	85.00%
`jess`	457	1695.3 (0.89)	100974	71.13%
`jflex`	237	13.2 (0.11)	4657	87.55%
`math`	166	26.2 (0.13)	5831	60.62%
`message`	84	23.7 (0.10)	4003	77.47%
`mindbright`	328	44.3 (0.11)	10618	82.18%
`pdf`	146	67.7 (2.32)	2261	79.21%
`pushback`	13	0.2 (0.30)	13	100%
`sql`	109	6.3 (0.08)	411	99.03%
`vector`	30	0.4 (0.21)	66	96.97%
`zip`	77	2.6 (0.12)	1075	88.74%

Table 5.1: Experimental results

The column labeled "Time" shows the time (in seconds) to run the analysis for all sequence diagrams in the component. We also normalized this running time by the number of analyzed CFG nodes; the number in parenthesis shows the time to analyze one thousand CFG nodes. The running times indicate that the cost of the analysis is practical and that running time will not deter the use of the analysis in real-world software tools.

Sequence diagrams generated by RED represent the behavior of a start method and its transitive callees within the component up to a user defined call chain depth. Call sites in these component methods are represented in reverse-engineered diagrams as messages. The column labeled "Number of call sites" shows the total number of call sites in all diagrams. The analysis solution can be used to determine which diagram object should be the receiver of each message. The last column in the table shows the percentage of call sites for which the analysis is able to determine a unique diagram object that should be the receiver. These are the call sites that are singleton or resolvable.

The results show that the analysis can successfully resolve the majority of call sites. Note that for 18 of the components, more than 75% of the call sites are associated with lattice elements other than $\perp$. These lattice elements correspond to receiver object sources such formal parameters, allocation sites, and fields, and can be represented in reverse-engineered diagrams by appropriate precise object names. For two of the components all of the call sites were resolved. For an additional six of the components more than 90% of the call sites were resolved precisely. This result is important, because it indicates that it is possible to have a naming scheme that accurately represents the objects involved in a message exchange. Such a scheme is valuable for

reverse-engineering tools such as RED, because the greater the diagram precision, the easier it is for the user to accurately understand the behavior of the code. Accurate understanding of source code is useful in system development, software maintenance, and the writing of test cases.

5.1.1 Examination of `bigdecimal` Component

Component `bigdecimal` has the lowest percentage of resolved call sites. To gain an understanding of why only 55.59% of the call sites in the component were resolved precisely, we examined each unresolved call site by hand. This examination showed that of the 143 unresolved call sites, 125 are non-singleton call sites, so no analysis can determine a single run-time receiver object for these calls. The analysis legitimately reported $\perp$ at all these call sites.

This section presents situations summarizing the unresolved call sites, both singleton and non-singleton, encountered when analyzing the `bigdecimal` component. The component contains the `BigDecimal` class, which is part of the standard library package `java.math`. Some of the following situations are accompanied by examples from this class.

We will start by looking at situations where the unresolved call sites are non-singleton. A typical situation occurs when a method `m` has branch statements and a variable `x` in the method is assigned different objects along different execution paths. For all subsequent calls through `x` there are multiple possible run-time receiver objects.

Also, in situations where `x` is the return value of `m`, or it is otherwise possible for `m` to return multiple possible objects, all subsequent calls on whatever is returned by calling `m` are non-singleton. Figure 5.1 shows an example of this situation. The

```
public class BigDecimal extends Number {
   public BigDecimal divide(BigDecimal val, int scale, int roundingMode)
         throws ArithmeticException, IllegalArgumentException {
      ...
      BigInteger r = i[1];
      int cmpFracHalf = r.abs().multiply(BigInteger.valueOf(2)).
                              compareTo(divisor.intVal.abs());
      ...
   }
}

public class BigInteger extends Number{
   public BigInteger abs(){
      return (signum >= 0 ?  this :  this.negate());
   }
   public BigInteger negate(){
      return new BigInteger(this.magnitude, -this.signum);
   }
}
```

Figure 5.1: Multiple return values

call to `multiply` made in `divide` is non-singleton because the call to `abs` of the `BigInteger` class can return `this` or an object newly created by the call to `negate`. In the component, there are 58 call sites described by these branching situations for which the analysis legitimately reported $\perp$.

Another frequently occurring situation in the component has to do with fields. When a variable `x` can refer to multiple possible objects at run time, all subsequent call sites where a call is made on fields of the object to which `x` refers are also non-singleton sites. All calls made on fields of such an object are non-singleton because it is not possible to determine the object referred to by a field of an indeterminable object. The analysis enhancement for instance fields requires that a new lattice element representing a field be added only when it is a field of an object that is not

$\perp$. In the `bigdecimal` component, there are 29 call sites whose receiver objects are legitimately associated with $\perp$ due to this type of situation.

The remaining six unresolved non-singleton call sites can be described by the following situation. A call site `x.n()` in a method `m` can be a non-singleton site when there are multiple possible call chains from the start method to method `m`. For example, if the start method has calls `a.m()` and `b.m()` where `a` and `b` do not refer to the same object and `x` is assigned `this` before the call site in `m`, then the call site is not a singleton site because there are multiple runtime receiver objects: the object referred to by `a` or the object referred to by `b`.

Of the unresolved sites in the `bigdecimal` component, 87.4% are non-singleton sites. For the small percentage of remaining sites, our analysis did not determine there was only one run-time receiver object. For most of these singleton sites, the analysis was unable to determine the run-time receiver object due to the treatment of arrays. As is standard for most program analysis, our analysis treats assignments involving arrays conservatively. The index of an array element is typically not taken into consideration, so when an object reference is assigned to some location in the array, the assumption is made that the reference could be stored at any array element. Likewise, when the reference stored at some location in an array is assigned to some variable, the assumption is made that any reference stored in the array could be read. Of course, it is sometimes possible to gather additional information about the objects stored in arrays. Figure 5.2 shows an example of this case. In this example, the start method is the `add` method of class `BigDecimal`. Note that in method `add`, the formal `this` is stored at index 0 of the array that is passed to method `matchScale`, and that no other object reference could possibly be stored at index 0 of the array

```
public class BigDecimal extends Number {
   public BigDecimal add(BigDecimal val){
      BigDecimal arg[] = new BigDecimal[2];
      arg[0] = this;
      arg[1] = val;
      matchScale(arg);
      return ...
   }
   public static void matchScale(BigDecimal[] val){
      if(val[0].scale < val[1].scale)
         val[0] = val[0].setScale(val[1].scale);
      ...
   }
}
```

Figure 5.2: References stores in arrays

before `setScale` is invoked. The call to `setScale` is singleton, but the analysis does not identify it as such due to the complexities involved in dealing with arrays. In the `bigdecimal` component, there are 12 unresolved singleton call sites where the receiver is an object stored in an array.

In `bigdecimal`, the remaining six unresolved singleton call sites were not identified as singleton because the analysis does not analyze branching conditions. Consider in Figure 5.3 the `setScale` method as the start method. (The code of the `BigDecimal` class has been modified slightly in this figure to ease understanding.) The call site we are discussing is at line 8 and is not identified by the analysis as a singleton call site. This site is in the `divide` method which is called by `setScale`. The only possible source for the receiver object at this site is the allocation site in method `valueOf`, so if this allocation site is singleton then the call site is singleton. The start method `setScale` can call `divide`, which in turn can call `setScale`. Since

```
  public class BigDecimal extends Number {
     public BigDecimal setScale(int scale, int roundingMode) {
        ...
1       if(scale == this.scale){...}
2       else if(scale > this.scale) {...}
3       else { /* scale < this.scale */
4          return divide(valueOf(1, 0), scale, roundingMode);
        }
     }
     public BigDecimal divide(BigDecimal val, int scale, int roundingMode)
           throws ArithmeticException, IllegalArgumentException {
        ...
5       BigDecimal divisor;
6       if(scale + val.scale >= this.scale) {...}
7       else {
8          divisor = val.setScale(this.scale - scale, ROUND_UNNECESSARY);
        }
        ...
     }
     public static BigDecimal valueOf(long val, int scale){
9       return new BigDecimal(BigInteger.valueOf(val), scale);
     }
     public BigDecimal(BigInteger.val, int scale) {
        ...
10      this.intVal = val;
11      this.scale = scale;
     }
  }
```

Figure 5.3: Missed singleton allocation site

`valueOf` is called in `setScale` (which can be called multiple times), without analyzing branching conditions, it is easy to assume that the allocation site can be executed multiple times. However, it is not possible for the allocation site located in line 9 to be executed more than once. In order for it to be executed at all, `scale` must be less than `this.scale`. The newly allocated object's `scale` is 0, so when `divide` is invoked `scale < this.scale` and `val.scale` is 0. This means that the else portion of the `divide` method will be executed. In this portion of code, when `setScale` is called, it is invoked on `val`, so inside `setScale`, for this second invocation, `this.scale` is 0 and `scale` is a positive value. This means the `else if` in `setScale` at line 2 will be `true` and the allocation site at line 9 is only executed once.

The results from the closer examination of the `bigdecimal` component are important because they show that the low percentage of resolved call sites for this component is due in large part to the its unusually high number of non-singleton call sites. This examination is also valuable for the insights it provides about the unresolved call sites. For example, in order to identify all singleton call sites in the component, we observed that a more precise treatment of arrays and a detailed analysis of branching conditions would be necessary.

5.2 Static and Instance Field Enhancements

The data-flow problem discussed in Chapter 3 takes a conservative approach with regard to fields. Sections 4.1 and 4.2 describe enhancements for the analysis to more precisely analyze the behavior of static and instance fields. The effects of these enhancements are shown in Table 5.2.

Component name	Num. of call sites	Percent of sites resolved without fld enhancements	Percent of sites resolved with all enhancements	Additional call sites resolved due to fld enhancements	Num. of iterations
bigdecimal	322	47.83%	55.59%	7.76%	2
boundaries	367	52.04%	83.38%	31.34%	2
bytecode	11012	71.20%	78.00%	6.80%	4
calendar	918	72.77%	87.47%	14.71%	3
checked	10	0%	100%	100%	2
collator	1154	62.65%	79.29%	16.64%	3
date	2173	67.14%	90.47%	23.33%	3
decimal	1361	75.02%	96.33%	21.31%	2
gzip	255	42.35%	93.33%	50.98%	3
html	4910	48.25%	84.05%	35.80%	4
io	300	74.67%	85.00%	10.33%	3
jess	100974	67.71%	71.13%	3.40%	5
jflex	4657	72.26%	87.55%	15.29%	5
math	5831	58.50%	60.62%	2.13%	3
message	4003	63.05%	77.47%	14.41%	3
mindbright	10618	65.17%	82.18%	17.01%	5
pdf	2261	75.85%	79.21%	3.36%	2
pushback	13	100%	100%	0%	1
sql	411	99.03%	99.03%	0%	1
vector	66	96.97%	96.97%	0%	1
zip	1075	60.09%	88.74%	28.65%	3

Table 5.2: Experimental results with field enhancements

The first two columns in this table are the same as the similarly labeled columns in the Table 5.1. They show the component names and numbers of call sites in all diagrams. The fourth column, labeled "Percent of sites resolved with all enhancements," is also the same as the last column in Table 5.1. It shows the percentage of call sites for which the analysis is able to determine which diagram object should be the receiver. To obtain these results and all other results in this table, the analysis was run with the refinements relating to non-singleton allocation sites (Section 4.3), limited propagation along call graph edges (Section 4.4), and the default call chain depth of 5. The column labeled "Percent of sites resolved without fld enhancements" shows the percentage of call sites that were resolved when the analysis is run with the conservative treatment of static and instance fields. The column labeled "Additional call sites resolved due to fld enhancements" shows the difference of the percentages of the previous two columns. From this last column, note that 18 of the 21 components benefit from the field enhancements and that these enhancements are responsible for resolving more than 20 percent of the total call sites for 7 out of the 21 components.

Recall that instance field enhancement involves repeatedly executing the analysis and updating the lattice by adding lattice elements associated with instance fields. Each time the analysis is executed is a separate *iteration*; the last column of Table 5.2 shows the number of iterations for the analysis solution to stabilize.

5.3 Non-Singleton Allocation Sites Enhancement

The results from running the analysis with the non-singleton allocation sites enhancement are shown in Table 5.3. These results were obtained by running the analysis with the rest of the enhancements and with a default call chain depth of 5. The

Component name	Number of call sites	Percent of call sites resolved without non-singleton allocation site enhancement	Percent of call sites resolved with all enhancements	Additional call sites resolved due to non-singleton allocation site enhancement
bigdecimal	322	17.70%	55.59%	37.89%
boundaries	367	82.56%	83.38%	0.82%
bytecode	11012	58.43%	78.00%	19.57%
calendar	918	72.66%	87.47%	14.81%
checked	10	100%	100%	0%
collator	1154	58.15%	79.29%	21.14%
date	2173	66.27%	90.47%	24.21%
decimal	1361	90.52%	96.33%	5.80%
gzip	255	73.73%	93.33%	19.61%
html	4910	49.31%	84.05%	34.75%
io	300	26.67%	85.00%	58.33%
jess	100974	6.60%	71.13%	64.54%
jflex	4657	66.29%	87.55%	21.26%
math	5831	20.10%	60.62%	40.52%
message	4003	39.35%	77.47%	38.12%
mindbright	10618	48.95%	82.18%	33.23%
pdf	2261	17.34%	79.21%	61.88%
pushback	13	76.92%	100%	23.08%
sql	411	69.34%	99.03%	29.68%
vector	66	75.76%	96.97%	21.21%
zip	1075	53.49%	88.74%	35.26%

Table 5.3: Experimental results with non-singleton allocation site enhancement

last column of the table shows that a substantial percentage of call sites are resolved due to this enhancement. Of the 21 components, the only component that does not benefit from this enhancement is a component for which all call sites were determined to be singleton without the enhancement. In 15 out of the 21 components, more than 20% of the total call sites are resolved due to this enhancement and in three of the components over half of the call sites are resolved due to this enhancement.

5.4 Limited Propagation Enhancement

The results from running the analysis with the limited propagation enhancement are shown in Table 5.4. These results were obtained by running the analysis with all other enhancements and with a default call chain depth of 5. The last column shows that while most components were not effected significantly by this enhancement, there were some components for which the enhancement had a substantial effect. Component `checked` showed the greatest improvement with a 100% improvement rate, but the results for three other components also improved by more than 10%.

5.5 Call Chain Depth

A user of RED can restrict the depth of calling relationships to be displayed in a sequence diagram. Tables 5.5 and 5.6 show the sensitivity of the analysis precision with different depths. The complexity of a sequence diagram increases along with the depth of calling relationships being shown. A call chain depth of much more than five is difficult to understand, so we are showing results of depths no greater than eight. The tables show (for call chain depths from two to eight) the percentage of call sites for which the analysis is able to determine which diagram object should be the

Component name	Number of call sites	Percent of call sites resolved without limited propagation enhancement	Percent of call sites resolved with all enhancements	Additional call sites resolved due to limited propagation enhancement
bigdecimal	322	54.35%	55.59%	1.24%
boundaries	367	83.38%	83.38%	0%
bytecode	11012	77.56%	78.00%	0.44%
calendar	918	87.15%	87.47%	0.33%
checked	10	0%	100%	100%
collator	1154	79.29%	79.29%	0%
date	2173	85.96%	90.47%	4.51%
decimal	1361	95.59%	96.33%	0.73%
gzip	255	56.86%	93.33%	36.47%
html	4910	77.68%	84.05%	6.37%
io	300	81.33%	85.00%	3.67%
jess	100974	68.32%	71.13%	2.81%
jflex	4657	87.09%	87.55%	0.45%
math	5831	51.47%	60.62%	9.16%
message	4003	67.35%	77.47%	10.12%
mindbright	10618	77.33%	82.18%	4.85%
pdf	2261	77.71%	79.21%	1.50%
pushback	13	100%	100%	0%
sql	411	99.03%	99.03%	0%
vector	66	96.97%	96.97%	0%
zip	1075	73.21%	88.74%	15.53%

Table 5.4: Experimental results with limited propagation enhancement

receiver. For each call chain depth, the table has a column showing the total number of call sites in all diagrams when limited to that depth, and a column showing the percentage of those sites that are resolved. Although the number of call sites grows, sometimes significantly, as the depth of the call chains increases, the percentage of resolved sites does not vary greatly. The last column in Table 5.6 shows the range of the percentage of call sites resolved. Of the 21 components, 14 vary by less than ten percent; this indicates that for call chain depths up to eight, the analysis precision is not very sensitive to the user-defined depth. Regardless of this depth, the analysis is able to resolve a substantial percentage of the call sites.

	Call Chain Depth					
	2		3		4	
Component name	Num. of call sites	Resolved	Num. of call sites	Resolved	Num. of call sites	Resolved
bigdecimal	176	63.64%	217	60.83%	253	60.08%
boundaries	265	87.92%	346	82.66%	355	82.82%
bytecode	3385	78.88%	6369	77.53%	8986	78.60%
calendar	451	94.24%	568	91.55%	787	87.17%
checked	10	100%	10	100%	10	100%
collator	572	88.64%	826	82.20%	1032	78.59%
date	898	94.32%	1480	93.38%	1829	91.96%
decimal	912	95.72%	1276	96.08%	1361	96.33%
gzip	118	100%	184	99.46%	229	96.94%
html	1120	95.27%	2185	92.31%	3618	85.82%
io	245	85.71%	300	85.00%	300	85.00%
jess	10514	66.11%	22556	72.90%	60916	72.60%
jflex	3481	93.74%	4173	90.73%	4552	88.38%
math	1256	74.68%	2849	68.37%	4603	64.31%
message	1195	85.27%	2030	83.10%	2763	78.14%
mindbright	4052	86.60%	6424	84.54%	8598	82.91%
pdf	2075	79.95%	2259	79.24%	2261	79.21%
pushback	12	100%	13	100%	13	100%
sql	358	98.88%	399	99.00%	411	99.03%
vector	59	96.61%	66	96.97%	66	96.97%
zip	507	98.82%	879	95.90%	1053	91.17%

Table 5.5: Experimental results with for call chain depths 2-4

Call Chain Depth								Range
5		6		7		8		2-8
Num. of call sites	Resolved	Num. of call sites	Resolved	Num. of call sites	Resolved	Num. of call sites	Resolved	
322	55.59%	368	53.53%	370	53.78%	370	53.78%	10.10%
367	83.38%	367	83.38%	367	83.38%	367	83.38%	5.20%
11012	78.00%	12115	77.64%	12440	77.00%	12545	76.68%	2.20%
918	87.47%	945	86.56%	946	86.58%	946	86.58%	7.67%
10	100%	10	100%	10	100%	10	100%	0%
1154	79.29%	1230	80.49%	1289	81.38%	1301	81.55%	10.05%
2173	90.47%	2467	91.00%	2765	88.50%	2765	88.50%	5.82%
1361	96.33%	1361	96.33%	1361	96.33%	1361	96.33%	0.60%
255	93.33%	279	90.32%	299	86.29%	303	84.49%	15.51%
4910	84.05%	6314	82.93%	6885	81.95%	7082	82.45%	13.32%
300	85.00%	300	85.00%	300	85.00%	300	85.00%	0.71%
100974	71.13%	122402	69.32%	138917	67.27%	155203	65.22%	7.68%
4657	87.55%	4699	87.17%	4708	87.04%	4711	86.99%	6.75%
5831	60.62%	6459	58.31%	6667	56.49%	6719	56.09%	18.59%
4003	77.47%	5008	78.25%	5722	76.97%	5906	73.55%	11.72%
10613	82.18%	11784	81.07%	12555	79.52%	12875	78.76%	7.84%
2261	79.21%	2261	79.21%	2261	79.21%	2261	79.21%	0.74%
13	100%	13	100%	13	100%	13	100%	0%
411	99.03%	411	99.03%	411	99.03%	411	99.03%	0.14%
66	96.97%	66	96.97%	66	96.97%	66	96.97%	0.36%
1075	88.74%	1077	88.77%	1077	88.77%	1077	88.77%	10.07%

Table 5.6: Experimental results with for call chain depths 5-8

CHAPTER 6

RELATED WORK

Reverse engineering of UML sequence diagrams can be done with a static analysis or a dynamic analysis. Dynamic analysis of object interactions is relatively easy. The basic approach is to trace information about sources and targets of method calls through executions. Some existing research has considered reverse engineering of sequence diagrams and other related types of diagrams through dynamic analysis [19, 10, 4, 8, 3]. A disadvantage of this approach is that the resulting diagrams are dependent upon a particular run-time execution. This means that if the run-time executions are not complete, possible object behaviors will not be modeled in the resulting diagrams. Representations created dynamically either use one diagram object per class and represent interactions with any object in a given class as occurring with the diagram object for that class, or they use a diagram object for each run-time object.

Relatively little research has been done on reverse engineering of UML sequence diagrams and other closely related types of UML diagrams through static analysis. As described earlier, information about the naming scheme used by Borland's Together ControlCenter tool has not been published, but the tool appears to use a naming scheme based on variable names. The disadvantages of this approach are discussed

in Section 1.1.1. Kollman and Gogolla [6] propose a static analysis for constructing collaboration diagrams (called communication diagrams in UML 2) which similarly to sequence diagrams are a UML representation of object interactions. Object naming issues are not discussed in this work.

Tonella and Potrich [20] propose static techniques for reverse engineering of both sequence diagrams and collaboration diagrams by using a points-to analysis similar to Andersen's analysis [1]. The naming scheme they define uses a separate diagram object for each allocation expression. We decided against using points-to analysis, because with this technique it is not possible to identify singleton call sites. Points-to analysis is used to determine relationships of the form "at program statement s_1 reference variable `v` may point to some object allocated by program statements s_2 or some object allocated by statement s_3." Even if `v` only refers to objects created at a single statement s_i, it is possible that the statement is a non-singleton allocation statement so there may be multiple objects to which `v` could potentially refer. Also, our goal was to create a sequence diagram modeling all possible executions of some given start method. With points-to analysis, if the start method is called from multiple places, singleton call sites (with respect to the start method) will frequently be identified as non-singleton. Consider the running example in Figure 2.1. If method `m` is called from two locations where the actual parameters passed to the method are different, then the points-to sets of reference variable `a` will contain two elements and call sites `a.p1()`, `f.p2()`, `g.p3()`, and `c.p4()` will not be recognized as singleton call sites.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

This thesis presents a partial naming scheme for the reverse engineering of UML sequence diagrams. Such reverse engineering is useful in iterative development, software maintenance, and program testing. The problem of creating sequence diagrams from existing code presents many challenging issues that are not adequately addressed by existing approaches.

The naming scheme presented in this work proposes a technique for precisely modeling the run-time behavior at singleton call sites. Our approach is based on classic data-flow techniques. Experimental evaluation of the analysis indicates that its cost is practical and it achieves high precision. The analysis successfully resolved the vast majority of call sites in our subject components by associating them with appropriate precise object names.

The analysis is a major step towards representing run-time receiver objects in reverse-engineered UML sequence diagrams, but clearly some open questions remain. To start with, additional refinements for resolving more singleton call sites should be investigated. Also, non-singleton call sites need to be handled by creating a naming scheme for representing such sites. Once a scheme has been created, a static analysis needs to be designed for obtaining the information required by the scheme.

APPENDIX: A

A data-flow problem is *distributive* if each of its transfer functions is distributive. A transfer function f for our problem is distributive if $f(x \wedge y) = f(x) \wedge f(y)$ for all maps x and y. Proof that all transfer functions defined in Chapter 3 distribute over the meet operation (defined in the same chapter) is as follows.

The simplest transfer function presented in Sections 3.3 and 3.5 is the identity function where $f(S_n) = S_n$. This transfer function is associated with program statements such as $v_1[i] = v_2$ and with interprocedural edges connecting call nodes to return nodes when the called method does not return an object. Since the transfer function is the identity and has no effect on the map, $\forall v \in V(f(S_a))(v) = S_a(v)$ and $(f(S_b))(v) = S_b(v)$ so $(f(S_a))(v) \wedge (f(S_b))(v)$ which is the same as $(f(S_a) \wedge f(S_b))(v)$ equals $S_a(v) \wedge S_b(v)$. Therefore, the map $f(S_a \wedge S_b)$ is the same as the map $S_a \wedge S_b$.

Another simple transfer function is $f(S_n) = \top$ which is associated with interprocedural edges connecting exit nodes to return nodes, when the called method does not return an object reference. Regardless of what value a variables is mapped to in S_n, everything is mapped to $\top$ in $f(S_n)$, so $\forall v \in V(f(S_a))(v) = \top$, $(f(S_b))(v) = \top$, and $(f(S_a \wedge S_b))(v) = \top$. The meet of $\top$ with $\top$ is $\top$, so $\forall v \in V(f(S_a))(v) \wedge (f(S_b))(v)$ also equals $\top$. Therefore, this transfer function also distributes over the meet operation.

Consider now the three transfer functions which only change the original map slightly by mapping a variable to a specific lattice element:

$$f(S_n) = S_n[v \mapsto \top], \qquad f(S_n) = S_n[v \mapsto l], \qquad f(S_n) = S_n[v \mapsto \bot]$$

transfer function	$f(S_a)$	$f(S_b)$	$f(S_a \wedge S_b)$	$f(S_a) \wedge f(S_b)$
$f(S_n) = S_n[v_c \mapsto \top]$	$v_c \mapsto \top$	$v_c \mapsto \top$	$v_c \mapsto \top$	$v_c \mapsto \top$
$f(S_n) = S_n[v_c \mapsto l]$	$v_c \mapsto l$	$v_c \mapsto l$	$v_c \mapsto l$	$v_c \mapsto l$
$f(S_n) = S_n[v_c \mapsto \bot]$	$v_c \mapsto \bot$	$v_c \mapsto \bot$	$v_c \mapsto \bot$	$v_c \mapsto \bot$

Table A.1: Assignment of constant value

As shown with the identity transfer function, for all the variables for which the transfer function has no effect on the original map, the following fact holds:

Identity fact: $(S_a \wedge S_b)(v) = (f(S_a \wedge S_b))(v) = (f(S_a) \wedge f(S_b))(v)$
$\forall v \in V$ such that $(f(S_n))(v) = S_n(v)$

As a result, in order to prove that these functions are distributive, we need only to show that for the variables that are assigned new values by the function, the function distributes over the meet operation. Let v_c be the variable that is mapped to a different lattice element as a result of the transfer function. Table A.1 shows that the function distributes over the meet operation for v_c. Therefore these transfer functions are distributive. Note that the value that v_c is originally mapped to in S_a and the value it is originally mapped to in S_b do not matter.

Let us now consider the transfer functions associated with basic assignment statements such as $w_i = v_i$. The function is $f(S_n) = S_n[w_i \mapsto S_n(v_i)]$. Once again, due to the identity fact, it is only necessary to examine the variable mapped to a new value by the transfer function. Let w_i be the variable that is assigned a new value, the value of v_i. Table A.2 shows that the function distributes over the meet operation for w_i. Note that element x and element y are distinct lattice elements and that the last row of the table applies when $x \neq \top$ and $y \neq \top$.

S_a	$f(S_a)$	S_b	$f(S_b)$	$S_a \wedge S_b$	$f(S_a \wedge S_b)$	$f(S_a) \wedge f(S_b)$
$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$v_i \mapsto \bot$	$w_i \mapsto \bot$ $v_i \mapsto \bot$	$v_i \mapsto \bot$	$w_i \mapsto \bot$ $v_i \mapsto \bot$	$w_i \mapsto \bot$ $v_i \mapsto \bot$
$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$v_i \mapsto \top$	$w_i \mapsto \top$ $v_i \mapsto \top$	$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$
$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$
$v_i \mapsto x$	$w_i \mapsto x$ $v_i \mapsto x$	$v_i \mapsto y$	$w_i \mapsto y$ $v_i \mapsto y$	$v_i \mapsto \bot$	$w_i \mapsto \bot$ $v_i \mapsto \bot$	$w_i \mapsto \bot$ $v_i \mapsto \bot$

Table A.2: Assignment of variable value

The only remaining transfer function is $f(S_n) = \top_{map}[w_1 \mapsto S_n(v_1)] \ldots [w_k \mapsto S_n(v_k)]$ which is associated with parameter passing. Like the transfer function that assigns every variable to $\top$, $\forall v \in V$ such that $(f(S_n))(v) = \top$, $(f(S_a \wedge S_b))(v) = (f(S_a) \wedge f(S_b))(v)$. As a result, once again, it is necessary only to examine the variables mapped to new values by the transfer function. This was done for the last function in Table A.2. The same table applies for this transfer function, showing that for all w_i such that $1 \leq i \leq k$ where w_i is assigned the value to which v_i is mapped, $(f(S_a \wedge S_b))(v) = (f(S_a) \wedge f(S_b)(v)$. Therefore this last transfer function is also distributive proving that all of the transfer functions in Chapter 3 are distributive over the meet operation.

BIBLIOGRAPHY

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994.

[2] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 1999.

[3] L. Briand, Y. Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. Technical Report SCE-03-03, Carleton University, 2003. Also available in *Working Conference on Reverse Engineering, 2003.*

[4] W. DePauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualising the execution of Java programs. In *Software Visualization*, LNCS 2269, pages 151–162, 2002.

[5] M. Fowler. *UML Distilled.* Addison-Wesley, 2nd edition, 2000.

[6] R. Kollman and M. Gogolla. Capturing dynamic program behavior with UML collaboration diagrams. In *European Conference on Software Maintenance and Reengineering*, pages 58–67, 2001.

[7] C. Larman. *Applying UML and Patterns.* Prentice Hall, 2002.

[8] R. Oechsle and T. Schmitt. JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In *Software Visualization*, LNCS 2269, pages 176–190, 2002.

[9] OMG. *UML 2.0 Infrastructure Specification.* Object Management Group, `www.omg.org`, Sept. 2003.

[10] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *International Conference on Software Maintenance*, pages 34–43, 2002.

[11] A. Rountev and B. H. Connell. Object naming analysis for reverse-engineered sequence diagrams. In *International Conference on Software Engineering*, May 2005. To appear.

[12] A. Rountev, S. Kagan, and M. Gibas. Static and dynamic analysis of call chains in Java. In *International Symposium on Software Testing and Analysis*, pages 1–11, July 2004.

[13] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java based on annotated constraints. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 43–55, Oct. 2001.

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

[15] A. Rountev, O. Volgin, and M. Reddoch. Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University, Mar. 2004.

[16] J. Rumbaugh, I. Jacobson, and G. Booch. *UML Reference Manual.* Addison-Wesley, 1999.

[17] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[18] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[19] T. Systä, K. Koskimies, and H. Muller. Shimba—an environment for reverse engineering Java software systems. *Software–Practice and Experience*, 31(4):371–394, Apr. 2001.

[20] P. Tonella and A. Potrich. Reverse engineering of the interaction diagrams from C++ code. In *International Conference on Software Maintenance*, pages 159–168, 2003.

[21] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.