Object Naming Analysis for Reverse-Engineered Sequence Diagrams

Atanas Rountev
Department of Computer Science and Engineering
Ohio State University
rountev@cse.ohio-state.edu

Beth Harkness Connell
Department of Computer Science and Engineering
Ohio State University
connelbe@cse.ohio-state.edu

ABSTRACT

UML sequence diagrams are commonly used to represent object interactions in software systems. This work considers the problem of extracting UML sequence diagrams from existing code for the purposes of software understanding and testing. A static analysis for such reverse engineering needs to map the interacting objects from the code to sequence diagram objects. We propose an interprocedural dataflow analysis algorithm that determines precisely which objects are the receivers of certain messages, and assigns the appropriate diagram objects to represent them. Our experiments indicate that the majority of message receivers can be determined exactly, resulting in highly-precise object naming for reverse-engineered sequence diagrams.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms

Algorithms

Keywords

Static analysis, UML, reverse engineering

1. INTRODUCTION

Sequence diagrams play a central role in UML modeling of object interactions [16, 9]. Such diagrams show several objects and the messages that are exchanged among these objects. The diagrams may also contain additional information about the flow of control during the interaction, such as if-then conditions ("if `c` send message `m`") and iteration ("send message `m` multiple times").

ICSE'05, May 15–21, 2005, St. Louis, Missouri, USA.

1.1 Reverse-Engineered Sequence Diagrams

Reverse engineering of sequence diagram allows the automatic extraction of such diagrams from existing code. This is often necessary during *iterative development*. A typical scenario is to perform design recovery through reverse engineering of class diagrams and sequence diagrams in the beginning of the current iteration, based on the last iteration's code [7]. The resulting design documents serve as the starting point for subsequent design work. Additional reverse engineering is also usually necessary during an iteration.

Software maintenance activities can also benefit from reverse-engineered sequence diagrams. These diagrams are particularly well-suited for representing object interactions in object-oriented software. The growing body of such software, even for newer languages like Java, will create challenging maintenance problems for many years into the future. Automatic design recovery of object interactions for the purposes of software understanding and maintenance can be made possible by tools for reverse engineering of sequence diagrams.

Object interactions are an essential consideration for *testing of object-oriented software* [2]. Various testing approaches consider the interactions represented by sequence diagrams (or the similar collaboration diagrams) as part of their coverage requirements. These coverage goals can be defined with respect to different elements of statically-constructed sequence diagrams which are extracted from the code under test. Subsequent run-time analysis during test execution can be used to determine the coverage of these diagram elements and to highlight potential test weaknesses [12].

1.2 Object Naming in the Diagrams

The analysis described in this paper was implemented as part of the ongoing work on the RED tool for reverse engineering of sequence diagrams. The goal of this project is to provide high-quality tool support for reverse engineering of UML 2.0 sequence diagrams from Java code. The tool uses several static analyses, including call graph construction [13], call chain analysis [11], control flow analysis [15], and the object naming analysis described here. One of the central questions we needed to answer in this project was: *Given some call site `x.m()` in the code, how should the receiver object(s) at this site be represented in the diagram?* The answer to this question defines an object naming scheme which maps the potential run-time receiver objects at call sites to different objects represented in a sequence

diagram. Our attempts to resolve this issue revealed various challenging technical problems, and eventually led to the object naming analysis described in this paper.

1.2.1 Singleton Call Sites

We approached the problem in two stages. First, we considered the following question: given a call site `x.m()`, is it true that there is *only one* possible run-time receiver object at this site, regardless of how many times the site is executed? Such call sites will be referred to as *singleton sites*. We designed a static analysis algorithm for Java that identifies singleton call sites. Whenever RED represents such a call site in the reverse-engineered diagram, it is guaranteed that a single diagram object is sufficient to represent *precisely* the receiver object at the site. If a call site is not a singleton site, in general there is no guarantee that it is possible to have precise diagram representation of the run-time receiver objects for that site; later in the paper we present examples that illustrate this point.

In addition to identifying singleton sites, the algorithm determines an equivalence relation between such sites. Two sites are equivalent if their unique run-time receiver objects are guaranteed to be the same. This information is important in the construction of the diagrams, because if two run-time messages are sent to *the same run-time object*, the corresponding messages in the diagram should be sent to *the same diagram object*. For such call sites, the diagrams are guaranteed to represent correctly the semantics of the analyzed code.

To design the algorithm, we first defined an interprocedural dataflow problem which formalizes the intuitions outlined above. This problem was inspired by the constant propagation dataflow problem which is traditionally used in compiler optimizations. The underlying machinery used in our approach is similar to the techniques used to construct constant propagation analyses. We then defined a flow- and context-sensitive algorithm that solves the dataflow problem precisely—that is, it computes *exactly* the meet-over-all-valid-paths solution, which is the standard notion of precision in interprocedural dataflow analysis [18].

1.2.2 Generalization and Evaluation

The second stage of the work generalized the analysis of singleton call sites in several dimensions. First, a more precise treatment of object fields was introduced. Additionally, the set of call sites for which precise naming is possible was extended to include certain non-singleton sites. This allowed the analysis to become more powerful in the sense of providing precise object naming for a larger set of call sites from the code.

We implemented the generalized analysis and evaluated its performance on a set of 21 subject components. Our results indicate that the proposed approach has practical cost and achieves very high precision. For 18 of the 21 components, the analysis successfully determined precise object names for more than 75% of the considered call sites; for 7 components, this percentage was higher than 90%. Thus, in the majority of cases, the diagrams that are based on the analysis provide precise object naming which reflects correctly the meaning of the underlying code. Ultimately, this analysis brings us a step closer to providing RED users with sequence diagrams that are precise, concise, and easy to use in the context of software understanding and testing.

```
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();
}
```

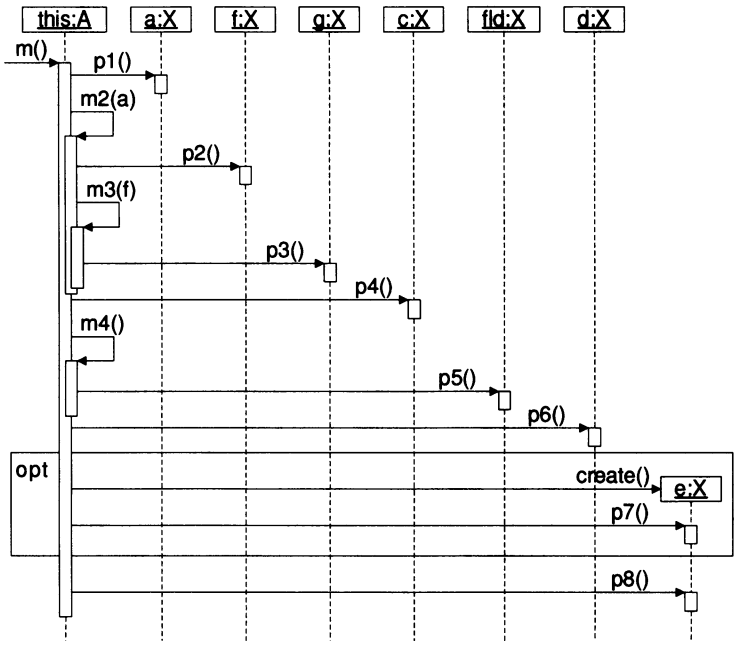
Figure 1: Running example.

2. PROBLEM DEFINITION

The input to RED contains a set of Java classes that form the *component under analysis*. The input also contains all other classes that are (transitively) referenced by component classes. The tool first builds a call graph for the component and all of its transitive callees. Our current implementation constructs the call graph using the points-to analysis from [13]; the technique from [14] is used to handle the case when the analyzed component is not a complete program.

A tool user chooses a method `m` from the analyzed component (we will refer to it as the *start method* for the diagram), and RED produces a sequence diagram that represents the potential sequences of run-time events that could be observed when `m` is invoked. For example, consider Figure 1, and assume that the analyzed component contains classes `A` and `X`. Furthermore, for the sake of brevity, assume that methods `p1` through `p8` in class `X` do not make any calls. If a tool user chooses start method `m`, Figure 2 shows two possible sequence diagrams for this method. The part of the diagram labeled *opt* represents optional behavior guarded by some condition, as defined in UML 2.0 [9].

As part of the diagram construction, RED needs to decide how to represent the run-time objects that are possible receivers at call sites. For example, for call site `c.p4()` from Figure 1, the first diagram in Figure 2 uses the diagram object labeled `c` to represent the run-time receiver object of message `p4`. The choice of this representation scheme has very significant impact on the quality of the produced diagrams. In the rest of the paper, we will refer to this scheme as the *object naming scheme* for the reverse-engineering analysis. Note that by "name" here we mean the actual object shown in the diagram (e.g., the object la-



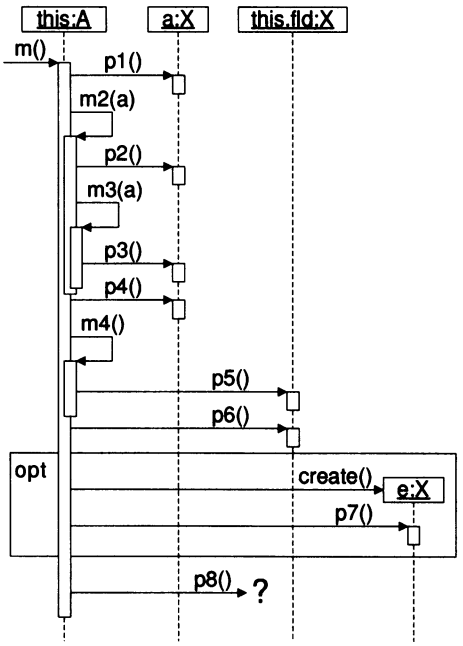


Figure 2: (a) Naming scheme in ControlCenter **(b) Naming scheme in our approach**

beled c), rather than simply the label (e.g., c) used inside that object.

2.1 Naming Based on Variable Names

The diagram in Figure 2(a) was constructed from the code in Figure 1 using the reverse-engineering functionality of the Borland Together ControlCenter modeling tool. (For ease of presentation, we slightly modified the visual representation of the diagram without altering its meaning). The object naming scheme used in this state-of-the-art commercial tool has not been published, but the diagram suggests that the scheme is based on the names of the variables that are used in invocation expressions. As a result, the same run-time object may be represented by several diagram objects. For example, the objects labeled `a`, `c`, `d`, `f`, and `g` actually correspond to the same run-time object—namely, the object that is referred to by the first formal parameter of `m`.

Such a naming scheme has serious shortcomings. First, it may introduce redundant objects in the diagram. Furthermore, it may incorrectly show that messages are sent to different objects when in reality they are sent to the same object. For example, messages `p1`, `p2`, `p3`, and `p4` have different receiver objects in Figure 2(a), even though at run time they are sent to the same object. It is also possible to have messages that appear to be sent to the same object, but in reality may be sent to different objects. For example, messages `p7` and `p8` are not *necessarily* sent to the same object at run time, but in the diagram the two messages have the same receiver object.

A programmer or tester that examines such a diagram may be easily confused, and may have to spend valuable time and effort investigating the code in order to recover the true nature of object interactions. The imprecision due to this naming scheme is likely to occur often in practice, because object references are routinely used as parameters and return values of method calls, and typically several variables in different methods refer to the same object. To address this problem, we propose the use of a naming scheme that is based on *interprocedural dataflow analysis* which tracks the flow of object references across method boundaries.

2.2 Singleton vs. Non-singleton Call Sites

Consider again the code in Figure 1. The calls to methods `p1`, `p2`, `p3`, and `p4` are guaranteed to have as a receiver a single run-time object, which is the object that formal `a` refers to when `m` is invoked. Similarly, the calls to `p5` and `p6` have as a receiver the object to which `this.fld` refers to before `m` is called. Finally, the receiver for the call to `p7` is the object created by the allocation expression inside `m`. Having this information, a reverse-engineering analysis can construct the diagram shown in Figure 2(b). Clearly, this diagram reflects the behavior of the code more precisely than the one in Figure 2(a), and therefore is preferable for the purposes of program understanding and testing. Our goal is to define a static analysis that makes possible the creation of this more precise diagram.

A call site c is a *singleton call site* if, for a given calling context of the start method, there is only one possible receiver object at c. (A formal definition of this notion is available in [4].) All call sites in Figure 1 except `e.p8()` are singleton sites with respect to start method `m`. We want to distinguish between singleton and non-singleton call sites because a reverse-engineered diagram can represent precisely the receiver objects for singleton sites, as illustrated by the diagram in Figure 2(b). For a non-singleton site, the representation is not as straightforward. For example, consider `e.p8()` which has two possible receivers. There are several possible representations of this call. First, we could show a message to only one of the two possible receivers, as done in Figure 2(a). Second, we could introduce an auxiliary "helper" object that represents either of the two possible receivers. Third, the diagram could show two messages, one sent to the object labeled `this.fld` and the other sent to the object labeled `e`. Fourth, a reverse-engineering tool could

create two separate diagrams, one for each of the two alternatives. Each of these possible treatments of non-singleton sites has potential advantages and disadvantages. For example, introducing helper objects makes the diagrams larger and somewhat imprecise, since multiple diagram objects now represent the same run-time object. Showing multiple receiver objects for the same call site may significantly clutter the diagram, especially if there are many non-singleton call sites.

The complexity introduced by non-singleton sites motivated us to separate our work into two stages. First, we defined techniques for identifying singleton call sites and for creating the appropriate diagram objects for them. The results from this work are described in this paper. As the experiments from Section 6 show, our approach successfully creates precise diagram objects for the majority of call sites. Second, we considered the different possible ways of handling non-singleton call sites, and the inherent tradeoffs of these techniques. This effort is currently under way, and it is not discussed further in this paper.

2.3 Naming Based on Equivalence Classes

Intuitively, two singleton call sites c_1 and c_2 are *equivalent* if they always have the same run-time receiver object. (A formal definition of equivalence is available in [4].) The equivalence relation can be used to define equivalence classes of singleton call sites. For start method `m` in Figure 1, there are four equivalence classes:

{`this.m2(a)`, `this.m3(f)`, `this.m4()`}
{`a.p1()`, `f.p2()`, `g.p3()`, `c.p4()`}
{`this.fld.p5()`, `d.p6()`}
{`X()`, `e.p7()`}

Each equivalence class corresponds to a diagram object in Figure 2(b). More generally, consider any static analysis that identifies a set of call sites as singleton sites and produces an equivalence partitioning of this set. Such an analysis defines an object naming scheme in which each equivalence class is mapped to a different object in the reverse-engineered diagram, and the messages at singleton sites are represented accordingly. Of course, this naming scheme is partial because it does not apply to non-singleton sites.

The goal of this work is to identify a large number of singleton call sites and to determine their equivalence. Our initial attempt to solve this problem considered *points-to analysis*. Points-to analysis is a popular form of interprocedural analysis that computes relationships of the form "at program statement s_1 reference variable v may point to some object allocated by program statement s_2". Unfortunately, such relationships cannot be used to identify singleton call sites. Even if we know that at `v.m()` variable `v` points *only* to objects allocated by s_2, it is of course possible for s_2 to create multiple objects (e.g., under different calling contexts). Furthermore, `v.m()` could be a singleton site even if `v` may point to objects created by several program statements. For example, in Figure 1, method `m` could by invoked by other parts of the system using as actual parameters different instances of `X` allocated by many distinct object allocation sites. Points-to analysis will report that formal `a` inside `m` may point to all of these objects. However, *during a particular invocation of* `m`, formal `a` points to only one object, and therefore `a.p1()` is a singleton call site.

The approach we have taken uses different techniques, based on ideas from traditional dataflow analysis. The next section defines a dataflow analysis problem that is at the core of this approach.

3. DATAFLOW PROBLEM

Our analysis identifies singleton call sites and finds the *sources* of the receiver objects at these sites. This section describes two such sources; later in the paper we provide a more general discussion of other sources. One category of sources are the formal parameters of the start method. For example, for `m` in Figure 1, there are two such sources: `a` and the implicit formal `this`. We would like to associate certain call sites with these sources, in the following sense: if a site c_i is associated with a formal f_j of the start method, then c_i is a singleton site at which the receiver object is guaranteed to be the same as the object that f_j refers to at the time when the start method is invoked.

Another category of sources are object allocation sites s_j that represent the creation of *exactly one* object during the execution of the start method and its transitive callees. We will refer to such allocation sites as *singleton allocation sites*. Statement `e = new X()` in Figure 1 illustrates this property. We want to associate a call site c_i with a singleton allocation site s_j whenever it is true that the receiver at c_i is definitely the unique run-time object created by s_j; of course, this implies that c_j is a singleton call site. Not every object allocation site creates a single run-time object. In particular, if an allocation site is inside a method that could be executed multiple times (i.e., a method that is reachable along multiple executions paths from the start method), it potentially creates multiple objects and cannot be considered as the source of receiver objects for singleton call sites. If a method is reachable from the start method only along a single execution path, an allocation site in this method is a singleton site only if it is not located inside a loop. It it straightforward to analyze the call graph and the control-flow graphs of all reachable methods in order to identify allocation sites s_j that are definitely executed only once.

3.1 Lattice

Based on these observations, we define a lattice of values that is used to define the dataflow problem. Each reference-typed formal of the start method corresponds to a distinct lattice element. Similarly, each singleton allocation site is represented by a separate lattice element. In addition, the lattice contains a top element $\top$ and a bottom element $\bot$. The lattice for our running example is $\{\top, \bot, l_{this}, l_a, l_{alloc1}\}$, where l_{alloc1} corresponds to `e = new X()`.

The goal of the analysis is to associate lattice elements with different program variables. For example, in the final solution computed for our running example, l_a will be associated with variables `a`, `c`, `f`, and `g` at the calls through these variables. This means that there is only one possible receiver at all such calls: the object that `a` points to when `m` is invoked. This information defines an equivalence class for these calls.

If $\bot$ is associated with some variable `v`, this means that `v` could refer to more than one object, and therefore calls through `v` are not singleton calls. Variable `e` in the running example has this property: at call site `e.p8()` the analysis solution associates $\bot$ with `e`, which shows that this call site is not a singleton site.

The partial order in the lattice is $\bot \leq l_i \leq \top$, and the

meet operation $\wedge$ is defined as follows:

$$x \wedge \bot = \bot, \quad x \wedge \top = x, \quad x \wedge x = x, \quad x \wedge y = \bot$$

The meet operation is used by the analysis to merge information about values that are propagated along different execution paths. In particular, consider the last rule (applicable when $x \neq y$, $x \neq \top$, and $y \neq \top$). If a variable may refer to one object along one execution path, and to another object along a different execution path, the variable is associated with $\bot$ and calls through it are not singleton calls. This lattice resembles the lattice for the constant propagation problem, which is a classic dataflow problem. Constant propagation determines expressions that definitely have the same value along all execution paths; similarly, our analysis determines variables that are guaranteed to refer to the same unique object along all execution paths.

3.2 Control-Flow Graphs

Consider the control-flow graphs (CFGs) for all methods that are reachable in the call graph from the start method. We will use V to denote the set of all reference-typed formal parameters and local variables in these methods. CFG nodes represent program statements, and CFG edges represent possible flow of control between these statements. For brevity, we discuss only 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 a field
- $v_1.fld = v_2$, where $v_1, v_2 \in V$ and *fld* is a field
- $v = new\ X$, 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.
- irrelevant node: e.g., `i = 5`

Call expressions have two possible forms. An instance call expression is $v_0.m(v_1, \ldots, v_n)$, where $v_i \in V$ and m is an instance method. A static call expression is $m(v_1, \ldots, v_n)$, where $v_i \in V$ and m is a static method.

Assumptions similar to the ones from above are commonly used to simplify the definition of dataflow problems. The assumptions can typically be ensured by introducing (implicit or explicit) temporary variables: e.g., statement `this.fld.p4()` can be broken down to `temp=this.fld` and `temp.p4()`. It is important to note that our implementation of the analysis (used for the experiments from Section 6) takes into account the full generality of possible Java statements, with the following exceptions. First, the analysis is designed for single-threaded code, and therefore we do not model multiple threads and their interactions. Second, since RED currently does not represent exceptional behavior in the sequence diagrams, all code related to `throw` and `catch` statements in Java is ignored. Finally, indirect accesses through reflection are not taken into account by the analysis.

3.3 Transfer Functions

We associate a map $S_n : V \to L$ with each CFG node n; here L is the lattice described earlier. If $S_n(v)$ is some lattice element other than $\top$ and $\bot$, the value of v immediately before the execution of n is guaranteed to be the unique object corresponding to that element. A value $S_n(v) = \bot$ shows that the analysis could not 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.

The effects of program statements on the solution can be represented by *dataflow transfer functions*. For each CFG node n, the analysis defines a function $f_n : (V \to L) \to (V \to L)$. If S_n provides information about the values of variables immediately before n, 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 that is the same as S except for the value associated with $v \in V$, which is changed to $l \in L$. The transfer functions are as follows:

- for $v_1 = v_2$: $f_n(S) = S[v_1 \mapsto S(v_2)]$
- for $v_1 = v_2.fld$: $f_n(S) = S[v_1 \mapsto \bot]$
- for $v_1.fld = v_2$: $f_n(S) = S$
- for $v = new\ X$, if this is a singleton allocation site with lattice element l_{alloc}: $f_n(S) = S[v \mapsto l_{alloc}]$
- for a non-singleton allocation site $v = new\ X$: $f_n(S) = S[v \mapsto \bot]$
- for calls and returns: discussed below
- for a branch node or an irrelevant node: $f_n(S) = S$

For an assignment $v_1 = v_2$, the analysis propagates the current value of v_2 to v_1. When the value is obtained through an object field in $v_1 = v_2.fld$, a conservative assumption is made that *any* object reference could be assigned to v_1, and therefore $\bot$ is propagated. Later in the paper we discuss an approach for more precise treatment of fields.

The handling of calls requires the introduction of an interprocedural CFG (ICFG) [18], in which the method CFGs are linked through interprocedural edges. Each method-level CFG is assumed to have an artificial *start node* and an artificial *exit node*. Each CFG node that represents a call is broken down into two nodes: a *call-site* node and a *return-site* node. There are interprocedural edges from a call-site note to the start nodes of all methods that could be invoked by the call; there are also corresponding edges from the exit nodes of these methods to the return-site node. Transfer functions are associated with these (call-site,start) and (exit,return-site) edges to represent the effects of parameter passing and return values. These effects are similar to assignments: for example, for a formal v_1 with a corresponding actual v_2, the effects are analogous to an assignment $v_1 = v_2$. For brevity, we omit the technical details of this aspect of the problem.

The transfer function f_p for a path p in the ICFG is the composition of the functions for the nodes and the interprocedural edges on the path. Not all ICFG paths represent possible executions. A *valid* path has interprocedural edges that are properly matched: each (exit,return-site) edge is matched correctly with the last unmatched (call-site,start) edge on the path, in the sense that both edges correspond to the same call site. Intuitively, on a valid path, a method always returns to the appropriate call site. The precise solution of the dataflow problem is defined with respect to the set of all valid ICFG paths.

The *meet-over-all-valid-paths solution* MVP_n for a CFG node n describes the variable values immediately before the execution of n. This solution is defined as

$$MVP_n = \bigwedge_{p \in VP(n)} f_p(MVP_{start})$$

where

- *start* is the start CFG node for the start method of the sequence diagram
- $VP(n)$ is the set of all valid paths p leading from *start* to n (p does not include n itself)
- MVP_{start} is the solution immediately before *start*. This solution takes into account the lattice elements that correspond to the formals of the start method. For any reference-typed formal v of the start method with a corresponding lattice element l_v, $MVP_{start}(v) = l_v$. For all other $v \in V$, $MVP_{start}(v) = \top$ indicating that currently there is no information about the value of v.

4. ANALYSIS ALGORITHM

The dataflow problem from the previous section is an example of an interprocedural distributive environment (IDE) problem. In IDE problems, the information at a program point is represented by a map from symbols to values (in our case a map $V \to L$). Sagiv et al. [17] define a general approach for solving such problems precisely. We have instantiated and adapted their approach to apply to the problem under consideration. The resulting flow- and context-sensitive algorithm, described in this section, is *provably precise* in the sense of computing the meet-over-all-valid-paths solution for each node.

4.1 Phase I: Flow of Values from Formals

The first phase of the analysis computes information that relates the values of local variables inside a method to the values of the formal parameters of this method. This information is encoded in a set F of triples (n, v_1, v_2), where n is an CFG node in some method m, v_1 is a reference formal of m, and v_2 is a local variable or a formal parameter of m. A triple $(n, v_1, v_2) \in F$ shows that the value of v_2 immediately before n could be the same as the value of v_1 at the entry of the method. Essentially, this set encodes the flow of values from formals to locals/formals within the same method.

In method `m` from Figure 1, the value of `c` after the call to `m2` is the same as the value of actual `a` at the entry of the method, and therefore $(n, \mathtt{a}, \mathtt{c}) \in F$ for all subsequent CFG nodes n. Since `a` is not assigned, we also have $(n, \mathtt{a}, \mathtt{a}) \in F$ for all nodes n in the method. In `m2`, the value of `q` is the same as the value of formal `f`, and $(n, \mathtt{f}, \mathtt{q}) \in F$ for all appropriates nodes n.

The computation of F requires information about the effects of method calls. This information is encoded by a set SF which summarizes the flow of values through certain calls. A pair $(n, v) \in SF$ corresponds to a call-site node n at which v is a reference actual used in the call. If the pair is in SF, this means that the return value of some method invoked by n may be the value that v had immediately before the call. In other words, the value of v may flow back to the call site as a return value. For example, in Figure 1, for call site `c = this.m2(a)` we have $(n, \mathtt{a}) \in SF$. Similarly, for `q = this.m3(f)`, a pair $(n, \mathtt{f})$ is used to show that the called method returns the value of `f`.

The computation of F and SF is based on a worklist algorithm. Set F is initialized with triples (n, f_i, f_i) where f_i is a formal and n is the start node of the corresponding method. Initially SF is empty. Whenever a new triple is added to F, it is also put on the worklist and is eventually processed. The processing of a triple (n, v_1, v_2) depends on the type of CFG node n. If the node does not assign a value to v_2, new triples (n', v_1, v_2) are created for all CFG successors n' of n. If the node is an assignment $v_3 = v_2$, new triples (n', v_1, v_3) are created and propagated to the successor nodes n'.

Whenever the current triple (n, v_1, v_2) corresponds to a call site n of the form $v_3 = c$, where c is a call expression which uses v_2 as an actual, set SF is consulted to decide whether (n', v_1, v_3) should be created. Pairs are added to SF when a triple (n, v_1, v_2) corresponds to *return* v_2: all callers of the surrounding method are examined and SF is updated with pairs (n, v_4), where v_4 is the actual at call site n which corresponds to formal v_1. At this point of time, the current solutions for all such v_4 are propagated to the corresponding left-hand-side variables at the call sites.

4.2 Phase II: Internal/Backward Propagation

The second and third phase of the algorithm compute the actual lattice values. The second phase propagates such values within a method and from a method back to its callers, while the third phase propagates information from callers to callees. Both phases use an array $S(n, v)$ to store the lattice element associated with $v \in V$ on top of CFG node n.

Whenever some lattice element l is propagated to $S(n, v)$, the value is updated as follows: $S(n, v) := S(n, v) \wedge l$. If $S(n, v)$ changes as a result, the pair (n, v) is put on a worklist and eventually processed. The processing depends on the type of statement that n represents. For example, if the value of v is not changed by n, the current lattice element in $S(n, v)$ is simply propagated to $S(n', v)$ for all CFG successors n'. For an assignment $w = v$, $S(n, v)$ is propagated to $S(n', w)$.

If a statement assigns a specific lattice element to v, this element is propagated to $S(n', v)$ in the beginning of the second phase. For our running example, due to the singleton allocation site `e = new X()`, the solution for `e` is set to l_{alloc1} at `e.p7()`. For formal parameters f_i of the start method of the sequence diagram, $S(n, f_i)$ is initialized to l_{f_i} at the start node n of that method.

If the current pair (n, v) represents a statement *return* v, the value of $S(n, v)$ is propagated back to all left-hand-side variables at the corresponding invoking call sites. For the running example, `this.fld` is returned by method `m4`, and value $\bot$ is propagated back to `d` at the call site (recall that the transfer functions treat the values of object fields conservatively). Thus, $\bot$ is propagated to `e` after the assignment

`e = d`. Since both $\perp$ and l_{alloc1} are propagated to `e` at node `e.p8()`, the meet of these two lattice elements is taken, and $\perp$ is associated with `e` at this node.

If n represents a call site at which v is used as an actual, set SF from phase I is examined to determine whether the value of v should be propagated to the left-hand-side variable at the call. For `c = this.m2(a)`, since $(n, \texttt{a}) \in SF$, the value l_a associated with `a` before the call is also associated with `c` after the call. The partial solution after phase II is the following:

- $S(n, \texttt{this}) = l_{this}$ and $S(n, \texttt{a}) = l_a$ for all CFG nodes n in method `m`
- $S(n, \texttt{c}) = l_a$ for all nodes n after the call to `m2`
- $S(n, \texttt{d}) = \perp$ for all nodes n after the call to `m4`
- $S(n, \texttt{e}) = \perp$ before the `if` statement
- $S(n, \texttt{e}) = l_{alloc1}$ before `e.p7()`
- $S(n, \texttt{e}) = \perp$ before `e.p8()`
- $S(n, v) = \top$ for all other pairs (n, v)

4.3 Phase III: Internal/Forward Propagation

The last phase of the algorithm propagates information from callers to callees. If n is a call node at which v is used as an actual, the value of $S(n, v)$ is propagated to the corresponding formal(s). For example, l_a is propagated to `f` due to the call to `m2` in Figure 1. New values at formals are propagated further into the bodies of the corresponding methods, using the information computed in Phase I. For method `m2` we have $(n, \texttt{f}, \texttt{q}) \in F$ for all n after the call to `m3`. This means that the value of `q` at these nodes could be the same as the value that `f` had upon entry of `m2`. Therefore, l_a is propagated to $S(n, \texttt{q})$. Furthermore, since `f` is used an actual in the call to `m3`, l_a is also propagated to `g`. The additional values computed during this phase are:

- $S(n, \texttt{this}) = l_{this}$ for all occurrences of `this` in methods `m2`, `m3`, and `m4`
- $S(n, \texttt{f}) = l_a$ for all CFG nodes n in `m2`
- $S(n, \texttt{q}) = l_a$ for all nodes n after the call to `m3`
- $S(n, \texttt{g}) = l_a$ for all nodes n in `m3`

4.4 Equivalence Classes

Consider all call sites through `this` in `m` and `m2`. Since after Phase III we have $S(n, \texttt{this}) = l_{this}$ at all such sites, the analysis reports the following equivalence class of singleton call sites:

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

Similarly, since `a`, `f`, `g`, and `c` are associated with l_a at the corresponding call sites, the analysis computes equivalence class

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

Finally, the call to `p7` and the call to the constructor of `X` are both associated with l_{alloc1}, resulting in equivalence class

{`X()`, `e.p7()`}

4.5 Correctness and Complexity

It can be proven that the algorithm is an instance of a more general algorithm for solving interprocedural distributive environment (IDE) problems [17]. A corollary of this result is that our algorithm terminates with a solution which is exactly the same as the meet-over-all-valid paths solution for the dataflow problem described in Section 3.

Assuming that the number of formals for each method is bounded by some small constant, the computation in phase I has time complexity in the order of $E_{call} + \sum_m E_m \times V_m$. Here E_{call} denotes the number of edges in the call graph, and m is a node in this call graph (that is, m is a method transitively reachable from the start method). The number of edges in the CFG of m is E_m, and the number of locals and formals in m is V_m. The term $E_m \times V_m$ corresponds to the cost of constructing set F for method m, and E_{call} corresponds to the interprocedural propagation through set SF. In phases II and III the value of $S(n, v)$ can change at most two times. Thus, the cost of intraprocedural and interprocedural propagation during these two phases is also in the order of $E_{call} + \sum_m E_m \times V_m$.

The algorithm can be optimized in a straightforward manner as follows. Consider a local/formal v in some method m, and suppose that there is at most one assignment $v = \ldots$ in m. In this case it is not necessary to maintain different values for v at all CFG nodes inside m. Thus, instead of computing a separate $S(n, v)$ for each CFG node n, the algorithm can compute a single lattice element $S(v)$ for the entire method. It is easy to prove that this optimization does not effect the correctness or precision of the analysis. If there is only a constant number of v for which this optimization *cannot* be applied, the complexity of the algorithm is in the order of $E_{call} + \sum_m V_m + \sum_m E_m$. Here $\sum_m V_m$ represents the cost of processing the optimized variables v, and $\sum_m E_m$ corresponds to the cost of handling the non-optimized variables. The implementation of the algorithm uses this optimization, since our experience indicates that variables are typically not assigned multiple times.

5. ANALYSIS ENHANCEMENTS

The analysis described in the previous section can be refined in several dimensions. Our implementation (used for the experiments described in Section 6) employs all of these refinements.

5.1 Static and Instance Fields

If a *static field* is not modified by any method reachable from the starting method of the sequence diagram, the field refers to the same object at all times and therefore can be used as another source of receiver objects for singleton call sites. For each read-only static field sf, we introduce a corresponding lattice element l_{sf} and modify the transfer functions accordingly. Since in Java code static fields are very often `final` (i.e., immutable), this enhancement identifies additional singleton call sites.

The algorithm from the previous section treats *object fields* conservatively, and propagates $\perp$ whenever an object field is read. As a result, in the running example, the calls to `p5` and `p6` are determined to be non-singleton calls. However, there is only one possible receiver object at these call sites: the object to which `this.fld` refers to in the beginning of start method `m`.

This problem can be solved by the following generalization. After running the analysis described earlier, we examine all statements of the form $v_1 = v_2.fld$ and determine whether the computed value of v_2 at this assignment is different from $\perp$. If this is true, then v_2 refers to some unique object o_i at this program point. Furthermore, suppose that *fld* is never assigned in the start method or any of its callees. In this case we know that the value of *fld* in object o_i does not change, and therefore expression $v_2.fld$ is guaranteed to refer to a unique object. A new lattice element can be introduced to represent this unique object, and this element can be propagated to v_1 at assignment $v_1 = v_2.fld$. This approach requires straightforward changes to the propagation algorithm described in the previous section.

In the running example, `this` in method `m4` is associated with l_{this} and therefore "`return this.fld`" propagates a lattice element $l_{this.fld}$ to `d`. As a result, the calls to `p5` and `p6` are associated with this new element; thus, both calls are singleton calls and they form an equivalence class. The process of introducing these 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 $l_{x.fld1.fld2.fld3}$. Our experiments indicate that the solution typically stabilizes with maximum chain length between 3 and 5.

5.2 Non-singleton Object Allocation Sites

The second enhancement considers non-singleton allocation sites. For example, consider a method m_i which contains a statement s_j of the form `x = new X()`, and suppose that s_j is not located inside a CFG loop. If m_i is called by several other methods, s_j is not a singleton allocation site according to the definition presented earlier. However, if for example we have statements `x.k(); y = x; y.n()` after s_j inside m_i, in RED we would like to show that the receiver object at these two calls is the same as the one created by s_j. To achieve this, we introduce a special lattice element l_{alloc_j} corresponding to s_j, and we propagate it inside method m_i during phase II. However, during this phase, the value is not propagated back to the callers: if the value reaches a return statement, instead of back-propagating l_{alloc_j} we back-propagate $\perp$. This guarantees that the value is confined within the allocating method m_i. If allocation site s_j is inside a loop, RED uses a single diagram object to represent the unbounded set of run-time objects allocated by the statement; thus, in this case we also introduce and propagate a special lattice element l_{alloc_j}. During phase III (which propagates values from callers to callees), the new lattice values are propagated to the transitive callees of m_i. For example, for the call `y.n()` from above, l_{alloc_j} is propagated to `this` in `n`. Thus, if `n` contains a call `this.p()`, the diagram will show that the receiver object of this call is the one created by s_j.

5.3 Limited Forward Propagation

A RED user can choose to limit the reverse-engineered diagram in two ways. First, calls made by "uninteresting" methods—for example, certain library methods—are omitted when displaying the diagram. (Of course, such methods are still analyzed by the static analysis). Second, the depth of calling relationships can be restricted to make the diagrams easier to comprehend. The depth of call chains (i.e., relationships "the start method calls `m1` which calls `m2` which calls `m3` etc.") in the visual representation can be limited by a user-defined parameter; the default depth limit is 5. Based on these constraints, the analysis can determine the set of call graph edges that will be represented visually as messages in the reverse-engineered diagram. In phase III, values are forward-propagated from callers to callees only along such call edges.

Comp	Start	Time [s]	Sites	Resolved
`checked`	10	0.2 (0.48)	10	100%
`pushback`	18	0.2 (0.30)	13	100%
`bigdecimal`	30	1.1 (0.09)	322	55.6%
`vector`	30	0.4 (0.21)	66	97.0%
`gzip`	32	1.1 (0.11)	255	93.3%
`boundaries`	39	0.8 (0.17)	367	83.4%
`io`	46	8.7 (0.14)	300	85.0%
`decimal`	48	3.6 (0.09)	1361	96.3%
`date`	56	9.3 (0.08)	2173	90.5%
`collator`	62	2.2 (0.12)	1154	79.3%
`zip`	77	2.6 (0.12)	1075	88.7%
`message`	84	23.7 (0.10)	4003	77.5%
`calendar`	101	12.0 (0.13)	918	87.5%
`sql`	109	6.3 (0.08)	411	99.0%
`pdf`	146	67.7 (2.32)	2261	79.2%
`math`	166	26.2 (0.13)	5831	60.6%
`html`	214	25.7 (0.11)	4910	84.1%
`jflex`	237	13.2 (0.11)	4657	87.6%
`mindbright`	328	44.3 (0.11)	10618	82.2%
`bytecode`	450	32.1 (0.13)	11012	78.0%
`jess`	457	1695.3 (0.89)	100974	71.1%

Table 1: Experimental results.

6. EMPIRICAL STUDY

We performed an experimental evaluation of the analysis on the 21 subject components listed in Table 1. The analysis implementation is based on the Soot framework [21] and was executed on a 900 MHz Sun Fire 280-R machine. The components are from various domains and typically are parts of reusable libraries. The column labeled "Start" shows the number of component methods that contain at least one interesting call.[1] Each of these methods was considered as a potential start method of a sequence diagram, and the analysis was executed on the method and all of its transitive callees, including non-component callees. Call graph edges were determined using the approach from [13, 14]. The input to the analysis is the program representation produced by Soot for all these methods, together with the corresponding method-level control-flow graphs. The analysis output are the solutions $S(n, v)$ inside all methods whose behavior would be need to be represented in a reverse-engineered diagram.

The total time (in seconds) to run the analysis for all start methods is shown in the third column of the table. For example, for `checked`, the table shows the time to run the analysis 10 times, once for each of the 10 potential start methods. The running time includes phases I, II, and III of the algorithm from Section 4, using the enhancements

[1]We did not consider as interesting the calls to methods from `java.lang.String` and from the standard numeric types (e.g., `java.lang.Integer`), since these are essentially primitive types.

described in Section 5. It is important to note that for each start method, there are usually dozens or hundreds of non-component library methods that are included in the set of all transitive callees. All these methods are processed fully by the analysis, and their processing time is included in the measurements in the third column of Table 1.

We also normalized the absolute running time by the number of analyzed CFG nodes; the number in parentheses in the third column shows the time to analyze one thousand CFG nodes. Typically, the running time was around a hundred to a few hundred milliseconds per thousand CFG nodes, with the exception of `pdf` and to a lesser degree `jess`. These results indicate that the analysis cost is practical, and therefore running time will not be an obstacle for the use of the algorithm in real-world tools.

Sequence diagrams in RED represent the effects of component methods up to a certain depth of call chains, as described in Section 5.3. A call site in such a method will be shown as a message in a reverse-engineered diagram, and the analysis solution can be used to determine which diagram object should be the receiver of that message. The next-to-last column contains the total number of such call sites for all diagrams.[2] The last column shows what percentage of these call sites were resolved by the analysis to a value other than $\perp$. The higher the percentage, the more call sites are guaranteed to have precise object naming in the diagrams.

The results show that the analysis can successfully resolve the majority of call sites. For 18 components, more than 75% of the call sites were associated with the appropriate precise object names. For 7 components, this percentage was higher than 90%. Therefore, for the majority of messages, the reverse-engineered diagrams are guaranteed to provide precise object naming which represents correctly the behavior of the analyzed code. This result is important because it indicates that it is possible to create precise diagrams in RED and in similar reverse-engineering tools. Users of such tools can benefit from this precision when performing software maintenance tasks and when writing tests for Java software. Of course, there preliminary results need to be confirmed with additional data points and eventually by real-world tool users.

We examined component `bigdecimal`, which has the lowest resolution percentage. This component contains class `java.math.BigDecimal`, which has several occurrences of the following situation: method `m` returns one of several possible objects, depending on a variety of conditions. Another method contains a call `y = x.m()`. For all subsequent calls through `y`, there are multiple possible run-time receiver objects that are generated from different sources (e.g., some objects come from static fields read by `m`, and others are newly created by `m`). Due to this pattern (and similar ones), the analysis legitimately reported $\perp$ for 125 out of the 143 call sites that could not be resolved. We are currently considering generalizations of the naming scheme that can handle this situation, as part of our ongoing work on naming techniques for non-singleton call sites.

7. RELATED WORK

Reverse engineering of sequence diagrams could be done through static or dynamic analysis. Some approaches analyze run-time program behavior and build sequence diagrams or similar representations [19, 10, 5, 8, 3]. As with many other program analysis problems, static and dynamic approaches have both advantages and disadvantages. A static approach produces a conservative model of run-time behavior, and therefore may report infeasible object interactions. Furthermore, the level of detail in the produced information may be too high, and some abstraction mechanisms may be necessary to make the reverse-engineered diagrams easier to comprehend. As described in Section 5.3, RED employs two such mechanisms. However, it is clear that a more powerful set of abstraction techniques will be needed to make the tool practical in real-world use.

A potential problem for dynamic analysis techniques is the dependence of the diagram on the particular run-time execution that was observed. In some cases input data for such execution may not be available, especially for incomplete systems (e.g., reusable modules) that cannot be executed in stand-alone manner. Furthermore, it is not possible to know how well the execution covers all possible aspects of the interaction. For example, it is not possible to have high confidence in the consistency between design and code, if this consistency is judged from sequence diagrams that were constructed from execution traces. Similarly, for reengineering tasks, the incomplete run-time information may mislead the programmer into performing incorrect code modifications. Another potential disadvantage is that sequence diagrams produced only with dynamic analysis cannot be used for evaluating the adequacy of testing.

In reverse engineering through static analysis, the problem of object naming has not been investigated sufficiently. As described earlier, the ControlCenter tool appears to use a naming scheme that is based on variable names. Figure 2 illustrates the disadvantages of such a scheme. Kollman and Gogolla [6] propose a static analysis for constructing collaboration diagrams (which are similar to sequence diagrams); they do not discuss issues of object naming. Tonella and Potrich [20] present reverse engineering techniques for sequence diagrams and collaboration diagrams. They use a points-to analysis (similar to Andersen's analysis [1]) to construct the call graph and to define an object naming scheme: there is a separate diagram object for each `new` expression. We decided against using points-to information as a naming scheme; the rationale for this decision was discussed in Section 2.3.

Naming based on points-to analysis is an example of an approach that uses *may-alias* information. For reverse-engineered sequence diagrams it is more appropriate to use *must-alias* information—that is, knowledge that expressions are aliased for all run-time executions, rather than for some run-time executions. Our analysis can be thought of as a limited form of must-alias analysis. While there has been some work on must-alias analyses (e.g., for C programs), we are not aware of any approaches that use techniques similar to constant propagation.

8. CONCLUSIONS AND FUTURE WORK

We propose a low-cost algorithm that generates precise diagram objects for the majority of call sites in our subject components. The algorithm is a major step towards defining a complete naming scheme for reverse-engineered sequence diagrams. We are currently investigating approaches for object naming at non-singleton call sites.

[2] Call sites `this.m(..)` are excluded from this count because for them a sequence diagram by default will show self-messages.

One potential problem for static reverse engineering of sequence diagrams is the complexity of the produced diagrams. In case the analyzed methods have complicated behavior (e.g., significant intraprocedural flow of control, behavioral variations based on calling context, etc.), the resulting diagram may be hard to comprehend. Clearly, this issue calls for extensive future investigations. It is likely that some abstraction mechanisms will be necessary in order to make complicated diagrams easier to understand and use. One possible approach is to generate multiple sequence diagrams for the same start method, where each diagram corresponds to some restricted subset of the possible behaviors.[3] With this approach, the object naming analysis for each diagram should consider only the behaviors represented by that diagram, which could increase the number of reported singleton call sites. Another possibility is to construct a detailed diagram and then get user feedback about uninteresting diagram elements in order to construct a simplified diagram. This means that the reverse-engineering tool should support an "exploration mode" in which the user interactively refines the constructed diagram. This approach may require modifying the naming analysis to take advantage of the user-defined constraints in order to improve the precision of the analysis solution. Finally, it may be desirable to move parts of the diagram into separate subdiagrams, which would produce a hierarchical structure that may be easier to understand, display, and navigate.

Acknowledgments. We would like to thank the ICSE reviewers for their valuable comments and suggestions.

9. REFERENCES

[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. In *Working Conference on Reverse Engineering*, pages 57–66, 2003.

[4] B. H. Connell. Object naming in reverse engineering of UML sequence diagrams. Master's thesis, Ohio State University, Nov. 2004.

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

[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 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, 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.

[12] A. Rountev, S. Kagan, and J. Sawin. Coverage criteria for testing of object interactions in sequence diagrams. In *Fundamental Approaches to Software Engineering*, 2005. To appear.

[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.

[3] This approach was suggested by one of the ICSE reviewers.