Ownership and Immutability Inference for UML-based Object Access Control

Yin Liu
Department of Computer Science
Rensselaer Polytechnic Institute
liuy@cs.rpi.edu

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Abstract

We propose a mechanism for object access control which is based on the UML. Specifically, we propose use of ownership *and* immutability *constraints on UML associations and verification of these constraints through reverse engineering. These constraints inherently support software design principles, and impose requirements on the implementation that may help prevent serious program flaws.*

We propose implementation-level models for ownership and immutability that capture well the meaning of these concepts in design, and we develop novel static ownership and immutability inference analyses. We perform an empirical investigation on several small-to-large Java programs. The results indicate that the inference analyses are precise and practical. Therefore, the analyses can be integrated in reverse engineering tools and can help support effective reasoning about software quality and security.

1 Introduction

Unexpected object access can seriously compromise software quality and software security. For example, in Java 1.1 the security function `Class.getSigners` mistakenly returned a reference to an internal array; untrusted clients could modify this array and compromise the security of the system. Current languages such as Java do not provide effective mechanisms for preventing unexpected object access. Therefore, it is important to develop such mechanisms and advance their usage in software practice.

This paper proposes use of access-control constraints on Unified Modeling Language (UML) class diagrams, and verification of these constraints through reverse engineering. UML class diagrams describe the architecture of the program in terms of classes and *associations* that model interclass relationships; they are informative models, widely used in software engineering practice.

Specifically, we propose the use of *ownership* and *immutability* constraints on UML associations. An association from class A to class B marked as `owned` at design level, states a requirement for ownership and no *representation exposure* at implementation level: an A object must control the B objects it references through this association. An association from class A to class B marked as `read-only` states a requirement for immutability at the implementation level: an A object cannot modify the heap structure rooted at the B object it references through this association.

Ownership and immutability constraints on UML associations inherently support software design principles such as "Low Coupling" and "Information Expert" [13]. Most importantly, the constraints force reasoning about object access control at design level and impose requirements on the implementation. These requirements can be continually verified through reverse engineering which may prevent serious program flaws such as the `Signers` security bug.

The goals of this work are (i) to define implementation-level ownership and immutability models that capture the meaning of these concepts in design, and (ii) to develop practical and precise analyses that infer ownership and immutability in accordance with these models. The definition of implementation-level ownership is based on *owners-as-dominators* [7, 21]—that is, all access paths to an owned object should pass through its owner. The definition of immutability requires that an enclosing object have read-only access to an enclosed immutable object—that is, the methods invoked on the enclosing object cannot change (directly, or through callees) the heap structure rooted at the enclosed immutable object.

We propose two novel static analyses for Java, one for ownership inference and one for immutability inference; the analyses work directly on Java code and do not require annotations by the programmer. Consider a reverse engineered association from class A to class B. If the ownership inference determines that all A objects own the corresponding B objects referenced through this association, the analysis marks the association as `owned`. Analogously, if the immutability inference determines that all A objects do not modify the corresponding B objects, the analysis marks the association as `read-only`. It is important to note that the analyses can work on complete programs (i.e., whole pro-

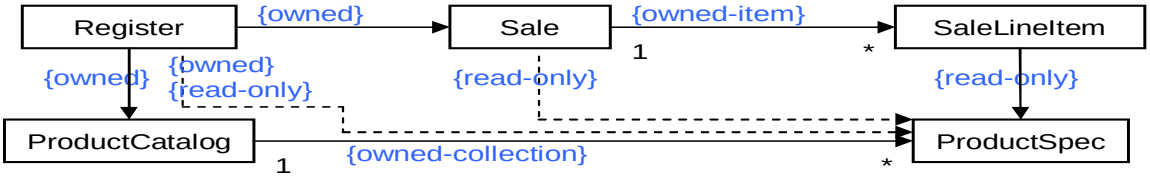

Figure 1. Ownership and immutability constraints on UML associations.

grams) as well as on incomplete programs (i.e., software components). This paper focuses on complete programs in order to (i) clearly present the underlying algorithms, and (ii) emphasize analysis scalability on large programs.

Surprisingly, while applying our analysis on the code from a popular textbook [13], we discovered a bug in this code (this is explained in detail in Section 4.3).

Furthermore, we performed an empirical study on several small to relatively large Java benchmarks. In our experiments, on average 28% of the reverse-engineered associations were determined to be `owned`, and 27% were determined to be `read-only`. We present a precision evaluation which indicates that the analyses achieve adequate precision—the ownership inference almost never misses an `owned` association and the immutability inference rarely misses a `read-only` association. The ownership and immutability analyses are practical, running in less than 7 minutes on all but one benchmark. The experience indicates that (i) the models capture well the meaning of ownership and immutability in design, and the analyses produce useful results and (ii) the analyses are precise and practical. Therefore, the analyses can be incorporated in software tools and can effectively support verification of ownership and immutability; this will lead to high quality, secure, understandable and maintainable software systems.

This work has the following contributions:

- We propose a new mechanism for object access control. It is based on the UML and light-weight verification of properties related to software quality and software security.
- We develop implementation-level models for ownership and immutability that capture well the meaning of these concepts in design.
- We develop novel static ownership and immutability inference analyses.
- We present an empirical study on small to relatively large Java programs. It demonstrates that the analyses are adequately precise and practical.

2 Motivating Example

This section motivates the idea of UML-based object access control. Consider the UML class diagram in Figure 1. It illustrates the design of a supermarket Point-of-Sale system and is taken directly from a popular textbook on software design and the UML [13]. The solid lines represent permanent associations (implemented through instance fields), and the dashed lines represent temporary dependencies (typically implemented through local variables). We have added ownership and immutability constraints based on the description in the textbook—these constraints formalize the design principles emphasized in the textbook.

A `Register` object, an abstraction for the cash register, controls the sale logic. It creates a `ProductCatalog` object that stores the specifications of all products (i.e., the `ProductSpec` objects). The `Register` object creates a `Sale` object, initiates the sale, passes information about sale items to the `Sale` object and completes the sale. When a new sale item is processed, the `Register` fetches the corresponding `ProductSpec` object from the catalog, and passes that object to the `Sale` object. The `Sale` object creates a new `SaleLineItem` object for each sale item and passes the `ProductSpec` object to it.

The association from `Register` to `Sale` is marked as `owned`. Thus, the `Register` *owns* each `Sale` object it refers through this association—intuitively, the `Register` may create a `Sale` object, pass it to other parts of its representation, but cannot leak the `Sale` object to outside parts (e.g., objects that are part of the User Interface (UI) of the system). Furthermore, the association between `SaleLineItem` and `ProductSpec` is marked as `read-only`. Thus, the `SaleLineItem` object cannot modify the `ProductSpec` object it refers to. Note that for one-to-many associations (e.g., `ProductCatalog` to `ProductSpec`) one can specify constraints on the collection and on the items. For example, a ProductCatalog owns the collection that stores `ProductSpec`s, but does not own the `ProductSpec` items stored in this collection.

The ownership and immutability constraints inherently support reasoning about software design principles such as ”Low Coupling”, ”Information Expert”, etc. [13]. The constraint that `Register` owns the `Sale` objects forbids coupling from UI classes to `Sale` which helps achieve low coupling and separation of the UI layer from the domain layer. The constraint that `SaleLineItem` has read-only access to `ProductSpec` forbids `SaleLineItem`s from modifying `ProductSpec`s; in fact, the `ProductCatalog` is the ”information expert” and the only object that can ini-

tialize and update product information. Most importantly, the ownership and immutability constraints impose requirements on the implementation. These requirements can be continually verified in a light-weight manner through reverse engineering of the UML class diagram.

Surprisingly, when we applied our analysis on the Java code from [13, Chapter 20] that corresponds to this diagram, the association between `SaleLineItem` and `ProductSpec` was reported as non-`read-only`. A brief examination of the code revealed a problem that could be serious—the `SaleLineItem` object mistakenly modified the *price* field of the `ProductSpec` object. As a result, subsequent sales fetched `ProductSpec`s with wrong prices and computed incorrect sale totals.

In summary, verifying and enforcing ownership and immutability constraints will lead to higher quality, more secure, understandable and maintainable software systems.

3 Problem Statement

Conventionally, software tools reverse engineer UML associations by examining instance fields of reference type in the code (e.g., a field f of type B in class A is reverse engineered into an association from A to B labeled with f).[1] The ownership inference problem is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$, o owns o'. Similarly, the immutability inference problem is to find the fields f such that for each run-time edge $o \xrightarrow{f} o'$, o does not mutate o'. It remains to give suitable definitions of implementation-level ownership and immutability.

3.1 Ownership Model

The ownership model is based on the notion of *owners-as-dominators* [7, 6, 21]. In this model each program execution is represented by an *object graph* that shows access relationships between run-time objects. There is an edge $o \rightarrow o'$ if at some point of the program execution one of the following is true.

- Reference instance field f in o refers to o'.
- Object o is an array object with element o'.
- An instance method invoked on receiver o has local variable r that refers to o', or a static method called from an instance method invoked on o, has a local variable r that refers to o'.[2]

We say that o *owns* o' if and only if o is the immediate dominator of o' in the object graph.[3] Consider the object

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

Figure 2. Simplified vector and its iterator.

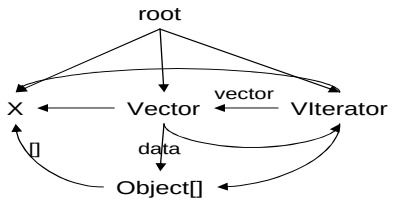

Figure 3. Object graph for Figure 2.

graph in Figure 3 which represents the execution of `main` in Figure 2. Node `root` represents the start of program execution. The other nodes correspond to the objects created at the appropriate allocation sites in Figure 2. `Vector` does not own `Object[]` because during the execution of `next` there is a temporary access from `VIterator` to `Object[]`. `Vector` would own `Object[]` if `next` was never executed (i.e., line 15 is removed from `main`).

3.2 Immutability Model

Let e be an execution of a method m on receiver object o. e modifies an object o' if it triggers a change in the object structure rooted at o'—that is, e leads to a statement $p.f = q$ which modifies an object o'' reachable from o' (i.e., o'' is o' itself or o'' is reachable on a path of field edges). For example, the execution of `add` with receiver `Vector` (line 13 in Figure 2) modifies `Object[]`. We say that o *has*

[1] The rest of the paper focuses on permanent associations (implemented with instance fields). Although our models and analyses are general and can handle temporary dependencies, we omit their discussion for clarity.

[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 $r_1 = r.m(); r_1.n()$).

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

read-only access to o' if no execution of a method m on receiver o modifies o'. Thus, in the above example `Vector` does not have read-only access to `Object[]`.

The model does not treat constructor invocations and the corresponding initialization statements `this.f=q` as modifications of the newly constructed object. This is done to capture the intuitive meaning of immutability in the context of class diagrams.

4 Ownership and Immutability Analyses

The ownership and immutability analyses can be applied on complete programs, as well as on incomplete programs (i.e., components); intuitively, the whole-program analysis can be adapted to work on incomplete programs by utilizing a technique called *fragment analysis* [25, 16]. We present the analyses in the whole-program setting in order to (i) emphasize the underlying algorithms, and (ii) demonstrate scalability on real, large-size Java benchmarks.

4.1 Points-to Analysis

The ownership and immutability analyses are built as independent clients of a *points-to analysis*. Points-to analysis determines the set of objects that a given reference variable or a reference object field may refer to. There is a large body of work on points-to analysis. For the purposes of ownership and immutability inference we consider the well-known Andersen-style flow- and context-insensitive points-to analysis for Java [24, 14].

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

- Let $r \in R$ and $h \in O$. An edge (r, h) in the points-to graph means that at run time r may refer to some object that is represented by h.
- Let $f \in F$ be a reference instance field in objects represented by some $h \in O$. An edge $(h.f, h_2)$ means that at run time field f of some object represented by h may refer to some object represented by h_2.
- Let h represent array objects. An edge $(h[], h_2)$ shows that at run time some array represented by h may contain an element represented by h_2.

For the rest of the paper we use notation o to refer to run-time objects (e.g., o, o', o_i, etc.); we use notation h to refer to analysis names that abstract the run-time objects (e.g., h, h', h_i, etc.).

4.2 Ownership Client

The output of the points-to analysis is needed to construct the *approximate object graph* Ag which approximates all possible run-time object graphs. Subsequently, Ag is used for ownership inference.

input *Stmt*: set of statements *Pt*: $R \cup O \rightarrow \mathcal{P}(O)$
output $Ag : O \rightarrow \mathcal{P}(O)$
[1] **foreach** statement s in method m
  $s_i : l = new\ C(...)$
[2] add $\{c \rightarrow h_i \mid c \in \mathcal{C}_m\}$ to Ag
  //**creation** flow into the receiver of m
[3] **foreach** statement s in method m
  $s : l = r.n(...)$ s.t. $r \neq$ `this`,
  $s : l = r.f$ s.t. $r \neq$ `this`
[4] add $\{c \rightarrow h_j \mid c \in \mathcal{C}_m \wedge (l, h_j) \in Pt\}$ to Ag
  //**outflow** from a callee into the receiver of m
[5] **foreach** statement s in method m
  $s : l = new\ C(r)$,
  $s : l.n(r)$ s.t. $l \neq$ `this`,
  $s : l.f = r$ s.t. $l \neq$ `this`
[6] add $\{h_i \rightarrow h_j \mid (l, h_i) \in Pt \wedge (r, h_j) \in Pt\}$ to Ag
  //**inflow** into the receiver of the callee from m
[7] label with f each $h_i \rightarrow h_j \in Ag$ s.t. $(h_i.f, h_j) \in Pt$

Figure 4. Construction of Ag. $\mathcal{P}(X)$ denotes the power set of X. Ag is initially empty.

Approximate Object Graph. The nodes in Ag are taken from the set of object names O and the edges represent the access relationships. Figure 4 outlines the construction of Ag given a points-to graph Pt. Intuitively, the algorithm tracks flow of objects from one object to another. Notation $\mathcal{C}_m$ stands for the set of receiver objects of method m. It is computed as follows. If m is an instance method, $\mathcal{C}_m$ equals to the points-to set of the implicit parameter `this` of m. If m is a static method, $\mathcal{C}_m$ includes the points-to sets of all implicit parameters `this` of instance methods n reachable backwards from m on a chain of static calls; if `main` is reachable backwards from m on a chain of static calls, $\mathcal{C}_m$ includes the special node `root`.

Lines 1-2 account for object creation. At object creation sites (i.e., constructor calls) new edges are added to Ag from each receiver of the enclosing method m, to the newly created object. Intuitively, the newly created object becomes accessible to the receiver of m. Lines 3-4 account for flow out from other objects to the receiver of m. For example, at an instance call not through `this` new edges are added from each receiver of m to each returned object. Intuitively, the returned object becomes accessible to the receiver of m. Lines 5-6 account for flow from m into other objects. For example, at an instance call $l.n(r)$, edges are added from each object in the points-to set of l to each object in the points-to set of reference argument r. Intuitively, the object in the points-to set of the actual argument becomes accessible to the receiver of the call. Finally, line 7 labels with

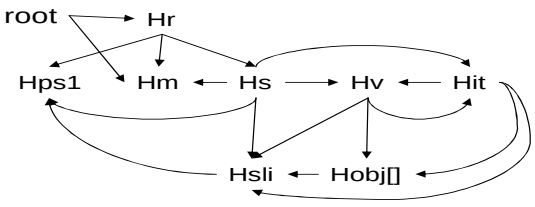

Figure 5. Partial Ag for Section 2.

field identifier f each edge $h_i \rightarrow h_j \in Ag$ for which there is an edge $(h_i.f, h_j) \in Pt$.

Consider the code in Figure 2. In this case, the algorithm in Figure 4 constructs precisely the run-time object graph in Figure 3. Edges `root`→`Vector`, `root`→`X`, `Vector`→`Object[]` and `Vector`→`VIterator` are due to code lines 11, 12, 1 and 4 respectively (lines 1-2 in the algorithm). Edges `Vector`→`X`, `Object[]`→`X` and `VIterator`→`Vector` are due to code lines 13, 2 and 4 respectively (lines 5-6 in the algorithm). Finally, edges `root`→`VIterator`, `VIterator`→`Object[]` and `VIterator`→`X` are due to code line 14, 7 and 10 respectively (lines 3-4 in the algorithm).

The object graph construction and ownership inference need to consider two special cases: (i) static fields and (ii) self-references (i.e., an object references itself through `this` as in `r.m(this)`). For brevity, we do not discuss these cases; our implementation handles them correctly.

Ownership Inference. The ownership inference uses Ag to reason about object ownership. Consider the partial object graph in Figure 5, extracted from the code for Section 2 from [13]. Node `root` represents the special context of `main` and node *Hr* represents the `Register` object (created in `main`). *Hps1* represents `ProductSpec` objects (created in `ProductCatalog`), *Hm* represents `Money` objects (created in `main` to account for payment for a sale), and *Hs* represents `Sale` objects (created in `Register` when initiating a new sale). *Hsli* represents `SaleLineItem` objects (created in `Sale` when processing a new line item) and *Hv* represents the collection needed to store the `SaleLineItem`s. Finally, *Hit* represents iterators over the collection of `SaleLineItem`s (used in `Sale` when calculating the sale total).

The inference analysis (Figure 6) examines an edge $h_i \rightarrow h_j$ in the object graph and attempts to prove that for *each* run-time instance $o_i \rightarrow o_j$ of that edge o_i dominates o_j; intuitively, it reasons about the flow of run-time objects based on the object graph abstraction of this flow. The inference is based on the following intuition: an object o_j can flow from o_i into some o_k only if one of the following is true: (1) o_k has a handle to both o_i and o_j (and hence Ag contains edge triple $h_k \rightarrow h_i$, $h_k \rightarrow h_j$ and $h_i \rightarrow h_j$), or (2) o_i has a handle to both o_k and o_j (and hence Ag contains edge triple $h_i \rightarrow h_k$, $h_i \rightarrow h_j$, $h_k \rightarrow h_j$). In Figure 5

input Ag: $O \rightarrow \mathcal{P}(O)$ $h_i \rightarrow h_j$: $O \times O$
output $Closure$: $O \rightarrow \mathcal{P}(O)$, isClosed: boolean
[0] if $isOutside(h_i \rightarrow h_j)$ **return** false
[1] $Closure$=$\{h_i, h_j\}$, W=$\{h_i\}$
[2] while W not empty
[3] take h_k from W
[4] foreach $h_m \in Tgts(h_k) \cap Closure$
[5] foreach $h_n \in Tgts(h_k) \cap Srcs(h_m)$, $h_n \notin Closure$
[6] if $isOutside(h_i \rightarrow h_n)$ **return** false
[7] if $valid(h_k, h_n, h_m)$ add h_n to $Closure$ and to W
[8] foreach $h_m \in Srcs(h_k) \cap Closure$
[9] foreach $h_n \in Tgts(h_m) \cap Srcs(h_k)$, $h_n \notin Closure$
[10] if $isOutside(h_i \rightarrow h_n)$ **return** false
[11] if $valid(h_m, h_n, h_k)$ add h_n to $Closure$ and to W
[12] **return** true

procedure $valid$
input h_i, h_k, h_j, where $h_i \rightarrow h_j$, $h_i \rightarrow h_k$, $h_k \rightarrow h_j$
output isValid: boolean
[1] if $isIn(h_k \rightarrow h_j)$ and $h_i \in In(h_k \rightarrow h_j)$ **return** true
[2] if $isOut(h_i \rightarrow h_j)$ and $h_k \in Out(h_i \rightarrow h_j)$ **return** true
[3] **return** false

Figure 6. Ownership inference: computing the closure of edge $h_i \rightarrow h_j$. $Tgts(h)$ stands for $\{h' \mid h \rightarrow h' \in Ag\}$ and $Srcs(h)$ stands for $\{h' \mid h' \rightarrow h \in Ag\}$.

edge triple *Hr*→*Hps1*, *Hr*→*Hs*, *Hs*→*Hps1* represents the fact that a `ProductSpec` object flows into a `Sale` object from the `Register` object. Note that if an edge $h_i \rightarrow h_j$ in Ag does not have an h_k such that either (1) h_k has handles to both h_i and h_j, or (2) h_i has handles to both h_k and h_j, we have that each o_i exclusively owns each o_j it refers to (i.e., o_i is the only object that has a reference to o_j). In Figure 5 *Hr*→*Hs* is such an edge; it represents that the `Register` exclusively owns the `Sale` objects it creates.

Consider the algorithm in Figure 6, lines 0 to 12, assuming that $valid$ always returns true; the role of $valid$ will be explained shortly. The algorithm makes use of a predicate $isOutside(h_i \rightarrow h_j)$ (lines 0, 6 and 10)—an edge $h_i \rightarrow h_j$ is an *outside edge* if there exists an h_k such that h_k has handles to both h_i and h_j. Intuitively, $isOutside$ conservatively captures the situation when some o_j flows from (or into) an "outside" object o_k and therefore there may be an access path to o_j that does not pass through o_i. In Figure 5, edge *Hs*→*Hm* is an outside edge. The `Money` object is passed from the `Register` to a `Sale` and the `Sale` object does not own it. If the edge that is examined, namely $h_i \rightarrow h_j$, is not an outside edge, the algorithm proceeds to compute the $Closure$ of $h_i \rightarrow h_j$. The algorithm finds all paths from h_i to h_j. It examines each edge $h_1 \rightarrow h_2$ in $Closure$ and adds

nodes h_3 such that there is a triple $h_1 \rightarrow h_2$, $h_1 \rightarrow h_3$ and $h_3 \rightarrow h_2$. If at some point the algorithm detects a path that originates in an outside edge, it returns false (lines 6 and 10). If the algorithm returns true, it is guaranteed that for each edge $o_i \rightarrow o_j$ represented by $h_i \rightarrow h_j$, all paths from o_i to o_j are internal (i.e., o_i dominates, and thus owns o_j). The correctness argument for this statement is given in [15].

Consider edge *Hr→Hps1* in Figure 5. The algorithm processes h_k equal to *Hr*, *Hs*, *Hsli*, *Hv*, *Hit*, and *Hobj[]*, in this order. It returns true and computes the closure which consists of the above nodes plus *Hps1*. The closure captures all nodes where the `ProductSpec` objects may flow; they are all within the boundary of the `Register`.

If the algorithm in Figure 6 returns true for every edge labeled with f, the ownership analysis concludes that the association through f is `owned`.

Improved Ownership Inference. Note that the analysis, as described above may incur substantial imprecision and cost. This is due to the fact that not all edge triples $h_i \rightarrow h_j$, $h_i \rightarrow h_k$, $h_k \rightarrow h_j$ represent valid flow. For example, suppose that edges $h_i \rightarrow h_j$ and $h_k \rightarrow h_j$ are due to object creation (lines 1-2 in the algorithm in Figure 4) and $h_i \rightarrow h_k$ is due to inflow (lines 5-6). Clearly, edges $h_i \rightarrow h_j$ and $h_k \rightarrow h_j$ refer to two distinct run-time objects that are represented with the same name, h_j. However, the analysis concludes that there might be an o_j that flows from some o_i into some o_k and erroneously infers that edge $h_i \rightarrow h_j$ is not owned. Invalid triples affect not only precision but cost as well. In the above example, when reasoning about edge $h_i \rightarrow h_j$ the analysis needs to reason about edges $h_i \rightarrow h_k$ and $h_k \rightarrow h_j$, which is clearly redundant as these edges are irrelevant to $h_i \rightarrow h_j$.

The edges in the object graph may be characterized as *creation* (due to lines 1-2 in Figure 4), *outflow* (due to lines 3-4) and *inflow* (due to lines 5-6). First, let $h_i \rightarrow h_j$ be an outflow edge. A triple with h_k (i.e., $h_i \rightarrow h_j$, $h_i \rightarrow h_k$ and $h_k \rightarrow h_j$) will be a valid triple only if for some statement $l = r.n()$ that produces outflow edge $h_i \rightarrow h_j$ we have that r point to h_k. Second, let $h_k \rightarrow h_j$ be an inflow edge. A triple with h_i (i.e., $h_k \rightarrow h_j$, $h_i \rightarrow h_k$ and $h_i \rightarrow h_j$) will be a valid triple only if for some statement $l.n(r)$ that produces this edge we have that the `this` pointer of the enclosing method of $l.n(r)$, point to h_i.

The algorithm in Figure 4 is augmented to track valid sources for outflow and inflow edges. Lines 4' and 6' below are added respectively after lines 4 and 6; there is a set Out for each outflow edge and a set In for each inflow edge.

[4'] add $Pt(r)$ to $Out(c \rightarrow h_j)$

[6'] add $Pt(\texttt{this}_m)$ to $In(h_i \rightarrow h_j)$

Subsequently, the ownership inference in Figure 6 uses procedure $valid$ to filter out invalid triples. For example, if $h_k \rightarrow h_j$ is an inflow edge, h_i must appear in $In(h_k \rightarrow h_j)$.

input Pt: $R \rightarrow \mathcal{P}(O)$
output Mod: $m \rightarrow \mathcal{P}(R)$
[0] foreach instance field write s: $p.f = q$
  where p≠`this` OR $EnclMethod(s)$ not a constructor
[1] add p to $Mod(EnclMethod(s))$
[2] while changes occur in Mod
[3] foreach call s: $C.m()$ or $r.m()$
[4] foreach target m' of the call
[5] add $Mod(m')$ to $Mod(EnclMethod(s))$

input $h_i \rightarrow h_j \in O \times O$ $Mod : m \rightarrow \mathcal{P}(R)$
output readOnly: boolean
[6] foreach call s: $r.m(...)$ s.t. $r \neq$ `this` and $h_i \in Pt(r)$
[7] if $TrClosure(h_j) \cap Pt(Mod(target(h_i, m))) \neq \emptyset$
[8] **return** false
[9] **return** true

Figure 7. Immutability inference: computing the read-only status of $h_i \rightarrow h_j$.

4.3 Immutability Client

Immutability Inference. The immutability inference is presented in Figure 7. Lines 0-5 perform standard side-effect analysis [26, 17] which computes a Mod set for each method m. Lines 0-1 process each statement s: $p.f = q$ and store p in the Mod set for the enclosing method of s. Subsequently, lines 2-5 propagate the Mod sets backwards on the call graph. Set $Mod(m)$ contains all reference variables p on the left-hand side of an instance field write, reachable on a call chain from m. The union of the points-to sets of these variables approximates the set of objects that may by modified during the invocation of m.

Finally, lines 6-9 take an edge $h_i \rightarrow h_j \in Ag$ as input and attempt to show that for all run-time edges $o_i \rightarrow o_j$ represented by this edge o_i has read-only access to o_j. The analysis examines each method call $r.m(...)$ on receiver h_i (i.e., $h_i \in Pt(r)$). $TrClosure(h_j)$ denotes the transitive closure of h_j on the points-to graph—that is, the set of all nodes reachable from h_j on a path of field edges. $Pt(S)$ extends the Pt notation over sets as follows: $Pt(S) = \bigcup_{p \in S} Pt(p)$. If for some call the transitive closure of h_j intersects with the set of modified objects of the run-time target of the call (i.e., $target(h_i, m)$), the analysis determines that edge $h_i \rightarrow h_j$ is mutable. If this intersection is always empty, the analysis determines that $h_i \rightarrow h_j$ is immutable.

In the Point-of-Sale code method `getTotal` in `Sale` iterates over the collection of `SaleLineItem`s and calls `getSubtotal` on each `SaleLineItem` object. The body of method `getSubtotal` is as follows:

```
return spec.getPrice().times(quantity);
```

We have that field *spec* of *Hsli* points to *Hps1* and field *price* of *Hps1* points to *Hm1* (*Hm1* represents

```
class A {
 B b1;
 A(B _b1) { b1 = _b1; ... }
 m() { B b2 = new B(); b2.setField(10); }
 }
main() {
 B b1 = new B(); b1.setField(5);
 A a = new A(b1); a.m();
 }
```

Figure 8. Imprecision of immutability inference.

the `Money` object that holds the price of the product). Thus, `getSubtotal` calls method `times` on *Hm1*. The analysis determines that $Mod($`times`$)$ equals {`times.this`}—that is, `times` *changes* the value of the receiver object. Thus, $Mod($`getSubtotal`$)$ equals {`times.this`} and we have that *Hm1* is included in set $Pt(Mod($`getSubtotal`$))$.

There is a call to method `getSubtotal` on receiver *Hsli* in `getTotal` in `Sale`. Consider its effect on edge *Hsli* $\stackrel{spec}{\rightarrow}$ *Hps1*. The intersection of the set of objects modified by `getSubtotal` and the transitive closure of *Hps1* is non-empty; it includes *Hm1*. The analysis determines that a `SaleLineItem` object can modify a `ProductSpec` object which is a violation of the immutability constraint in Figure 1. Further examination revealed that this was a bug in the code in [13]; it caused subsequent sales to fetch wrong product prices and compute incorrect totals.

If the procedure for checking an edge returns true for every edge labeled with f, the immutability analysis concludes that the association through f is `read-only`.

Improved Immutability Inference. The algorithm in Figure 7 may incur substantial imprecision. Consider the code in Figure 8. Field $b1$ is immutable in A. The B object created in `main` and referred by field $b1$ is denoted by name H_{b1}, and the B object created in `m` is denoted by H_{b2}. $Mod($`setField`$)$ equals {`setField.this`}; it is propagated to $Mod($`m`$)$ and we have that $Mod($`m`$)$ equals {`setField.this`} as well. The points-to set of `setField.this` contains both H_{b1} and H_{b2} and the analysis concludes imprecisely that $b1$ is mutable in A.

To improve the analysis we introduce a limited form of context sensitivity. When propagating the Mod set of the callee (line 5), the analysis "maps" modified formal parameters to their corresponding actuals. More precisely, it examines every variable $v \in Mod(m')$. If v is an unassigned formal parameter of m', v is mapped to the corresponding actual at the call and the actual is added to $Mod(EnclMethod(s))$; otherwise v itself is added to $Mod(EnclMethod(s))$.[4] Consider again the code in Figure 8. When propagating $Mod($`setField`$)$ to $Mod($`m`$)$ the analysis maps `setField.this` to the actual argument at the call, namely variable `b2`. As a result $Mod($`m`$)$ equals {`b2`}. Since `b2` points to H_{b2} only, the intersection of the transitive closure of H_{b1} and $\{H_{b2}\}$ is empty and the analysis concludes that $b1$ is immutable in A.

4.4 Complexity

Let N be the size of the program being analyzed—that is, the number of statements, the number of object names and the number of variables is of order N. The complexity of the underlying Andersen-style points-to analysis is $O(N^3)$.

One can see from Figures 4, 6 and 7 that the client analyses are dominated by the ownership inference in Figure 6 which has complexity $O(N^5)$ [15].

5 Empirical Results

The goal of the empirical study is to address three questions. First, do the analyses scale to large Java applications? Second, how often do our analyses discover owned and immutable fields? Third, how *imprecise* the analyses are—that is, how often they miss owned or immutable fields?

The ownership and immutability clients are implemented in Java using the Soot 2.2.3 [30] and Spark [14] frameworks; they are implemented as clients 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 1GB.

Native methods are handled 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 [28].

Our benchmark suite includes several relatively small applications, `soot-c` and `sablecc-j` from the Ashes suite [1], relatively large benchmarks from the DaCapo benchmark suite version beta051009 [2] and the Polyglot Java front-end. The suite is described in Table 1. The number of user classes and user methods fetched by Soot are shown in the first two columns of multicolumn (3); these numbers exclude the standard libraries but include other libraries shipped with the application. The last column shows the number of methods (user and library), determined to be reachable by Spark.

Results. We applied the ownership and immutability inference algorithms on instance fields of reference type in user classes.[5] Table 2 shows the running time of the analysis. The first column shows the running time for Soot and Spark, and the two subsequent columns show the running times for the ownership and immutability clients. Clearly, our analyses scale well, even on applications with close to

[4] Implicit parameter `this` cannot be assigned, and other formal parameters are rarely assigned.

[5] Our experiments exclude fields of type `String` because they do not correspond to associations in the UML class diagram.

(1)Program	(2)Description	(3)Size		
		#User Classes	#User Methods	#Reachable Methods
jdepend-2.9.1	A quality metrics suite for Java	17	225	3962
javad	Classfile decompiler	41	156	3838
JATLite-0.4	Template for writing software agents	45	442	6279
undo	Undo functionality for sysadmins	237	1709	5644
hsqldb-1.8.0	Relational database engine and tools	196	3743	7177
soot-c	Analysis framework for Java	579	2935	6046
sablecc-j	Java parser generator	300	2024	7970
polyglot-1.3.2	Framework for Java language extensions	267	3418	7449
antlr	Parser and lexical analyzer generator	126	1738	5102
bloat	Java bytecode optimizer	289	3232	6402
jython	Python interpreter	163	2892	5606
pmd	Java source code analyzer	718	7057	8653
ps	Postscript interpreter	200	908	5396

Table 1. Information about the Java benchmarks.

Program	Points-to Analysis	Ownership Analysis	Immutability Analysis
jdepend	1m35s	32s	10s
javad	1m33s	27s	3s
JATLite	2m37s	1m29s	35s
undo	3m3s	1m52s	37s
hsqldb	2m57s	2m15s	2m31s
soot	2m23s	1m13s	1m38s
sablecc	3m5s	1m49s	1m30s
polyglot	9m39s	2m44s	3m38s
antlr	2m25s	1m4s	35s
bloat	2m36s	1m57s	3m8s
jython	1m58s	1m21s	3m9s
pmd	4m17s	2m22s	8m16s
ps	2m19s	1m51s	29s

Table 2. Analysis times.

9000 reachable methods. The combined time for ownership and immutability analysis does not exceed 7 minutes on twelve out of thirteen benchmarks; on the most expensive benchmark, pmd, it still runs in under 11 minutes.

The first column of Table 3 shows the number of reference instance fields in user classes. On average, the ownership analysis identified 28% of the fields as owned (column #Owned).Also, on average, the immutability analysis identified 27% of the fields as read-only (column #Immutable).

Analysis Precision. The issue of analysis precision is of crucial importance for software tools. If the ownership analysis is imprecise, it may report that an association is non-owned while in reality it is owned (i.e., the analysis reports that certain representation may be exposed while in fact it is not). Similarly, the immutability analysis may report that an association is non-read-only, while in reality it is. Such information is not useful and may confuse the user. For example, if a user attempts to verify lack of representation exposure, imprecision will mean that potentially large amount of code will have to be examined manually. Therefore, imprecision must be carefully evaluated by analysis designers.

We performed a study of absolute precision [25, 16] on a subset of the fields. Specifically, we considered all fields in the two smallest benchmarks, jdepend and javad, and all fields in the class with the largest number of fields for the four largest benchmarks, hsqldb, polyglot, sablecc and pmd (the size metric that we used was the number of reachable methods, shown in Column (3) of Figure 1). This accounted for a set of 153 instance fields, of which 88 fields were reported non-owned, and 97 fields were reported non-read-only. For this set, we examined manually *each* non-owned field and attempted to prove exposure (i.e., that there is an execution such that an object stored in this field would be exposed outside of its enclosing object). In *all cases* we were able to show exposure—that is, for this set of fields the ownership analysis achieved perfect precision. Similarly, we examined each non-read-only field and attempted to prove mutability (i.e., that there is an execution for which an object stored in this field will be mutated by its enclosing object). In *all but 7 cases* [15] we were able to show mutability—that is, the immutability analysis achieved very good precision as well.

Conclusions. The empirical study leads to the following observations. First, the analyses scale to large programs, analyzing close to ten thousand reachable methods in only several minutes. Second, the ownership and immutability models capture well the meaning of these notions in modeling. Clarke et al. [7] argue that the owners-as-dominators model captures well the notions of ownership and composition in modeling; our study reaffirmed this observation. The immutability model captures relationships intuitively as well; it led us to a bug in the code for our motivating ex-

Program	#Fields	#Owned	#Immutable
jdepend	33	19 (58%)	6 (18%)
javad	40	19 (48%)	40 (100%)
JATLite	142	35 (27%)	13 (9%)
undo	325	73 (22%)	162 (50%)
hsqldb	383	89 (23%)	70 (18%)
soot	340	77 (23%)	57 (17%)
sablecc	304	30 (10%)	40 (13%)
polyglot	435	51 (12%)	92 (21%)
antlr	161	45 (28%)	25 (16%)
bloat	529	81 (15%)	73 (14%)
jython	215	69 (32%)	21 (10%)
pmd	914	318 (35%)	162 (18%)
ps	19	7 (37%)	8 (42%)
Average		28%	27%

Table 3. Ownership and immutability results.

ample. Overall, the analyses produce useful results, easy to interpret in the context of UML class diagrams. Third, the analyses are relatively precise, rarely missing `owned` and `read-only` associations. In summary, the empirical study indicates that the analyses can effectively support model-driven development and reasoning about software quality and security.

6 Related Work

The ownership and immutability inference analyses improve substantially upon our previous work on composition inference [16] and side-effect analysis [17] respectively. The main new analysis idea is to employ an inexpensive context-insensitive points-to analysis and improve precision by limited context sensitivity in the clients. This was crucial for precision and scalability; in fact, the old analysis was not only potentially imprecise, but it did not scale beyond the smallest benchmarks in our suite. Further, the analyses are employed towards a new practical purpose—improving the capabilities of UML tools, which will enhance object access control and thus software security and software quality in practice.

Ownership and immutability type systems. Our work is related to work on ownership type systems [18, 7, 4, 6, 5, 12] and work on immutability type systems [11, 19, 29]. Similarly to our work, these articles emphasize the importance of the concepts of ownership and immutability in software development. Unlike our work they focus on type-theoretic approaches and require type annotations provided by the programmer; generally, these approaches require extensions of the language, compiler and run-time environment and therefore will be difficult to adopt in practice. Our approach uses automatic inference and works directly on Java code; it is based on the universally-known UML and therefore may help advance object access control through ownership and immutability in practice.

Ownership inference. Grothoff et al. [9] present an analysis for Java that infers whether a class is confined within its package. Clarke et al. [8] present a confinement checking tool, related to [9], that warns against certain kinds of violating program statements. These analyses work on the class level while our analyses work on the object level. They are more restrictive than ours (e.g., they do not handle pseudo-generic containers well), and do not address the kind of ownership needed for UML-based object access control.

Heine and Lam [10] present an ownership inference algorithm for the purposes of memory leak detection. Their notion of ownership is substantially different than the notion of owners-as-dominators used in our work.

Aldrich et al. [4] present a type inference analysis in accordance with a type system that they develop. Again, our analysis solves a different problem—ownership inference in accordance with the owners-as-dominators model which is different than the type system in [4] (e.g., the `owned` type in [4] captures exclusive ownership only, although access can be allowed through user-specified alias parameters). The inference analysis in [4] is conceptually different than ours; it infers type annotations at a fine level of granularity (i.e., for each variable and expression) and that appears to hinder scalability. Our analysis, which is based on Soot, and the efficient inclusion-based Andersen-style points-to analysis in Spark, appears to scale better, both in terms of time and memory.

Agarwal and Stoller [3] infer ownership types for race-free Java using dynamic analysis; thus, the inferred types may be unsound. Our analysis is a safe static analysis.

Recent work by Rayside et al. [22] emphasizes the relevance of ownership inference and visualization. The paper however, appears to be preliminary because it does not present empirical results. Our work uses a related ownership model, but a conceptually different inference analysis. It presents a detailed empirical investigation that indicates that the analyses are practical and adequately precise.

Immutability inference. Porat et al. [20] describe an analysis that detects immutable fields. Their analysis is context-insensitive, libraries are not analyzed and the paper discusses only static fields. Our immutability analysis incorporates limited context sensitivity, analyses large libraries and focuses on instance fields.

Ryder et al. [26] present a framework for side-effect analysis for C that is parameterized by points-to analysis. Our inference analysis uses the same general idea for propagation of side-effects. However, we consider underlying context-insensitive points-to analysis combined with limited context sensitivity during propagation; this combination helps achieve scalable analysis.

Rountev [23], and Salcianu and Rinard [27] present analyses that identify side-effect-free methods in Java programs.

In both cases the analyses are applied on relatively small programs (hundreds of reachable methods). Our analysis identifies immutable fields and is applied on substantially larger programs (close to ten thousand reachable methods).

7 Conclusions and Future Work

We presented a new mechanism for object access control which is based on the UML and light-weight verification of ownership and immutability. We presented models for ownership and immutability and corresponding inference analyses. We performed an empirical study that indicated that the analyses were practical and adequately precise.

One limitation of our study is that it is unclear whether the precision results will extend to other data. Another is that it is unclear whether our analysis revealed unintended representation exposure or mutability; we could not perform a study of that because we were not familiar with the benchmarks and their intended design requirements.

In the future, we will perform larger studies of precision as well as studies that relate design requirements with implementation. Furthermore, we plan to integrate the analyses into an open-source UML tool.

8 Acknowledgements

This work was supported by an IBM Eclipse Innovation Award for 2006. We would like to thank Manu Sridharan for answering our questions about his paper, and the ICSE'07 reviewers whose suggestions greatly improved this paper.

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] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.

[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] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, pages 241–253, 2001.

[10] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI*, pages 168–181, 2003.

[11] G. Kniesel and D. Theisen. JAC-access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[12] 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.

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

[14] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.

[15] Y. Liu and A. Milanova. UML-based alias control. Technical Report RPI/DCS-06-10, Rensselaer Polytechnic Institute, Sept. 2006.

[16] A. Milanova. Precise identification of composition relationships for UML class diagrams. In *ASE*, pages 76–85, 2005.

[17] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–12, 2002.

[18] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, pages 158–185, 1998.

[19] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, 2002.

[20] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, 2000.

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

[22] D. Rayside, L. Mendel, R. Seater, and D. Jackson. An analysis and visualization for revealing object sharing. In *Workshop on Eclipse technology eXchange*, pages 11–15, 2005.

[23] A. Rountev. Precise identification of side-effect free methods. In *ICSM*, pages 82–91, 2004.

[24] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

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

[26] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, Mar. 2001.

[27] A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[28] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[29] M. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

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