Composition Inference for UML Class Diagrams

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute

milanova@cs.rpi.edu

ABSTRACT

Knowing which associations are compositions is important in a tool for the reverse engineering of UML class diagrams. Firstly, recovery of composition relationships bridges the gap between design and code. Secondly, since composition relationships explicitly state a requirement that certain representations cannot be exposed, it is important to determine if this requirement is met by component code. Verifying that compositions are implemented properly may prevent serious program flaws due to representation exposure.

We propose an implementation-level composition model based on ownership and a novel approach for identifying compositions in Java software. Our approach is based on static ownership inference; it is parameterized by class analysis and is designed to work on incomplete programs. We present empirical results from two instantiation of our approach. For one of these instantiations, on average 40% of the examined fields account for relationships that are identified as compositions. In addition, we present a precision evaluation which shows that the analysis achieves almost perfect precision—that is, it almost never misses composition relationships. The results indicate that precise identification of interclass relationships can be done with a simple and inexpensive analysis, and thus can be easily incorporated in reverse engineering tools that support iterative model-driven development.

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

UML, points-to analysis, reverse engineering, ownership

1. INTRODUCTION

In modern software development design recovery through reverse engineering is performed often; in iterative development processes, it is recommended that reverse engineering is performed at the beginning of every iteration to recover the design from the previous iteration [22].

UML class diagrams describe the architecture of the program in terms of classes and interclass relationships; they are scalable, informative and widely-used design models. While the UML concepts of class and inheritance have corresponding first-class concepts in object-oriented programming languages, the UML concepts of *association, aggregation* and *composition* do not have corresponding language concepts. Thus, while the reverse engineering of classes and inheritance hierarchies is straightforward, the reverse engineering of associations presents various challenges.

UML associations model relatively permanent interclass relationships; conventionally, they are implemented using instance fields of reference type [22] (e.g., an association from class A to class B is implemented using a reference field of type B in class A). Thus, reverse engineering tools infer associations by examining instance fields of reference type; however, the inference is often non-trivial. One challenge is the recovery of one-to-many associations implemented using pseudo-generic containers (e.g., `Vector`). Another challenge is the recovery of compositions. Modern reverse engineering tools such as Rational Rose do not address these challenges and produce inconsistent class diagrams (see Guéhéneuc and Albin-Amiot [19] for detailed examples). Clearly, this leads to a gap between design class diagrams and reverse engineered class diagrams which hinders understanding, round-trip engineering and identification of design patterns.

Towards the goal of bridging this gap, this paper proposes a methodology for inference of binary associations for UML class diagrams. Our major focus is the inference of composition relationships, which we believe is challenging and inadequately addressed in previous work. While the UML concept of aggregation is "strictly meaningless" [13, Chapter 5] (i.e., it has no well-defined semantics to distinguishe it from association), the UML concept of composition has a well-defined semantics that emphasizes the notion of *ownership*: a "composition is a strong form of [whole-part] association with strong ownership of parts by the composite and coincident lifetime of parts with the composite. A part may belong to only one component at a time" [36, Chapter 14]. Therefore, a composition relationship at design level states the requirement for ownership and no *representation exposure* at implementation level (i.e., the owned component object cannot be exposed outside of its composite owner object); if composition is implemented correctly ownership must be preserved.

It is important to investigate techniques for recovery of composition relationships. Firstly, it helps bridge the gap between the design class diagram and the reverse engineered diagram. Secondly, since composition relationships explicitly state a requirement that certain representations cannot be exposed, it is important to determine if this requirement is met by component code. Verifying early on, through the reverse engineering of the class diagram, that compositions are implemented properly may prevent serious program flaws due to representation exposure such as the well-known `Signers` bug in Java 1.1.[1]

Therefore, the goals of this work are (i) to define an implementation-level ownership model that captures the notion of composition in design and (ii) to design an analysis algorithm that infers ownership and composition using this model. Our definition of implementation-level composition is based on the *owners-as-dominators* ownership model [9, 30]; in this model the owner object (the composite) should dominate an owned object (a component)—that is, all access paths to the owned object should pass through its owner. Intuitively, an owned object may be accessed by its owner as well as other objects within the boundary of the owner (e.g., an owned object stored in an instance field may be passed to an owned container). As pointed out by Clarke et al. [9, 30] and observed during our empirical study, the owners-as-dominators model captures well the notion of composition in modeling.

We propose a novel static analysis for ownership inference. If the ownership inference determines that all objects stored in a field are owned by their enclosing object, the analysis identifies a composition through that field. Our approach works on incomplete programs. This is an important feature because in the context of reverse engineering tools it is essential to be able to perform separate analysis of software components. For example, it is typical to have to analyze a component without having access to the clients of that component. Our ownership inference analysis is parameterized by *class analysis*, which determines the classes of the objects a reference variable or a reference object field may refer to. We use the class analysis solution to approximate the possible accesses between run-time objects. Our work defines a general framework for ownership and composition inference; it encompasses a wide range of analyses with different degrees of cost and precision.

We have implemented two instantiation of this framework and present empirical results on several components. For one of these instantiations, on average 40% of the examined fields account for relationships that are identified as compositions. Additionally, we present a precision evaluation which shows that the analysis achieves almost perfect precision—that is, it almost never misses composition relationships. The results indicate that precise identification of interclass relationships can be done with a simple and inexpensive analysis, and thus can be easily incorporated in reverse engineering tools that support iterative development.

[1]In Java 1.1 the security system function `Class.getSigners` returned a pointer to an internal array allowing clients to modify the array and compromising the security of the system.

This work has the following contributions:

- We propose an implementation-level ownership and composition model that captures well the notion of composition in modeling.
- We propose a general analysis framework for static inference of ownership and composition relationships in accordance with our model; the analysis framework is parameterized by class analysis and works on incomplete programs.
- We present an empirical study that evaluates two instantiations of our framework. The results indicate that precise identification of composition relationships can be achieved with relatively inexpensive analysis.

The paper is organized as follows. Section 2 defines the static analysis problem. Section 3 presents the generalized class analysis, and Section 4 outlines the technique that allows application of class analysis on incomplete programs. Sections 5 and 6 describe the ownership and composition inference analysis. Section 7 discusses analysis complexity. Section 8 presents the empirical results. Section 9 discusses related work and Section 10 concludes the paper.

2. PROBLEM STATEMENT

Reverse engineering tools typically infer associations by examining instance fields of reference type in the code. In our model, an association relationship from class A to class B through a field f is refined as composition if it can be proven that every instance of A owns the instances of B that its f field refers to. Thus, given a suitable definition of implementation-level ownership and composition, our goal is to design a static analysis that answers the question: given a set of Java classes (i.e, a component to be analyzed) for what instance fields we observe implementation-level composition throughout all possible executions of arbitrary client code built on top of these classes? The output is a set of fields for which the relationship is guaranteed to be a composition for arbitrary clients.

The input to the analysis contains a set *Cls* of interacting Java classes. We will use "classes" to denote both Java classes and interfaces as the difference is irrelevant for our purposes. A subset of *Cls* is designated as the set of *accessible classes*; these are classes that may be accessed by unknown client code from outside of *Cls*. Such client code can only access fields and methods from *Cls* that are declared in some accessible class; these accessible fields and methods are referred to as *boundary fields* and *boundary methods*.

Sections 2.1 and 2.2 describe the ownership model and the notion of implementation-level composition based on it. Section 2.3 discusses some constraints to the model that allow more precise detection of ownership and composition.

2.1 Ownership Model

The ownership model is based on the notion of *owners-as-dominators* [9, 8, 30]. It is essentially the model proposed by Potter et al. [30] with several modifications that allow more precise handling of popular object-oriented patterns such as iterators, composites and factories [14]. In this model, each execution is represented by an *object graph* which shows access relationships between run-time objects:

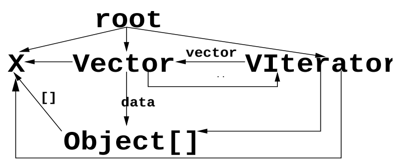
```
public class Vector {
 protected Object[] data;
 public Vector(int size) {
1 data = new Object[size]; }
 public void addElement(Object e,int at) {
2 data[at] = e; }
 public Object elementAt(int at) {
3 return data[at]; }
 public Enumeration elements() {
4 return new VIterator(this); }
}
final class VIterator implements Enumeration {
 Vector vector;
 int count;
 VIterator(Vector v) {
5  this.vector = v;
6  this.count = 0; }
 Object nextElement() {
7  Object[] data = vector.data;
8  int i = this.count;
9  this.count++;
10 return data[i]; }
}
main() {
11 Vector v = new Vector(100);
12 X x = new X();
13 v.addElement(x,0);
14 Enumeration e = v.elements();
15 x = (X) e.nextElement();
16 x.m();
}
```

Figure 1: Simplified vector and its iterator.

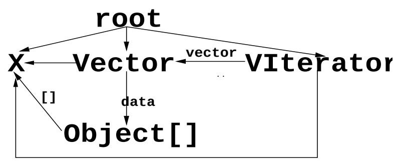
- Let f be a reference instance field in a run-time object o. There is an edge $o \xrightarrow{f} o'$ in the object graph if and only if field f in o refers to o' at some point of program execution.[2]
- There is an edge $o \xrightarrow{[]} o'$ if and only if some element of array o refers to o' at some point of program execution.
- There is an edge $o \rightarrow o'$ if and only if an instance method or constructor invoked on receiver o has local variable r that refers to o', or a static method called from an instance method or constructor invoked on o, has a local variable r that refers to o'. There is an edge of this kind only if there is no edge of the first kind from o to o'.

The start of program execution is expressed with a special node `root`; `root` represents `main` and objects referenced from static fields. For example, executing `main` in Figure 1 results in the object graph in Figure 2(a). Node `Vector` corresponds to the object created at the *new* site at line 11, node `Object[]` corresponds to the array created at the site at line 1, node `VIterator` corresponds to the iterator created at the site at line 4, and node `X` corresponds to the object created at the site at line 12.

[2]We require that all newly created objects appear in the object graph explicitly [9]. That is, at the point of creation a new object is stored in a new local variable; this does not change program semantics.



(a) Original Object Graph



(b) Relaxed Object Graph

Figure 2: Object graphs for Figure 1.

The owners-as-dominators model states that the owner of an object o is the immediate dominator of o in the object graph [30].[3] Thus, according to this model `Object[]` is not owned by its enclosing `Vector` object for this execution due to the access relationship (although only temporary) between `VIterator` and `Object[]`.

To make the model less restrictive, we introduce the *relaxed object graph* which omits edges due to certain temporary access relationships. We consider two kinds of temporary access relationships. The first kind arises when an object is created by one object and immediately passed to another object without being used; the relationship between the creating object and the new object is only temporary but if shown on the graph it is likely to restrict ownership. This notion captures the situations when an object is created and immediately returned (e.g., as in *return new VIterator*(*this*); in method `elements` in Figure 1) and when an object is created and immediately passed to another object (e.g., as in *new BufferedReader*(*new FileReader*(*fileName*))). This situation occurs in popular object-oriented design patterns such as factories, decorators and composites; in these cases the temporary relationship between the creating object and the newly created one is a matter of safety and flexibility of the implementation rather than an intention of the design. The second kind of temporary access relationships arises from field read statements $r = l.f$, where r is not assigned, passed as an implicit or explicit argument, or returned. This notion captures the situation that arises in iterators (consider statement $data = vector.data$ in `nextElement` in Figure 1)—iterator objects have temporary references to the representation of their collections, which allows efficient access of collection elements; however, the collection object is always in scope. Therefore, if all accesses of o' in the context of o are due to such temporary access relationships,

[3]Node m *dominates* node n if every path from the root of the graph that reaches node n has to pass through node m. The root dominates all nodes. Node m *immediately dominates* node n if m dominates n and there is no node p such that m dominates p and p dominates n.

edge $o \rightarrow o'$ is not shown in the relaxed object graph.

The relaxed object graph for the execution of `main` in Figure 1 is shown in Figure 2(b). Edge `Vector`→`VIterator` is omitted because it is due to a temporary access relationship of the first kind; edge `VIterator`→`Object[]` is omitted as well because it is due to a temporary access relationship of the second kind. The owner of o is the immediate dominator of o in the relaxed object graph. Thus, `root` owns `X`, `Vector` and `Viterator`, and `Vector` owns `Object[]`.

2.2 Implementation-level Composition

Let A be a class in *Cls*, and f be a field of type B declared in A where B is a reference type (class, interface or array type [15]). The ownership property holds for f if throughout all possible executions of arbitrary clients of *Cls*, every instance of A owns the instances of B that its f field refers to. Consider the case when f is a collection field—that is, all objects stored in the field are arrays or instances of one of the standard `java.util` collection classes (e.g., `java.util.Vector`). If every instance of A owns all corresponding instances stored in the collection, there is a *one-to-many composition* relationship between A and D, where D is the lowest common supertype of the instances stored in the collection; [4]; otherwise, there is a one-to-many regular association. For collection fields for which the ownership property holds, there is an attribute of the association {*owned collection*} that indicates that the collection is owned by its enclosing object. Consider the case when f is not a collection field. If the ownership property holds for f, the association between A and B is a *one-to-one composition*; otherwise it is a regular one-to-one association.

Example. Consider the package in Figure 3. This example is based on classes from the standard Java library package `java.util.zip`, with some modifications made to simplify the presentation and better illustrate the problem and our approach. *Cls* contains the classes from Figure 3 plus class `ZipEntry`. The accessible classes are `ZipInputStream`, `ZipOutputStream` and `ZipEntry` and the boundary methods are all public methods declared in those classes (i.e., the component can be accessed from client code through the public methods declared in these classes).

Clearly, the `CRC32` objects are always owned by their enclosing streams. Thus, there is a one-to-one composition relationship between class `ZipInputStream` and class `CRC32` through field `crc`. Similarly, there is a one-to-one composition relationship between `ZipOutputStream` and `CRC32` through field `crc`. There is a regular one-to-one association through field `entry` in `ZipInputStream`; it is easy to construct client code on top of these classes such that the `ZipEntry` instances created in `ZipInputStream` objects are leaked to client code from `getNextEntry`. Similarly, there is a regular one-to-one association through `entry` in `ZipOutputStream` because the `ZipEntry` objects are passed from client code to `putNextEntry`. The associations through fields `names` and `entries` are both one-to-many regular associations between `ZipOutputStream` and `ZipEntry`; both have attribute {*owned collection*}. The `ZipOutputStream` instance trivially owns the `Hashtable` instance. It owns

[4]Note that in Java, a unique non-trivial (i.e., non-`Object`) common supertype may not exist. The analysis used `Object` as the common supertype in this case.

```
package zip;
public class InflaterInputStream {
  protected Inflater inf;
  protected byte[] buf;
  public InflaterInputStream(Inflater inf,
   int size) {
    this.inf=inf;
    buf=new byte[size]; }
  public InflaterInputStream(Inflater inf) {
    this(inf, 512); }
  // methods read and fill contain instance calls on inf
}
public class ZipInputStream extends
 InflaterInputStream {
  private ZipEntry entry;
  private CRC32 crc=new CRC32();
  public ZipInputStream() {
    super(new Inflater(true), 512); }
  public ZipEntry getNextEntry() {
    crc.reset();
    inf.reset();
    if ((entry=readLOC())==null) return null;
    return entry; }
  private ZipEntry readLOC() {
    ZipEntry e=new ZipEntry();
    // code reads and writes fields of e
    return e; }
}
public class ZipOutputStream extends
 DeflaterOutputStream {
  private ZipEntry entry;
  private Vector entries=new Vector();
  private Hashtable names=new Hashtable();
  private CRC32 crc=new CRC32();
  public ZipOutputStream() {
    super(new Deflater(...)); }
  public void putNextEntry(ZipEntry e) {
    // code reads and writes fields of e
    if (names.put(e.name, e)!=null) { ...  }
    entries.addElement(e);
    entry=e; }
  public void closeEntry() {
    ZipEntry e=entry;
    // code reads and writes fields of e
    crc.reset();
    entry=null; }
  public void finish() {
    Enumeration enum=entries.elements();
    while (enum.hasMoreElements()) { ...  } }
}
```

Figure 3: Sample package zip.

the `Vector` instance as well, although the `Vector` instance is referred to in the context of its iterator (recall the example in Figure 1); however, the iterator is a local object owned by the enclosing `ZipOutputStream` object which ensures that the `Vector` instance is dominated by the enclosing `ZipOutputStream`.

2.3 Discussion

In order to allow more precise detection of implementation-level composition, we employ the following constraint, standard for other problem definitions that require analysis of incomplete programs [34, 32]. We only consider executions in which the invocation of a boundary method does not leave *Cls*—that is, all of its transitive callees are also in *Cls*. In particular, if we consider the possibility of unknown subclasses, all instance calls from *Cls* could potentially be "redirected" to unknown external code that may affect the composition inference. For example, a field may be identified as composition in the current set of classes but an unknown subclass may override some method and the overriding method may leak the field (e.g., by passing it to a static field).

Thus, *Cls* is augmented to include the classes that provide component functionality as well as all other classes transitively referenced. In the experiments presented in Section 8 we included all classes that were transitively referenced by *Cls*. This approach restricts analysis information to the currently "known world"—that is, the information may be invalidated in the future when new subclasses are added to *Cls*. Another approach is to change the analysis to make worst case assumptions for calls that may enter some unknown overriding methods. However, in this case, the analysis will be overly conservative and likely report fewer compositions. Thus, we believe that it is more useful to restrict the analysis to the known world; of course, the analysis user must be aware that the information is valid only for the given set of known classes.

3. GENERALIZED CLASS ANALYSIS

Class analysis determines the set of objects that a given reference variable or a reference object field may refer to. In this work, class analysis information is needed to construct a graph that approximates all possible object graphs that can happen when arbitrary client code is built on top of *Cls*; subsequently the approximation of the object graph is used to infer ownership and composition relationships. There is a large body of work on class analysis with different trade-offs between cost and precision [28, 29, 1, 3, 18, 35, 37, 40, 39, 24, 33, 17, 25, 26, 43, 23, 5, 44]. In this paper, we propose a general framework for class analysis that encompasses a large number of analyses with varying degrees of cost and precision. The framework can be instantiated to relatively inexpensive and imprecise analyzes such as Rapid Type Analysis (RTA) [3] as well as relatively expensive and precise analyses such as object-sensitive points-to analysis [25, 26]. The object graph construction uses the output of the class analysis. Therefore, the precision and cost of the object graph construction and subsequent ownership inference depend on the precision and cost of the underlying class analysis.

This section is organized as follows. Section 3.1 describes the major dimensions of precision in class analysis; these are modeled by the generalized class analysis algorithm. Section 3.2 describes the relevant notation, Section 3.3 defines the generalized algorithm and finally, Section 3.4 defines specific instantiations that span the cost/precision spectrum.

3.1 Dimensions of Precision in Class Analysis

The two major dimensions of precision in class analysis are *flow sensitivity* and *context sensitivity*. Flow-sensitive analyses take into account the flow of control between program points; they are more precise and more expensive than flow-insensitive ones. Context-sensitive analyses distinguish between different calling contexts of a method and are more precise and more expensive than context-insensitive ones.

Other dimensions of precision include *field sensitivity*, *directionality*, the *call graph construction scheme*, the *reference representation scheme* and the *object naming scheme*. Field-sensitive analyses are able to distinguish flow through different object fields while field-insensitive analyses merge flow through different fields of an object; thus field-sensitive analyses are more precise than field-insensitive ones. Directional analyses, also referred to as propagation-based analyses, process assignments in one direction, while bi-directional analyses, also referred to as unification-based analyses process assignments in both directions. For example, a directional analysis processes assignment $l = r$ by propagating the set of classes for r into l; in contrast, a bi-directional analysis propagates the set of classes for r into l and the set of classes for l into r. With respect to call graph construction, the analysis may construct a call graph on-the-fly while propagating classes, or pre-compute the call graph using an inexpensive technique such as Class Hierarchy Analysis (CHA) [11].[5] Typically, the analyses that construct the call graph on-the-fly are significantly more precise than the ones that pre-compute the call graph.

Yet another dimension of precision is the reference representation scheme. This dimension refers to the number of analysis variables used to represent reference variables; more precise analyses use more variables, while less precise ones use less variables and thus "merge" information for distinct variables. A typical reference representation scheme is to use one analysis variable for each reference variable in the program. Finally, the object naming scheme refers to the number of object names used to represent heap objects. More precise analyses use more object names, while less precise ones use less names and thus "merge" distinct run-time objects. One popular naming scheme is to represent each object by its class. Another popular naming scheme is to represent each object by its allocation site. Note that the reference representation and object naming schemes are related to context sensitivity and the degrees of context sensitivity—a context-sensitive analysis typically defines a representation for reference variables and objects; however, as demonstrated in this paper, one may vary the reference and object naming schemes within context-insensitive analysis as well.

In this paper we consider analyses that are *flow-insensitive*, *field-sensitive*, *directional* and construct the call graph *on-the-fly*. The analysis designer may vary the *context sensi-*

[5]Class Hierarchy Analysis (CHA) examines the declared type of the receiver variable and the class hierarchy and computes a set of possible run-time targets.

tivity scheme, the *reference representation* and the *object naming scheme* in order to achieve analysis of the desired precision and cost.

3.2 Notation

The generalized class analysis is defined in terms of the following sets. Set R is the set of locals, formals and static fields of reference type in the program. Set S is the set of all object allocation sites s_i and set I is the set of all call sites in the program. Set F contains all instance fields in program classes and set C contains all program classes. There are three sets that are used to express analysis representation. Set $\mathcal{C}$ represents the set of all method contexts and $\mathcal{C}_m \subseteq \mathcal{C}$ is the set of contexts for method m. When representing context-insensitive analysis, we use notation ϵ—that is $\mathcal{C}_m = \mathcal{C} = \{\epsilon\}$ for each method m in the program. Set V is the set of reference variable representatives and set H is the set of object names used by the analysis.

Analysis parameterization is expressed in terms of three functions. Function $\mathit{findNewContext} : I \times H \times \mathcal{C} \rightarrow \mathcal{C}$ defines the context sensitivity scheme. It is applied at each call site i and takes into account the call site identifier $i \in I$, the receiver object $h \in H$ and the context c of invocation of the enclosing method. It computes a new context of invocation for each target at i. This function is discussed additionally in Section 3.3.

Function $v(r, c) : R \times \mathcal{C} \rightarrow V$, defines the reference representation scheme; it takes a reference variable $r \in R$ and a context $c \in \mathcal{C}$ and outputs the representative of r in context c. The reference representation scheme may be context-insensitive or context-sensitive. For example, one context-insensitive scheme uses a single analysis variable v to represent all reference variables in the program—that is, we have $v(r, \epsilon) = v$ for all $r \in R$. The most common context-insensitive scheme uses an analysis variable for each reference variable in the program—that is, we have $v(r, \epsilon) = r$ for every r. A context-sensitive representation scheme typically uses an analysis variable for each reference variable and each context—that is, we have $v(r, c) = r^c$.

Function $h(s_i, c) : S \times \mathcal{C} \rightarrow H$ defines the object naming scheme. Analogously to the reference representation scheme, the object naming scheme may be context-insensitive or context-sensitive. For example, one frequently used context-insensitive object naming scheme uses the class of the object as its representative—that is, we have $h(s_i, \epsilon) = A$, where A is the class of the object allocated at site s_i. Another scheme uses the allocation site as its representative—that is, we have $h(s_i, \epsilon) = h_i$. Context-sensitive naming schemes are more precise and use multiple object names for each allocation site. For example, suppose that context is distinguished by call site and let s_i be an object allocation site in method m. Context-sensitive object naming for the objects allocated at s_i is achieved by using a separate object name for each call site j of m—that is, we have $h(s_i, j) = h_{ij}$. Intuitively, this scheme distinguishes the objects allocated at site s_i when m is invoked at call site j, from the objects allocated at site s_i when m is invoked at call site k.

3.3 Generalized Algorithm

The analysis solution is a *points-to graph Pt* where the nodes in the graph are taken from sets V and H, and the edges represent "may-refer-to" relationships as follows:

- An edge $\langle v, h\rangle$ means that at run-time some reference variable $r \in R$ represented by $v \in V$ may refer to some object represented by $h \in H$.
- Let $f \in F$ be a reference instance field in objects represented by some h. An edge $\langle h.f, h_2\rangle$ means that at run time field f of some object represented by $h \in H$ may refer to some object represented by $h_2 \in H$.
- If h represents array objects, $\langle h[\,], h_2\rangle$ shows that some element of some array represented by $h \in H$ may refer at run time to an object represented by $h_2 \in H$.

Analysis semantics can be defined by transfer functions that add new edges to these points-to graphs. Initially, $Pt = \emptyset$ and $\mathcal{C}_m = \emptyset$ for each method m except for the `main` method, for which we have $\mathcal{C}_{\texttt{main}} = \{\epsilon\}$. The analysis computes the closure of the empty graph under the application of all transfer functions for program statements. The transfer functions are presented in Figure 4.

For brevity, we discuss the analysis in this section as well as the analysis in Section 5 in terms of the kinds of statements listed below.[6] Other kinds of statements (e.g., constructor calls, static calls) are handled in a similar fashion.

Object allocation:

$$\mathcal{F}(Pt, \mathcal{C}_m, s_i\!: \ l = \mathit{new}\ A) = Pt \cup \bigcup_{c \in \mathcal{C}_m} \langle v(l, c), h(s_i, c)\rangle$$

Direct assignment:

$$\mathcal{F}(Pt, \mathcal{C}_m, l = r) = Pt \cup \bigcup_{c \in \mathcal{C}_m} \{\langle v(l, c), h\rangle \mid \langle v(r, c), h\rangle \in Pt\}$$

Indirect field write:

$$\mathcal{F}(Pt, \mathcal{C}_m, l.f = r) =$$
$$Pt \cup \bigcup_{c \in \mathcal{C}_m} \{\langle h.f, h_2\rangle \mid \langle v(l, c), h\rangle \in Pt \wedge \langle v(r, c), h_2\rangle \in Pt\}$$

Indirect field read:

$$\mathcal{F}(Pt, \mathcal{C}_m, l = r.f) =$$
$$Pt \cup \bigcup_{c \in \mathcal{C}_m} \{\langle v(l, c), h\rangle \mid \langle v(r, c), h_2\rangle \in Pt \wedge \langle h_2.f, h\rangle \in Pt\}$$

Virtual call:

$$\mathcal{F}(Pt, \mathcal{C}_m, i\!: \ l = r_0.n(r_{actual})) =$$
$$Pt \cup \bigcup_{c \in \mathcal{C}_m} \{\mathit{resolve}(Pt, n, i, c, h, r_{actual}, l) \mid \langle v(r_0, c), h\rangle \in Pt\}$$

```
$resolve(Pt, n, i, c, h, r_{actual}, l)$ =
 let $n_j(this, p_{formal}, ret_{n_j}) = dispatch(h, n)$ in
   $c' = findNewContext(i, h, c)$; add $c'$ to $\mathcal{C}_{n_j}$
   $\{\langle v(this, c'), h\rangle\} \cup$
   $\{\langle v(p_{formal}, c'), h_{actual}\rangle \mid \langle v(r_{actual}, c), h_{actual}\rangle \in Pt\} \cup$
   $\{\langle v(l, c), h_{ret}\rangle \mid \langle v(ret_{n_j}, c'), h_{ret}\rangle \in Pt\}$
```

Figure 4: Effects of program statements.

[6] Assumptions that the program consists of these kinds of statements are often used in the literature on program analysis in order to simplify the presentation. If necessary, temporary variables may be introduced to achieve these restrictions.

- Object creation: $l = \mathit{new}\ A$
- Direct assignment: $l = r$
- Indirect field write: $l.f = r$
- Indirect field read: $l = r.f$
- Virtual call: $l = r_0.n(r_{actual})$

Consider the transfer function for object allocation sites $l = \mathit{new}\ A$. It considers all contexts of the enclosing method $c \in \mathcal{C}_m$ and creates points-to edges from each $v(l, c)$ to the appropriate object name, namely $h(s_i, c)$. Similarly, for assignment statement $l = r$ the analysis considers all contexts $c \in C_m$ and infers that for each context c, the representative of l in c, namely $v(l, c)$, may refer to the objects that the representative of r in c, $v(r, c)$ refers to.

At virtual calls the transfer function considers all contexts of the enclosing method as well. For each context c, it performs resolution based on each receiver object h in the points-to set of $v(r_0, c)$; thus, it constructs the call graph on-the-fly based on the current class analysis information. Consider function *resolve*. First, function *dispatch* finds the run-time target n_j based on the class of h and compile-time target n. Second, function *findNewContext* takes the index of the call site i, the current object name h and the enclosing context c and computes a new context c' for n_j. For context-insensitive analysis *findNewContext* always returns ϵ. For a context-sensitive analysis that distinguishes context per call site, we have $\mathit{findNewContext}(i, h, c) = i$. Furthermore, for a context-sensitive analysis that distinguishes context by the last two enclosing call sites, we have $\mathit{findNewContext}(i, h, jk) = ij$. Other context sensitivity schemes can be modeled as well. For a context-sensitive analysis that distinguishes context by the receiver object (i.e., object-sensitive analysis), $\mathit{findNewContext}(i, h, c) = h$. Generally, analysis designers can specify a wide variety of context sensitivity schemes that take into account both the call site and the receiver object. Finally, *resolve* computes a set of edges due to flow from actuals to formals, and to flow from the return variable of n_j to the left-hand side of the call. Thus, h is propagated to $v(\mathit{this}, c')$ (i.e., the representative of implicit parameter `this` of n_j in context c'), the set for $v(r_{actual}, c)$ is propagated to $v(p_{formal}, c')$, and the set for $v(ret_{n_j}, c')$ is propagated to $v(l, c)$. This set of new edges is returned as the result of *resolve* and added to the points-to graph Pt.

Note that the transfer functions incorporate reachability. Initially all methods except `main` have $\mathcal{C}_m = \emptyset$ which means that they are not reachable in any context. When a method n_j becomes reachable in context c', the transfer functions are applied on the statements in n_j which adds new edges to the points-to graph.

3.4 Instances of the Generalized Algorithm

The generalized analysis outlined above can be instantiated to existing class analyses, ranging from the inexpensive RTA analysis to relatively expensive and precise context-sensitive analyses. Below, we present four representative instantiations: RTA, 0-CFA, Andersen-style points-to analysis and object-sensitive points-to analysis.

3.4.1 Rapid Type Analysis (RTA)

RTA is a popular form of class analysis primarily used for call graph construction [3]. Intuitively, it starts from `main`, and proceeds to compute a set of reachable methods and a set of instantiated classes. RTA analyzes two kinds of program statements in reachable methods: *call sites* which contribute new reachable methods, and *allocation sites* which contribute new instantiated classes. When RTA processes a call site, it examines all potential target methods according to the hierarchy and for each target method records the classes that trigger that target. If at least one of these classes is in the set of instantiated classes, the target becomes reachable. When RTA processes an allocation site that instantiates class A, it adds A to the set of instantiated classes and makes reachable all previously visited targets that are triggered by A. Note that in the context of our class analysis framework RTA can be regarded as computing a class set (i.e., the set of instantiated classes) for a single variable v [40]; intuitively, the analysis adds edges from v to the newly discovered instantiated classes. In order to find the set of classes for a variable $r \in R$ one can intersect the set of instantiated classes with the set of valid class types for r.

In order to instantiate our framework we need to define function *findNewContext* and the set of contexts $\mathcal{C}$, function $v(r, c)$ and the set of reference variable representatives V, and function $h(s_i, c)$ and the set of object names H. Clearly, RTA is a context-insensitive analysis and there is a single context, ϵ. Thus, we have that *findNewContext* equals ϵ and $\mathcal{C}$ equals $\{\epsilon\}$. Since RTA can be regarded as keeping a single variable v that represents *all* reference variables, we have that $v(r, \epsilon) = v$ for every r and thus $V = \{v\}$. Finally, RTA represents objects by their class. Thus, $h(s_i, \epsilon){=}A$ where A is the class of the object allocated at allocation site s_i and we have $H = C$.

To see that this analysis is equivalent to RTA, consider the instantiations of the transfer functions. First, reachability is incorporated through special context ϵ—a method m is reachable when $\mathcal{C}_m = \{\epsilon\}$ and not reachable when $\mathcal{C}_m = \emptyset$. Initially, we have $\mathcal{C}_{\texttt{main}} = \{\epsilon\}$ and $\mathcal{C}_m = \emptyset$ for every other m, which denotes that only `main` is initially reachable.

Next, consider the transfer functions for the two significant kinds of program statements: call sites, and allocation sites. At virtual calls, the transfer function states that if $\epsilon \in \mathcal{C}_m$ (i.e., m is reachable) $Pt = Pt \cup \{\mathit{resolve}(Pt, n, A) \mid \langle v, A\rangle \in Pt\}$. When the analysis processes a call site, it finds all targets n_j triggered by a class that is in the set of instantiated classes (i.e., we have $\langle v, A\rangle \in Pt$). At allocation sites, the transfer function states that if $\epsilon \in \mathcal{C}_m$ (i.e., m is reachable) $Pt = Pt \cup \langle v, A\rangle$. The processing of an allocation site triggers processing of *resolve* and thus the identification of new reachable target methods n_j. Clearly, the transfer functions for direct assignments, indirect writes and indirect reads have no effect on the RTA solution as they deteriorate to processing $v = v$, $v.f = v$ and $v = v.f$.

3.4.2 0-CFA

0-CFA [28, 40] is another well-known class analysis at the lower end of the cost/precision spectrum; it propagates sets of classes to reference variables and reference object fields. It is context-insensitive, maintains a set for each reference

Object allocation:
$\mathcal{F}(Pt, s_i\colon\ l = new\ A) = Pt \cup \langle l, h_i\rangle$

Direct assignment:
$\mathcal{F}(Pt, l = r) = Pt \cup \{\langle l, h\rangle \mid \langle r, h\rangle \in Pt\}$

Indirect field write:
$\mathcal{F}(Pt, l.f = r) = Pt \cup \{\langle h.f, h_2\rangle \mid \langle l, h\rangle \in Pt \wedge \langle r, h_2\rangle \in Pt\}$

Indirect field read:
$\mathcal{F}(Pt, l = r.f) = Pt \cup \{\langle l, h\rangle \mid \langle r, h_2\rangle \in Pt \wedge \langle h_2.f, h\rangle \in Pt\}$

Virtual call:
$\mathcal{F}(Pt, i\colon\ l = r_0.n(r_{actual})) =$
$Pt \cup \{resolve(Pt, n, h, r_{actual}, l) \mid \langle r_0, h\rangle \in Pt\}$

```
$resolve(Pt, n, h, r_{actual}, l)$ =
 let $n_j(this, p_{formal}, ret_{n_j}) = dispatch(h, n)$ in
   add $\epsilon$ to $\mathcal{C}_{n_j}$
   $\{\langle this, h\rangle\} \cup \{\langle p_{formal}, h_{actual}\rangle \mid \langle r_{actual}, h_{actual}\rangle \in Pt\} \cup$
   $\{\langle l, h_{ret}\rangle \mid \langle ret_{n_j}, h_{ret}\rangle \in Pt\}$
```

Figure 5: Andersen's points-to analysis for Java.

variable and represents objects by their class. For example, for statement $l = new\ A$ the analysis adds class A to the set for reference variable l. Similarly, for statement $l = r$, the analysis propagates the set of classes for variable r to the set of classes for variable l.

In our framework we have that *findNewContext* equals ϵ and $\mathcal{C}$ equals to $\{\epsilon\}$. Furthermore, $v(r, \epsilon) = r$ and $h(s_i, \epsilon) = A$ where A is the class of the object instantiated at allocation site s_i. Therefore, we have $V = R$ and $H = C$. 0-CFA is more precise than RTA because it keeps a separate analysis variable for every reference variable; in terms of the other dimensions, it uses the same representation as RTA.

We consider two other analyses at the lower end of the cost/precision spectrum; these analyses are inspired by the XTA-style analyses from [40]. The analyses vary the reference representation between the singleton representation of RTA (i.e., $v(r, \epsilon) = v$) and the R representation of 0-CFA (i.e., $v(r, \epsilon) = r$), while keeping $\mathcal{C}$ and H as in RTA and 0-CFA. Thus, the precision of these analyses varies between RTA and 0-CFA. The first instance, referred to as *mTA*, maps each reference variable r to a representative valid for the enclosing method m of r—that is, we have $v(r, \epsilon) = v_m$. The second instance, referred to as *cTA*, maps each variable r to a representative valid for the enclosing class of r—that is, we have $v(r, \epsilon) = v_c$. Therefore, RTA is the least precise analysis, followed by cTA, mTA and 0-CFA.

3.4.3 Andersen-style Points-to Analysis

So far, we considered analyses that represent heap objects by their class. Another group of class analyses, typically referred to as points-to analyses, represents heap objects more precisely, usually by allocation site. The Andersen-style points-to analysis for Java is a well-known flow- and context-insensitive points-to analysis [23, 5, 33, 37]. It uses an analysis variable for each reference variable and represents heap objects by their allocation site. In terms of our framework we have that the set of contexts includes only ϵ (i.e, *findNewContext* $= \epsilon$ and $\mathcal{C} = \{\epsilon\}$), $v(r, \epsilon) = r$ and $h(s_i, \epsilon) = h_i$; thus, we have $V = R$ and $H = S$. This analysis is at the heart of our implementation of ownership and composition inference; we believe that it is most suitable in terms of precision and scalability.

The instantiation of the transfer functions from Figure 4 for the Andersen-style points-to analysis for Java is given in Figure 5. To simplify the presentation of these functions, we have omitted the contexts $\mathcal{C}_m$ from the transfer functions; the actual analysis applies transfer functions only on statements in reachable methods (i.e., $\mathcal{C}_m = \{\epsilon\}$). The effects of the functions are as follows. At object allocation sites the analysis adds an edge from l to h_i, the object name that corresponds to the objects created at allocation site s_i. Similarly, at direct assignments, the analysis creates edges from l to each h in the set of r. At virtual calls, the analysis considers each h in the set of r_0. It performs dispatch for h and compile-time target n, and finds a run-time target n_j. Subsequently, ϵ is added to the set of contexts $\mathcal{C}_{n_j}$ of n_j (i.e., n_j becomes reachable). Finally, the analysis takes into account frow from actuals to formals, and from the return variable to the left-hand side of the call.

3.4.4 Object-sensitive Points-to Analysis

Finally, we consider a context-sensitive points-to analysis. Specifically, we consider *object-sensitive* points-to analysis [25, 26]. With object sensitivity, each instance method is analyzed separately for each object on which this instance method may be invoked. Roughly, if an instance method is invoked on objects represented by object name h, the object-sensitive analysis maintains a separate contextual version of that method for contexts derived from h. For static methods, the analysis uses the special context ϵ.

In terms of our framework, the analysis is defined as follows. The set of contexts $\mathcal{C}$ equals S—that is, the set of contexts is the set of object allocation sites. Map $v(r, c) = r^c$ where $r^c \in V$; thus, we have $V = R \times \mathcal{C} = R \times S$. Intuitively, r^c is the context copy of r which corresponds to the invocation of the enclosing method of r in context c. Similarly, map $h(s_i, c) = h_{ic}$ where h_{ic} is in the set of context-sensitive object names H; thus we have $H = S \times S = S^2$. Intuitively, h_{ic} represents the objects that are allocated at the site of s_i when the enclosing method is invoked in context c—for example, object name h_{ij} represents all objects allocated at site s_i when the enclosing method is invoked on an object allocated at site s_j.

The meaning of most of the transfer functions is straightforward. At object allocation sites the analysis examines all contexts $c \in \mathcal{C}_m$ and for each c computes $v(l, c)$ and $h(s_i, c)$ and creates an edge from $v(l, c)$ to $h(s_i, c)$. For example, if m is invoked on an object allocated at site s_j we have $s_j \in \mathcal{C}_m$; furthermore, $v(l, s_j) = l^j$ and $h(s_i, s_j) = h_{ij}$ and the analysis creates an edge $\langle l^j, h_{ij}\rangle$.

At virtual calls, the analysis resolves the call based on the heap object h_{kl} and the compile-time target n. Recall that this analysis distinguishes context by the *allocation site* of the receiver object. Thus, *findNewContext*$(i, c, h_{kl}) = h_k$—that is, the new context c' for the run-time target n_j is h_k which denotes that n_j is analyzed separately for the set of receivers allocated at site s_k. Further, edge $\langle this^{h_k}, h_{kl}\rangle$ is added to the points-to graph Pt. Also, edges $\langle p^{h_k}_{formal}, h_{actual}\rangle$ are added for every h_{actual} in the points-to set of r^c_{actual} to

```
void main() {
    ZipEntry ph_ZE;
    ZipInputStream ph_ZIS;
    ZipOutputStream ph_ZOS;
    ph_ZE = new ZipEntry();
    ph_ZIS = new ZipInputStream();
    ph_ZOS = new ZipOutputStream();
    ph_ZE.setCRC(0);
    ph_ZE = ph_ZIS.getNextEntry();
    ph_ZOS.putNextEntry(ph_ZE);
    ph_ZOS.closeEntry();
    ph_ZOS.finish();
}
```

Figure 6: Placeholder `main` method for `zip`.

account for flow from actuals to formals. Similarly, edges $\langle l^c, h_{ret}\rangle$ are added for every h_{ret} in the points-to set of $ret_{n_j}^{h_k}$ to account for flow from the return variable of n_j to the left-hand side of the enclosing method.

Note that this analysis maintains a context-sensitive variable representation and a context-sensitive object naming scheme. This is just one particular instantiation of the framework. One can define a wide variety of other representation schemes; most importantly, one can define *targeted* sensitivity schemes, where the analysis uses a precise (and more expensive) representation only for a subset of all variables and objects, and a less precise (and less expensive) representation for the majority of variables and objects.

4. FRAGMENT CLASS ANALYSIS

Class analyses are typically designed as *whole-program analyses*; they take as input a complete program and produce points-to graphs that reflect relationships in the entire program. However, the problem considered in this paper requires class analysis information for a partial program. The input is a set of classes *Cls* and the analysis needs to construct an approximate object graph that is valid across all possible executions of arbitrary client code built on top of *Cls*. To address this problem we make use of a general technique called *fragment analysis* due to Nasko Rountev [31, 34, 32]. Fragment analysis works on a program fragment rather than on a complete program; in our case the fragment is the set of classes *Cls*.

Initially, the fragment analysis produces an artificial `main` method that serves as a placeholder for client code written on top of *Cls*. Intuitively, the artificial `main` simulates the possible flow of objects between *Cls* and the client code. Subsequently, the fragment analysis attaches `main` to *Cls* and uses some whole-program class analysis engine to compute a points-to graph which summarizes the possible effects of arbitrary client code. The fragment analysis approach can be used with a wide variety of class analyses, including all analyses described in Section 3.4.

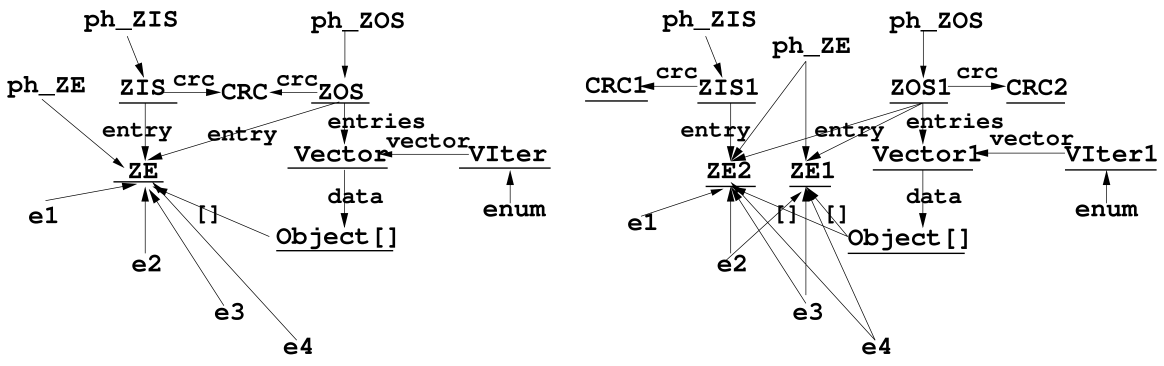
The placeholder `main` method for the classes from Figure 3 is shown in Figure 6. The method contains variables for types from *Cls* that can be accessed by client code. The statements represent different possible interactions involving *Cls*; their order is irrelevant because the whole-program analysis is flow-insensitive. Method `main` invokes all public methods from the classes in *Cls* designated as accessible.

The details of the fragment analysis will not be discussed here; they can be found in [34]. For the purposes of our analysis we discuss the *object reachability* [32] property of the results computed by the fragment analysis. Consider some client program built on top of *Cls* and an execution of this program (the program must satisfy the constraints discussed in Section 2.3). Let $r \in R$ be a variable declared in *Cls* and at some point during execution r is the start of a chain of object references that leads to some heap object. In the fragment analysis solution, there will be a chain of points-to edges that starts at the representative of r, $v \in V$ and leads to some object name $h \in H$ that represents the run-time object. A similar property holds if r is declared outside of *Cls*. In this case, in the fragment analysis solution, the starting point of the chain is the representative of the variable from `main` that has the same type as r. This property is relevant for the ownership and composition analysis described in Section 6 as the points-to graph is used to approximate all possible object graphs and thus all possible accesses must be taken into account.

Consider the code from Figures 3 and 6. There are three allocation sites in the `main` method; they are denoted by names `ZE1`, `ZIS1` and `ZOS1`. The allocation site in class `InflaterInputStream` is denoted by name `byte[]`. There are three allocation sites in class `ZipInputStream`; they are denoted by names `CRC1`, `Inflater1` and `ZE2`. There are four allocation sites in class `ZipOutputStream`; they are denoted by `Vector1`, `Hashtable1`, `Deflater1` and `CRC2`. In addition, we consider the allocation sites in `Vector` (recall Figure 1), which are transitively reachable; they are denoted by `Object[]` and `VIter1`. The corresponding classes are denoted in identical abbreviated fashion; for example, we use `ZE` to denote class `ZipEntry`, `ZIS` to denote `ZipInputStream`, `ZOS` to denote `ZipOutputStream` and `CRC` to denote `CRC32`.

The points-to graphs computed from the code in Figures 6, 3 and 1 when the generalized class analysis algorithm is instantiated to 0-CFA and to the Andersen-style points-to analysis are shown in Figures 7(a) and 7(b). Object names are underlined in order to distinguish them from reference variables. For simplicity, implicit parameters `this`, and the `Inflater`, `byte[]`, `Hashtable` and `Deflater` objects are not shown.

Recall that both 0-CFA and the Andersen-style points-to analysis use an analysis variable for each reference variable; the two analyses use the same set of reference variable representatives. Variable `e1` denotes variable `readLOC.e` (i.e., local variable `e` in method `readLOC` in `ZipInputStream`); `e2` denotes `putNextEntry.e`, `e3` denotes `closeEntry.e` and `e4` denotes `Vector.addElement.e`. Variable `enum` denotes `finish.enum`. The rest of the variables are the placeholder variables declared in `main` by the fragment analysis. In the case of 0-CFA (Figure 7(a)), the objects are represented by their corresponding classes. As a result, there is a single name for the two `CRC` objects and a single name for the `ZipEntry` objects. In the case of the Andersen-style points-to analysis (Figure 7(b)) the analysis represents objects by their allocation site. Therefore, there are separate object names, `CRC1` and `CRC2` for the two `CRC` objects, and there are two object names, `ZE1` and `ZE2` for the `ZipEntry` objects.



(a) Points–to graph for 0–CFA analysis

(b) Points–to graph for Andersen–style points–to analysis

Figure 7: Points-to graphs computed by the fragment points-to analysis.

5. APPROXIMATE OBJECT GRAPH

The output of the fragment class analysis is needed to construct the *approximate object graph* Ag which approximates all possible run-time object graphs that can happen when client code is built on top of *Cls*. Subsequently, Ag is used for ownership inference. The nodes in Ag are taken from the set of object names H and the edges represent "may-access" relationships. Clearly, the precision of the object graph depends on the precision of the underlying class analysis.

Section 5.1 outlines the construction of the approximation of the original object graph and Section 5.2 outlines the changes needed to obtain a relaxed object graph (recall Section 2.1).

5.1 Original Object Graph

5.1.1 Preliminaries

Figure 8 outlines the construction of Ag given a points-to graph Pt. The construction analysis processes program statements that contribute edges to the approximate object graph. Recall that set $\mathcal{C}$ represents the contexts of invocation of a method for the purposes of class analysis. The construction analysis processes each statement in each applicable context. For analyses that are context-insensitive (e.g., RTA, 0-CFA and the Andersen-style points-to analysis) a statement s is processed once. For analyses that are context-sensitive (e.g., the object-sensitive points-to analysis) a statement s is processed multiple times, once for each context of the enclosing method of s.

Set $\mathcal{RC}_{m,c}$ denotes the set of object names that represent the receivers of m when m is invoked in context c; these sets are pre-computed based on class analysis information. If m is an instance method or constructor, $\mathcal{RC}_{m,c}$ is the points-to set of $v(\texttt{this}, c)$ (i.e., the points-to set of the context copy of implicit parameter `this` for context c). If m is a static method, there is a single set for m, namely $\mathcal{RC}_{m,\epsilon}$. $\mathcal{RC}_{m,\epsilon}$ includes the union of the points-to sets of `this` for all instance methods and contexts that may call m (directly or through a sequence of static calls); it includes `root` if m is `main` or may be called through a sequence of static calls from `main`. Recall from Section 2.1 that there is a run-time access edge from o to o' if a method m invoked on receiver o has a local variable that refers to an object o'. The $\mathcal{RC}$ sets are needed to approximate the receivers of m and thus the sources of these access edges.

```
input   $Stmt$: set of statements   $Pt$: $V \cup H \rightarrow \mathcal{P}(H)$
output  $Ag : H \rightarrow \mathcal{P}(H)$
[1]  foreach statement $s \in Stmt$ in method $m$
       $s: l = new\ A(...)$
       $s: l = r.m(...)$ s.t. $r \neq \texttt{this}$,
       $s: l = r.f$ s.t. $r \neq \texttt{this}$
[2]    foreach context $c \in \mathcal{C}_m$
[3]      add $\{h \rightarrow h_j \mid h \in \mathcal{RC}_{m,c} \wedge \langle v(l,c), h_j\rangle \in Pt\}$ to $Ag$
       // add access edges due to flow from callees to callers
[4]  foreach statement $s \in Stmt$ in method $m$
       $s: l = new\ A(r)$,
       $s: l.m(r)$ s.t. $l \neq \texttt{this}$,
       $s: l.f = r$ s.t. $l \neq \texttt{this}$
[5]    foreach context $c \in \mathcal{C}_m$
[6]      add $\{h_i \rightarrow h_j \mid \langle v(l,c), h_i\rangle \in Pt \wedge \langle v(r,c), h_j\rangle \in Pt\}$ to $Ag$
       // add access edges due to flow from callers into callees
[7]  foreach $s \in Stmt$ in method $m$ with a this access
[8]    foreach context $c \in C_m$
[9]      add $\{h \rightarrow h \mid \langle v(\texttt{this}, c), h\rangle \in Pt\ \}$ to $Ag$
       // add self-loops due to this access (e.g., $r.n(\texttt{this})$)
[10] label $h_i \rightarrow h_j \in Ag$ with $f$ if $\langle h_i.f, h_j\rangle \in Pt$
```

Figure 8: Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty.

During the description of the algorithm we consider an underlying class analysis that is based on the Andersen-style points-to analysis. Therefore, the construction of the object graph is context-insensitive and we have $\mathcal{C} = \{\epsilon\}$. In this case, the algorithm in Figure 8 constructs the object graph in Figure 2(a) from the code in Figure 1; we use this code as a running example throughout the description of the algorithm.

5.1.2 Algorithm Description

Recall that for presentation purposes we assume that the program consists of the kinds of statements given in Section 3. The algorithm in Figure 8 groups these statements in three categories: first, statements that contribute edges due to flow from a callee to a caller (these statements are processed at lines 1-3), second, statements that contribute edges due to flow from a caller to a callee (lines 4-6), and third, statements that contribute special self-access edges (lines 7-9). Note that a statement may fall in more than one category. For example, a virtual call statement $l = r.m(r_{actual})$ is typically processed both at lines 1-3 (as $l = r.m(...)$), and at lines 4-6 (as $l.m(r)$). Below we describe in detail the processing of each category.

Lines 1-3 in Figure 8 account for edges due to flow from a callee method to a caller method. For example, at a constructor call new edges are added to Ag from each receiver h of the method enclosing the call, to the name representing the newly created object. Similarly, at an instance call not through `this`, new edges are added from receiver h of the method enclosing the call, to each returned object h_j; intuitively, the returned object has been accessible to the receiver of the callee and after it is returned it becomes accessible to the receiver of the caller. Recall the code in Figure 1. Lines 1-3 of the algorithm are applied on code lines 11, 12 and 14 in `main`, on code line 1 in constructor `Vector`, on code line 4 in `elements`, and on code lines 7 and 10 in `nextElement`. Consider method `main`. We have that $\mathcal{RC}_{\texttt{main}} = \{\texttt{root}\}$ (since we consider context-insensitive analysis, we omit the context from the $\mathcal{RC}$ notation because it is ϵ in all cases). Therefore, there are edges in the object graph `root`→`Vector` due to the constructor call at line 11, `root`→`X` due to the constructor call at line 12, and `root`→`VIterator` due to code line 14, as we have that $Pt(e) = \{\texttt{VIterator}\}$. Next, consider constructor `Vector`. We have that $\mathcal{RC}_{\texttt{Vector}} = Pt(\texttt{this}_{\texttt{Vector}}) = \{\texttt{Vector}\}$. Therefore, we have edge `Vector`→`Object[]` due to the constructor call at line 1. Next, consider method `elements`. We have that $\mathcal{RC}_{\texttt{elements}} = Pt(\texttt{this}_{\texttt{elements}}) = \{\texttt{Vector}\}$. Therefore, there is an edge `Vector`→`VIterator` due to the constructor call in line 4. Finally, consider method `nextElement`. We have that $\mathcal{RC}_{\texttt{nextElement}} = Pt(\texttt{this}_{\texttt{nextElement}}) = \{\texttt{VIterator}\}$. Thus, we have edge `VIterator`→`Object[]` due to the instance field read at line 7 as $Pt(data) = \{\texttt{Object[]}\}$, and edge `VIterator`→`X` due to the instance field read at line 10, as $Pt(data[i]) = \{\texttt{X}\}$.

Lines 4-6 account for edges due to flow from the caller method to the callee method. For example, at an instance call, edges are added from each object h_i in the points-to set of the receiver of the callee, to each object h_j in the points-to set of a reference argument; in this case, the object h_j has been accessible to the receiver of the caller, and at the call it is passed and becomes accessible to the receiver h_i of the callee. Continuing with the example in Figure 1, lines 4-6 in the algorithm are applied on code line 13 in `main`, on code line 2 in `addElement`, and on code line 4 in `elements`. Consider method `main`. We have edge `Vector`→`X` due to the virtual call at line 13, since $Pt(v) = \{\texttt{Vector}\}$ and $Pt(x) = \{\texttt{X}\}$. Next, consider method `addElement`. We have edge `Object[]`→`X` due to the instance field write at line 2 since $Pt(data) = \{\texttt{Object[]}\}$ and $Pt(e) = \{\texttt{X}\}$. Finally, consider code line 4. It leads to an edge `VIterator`→`Vector` since we have $Pt(\texttt{this}_{\texttt{elements}}) = \{\texttt{Vector}\}$.

Lines 7-9 create self-loops due to a reference through implicit parameter `this` (e.g., $x.m(...\texttt{this}...)$). Intuitively, the object has access to itself through `this` and may pass itself to other objects; it is important to account for such access when reasoning about ownership. In our running example, line 4 leads to a self loop around `Vector` (this edge is not shown in the graph in Figure 2(a)). Finally, line 10 labels the edges with the appropriate field identifier. For clarity, we omit detailed discussion of static fields. The actual implementation creates edges from `root` to each object in the points-to set of a static field; the case is handled correctly by this algorithm and by the algorithm in Section 6.1.

```
[Case 1a] // an object is created and immediately passed
[1] foreach statement $s \in Stmt$ in method $m$
      $s: l = new\ B(\mathbf{new\ A(r)})$,
      $s: l.m(\mathbf{new\ A(r)})$ s.t. $l \neq \texttt{this}$,
      $s: l.f = \mathbf{new\ A(r)}$ s.t. $l \neq \texttt{this}$
[2]  foreach context $c \in \mathcal{C}_m$
[3]    add $\{h(h_A, c) \rightarrow h_j) \mid \langle v(r,c), h_j \rangle \in Pt\}$ to $Ag$
[4]    add $\{h_i \rightarrow h(h_A, c) \mid \langle v(l,c), h_i \rangle \in Pt\}$ to $Ag$
[5]    add $\{h_i \rightarrow h_j \mid \langle v(l,c), h_i \rangle \in Pt \wedge \langle v(r,c), h_j \rangle \in Pt\}$ to $Ag$

[Case 1b] // an object is created and immediately returned
[6] foreach statement $s \in Stmt$ in method $m$
      $s: return\ \mathbf{new\ A(r)}$
[7]  foreach context $c \in \mathcal{C}_m$
[8]    add $\{h(h_A, c) \rightarrow h_j \mid \langle v(r,c), h_j \rangle \in Pt\}$ to $Ag$
[9]    add $\{h_i \rightarrow h(h_A, c) \mid h_i \in \mathcal{RC}_{Callers(m,c)}\}$ to $Ag$
[10]   add $\{h_i \rightarrow h_j \mid h_i \in \mathcal{RC}_{Callers(m,c)} \wedge \langle v(r,c), h_j \rangle \in Pt\}$ to $Ag$

[Case 2] // temporary access with owner in scope
[11] foreach statement $s \in Stmt$ in method $m$
      $s: l = r.f$ s.t. $r \neq \texttt{this}$ and $l$ is never assigned
      (i.e., we have only statements $l.f = l'$ and $l.f = l'$)
[12]  foreach context $c \in \mathcal{C}_m$
[13]    add $\{h_i \rightarrow h_j \mid \langle v(r,c), h_i \rangle \in Pt \wedge \langle v(l',c), h_j \rangle \in Pt\}$ to $Ag$
```

Figure 9: Relaxing the construction of Ag.

We discuss the *reachability* property of the approximate object graph. Consider some client program built on top of *Cls* and an execution of this program (the program must satisfy the constraints discussed in Section 2.3). Let c be `root` or a heap object and at some point during execution we have that c is the start of a chain in the object graph that leads to some heap object o. In Ag, there will be a chain of edges that starts at the representative of c and leads to the representative of o.

5.2 Relaxed Object Graph

5.2.1 Algorithm Description

Recall that Section 2.1 identifies two kinds of temporary access relationships: (1) when an object is created by one object and immediately passed to another one (e.g., **new A** in $new\ B(\mathbf{new\ A(r)})$), and (2) when an object is temporarily accessed while its owner remains in scope (e.g., as in $data = vector.data$ when the iterator accesses the vector elements while the vector remains in scope). When constructing the relaxed object graph, we identify statements **new A** for which (1) occurs and relax the processing of these statements —that is, we avoid creation of edges from the receivers of the enclosing method to the newly created A object. The processing of such **new A** is done as specified in Figure 9 (Case 1a and Case 1b), instead of as in Figure 8. Similarly, we identify statements $\mathbf{l} = \mathbf{r.f}$ for which (2) occurs and avoid the creation of edges from the receivers of the enclosing method to the objects in the points-to set of l. The processing of such $\mathbf{l} = \mathbf{r.f}$ is done as specified in Figure 9 (Case 2), instead of as in Figure 8.

Consider Case 1a. It handles statements **new A** statements for which the newly created A object is passed immediately to another object as an actual argument. We find such statements by local analysis that identifies the following sequences: $l = new\ B(\mathbf{new\ A(r)})$, $l.m(\mathbf{new\ A(r)})$, and $l.r = \mathbf{new\ A(r)}$. Consider sequence $l = new\ B(\mathbf{new\ A(r)})$ and let h_A denote the newly created A object and h_B denote the newly created B object. Line 3 accounts for the creation of edges from h_A to the objects in the points-to set of r. Line 4 accounts for the creation of an edge from h_B to h_A. This edge is added here only for clarity—statement $l = new\ B(l_1)$; itself is processed in Figure 8 and that processing accounts for this edge. Finally, line 5 adds edges from h_B to the objects in the points-to set of r; these edges are needed to ensure the correctness of the subsequent ownership and composition inference from Section 6. Note that this processing of $\mathbf{new\ A(r)}$ avoids the creation of edges from the receivers of m to h_A which is our goal.

Consider Case 1b. It handles statements **newA** for which the newly crated A object is immediately returned. The effect of lines 5-10 is similar to that of lines 1-5. Line 8 is analogous to line 3; it adds edges from h_A to the objects in the points-to set of r. At line 9 we use notation $\mathcal{RC}_{Callers(m,c)}$ which extends the $\mathcal{RC}$ notation as follows: $\mathcal{RC}_{Callers(m,c)} = \bigcup_{m',c' \in Callers(m,c)} \mathcal{RC}_{m',c'}$. Intuitively, m' in context c' is the caller of m in context c and set $\mathcal{RC}_{Callers(m,c)}$ contains all appropriate receivers. For example, in Figure 1 method `elements` is called from method `main`; therefore, we have $\mathcal{RC}_{Callers(\texttt{elements},\epsilon)} = \{\texttt{root}\}$ (assuming a context-insensitive analysis). Line 9 adds edges $h_i \rightarrow h_A$ for each $h_i \in \mathcal{RC}_{Callers(m,c)}$. Note that for non-`this` calls to m, these edges are already captured. In Figure 1, the edge from `root` to `VIterator` is captured by code line 14. The $\mathcal{RC}_{Callers(m,c)}$ sets are needed to account for `this` calls to m. To see why, recall that the original construction analysis does not process `this` calls as they do not contribute new edges; however, the edge to h_A must be recorded, because h_A becomes accessible to the receiver of m through a `this` call to m in a caller m'. Finally, line 10 adds edges $h_i \rightarrow h_j$ analogously to line 5, which again is needed to ensure correctness of the ownership and composition inference.

Finally, consider Case 2. Recall that the algorithm for construction of the original object graph processes an indirect read $l = r.f$ by creating edges from each receiver h of the enclosing method to each object h' in the points-to set of l even if the access is only temporary and the owner is still in scope. In the construction of the relaxed object graph these edges are omitted for indirect reads $l = r.f$ for which l is never assigned or passed as an argument—that is, l may only be used in statements of the form $l' = l.f$ and $l.f = l'$. Again, the edges in line 13 are needed to ensure correctness of the ownership and composition inference.

5.2.2 *Example*

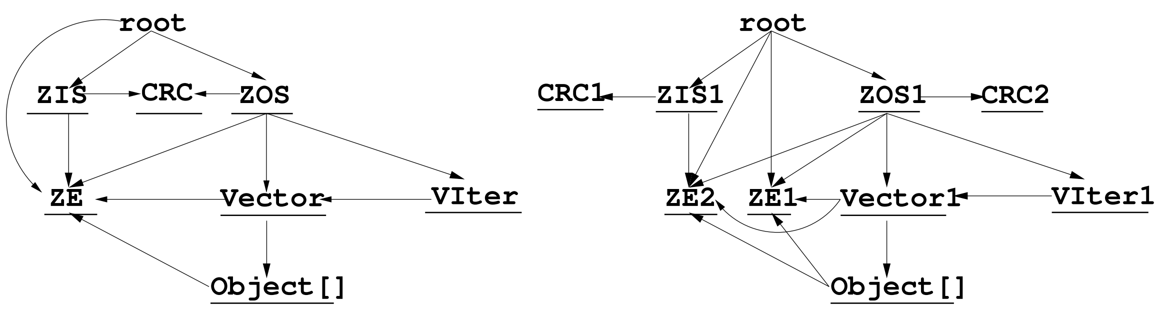
Figure 10 shows the approximate object graphs computed from the code on Figures 3, 6 and 1, and the points-to graphs in Figure 7 (only object names from Figure 7 are shown and fields are omitted for clarity). The graph in Figure 10(a) is based on 0-CFA analysis and the graph in Figure 10(b) is based on the Andersen-style points-to analysis. Thus, the nodes in the graph in Figure 10(a) are taken from the set of classes C while the nodes in the graph in Figure 10(b) are taken from the set of object names corresponding to allocation sites.

Consider the graph in Figure 10(b). For the majority of statements inference is done with the construction analysis in Figure 8. Consider method `main` in Figure 6. The three constructor calls lead to edges `root`→`ZE1`, `root`→`ZIS1` and `root`→`ZOS1` (lines 1-3 in the construction analysis in Figure 8). Statement $ph\_ZE{=}ph\_ZIS.getNextEntry()$ leads to edge `root`→`ZE2` and edge `root`→`ZE1` as we have that $Pt(ph\_ZE) = \{\texttt{ZE1}, \texttt{ZE2}\}$ (lines 1-3 in Figure 8); however, note that the second edge is redundant since it was already created due to the first constructor call in `main`. Statement $ph\_ZOS.putNextEntry(ph\_ZE)$ leads to edges `ZOS1`→`ZE2` and `ZOS1`→`ZE1` (lines 4-6 in Figure 8). Next, consider the methods in class `ZipInputStream` in Figure 3. Statement $crc = new\ CRC32()$ leads to edge `ZIS1`→`CRC1` and statement $e = new\ ZipEntry()$ in `readLOC` leads to `ZIS1`→`ZE2` (lines 1-3 in Figure 8). Next, consider the methods in `ZipOutputStream`. Statement $entries = new\ Vector()$ leads to `ZOS1`→`Vector1` and statement $crc = new\ CRC32()$ leads to `ZIS1`→`CRC2` (lines 1-3 in Figure 8). Statement $entries.addElement(e)$ in `putNextEntry` leads to `Vector1`→`ZE1` and `Vector1`→`ZE2` as we have that $Pt(entries) = \{\texttt{Vector1}\}$ and $Pt(e) = \{\texttt{ZE1,ZE2}\}$. Statement $enum = entries.elements()$ leads to edge `ZOS1`→`VIter1`. Furthermore, consider again the methods in classes `Vector` and `VIterator` in Figure 1. Statement $data = new\ Object[size]$ leads to edge `Vector1`→`Object[]` (lines 1-3 in Figure 8), and statement $data[i] = e$ leads to edges `Object[]`→`ZE1` and `Object[]`→`ZE2` (lines 4-6 in Figure 8). Finally, recall statement $return\ new\ VIterator(this)$ in `elements`. It is processed as in Figure 9 (Case 1b) instead of as in Figure 8 and as a result the analysis avoids creating an edge from `Vector1` to `VIter1`. This statement contributes only one new edge: `VIter1`→`Vector1` (line 8 in Figure 9).

6. COMPOSITION INFERENCE

Recall from Section 2.1 that the run-time object graph defines an ownership boundary for each run-time object o—intuitively, this is the subgraph rooted at o which includes all run-time paths that are dominated by o. The analysis goal therefore, is to use the approximate object graph Ag and identify an approximate ownership boundary for each analysis object h; the approximate ownership boundary contains paths that are guaranteed to represent internal flow. If we have that an edge $h{\rightarrow}h_j \in Ag$ appears in the approximate ownership boundary of its source h, it is guaranteed that for each run-time access edge $o \rightarrow o_j$, represented by $h \rightarrow h_j$, o dominates o_j and thus o owns o_j. Subsequently, this information is used to identify composition relationships.

For the rest of the paper we use the following notational convention: we use notation o, o_i, o_1, etc., to denote run-time objects, and notation h, h_i, h_1, etc. to denote their corresponding analysis names. For brevity we often use terms



(a) Object graph based on the 0−CFA points−to graph

(b) Object graph based on the Andersen−style points−to graph

Figure 10: Approximate object graphs computed by the algorithm in Figure 8.

```
procedure findClosureSet // of $h \to h_j$ w.r.t. $h_i$
input      $Ag$: $H \to \mathcal{P}(H)$   $h \to h_j$: $H \times H$   $h_i$: $H$   $n$: $Int$
output     $Closure(h_i,n)$: $\mathcal{P}(H \times H)$   $Prt(h_i,n)$: $\mathcal{P}(H \times H)$
initialize $Wl$={}, $Closure(h_i,n)$={}, $Prt(h_i,n)$={}
[1] mark $h \to h_j$, add it to $Wl$ and to $Closure(h_i,n)$
[2] while $Wl$ not empty
[3]   remove $h \to h_j$ from $Wl$
[4]   foreach $h_k \to h_j$ s.t. $h_k \to h$ and $h_k$ reachable from $h_i$
[5]     if $h_k \to h_j$ is unmarked
[6]       mark $h_k \to h_j$, add it to $Wl$ and $Closure(h_i,n)$
[7]       add $h_k \to h$ to $Prt(h_i,n)$
[8]   foreach $h_k \to h_j$ s.t. $h \to h_k$
[9]     if $h_k \to h_j$ is unmarked
[10]      mark $h_k \to h_j$, add it to $Wl$ and $Closure(h_i,n)$
[11]      add $h \to h_k$ to $Prt(h_i,n)$
```

```
procedure computeBoundary // of $h_i$
input      $Ag$: $H \to \mathcal{P}(H)$     $h_i$: $H$
output     $Bndry(h_i)$: $\mathcal{P}(H \times H)$
initialize $n$=0
[1] foreach unmarked edge $h \to h_j$ reachable from $h_i$
[2]   findClosureSet($h \to h_j$,$o_i$,$n$++)
[3] foreach $h_i \to h_j$ s.t. $\exists h_k$ s.t. $h_k \to h_i$ and $h_k \to h_j$
[4]   mark the $Closure$ set of $h_i \to h_j$ as forbidden
[5] while empty $Prt(h_i,k)$ and $Closure(h_i,k)$ not forbidden
[6]   add $Closure(h_i,k)$ to $Bndry(h_i)$
[7]   foreach $e \in Closure(h_i,k)$ remove $e$ from each $Prt$
[8]   remove $Prt(o_i,k)$ and $Closure(h_i,k)$
```

Figure 11: Ownership analysis.

ownership boundary and boundary to refer to the approximate ownership boundary.

This section is organized as follows. Section 6.1 describes the computation of the approximate ownership boundary and Section 6.2 describes the identification of composition relationships.

6.1 Ownership Boundary

6.1.1 Preliminaries

Procedure **computeBoundary** in Figure 11 takes Ag and object name h_i as input and outputs subgraph $Bndry(h_i)$. Subgraph $Bndry(h_i)$ contains paths that are guaranteed to represent internal flow for each o_i represented by h_i. More precisely, we have the following *lemma*. Let o_i be a heap object represented by h_i. For every edge e: $h \to h_j \in Bndry(h_i)$ we have that if o_i dominates o (represented by h), then o_i dominates the o_j (represented by h_j) that o refers to. A corollary of this lemma is that for every o_i and run-time path p: $o_i \to ...o \to o_j$, with representative in $Bndry(h_i)$, we have that o_i dominates o_j.

Consider the object graph in Figure 10(b). The boundary of ZOS1 includes nodes ZOS1,CRC2,Vector1,Object[] and VIter1 and the edges between them. There are paths ZOS1→CRC2, ZOS1→VIter1, ZOS1→Vector1, ZOS1→VIter1 →Vector1, ZOS1→Vector1→ Object[] and ZOS1→VIter1 →Vector1→Object[]. It is easy to see that for example for every run-time $\texttt{ZOS1}^r \to \texttt{Vector1}^r$, $\texttt{ZOS1}^r$ dominates $\texttt{Vector1}^r$.[7] As another example, consider the object graph in Figure 10(a). In this case the boundary of ZOS includes nodes ZOS,CRC,Vector,Object[] and VIter. Although the graph in Figure 10(a) uses a coarser, less precise representation of run-time objects than the graph in Figure 10(b), this boundary captures exactly the same paths and dominance relations as the boundary for the more precise graph.

6.1.2 Algorithm Description

Below we outline the algorithm and the correctness argument. The algorithm uses the fact that o_j flows from object o_i to some object o_k only if one of the following is true: (1) o_k has a handle to both o_i and o_j (and due to the reachability property Ag contains edges $h_k \to h_i$, $h_k \to h_j$, $h_i \to h_j$), or (2) o_i has a handle to both o_k and o_j (and Ag contains edges $h_i \to h_k$, $h_i \to h_j$, $h_k \to h_j$). This observation helps identify encapsulation more precisely. Consider the graph in Figure 10(a). A naive algorithm may identify root as the dominator of the CRC objects, and fail to identify the composition relationship between ZipInputStream and CRC32. In fact, the CRC object is created and dominated by its enclosing ZIS object because there is no h_k such that either (1) h_k has handles to both ZIS and CRC, or (2) ZIS has handles to both h_k and CRC. Thus, edge ZOS → CRC represents that the ZOS object creates and exclusively owns the CRC. This

[7]Superscripts r are used to denote run-time instances of classes ZipOutputStream and Vector.

example underscores the need for a specialized dominance analysis: using a standard dominance analysis on *Ag* will likely lead to imprecise (and even incorrect) results because multiple run-time objects are represented by a single analysis object name (e.g., in Figure 10(a) the two distinct CRC objects, for the `ZipInputStream` and the `ZipOutputStream` objects, are represented by a single name).

The algorithm builds the boundary of an object name h_i by adding edges. Lines 1-2 in **computeBoundary** partitions the edges reachable from h_i into closure sets using auxiliary procedure **findClosureSet**. Intuitively, the closure set of edge $h \rightarrow h_j$ contains all edges $h_k \rightarrow h_j$ in the transitive closure of h_i, such that some o_k and o may refer to the same o_j. For example, in Figure 10(b), the closure set of `ZOS1`→`Vector1` is {`ZOS1`→`Vector1`,`VIter1`→ `Vector1`} and the closure set of `ZOS1`→`ZE1` is {`ZOS1`→`ZE1`,`Vector1`→`ZE1`, `Object[]`→`ZE1`}. Procedure **findClosureSet** computes a parent set *Prt* for each closure set *Closure*. The role of the parent set is to ensure that the relevant paths to $h \rightarrow h_j$ stay in boundary; the parent sets will be discussed in detail throughout the description of the algorithm. In the above example the parent set of `ZOS1`→`Vector1` is {`ZOS1`→`VIter1`}. Procedure **findClosureSet** indexes the closure/parent pairs of sets.

Furthermore, lines 3-4 mark some of the closures sets as forbidden. Intuitively, a forbidden closure set $\{h_1 \rightarrow h_j, h_2 \rightarrow h_j, ..., h_n \rightarrow h_j\}$ denotes that for some run-time instance o_i some o_j may flow out of the ownership boundary of o_i (i.e., o_i may not own o_j). For example, consider the object graph in Figure 10(a) and the closure set of edge `ZOS` → `ZE`. This closure set consists of edges `ZOS`→`ZE`,`Vector`→`ZE` and `Object[]`→`ZE`. It is marked as forbidden because of edge triple `root`→`ZE`, `root`→`ZOS`, and `ZOS`→`ZE` which indicates that the `ZE` object accessible within the `ZOS` object may flow from `root` to the `ZOS` object (or to `root` from the `ZOS` object). A look at the code in Figure 3 shows that the `ZE` object created in `main` flows from `root` to the `ZOS` object through method `putNextEntry`. Thus, clearly, the `ZOS` object does not own the `ZE` object that it accesses.

Finally, lines 5-8 in **computeBoundary** grow the boundary graph as follows: if for some k, the closure set indexed by k is not forbidden, and the corresponding parent set is empty, the closure set is added to $Bndry(h_i)$. The argument that the lemma from Section *6.1.1* holds is made by induction on the number of edges in $Bndry(h_i)$. $Bndry(h_i)$ grows from zero to one edge, $h_i \rightarrow h_j$, when (1) there is no h_k that has handles to both h_i and h_j and (2) there is no h_k such that h_i has a handle to h_k, and h_k has a handle to h_j. The first condition is guaranteed by the check that the *Closure* set of $h_i \rightarrow h_j$ is not forbidden, and the second condition is guaranteed by the check that the *Prt* set of $h_i \rightarrow h_j$ is empty; both checks are performed at line 5. Thus, an edge $h_i \rightarrow h_j$ is added to the empty boundary of h_i only when it is guaranteed that the h_i object accesses the h_j object exclusively (i.e., no other object has a handle to it). Examples of such edges are `ZIS1`→`CRC1` and `ZOS1`→`CRC2`. Clearly, the lemma holds in this case.

Consider an edge $h \rightarrow h_j$ that is added to $Bndry(h_i)$ at line 6. Let o_i be any run-time object represented by h_i and let o be an object represented by h which is dominated by o_i. We need to examine all o_k such that some o_j referred by o may flow to or from o_k (i.e., there is an edge $o_k \rightarrow o_j$ in the relaxed object graph). If all these o_k are dominated by o_i then o_j is dominated by o_i.

Object o_j flows from o into some o_k when one of the following conditions is true. First, o_k has handles to both o and o_j. Since o_i dominates o we have that o_i dominates o_k. This case is handled at lines 4-7 in **findClosureSet** and $h_k \rightarrow h_j$ is examined in order to find the representatives of the objects that o_j may flow to from o_k. Second, o_j may flow from o into some o_k such that o has handles to both o_k and o_j. Clearly, in this case we have that $h \rightarrow h_k \in Bndry(h_i)$ because $h \rightarrow h_k$ is in the *Prt* set of $h \rightarrow h_j$; recall that an edge is removed from a *Prt* set only when it is added to the boundary at lines 6-7 in **computeBoundary**. We may assume (by the inductive hypothesis), that the lemma holds for $h \rightarrow h_k \in Bndry(h_i)$—that is, if o_i dominates o then o_i dominates the o_k referred to by o. Thus, we have that o_i dominates o_k. This case is handled at lines 8-11 in **findClosureSet** and edges $h_k \rightarrow h_j$ are examined appropriately.

6.1.3 Example

We illustrate the algorithm on the graph in Figure 10(b). Consider the boundary of `ZIS1`. There is a single closure set that is not forbidden, *Closure*(`ZIS1`, 0)={`ZIS1`→`CRC1`} with corresponding parent set *Prt*(`ZIS1`, 0)={} and edge `ZIS1`→`CRC1` is added to *Bndry*(`ZIS1`) at line 6. Consider the boundary of `ZOS1`. As a result of **findClosureSet** in lines 1-2 there are four closure sets that are not forbidden: *Closure*(`ZOS1`, 0)={`ZOS1`→`CRC2`}, *Closure*(`ZOS1`, 1) ={`ZOS1`→`VIter1`}, *Closure*(`ZOS1`, 2)={`ZOS1`→`Vector1`,`VIter1`→`Vector1`} and *Closure*(`ZOS1`, 3)={`Vector1`→`Object[]`}. Their corresponding parent sets are *Prt*(`ZOS1`, 0)={}, *Prt*(`ZOS1`, 1) ={}, *Prt*(`ZOS1`, 2)={`ZOS1`→`VIter1`}, and *Prt*(`ZOS1`, 3)={}. The algorithm processes the first closure set and adds edge `ZOS1`→`CRC2` to *Bndry*(`ZOS1`). Then it adds the second closure set—that is, edge `ZOS1`→`VIter1` to the boundary and deletes the edge from the third parent set. The third parent set becomes empty and `ZOS1`→`Vector1` and `VIter1`→`Vector1` are added to the boundary. Finally, edge `Vector1`→`Object[]` is added to the boundary. Thus we have the following boundary graphs: *Bndry*(`ZIS1`) = {`ZIS1`→`CRC1`}, *Bndry*(`Vector1`) = {`Vector1`→`Object[]`} and *Bndry*(`ZOS1`) = {`ZOS1`→`CRC2`, `ZOS1`→`Vector1`, `ZOS1`→`VIter1`, `Vector1`→`Object[]`,`VIter1` →`Vector1`}.

6.2 Identifying Composition Relationships

A corollary of the lemma is that whenever we have an edge $h_i \rightarrow h_j \in Bndry(h_i)$ for each edge $o_i \rightarrow o_j$ represented by it we have that o_i owns o_j. If for every edge labeled with f we have $h \xrightarrow{f} h_j \in Bndry(h)$ the analysis identifies one-to-one implementation-level composition or collection ownership. For example, consider edges $\texttt{ZIS1}\xrightarrow{crc}\texttt{CRC1}$ and $\texttt{ZOS1}\xrightarrow{crc}\texttt{CRC2}$ in Figure 10(b) (recall that field labels was omitted for clarity). Since we have that $\texttt{ZIS1}\xrightarrow{crc}\texttt{CRC1} \in Bndry(\texttt{ZIS1})$, the analysis identifies a one-to-one composition relationship between classes `ZipInputStream` and `CRC32` through field *crc*. Similarly, since we have that $\texttt{ZOS1}\xrightarrow{crc}\texttt{CRC2} \in Bndry(\texttt{ZOS1})$, the analysis identifies a one-to-one composition relationship between classes `ZipOutputStream` and `CRC32` through field *crc*. Furthermore, consider edge $\texttt{ZOS1}\xrightarrow{entries}\texttt{Vector1}$. Again, since we have $\texttt{ZOS1} \overset{entries}{\longrightarrow} \texttt{Vector1} \in \mathit{Bndry}(\texttt{ZOS1})$ and `Vector` is a collection class, the analysis identifies an owned collection through field *entries*.

7. ANALYSIS COMPLEXITY

We discuss the complexity of the analysis in terms of sets H, V and $\mathcal{C}$. Let N be the size of the program being analyzed (i.e., *Cls* and the placeholder main)—that is, the number of statements, the number of object allocation sites and the number of reference variables is of order N. To reason about the complexity of the generalized class analysis in Figure 4 we consider a standard set-constraint-based solution procedure [12, 38, 40, 33]. In set-constraint-based analyses the solution is divided into *constraint generation* and *constraint resolution* where complexity is clearly dominated by constraint resolution. For example, when the algorithm in Figure 4 is instantiated into Andersen-style points-to analysis constraint generation processes each statement once and generates constraints of the form $v_r \subseteq v_l$ (i.e., this constraint denotes that the points-to set of r flows to the points-to set of l). Solving for Andersen's analysis requires propagating object names h_i to each v_r which clearly dominates the linear generation. To reason about constraint resolution in terms of sets H and V consider that $O(|H|)$ object names need to be propagated towards $O(|V|)$ reference variables through constraints of the form $h \subseteq v_i$ and $v_i \subseteq v_j$. Therefore, the complexity of propagation for the generalized analysis is $O(|H| * |V|^2)$ because each constraint $h_i \subseteq v_j$ may be discovered though $O(|V|)$ intermediate variables. As a result for 0-CFA we have $O(|C|{*}|R|^2){=}O(N^3)$, for the Andersen-style points-to analysis we have $O(|S|{*}|R|^2){=}O(N^3)$ and for the object-sensitive points-to analysis we have $O(|S| * |S| * (|R| * |S|)^2){=}O(N^6)$ (note that although the number of classes is of order N, in practice it is substantially smaller than the number of allocation sites).

The complexity of the construction of the approximate object graph in Figure 8 is $O(N * |\mathcal{C}| * |H|^2)$: there are $O(N)$ statements, each statement is processed in at most $O(|\mathcal{C}|)$ contexts and for each statement the algorithm performs at most $O(H^2)$ work (due to lines 2 and 4). Thus, for 0-CFA we have $O(N * |C|^2){=}O(N^3)$, for the Andersen-style points-to analysis we have $O(N * |S|^2){=}O(N^3)$ and for the object-sensitive points-to analysis we have $O(N * |S| * (|S| * |S|)^2){=}O(N^6)$.

Finally, consider procedure **computeBoundary** in Figure 11. The code for partitioning the edges in the transitive closure of h_i into closure sets (lines 1-2) examines each edge and for each edge performs at most $O(|H|)$ work: for edge $h \rightarrow h_j$ there may be at most $O(|H|)$ nodes h_k such that $h_k \rightarrow h$ and $h_k \rightarrow h_j$ (examined at lines 4-7 in **findClosureSet**); similarly, there may be at most $O(|H|)$ nodes h_k such that $h \rightarrow h_k$ and $h_k \rightarrow h_j$ (examined at lines 8-11 in **findClosureSet**). Therefore, the complexity of lines 1-2 is $O(|H|^3)$. The while loop that adds edges to the boundary (lines 5-8) examines each edge at most once, and each edge is removed from at most $O(|H|)$ parent sets. Therefore, the complexity of lines 5-8 is $O(|H|^3)$ as well. To conclude, the complexity of the computation of the boundary sets is $O(|H|^4)$ as for each $h \in H$ the analysis performs $O(|H|^3)$ work. Thus, for 0-CFA we have $O(|C|^4) = O(N^4)$, for the Andersen-style points-to analysis we have $O(|S|^4) = O(N^4)$, and for the object-sensitive analysis we have $O((|S| * |S|)^4) = O(N^8)$. Clearly, the complexity of the entire analysis is dominated by the complexity of the ownership boundary computation.

8. EXPERIMENTAL STUDY

We implemented two instances of our framework—in particular, we considered 0-CFA class analysis and Andersen-style points-to analysis, and the object graph construction and ownership inference based on them. For the rest of this section the analysis based on 0-CFA is denoted by *0-CFA* and the analysis based on the Andersen-style points-to analysis is denoted by *And*. The goal of the empirical study is to address two questions. First, how often do our analyses discover implementation-level composition? Second, how *imprecise* the analyses are—that is, how often they miss implementation-level composition?

We performed experiments on the 7 Java components listed in Table 1. The analysis implementation is based on the Soot framework [42]. The components are from the standard library packages `java.text` and `java.util.zip`, also used in [32]. The components are described briefly in the first two columns of Table 1. Each component contains the set of classes in *Cls* (i.e., the classes that provide component functionality plus all other classes that are directly or transitively referenced by the functionality classes). The number of classes in *Cls* is given in column (3), and the number of classes that implement the component functionality is shown in column (4). We considered all reference instance fields in the classes that implement the component functionality; this number is given in column (5).

8.1 Results

We applied *0-CFA* and *And* in order to determine which fields accounted for composition relationships. Column (2) in Table 2 shows how many of the fields from column (5) in Table 1 are identified as one-to-one compositions, and column (3) shows how many are identified as owned collections (i.e., arrays and standard `java.util` collections). On average, the *0-CFA* analysis reported 28% one-to-one compositions and 6% owned collections, and the *And* analysis reported 30% one-to-one compositions and 10% owned collections.

8.2 Analysis Precision

The issue of analysis precision is of crucial importance for software tools. If an analysis is imprecise, it may report that the relationship between two classes is not a composition while in reality it is, or that a collection is not owned while in reality it is owned (i.e., the analysis reports that certain representation may be exposed while in fact it is not). Such information is not useful and may confuse the user and even render the tool unusable. For example, if a user attempts to ensure the consistency between the code and the composition relationships in UML design class diagrams, imprecision will mean that a large chunk of code will have to be examined manually. Since imprecision results in waste of human time, analysis designers must carefully and precisely identify and evaluate any sources of imprecision.

As expected, the *0-CFA* analysis is less precise than *And*.

(1)Component	(2)Functionality	(3)#Functionality classes	(4)#Classes in *Cls*	(5)#Fields
`gzip`	GZIP IO streams	6	199	7
`zip`	ZIP IO streams	6	194	10
`checked`	IO streams with checksums	4	189	2
`collator`	text collation	15	203	24
`date`	date formatting	17	205	20
`number`	number formatting	10	198	3
`boundary`	iteration over boundaries in text	13	199	7

Table 1: Java components.

(1)Component	(2)#One-to-one			(3)#Owned collections		
	0-CFA	*And*	*Perfect*	*0-CFA*	*And*	*Perfect*
`gzip`	4(57%)	4(57%)	4(57%)	0(0%)	0(0%)	0(0%)
`zip`	3(30%)	3(30%)	3(30%)	2(20%)	2(20%)	2(20%)
`checked`	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)
`collator`	**7(29%)**	10(42%)	10(42%)	6(25%)	6(25%)	6(25%)
`date`	**3(15%)**	**3(15%)**	4(20%)	**0(0%)**	5(25%)	5(25%)
`number`	2(67%)	2(67%)	2(67%)	0(0%)	0(0%)	0(0%)
`boundary`	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)	0(0%)
`Average`	28%	30%	31%	6%	10%	10%

Table 2: Implementation-level compositions.

Furthermore, we performed a study that evaluated the absolute precision of *And*. In particular, we examined the fields that were not identified by *And* as compositions or owned collections. We attempted to prove that it was possible to write client code s.t. an object stored in such a field would be exposed (i.e., it would not be owned by its enclosing object in accordance with the ownership model in Section 2.1). In all cases, except one, we were able to prove exposure. Thus, *And* achieves almost perfect precision.

Field `defaultCenturyStart` in component `date` accounted for the one case of imprecision. The object stored in the field comes from a call to a method `getTime` which creates and immediately returns a `Date` object. Although the `Date` object stored in `defaultCenturyStart` does not flow out of its enclosing `SimpleDateFormat` object, other `Date` objects created by `getTime` in `SimpleDateFormat` are being returned (i.e., there are edges in *Ag* to the `SimpleDateFormat` object and the only representative of `Date`). This imprecision may be resolved by using an instance of our framework that employs more precise object naming. In the case of `getTime` it may distinguish the `Date` objects for different call sites of `getTime`—for example, an analysis which distinguishes context by the last enclosing call site would produce precise results; in this case, the target of the `defaultCenturyStart` edge would be a separate `Date` object that does not flow out and the ownership inference algorithm will correctly identify that there is a composition relationship through this field. However, it remains to be seen whether a more precise context-sensitive points-to analysis will result in substantial benefits for the ownership and composition analyses.

8.3 Conclusions

Our results indicate that the ownership model captures conceptual composition relationships appropriately—we encountered many cases when values of private fields were stored in other parts of the object representation. Thus, a model based on exclusive ownership (i.e., a model which requires that an owned object is referenced only by its owner) would not have been sufficient. The results also show that composition relationships occur often. Therefore, the analysis can provide useful information for reverse engineering tools.

In addition, *And* achieves almost perfect precision; *0-CFA*, although less precise than *And*, achieves acceptable precision as well. We believe that framework instantiations in the range of *0-CFA* and *And* are most suitable for composition inference. It is important that precise results can be obtained with practical analysis—the combined running time of the class analysis, object graph construction and composition inference analyses does not exceed 10 seconds on any component (executed on a 900MHz Sun Fire 380R).

Clearly, a threat to the validity of our results is the relatively small code base used in our experiments. Although the components used in this study are representative program fragments, the results need to be confirmed on more components. In the future we plan to investigate the impact of other framework instances, especially context-sensitive ones, on more components.

9. RELATED WORK

Work by Kollmann and Gogolla [21] and more recently by Guéhéneuc and Albin-Amiot [19] presents definitions and identification algorithms for implementation-level association, composition and aggregation relationships. Our work focuses on compositions and differs from [21] and [19] in both the definition of implementation-level composition and in the identification algorithm. The definition of composition in [21] and [19] is based on exclusive ownership. This may not be sufficient to model commonly used patterns such as iterators, decorators, and factories [14], as well as the common situation when instance fields refer to owned objects that are temporarily accessed by other parts of the representation of the owner. Our definition is based on the owners-as-dominators model which does not require exclusive relationship with the owner; as observed by us and other researchers [9, 30], this model captures well the notion of composition in modeling [36].

We present an identification algorithm that may be more appropriate. Guéhéneuc and Albin-Amiot propose the use of dynamic analysis, but point out serious disadvantages. First, dynamic analysis is slow, second, it requires a complete program, and third, the results that are obtained may be incomplete because they are based on particular runs of particular clients of the component. Kollmann and Gogolla use dynamic analysis as well. Our detection algorithm is based on practical static analysis that works on incomplete programs and produces a solution that is valid over all unknown clients of the component.

Work in [20] and [41] addresses the issue of recovering one-to-many associations through containers, since reverse engineering tools typically loose the association between the enclosing class and the class whose instances are stored in the container field (recall the `entries` field of `Vector` type in Figure 3). Identification of composition is not addressed in these papers.

Ownership type systems disallow certain accesses of object representation [27, 9, 8, 2, 6]. These systems require type annotations and typically do not include automatic inference algorithms or empirical investigations. In contrast, we infer ownership automatically and present an empirical study of the effectiveness of our approach; we believe that our analysis can be usefully incorporated in software tools for reverse engineering of class diagrams from Java code. The only type annotation inference analysis that we are aware of is given by Aldrich et al. [2] for the purposes of alias understanding. Similarly to [19], the `owned` annotation is used only when the analysis is able to prove *exclusive* ownership; in the majority of cases it infers alias parameters. Our work focuses on a different problem, composition inference, and infers ownership using a model that captures better the notion of composition in modeling. Grothoff et al. [16] and Clarke et al. [10] present tools for checking of confinement within a package and within a class respectively. They define confinement rules and the tools check if code conforms to these rules. Our work focuses on a different problem, composition inference, and takes a different approach, the use of semantic analysis that is based on points-to analysis. We believe that such analysis may be more appropriate than confinement rules for the purposes of the identification of object ownership and composition; for example, the rules in [16] and [10] do not handle pseudo-generic containers well.

Bruel et al. [7] and Barbier et al. [4] formalize UML interclass relationships by defining sets of characteristics for association, aggregation and composition; they do not address implementation-level relationships and the problem of reverse engineering. In contrast, we consider implementation-level relationships and propose a methodology for their reverse engineering with an empirical investigation.

10. CONCLUSIONS AND FUTURE WORK

We present an analysis that identifies composition relationships in Java components. We define an ownership-based implementation-level composition model and a static analysis framework for inference of composition relationships in incomplete programs. Our experimental study indicates that (i) the ownership-based model captures well the notion of composition in modeling and (ii) implementation-level compositions occur often and almost *all* such compositions can be identified using a relatively simple and inexpensive analysis. Clearly, no definitive conclusions can be drawn from these limited experiments. In the future, we plan to focus on further empirical investigation placing special emphasis on framework instances based on context-sensitive class analyses.

11. ACKNOWLEDGEMENTS

The author is very grateful to Nasko Rountev for providing the component data, and to the ASE'05 reviewers whose valuable comments improved this paper enormously.

12. REFERENCES

[1] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming*, pages 2–26, 1995.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 311–330, 2002.

[3] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.

[4] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J.-M. Bruel. Formalization of the whole-part relationship in the unified modeling language. *IEEE Transaction Software Engineering*, 29(5):459–470, 2003.

[5] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDD's. In *Conference on Programming Language Design and Implementation*, pages 103–114, 2003.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Symposium on Principles of Programming Languages*, pages 213–223, 2003.

[7] J.-M. Bruel, B. Henderson-Sellers, F. Barbier, A. L. Parc, and R. B. France. Improving the UML metamodel to rigorously specify aggregation and composition. In *International Conference on Object-Oriented Information Systems*, pages 5–14, 2001.

[8] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 292–310, 2002.

[9] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–64, 1998.

[10] D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment time confinment checing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 374–387, 2003.

[11] J. Dean, D. Grove, and C. Chambers. Optimizations of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*, pages 77–101, 1995.

[12] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[13] M. Fowler. *UML Distilled Third Edition.* Addison-Wesley, 2004.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification.* Addison-Wesley, 2nd edition, 2000.

[16] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 241–253, 2001.

[17] D. Grove and C. Chambers. Call graph construction in object-oriented languages. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[18] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 108–124, 1997.

[19] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 301–314, 2004.

[20] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. *IEEE Transaction Software Engineering*, 27(2):156–169, 2001.

[21] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *Working Conference on Reverse Engineering*, pages 81–91, 2001.

[22] C. Larman. *Applying UML and Patterns.* Prentice Hall, 2nd edition, 2002.

[23] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, pages 153–169, 2003.

[24] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 73–79, 2001.

[25] A. Milanova, A. Rountev, and B. Ryder. Parameterized object-sensitivity for points-to and side-effect analyses for Java. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2002.

[26] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–42, 2005.

[27] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *European Conference on Object-Oriented Programming*, pages 158–185, 1998.

[28] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, 1991.

[29] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–340, 1994.

[30] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.

[31] A. Rountev. *Dataflow Analysis of Software Fragments.* PhD thesis, Rutgers University, 2002.

[32] A. Rountev. Precise identification of side-effect free methods. In *International Conference on Software Maintenance*, pages 82–91, 2004.

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

[34] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java softwhare. *IEEE Transaction Software Engineering*, 30(6):372–386, June 2004.

[35] E. Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, pages 208–218, 2000.

[36] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2nd edition, 2004.

[37] M. Streckenbach and G. Snelting. Points-to for Java: A general framework and an emprirical comparison. Technical report, U. Passau, Sept. 2000.

[38] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[39] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon, and C. Godin. Practical virtual method call resolution for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280, 2000.

[40] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 281–293, 2000.

[41] P. Tonella and A. Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In *International Conference on Software Maintenance*, pages 376–385, 2001.

[42] 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.

[43] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Static Analysis Symposium*, pages 180–195, 2002.

[44] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Conference on Programming Language Design and Implementation*, pages 131–144, 2004.