Precise Identification of Composition Relationships 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 uses static ownership inference based on points-to analysis and is designed to work on incomplete programs. In our experiments, on average 40% of the examined fields account for relationships that are identified as compositions. We also present a precision evaluation which shows that for our code base our 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

ASE'05, November 7–11, 2005, Long Beach, California, USA.

1. INTRODUCTION

In modern software development design recovery through reverse engineering is performed often; in a typical iterative development process reverse engineering is performed at the beginning of every iteration to recover the design from the previous iteration [15].

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 [15] (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 [12] 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" [8, Chapter 5] (i.e., it has no well-defined semantics to distinguishes 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" [23, 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 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 [6, 18]; 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. The owners-as-dominators model defines an ownership boundary for each 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. [6, 18] 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 based on *points-to analysis*, which determines the set of objects a reference variable or a reference object field may point to. We use the points-to analysis solution to approximate the possible accesses between run-time objects.

We present empirical results on several components. In our experiments, on average 40% of the examined fields account for relationships that are identified as compositions. We also present a precision evaluation which shows that for our code base, the analysis achieves almost perfect precision—that is, it almost never misses composition relationships identified in our model. 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.

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 static analysis for identifying composition relationships in accordance with our model; the analysis works on incomplete programs.
- We present an empirical study that evaluates our analysis on several Java components.

[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.

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 through a field f is refined as composition if it can be proven that all objects referred by f are owned by their enclosing object. 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* [6, 5, 18]. It is essentially the model proposed by Potter et al. [18] with several modifications that allow more precise handling of popular object-oriented patterns such as iterators, composites and factories [9]. In this model, each execution is represented by an *object graph* which shows access relationships between run-time objects:

- 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 iff field f in o refers to o' at some point of program execution.[2]
- There is an edge $o \xrightarrow{[]} o'$ iff some element of array o refers to o' at some point of program execution.
- There is an edge $o \rightarrow o'$ iff 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'.

[2]We require that all newly created objects appear in the object graph explicitly [6]. That is, at the point of creation a new object is stored in a new local variable; this does not change program semantics.

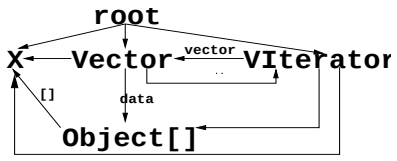
```
public class Vector {
 protected Object[] data;
 public Vector(int size) {
1 data = new Object[size]; }
 public void addElement(Object e,int at) {
2 data[at] = e; }
 public Object elementAt(int at) {
3 return data[at]; }
 public Enumeration elements() {
4 return new VIterator(this); }
}
final class 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.

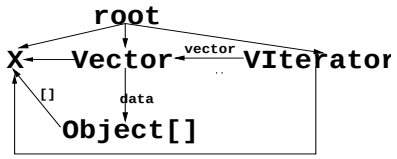
A run-time object o' is accessed in the *context* of o iff there is an edge from o to o' in the object graph. The start of program execution is expressed with a special node `root`. Context `root` represents the context for `main` and for objects referenced by static fields. For example, executing `main` 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.

The owners-as-dominators model states that the owner of an object o is the immediate dominator of o in the object graph [18].[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 in one

[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.



(a) Original Object Graph



(b) Relaxed Object Graph

Figure 2: Object graphs for Figure 1.

context and immediately passed to another context 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 context (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, 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 [10]). 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 C, where C 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 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` and may be accessed only within its ownership boundary.

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 [22, 20]. 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).

[4]Note that in Java, a unique non-trivial (i.e., non-`Object`) common supertype may not exist. A detailed discussion appears in [16].

```
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`.

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 5 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. POINTS-TO ANALYSIS

Points-to analysis determines the set of objects that a given reference variable or a reference field may point to. This information has a wide variety of uses in software tools and optimizing compilers. In this paper, points-to information is used for ownership inference. It 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*. There is a large body of work on points-to analysis with different trade-offs between cost and precision. In this paper, we consider ownership inference based on the Andersen-style flow- and context-insensitive points-to analysis for Java from [21].[5]

3.1 Points-to Analysis for Java

The points-to analysis is defined in terms of three sets. Set R is the set of locals, formals and static fields of reference type. Set O is the set of object names; the objects created at an allocation site s_i are represented by object name $o_i \in O$. Set F contains all instance fields in program classes. The analysis solution is a *points-to graph* where the edges represent the following "may-refer-to" relationships:

- Let $r \in R$ and $o \in O$. An edge $r \rightarrow o$ in the points-to graph means that at run time r may refer to some object that is represented by o.
- Let $f \in F$ be a reference instance field in objects represented by some $o \in O$. An edge $o \xrightarrow{f} o_2$ means that at run time field f of some object represented by o may refer to some object represented by o_2.
- If o represents array objects, $o \xrightarrow{[]} o_2$ shows that some element of some array represented by o may refer at run time to an object represented by o_2.

The Andersen-style points-to analysis for Java from [21] is a relatively precise flow- and context-insensitive inclusion-based analysis. It propagates may-refer-to relationships by analyzing program statements. For example, when it analyzes statement "$p = q$" it infers that p may refer to any object that q may refer to.

[5]Flow-insensitive analyses do not take into account the flow of control between program points and are less precise and less expensive than flow-sensitive analyses. Context-sensitive analyses distinguish between different calling contexts of a method and are more precise and more expensive than context-insensitive ones.

```
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 4: Placeholder `main` method for `zip`.

3.2 Fragment Points-to Analysis

Points-to analyses and Andersen's analysis in particular 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 points-to analysis of 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 [19, 22, 20]. 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 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 points-to and class analyses; for the purposes of this paper we only consider fragment analysis used with the Andersen-style points-to analysis from [21].

The placeholder `main` method for the classes from Figure 3 is shown in Figure 4. 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 [22]. For the purposes of our analysis we discuss the *object reachability* [20] 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 r and leads to some object name o 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

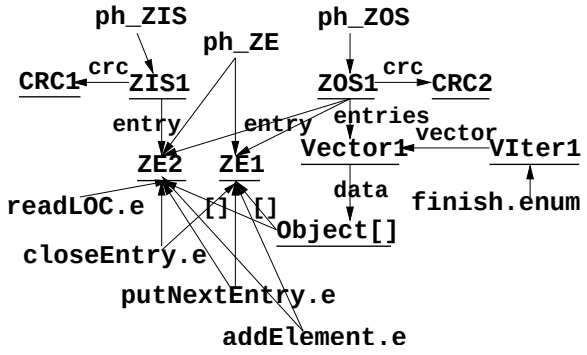


Figure 5: Points-to graph computed by the fragment points-to analysis.

the variable from `main` that has the same type as r. This property is relevant for the ownership and composition analysis described in Section 4 as the points-to graph is used to approximate all possible object graphs and thus all possible accesses must be taken into account.

We illustrate this property for our points-to analysis. Consider the example from Figures 3 and 4. There are three allocation sites in the `main` method; they are denoted by names `ZE1`, `ZIS1` and `ZOS1`. Name `byte[]` corresponds to the allocation site in class `InflaterInputStream`. 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 points-to graph computed by Andersen's analysis from the code in Figures 4, 3 and 1 is shown in Figure 5. Heap object names are underlined and reference variable names are prefixed by the name of their declaring method. For simplicity, implicit parameters `this` and object names `Inflater1, byte[], Hashtable1` and `Deflater1` are not shown.

4. IDENTIFYING COMPOSITION RELATIONSHIPS

We propose a novel analysis for ownership inference that is based on the output of the fragment points-to analysis. The ownership analysis constructs 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*. The analysis uses Ag to identify a *boundary* subgraph rooted at o for each object name o; the subgraph contains paths that are guaranteed to represent flow within the ownership boundary of o. Whenever the edge appears in the boundary of its source for *all* edges labeled with f, the relationship through f is identified as composition.

4.1 Approximate Object Graph

The nodes in Ag are taken from the set of object names O and the edges represent "may-access" relationships. Figure 6 outlines the construction of Ag given a points-to graph Pt. Set $\mathcal{C}_m$ denotes the set of object names that represent the contexts of invocation of method m. If m is an instance method or constructor, $\mathcal{C}_m$ is the points-to set of the implicit

input $Stmt$: set of statements Pt: $R \cup O \rightarrow \mathcal{P}(O)$
output $Ag : O \rightarrow \mathcal{P}(O)$
[1] **foreach**
$s: l = new\ C(...)$ s.t. l not immediately passed or immediately returned to another context,
$s: l = r.m(...)$ s.t. $r \neq$ `this`,
$s: l = r.f$ s.t. $r \neq$ `this` and l assigned to a variable
[2] add $\{c \rightarrow o_j \mid c \in \mathcal{C}_{EnclMethod(s)} \wedge o_j \in Pt(l)\}$ to Ag
// add access edges due to flow from callees to callers
[3] **foreach**
$s: l = new\ C(r)$,
$s: l.m(r)$ s.t. $l \neq$ `this`,
$s: l.f = r$ s.t. $l \neq$ `this`
[4] add $\{o_i \rightarrow o_j \mid o_i \in Pt(l) \wedge o_j \in Pt(r)\}$ to Ag
// add access edges due to flow from callers into callees
[5] **foreach** $o_i \xrightarrow{f} o_j \in Pt$ label with f each $o_i \rightarrow o_j \in Ag$

Figure 6: Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty.

parameter `this` of m. If m is a static method $\mathcal{C}_m$ includes the union of the points-to sets of `this` for all instance methods or constructors that may call m (directly or through a sequence of static calls); it includes `root` if m is `main` or may be called from `main`.

Lines 1-2 account for edges due to flow from the contexts of the callee to the contexts of the caller. For example, at a constructor call new edges are added to Ag from each context enclosing the call to the name representing the newly created object. Similarly, at an instance call not through `this` new edges are added from each context enclosing the call to each returned object. Note that when the newly constructed object is immediately passed to another context (e.g., as in $new\ A(new\ B(...))$), or immediately returned to another context (e.g., as in $return\ new\ VIterator(this)$), no new edges are added to that object from the contexts enclosing the constructor call. Also, at indirect read statements, no edges are added when variable l is not assigned or passed as an explicit or implicit argument later (e.g., it is used only to access instance or array fields such as in x=$l[i]$). This is consistent with the definition of the relaxed object graph in Section 2.1. Lines 3-4 account for edges due to flow from the contexts of the caller to the contexts of the callee. For example, at instance calls edges are added to each object in the points-to set of a reference argument, from each object in the points-to set of the receiver. Finally, line 5 labels the edges with the appropriate field identifier. For brevity, we omit 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 4.2.

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 a context (i.e., `root` or a heap object) and at some point during execution c is the start of a chain in the relaxed object graph that leads to some heap object o^r. In Ag, there will be a chain of edges that starts at the representative of c and leads to the representative of o^r. Figure 7 shows the approximate object graph computed from the code on Figures 3, 4 and 1, and the points-to graph in Figure 5 (only

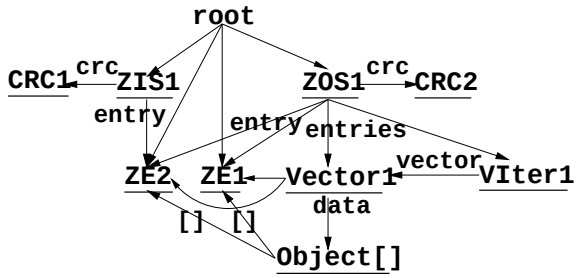


Figure 7: Approximate object graph computed by the algorithm in Figure 6.

object names from Figure 5 are shown). For the majority of edges inference is straight-forward. For example, edges `root`→`ZIS1`, `root`→`ZIS2` and `root`→`ZE1` are due to the constructor calls in `main` and edges `ZIS1`→`CRC1` and `ZIS1`→`ZE2` are due to the constructor calls in class `ZipInputStream`. Edge `ZOS1`→`VIter1` is due to call *enum=entries.elements*() in method `finish`. Edge `VIter1`→`Vector1` is due to statement *return new VIterator*(*this*) in method `elements`; note that there is no edge `Vector1`→`VIter1` due to this statement. Edge `root`→`ZE2` is due to statement $ph\_ZE = ph\_ZIS.getNextEntry()$ in `main`, and edges `ZOS1`→`ZE2` and `ZOS1`→`ZE1` are due to statement $ph\_ZOS.putNextEntry(ph\_ZE)$ in `main`. Edges `Object[]`→`ZE2` and `Object[]`→`ZE1` are due to flow at statement $data[at] = e$ in `addElement`.

4.2 Ownership Boundary

Procedure **computeBoundary** in Figure 8 takes Ag and object name o_i as input and outputs subgraph $Bndry(o_i)$. Subgraph $Bndry(o_i)$ contains paths that are guaranteed to represent flow within the ownership boundary of an instance represented by o_i. More precisely, we have the following lemma. Let o_i^r be a heap object represented by o_i. For every edge e: $o \rightarrow o_j \in Bndry(o_i)$ we have that if o_i^r dominates some o^r then o_i^r dominates the o_j^r that o^r refers to. Therefore, for every o_i^r and run-time path p: $o_i^r \rightarrow ...o^r \rightarrow o_j^r$, whose representative is in $Bndry(o_i)$, we have that o_i^r dominates o_j^r. For example, 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 $\texttt{ZOS1}^r \rightarrow \texttt{Vector1}^r$, $\texttt{ZOS1}^r$ dominates $\texttt{Vector1}^r$.

Below we briefly outline the algorithm and the correctness argument. The algorithm uses the fact that o_j^r flows from object o_i^r to some object o_k^r only if one of the following is true: (1) o_k^r has a handle to both o_i^r and o_j^r (and due to the reachability property Ag contains edges $o_k \rightarrow o_i$, $o_k \rightarrow o_j$, $o_i \rightarrow o_j$), or (2) o_i^r has a handle to both o_k^r and o_j^r (and Ag contains edges $o_i \rightarrow o_k$, $o_i \rightarrow o_j$, $o_k \rightarrow o_j$). This observation helps identify encapsulation more precisely. Suppose that our running example has another input stream object, created by `root` and denoted by name `ZIS2`. The relationship between `ZIS2` and its crc object would be represented by edge `ZIS2`→`CRC1` in Figure 7. 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 `CRC1` object is created and dominated by its enclosing `ZIS1` object because there is no o_k

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

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

Figure 8: Ownership analysis.

such that either o_k has handles to both `ZIS1` and `CRC1`, or `ZIS1` has handles to both o_k and `CRC1`; thus, the `CRC1` object created by the `ZIS1` object does not flow to or from any other context.

The algorithm builds the boundary of an object name o_i by adding edges. First, **computeBoundary** partitions the edges reachable from o_i into appropriate closure sets using auxiliary procedure **findClosureSet**. Intuitively, the closure set of edge $o \rightarrow o_j$ contains all edges $o_k \rightarrow o_j$ in the transitive closure of o_i, such that some o_k^r and o^r refer to the same o_j^r. For example, 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`}. The role of the parent set Prt (discussed later) is to ensure that the relevant paths to $o \rightarrow o_j$ stay in boundary. $Bndry(o_i)$ grows from zero to one edge, $o_i \rightarrow o_j$, when (i) there is no o_k that has handles to both o_i and o_j and (ii) there is no o_k such that o_i has handles to both o_k and o_j, and o_k has a handle to o_j. The first condition is guaranteed by the check that the *Closure* set of $o_i \rightarrow o_j$ is not forbidden, and the second condition is guaranteed by the check that the Prt set of $o_i \rightarrow o_j$ is empty; both checks are performed at line 5. Thus, an edge $o_i \rightarrow o_j$ is added to the empty boundary of o_i only when it is guaranteed that the o_i object accesses the o_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 $o \rightarrow o_j$ that is added to $Bndry(o_i)$ at line 6. 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 o_i^r be any run-time

object represented by o_i and o^r be an object dominated by o_i^r. We need to examine all o_k such that some o_j^r referred by o^r may flow to or from o_k^r (i.e., there is an edge $o_k^r \rightarrow o_j^r$ in the relaxed object graph). If all these o_k^r are dominated by o_i^r then o_j^r is dominated by o_i^r.

Object o_j^r flows from o^r into some o_k^r when one of the following conditions is true. First, o_k^r has handles to both o^r and o_j^r (e.g., o_j^r may be returned to o_k^r from a method invoked on o^r, or it may be passed as an argument from o_k^r to a method invoked on o^r). Since o_i^r dominates o^r we have that o_i^r dominates o_k^r. This case is examined at lines 4-7 in **findClosureSet** and $o_k \rightarrow o_j$ is added to the worklist; it is examined in a subsequent iteration of the while loop in **findClosureSet** in order to find the representatives of the objects that o_j^r may flow to from o_k^r. In addition, $o_k \rightarrow o_j$ is added to $Closure(o_i, n)$, the closure set of $o \rightarrow o_j$. Second, o_j^r may flow from o^r into some o_k^r such that o^r has handles to both o_k^r and o_j^r. Clearly, in this case we have that $o \rightarrow o_k \in Bndry(o_i)$ because $o \rightarrow o_k$ is in the Prt set of $o \rightarrow o_j$; 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 that the lemma holds for $o \rightarrow o_k \in Bndry(o_i)$—that is, if o_i^r dominates o^r then o_i^r dominates the o_k^r referred to by o^r. Thus, we have that o_i^r dominates o_k^r. This case is examined at lines 8-11 in **findClosureSet** and appropriate $o_k \rightarrow o_j$ are added to the worklist and to the closure set.

We briefly illustrate the algorithm on our running example. 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`}.

A corollary of the lemma is that whenever we have an edge $o_i \rightarrow o_j \in Bndry(o_i)$ each o_i^r owns the o_j^r instances that it may refer to. If for every edge labeled with f we have $o \xrightarrow{f} o' \in Bndry(o)$ the analysis identifies one-to-one implementation-level composition or collection ownership. Due to space constraints inference of one-to-many relationships is not discussed here; it is addressed in detail in [16].

4.3 Analysis Complexity

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 names and the number of reference variables is of order N. The complexity of the Andersen-like points-to analysis for Java from [21] is $O(N^3)$. The complexity of the construction of the approximate object graph in Figure 6 is $O(N^3)$ as well; there are $O(N)$ statements and for each statement the algorithm performs at most $O(N^2)$ work (due to lines 2 and 4). Consider procedure **computeBoundary** in Figure 8. The code for partitioning the edges in the transitive closure of o_i into closure sets (lines 1-2) examines each edge and for each edge performs at most $O(N)$ work: for edge $o \rightarrow o_j$ there may be at most $O(N)$ nodes o_k such that $o_k \rightarrow o$ and $o_k \rightarrow o_j$ (examined at lines 4-7 in **findClosureSet**); similarly, there may be at most $O(N)$ nodes o_k such that $o \rightarrow o_k$ and $o_k \rightarrow o_j$ (examined at lines 8-11 in **findClosureSet**). Therefore, the complexity of lines 1-2 is $O(N^3)$. 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 $O(N)$ parent sets. Therefore, the complexity of lines 5-8 is $O(N^3)$ as well. To conclude, the complexity of our analysis is dominated by the computation of the boundary sets which is worst-case $O(N^4)$.

5. EXPERIMENTAL STUDY

The goal of the study is to address two questions. First, how often does our analysis discover implementation-level composition? Second, how *imprecise* the analysis is—that is, how often it misses implementation-level composition?

We performed experiments on the 7 Java components listed in Table 1. The analysis implementation is based on the Soot framework [25]. The components are from the standard library packages `java.text` and `java.util.zip`, also used in [20]. 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). The number of classes in *Cls* and the number of classes that implement the component functionality is shown in column (3). We considered all reference instance fields in the classes that implement the component functionality; this number is given in column (4).

5.1 Results

We applied the algorithm described earlier in order to determine which fields accounted for composition relationships. Column (5) in Table 1 shows how many of the fields from column (4) 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 analysis reported 30% one-to-one compositions and 10% owned collections—that is, 40% of the reference instance fields account for representation that is not being exposed outside of its enclosing object.

5.2 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

(1)Component	(2)Functionality	(3)#Classes *Cls*/Functionality	(4)#Fields	Compositions			
				(5)#One-to-one		(6)#Owned collections	
				Analysis	Perfect	Analysis	Perfect
`gzip`	GZIP IO streams	199/6	7	4(57%)	4(57%)	0(0%)	0(0%)
`zip`	ZIP IO streams	194/6	10	3(30%)	3(30%)	2(20%)	2(20%)
`checked`	IO streams with checksums	189/4	2	0(0%)	0(0%)	0(0%)	0(0%)
`collator`	text collation	203/15	24	10(42%)	10(42%)	6(25%)	6(25%)
`date`	date formatting	205/17	20	3(15%)	**4(20%)**	5(25%)	5(25%)
`number`	number formatting	198/10	3	2(67%)	2(67%)	0(0%)	0(0%)
`boundary`	iter. over boundaries in text	199/13	7	0(0%)	0(0%)	0(0%)	0(0%)
`Average`				30%	31%	10%	10%

Table 1: Java components and implementation-level compositions.

waste of human time, analysis designers must carefully and precisely identify and evaluate any sources of imprecision.

In our experiments, we examined the fields that were not identified 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 (the case is described in detail in [16]). Thus, on our code base, the analysis achieves almost perfect precision.

5.3 Conclusions

Our results indicate that the ownership model captures conceptual composition relationships appropriately—we encountered several 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. It is important that precise results can be obtained with practical analysis—the combined running time of the points-to and composition inference analyses does not exceed 10 seconds on any component (executed on a 900MHz Sun Fire 380R). Of course, a threat to the validity of our results is the relatively small code base used in the experiments; the results need to be confirmed on more components.

6. RELATED WORK

Work by Kollmann and Gogolla [14] and more recently by Guéhéneuc and Albin-Amiot [12] presents definitions and identification algorithms for implementation-level association, composition and aggregation relationships. Our work focuses on compositions and differs from [14] and [12] in both the definition of implementation-level composition and in the identification algorithm. The definition of composition in [14] and [12] is based on exclusive ownership. This may not be sufficient to model commonly used patterns such as iterators, decorators, and factories [9], 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 [6, 18], this model captures well the notion of composition in modeling [23].

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 [13] and [24] 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. Although our work focuses on identification of compositions, our methodology identifies one-to-many associations as well as described in [16].

Ownership type systems disallow certain accesses of object representation [17, 6, 5, 1, 3]. 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. [1] for the purposes of alias understanding. Similarly to [12], 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. [11] and Clarke et al. [7] 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 [11] and [7] do not handle pseudo-generic containers well.

Bruel et al. [4] and Barbier et al. [2] formalize UML interclass relationships by defining sets of characteristics for asso-

ciation, 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.

7. 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 that infers 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. Clearly, no definitive conclusions can be drawn from these limited experiments. In the future, we plan to focus on further empirical investigation.

8. 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.

9. REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

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

[14] 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.

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

[16] A. Milanova. Precise identification of composition relationships for UML class diagrams. Technical Report RPI/DCS-05-10, Rensselaer Polytechnic Institute, 2005.

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

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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 *CC*, LNCS 1781, pages 18–34, 2000.