Practical Static Ownership Inference

Ana Milanova and Yin Liu

Rensselaer Polytechnic Institute, Troy NY 12110, USA,
milanova@cs.rpi.edu, liuy@cs.rpi.edu

Abstract. There are many proposals for ownership type systems designed to control aliasing in object-oriented programs. Most systems require significant annotation effort and therefore it may be difficult to adopt these systems in software practice.
Ownership inference has received less attention, while it is an important problem because it could ease the annotation effort and facilitate application of ownership type systems in real-world software systems.
This paper presents novel static analyses for Java that infers ownership according to two known ownership protocols: the *owner-as-dominator* protocol, and the *owner-as-modifier* protocol. Our analyses do not require annotations. They are based on the cubic Andersen-style points-to analysis, and therefore, remains relatively inexpensive.
We perform experiments on a set of Java programs. The experiments show that the analyses are practical and relatively precise. In addition, the experiments show that ownership occurs frequently in real-world applications, and that the owner-as-dominator protocol and the owner-as-modifier protocol capture distinct ownership properties.

1 Introduction

It is widely acknowledged that reasoning about ownership in object-oriented languages has important benefits for software development; it can help control aliasing and prevent certain unexpected object accesses, which could improve software quality and software security. Consequently, there are many proposals for ownership type systems designed to control aliasing. Most of these systems require significant annotation effort and therefore, it may be difficult to adopt these systems in practice.

Ownership inference, the problem of recovering the ownership structure of an object-oriented program, has received less attention. We believe that ownership inference is an important problem for several reasons.

First, ownership inference could ease the annotation effort and facilitate the application of ownership type systems in real world software systems. It could help bridge the gap between ownership type theory and software practice.

Second, ownership inference could enable the study of the occurrence of ownership in real-world software systems. Although there are many ownership protocols proposed in literature, experience with them is limited. Practical ownership inference could facilitate the comparative study of ownership protocols and help design new appropriate protocols.

Last, but not least, ownership has implication to a relevant and pressing problem: understanding and verification of concurrent software systems. More concretely, ownership guarantees that appropriate synchronization on a shared object o protects o as well as all objects encapsulated in o; conversely, lack of appropriate synchronization on o may expose concurrency errors such as data races on o as well as on objects encapsulated (sometimes deeply) in o. We believe that ownership inference may lead to better understanding of concurrent software systems, and better algorithms for detection of concurrency errors.

Therefore, we believe that it is important to study ownership inference and to develop practical and precise ownership inference techniques.

This paper presents novel static analyses for Java that infer ownership according to two well-known protocols: *owner-as-dominator* and *owner-as-modifier*. Our analyses work directly on Java programs and *do not require annotations* by the programmer. They are based on the cubic Andersen-style points-to analysis, and therefore remain relatively inexpensive.

We implemented the analyses and performed an empirical study on a set of Java benchmarks. The study shows that ownership occurs frequently in Java programs, and that the owner-as-dominator and the owner-as-modifier protocols capture distinct ownership properties. In addition, the study shows that the analyses are relatively practical and adequately precise.

This work has the following contributions:

- We develop a novel static analysis for ownership inference according to the owner-as-dominator protocol.
- We develop a novel static analysis for ownership inference according to the owner-as-modifier protocol.
- We present an empirical study on small to relatively large Java programs.

2 Problem Statement

This paper considers ownership inference according to two known protocols: the *owner-as-dominator* protocol and the *owner-as-modifier* protocol. Owner-as-dominator is exemplified by Clarke et al.'s classical ownership type system [7]. It enforces representation containment: it requires that all accesses to an object must go through its owner (i.e., an object can be accessed *only* by its owner or objects from the same boundary). Owner-as-modifier is exemplified by the Universes ownership type system [10]. Informally, it requires that all modifications to an object must go through its owner (i.e., an object can be modified *only* by its owner or peers from the same boundary).

Throughout the paper, run-time objects are denoted by o with superscript r: e.g., o^r, o_1^r, o_i^r. Abstract objects (i.e., representatives of run-time objects), are denoted using exactly the same notation, but without superscript r: e.g., o, o_1 and o_i are the representatives of o^r, o_1^r and o_i^r respectively. The set of all abstract objects is denoted by O. In our analysis, run-time objects are represented by their allocation sites: for each allocation site s_i, there is an object $o_i \in O$ which represents all run-time objects created at this site. Objects from the code

examples in the paper are denoted using boldface: e.g., $\mathbf{o_A}$ denotes the `A` object created at line 2 in Figure 1. The examples are simple enough and in most cases one abstract object represents exactly one run-time object; whenever this is not the case (i.e., one abstract object represents more than one run-time objects), this is stated explicitly.

Informally, the analysis first infers an *abstract object graph*, which approximates the accesses between run-time objects. The analysis subsequently infers two sets of ownership annotations associated to the edges of the abstract object graph. The first set of annotations are the *dominator annotations*: they induce ownership trees consistent with the owner-as-dominator protocol. The second set of annotations are the *modifier annotations*: they induce ownership trees consistent with the owner-as-modifier protocol.

2.1 Owner-as-dominator Protocol

The analysis infers two dominator annotations: **owned** and **any**. An abstract edge $o_i \rightarrow o_j$ annotated as **owned** states that for every run-time edge $o_i^r \rightarrow o_j^r$ represented by it, o_i^r is the *owner* of o_j^r (i.e., o_i^r is the parent of o_j^r in the ownership tree). An abstract edge annotated as **any** states that o_i^r is not the owner of o_j^r; it does not specify other ownership information.

The correctness requirements imposed by the owner-as-dominator protocol are stated informally below; these requirements are formalized in a theorem in Section 4.

(1) The inferred annotations induce an ownership tree (in other words, in the run-time ownership hierarchy, each object has exactly one owner).
(2) If o_i^r is the owner of o_j^r (i.e., o_i^r is the parent of o_j^r in the ownership tree), then o_i^r *dominates* o_j^r in the run-time object graph (in other words, all accesses to o_j^r go through its owner o_i^r as required by the owner-as-dominator ownership protocol).

From now on, we refer to these annotations as *dominator annotations* and to the ownership tree induced by them as *dominator ownership tree* or just *dominator tree*.

2.2 Owner-as-modifier Protocol

The analysis infers three modifier annotations: **owned**, **peer** and **any**. Analogously to dominator annotations, an abstract edge $o_i \rightarrow o_j$ annotated as **owned**, states that for every run-time edge $o_i^r \rightarrow o_j^r$ represented by it, o_i^r is the *owner* of o_j^r (i.e., o_i^r is the parent of o_j^r in the ownership tree). An abstract edge $o_i \rightarrow o_j$ annotated as **peer**, states that for every run-time edge $o_i^r \rightarrow o_j^r$ represented by it, o_i^r and o_j^r are peers — that is, they have the same owner (the same parent in the ownership tree). An abstract edge annotated as **any** does not specify ownership information.

The correctness requirements imposed by the owner-as-modifier protocol are stated informally bellow; again, the requirements are formalized in a theorem later in the paper.

(1) The inferred annotations induce an ownership tree (in other words, each run-time object has exactly one owner).
(2) If object o_i^r *modifies* object o_j^r[1], then one of the following is true: o_i^r is the owner of o_j^r (i.e., o_i^r is the parent of o_j^r in the ownership tree), or o_i^r and o_j^r are peers (i.e., siblings in the ownership tree). In other words, an object o_j^r is modified only by its owner or its peers as required by the owner-as-modifier ownership protocol.

From now on, we refer to these annotations as *modifier annotations* and to the ownership tree induced by them as *modifier ownership tree* or just *modifier tree*.

```
class Demo {
  public static void main(String[] args) {
1   new Demo().testA(args.length > 0);        //oDemo
  }
  public void testA(boolean b) {
2   A a = new A(b);                           //oA
  }
}
class A {
  boolean mod;
  /*@ owned/peer @*/ B b;
  A(boolean m) {
3   mod = m;
4   b = new B(this);                          //oB
  }
  void off() {
5   mod = false;
  }
}
```

```
class B {
  /*@ owned/peer @*/ C c;
  /*@ owned/owned @*/ D d;
  B(A a) {
6   c = new C(a);                             //oC
7   d = new D();                              //oD
  }
}
class C {
  /*@ any/peer @*/ A a;
  C(A na) {
8   a = na
9   if (a.mod) { a.off();}
  }
}
class D {
  int i;
10 D() { i = 0; }
}
```

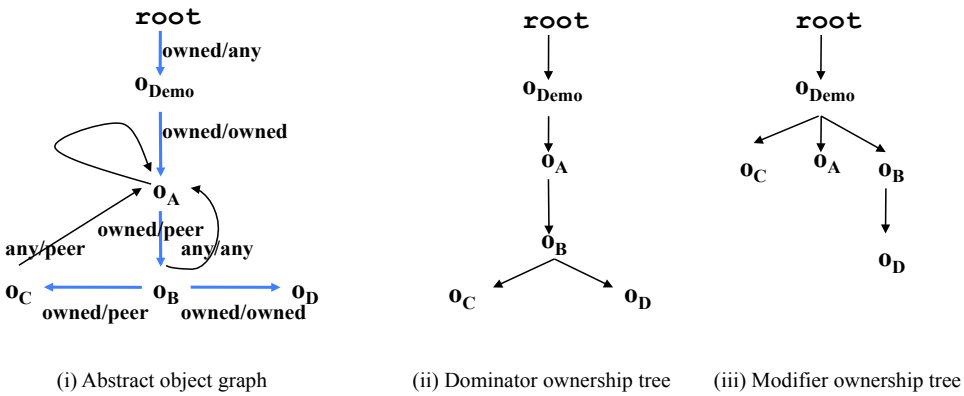
Fig. 1. Example 1.

2.3 Examples

We illustrate the ownership protocols with two examples.

Consider the code in Figure 1 (the example is due to Dietl and Muller [11]). For readability, fields are annotated with the inferred annotations. Figure 2(i) shows the abstract object graph for this code. The edges in the abstract object graph are annotated with the inferred annotations. Each edge (except for $\mathbf{o_A} \rightarrow \mathbf{o_A}$) has two annotations: the first one is the dominator annotation, and the second one is the modifier annotation. Figure 2(ii) shows the dominator ownership tree — that is, the run-time tree induced by the dominator annotations. Figure 2(iii) shows the modifier ownership tree — the run-time tree induced

[1] A precise definition of "object o_i^r modifies object o_j^r" is given in Section **??**.



(i) Abstract object graph (ii) Dominator ownership tree (iii) Modifier ownership tree

Fig. 2. Abstract object graph and ownership trees for Example 1. Blue (thick) edges denote *create* edges.

by the modifier annotations. Note that the nodes in the ownership trees are run-time objects, not abstract objects.

Consider edge $\mathbf{o_A} \rightarrow \mathbf{o_B}$ in the abstract object graph which represents the access from $\mathbf{o_A}$ to $\mathbf{o_B}$ through field `B b` in class `A`. Its dominator annotation is **owned** because clearly, $\mathbf{o_B}$ is dominated by $\mathbf{o_A}$ (i.e., all accesses to $\mathbf{o_B}$ go through $\mathbf{o_A}$). Therefore $\mathbf{o_A}$ is the owner of $\mathbf{o_B}$, and in the dominator tree, $\mathbf{o_A}$ is the parent of $\mathbf{o_B}$. Its modifier annotation is not **owned**, but **peer** — that is, $\mathbf{o_A}$ is not the owner of $\mathbf{o_B}$ but its peer. This is because $\mathbf{o_B}$ can cause an update to field `mod` of $\mathbf{o_A}$ (indirectly, through $\mathbf{o_C}$); the modification of $\mathbf{o_A}$ by $\mathbf{o_C}$ forces $\mathbf{o_A}$, $\mathbf{o_C}$ and $\mathbf{o_B}$ to be peers. Therefore, in the modifier tree $\mathbf{o_A}$, $\mathbf{o_C}$ and $\mathbf{o_B}$ are siblings, children of $\mathbf{o_{Demo}}$.

Figure 3 presents another example. It shows a simplified container (class `Container`) and its iterator (class `Iterator`). The abstract object graph with the inferred annotations is given in Figure 4(i). Again, Figure 4(ii) shows the corresponding dominator tree and Figure 4(iii) shows the corresponding modifier tree.

Consider edge $\mathbf{o_X} \rightarrow \mathbf{o_{Cx}}$. It has dominator and modifier annotations **owned**. The enclosing `X` object dominates its container, and also, the enclosing object is the only object that can cause a modification to its container. The **owned** dominator annotation causes $\mathbf{o_X}$ to be the parent of $\mathbf{o_{Cx}}$ in the dominator tree. Analogously, the **owned** modifier annotation causes $\mathbf{o_X}$ to be the parent of $\mathbf{o_{Cx}}$ in the modifier tree.

Consider edge $\mathbf{o_{Cx}} \rightarrow \mathbf{o_{d[]}}$. It has dominator annotation **any** — this is because the array of `X`'s container can be accessed through its iterator and therefore it is not dominated by its creating container. It has modifier annotation **owned** though — this is because the creating container is the only object that can cause a modification to the array; the iterator accesses the data array in a read-only manner.

```
class Main {
  public static void main(String[] args) {
1   X x = new X();                          //o_X
2   x.mx();
3   Y y = new Y();                          //o_Y
4   y.my();
  }
}
class X {
  /*@ owned/owned @*/ Container cx;
  void mx(Z zx) {
5   cx = new Container(10);                 //o_Cx

6   cx.put(0);
7   Iterator itx = cx.getIt();
  }
}
class Y {
  /*@ owned/owned @*/ Container cy;
  void my() {
8   cy = new Container(10);                 //o_Cy
9   cy.put(0);
  }
}
```

```
class Container {
  /*@ any/owned @*/ int[] data;
  Container(int size) {
10  data = new int[size];                   //o_d[]
  }
  void put(int i) {
11  data[i] = 1;
  }
  Iterator getIt() {
12  return new Iterator(this);              //o_I
  }
}

class Iterator {
  /*@ any/any @*/ int[] data;
  Iterator(Container c) {
13  data = c.data;
  }
}
```

Fig. 3. Example 2.

Note objects $\mathbf{o_{d[],Cx}}$ and $\mathbf{o_{d[],Cy}}$ in the ownership trees. The first object is the data array of `X`'s container, and the second one is the data array of `Y`'s container; these two objects are represented by the same abstract object, $\mathbf{o_{d[]}}$. The dominator annotations on $\mathbf{o_X} \rightarrow \mathbf{o_{d[]}}$ and $\mathbf{o_I} \rightarrow \mathbf{o_{d[]}}$ are **any**; this disallows run-time objects $\mathbf{o_X}$ and $\mathbf{o_I}$ from being the owners of run-time object $\mathbf{o_{d[],Cx}}$ and $\mathbf{o_{d[],Cx}}$ remains without an owner. Our analysis handles this case by forcing $\mathbf{o_{d[],Cx}}$ up the dominator tree, as a child of `root`. On the other hand, the modifier annotation on $\mathbf{o_X} \rightarrow \mathbf{o_{d[]}}$ is **owned** which causes $\mathbf{o_{d[],Cx}}$ to be owned by $\mathbf{o_{Cx}}$.

2.4 Discussion

On one hand, the owner-as-modifier protocol "relaxes" the owner-as-dominator protocol by allowing observational exposure. Thus, an exposed object that has **any** dominator annotation, may still have **owned** modifier annotation. This was the case with edge $\mathbf{o_{Cx}} \rightarrow \mathbf{o_{d[]}}$ in Figures 3 and 4: the data array $\mathbf{o_{d[]}}$ is exposed to the iterator which causes a path that does not go through the enclosing container (hence the **any** dominator annotation); however, the iterator access is read-only (hence the **owned** modifier annotation).

On the other hand, the owner-as-modifier protocol is "more strict" than the owner-as-dominator protocol because it disallows modifications to objects that belong to enclosing boundaries. Thus, an object that has **owned** dominator annotation can have non-**onwed** modifier annotation. This was the case with edge $\mathbf{o_A} \rightarrow \mathbf{o_B}$ in Figures 1 and 2. This edge is **owned** according to the owner-

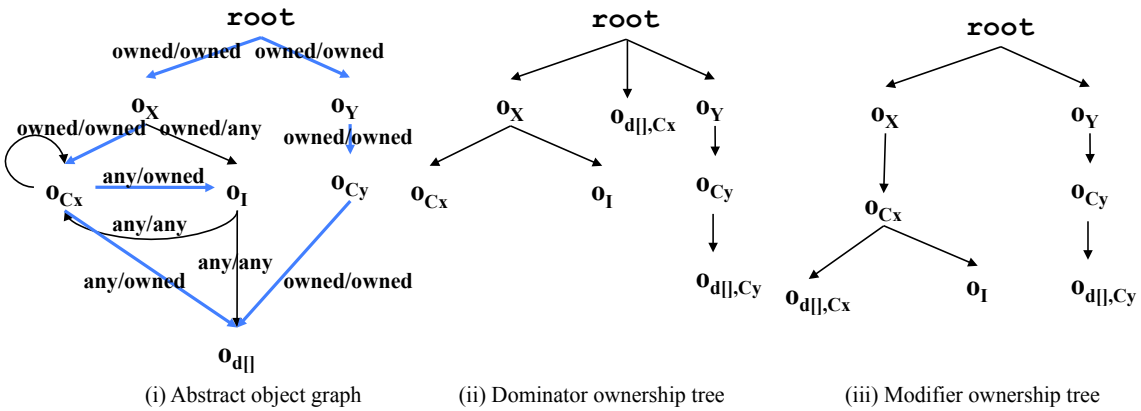


Fig. 4. Abstract object graph and ownership trees for Example 2. Blue (thick) edges denote *create* edges.

as-dominator protocol, because $\mathbf{o_B}$ is dominated by its enclosing object $\mathbf{o_A}$. However, it is non-**owned** (i.e., it is **peer**), according to the owner-as-modifier protocol due to the fact that $\mathbf{o_B}$ triggers an update to field `mod` of $\mathbf{o_A}$ (i.e., causes a modification to $\mathbf{o_A}$), and $\mathbf{o_A}$ belongs to an enclosing boundary.

3 Analyses Needed for Inference of Dominator Annotations

In this section, we describe the analyses that are needed for the inference of dominator annotations. Later, in Section 5, we describe the additional analyses that are needed for the inference of modifier annotations.

Section 3.1 defines the notions of *object graph*, and *dominance boundary*. Section 3.2 briefly describes the underlying points-to analysis. Section 3.3 describes the computation of the object graph, and Section 3.4 describes the computation of the dominance boundary.

To simplify the presentation, which is quite intense, we do not discuss static methods and static fields. They are handled correctly in the analyses and in the implementation; aspects of the handling are described in our previous work [19].

3.1 Notation and Terms

Notation for Objects, Methods and Variables Run-time objects are denoted by o with superscript r: e.g., o^r, o_1^r, o_i^r. Analysis objects (i.e., representatives of run-time objects), are denoted without the superscript r: e.g., o, o_1, o_i; the set of all analysis object is denoted by O. In our analyses, run-time objects are represented by their allocation sites: for each allocation site s_i, there is an analysis object $o_i \in O$ which represents all run-time objects crated at this site. For the rest of the paper we use the following notational convention: run-time

objects are denoted with superscript r and their analysis representatives are denoted using exactly the same o notation but without the superscript; for example, o_1^r's representative is o_1, and o_k^r's representative is o_k. Analysis objects from the concrete code examples are denoted using boldface without a superscript: e.g., $\mathbf{o_A}$ denotes the `A` object created at line 3 in Figure 1.

Methods are denoted by m and n with various subscripts: e.g., m_i, m_1. Methods from our code examples are denoted with their class: e.g., `C.C` denotes the constructor of class `C` in Figure 1, and `A.off` denotes method `off` in class `A`.

Local variables are denoted by lower case letters: e.g., l, r, p, q. Variables from our code example are denoted with a subscript that shows their enclosing method: e.g., $\texttt{this}_{\texttt{C.C}}$ denotes implicit parameter `this` of `C.C` in Figure 1.

Object Graph The notion of the object graph is central to our inference analyses. A *run-time object graph* represents a program execution. The nodes in the run-time object graph are the run-time objects, and the edges represent the access relationships between these run-time objects.

Let P_e be an execution of program P, and let $Og^r_{P_e}$ be the run-time object graph for this execution. $Og^r_{P_e}$ is constructed as follows:

- There is an edge $o^r \rightarrow o_1^r$ in $Og^r_{P_e}$ if at some point of the execution P_e, a field of object o^r refers to object o_1^r.
- There is an edge $o^r \rightarrow o_1^r$ in $Og^r_{P_e}$ if o^r is an array object, and at some point of the execution P_e, o^r has element o_1^r.
- There is an edge $o^r \rightarrow o_1^r$ in $Og^r_{P_e}$ if at some point of the execution P_e, an instance method invoked on receiver object o^r has local variable l, $l \neq \texttt{this}$, that refers to object o_1^r [2].

The `main` method is treated as a special instance method executed on a special receiver object `root` — that is, if `main` has a local variable l that refers to an object o_1^r, then there is an edge $\texttt{root} \rightarrow o_1^r \in Og^r_{P_e}$.

This definition of the run-time object graph is consistent with earlier definitions [7, 30]. Note that the run-time object graph $Og^r_{P_e}$ "accumulates" edges as the program executes and never "deletes" edges; at the end of the execution, $Og^r_{P_e}$ contains all edges that have been active during the program run.

Dominance Boundary Let P_e be an execution of program P, and let $Og^r_{P_e}$ be the run-time object graph for this execution. Let o^r be any object in $Og^r_{P_e}$. The *dominance boundary* of o^r, denoted by $Boundary(o^r)$, is the subgraph of $Og^r_{P_e}$ dominated by o^r — that is, all paths from `root` to the objects in $Boundary(o^r)$ go through o^r.

3.2 Points-to Analysis

Points-to analysis determines the set of objects that a given reference variable or a reference field may point to. This analysis is the foundation of all other

[2] We require that there be an explicit reference variable for each object that is accessed (i.e., a statement $r.m().n()$ is re-written into an equivalent sequence $r_1 = r.m(); r_1.n()$).

analyses in this paper. For the purposes of this paper, we use the well-known Andersen-style flow- and context-insensitive points-to analysis for Java from [33, 18].[3] The Andersen-style analysis is inclusion-based, and it distinguishes objects per allocation sites — each allocation site s_i corresponds to analysis object $o_i \in O$. It has cubic worst case complexity, it is well-understood, and there are several scalable publicly available implementations. The fact that our analyses are based on this relatively inexpensive points-to analysis allows them to remain efficient and practical.

The points-to analysis computes the points-to graph, Pt, of the program. We extend the Pt notation to denote points-to sets. $Pt(l)$ denotes the points-to set of variable l and $Pt(o.f)$ denotes the points-to set of reference field f of object o. We assume that the reader is familiar with this analysis, and do not elaborate on its semantics.

3.3 Object Graph Analysis

This object graph analysis uses the result of the points-to analysis, Pt, and constructs Og, the approximation of all run-time object graphs $Og^r_{P_e}$. If there is an execution P_e with a run-time object graph $Og^r_{P_e}$, such that edge $o^r \rightarrow o^r_1 \in Og^r_{P_e}$, then there is an edge $o \rightarrow o_1 \in Og$ where o is the representative of o^r and o_1 is the representative of o^r_1.

$\langle s_i\text{: } l = new\ C(r_1) \text{ in } m\rangle$	$Og \cup \{o \rightarrow o_i \mid o \in Pt(\texttt{this}_m)\}$	// *create*
	$\cup \{o_i \rightarrow o_j \mid o_j \in Pt(r_1)\}$	// *in*
$\langle l = r.f, r \neq \texttt{this}, l = r[k] \text{ in } m\rangle$	$Og \cup \{o \rightarrow o_i \mid o \in Pt(\texttt{this}_m) \wedge o_i \in Pt(l)\}$	// *out*
$\langle l.f = r, l \neq \texttt{this}, l[k] = r \text{ in } m\rangle$	$Og \cup \{o_i \rightarrow o_j \mid o_i \in Pt(l) \wedge o_j \in Pt(r)\}$	// *in*
$\langle l = r.n(r_1), r \neq \texttt{this} \text{ in } m\rangle$	$Og \cup \{o \rightarrow o_j \mid o \in Pt(\texttt{this}_m) \wedge o_j \in Pt(l)\}$	// *out*
	$\cup \{o_i \rightarrow o_j \mid o_i \in Pt(r) \wedge o_j \in Pt(r_1)\}$	// *in*
$\langle l = r.m(\texttt{this}), ... = \texttt{this}, \text{ in } m\rangle$	$Og \cup \{o_i \rightarrow o_i \mid o_i \in Pt(\texttt{this})\}$	// *self*

foreach $o_i \rightarrow o_j \in Og$ s.t. $o_j \in Pt(o_i.f)$ label the edge with f: $o_i \xrightarrow{f} o_j \in Og$

Fig. 5. Transfer functions for construction of Og.

Figure 5 gives the transfer functions for construction of the object graph Og. The analysis starts with an empty Og and adds edges to Og as it processes program statements. There are four kinds of edges: (i) *create* edges (due to object creation), (ii) *in* edges (due to arguments), (iii) *out* edges (due to return), and (iv) *self* edges (due to leak of `this`); an edge $o_i \rightarrow o_j$ may be of more than one kind. The significance of these kinds of edges will become clear in Section 3.4.

[3] 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.

An object creation statement s_i: $l = new\ C(r_1)$, in method m, results in two kinds of edges. First, there are edges from each receiver of method m to o_i, the object created at that site. These edges are labeled as *create* edges. They capture run-time object creation: when an object is created, this newly created object becomes accessible to the receiver of method m. Second, there are edges from o_i, the newly created object, to each o_j in the points-to set of argument r_1. These edges are labeled as *in* edges. They capture run-time access "due to arguments": when an object is passed as an argument to a constructor, it becomes accessible to the newly crated object.

An instance field read statement $l = r.f$, $r \neq \texttt{this}$, and an array read statement $l = r[k]$ in method m, result in edges from every receiver o_i of m to each o_j in the points-to set of l. These edges are labeled as *out* edges. They capture run-time access "due to return": when an object accessible to the object referred by r, is "returned" at this statement, it becomes accessible to the receiver of m.

An instance field write statement $l.f = r$, $l \neq \texttt{this}$, and an array write statement $l[k] = r$ in method m, result in edges from every object o_i in the points-to set of l to every object o_j in the points-to set of r. These edges are *in* edges. They capture access "due to arguments": when an object referred by r (and accessible to the receiver of m) is assigned to $l.f$, this object becomes accessible to the object that l refers to.

A virtual call $l = r.n(r_1)$, $r \neq \texttt{this}$ in method m results in two kinds of edges. First, there are edges from each receiver of m to each object o_j in the points-to set of l. These edges are labeled as *out* edges. They capture access "due to return": when the object, accessible to the object referred by r, is returned due to this call, it flows to the receiver of m. Second, there are edges from each object o_i in the points-to set of r to each object o_j in the points-to set of r_1. These edges capture flow "due to arguments", and are labeled as *in* edges.

Statements where implicit parameter `this` is leaked (namely, statements $l = r.m(\texttt{this})$, $l.f = \texttt{this}$, and $l = \texttt{this}$), result in *self* edges. These edges account that the `this` object accesses itself, and thus, it may pass a reference to itself to other objects. These edges are needed for the correctness of the dominance boundary analysis presented in Section 3.4.

The final line in Figure 5 examines each edge $o_i \rightarrow o_j$ in the constructed Og, and if o_j is in the points-to set of some field f of o_i (i.e., $o_j \in Pt(o_i.f)$), the edge is determined to be a field edge and is labeled with f: $o_i \xrightarrow{f} o_j$.

Example Consider the code in Figure 3.Line 1, an object creation statement, results in *create* edge $\texttt{root} \rightarrow \mathbf{o_{demo}}$: we have $Pt(\texttt{this}_{\texttt{main}}) = \{\texttt{root}\}$, and the object created at site 1 is $\mathbf{o_{demo}}$. Line 2 does not result in edges because the call has no reference arguments and no reference return. Line 3 results in *create* edge $\mathbf{o_{demo}} \rightarrow \mathbf{o_A}$. Line 5 results in *craete* edge $\mathbf{o_A} \rightarrow \mathbf{o_B}$, *in* edge $\mathbf{o_B} \rightarrow \mathbf{o_A}$, and *self* edge $\mathbf{o_A} \rightarrow \mathbf{o_A}$. Line 7 results in *create* edge $\mathbf{o_B} \rightarrow \mathbf{o_C}$ and *in* edge $\mathbf{o_C} \rightarrow \mathbf{o_A}$. Finally, statement 8 results in *create* edge $\mathbf{o_B} \rightarrow \mathbf{o_C}$. The object graph constructed by the analysis is given in Figure 2(i). The field edges are the following: $\mathbf{o_A} \xrightarrow{\mathbf{b}} \mathbf{o_B}$, $\mathbf{o_B} \xrightarrow{\mathbf{c}} \mathbf{o_C}$, $\mathbf{o_B} \xrightarrow{\mathbf{d}} \mathbf{o_D}$ and $\mathbf{o_C} \xrightarrow{\mathbf{a}} \mathbf{o_A}$.

Note that the analysis ignores statements through `this` (`this`.$f = r$, $l =$ `this`.f and `this`.$n(r_1)$) and direct assignments ($l = r$); this is correct because these statements do not result in new run-time edges. For example, consider `this`.$f = r$ in method m; the run-time access edge between the receiver of m and the object referred by r is already there; it is due to object creation (e.g, $r = new\ C()$), due to a return (e.g., $r = r1.n()$), or it is due to arguments.

Note that a straight-forward (and naive) object graph can be obtained directly from points-to information as follows. First, one can add an edge $o_i \rightarrow o_j$ to Og for each field edge in the points-to graph (i.e., $o_j \in Pt(o_i.f)$). Second, one can consider each variable l in method m, and for each $o_i \in Pt(\texttt{this}_m)$ and each $o_j \in Pt(l)$, add edge $o_i \rightarrow o_j$ to Og. This construction inherits significant imprecision from the context-insensitive Andersen's points-to analysis. For example, consider the following common object-oriented code, which initializes an instance field through an instance method. We assume that classes `Y` and `Z` inherit from superclass `X`.

```
class A {
   X f;
   m(X xa) { this.f = xa; }
  }
A a1 = new A(); // o_a1   a1.m(new Y()); // o_y
A a2 = new A(); // o_a2   a2.m(new Z()); // o_z
```

The context-insensitive points-to analysis merges the contexts of invocation of method `m`, which results in having fields f of each of the `A` objects, $\mathbf{o_{a1}}$ and $\mathbf{o_{a2}}$, point to both $\mathbf{o_y}$ and $\mathbf{o_z}$. Using the naive construction of the object graph results in access edges from $\mathbf{o_{a1}}$ to both $\mathbf{o_y}$ and $\mathbf{o_z}$, and from $\mathbf{o_{a2}}$ to both $\mathbf{o_y}$ and $\mathbf{o_z}$. This is imprecise — $\mathbf{o_{a1}}$ accesses only $\mathbf{o_y}$, and $\mathbf{o_{a2}}$ accesses only $\mathbf{o_z}$.

The object graph analysis in Figure 5 handles this case and other idiomatic cases precisely. It ignores `this.f=xa`, the statement that would have caused imprecision; it processes statements `a1.m(new Y())` and `a2.m(new Z())`; the first statement results in an access edge from $\mathbf{o_{a1}}$ to $\mathbf{o_y}$, and the second statement results in an edge from $\mathbf{o_{a2}}$ to $\mathbf{o_z}$. The analysis achieves good precision "for free" — its worst-case complexity is cubic, the same as the worst-case complexity of Andersen's analysis.

3.4 Dominance Boundary Analysis

In this section, we present the dominance boundary analysis. This analysis is at the heart of the inference of dominator and modifier annotations, and the central contribution of this paper. It uses the object graph Og described above.

The dominance boundary of object $o_i \in O$ is a subgraph of Og rooted at o_i. The analysis that computes the dominance boundary of o_i is presented in Figure 7. It uses Og (as well as other information which will be explained shortly), takes as input o_i, and computes the dominance boundary of o_i, $Boundary(o_i)$. The correctness of this computation is stated by the following lemma.

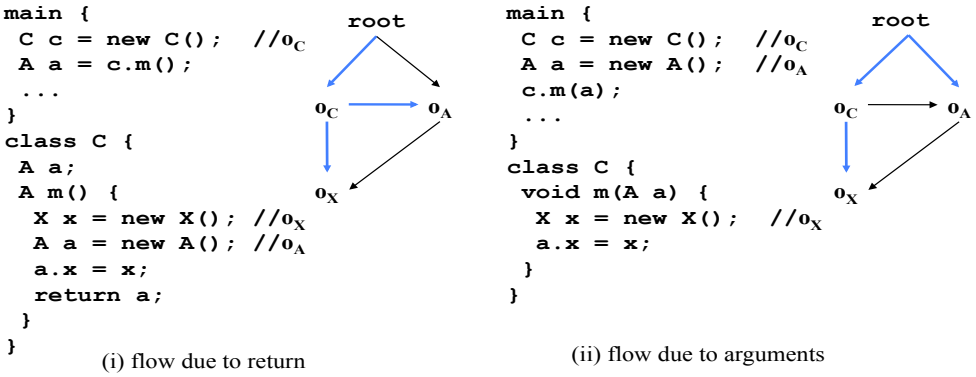


Fig. 6. Object flows. Blue (thick) edges denote *create* edges.

Lemma 1. *Let o_i be any analysis object and let Boundary(o_i) be the dominance boundary of o_i computed by the analysis in Figure 7. Let $Og^r_{P_e}$ be any run-time object graph and let $o^r_i \in Og^r_{P_e}$ be any run-time object represented by o_i.*

For every path p: $o^r_i \rightarrow ... \rightarrow o^r_j \in Og^r_{P_e}$, such that the representative of p is in Boundary(o_i), we have that o^r_i dominates *o^r_j in $Og^r_{P_e}$.*

Informally, the lemma states that the computed static boundary of o_i (under) approximates the run-time dominance boundary of each object o^r_i represented by o_i. The rest of this section describes the intuition behind the analysis. The correctness proof of the lemma is given in Appendix A.

***Create* Reachability** One important observation is that in order for an object o^r_j to be in the boundary of o^r_i, o^r_j *must have been created* by o^r_i, directly or indirectly. Let set *createClosure*(o_i) include o_i and the set of objects reachable on *create* edges from o_i in the object graph Og. In the running example from Figures 1 and 2(i), *createClosure*$(\mathbf{o_{demo}}) = \{\ \mathbf{o_{demo}}, \mathbf{o_A}, \mathbf{o_B}, \mathbf{o_C}, \mathbf{o_D}\}$ — $\mathbf{o_{demo}}$ creates $\mathbf{o_A}$, then $\mathbf{o_A}$ creates $\mathbf{o_B}$, and $\mathbf{o_B}$ creates $\mathbf{o_C}$ and $\mathbf{o_D}$.

createClosure(o_i) is an upper bound on the nodes in *Boundary*(o_i); intuitively, an object o_j in *createClosure*(o_i) stays in the boundary until (roughly) o_j flows to an "outside" object o_k.

Object Flow Another important observation is the flow of objects. Let o^r_i be a run-time object that has access to another object, o^r_j — that is, there is an access edge $o^r_i \rightarrow o^r_j$ in the current run-time object graph. Object o^r_j can flow from o^r_i to another object, o^r_k, in one of two cases: (i) due to a return: o^r_j is returned from o^r_i to o^r_k, or (ii) due to an argument: o_j is passed as an argument from o^r_i to o^r_k. Below, we describe the two cases:

(i) Flow due to return. Recall that o^r_i has access to o^r_j — i.e., there is edge $o^r_i \rightarrow o^r_j$ in the current object graph. Object o^r_j flows from o^r_i to o^r_k due to a return if we have (1) o^r_k has access to o^r_i, i.e., there is edge $o^r_k \rightarrow o^r_i$, and (2) method m invoked on receiver o^r_k executes statement $l = r.n()$ where r points to o^r_i, and l points to o^r_j (i.e., o^r_j is returned from o^r_i to o^r_k due to

statement $l = r.n()$)[4]. *Og* reflects this flow by *edge triple* $o_k \rightarrow o_i$, $o_i \rightarrow o_j$ and $o_k \rightarrow o_j$.

(ii) Flow due to arguments. Again, we have $o_i^r \rightarrow o_j^r$. Object o_j^r flows from o_i^r to o_k^r due to arguments if we have (1) $o_i^r \rightarrow o_k^r$, and (2) method m invoked on receiver o_i^r, executes statement $r.n(r_1)$ where r points to o_k^r and r_1 points to o_j^r (i.e., o_j^r is passed from o_i^r to o_k^r as an argument in statement $r.n(r_1)$). *Og* reflects this flow by edge triple $o_i \rightarrow o_k$, $o_k \rightarrow o_j$ and $o_i \rightarrow o_j$.

Consider Figure 6(i) which illustrates case (i). We have that `root` has access to $\mathbf{o_C}$ (`root` creates $\mathbf{o_C}$) and $\mathbf{o_C}$ has access to $\mathbf{o_A}$ (again, $\mathbf{o_C}$ creates $\mathbf{o_A}$). Subsequently statement `a = c.m()` in `main` returns $\mathbf{o_A}$ to `root` which results in an access edge from `root` to $\mathbf{o_A}$. The flow of $\mathbf{o_C}$ from $\mathbf{o_A}$ to `root` is reflected by edge triple `root`$\rightarrow \mathbf{o_C}$, $\mathbf{o_C} \rightarrow \mathbf{o_A}$, `root`$\rightarrow \mathbf{o_A}$. Now consider Figure 6(ii) which illustrates case (ii). We have that `root` accesses $\mathbf{o_C}$ and $\mathbf{o_A}$ (`root` creates both objects), and statement `c.m(a)` passes $\mathbf{o_A}$ to $\mathbf{o_C}$ which results in an access edge from $\mathbf{o_C}$ to $\mathbf{o_A}$. The flow of $\mathbf{o_a}$ from `root` to $\mathbf{o_C}$ is reflected by edge triple `root` $\rightarrow \mathbf{o_C}$, $\mathbf{o_C} \rightarrow \mathbf{o_A}$, `root`$\rightarrow \mathbf{o_A}$ (note that this triple is identical to the triple in case (i)).

The notion of the edge triple is central to the analysis; the tracking of flow of objects is at the heart of the precise computation of dominance boundary information. From now on, we will interchangeably denote an edge triple as a triple of edges, or as an ordered triple of nodes. An edge triple $o_k \rightarrow o_i$, $o_i \rightarrow o_j$, $o_k \rightarrow o_j$ is denoted as an ordered triple of nodes as follows: o_k, o_i, o_j.

Valid Triple Yet another important observation is that not every edge triple $o_k \rightarrow o_i, o_i \rightarrow o_j, o_k \rightarrow o_j$ represents valid object flow. For example, consider triple $\mathbf{o_c}, \mathbf{o_A}, \mathbf{o_A}$ from the graph in Figure 2(i) (this triple involves self edge $\mathbf{o_A} \rightarrow \mathbf{o_A}$). It is easy to see that this triple does not represent valid flow: there is no flow from $\mathbf{o_a}$ to $\mathbf{o_C}$ due to a return, and there is no flow from $\mathbf{o_C}$ to $\mathbf{o_A}$ due to arguments. The only triple that involves $\mathbf{o_A} \rightarrow \mathbf{o_A}$ and represents valid flow is $\mathbf{o_A}, \mathbf{o_B}, \mathbf{o_A}$: $\mathbf{o_A}$ accesses $\mathbf{o_B}$, $\mathbf{o_A}$ accesses itself through `this`, and $\mathbf{o_A}$ passes itself to $\mathbf{o_B}$ as an argument in `new B(this)`.

The analysis uses predicate $validTriple(o_k, o_i, o_j)$ to filter out invalid triples. Predicate *validTriple* is implemented by recording the statement that causes the creation of an edge; $validTriple(o_k, o_i, o_j)$ examines a triple o_k, o_i, o_j and checks the two cases: (i) if $o_k \rightarrow o_j$ is an *out* edge, and there is a statement $l = r.n()$ associated to this edge such that $o_i \in Pt(r)$, $validTriple(o_k, o_i, o_j)$ returns true; (ii) if $o_i \rightarrow o_j$ is an *in* edge and there is a statement $l.n(r)$ in method m associated to it such that $o_k \in Pt(\texttt{this}_m)$, $validTriple(o_k, o_i, o_j)$ returns true as well; otherwise, $validTriple(o_k, o_i, o_j)$ returns false. The predicate increases the memory needed to store the object graph; however, the impact of *validTriple* on scalability and precision is significant — in fact, without it, the analysis does not scale even to a relatively small program.

The analysis makes use of predicate $isOutside(o_i \rightarrow o_j)$. $isOutside(o_i \rightarrow o_j)$ returns true if there exists o_k such that o_k, o_i, o_j is a valid triple — that is, there

[4] For brevity, we mention statement kind $l = r.n()$ only; the other statements that result in *out* edges, namely $l = r.f$, and $l = r[i]$ can be exectuted as well.

procedure *computeBoundary*
uses *Og*, *createClosure*, *validTriple*, *isOutside*
input o_i
output $Boundary(o_i)$
[1] $Out = \{o_j \mid isOutside(o_i \rightarrow o_j)\}$
[2] $In = createClosure(o_i)$ - Out
[3] $W = \{o_1 \rightarrow o_2 \mid o_1 \in In \wedge o_2 \in Out\}$

[4] while $W \neq \emptyset$
[5] remove $o \rightarrow o_j$ from W, mark it as visited
[6] if o_j is not visited $\wedge$ $o_j \in createClosure(o_i)$
[7] mark o_j as visited
[8] remove $createClosure(o_j)$ from In
[9] add $createClosure(o_j)$ to Out
[10] foreach $o_{k'} \in createClosure(o_j)$
[11] foreach *create* edge $o_{k''} \rightarrow o_{k'}$ s.t. $o_{k''} \in In$
[12] if $o_{k''} \rightarrow o_{k'}$ is not visited, add $o_{k''} \rightarrow o_{k'}$ to W
[13] foreach $validTriple(o, o_j, o_k)$
[14] add o_k to Out;
[15] if $o \rightarrow o_k$ is not visited, add $o \rightarrow o_k$ to W
[16] foreach $validTriple(o, o_{k'}, o_j)$ s.t. $o_{k'} \in In$
[17] if $o_{k'} \rightarrow o_j$ is not visited, add $o_{k'} \rightarrow o_j$ to W
[18] foreach $validTriple(o_{k'}, o, o_j)$ s.t. $o_{k'} \in In$
[19] if $o_{k'} \rightarrow o_j$ is not visited, add $o_{k'} \rightarrow o_j$ to W

[20] $Boundary(o_i) = \{o \rightarrow o_j \in Og \mid o \in In \wedge o_j \in In\}$.

Fig. 7. *computeBoundary* computes the boundary of o_i.

exists some "outside" o_k such that either (i) o_j is returned from o_i to o_k, or (ii) o_j is passed as an argument from o_k to o_i; as a result, there could be a path to o_j through o_k (and not through o_i) in which case o_i may not dominate o_j.

Analysis Description The analysis maintains sets *Out*, *In*, and worklist W. Set *Out* contains the current set of "outside" objects accessible to the boundary. These are objects that either (i) flow to the "outside" *from* the boundary of o_i, or (ii) they flow *to* the boundary of o_i from "outside". Set *Out* is initialized to the set of objects o_j such that $isOutside(o_i \rightarrow o_j)$ is true (line 1). The initial set *Out* captures the objects o_j such that one of the following is true: (i) o_j is directly returned from o_i (e.g., through a statement such as `a = c.m()` in Figure 6(i) which causes edge $\mathbf{o_C} \rightarrow \mathbf{o_A}$ to be an outside edge, and $\mathbf{o_A}$ to be in the initial *Out*), or (ii) o_j is passed from outside into o_i as an argument (e.g., through a statement such as `c.m(a)` in Figure 6(ii) which causes edge $\mathbf{o_C} \rightarrow \mathbf{o_A}$ to be an outside edge and $\mathbf{o_A}$ to be in the initial *Out*). Set *In* contains the current (over) approximation of the dominance boundary; it is initialized to $createClosure(o_i)$ minus the objects returned from o_i, i.e, $createClosure(o_i) - Out$ (line 2). Worklist W contains the set of cut edges — edges between the boundary *In* and the outside objects *Out* (line 3).

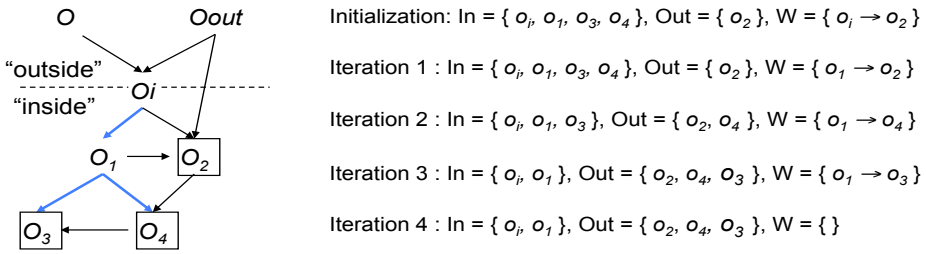


Fig. 8. Boundary computation example. The boxed objects are found to be in *Out*.

The analysis starts with initial sets *In*, *Out* and W and proceeds to identify all objects, originally in *In*, that are reachable from the initial *Out*. For every new edge $o \rightarrow o_j$ ($o \in$ *In* is an "inside" object, and $o_j \in$ *Out* is an "outside" object) taken from the worklist, the analysis does three things.

First, it examines o_j (lines 6-12). If o_j was in *In* (i.e., in $createClosure(o_i)$) and was found to be in *Out*, the analysis removes the entire $createClosure(o_j)$ from *In* and adds it to *Out* (lines 8-9). Clearly, of o_j is reachable from the "outside", then the objects reachable on *create* edges from o_j are also reachable from the "outside". Next, the analysis identifies *create* edges whose source $o_{k''}$ is in *In* and target $o_{k'}$ was just found to be in *Out*, and adds these edges to W(lines 10-12)[5].

Second, the analysis identifies objects o_k, such that o, o_j, o_k is a valid triple — in other words, we may have that o_k flows from o to "outside" o_j, or o_k flows from "outside" o_j to o. The analysis adds o_k to *Out* and $o \rightarrow o_k$ to W (lines 13-15). If o_k was in *In* until this point (i.e., o_k was an "inside" object, until it was passed to "outside" object o_j), when the edge is removed from W, o_k's *createClosure* will be removed from *In* and added to *Out*.

Third, the analysis identifies objects $o_{k'}$ in *In* such that "outside" object o_j flows to $o_{k'}$ (lines 16-19); this may cause an object deeper in the boundary to become reachable from outside, which will be discovered when edge $o_{k'} \rightarrow o_j$ is examined at line 5 in a subsequent iteration of the while loop. The analysis terminates because each object graph edge appears in W at most once.

Examples Consider the object graph in Figure 2(i), and consider the computation of the boundary of $\mathbf{o_A}$. *Out* and W are empty and *In* is initialized to $createClosure(\mathbf{o_A}) = \{\mathbf{o_A}, \mathbf{o_B}, \mathbf{o_C}, \mathbf{o_D}\}$. The while loop at lines 4-19 never executes and $Boundary(\mathbf{o_A})$ consists of nodes $\mathbf{o_A}, \mathbf{o_B}, \mathbf{o_C}, \mathbf{o_D}$ and the edges between them.

As another example, consider Figure 8. "Outside" object o_2 is passed as an argument to o_i. o_i then passes o_2 to "inside" object o_1; o_1 then passes "inside"

[5] We conjecture that these edges are actually unnecessary; however, they are needed by our correctness proof, which requires that all cut edges are seen on W.

object o_4 to o_2 and o_4 becomes "outside"; o_1 passes "inside" object o_3 to o_4 and o_3 becomes "outside". The workings of the analysis are shown in Figure 8.

These examples, which are intentionally simplified, may create the impression that dominance inference could be done by using known dominator algorithms on the object graph. This is not the case, because the object graph is a *static* representation of objects and object accesses — that is, a node in the object graph typically correspond to multiple run-time objects and an edge corresponds to multiple run-time access edges. Using standard dominator algorithms on the object graph would affect both correctness and precision. For example, consider Figure 2(ii). A dominator algorithm will determine that `Y`'s container, $\mathbf{o_{Cy}}$, does not dominate its array, $\mathbf{o_D}$, because of the multiple paths from `root` to $\mathbf{o_{d[]}}$ that do not go through $\mathbf{o_{Cy}}$; in contrast, our analysis determines that $\mathit{Boundary}(\mathbf{o_{Cy}})$ equals $\{\mathbf{o_{Cy}} \rightarrow \mathbf{o_{d[]}}\}$. Therefore, we have that $\mathbf{o_{Cy}}$ dominates its array $\mathbf{o_{d[]}}$.

4 Inference of Dominator Annotations

Function dom: $E(Og) \rightarrow \{\mathbf{owned}, \mathbf{any}\}$ gives the assignment of dominator annotations to the edges of Og. It is defined as follows:

$$\mathit{dom}(o_i \rightarrow o_j) = \begin{cases} \mathbf{owned} & \text{if } o_i \rightarrow o_j \in \mathit{Boundary}(o_i) \\ \mathbf{any} & \text{otherwise} \end{cases}$$

Let $Og^r_{P_e}$ be any run-time object graph. Function dom_{P_e}: $E(Og^r_{P_e}) \rightarrow \{\mathbf{owned}, \mathbf{any}\}$ gives the assignment of dominator annotations to the edges of $Og^r_{P_e}$. It is defined in the obvious way:

$$\mathit{dom}_{P_e}(o^r_i \rightarrow o^r_j) = \mathit{dom}(o_i \rightarrow o_j).$$

That is, each run-time edge $o^r_i \rightarrow o^r_j$ receives the dominator annotation assigned to its representative $o_i \rightarrow o_j$. dom_{P_e} induces an ownership tree as follows:

(Dominator ownership tree $\mathcal{D}_{P_e}$ for program execution P_e.) Let $Og^r_{P_e}$ be the run-time object graph for P_e and let dom_{P_e} be the assignment of dominator annotations to the edges of $Og^r_{P_e}$. $\mathcal{D}_{P_e}$ is construted as follows:

foreach $o^r_i \rightarrow o^r_j \in Og^r_{P_e}$ s.t. $\mathit{dom}_{P_e}(o^r_i \rightarrow o^r_j)$ is **owned**
  add $o^r_i \rightarrow o^r_j$ to $\mathcal{D}_{P_e}$
foreach $o^k_i \in Og^r_{P_e}$ without parent in $\mathcal{D}_{P_e}$
  add `root`$\rightarrow o^r_k$ to $\mathcal{D}_{P_e}$

The following theorem formalizes the correctness requirements imposed by the owner-as-dominator protocol:

Theorem 1. *(Correctness of owner-as-dominator inference). Let $Og^r_{P_e}$ be the run-time object graph for execution P_e, and let $\mathcal{D}_{P_e}$ be the dominator ownership tree for P_e as defined above. The following holds:*

(1) $\mathcal{D}_{P_e}$ is a tree.
(2) For every $o^r_i \rightarrow o^r_j \in \mathcal{D}_{P_e}$, o^r_i dominates o^r_j in $Og^r_{P_e}$.

The theorem follows easily from Lemma 1.

Note that our inference simplifies the classical ownership types [7] because it does not consider ownership parameters. We believe that the dominance boundary information can be successfully used to reason about ownership parameters, and we plan to extend our work with such reasoning in the future.

5 Analyses Needed for the Inference of Modifier Annotations

In addition to object graph and dominance boundary information, the inference of modifier annotations requires information about object modification. Traditionally, reasoning about object modification is done by using method purity annotations (i.e., annotations that designate a method as side-effect free) [11]. This approach has two disadvantages. First, it places a burden on the programmer to provide correct and precise method purity annotations. Second, method purity typically forbids *all updates*, which is a stronger requirement than needed for the inference of modifier annotations; using method purity annotations could lead to imprecise assignment of modifier annotations.

This section presents several analyses that are needed for the inference of modifier annotations. The ultimate goal is to capture necessary information about object modification automatically (i.e., without annotations) and precisely.

Section 5.1 defines two important notions: *method sequence* and *object modification*. Sections 5.2, 5.3, 5.4, and 5.5 present the analyses that capture information about object modification in the context of modifier annotations.

5.1 Notation and Terms

Notation for paths Notation $o_i^r \rightarrow^* o_j^r$ denotes a path of 0 or more edges, and $o_i^r \rightarrow^+ o_j^r$ denotes denotes a path of 1 or more edges in some run-time object graph $Og^r_{P_e}$. Analogously, $o_i \rightarrow^* o_j$ denotes a path of 0 or more edges, and $o_i \rightarrow^+ o_j$ denotes a path of 1 or more edges in Og.

Method Sequence The notion of method sequence is central to the analysis. It represents the transfer of control between distinct run-time objects.

Notation $o^r.m()$ denotes that instance method m is invoked on receiver o^r. Notation $o_1^r.m_1() \rightarrow o_2^r.m_2()$ denotes a run-time *method sequence*. It represents that instance method m_1 invoked on receiver o_1^r calls instance method m_2 on receiver o_2^r. Method sequence $o_1^r.m_1() \rightarrow o_2^r.m_2()$ happens as follows: m_1 invoked on receiver o_1^r, executes a call site $p.m_2()$, $p \neq$ `this` (i.e., $p.m_2()$ is in m_1), where p refers to o_2^r. This leads to the invocation of the appropriate m_2 on receiver o_2^r.[6]

[6] Method sequences "hide" calls through `this`; for example, if there is m_1 invoked on receiver o_1^r, then m_1 executes `this`.$m_1'()$ and in turn, m_1' executes a call site $p.m_2()$, $p \neq$ `this`, where p refers to o_2^r, then there is a method sequence $o_1^r.m_1() \rightarrow o_2^r.m_2()$. Calls through `this` require several special cases in the analysis; they are handled correctly in the implementation, but for brevity, discussion is omitted.

For example, consider method `C.C` in Figure 1. `C.C` is invoked on receiver $\mathbf{o_C}$, and when `a.mod` is true, it executes call site `a.off()`, which leads to the execution of method `A.off` on receiver $\mathbf{o_A}$. Therefore, there is a method sequence $\mathbf{o_C}$.C.C() $\rightarrow \mathbf{o_A}$.A.off().

For convenience, we treat field accesses not through `this` (i.e, $p = q.f$, $q \neq$ `this` and $p.f = q$, $p \neq$ `this`), and array accesses (i.e., $p = q[i]$ and $p[i] = q$) as special method calls. Notation $o_1^r.m_1() \rightarrow o_2^r.rd$ denotes that method $m_1()$ invoked on receiver o_1^r executes a read $p = q.f$, $p \neq$ `this` where q refers to o_2^r. Similarly, $o_1^r.m_1() \rightarrow o_2^r.wr$ denotes that method $m_1()$ invoked on receiver o_1^r executes a write $p.f = q, p \neq$ `this` where p refers to o_2^r.

Notation $o_1^r.m_1() \rightarrow^* o_2^r.m_2()$ denotes that $o_2^r.m_2()$ is reachable through zero or more method sequences from $o_1^r.m_1()$: we have $o_1^r.m_1() \rightarrow o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow ... \rightarrow o_2^r.m_2()$ (in other words, $o_1^r.m_1()$ eventually calls $o_2^r.m_2()$).

Object Modification Recall that the correctness theorem for modifier annotations is stated in terms of the notion of *object modification*; it is necessary to give a precise definition of object modification.

We say that $o^r.m()$ is an *update* of object o^r if m executes a statement `this`.$f = q$, or m is a *wr*. (i.e., $o^r.m()$ updates a field of o^r).

We say that object o_i^r *modifies* object o_j^r, if the following conditions are true:

(i) There is a method sequence path $o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow^* o_k^r.m_k()$ (i.e., a sequence of stack frames $o_i^r.m_i()$, $o_j^r.m_j()$, etc. that leads to $o_k^r.m_k()$.
(ii) $o_k^r.m_k()$ is an update.
(iii) There is an object o^r such that there is a local variable l (including `this`) on a stack frame before $o_j^r.m_j$, which refers to o^r, and $o^r \xrightarrow{f}{}^* o_k^r$ (i.e., o_k^r is part of visible state, and the update caused by $o_i.m_i() \rightarrow o_j.m_j()$ is visible after the execution of $o_j^r.m_j()$).

To the best of our understanding, this definition coincides with the definition of object modification in Universes [10, 25].

It is easy to see that this definition subsumes the more standard definition that uses the notion of method purity. That is, o_i^r modifies o_j^r if one of the following is true:

(1) Object o_i^r updates a field of o_j^r. That is, there is a method executed on receiver o_i^r which executes a statement $p.f = q$, $p \neq$ `this` and p refers to o_j^r.
or
(2) Object o_i^r calls an impure method on o_j^r. That is, there is a method executed on receiver o_i^r which executes a statement $p.m()$, $p \neq$ `this` where p refers to o_j^r and $p.m()$ dispatches to *impure* method m_j (i.e., $o_j^r.m_j()$ leads to an update of an object o_k^r which is visible *after* the execution of $o_j^r.m_j()$).

5.2 Method Sequence Analysis

The method sequence analysis infers method sequences $o_i.m_i() \rightarrow o_j.m_j()$ which approximate run-time method sequences as defined in Section 5.1. Method sequence information helps propagate updates, and reason about object modification precisely and efficiently.

procedure *computeMethodSequences*
uses Og
input -
output Og^+
[1] foreach statement $l.m_j()$, $l \neq$ `this`, in method m
[2] foreach $o_i \rightarrow o_j \in Og$, s.t., $o_i \in Pt(\texttt{this}_m) \wedge o_j \in Pt(l)$
[3] $m_j = dispatch(l.m_j(), o_j)$
[4] add $o_i.m_i() \rightarrow o_j.m_j$ to Og^+

Fig. 9. Method sequence analysis.

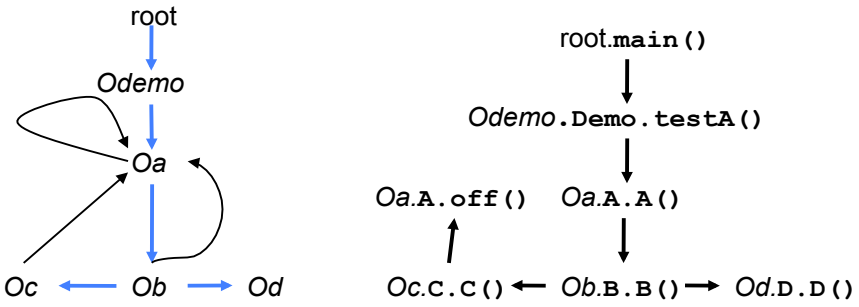


Fig. 10. Og and Og^+ for Example 1.

The analysis uses Og and outputs the augmented object graph Og^+. The nodes in Og^+ are the tuples $o.m()$, and the edges represent method sequences. If there is an execution that exhibits method sequence $o_i^r.m_i() \rightarrow o_j^r.m_j()$, then there is a representative method sequence $o_i.m_i() \rightarrow o_j.m_j() \in Og^+$.

Figure 10 presents the method sequence analysis. Line 1 identifies call statements $l.m_j()$; these statements trigger method sequences. Line 2 identifies edges $o_i \rightarrow o_j \in Og$ affected by $l.m_j()$—these are the edges where o_i is a receiver of the enclosing method m, and o_j is a receiver at the call $l.m_j()$. Line 3 identifies method m_j — the run-time target dispatched at call site $l.m_j()$ with receiver o_j. Line 4 adds $o_i.m_i() \rightarrow o_j.m_j()$ to the augmented object graph Og^+.

Example Consider the code in Figure 1. The object graph and the augmented object graph for this example are shown in Figure 10. Call `d.textA(...)` at line 2 causes method sequence `root.main()` $\rightarrow$ $\mathbf{o_{Demo}}$`.Demo.testA()`. Constructor call `a = new A()` at line 3 causes $\mathbf{o_{Demo}}$`.Demo.testA()` $\rightarrow$ $\mathbf{o_A}$`.A.A()` (we have $\mathbf{o_{Demo}} \in Pt(\texttt{this}_{\texttt{Demo.testA}})$ and $\mathbf{o_A} \in Pt(\texttt{a})$). Call `a.off()` at line 10 causes $\mathbf{o_C}$`.C.C()` $\rightarrow$ $\mathbf{o_A}$`.A.off()` (we have $\mathbf{o_C} \in Pt(\texttt{this}_{\texttt{C.C}})$ and $\mathbf{o_A} \in Pt(\texttt{a})$).

5.3 Minimal Boundary Analysis

The minimal boundary analysis takes as input an edge $o_i \rightarrow o_j$ and computes the set of "closest dominators" o_k, of $o_i \rightarrow o_j$. These o_k's are such that $Boundary(o_k)$ are the *minimal boundaries* enclosing $o_i \rightarrow o_j$ — roughly, that means that every other $Boundary(o'_k)$ enclosing $o_i \rightarrow o_j$ is larger than some $Boundary(o_k)$ (i.e., $Boundary(o'_k) \supset Boundary(o_k)$). This information is needed to confine

updates as deep in the dominance hierarchy as possible, and compute as deep an ownership tree as possible.

The minimal boundary analysis is given in Figure 11. It uses Og and boundary information, takes as input edge $o_i \rightarrow o_j$, and computes set $minBoundaries(o_i \rightarrow o_j)$. The correctness result for this computation is given by the following lemma.

Lemma 2. *Let $o_i^r \rightarrow o_j^r$, represented by $o_i \rightarrow o_j$, be an edge in some $Og_{P_e}^r$. Let $minBoundaries(o_i \rightarrow o_j)$ be the set computed by the analysis in Figure 11. There exists $o_k^r \in Og_{P_e}^r$, $o_k^r \neq o_i^r$ and $o_k^r \neq o_j^r$, represented by o_k, such that (1) $o_k \in minBoundaries(o_i \rightarrow o_j)$ and (2) the representative of every path $o_k^r \rightarrow^* o_i^r \rightarrow o_j^r$ is in $Boundary(o_k)$.*

Informally, the lemma states that set $minBoundaries(o_i \rightarrow o_j)$ "covers" all run-time edges $o_i^r \rightarrow o_j^r$. The proof of this lemma is presented in Appendix A.

procedure *computeMinBoundaries*
uses Og, $Boundary$
input $o_i \rightarrow o_j \in Og$, $createPath \subseteq O$, $o \in O$
output $minBoundaries(o_i \rightarrow o_j)$

[1] if $o_i \rightarrow o_j \in Boundary(o) \wedge createPath \subseteq Boundary(o) \wedge o \neq o_i \wedge o \neq o_j$
[2] add o to $minBoundaries(o_i \rightarrow o_j)$;
[3] else
[4] foreach *create* edge $o' \rightarrow o \in Og$
[5] $computeMinBoundaries(o_i \rightarrow o_j, createPath \cup \{o'\}, o')$

Fig. 11. Minimal boundaries analysis.

computeMinBoundaries is a recursive procedure; at the top level, it is called with $computeMinBoundaries(o_i \rightarrow o_j, \{o_i\}, o_i)$. It starts at o_i and follows *create* edges backwards, keeping the create edge path to o_i in *createPath*. When it reaches an o such that the boundary of o contains $o_i \rightarrow o_j$ and the create path *createPath*, and $o \neq o_i$ and $o \neq o_j$, the analysis adds o to $minBoundaries(o_i \rightarrow o_j)$ and the search stops.[7]

Example Consider the object graph in Figure 2(i), and consider the minimal boundary computation for edge $\mathbf{o_C} \rightarrow \mathbf{o_A}$. At the top level, *computeMinBoundaries* is called with arguments $\mathbf{o_C} \rightarrow \mathbf{o_A}$, $\{\mathbf{o_C}\}$, $\mathbf{o_C}$. The search proceeds along the create path until *computeMinBoundaries* is called with arguments $\mathbf{o_C} \rightarrow \mathbf{o_A}$, $\{\mathbf{o_C}, \mathbf{o_B}, \mathbf{o_A}, \mathbf{o_{Demo}}\}$, $\mathbf{o_{Demo}}$. The analysis adds $\mathbf{o_{Demo}}$ to $minBoundaries(\mathbf{o_C} \rightarrow \mathbf{o_A})$; in fact $\mathbf{o_{Demo}}$ is the only object in $minBoundaries(\mathbf{o_C} \rightarrow \mathbf{o_A})$.

[7] For simplicity, we assume that the create paths do not contain cycles; it may happen (since this is an analysis path) that a create path contains a cycle; our implementation handles this case correctly.

procedure *computeMod*
uses Og, Og^+, *Boundary*, *minBoundaries*
input -
output *Mod*

[1] foreach update $o_j.m_j()$ add $o_j.m_j()$ to $W1$
[2] while $W1$ not empty
[3] remove $o_j.m_j()$ from $W1$
[4] mark $o_j.m_j()$ as visited
[5] foreach $o_i.m_i() \rightarrow o_j.m_j() \in Og^+$
[6] add $o_i \rightarrow o_j$ to *Mod*
[7] if $o_i \rightarrow o_j \in Boundary(o_i)$
[8] if $o_i \overset{f}{\rightarrow}^* o_j \wedge o_i.m_i()$ not visited in $W1$
[9] add $o_i.m_i()$ to $W1$
[10] else
[11] foreach $o_k \in minBoundaries(o_i \rightarrow o_j)$
[12] *propUp*=false
[13] if $o_k \overset{f}{\rightarrow}^* o_j$ *propUp*=true
[14] *propagateMod*$(o_i.m_i(), o_k, propUp, W1)$

procedure *propagateMod*
uses -
input $o_i.m_i(), o_k, propUp, W1$
output -

[1] $W2 = \{o_i.m_i()\}$
[2] mark $o_i.m_i()$ as visited in $W2$
[3] while $W2$ is not empty
[4] take $o_i.m_i()$ from $W2$
[5] foreach $o.m() \rightarrow o_i.m_i() \in Og^+$, s.t., $o \rightarrow o_i \in Boundary(o_k)$
[6] add $o \rightarrow o_i$ to *Mod*
[7] if $o == o_k \wedge propUp \wedge$ $o.m()$ not visited in $W1$
[8] add $o.m()$ to $W1$
[9] if $o.m()$ not visited in $W2$
[10] add $o.m()$ to $W2$

Fig. 12. Object modification analysis.

5.4 Object Modification Analysis

This object modification analysis computes set *Mod* which approximates run-time object modification: if there is a run-time edge $o_i^r \rightarrow o_j^r$ such that o_i^r modifies o_j^r according to the definition in Section 5.1, then the representative of this edge, $o_i \rightarrow o_j$, is in *Mod*.

The following lemma formalizes this property.

Lemma 3. *Let $Og^r_{P_e}$ be any run-time object graph. Let $o_i^r \rightarrow o_j^r \in Og^r_{P_e}$, represented by $o_i \rightarrow o_j$, be an edge such that o_i^r modifies o_j^r. Then $o_i \rightarrow o_j$ is in Mod.*

The proof of the lemma is presented in Appendix A.

The analysis is presented in Figure 12. It uses Og, Og^+, boundary information and minimal boundary information, and computes set *Mod*. Informally, it considers each update $o_j.m_j()$ (i.e., m_j contains a statement `this`.$f = q$ or m_j is a *wr*), and propagates this update up Og until all object modifications are discovered; the analysis keeps an update as deep in the dominance hierarchy as possible.

The analysis considers each method sequence $o_i.m_i() \rightarrow o_j.m_j()$ (line 5), and adds the corresponding edge $o_i \rightarrow o_j$ to *Mod* (line 6); clearly, if $o_j.m_j()$ is an update, then $o_i \rightarrow o_j$ represents an object modification. The correctness for the computation is given by the following lemma.

Subsequently, the analysis considers two cases. If o_j is in the boundary of o_i (lines 7-9), the update may be hidden behind o_i (i.e., o_j is not a transitive

field of o_i) or it may not be hidden behind o_i (i.e., o_j is a transitive field of o_i). If hidden, propagation stops; if not hidden $o_i.m_i$ is added to the worklist $W1$ for further propagation up the graph. If o_j is not in the boundary of o_i (lines 10-14), the analysis considers each minimal boundary o_k. Again, the update may be hidden behind o_k (i.e., if o_j is not a transitive field of o_k), or it may not be hidden behind o_k (i.e., o_j is a transitive field of o_k). In both cases, the update is propagated within the boundary of o_k by *propagateMod*. If the update is hidden, *propUp* is set to false, and *propagateMod* does not update *W1* (i.e., propagation stops at o_k); otherwise, *propUp* is set to true, and *propagateMod* adds new tuples to *W1* (i.e., propagation proceeds accordingly).

Auxiliary procedure *propagateMod* propagates an update $o_i.m_i()$ in the boundary of o_k; it traverses backwards each method sequence path $o_k.m_k() \rightarrow^* o_i.m_i()$ (a path that leads to the update $o_i.m_i()$), and adds each edge $o \rightarrow o_i$ along this path to *Mod* (line 5 in *propagateMod*).

Example Consider the object graph in Figure 2(i), and the propagation of update $\mathbf{o_A}$.`A.off`(). The analysis discovers $\mathbf{o_C}$.`C.C`() $\rightarrow$ $\mathbf{o_A}$.`A.off`() (line 6), adds edge $\mathbf{o_C} \rightarrow \mathbf{o_A}$ to *Mod* (line 7), and continues the examination of the edge. Since $\mathbf{o_C} \rightarrow \mathbf{o_A} \notin \mathit{Boundary}(\mathbf{o_C})$, the analysis proceeds to lines 10-14. As we saw earlier, the only object in $\mathit{minBoundaries}(\mathbf{o_C} \rightarrow \mathbf{o_A})$ is $\mathbf{o_{Demo}}$; $\mathbf{o_A}$ is not a transitive field of $\mathbf{o_{Demo}}$ and therefore *propagateMod* is called with arguments $\mathbf{o_C}$.`C.C`(), $\mathbf{o_{Demo}}$, false and $W1$. *propagateMod* visits $\mathbf{o_B}$.`B.B`(), $\mathbf{o_A}$.`A.A`() and $\mathbf{o_{Demo}}$.`testA`(), and adds edges $\mathbf{o_B} \rightarrow \mathbf{o_C}$, $\mathbf{o_A} \rightarrow \mathbf{o_B}$ and $\mathbf{o_{Demo}} \rightarrow \mathbf{o_A}$ to *Mod*.

It is important to note that since *propUp* is false, $\mathbf{o_{Demo}}$.`testA`() *is not added* to $W1$ (i.e., the update is not propagated beyond the boundary of $\mathbf{o_{Demo}}$); This is correct and precise because the updates caused by `testA` are not visible after the execution of `testA`; as a result, edge `root` $\rightarrow$ $\mathbf{o_{Demo}}$ is typed **any**. In contrast, an approach based on method purity annotations would determine that `testA` is not pure because it causes many updates (even though all these updates are to invisible objects!); as a result, the approach would determine that `root` modifies $\mathbf{o_{Demo}}$ and the edge is typed **peer** or **owned**.

The *Mod* set equals $\{\mathbf{o_C} \rightarrow \mathbf{o_A}, \mathbf{o_b} \rightarrow \mathbf{o_D}, \mathbf{o_B} \rightarrow \mathbf{o_C}, \mathbf{o_A} \rightarrow \mathbf{o_B}, \mathbf{o_{Demo}} \rightarrow \mathbf{o_A}\}$.

5.5 Unique Modification Analysis

Another analysis needed by the inference of modifier annotations is the unique modification analysis. This analysis helps filter out some observational exposure, and allows us to assign modifier annotation **owned** to edges that have dominator annotation **any**.

Let $o_i \rightarrow o_j \in \mathit{Boundary}(o_k)$ be a modification edge, i.e., $o_i \rightarrow o_j \in \mathit{Mod}$. We define a predicate $\mathit{uniqueMod}(o_i \rightarrow o_j, o_k)$; it denotes whether edge $o_i \rightarrow o_j$ is a unique modification in the boundary of o_k. Predicate $\mathit{uniqueMod}(o_i \rightarrow o_j, o_k)$ returns true if the following conditions are true: (1) $o_i \rightarrow o_j$ is a *create* edge, and not an *in* or an *out* edge, and (2) there is no other edge $o \rightarrow o_j \in \mathit{Boundary}(o_k)$, such that $o \rightarrow o_j \in \mathit{Mod}$; the predicate returns false otherwise.

Unique modification is when o_i creates and modifies o_j, and o_i "lends" o_j to other objects o, but the access of o to o_j is only observational (i.e., read-only). [8]

Next, we define a predicate $uniqueMod(o_i \rightarrow o_j)$. It returns true if for every $o_k \in minBoundaries(o_i \rightarrow o_j)$ we have $uniqueMod(o_i \rightarrow o_j, o_k)$; it returns false otherwise. In other words, an edge $o_i \rightarrow o_j$ is a unique modification if it is a unique modification within each of its enclosing minimal boundaries.

Example Consider the example in Figure 3 and its corresponding object graph in Figure 2(ii). Consider edge $\mathbf{o_{Cx}} \rightarrow \mathbf{o_{d[]}}$. There is only one minimal boundary enclosing this edge, $Boundary(\mathbf{o_X})$. One can see that the conditions for $uniqueMod(\mathbf{o_{Cx}} \rightarrow \mathbf{o_{d[]}}, \mathbf{o_X})$ hold: (i) edge $\mathbf{o_{Cx}} \rightarrow \mathbf{o_{d[]}}$ is a *create* edge, but not an *in* or *out* edge, and (ii) the only other edge, $\mathbf{o_i} \rightarrow \mathbf{o_{d[]}}$ is not a modification edge (in fact, there is no method sequence associated with this edge at all). The concept of unique modification allows us to assign modifier annotation **owned** to edge $\mathbf{o_{Cx}} \rightarrow \mathbf{o_{d[]}}$, even though this edge has dominator annotation **any**.

6 Inference of Modifier Annotations

In general, there are many possible assignments of modifier annotations to object graph edges that meet the correctness requirements stated in Section 2. For example, one such assignment assigns annotations **owned** to object graph edges that originate at `root`, and annotations **peer** to all other edges. This assignment states that all objects in the program are peers owned by `root`. It creates a flat ownership tree — there is a single owner, `root`, and all other objects are children of `root`. This assignment is hardly useful. The goal of the inference of modifier annotations is to create as deep an ownership hierarchy as possible.

Function mod: $E(Og) \rightarrow \{\mathbf{any}, \mathbf{owned}, \mathbf{peer}\}$ gives the assignment of modifier annotations to the edges of Og. It is defined by the analysis in Figure 13.

First, the analysis assigns annotation **any** to each edge (line 1). Next, it examines each object modification edge $o_i \rightarrow o_j \in Mod$. If $o_i \rightarrow o_j$ is in the boundary of o_i (i.e., the modification is confined in the boundary of the triggering object), the analysis assigns annotation **owned** to it (line 3). If the edge is exposed outside of the boundary of o_i, but the exposure remains observational, the analysis assigns annotation **owned** as well (line 4). Otherwise, the analysis assigns annotation **peer** (line 5). This part of the analysis (lines 2-5) ensures that if o_i^r modifies o_j^r, then the edge $o_i^r \rightarrow o_j^r$ receives annotation **owned** or **peer**, and therefore, either o_i^r is the owner of o_j^r, or o_i^r and o_j^r are peers in the ownership tree, as required by owner-as-modifier.

Subsequently, the analysis calls procedure *checkConflict* on each edge $o_i \rightarrow o_j$ that has received annotation **owned**. This part of the analysis (lines 6-8) ensures that the **owned** and **peer** annotations induce well-defined ownership trees (i.e., no object has more than one owner).

[8] The condition that $o_i \rightarrow o_j$ is not an *in* or an *out* edge is needed for correctness: it ensures that an edge $o_i \rightarrow o_j$ cannot represent two distinct run-time edges that end at o_j^r (e.g., $o_{i_1}^r \rightarrow o_j^r$, and $o_{i_2}^r \rightarrow o_j^r$).

procedure *inferModifierAnnotations*
uses *Og*, *Boundary*, *Mod*, *uniqueMod*
input -
output *mod*
[1] foreach $o_i \rightarrow o_j \in Og$ $\quad mod(o_i \rightarrow o_j) = \textbf{any}$

[2] foreach $o_i \rightarrow o_j \in Mod$
[3] $\quad$ if $o_i \rightarrow o_j \in Boundary(o_i)$ $\quad mod(o_i \rightarrow o_j) = \textbf{owned}$
[4] $\quad$ else if $uniqueMod(o_i \rightarrow o_j)$ $\quad mod(o_i \rightarrow o_j) = \textbf{owned}$
[5] $\quad$ else $\quad mod(o_i \rightarrow o_j) = \textbf{peer}$

[6] while *mod* changes
[7] $\quad$ foreach $o_i \rightarrow o_j \in Og$ s.t. $mod(o_i \rightarrow o_j)$ is **owned**
[8] $\quad\quad$ *checkConflict*$(o_i \rightarrow o_j)$

procedure *checkConflict*
uses -
input $o_i \overset{\textbf{owned}}{\rightarrow} o_j$
output -
[1] if $\exists p$: $o_i \overset{\textbf{owned}}{\rightarrow} o_j \overset{\textbf{peer}}{\rightarrow}{}^{*} o' \overset{\textbf{peer}}{\rightarrow} o$ s.t. $p \notin Boundary(o_i)$
[2] $\quad mod(o_i \rightarrow o_j) = \textbf{peer}$
[3] else if $\exists p_1$: $o_i \overset{\textbf{peer}}{\rightarrow}{}^{*} o \wedge \exists p_2$: $o_i \overset{\textbf{owned}}{\rightarrow} o_j \overset{\textbf{peer}}{\rightarrow}{}^{*} o$
[4] $\quad mod(o_i \rightarrow o_j) = \textbf{peer}$

Note: $o_i \overset{\textbf{owned}}{\rightarrow} o_j$ denotes that $mod(o_i \rightarrow o_j)$ is **owned**.
$o_j \overset{\textbf{peer}}{\rightarrow}{}^{*} o'$ denotes the empty path o_j, and any path of one or more **peer** edges from o_j to o'.

Fig. 13. Inference of modifier annotations.

Procedure *checkConflict* performs two checks. First, it checks if there is a path $o_j \overset{\textbf{peer}}{\rightarrow}{}^{*} o' \overset{\textbf{peer}}{\rightarrow} o$ that is not in the boundary of o_i (lines 1-2 in *checkConflict*). Without loss of generality we may assume that $o_j \rightarrow^{*} o' \in Boundary(o_i)$, and $o' \rightarrow o \notin Boundary(o_i)$. This means that some object in the boundary of o_i, namely o' modifies an object o from an enclosing boundary; this modification forces o' and o to be peers, and the owner of these peers is an object from an enclosing boundary, not o_i. The annotation of $o_i \rightarrow o_j$ must be changed to **peer**. Second, the procedure checks if there are two paths from o_i to o, one that forces o_i to be the owner of o, and another that forces o_i to be a peer of o (lines 3-4 in *checkConflict*). Again, the annotation of $o_i \rightarrow o_j$ must be changed to **peer**. This essentially "flattens" the dominance boundary of o_i, making (some) of the objects in this boundary peers to the objects from an enclosing boundary.

Example. Consider our running example in Figures 1 and Figure 2(i). *Mod* equals $\{\mathbf{o_C} \rightarrow \mathbf{o_A}, \mathbf{o_B} \rightarrow \mathbf{o_D}, \mathbf{o_B} \rightarrow \mathbf{o_C}, \mathbf{o_A} \rightarrow \mathbf{o_B}, \mathbf{o_{Demo}} \rightarrow \mathbf{o_A}\}$. Edge $\mathbf{o_C} \rightarrow \mathbf{o_A}$ receives annotation **peer** ($\mathbf{o_C} \rightarrow \mathbf{o_A} \notin \mathit{Boundary}(\mathbf{o_C})$, and $\mathit{uniqueMod}(\mathbf{o_C} \rightarrow \mathbf{o_A})$ is *false*). All other edges in *Mod* receive annotation **owned** (due to line 3). Edges $\texttt{root} \rightarrow \mathbf{o_{Demo}}$ and $\mathbf{o_B} \rightarrow \mathbf{o_A}$ remain **any**.

However, this initial assignment does not induce an ownership tree. Edge $\mathbf{o_{Demo}} \overset{\mathbf{owned}}{\rightarrow} \mathbf{o_A}$ forces $\mathbf{o_{Demo}}$ to be the owner of $\mathbf{o_A}$, and edges $\mathbf{o_B} \overset{\mathbf{owned}}{\rightarrow} \mathbf{o_C} \overset{\mathbf{peer}}{\rightarrow} \mathbf{o_A}$ force $\mathbf{o_B}$ to be the owner of peers $\mathbf{o_C}$ and $\mathbf{o_A}$.

Procedure *checkConflict* is called on edge $\mathbf{o_B} \overset{\mathbf{owned}}{\rightarrow} \mathbf{o_C}$. It detects a conflict at lines 1-2 in *checkConflict*: namely, there is a path p: $\mathbf{o_B} \overset{\mathbf{owned}}{\rightarrow} \mathbf{o_C} \overset{\mathbf{peer}}{\rightarrow} \mathbf{o_A}$ which is not in the boundary of $\mathbf{o_B}$. *checkConflict* changes the annotation of $\mathbf{o_B} \rightarrow \mathbf{o_C}$ to **peer**.

Next, *checkConflict* is called on edge $\mathbf{o_A} \overset{\mathbf{owned}}{\rightarrow} \mathbf{o_B}$. It detects a conflict at lines 3-4: namely, there is a path p_1: $\mathbf{o_A} \overset{\mathbf{peer}}{\rightarrow}{}^{*} \mathbf{o_A}$ (the trivial empty path) which states that $\mathbf{o_A}$ is a peer of itself, and there is a path p_2: $\mathbf{o_A} \overset{\mathbf{owned}}{\rightarrow} \mathbf{o_B} \overset{\mathbf{peer}}{\rightarrow} \mathbf{o_C} \overset{\mathbf{peer}}{\rightarrow} \mathbf{o_A}$ which forces $\mathbf{o_A}$ to be the owner of itself. *checkConflict* changes the annotation of $\mathbf{o_A} \rightarrow \mathbf{o_B}$ to **peer**. The final set of modifier annotations is shown in Figure 2(i).

Again, let $Og^r_{P_e}$ be any run-time object graph. Function, mod_{P_e}: $E(Og^r_{P_e}) \rightarrow \{\mathbf{any}, \mathbf{owned}, \mathbf{peer}\}$ gives the assignment of modifier annotations to the edges of $Og^r_{P_e}$. It is defined in the obvious way:

$$mod_{P_e}(o^r_i \rightarrow o^r_j) = mod(o_i \rightarrow o_j).$$

mod_{P_e} induces an ownership tree as follows:

(Modifier ownership tree $\mathcal{M}_{P_e}$ for program execution P_e). Let $Og^r_{P_e}$ be the run-time object graph for P_e and let mod_{P_e} be the assignment of modifier annotations to the edges of $Og^r_{P_e}$. $\mathcal{M}_{P_e}$ is construted as follows:

foreach $o^r_i \overset{\mathbf{owned}}{\rightarrow} o^r_j \overset{\mathbf{peer}}{\rightarrow}{}^{*} o^r_k \in Og^r_{P_e}$
  add $o^r_i \rightarrow o^r_k$ to $\mathcal{M}_{P_e}$
foreach $o^r_k \in Og^r_{P_e}$ without parent in $\mathcal{M}_{P_e}$,
  add $\texttt{root} \rightarrow o^r_k$ to $\mathcal{M}_{P_e}$

Theorem 2. *(Correctness of owner-as-modifier inference). Let $Og^r_{P_e}$ be the run-time object graph for execution P_e, and let $\mathcal{M}_{P_e}$ be the dominator ownership tree for P_e as defined above. The following holds:*

(1) $\mathcal{M}_{P_e}$ is a tree.
(2) For every $o^r_i \rightarrow o^r_j \in Og^r_{P_e}$, such that o^r_i modifies o^r_j, either o^r_i is the owner of o^r_i (i.e, $o^r_i \rightarrow o^r_j \in \mathcal{M}_{P_e}$), or o^r_i and o^r_j are peers (i.e, siblings in $\mathcal{M}_{P_e}$).

7 Empirical Results

The ownership inference analysis is implemented in Java using the Soot 2.2.3 [36] and Spark [18] frameworks; specifically, it is implemented as a client of the Andersen-style points-to analysis provided by Spark. We performed whole-program analysis with the Sun JDK 1.4.1 libraries. All experiments were done on a 900MHz Sun Fire 380R machine with 4GB of RAM. The implementation which includes Soot and Spark, was run with a max heap size of 1.4GB; however, all benchmarks ran within a memory footprint of 800MB.

Native methods are handed by utilizing the models provided by Soot. Reflection is handled by specifying the dynamically loaded classes, which Spark uses to appropriately resolve reflection calls. This approach is used in other whole-program analyses based on Soot and Spark [35].

Our benchmark suite is presented in Table 1. It includes 6 software components (from `gzip` through `number`) which we have used in previous work [22, 20] and are familiar with. Each component is transformed into a whole program by attaching an artificial `main` method to it; the artificial `main` "completes" the component and allows whole-program analysis [34]. In addition, the suite includes 12 whole programs: `javad`, `jdepend`, `JATLite` and `undo`, benchmarks `soot-c` and `sablecc-j` from the Ashes suite [1], `polyglot`, and `antlr`, `bloat`, `jython`, `pmd` and `ps` from the DaCapo benchmark suite version beta051009 [2]. Column "Methods" in Table 1 shows the size of the benchmarks in terms of the number of methods (user and library) found to be reachable by Spark.

One goal of the empirical study is to contrast the owner-as-dominator and the owner-as-modifier protocols by addressing the following questions: (1) to what extent do **owned** annotations overlap and (2) to what extent do they differ? Another goal of the study is to show that the analysis scales to relatively large programs. Yet another goal of the study is to show that the analysis is adequately precise.

7.1 Results

We report results on instance fields of reference type.

We define the following ordering between dominator annotations: **owned** $\leq$ **any**. Clearly, **any** is less precise than **owned** because **any** "flattens" the ownership tree; if an edge $o_i^r \rightarrow o_j^r$ is **any**, this forces o_j^r to have an owner which is an antecedent of o_i^r in the ownership tree.

To assign a dominator annotation on field f we join the dominator annotations over all edges $o_i \rightarrow o_j \in Og$ that may represent field edges labeled with f:

$$dom(f) = \bigvee_{o_i \rightarrow o_j \in Og \ \wedge \ o_j \in Pt(o_i.f)} dom(o_i \rightarrow o_j)$$

Thus, a field is reported as **owned** only if all its instances in Og are **owned**; a field is reported as **any** otherwise.

We define the following ordering between modifier annotations: **any** $\leq$ **owned** $\leq$ **peer**. Annotation **any** is the "most precise" (i.e., best) because it does not

Program	Methods	Fields	o-as-d **owned**	o-as-m **owned**	**owned**	**peer**	**any**	**owned**	**peer**	**any**	Analysis Time Points-to	Ownership
`gzip`	3819	7	4	3	3	1	0	0	1	2	91s	20s
`zip`	3844	10	5	5	5	0	0	0	2	3	91s	19s
`checked`	3766	2	0	0	0	0	0	0	2	0	91s	18s
`collator`	3868	17	9	9	9	0	0	0	2	6	92s	22s
`breaks`	3822	7	0	0	0	0	0	0	1	6	92s	27s
`number`	3880	3	1	2	1	0	0	1	0	1	93s	31s
`javad`	3838	36	19(53%)	14(39%)	12(33%)	7	0	2	0	15	92s	18s
`jdepend`	3962	29	19(66%)	12(41%)	11(38%)	8	0	1	6	3	93s	22s
`JATLite`	6279	142	35(25%)	36(25%)	32(23%)	3	0	4	95	8	152s	225s
`undo`	5644	289	56(19%)	35(12%)	25(9%)	18	13	8	81	144	183s	174s
`soot`	6046	283	64(23%)	58(20%)	44(16%)	20	0	14	117	88	143s	623s
`sablecc`	7970	284	26(9%)	18(6%)	14(5%)	12	0	4	219	35	184s	165s
`polyglot`	7449	431	56(13%)	60(14%)	39(9%)	15	2	21	257	97	573s	1474s
`antlr`	5102	152	39(26%)	37(24%)	28(18%)	9	2	9	63	41	142s	71s
`bloat`	6402	449	80(18%)	72(16%)	53(12%)	26	1	19	234	116	155s	429s
`jython`	5606	206	60(29%)	69(33%)	52(25%)	5	3	17	92	37	137s	165s
`pmd`	8653	114	48(42%)	40(35%)	36(32%)	10	2	4	31	31	275s	382s
`ps`	5396	19	7(37%)	8(42%)	6(32%)	1	0	2	9	1	137s	582s
Average			30%	26%	21%							

Table 1. Ownership inference results.

impose constraints on the ownership tree; **peer** is the "least precise" (i.e., worst) because it "flattens" the ownership tree — if an edge $o_i^r \rightarrow o_j^r$ is **peer**, this not only forces o_j^r to go up the tree, but it may force a number of other objects to go up the tree (recall Example 1 in Figures 1 and 2).

To assign a modifier annotation on field f we join the modifier annotations over all field edges labeled with f:

$$mod(f) = \bigvee_{o_i \rightarrow o_j \in Og \ \wedge \ o_j \in Pt(o_i.f)} mod(o_i \rightarrow o_j)$$

Thus, a field is reported as **any** if all of its instances are **any**, it is reported as **owned** if all its instances are **onwed** or **any**, and it is reported as **peer** otherwise. Somewhat surprisingly, conflicts were rare (i.e., in the vast majority of cases, different instances of field edges received the same annotation). For example, only 10 fields in `bloat` (out of 449) had an **owned** instance and a **peer** instance. Less than 6 fields had an **owned** instance and a **peer** instance in every other benchmark. Clearly, those fields were counted as **peer**.

Note that the mapping from object graph edges to fields is done only for the purposes of meaningful reporting. It is not the goal of this paper to map the inferred annotations to ownership types [7] or Universe types [10]. We conjecture that such a mapping can be established and we plan to address this problem in the future.

Column "Fields" in Table 1 shows the number of fields (we examined all instance fields in user classes).[9] Column "o-as-d **owned**" shows how many fields were inferred as owned according to the owner-as-modifier protocol, and column "o-as-m **owned**" shows how many fields were inferred as **owned** according to the owner-as-modifier protocol.

The next six columns show the results in greater detail. The first column (under headings **owned** and **owned**) shows the number of fields that were inferred as **owned** according to the owner-as-dominator protocol, and as **owned** according to the owner-as-modifier protocol — i.e., the **owned/owned** fields. The next column (under headings **owned** above, and **peer** below) shows the number of fields that were inferred as **owned** according to the owner-as-dominator, and were inferred as **peer** according to the owner-as-modifier — i.e., the **owned/peer** fields; this column highlights the "strictness" of the owner-as-modifier protocol as it shows how often fields that are dominated by their `this` object become peers to their `this` object due to modifications of objects from enclosing boundaries. The column under headings **any** and **owned** shows the number of fields that were inferred as **any** according to the owner-as-dominator protocol, and were inferred as **owned** according to the owner-as-modifier protocol; this column highlights the "strictness" of the owner-as-dominator protocol as it shows how often fields that are exposed outside of their `this` object, are exposed in a read-olny, observational manner.

On average, for the 12 large benchmarks, 30% of all fields were reported as **owned** according to the owner-as-dominator protocol, and 26% were reported as **owned** according to the owner-as-modifier protocol. 21% of all fields overlapped (i.e., were found to be **owned** by both protocols). Therefore, we conclude that *ownership occurs frequently* in real-world object-oriented programs.

Furthermore, it is notable that almost one third of the o-as-d **owned** fields were reported as non-**owned** according to the owner-as-modifier protocol (i.e., the "strictness" of owner-as-modifier causes almost one third of all dominated fields to become **peer** due to modification of objects from enclosing boundaries). On the other hand, less than one fifth of the o-as-m **owned** fields were reported as non-**owned** according to the owner-as-dominator protocol (i.e., the "strictness" of the owner-as-dominator causes only less than a fifth of the o-as-m **owned** fields to become non-**owned** due to exposure outside of the `this` boundary). However, our investigation (see Sections 7.2 and 7.3) indicates that while the analysis captures o-as-d **owned** fields very precisely, it may underreport o-as-m **owned** fields. We estimate that the actual percentage of **any**/**owned** fields (i.e, the percentage of observationally exposed fields), is slightly higher than the reported 4-5%. Therefore, we conclude, that *the two ownership protocols give rise to different run-time ownership structures.*

[9] This number of fields differs from the number we reported earlier [19, 23]. This is because earlier we counted implicit references in inner classes to the outer class (these references are not present in code, but are present in bytecode and our intermediate representation Jimple). As in our previous work [19] and [22], which uses the same benchmarks, we did not include fields of type `String` and `StringBuffer`.

Multicolumn "Analysis Time" in Table 1 shows the running time (in seconds) of the analysis. Column "Points-to" shows the running time for Spark's points-to analysis, and column "Ownership" shows the running time for the ownership inference (it includes the inference of dominator and modifier annotations). Except for `polyglot` (an outlier both for points-to and ownership), the ownership inference analysis typically completes in much less than 500 seconds and we note that there are opportunities for performance improvement.[10] Therefore, *the analysis scales well even on relatively large programs.*

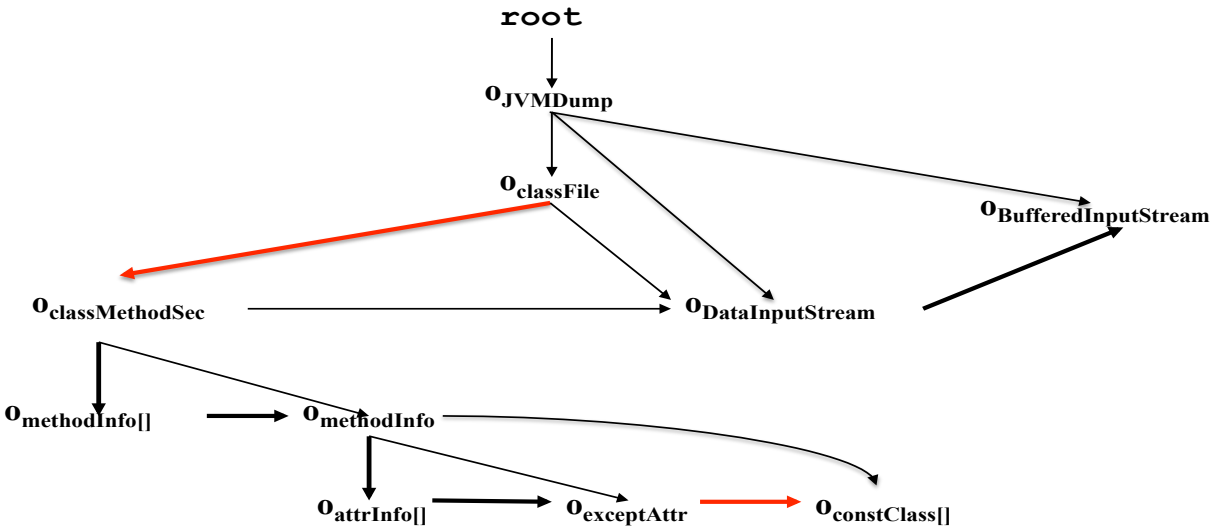


Fig. 14. Partial abstract object graph for `javad`. Thick edges denote field access and thin edges denote stack access.

7.2 Precision

Addressing the issue of analysis precision is highly non-trivial. To the best of our knowledge, there are no established large benchmark programs that have been annotated with ownership types [7] or Universe types [10], and could be used to objectively evaluate an ownership inference analysis.

In order to evaluate the precision of our ownership inference analysis, we performed a study of *absolute precision* [34, 19] on a subset of the fields. Specifically, we considered all fields in components `gzip` through `number` and all fields

[10] For example, running the analysis on newer hardware than the 900MHz Sun Fire 380R, would likely result in a significant improvement in reported wall-clock performance.

in benchmark `javad`. This accounted for a set of 82 fields. Of these, 38 were reported as **owned** and 44 were reported as **any** (i.e., non-owned) according to the owner-as-dominator protocol. 33 were reported as **owned**, 16 were reported as **peer**, and 33 were reported as **any** according to the owner-as-modifier protocol.

To evaluate the precision of the owner-as-dominator inference, we examined every **any** (i.e., non-**owned**) field f and attempted to prove exposure. That is, we attempted to show that there is an execution P_e, such that an object o_j^r stored in field f of object o_i^r, is exposed outside of o_i^r, or more formally, that o_i^r does not dominate o_j^r in the run-time object graph $Og_{P_e}^r$. In *every case*, we were able to prove exposure. In addition, we examined every **owned** field f. Although the analysis is proven safe (and therefore, an **owned** field must be indeed **owned**), we conducted the detailed examination in order to gain further confidence in the functional correctness of the implementation. Again, in *every case*, the **owned** field was indeed **owned**. Therefore, for this set of 82 fields, the owner-as-dominator inference achieves perfect precision.

To evaluate the precision of the owner-as-modifier inference, we examined every **peer** field f and attempted to assign a more precise modifier annotation to it. That is, we attempted to assign one of **any** or **owned** to f, which would result in a deeper (and better) ownership tree. In every case (16 in total), assigning **any** was impossible because the field edge constituted a modification: that is, there existed an execution P_e, such that o_i^r modifies the o_j^r object stored in field f of o_i^r. In 14 out of 16 cases, assigning **owned** was impossible: it lead to an invalid, ownership tree. For only 2 of 16 fields assigning **owned** instead of **peer** was possible. Specifically, fields `saveEntry` and `lastEntry` in class `MergeCollation` in component `collator` were reported as **peer**. These fields were exposed outside of their `this` object; however, they could have been annotated as **owned**, because the exposure was only observational. Therefore, we conclude that for this set of 82 fields, the owner-as-modifier inference achieves very good (but not perfect) precision.

To further evaluate the precision of the owner-as-modifier inference, we ran the analysis on all annotated code examples from [10, 11, 9]. This included `Producer-Consumer` and `Modifying Iterator` from [10], the example in Figure 1 from [11], and `Stack` from [9]. This accounted for a set of 18 fields. For 15 fields, our analysis inferred the same annotation as specified by the manually annotated code example. For 2 fields, our analysis inferred a more precise annotation than the one specified by the manually annotated example.[11] For 1 field, our analysis inferred annotation **peer** instead of **owned**. In this one case of imprecision, the cause of imprecision was exactly the same as the cause of the imprecision for `saveEntry` and `lastEntry`. Section 7.3 elaborates on the cause of imprecision.

Overall, we view these results as very promising. We claim that the analysis exhibits adequate precision on two grounds. First, the analysis reports a large percentage of owned fields: 30% for owner-as-dominator and 26% for owner-as-modifier. Second, our study of absolute precision, revealed very few instances of

[11] For 2 fields that were never referenced by the enclosing class, our analysis inferred annotation **any**, while the manually annotated code example had annotation **peer**.

```
class Main {
  public static void main() {
    Object o = new Object();   //o_Object1
    LinkedList l =
       new LinkedList();       //o_LinkedList
    Iter i = new Iter(l);      //o_Iter
    i.setValue(new Object()); //o_Object2
  }
}
class LinkedList {
  /*@ owned @*/ Node first;
  LinkedList(Object e) {
    first = new Node();        //o_Node
    first.elem = e;
  }
  void set(Node np, Object e) {
    Node n = np;
    n.elem = e;
  }
}
```

```
class Iter {
  /*@ peer @*/ LinkedList list;
  /*@ any @*/ Node pos;
  Iter(LinkedList l) {
    list = l;
    pos = l.first;
  }
  void setValue(Object o) {
    list.set(pos,o);
  }
}
class Node {
  /*@ peer @*/ Node next;
  /*@ any @*/ Object elem;
}
```

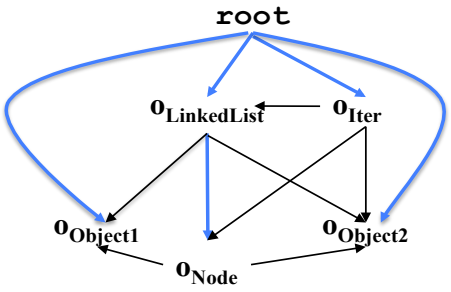


Fig. 15. `Modifying Interator` example.

imprecision: 0 out of 100 examined fields for owner-as-dominator, and 3 out of 100 examined fields for owner-as-modifier.

7.3 Case Studies

We further illustrate our results with two case studies.

Case Study: `javad` We conducted a case study on benchmark `javad`. `javad` is a Java class file disassembler [15]. It has approximately 40 user classes and 4,000 lines of code. `javad` is a relatively small program, and yet it exhibits interesting ownership that highlights the difference between owner-as-dominator and owner-as-modifier.

Figure 14 shows a partial abstract object graph for `javad`. The thick edges denote field access: for example, $o_{\mathbf{classFile}}$ has a field `classMethods` that refers to $o_{\mathbf{classMethodSec}}$. The thin edges denote stack access: for example, the constructor `classFile.classFile()` executes with receiver $o_{\mathbf{classFile}}$, and a local variable in it refers to $o_{\mathbf{DataInputStream}}$.

The disassembler uses a `FileInputStream` object to read a .class file. The `FileInputStream` object (not shown in the graph) is decorated by object $o_{\mathbf{BufferedInputStream}}$ and subsequently by $o_{\mathbf{DataInputStream}}$; the disassembler first accesses $o_{\mathbf{DataInputStream}}$ which redirects access to $o_{\mathbf{BufferedInputStream}}$ which in turn redirects access to the `FileInputStream` object. Object $o_{\mathbf{classFile}}$ represents the class file and $o_{\mathbf{classMethodSec}}$ represents the method section in the class file. $o_{\mathbf{classMethodSec}}$ has an array $o_{\mathbf{methodInfo}[]}$ that stores the representations of the methods, and $o_{\mathbf{methodInfo}}$ represents an individual method. In turn, $o_{\mathbf{methodInfo}}$ has an array $o_{\mathbf{attrInfo}}$ that stores the representations of the attributes of the method and

$\mathbf{o_{exceptAttr}}$ represents one attribute, the exceptions thrown by the method. In turn, $\mathbf{o_{exceptAttr}}$ has a field `exceptTable` and this field refers to array $\mathbf{o_{constClass[]}}$ which stores information about each exception.

Consider edge $\mathbf{o_{classFile}} \rightarrow \mathbf{o_{classMethodSec}}$ (as mentioned earlier, this is an instance of field `classMethods` in class `classFile`); the edge is shown in red. All accesses to objects $\mathbf{o_{classMethodSec}}$ go through object $\mathbf{o_{classFile}}$. Thus, the dominator annotation on this edge, and consequently, on field `classMethods` is **owned**. However, the modifier annotation on this edge is not **owned** but **peer**. This is because $\mathbf{o_{classMethodSec}}$ calls a method on $\mathbf{o_{DataInputStream}}$ which leads to an update of a field of $\mathbf{o_{BufferedInputStream}}$; therefore $\mathbf{o_{classMethodSec}}$ causes a modification of $\mathbf{o_{DataInputStream}}$, and this modification forces the two objects to be peers. Intuitively, an object from the boundary of $\mathbf{o_{classFile}}$, namely $\mathbf{o_{classMethodSec}}$ modifies an object from the enclosing boundary of $\mathbf{o_{JVMDump}}$, namely $\mathbf{o_{DataInputStream}}$; this flattens the dominance boundary of $\mathbf{o_{classFile}}$ causing its children to become its peers. Thus, $\mathbf{o_{classFile}}$ and $\mathbf{o_{classMethodSec}}$ become peers (children of $\mathbf{o_{JVMDump}}$). Consequently, edge $\mathbf{o_{classFile}} \rightarrow \mathbf{o_{classMethodSec}}$ and field `classMethods` receive modifier annotation **peer**. There are 5 other fields in class `classFile` whose dominator annotation is **owned** but whose modifier annotation is **peer** because of modification to the $\mathbf{o_{DataInputStream}}$ object.

Now, consider edge $\mathbf{o_{exceptAttr}} \rightarrow \mathbf{o_{constClass[]}}$ (this is an instance of field `exceptTable` in class `exceptAttr`); the edge is shown in red. $\mathbf{o_{JVMDump}}$ calls a method `print` on $\mathbf{o_{classFile}}$. In turn $\mathbf{o_{classFile}}$'s `print` calls a `print` on $\mathbf{o_{classMethodSec}}$, which in turn calls a `print` on $\mathbf{o_{methodInfo}}$. $\mathbf{o_{methodInfo}}$'s `print` obtains a reference to the table $\mathbf{o_{constClass[]}}$ and accesses the table to print the info about each exception. Due to this reference, $\mathbf{o_{constClass[]}}$ is no longer dominated by $\mathbf{o_{exceptAttr}}$, and therefore the dominator annotation on this edge, and on field `exceptTable` is **any**. The modifier annotation is **owned** however. This is because the edge $\mathbf{o_{methodInfo}} \rightarrow \mathbf{o_{constClass[]}}$ does not cause a modification, or in other words, the exposure of $\mathbf{o_{constClass[]}}$ remains only observational.

Case study: `Modifying Iterator` As another example, consider the code for the `Modifying Iterator` example, and the corresponding object graph in Figure 15. Classes `LinkedList`, `Iter` and `Node` and the modifier annotations on their fields, are taken from [10].

Our owner-as-modifier inference is imprecise when reasoning about edge $\mathbf{o_{LinkedList}} \rightarrow \mathbf{o_{Node}}$; it determines that the annotation on this edge is **peer** while it is in fact **owned**. Object $\mathbf{o_{Node}}$ is exposed outside of $\mathbf{o_{LinkedList}}$, to the iterator $\mathbf{o_{Iter}}$. The exposure to $\mathbf{o_{Iter}}$ however is only observational; edge $\mathbf{o_{LinkedList}} \rightarrow \mathbf{o_{Node}}$ represents unique modification because $\mathbf{o_{LinkedList}}$ is the only object that can modify $\mathbf{o_{Node}}$.

Our analysis is overly conservative when computing *uniqueMod*: it requires that an edge $o_i \rightarrow o_j$ is a *create* edge, but not an *in* or an *out* edge, in order to have $uniqueMod(o_i \rightarrow o_j)$ return *true*. In the above example, $\mathbf{o_{LinkedList}} \rightarrow \mathbf{o_{Node}}$ is a *create* edge ($\mathbf{o_{Node}}$ is created by $\mathbf{o_{LinkedList}}$), and also an *in* edge

($\mathbf{o_{Node}}$ is passed as an argument back to $\mathbf{o_{LinkedList}}$ by the iterator by calling `list.set(pos,e)`). Thus, *uniqueMod*($\mathbf{o_{LinkedList}} \rightarrow \mathbf{o_{Node}}$) returns *false*, and the edge is assigned annotation **peer** (at line 5 in Figure 13). Essentially, the analysis is unable to determine if the node is passed to the same linked list, or it is passed to a different linked list, represented by the same abstract object.

8 Related Work

While there are too many variants of ownership types to enumerate (for some examples [27, 7, 4, 6, 5, 17, 10]), they all share common characterstics. They restrict the valid patterns of references in the heap to guarantee some abstract property. The restrictions are usually specified by the programmer as annotations in the source code.

Static Ownership Inference. Somewhat surprisingly, ownership inference has received less attention. Work on static inference of ownership-like properties includes [13, 4, 8, 14, 28, 21, 26]. Aldrich et al. [4] present an ownership type system and briefly discuss an analysis that infers annotations. At a high-level, this analysis has similar goals to ours. However, the analysis is conceptually different from ours. Furthermore, the analysis has not been developed and evaluated. Ma and Foster [21] infer uniqueness-like and ownership-like properties in Java programs. They report that field ownership is infrequent. Our results suggest that this is not case, but this is likely due to the differences in the inferred ownership. They capture exclusive ownership rather than owner-as-dominator ownership. That is, if the contents of a field are passed temporarily to an object, the field is counted as non-owned even if it remains in the dominance boundary of the enclosing object; in contrast, our analysis handles this case more precisely. The papers on Universe types inference [26, 12] are likely more expensive than ours as they are based on a SAT-solver while our analysis is low polynomial.

This paper significantly extends our previous work [19, 22]. The owner-as-dominator inference computes the dominance boundary of o_i, which is a substantial extension of [19]. The owner-as-modifyer inference substantially improves on [22]. Furthermore, this paper focuses on the comparative evaluation of the two ownership protocols.

Shape analysis [32, 16], like our analysis, reasons statically about the structure of the heap. It is typically flow-sensitive and it reasons about more complex properties than ownership; therefore, but is generally more expensive.

Dynamic Ownership Inference. There has been work on analyzing the runtime behavior of object-oriented programs and use heap snapshots to observe ownership-like properties at runtime [3, 24, 11, 31, 12, 29]. Potanin et al. [29] present statistics such as average size of dynamic dominance boundaries, while Mitchell [24] studies more complex connected heap structures. Most notably, the main concern and contribution of these dynamic analyses is the handling of very large run-time object graphs. In contrast, the main concern and contribution of our work, is getting the best out of the relatively small and conservative abstract

object graphs. Dietl and Müller propose a dynamic analysis for inference of Universe types [11] which is the basis of our owner-as-modifier inference. Our analysis has roughly the same structure — it attempts to confine a modification as deep in the dominance hierarchy as possible. However, our analysis is static and therefore is safe and [11] relies on user-provided method purity annotations; our analysis employs targeted and precise purity (i.e., side-effect) analysis.

9 Conclusions

We presented a novel static analysis that infers ownership according to the owner-as-dominator and owner-as-modifier protocols. We implemented the analysis and performed experiements on a set of small-to-large Java programs. The experiments indicate that the analysis is adequately precise and practical.

References

1. Ashes suite collection. http://www.sable.mcgill.ca/software.
2. Dacapo benchmark suite. http://www-ali.cs.umass.edu/ dacapo/gcbm.html.
3. R. Agarwal and S. Stoller. Type inference for parameterized race-free Java. In *VMCAI*, pages 149–160, 2004.
4. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.
5. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.
6. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *PLDI*, pages 324–337, 2003.
7. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
8. D. Clarke, M. Richmond, and J. Noble. Saving the world from bad beans: Deployment time confinement checking. In *OOPSLA*, pages 374–387, 2003.
9. D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Muller, and A. Summers. Universe types for topology and encapsulation. In *FMCO*, 2008.
10. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
11. W. Dietl and P. Müller. Runtime Universe type inference. In *IWACO*, 2007.
12. A. Fuerer. Combining run-time and static Universe Type Inference. Master's thesis, ETH Zurich, 2007.
13. C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, pages 241–253, 2001.
14. D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.
15. Javad. http://www.bearcave.com/software/java/javad/index.html. 2004.
16. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *SAS*, 2004.
17. P. Lam and M. Rinard. A type system and analysis for the automatic extraction and enforcement of design information. In *ECOOP*, pages 275–302, 2003.
18. O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.

19. Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.
20. Y. Liu and A. Milanova. Practical static analysis for inference of security-related program properties. In *ICPC*, 2009.
21. K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.
22. A. Milanova. Static inference of Universe types. In *IWACO*, 2008.
23. A. Milanova and Y. Liu. Practical static ownership inference. Technical Report RPI/DCS-09-04, Rensselaer Polytechnic Institute, Dec. 2009.
24. N. Mitchell. The runtime structure of object ownership. In *ECOOP*, pages 74–98, 2006.
25. P. Müller. Personal communication, Sept. 2008.
26. M. Niklaus. Static Universe Type Inference using a SAT-solver. Master's thesis, ETH Zurich, 2006.
27. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.
28. A. Poetzsch-Heffter, K. Geilmann, and J. Schafer. Infering ownerhsip types for encapsulated object-oriented program components. In T. Reps, M. Sagiv, and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm on the occasion of His 60th Birthday*, volume 4444 of *LNCS*, pages 120–144. Springer, 2007.
29. A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement. *Concurrency - Practice and Experience*, 16(7):671–687, 2004.
30. J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.
31. D. Rayside and L. Mendel. Object ownership profiling: a technique for finding and fixing memory leaks. In *ASE*, pages 194–203, 2007.
32. N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, 2005.
33. A. Rountev, A. Milanova, and B. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.
34. A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in Java software. *IEEE TSE*, 30(6):372–386, 2004.
35. M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *ACM Conference on Programming Language Design and Implementation*, pages 387–400, 2006.
36. 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*, pages 18–34, 2000.

Appendix A

Proof of Lemma 1

We present the proof of Lemma 1. Although the analysis algorithm is simple to state, proving its correctness was surprisingly difficult. Initially, we attempted a proof by induction on the steps of the operational semantics that gives the construction of $Og^r_{P_e}$. However, we found it difficult to reason about $l = r.m()$. For example, let o^r_i be the object referred by r and let o^j_r be the object referred by l after the return. In this case, all objects previously in the dominance boundary of o^r_i, and reachable by o^r_j would no longer be in the dominance boundary of o^r_i. We could not easily show that the analysis would indeed remove all representatives of objects reachable from o_j from the abstract boundary.

As a result, we constructed a somewhat nonstandard proof by contradiction.

Consider a node $o_i \in O$. Sets $Out \subseteq O$ and $In \subseteq O$ are computed by the analysis in Figure 7 — set In contains the nodes in the dominance boundary of o_i and set Out is the subset of all nodes at the border of the dominance boundary, that are seen by the analysis algorithm. Set $InOut$ contains all edges $o_i \rightarrow o_j$ that have been visited on the worklist W.

Let P_e be any execution of P and let the object graph of this execution be $Og^r_{P_e}$. Let $o^r_i \in Og^r_{P_e}$ be any run-time object represented by o_i. For the rest of this section we use the following notational convention: run-time objects are denoted with superscript r and their analysis representatives are denoted using exactly the same o notatation but without the superscript; for example, o^r_1's representative is o_1, o^r_2's representative is o_2, o_k's representative is o_k, etc.

We define set $Out^+ \subseteq O$ as follows:

$$\begin{aligned} Out^+ = \{o_2 \mid o_2 \notin In \wedge \exists o^r_1 \rightarrow o^r_2 \in Og^r_{P_e} \wedge o^r_1 \in createClosure(o^r_i) \wedge o^r_2 \notin createClosure(o^r_i)\} \\ \bigcup \{o_1 \mid o_1 \notin In \wedge \exists o^r_1 \rightarrow o^r_2 \in Og^r_{P_e} \wedge o^r_2 \neq o^r_i \wedge o^r_1 \notin createClosure(o^r_i) \wedge o^r_2 \in createClosure(o^r_i)\} \end{aligned}$$

Informally, set Out^+ contains the representatives of run-time objects that border $createClosure(o^r_i)$ — that is, the representatives of sources of edges that begin outside of $createClosure(o^r_i)$, and end inside $createClosure(o^r_i)$, and the representatives of targets of edges that begin inside of $createClosure(o^r_i)$ and end outside of $createClosure(o^r_i)$. Set Out^+ excludes objects that are in set In.

Another set, set $\overline{Out}$ is defined as follows:

$$\overline{Out} = Out^+ - Out$$

This set contains the objects that are in Out^+, but are not in Out, i.e., objects that are at the border of $createClosure(o^r_i)$ but are never seen by the analysis.

Next, we define the notion of *forbidden edge*. An edge $o^r_1 \rightarrow o^r_2 \in Og^r_{P_e}$ is *forbidden* if it is of one of the following kinds:

(1) $o_1 \in In \wedge o^r_1 \in createClosure(o^r_i) \wedge o_2 \in \overline{Out}$
(2) $o_1 \in Out \cup \overline{Out} \wedge o^r_2 \in In \wedge o^r_2 \in createClosure(o^r_i)$

(3) $o_1 \in In \wedge o_1^r \in createClosure(o_i^r) \wedge o_2 \in In \wedge o_2^r \notin createClosure(o_i^r)$
(4) $o_1 \in In \wedge o_1^r \notin createClosure(o_i^r) \wedge o_2 \in In \wedge o_2^r \in createClosure(o_i^r)$
(5) $o_1 \in In \wedge o_1^r \in createClosure(o_i^r) \wedge o_2 \in Out \wedge o_1 \rightarrow o_2 \notin InOut$

We show that there is no forbidden edge in $Og^r_{P_e}$. The proof assumes that there exists a forbidden edge, and considers the first forbidden edge added to $Og^r_{P_e}$; it proceeds to show that if there is such a first forbidden edge, then there must be an earlier forbidden edge, which is a contradiction.

Let $o_1^r \rightarrow o_2^r$ be the first forbidden edge added to $Og^r_{P_e}$. There are five cases, which we enumerate below.

(1) In case (1), $o_1^r \rightarrow o_2^r$ is of kind (1) — that is, we have that $o_1 \in In \wedge o_1^r \in createClosure(o_i^r) \wedge o_2 \in \overline{Out}$. Consider the creation of this edge. It can be created due to arguments, due to object creation, or due to a return; therefore it can be an *in* edge, a *create* edge, or an *out* edge. We consider the three cases.

1.i In case 1.i edge $o_1^r \rightarrow o_2^r$ is an *in* edge. Therefore, there exists o_k^r such that there is a valid edge triple o_k^r, o_1^r, o_2^r where o_2^r is passed as an argument from o_k^r to o_1^r; edges $o_k^r \rightarrow o_1^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, there would be an earlier forbidden edge, namely edge $o_k^r \rightarrow o_2^r$, of kind (1). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o^k \rightarrow o_1^r$, of kind (4).
If $o_k \in Out \cup \overline{Out}$, then there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_1^r$, of kind (2).
Therefore, edge $o_1^r \rightarrow o_2^r$ cannot be an *in* edge.

1.ii In case 1.ii edge $o_1^r \rightarrow o_2^r$ is a *create* edge. Then it is impossible to have $o_2 \in \overline{Out}$ due to the fact that $o_2 \in createClosure(o_i)$ and o_2 would be added to the original In (line 2 of the analysis algorithm), and taken out of the original In only if in Out (lines 3, 8-9, and 14 of the analysis in Figure 7). Thus, o_2 is in In or in Out, but not in $\overline{Out}$.
Therefore, the edge cannot be a *create* edge.

1.iii In case 1.iii edge $o_1^r \rightarrow o_2^r$ is an *out* edge. Therefore, there exists o_k^r such that there is an edge triple o_1^r, o_k^r, o_2^r where o_2^r is passed due to return from o_k^r to o_1^r; edges $o_1^r \rightarrow o_k^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_2^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, there would be an earlier forbidden edge, namely edge $o_k^r \rightarrow o_2^r$, of kind (1). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o_1^r \rightarrow o_k^r$, of kind (3). If o_k is in Out, there are two cases: if $o_1 \rightarrow o_k$ is in $InOut$ then, o_2 would have been visited by the analysis algorithm (lines 13-15), and o_2 would have been added to Out; otherwise, that is, if $o_1 \rightarrow o_k \notin InOut$ we have an earlier forbidden edge, $o_1^r \rightarrow o_k^r$, of kind (5).

If $o_k \in \overline{Out}$, then there would be an earlier forbidden edge, edge $o_1^r \rightarrow o_k^r$, of kind (1).

Therefore, the edge cannot be an *out* edge.

(2) In case (2), $o_1^r \rightarrow o_2^r$ is of kind (2) — that is, we have that $o_1 \in Out \cup \overline{Out} \wedge o_2 \in In \wedge o_2^r \in createClosure(o_i^r)$. Analogously to the previous case, consider the creation of this edge. It can be created due to arguments, due to object creation, or due to a return; therefore it can be an *in* edge, a *create* edge, or an *out* edge. We consider the three cases.

2.i In case 2.i edge $o_1^r \rightarrow o_2^r$ is an *in* edge. Therefore, there exists o_k^r such that there is a valid edge triple o_k^r, o_1^r, o_2^r where o_2^r is passed as an argument from o_k^r to o_1^r; edges $o_k^r \rightarrow o_1^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.

If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$ and $o_1 \in Out$ there are two cases: if $o_k \rightarrow o_1$ is in $InOut$, then o_2 would not be in In — it would have been visited, removed from In and added to Out (lines 13-15); otherwise, if $o_k \rightarrow o_1$ is not in $InOut$, there would be an earlier forbidden edge, namely edge $o_k^r \rightarrow o_1^r$ of kind (5). Continuing with the case when $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, if $o_1 \in \overline{Out}$, then we would have an earlier forbidden edge, namely edge $o_k^r \rightarrow o_1^r$, of kind (1). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o_k^r \rightarrow o_2^r$, of kind (4).

If $o_k \in Out \cup \overline{Out}$, then there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_2^r$, of kind (2).

Therefore, the edge cannot be an *in* edge.

2.ii In case 2.ii edge $o_1^r \rightarrow o_2^r$ is a *create* edge. If $o_k^r \in createClosure(o_i^r)$, then o_1 is in Out, and o_2 cannot be in In (it would have been removed from In at line 8). Otherwise, if $o_k^r \notin createClosure(o_i^r)$, o_2^r cannot be in create closure, because clearly, it has been created by the outside object o_1^r.

Therefore, the edge cannot be a *create* edge.

2.iii In case 2.iii edge $o_1^r \rightarrow o_2^r$ is an *out* edge. Therefore, there exists o_k^r such that there is an edge triple o_1^r, o_k^r, o_2^r where o_2^r is passed due to return from o_k^r to o_1^r; edges $o_1^r \rightarrow o_k^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_2^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.

If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, there would be an earlier forbidden edge, namely edge $o_1^r \rightarrow o_k^r$, of kind (2). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o_k^r \rightarrow o_2^r$, of kind (4).

If $o_k \in Out \cup \overline{Out}$, there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_2^r$, of kind (2).

Therefore, the edge cannot be an *out* edge.

(3) In case (3), $o_1^r \rightarrow o_2^r$ is of kind (3) — that is, we have that $o_1 \in In \wedge o_1^r \in createClosure(o_i^r) \wedge o_2 \in In \wedge o_2^r \notin createClosure(o_i^r)$. Analogously to the previous cases, consider the creation of this edge. It can be created due to

arguments, due to object creation, or due to a return; therefore it can be an *in* edge, a *create* edge, or an *out* edge. We consider the three cases.

3.i In case 3.i edge $o_1^r \rightarrow o_2^r$ is an *in* edge. Therefore, there exists o_k^r such that there is an edge triple o_k^r, o_1^r, o_2^r where o_2^r is passed as an argument from o_k^r to o_1^r; edges $o_k^r \rightarrow o_1^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, then there would be an earlier forbidden edge, namely edge $o_k^r \rightarrow o_2^r$, of kind (3). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o^k \rightarrow o_1^r$, of kind (4).
If $o_k \in Out \cup \overline{Out}$, then there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_1^r$, of kind (2).
Therefore, the edge cannot be an *in* edge.

3.ii In case 3.ii edge $o_1^r \rightarrow o_2^r$ is a *create* edge. Since $o_1^r \in createClosure(o_i^r)$, then o_2^r must be in $createClosure(o_i^r)$ as well by the definition of *createClosure*. Therefore, the edge cannot be a *create* edge.

3.iii In case 3.iii edge $o_1^r \rightarrow o_2^r$ is an *out* edge. Therefore, there exists o_k^r such that there is an edge triple o_1^r, o_k^r, o_2^r where o_2^r is passed due to return from o_k^r to o_1^r; edges $o_1^r \rightarrow o_k^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, there would be an earlier forbidden edge, namely edge $o_k^r \rightarrow o_2^r$, of kind (3). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o_1^r \rightarrow o_k^r$, of kind (3).
If $o_k \in \overline{Out}$, there would be an earlier forbidden edge, edge $o_1^r \rightarrow o_k^r$, of kind (1). If $o_k \in Out$ there are two cases again: if $o_1 \rightarrow o_k \in InOut$, then o_2 would not be in In—it would have been visited and added to Out (lines 13-15); otherwise, if $o_1^r \rightarrow o_k^r$ is not in $InOut$, then we would have an earlier forbidden edge, namely $o_1 \rightarrow o_k^r$, of kind (5).
Therefore, the edge cannot be an *out* edge.

(4) In case (4), $o_1^r \rightarrow o_2^r$ is of kind (4) — that is, we have that $o_1 \in In \wedge o_1^r \notin createClosure(o_i^r) \wedge o_2 \in In \wedge o_2^r \in createClosure(o_i^r)$. Analogously to the previous cases, consider the creation of this edge. It can be created due to arguments, due to object creation, or due to a return; therefore it can be an *in* edge, a *create* edge, or an *out* edge. We consider the three cases.

4.i In case 4.i edge $o_1^r \rightarrow o_2^r$ is an *in* edge. Therefore, there exists o_k^r such that there is an edge triple o_k^r, o_1^r, o_2^r where o_2^r is passed as an argument from o_k^r to o_1^r; edges $o_k^r \rightarrow o_1^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, then there would be an earlier forbidden edge, namely edge $o_k^r \rightarrow o_1^r$, of kind (3). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o^k \rightarrow o_2^r$, of kind (4).

If $o_k \in Out \cup \overline{Out}$, then there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_2^r$, of kind (2).
Therefore, the edge cannot be an *in* edge.

4.ii In case 4.ii edge $o_1^r \rightarrow o_2^r$ is a *create* edge. Since $o_1^r \notin createClosure(o_i^r)$, then o_2^r cannot be in $createClosure(o_i^r)$. Therefore, the edge cannot be a *create* edge.

4.iii In case 4.iii edge $o_1^r \rightarrow o_2^r$ is an *out* edge. Therefore, there exists o_k^r such that there is an edge triple o_1^r, o_k^r, o_2^r where o_2^r is passed due to return from o_k^r to o_1^r; edges $o_1^r \rightarrow o_k^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, there would be an earlier forbidden edge, namely edge $o_1^r \rightarrow o_k^r$, of kind (4). If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o_k^r \rightarrow o_2^r$, of kind (4).
If $o_k \in \overline{Out} \cup Out$, there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_2^r$, of kind (2).
Therefore, the edge cannot be an *out* edge.

(5) In case (5), $o_1^r \rightarrow o_2^r$ is of kind (5) — that is, we have that $o_1 \in In \wedge o_1^r \in createClosure(o_i^r) \wedge o_2 \in Out \wedge o_1 \rightarrow o_2 \notin InOut$. Analogously to the previous cases, consider the creation of this edge. It can be created due to arguments, due to object creation, or due to a return; therefore it can be an *in* edge, a *create* edge, or an *out* edge. We consider the three cases.

5.i In case 5.i edge $o_1^r \rightarrow o_2^r$ is an *in* edge. Therefore, there exists o_k^r such that there is an edge triple o_k^r, o_1^r, o_2^r where o_2^r is passed as an argument from o_k^r to o_1^r; edges $o_k^r \rightarrow o_1^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.
If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, then there are two cases. If edge $o_k \rightarrow o_2 \in InOut$, then edge $o_1 \rightarrow o_2$ would have been visited at lines 16-17) and added to the worklist; otherwise, if $o_k \rightarrow o_2 \notin (InOut)$ means that edge $o_k^r \rightarrow o_2^r$, of kind (5), is an earlier forbidden edge. If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o^k \rightarrow o_2^r$, of kind (4).
If $o_k \in Out \cup \overline{Out}$, then there would be an earlier forbidden edge, edge $o_k^r \rightarrow o_1^r$, of kind (2).
Therefore, the edge cannot be an *in* edge.

5.ii In case 5.ii edge $o_1^r \rightarrow o_2^r$ is a *create* edge. This is impossible because the analysis algorithm detects each such edge (lines 10-12), and adds each such edge to the worklist. Therefore, the edge cannot be a *create* edge.

5.iii In case 5.iii edge $o_1^r \rightarrow o_2^r$ is an *out* edge. Therefore, there exists o_k^r such that there is an edge triple o_1^r, o_k^r, o_2^r where o_2^r is passed due to return from o_k^r to o_1^r; edges $o_1^r \rightarrow o_k^r$ and $o_k^r \rightarrow o_2^r$ must exist *before* edge $o_1^r \rightarrow o_2^r$. Since o_1^r is in $createClosure(o_i^r)$ by the definition of Out^+ and $\overline{Out}$, we have that o_k is in one of In, $\overline{Out}$, or Out.

If $o_k \in In$ and $o_k^r \in createClosure(o_i^r)$, there are two cases. If $o_1 \rightarrow o_k \in InOut$, edge $o_1 \rightarrow o_2$ would have been visited (lines 18-19) and added to the worklist; therefore, it would not be a forbidden edge. Otherwise, if $o_1 \rightarrow o_k \notin InOut$, then that edge would be an earlier forbidden edge, of kind (5).
If $o_k \in In$ and $o_k^r \notin createClosure(o_i^r)$, there would be an earlier forbidden edge as well, edge $o_1^r \rightarrow o_k^r$, of kind (3).
If $o_k \in \overline{Out}$, there would be an earlier forbidden edge, edge $o_1^r \rightarrow o_k^r$, of kind (1). If $o_k \in Out$, there are two cases again. If edge $o_1 \rightarrow o_k$ is in $InOut$, then $o_1 \rightarrow o_2$ would have been added to the worklist (lines 13-15). Otherwise, edge $o_1^r \rightarrow o_k^r$ is an earlier forbidden edge, of kind (5). Therefore, the edge cannot be an *out* edge.

We conclude the proof of the theorem. Suppose now that there exists a path p: $o_i^r \rightarrow o_1^r \rightarrow ...o_{k-1}^r \rightarrow o_k^r$ such that the representative of p is in In, and o_k^r is not dominated by o_i^r. By simple induction, we have that $o_k^r \in createClosure(o_i^r)$. Clearly, o_i^r is in $createClosure(o_i^r)$. Assume that all of $o_1^r...o_{k-1}^r$ are in $crateClosure(o_i^r)$ as well. If o_k^r is not in *createClosure*, we would have that edge $o_{k-1}^r \rightarrow o_k^r$ is a forbidden edge, which is a contradiction since we showed that there are no forbidden edges in $Og_{P_e}^r$. Therefore, o_k^r is in $createClosure(o_i^r)$.

Therefore, there must exist an object, o_k^r that has the following three properties: (1) $o_k^r \in createClosure(o_i^r)$, (2) $o_k \in In$ and (3) o_k^r is not dominated by o_i^r. Let o_k^r be the first object that acquires these properties. Therefore, there must be a predecessor $o_{k'}^r$ of o_k^r (i.e., $o_{k'}^r \rightarrow o_k^r \in Og_{P_e}^r$) such that o_i^r does not dominate $o_{k'}^r$ (i.e., and o_k^r acquires the properties due to its flow to this predecessor). Consider object $o_{k'}^r$. If $o_{k'}^r \in createClosure(o_i^r)$ then $o_{k'}^r$ must be in In (otherwise, if $o_{k'}$ is in Out, edge $o_{k'}^r \rightarrow o_k^r$ would be a forbidden edge of kind (2)); however, this leads to a contradiction because then $o_{k'}^r$ would be the first object to acquire the properties. On the other hand, $o_{k'}^r \notin createClosure(o_i^r)$ is impossible as well, because then $o_{k'}^r \rightarrow o_k^r$ would be a forbidden edge as well.

An easier argument for the above statement may be stated as follows. Suppose that there is o_k^r such that (1) $o_k^r \in createClosure(o_i^r)$, (2) $o_k \in In$, and (3) o_k^r is not dominated by o_i^r. Then there must exist a path from some $o_{k'}^r$ that is not in $createClosure(o_i^r)$ to o_k^r (if no backwards path from o_k^r leads outside of the create closure, we would have that all backwards paths pass through o_i^r and therefore o_i^r dominates o_k^r). WLG we may assume that there is a path $o_{k'}^r \rightarrow o_1^r \rightarrow^* o_k^r$ such that $o_1^r...o_k^r$ are all in *createClosure*. But then there must exist a forbidden edge somewhere along this path, which is impossible.

Proof of Lemma 2

This section presents the proof of Lemma 2. There exists a create path `root` $\rightarrow^*$ $o_{k_n}^r \rightarrow^* ... \rightarrow^* o_{k_1}^r \rightarrow^* o_i^r$ where $o_{k_1}^r$, ...,$o_{k_n}^r$ dominate $o_i^r \rightarrow o_j^r$ ($o_{k_1}^r$ dominates o_i^r and o_j^r, $o_{k_2}^r$ dominates $o_{k_1}^r$ and o_i^r and o_j^r, etc.; `root` dominates all nodes). The analysis algorithm in Figure 11 traverses the representative of this path backwards, until it finds an o_k such that $Boundary(o_k)$ contains all the nodes on

the create path from o_k to o_i (we have that $createPath \subseteq Boundary(o_k)$). Clearly, o_k is added to *minBoundaries* and we have (1) that $o_k \in minBoundaries(o_i \rightarrow o_j)$.

We proceed to prove (2), namely that every path from o_k^r to $o_i^r \rightarrow o_j^r$ is represented in $Boundary(o_k)$. We have that the representative of the create path from o_k^r to o_i^r is in $Boundary(o_k)$ (because the representative of the create path is acyclic, it is guaranteed that *createPath* will contain all the nodes on the path from o_k to o_i; the case when the path is cyclic is a simple extension of the current case). It follows that o_k^r dominates o_i^r (by Lemma 1). Next, we show that every path from o_k^r to o_i^r is represented in $Boundary(o_k)$. Let us assume that there exists a path whose representative is not in $Boundary(o_k)$. Since o_k^r dominates o_i^r, it follows that the path must be in $createClosure(o_k^r)$, and therefore the nodes on this path would be in set *In* or in set *Out*. Since $o_i \in In$, and the representative of the path is not in $Boundary(o_k)$, there must be an edge $o \rightarrow o'$, part of the representative of the path, such that $o \in Out$ and $o' \in In$. However, this would be a a forbidden edge (see Lemma 1). Therefore, it is impossible to have a path from o_k^r to o_i^r which is not represented in $Boundary(o_k)$; this proves condition (2) of the theorem.

Proof of Lemma 3

We now prove Lemma 3. Let P_e be any execution of P and let the object graph of this execution be $Og_{P_e}^r$. Let $o_i^r \rightarrow o_j^r \in Og_{P_e}^r$ be a run-time edge represented by $o_i \rightarrow o_j$ such that o_i^r modifies o_j^r. We show that edge $o_i \rightarrow o_j$ is in *Mod*.

By the definition of o_i^r modifies o_j^r we have

(i) There is a method sequence path $o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow^* o_k^r.m_k()$ (i.e., a sequence of stack frames $o_i^r.m_i$, $o_j^r.m_j$, etc. that leads to $o_k^r.m_k()$.
(ii) $o_k^r.m_k()$ is an update.
(iii) There is an object o^r such that there is a local variable l (including `this`) on a stack frame before $o_j^r.m_j$, which refers to o^r, and $o^r \overset{f}{\rightarrow}^* o_k^r$ (i.e., o_k^r is part of visible state, and the update caused by $o_i.m_i() \rightarrow o_j.m_j()$ is visible after the execution of $o_j^r.m_j()$).

Consider object o_k^r. There are two cases.

In the first case o_k^r is not a transitive field of o_j^r. Therefore, path $o^r \overset{f}{\rightarrow}^* o_k^r$ does not go through o_j^r (or otherwise, o_k^r would have been a transitive field of o_j^r). Consider method sequence $o_k^r.m_{k-1}() \rightarrow o_k^r.m_k()$ (the last sequence in the method sequence path). By Lemma 2, there exists o_x^r such that the representative of every path form o_x^r to $o_{k-1}^r \rightarrow o_k^r$ is in $Boundary(o_x)$, and o_x is in $minBoundaries(o_{k-1} \rightarrow o_k)$.Then o_x^r dominates o_{k-1}^r and o_k^r, and the representative of the path p: $o_x^r \rightarrow^* o_i^r \rightarrow o_j^r \rightarrow^* o_{k-1}^r \rightarrow o_k^r$ is in the boundary of o_x. Since o_x is in *minBoundary*, *propagateMod* will be invoked at line 14 of the algorithm in Figure 12 with arguments $o_k.m_k()$, o_x, and path p will be visited in *propagateMod* resulting in edge $o_i \rightarrow o_j$ being added to *Mod* at line 6 in *propagateMod*. Therefore, in this case, the Lemma 3 holds.

In the second case o_k^r is a transitive field of o_j^r. First we show the following: If the following conditions hold:

(i) There is a method sequence path $o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow^* o_k^r.m_k()$.
(ii) $o_k.m_k()$ is added to $W1$ by the algorithm in Figure 12.
(iii) $o_j^r \overset{f}{\rightarrow}^* o_k^r$.

Then $o_i \rightarrow o_j$ is added to *Mod*.

We prove the above by induction on the length of the field chain from o_j^r to o_k^r. It trivially holds for a path of length 0: (i) we have $o_i^r.m_i() \rightarrow o_j^r.m_j()$, (ii) $o_j.m_j()$ is added to $W1$, and (iii) $o_j^r \overset{f}{\rightarrow}^* o_j^r$. The algorithm will take $o_j.m_j()$ out of $W1$, and at line 6, it will add $o_i \rightarrow o_j$ to *Mod*.

Now assume that if the following conditions hold:

(i) There is a method sequence path $o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow^* o_k^r.m_k()$.
(ii) $o_k.m_k()$ is added to $W1$ by the algorithm in Figure 12.
(iii) $o_j^r \overset{f}{\rightarrow}^* o_k^r$, where the length of the field path is $\leq n$.

Then we have that $o_i \rightarrow o_j$ is added to *Mod*.

We need to show that if the following conditions hold:

(i) There is a method sequence path $o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow^* o_k^r.m_k()$.
(ii) $o_k.m_k()$ is added to $W1$ by the algorithm in Figure 12.
(iii) $o_j^r \overset{f}{\rightarrow}^* o_k^r$, where the length of the field path is $n+1$.

Then we have that $o_i \rightarrow o_j$ is added to *Mod*.

Consider again the last edge in the method sequence path, $o_{k-1}^r.m_{k-1}() \rightarrow o_k^r.m_k()$. By Lemma 2 there exists o_x^r such that o_x is in $minBoundaries(o_{k-1} \rightarrow o_k)$ and all paths from o_x^r to $o_{k-1}^r \rightarrow o_k^r$ are in the boundary of o_x. If o_x^r is before $o_i^r \rightarrow o_j^r$ — that is, we have a path $o_x^r \rightarrow^* o_i^r \rightarrow o_j^r \rightarrow^* o_{k-1}^r \rightarrow o_k^r$, then $o_i \rightarrow o_j$ will be visited in *propagateMod* and will be added to *Mod* at line 6 in *propagateMod*. Otherwise, if o_x^r is after $o_i^r \rightarrow o_j^r$ — that is, we have a path $o_i^r \rightarrow o_j^r \rightarrow^* o_x^r \rightarrow^* o_{k-1}^r \rightarrow o_k^r$, the following happens. Since o_x^r dominates o_k^r we have $o_j^r \overset{f}{\rightarrow}^* o_x^r \overset{f}{\rightarrow}^* o_k^r$; and therefore *propUp* is set to true. *propagateMod* is called with arguments $o_k.m_k()$, o_x, true and $W1$; *propagateMod* adds $o_x.m_x()$ to $W1$ (because the *propUp* flag is set to true), and therefore we have:

(i) There is a method sequence path $o_i^r.m_i() \rightarrow o_j^r.m_j() \rightarrow^* o_x^r.m_x()$.
(ii) $o_x.m_x()$ is added to $W1$ by the algorithm in Figure 12.
(iii) $o_j^r \overset{f}{\rightarrow}^* o_x^r$, where the length of the field path is $\leq n$.

By the inductive hypothesis, $o_i \rightarrow o_j$ is added to *Mod*. This concludes the proof of Lemma 3.

Proof of Theorem 2

Proof of Theorem 2(1) This part of the theorem states that the **owned** and **peer** annotations are assigned in such a way that they induce an ownership tree. There are two ways that the assignment can violate the tree properties: (1) it allows an object to have two owners (two parents in the tree), and (2) it allows a cycle.

We prove that neither (1) or (2) is possible.

For (1), suppose that there is an execution P_e of program P such that an object o^r in this execution is forced to have more than one owners (i.e., more than one parents in the ownership tree). By the construction of $\mathcal{M}_{P_e}$ this can happen only if there is a path p_1: $o^r_x \overset{\textbf{owned}}{\rightarrow} o^r_1 \overset{\textbf{peer}}{\rightarrow}{}^* \; o^r \in Og^r_{P_e}$, and there is another path, p_2: $o^r_y \overset{\textbf{owned}}{\rightarrow} o^r_2 \overset{\textbf{peer}}{\rightarrow}{}^* \; o^r \in Og^r_{P_e}$. Where $o^r_x \neq o^r_y$. The first path, p_1, forces o^r_x to be an owner (i.e. parent) of o^r (it results in adding an edge $o^r_x \rightarrow o^r$ to $\mathcal{M}_{P_e}$), and the second path, p_2, forces o^r_y to be an owner (i.e. parent) of o^r (it results in adding an edge $o^r_y \rightarrow o^r$ to $\mathcal{M}_{P_e}$).

We show that an assignment that permits p_1 and p_2 is impossible.

Suppose that p_1 and p_2 do exist, and consider path p_1. We will show that the representative of p_1 must be in the boundary of its source o_x. Consider edge $o^x_r \rightarrow o^r_1$ whose representative has received annotation **owned**. There are two cases when **owned** is assigned: (i) when $o_x \rightarrow o_1 \in \mathit{Boundary}(o_x)$, and (ii) when $o_x \rightarrow o_1 \notin \mathit{Boundary}(o_x)$ and $\mathit{uniqueMod}(o_x \rightarrow o_1)$ is true. In case (i), we have that $o_x \rightarrow o_1 \in \mathit{Boundary}(o_x)$. Therefore when the **peer** path following o^r_1 is of length 0, we have that $p_1 \in \mathit{Boundary}(o_x)$; when the **peer** path following o^r_1 in p_1 is of length 1 or more, we have $p_1 \in \mathit{Boundary}$ as well (otherwise, *checkConflict* would have changed the annotation of $o_x \rightarrow o_1$ to **peer** at lines 1-2). Therefore, we have that in case (i), the representative of p_1 is in $\mathit{Boundary}(o_x)$.

In case (ii) the **owned** annotation is due to *uniqueMod*. In this case, we are interested in the case when the **peer** path has length 0, that is, p_1 is $o^r_x \rightarrow o^r$ (the case when the **peer** path is of length 1 or more is impossible, because we have that p_1 is not in $\mathit{Boundary}(o_x)$ and the **owned** annotation would have been changed to **peer** at lines 1-2 in *checkConflict*). Again, consider p_1: $o^r_x \rightarrow o^r$ and $\mathit{uniqueMod}(o_x \rightarrow o)$ is true. However, because $o_x \rightarrow o$ is a unique modification, and we cannot have a p_2 which leads into o^r as well (the fact that $o_x \rightarrow o$ is not an *in* or an *out* edge rules out the case when the object that modifies o^r in p_2 would be represented by o_x as well).

Therefore, we have that the representative of p_1 is in the boundary of its source o_x. Analogously, the representative of p_2 is in boundary of its source o_y.

Therefore, we must have that o^r_y is on the path p_1 (since o^r_y dominates o^r). But then we would have that there is a **peer** path from o^r_y, namely $o^r_y \overset{\textbf{peer}}{\rightarrow}{}^* \; o^r$ and a **owned** path from o^r_y, namely $o^r_y \overset{\textbf{owned}}{\rightarrow} o^r_2 \overset{\textbf{peer}}{\rightarrow}{}^* \; o^r$. But then, *checkConflict* would have changed the annotation of $o_y \rightarrow o_2$ to **peer** at lines 3-4, which leads to a contradiction.

For (2), suppose that there is a cycle $o_1^r \overset{\textbf{owned}}{\rightarrow} o_2^r \overset{\textbf{peer}}{\rightarrow}{}^* o_3^r \overset{\textbf{owned}}{\rightarrow} o_4^r \ldots o_{k-1}^r \overset{\textbf{owned}}{\rightarrow} o_k^r \overset{\textbf{peer}}{\rightarrow}{}^* o_1^r \in Og_{P_e}^r$. Here o_1^r is the owner of peers o_2^r and o_3^r, then o_3^r is the owner of o_4^r, etc. We have that o_1^r is a transitive owner of o_{k-1}^r, which is the owner of o_k^r and o_1^r, which creates a cycle in the ownership tree.

We show that a type assignment that would permit this cycle is impossible. Let o^r be the first object on the cycle that has been created, and let its creating object be o_x^r. Thus, there is a create edge $o_x^r \rightarrow o^r$ and o_x^r is not on the cycle (or otherwise it would have been the first created object). Now, let edge $o_1^r \rightarrow o_2^r$ be the first edge on the cycle annotated **owned** and reachable backwards from o^r — that is, we have $o_1^r \overset{\textbf{owned}}{\rightarrow} o_2^r \overset{\textbf{peer}}{\rightarrow}{}^* o^r$. Consider two cases. In case (i) the peer path from o_2^r to o^r is of length 1 or more. Then since o^r is accessible through o_x^r, we have that o^r is not dominated by o_1^r, and therefore the representative of path $o_1^r \rightarrow o_2^r \rightarrow^* o^r$ is not in $Boundary(o_i)$. In this case, the annotation of $o_1 \rightarrow o_2$ would have been turned to **peer** by lines 1-2 in *checkConflict*, and therefore the cycle is impossible. In case (ii) the peer path is of length 0, that is we have $o_x^r \rightarrow o^r$ and we have $o_1^r \overset{\textbf{owned}}{\rightarrow} o^r$, where the latter edge is on the cycle. Consider the **owned** annotation of edge $o_1 \rightarrow o$. Clearly, the **owned** annotation cannot be due to the fact that the edge is in the boundary of o_1 because o^r is not dominated by o_1^r. Therefore, it must be due to *uniqueMod*. However, *uniqueMod* requires that $o_1 \rightarrow o$ is not an *in* or an *out* edges, which is impossible. Clearly, $o_1^r \rightarrow o^r$ must be due to argument (i.e., an *in* edge), or due to a return (an *out* edge) since o^r is created by o_x^r; therefore $o_1 \rightarrow o$ will be an *in* or an *out* edge and $uniqueMod(o_1 \rightarrow o)$ would have been false! Therefore, a **owned** assignment due to *uniqueMod* is impossible as well. This leads to a contradiction and proves that such a cycle cannot exist.

We conclude that the **owned** and **peer** annotations induce a tree.

Sketch of Proof of Theorem 2(2) The above proof shows that every object has exactly one owner in $\mathcal{M}_{P_e}$. Now, we need to show, that for every edge $o_i^r \rightarrow o_j^r$ such that o_i^r modifies o_j^r, then either o_i^r is the owner of o_j^r in $\mathcal{M}_{P_e}$, or o_i^r and o_j^r are peers.

By Lemma 3 we have that $o_i \rightarrow o_j$ is in *Mod*, and therefore, it will be annotated as **owned** or **peer**. If it is annotated as **owned**, then by the construction of $\mathcal{M}_{P_e}$ and uniqueness of the owner, we have that o_i^r is the owner of o_j^r (i.e., o_i^r is the parent of o_j^r in $\mathcal{M}_{P_e}$).

If it is annotated as **peer**, then following case may arise. There exists a path p_1: $o_x^r \overset{\textbf{owned}}{\rightarrow} \ldots \overset{\textbf{peer}}{\rightarrow} o_j^r$, which forces o_x^r to be the parent (i.e., owner) of o_j^r. And also, all incoming paths into o_i^r are of kind p_2: $\texttt{root} \overset{\textbf{any}}{\rightarrow}{}^* o_z^r \overset{\textbf{peer}}{\rightarrow}{}^* o_i^r$; this would force o_i^r to be a child of `root`, and not of o_x^r which means that o_i^r and o_j^r are not peers.

We show that the case described above is impossible. As in Theorem 2(1) the representative of p_1 must be in the boundary of o_x. Therefore o_x^r dominates

o_i^r and o_j^r. Also, for simplicity, assume that o_x^r is the immediate dominator of o_i^r and o_j^r. Therefore, we have a path $o_x^r \rightarrow^+ o_z^r \rightarrow^+ o_i^r$. One can show however, that since $o_i^r \rightarrow o_j^r$ is a modification edge, the representatives of all edges on the path $o_x^r \rightarrow^+ o_z^r \rightarrow^+ o_i^r$ would have been added to *Mod* and therefore could not have been annotated **any**.