Object Control Invariants

Derek Rayside and Lucy Mendel
MIT CSAIL 6.883, Prof Ernst

Abstract

Sharing mutable data (via aliasing) is a powerful programming technique. To facilitate sharing, object-oriented programming languages permit the programmer to selectively break encapsulation boundaries. However, sharing data makes programs harder to understand and reason about, because, unlike encapsulated data, shared data cannot be reasoned about in a modular fashion. This paper presents object control invariants: a set of program properties to help programmers understand and reason about shared data.

1 Introduction

Sharing mutable data (via aliasing) is a powerful programming technique. For example, the model-view-controller design pattern [19] captures the essential structure of many graphical user interfaces: many controllers and views share one model object.

To facilitate sharing, object-oriented programming languages permit the programmer to selectively break encapsulation boundaries. Visibility keywords such as `private` suggest that some data should be encapsulated, but do not prevent public methods from returning aliases to that (supposedly) internal data.

However, sharing data makes programs harder to understand and reason about, because, unlike encapsulated data, shared data cannot be reasoned about in a modular fashion.

Encapsulating mutable data facilitates modular reasoning about object invariants. For example, consider a linked list implementation with a sentinel at the head and the invariant that the next field of the elements forms a cycle. If we know that elements are only manipulated by the list that owns them, then we need only examine the code of LinkedList and LinkedListElement in order to verify the invariant. The more data is encapsulated, the easier it is to reason about the program.

Previous work has developed various type-theoretic notions of object ownership to enable the programmer to specify and enforce encapsulation [2, 4, 10, 14, 22, 25, 28, 30]. We borrow the notion of ownership from this work, but consider it from an analytical rather than a type-theoretic perspective. Instead of having the programmer specify the data that is encapsulated (ie, not shared), our tool shows the programmer which data is shared (ie, not encapsulated). The primary contributions of this work are:

- **An analytic approach to characterizing sharing and encapsulation.** Most ownership type systems have the programmer annotate the objects that are encapsulated. Our system, on the other hand, visualizes the objects that are shared. This approach has five key benefits:
  - It focuses the programmer's effort on what should be the exceptional case (sharing), rather than what should be the normal case (encapsulation).
  - It characterizes the sharing that exists.
  - It encourages the programmer to reduce unnecessary sharing, eliminate erroneous sharing, and document essential sharing.
  - It requires less up-front effort from the programmer, because the program does not need to be annotated with ownership type declarations.
  - It characterizes sharing so that the program may be refactored to use shared objects in a more parsimonious and orderly fashion.
- **A lightweight ownership inference algorithm.** Most previous work on ownership inference (eg, [1, 2, 10, 21, 34]) has focused on inferring annotations that will typecheck in some type system. We just infer the ownership structure, without inferring local annotations, which is both faster and easier.

We previously developed an imprecise static analysis (based on RTA [5, 6]) for the purpose of inferring ownership and characterizing shared data [31]. This paper differs from the previous work in two important ways:

- We define *object control invariants* to precisely characterize our notion of ownership and encapsulation.
- We have designed and implemented a brand new dynamic analysis to compute object control invariants.

Contents

2 Object Control Invariants

2.1 Object Ownership Criteria

Programmers view object x as owning object y if y is part of the abstract state of x. Since it is difficult to mechanically determine what is part of the abstract state of an object, a number of different analytical criteria have been proposed in the literature:

- *owner-as-heap-dominator:* all paths in the heap to the owned object must pass through the owner (eg, [30])
- *owner-as-mutator:* whoever mutates an object owns it (eg, [28])
- *owner-as-allocator:* whoever allocates (or de-allocates) an object owns it (eg, [21])

We propose a synthesis of these criteria:

[Defn 1] ***Ownership.*** Object x owns object y if x is an immediate dominator of y in the program's write control graph.

The following definitions clarify what we mean by this:

[Defn 2] ***Control.*** An object is in control when it is the receiver of a method on the call stack.

[Defn 3] ***Control Chain.*** A control chain is the list of objects in control, starting with the entry point.

[Defn 4] ***Write Control Chain.*** A write control chain is a control chain that corresponds to writing a field: ie, where the last object in the chain executes a field store instruction.

[Defn 5] ***Control Chain Link.*** A link in a control chain is a tuple $\langle x, y\rangle$ where x immediately preceedes y in the chain.

[Defn 6] ***Controlled-by.*** Object y is controlled-by object x when x preceeds y in some control chain.

[Defn 7] ***Control Graph.*** A control graph is the union of a set of control chains. Consider a control chain as a set of links; a graph is the union of these link sets from a collection of chains.

[Defn 8] ***Write Control Graph.*** A write control graph is a control graph comprised of only write control chains.

[Defn 9] ***Dominator.*** In a directed graph with a distinguished root node, node x dominates node y if every path from the root to y must pass through x. Each node has a set of dominators.

[Defn 10] ***Immediate Dominator.*** Node x is the immediate dominator of y if it is the 'closest' dominator. More formally, $\mathsf{dom}(y) - \mathsf{dom}(x) = \{x\}$. Every node (in a directed graph with a distinguished root element) has exactly one immediate dominator.

Our definition of ownership is based on object control, rather than on the heap structure. In this view, it doesn't matter how one got an alias to an object, it just matters what one does with that alias.

2.2 Object Control Properties

Each object has one of the following object control properties that characterizes how it is controlled in the program:

[Defn 11] ***Unique Controller.*** Object x uniquely controls object y if if x immediately preceeds y in every control chain.

[Defn 12] ***Non-Linear Controller.*** Object x is a non-linear controller of y if x controls y both before and after some other object z controls y.

[Defn 13] ***Shared Control.*** Object y is said to be shared if it has one or more non-linear controllers.

[Defn 14] ***Linear Controller.*** Object y is said to be controlled linearly if it has neither a unique controller nor a non-linear controller.

All of these definitions can be re-phrased in terms of write control chains, which is our primary interest in exploring our notion of ownership.

2.3 Sets of Objects

Thus far our discussion has been in terms of individual objects. In order speak about the program more generally, we must speak of groups of objects. We define the following ways to group objects:

[Defn 15] ***Field Group.*** The field group of field f is all objects stored into f.

[Defn 16] ***Class Group.*** The class group of class c is all instances of class c.

[Defn 17] ***Site Group.*** The site group of instantiation site s is all objects instantiated at s.

There are a variety of criteria that can be used to identify sets of objects. For example, Kuncak et al. [24] identify sets of objects via reachability and heap referencing relationships.

2.4 Object Group Control Properties

The possible control properties for a set of objects are as follows:

[Defn 18] ***Master Unique Controller.*** A set of objects s is said to have a master unique controller if some object x is the unique controller of every object in the set.

[Defn 19] ***Individual Unique Controllers.*** A set of objects s is said to have individual unique controllers if every object in the set has a unique controller, but not the same controller.

[Defn 20] ***Linear Individual Controller.*** A set of objects s is said to have a linear individual controllers if every object in the set controlled linearly.

[Defn 21] ***Shared.*** A set of objects s is said to be shared if every object in the set is shared.

[Defn 22] ***Inconsistent.*** A set of objects s is said to have inconsistent control if some objects in the group are shared and others are not.

2.5 Messages

We consider an object to be a *message* if one of its fields is read. Almost all stateful objects will be messages. We define the following kinds of messages:

Msg Kind	**Readers**	**Writers**	**Note**
Personal	1	1	reader = writer
Simple	1	1	reader $\neq$ writer
Broadcast	>1	1	
Blackboard	>1	>1	

[Defn 23] ***Personal Message.*** Personal messages are fully encapsulated internal state – ie, 'note to self'. These can be reasoned about locally.

[Defn 24] ***Simple Message.*** Simple messages are what one would expect in a pipe and filter kind of design: the data passes through each component once.

[Defn 25] ***Broadcast Message.*** Broadcast messages have many readers and one writer. Immutable objects will likely be classified as broadcast messages.

[Defn 26] ***Blackboard Message.*** Blackboard messages indicate shared data without any known constraints. The programmer should be made aware of these and they should be well documented.

3 Dynamic Detection of Likely Object Control Invariants

It is possible to produce object control graphs from a static analysis or a dynamic analysis. In this paper we present a dynamic analysis along the lines of Ernst [18]:

1. Capture trace data during program execution.
2. Compute properties for individual objects.
3. Compute properties for sets of objects by looking for the strongest property that holds for every object in the set.

We record the read and write control chains observed during execution, as well as the values written into and read out of fields. We use the latter points-to information for identifying the set of objects stored in each field.

We implemented our instrumenter with AspectJ. In comprises two aspects: one that actually instruments the code, and one that controls the initialization, threading, and I/O behaviour of the instrumenter.

The latter two steps are implemented with the Crocopat relational calculator by Beyer et al. [8]. As discussed in the scalability section below, these scripts are surprisingly not scaling to real world examples, and we are re-implementing them in other languages, which take longer to code but are faster to execute.

Our tool reports results back to the programmer as Eclipse bookmarks.

4 Toy Examples

We discuss the results of running our analysis on four toy example programs. The observer design pattern example is from our previous work [31]. The two ArchJava examples are from the current ArchJava distribution.

For the ArchJava examples we analyzed the Java files produced by the ArchJava compiler, and we also manually translated the ArchJava code into equivalent Java code. The manually translated code has fewer mechanical artefacts in it, as one would expect.

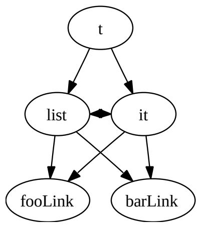
The graphs in this section were mechanically produced by our tool. For some of them we renamed the nodes or removed the distinguished unnamed root node our analysis produces. In the case of cycles we have also manually given `dot` some hints on the layout.

4.1 Linked List

Figure 8 presents a simple program that adds two strings from a list, iterates over the list, prints the strings, and removes them from the list.

Figure 3 lists the write control chains collected from the execution of this program. Figure 1 shows the write control graph formed by merging the chains from Figure 3. Note that there is no Main object in the graph because one is never constructed by the program. There is a Thread object t that is in control and invokes the main method.

Figure 1 Write control graph for linked list example



For this example our analysis infers that:

- all of the objects are owned by the Thread t;
- the two Link objects are controlled linearly (ie, control passes from the list to the iterator)

4.2 Observer Design Pattern

Figure 2 shows the write control graph for the observer example in our previous paper [31]. In this program a subject holds a date that it notifies an observer about changes to. Our previous (static) analysis [31] reported that the subject's representation (a Date) was exposed to the observer. Our current analysis reports that the date is controlled linearly. The difference is that our previous analysis considered the allocator to be the owner, and so reported mutations by non-owners. Our current analysis does not use allocation as a special ownership criteria (allocation is a form of mutation); rather, it reports that the date is owned by the thread object.

Figure 2 Observer design pattern write control graph

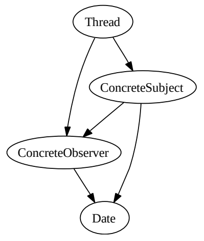


Figure 3 Write control chains from linked list example

Time	Source Code	Control Chain
	instantiating the list	
1	`head = null;`	t → list
	adding ‘foo’ to the list	
2	`data = d;`	t → list → fooLink
3	`next = n;`	t → list → fooLink
	adding ‘bar’ to the list	
4	`data = d;`	t → list → barLink
5	`next = n;`	t → list → barLink
	creating the iterator	
6	`list = l;`	t → list → it
7	`current = l.head;`	t → list → it
	getting the first element	
8	`current = current.next;`	t → it
	removing the first element	
9	...	t → it → fooLink
10	...	t → it → list
	getting the second element	
11	`current = current.next;`	t → it
	removing the second element	
12	...	t → it → barLink
13	...	t → it → list

4.3 ArchJava Pipeline (`Pipeline.archj`)

`Pipeline.archj` is a simple pipe-and-filter style program: data is constructed by a source, read an modified by a filter, and finally sent to a sink. `Pipeline.archj` is annotated to statically guarantee this behaviour. Our analysis discovers this behaviour dynamically: ie, that the data is controlled linearly, and owned by the pipline.

Figure 4 shows the write control graph for the Java source code produced by the ArchJava compiler. Figure 5 shows the write control graph for the Java source code produced by manually translating `Pipline.archj` into Java. Our analysis infers the same results for each variant.

4.4 ArchJava Repository (`Repository.archj`)

Repository.archj has a DataStore that is ‘shared’ by two Modules. Our analysis discovered that the objects stored into DataStore.data are owned by the first module. This initially surprised us, and caused us to read the source code more carefully. We had expected that the RepData objects owned by each Module object would be stored in that field.

Figure 6 shows the write control graph for the Java source code produced by the ArchJava compiler. Figure 7 shows the write control graph for the Java source code produced by manually translating `Repository.archj` into Java.

Figure 4 Write control graph for `Pipeline.java` generated by the ArchJava compiler.

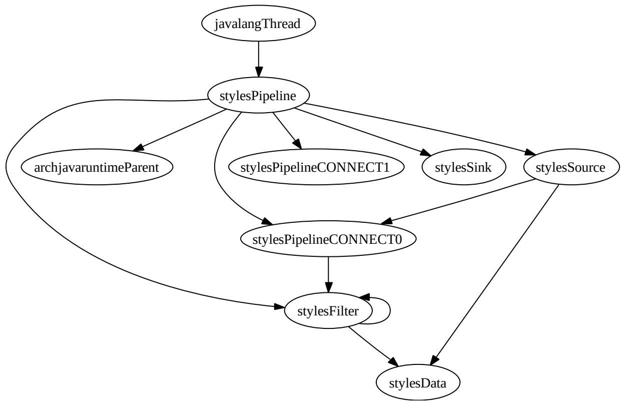


Figure 5 Write control graph for manually generated `Pipeline.java`

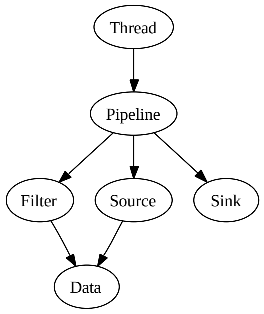


Figure 6 Write control graph for `Repository.java` generated by the ArchJava compiler.

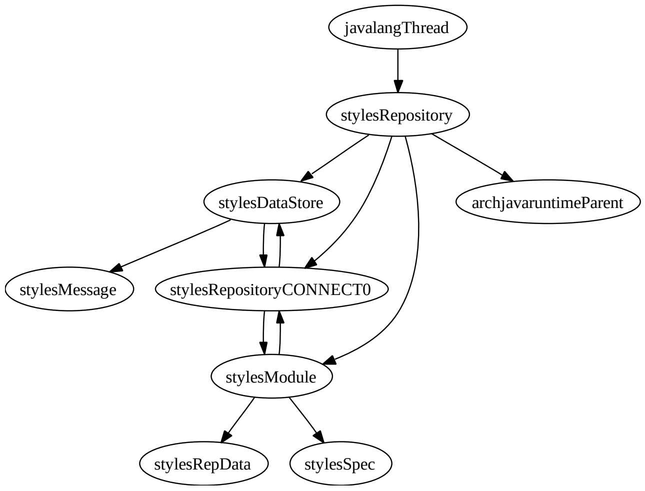


Figure 7 Write control graph for manually generated `Repository.java`

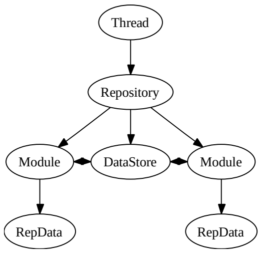


Figure 8 LinkedList example source code

```
public class Main {
    public static void main(String[] args) {
        List l = new LinkedList();
        l.add("foo");
        l.add("bar"):
        for (Iterator i = l.iterator(); i.hasNext(); ) {
            Object x = i.next();
            i.remove();
            System.out.println(x);
        }
    }
}

public class LinkedList {
    private Link head;

    public LinkedList() {
        head = null;
    }

    public add(Object d) {
        ...
        new Link(d);
        ...
    }

    public iterator() {
        return new ListIterator(head);
    }

    private static class Link {
        final Object data;
        Link next;
        Link(Object d, Link n) {
            data = d;
            next = n;
        }
    }

    private static class ListIterator implements Iterator {
        private List list;
        private Link current;
        ListIterator(List l) {
            list = l;
            current = l.head;
        }
        public boolean hasNext() { ... }
        public Object next() {
            Object d = current.data;
            current = current.next;
            return d;
        }
        public void remove() { ... mutate a link ... }
    }
}
```

5 Scalability: TSafe Case Study

Once we were happy that the basic analysis was working, we tried to run it on TSafe [16]. TSafe is a program that reads a feed describing aircraft positions, plots their trajectories, and warns of airplanes that are too close to each other. The TSafe code comprises 227 classes. Typical inputs to TSafe are hundreds of megabytes. We ran TSafe on a small 450kb input file and noticed some scalability issues.

5.1 Reducing Trace Size and Overhead

Our original instrumenter produced a 550mb trace from the 450kb sample input. The second column of the table in Figure 9 characterizes this trace. From these figures we see that reads are approximately three times more frequent than writes, and that the largest parts of the trace are the chains.

Figure 9 Characterizing the trace. Size of relations measured in tuples.

Relation	Original Size	New Size	Change
Root	1	1	—
Thread	6	6	—
New	137,751	129,758	—
Constructor	155,344	149,404	—
Write	273,812	259,471	—
Read	805,179	717,174	—
WriteChain	1,401,755	443,404	31.63%
ReadChain	4,903,825	1,552,408	31.65%
Trace (bytes)	554,019,011	252,282,913	31.66%

Caching a hash of the last chain. It dawned on us that it's very common for a method to read or write multiple fields of an object. Each of these consecutive reads or writes would produce essentially the same control chain, just at a different time. Our analysis is only concerned with the temporal interleavings of control chains, not with the times they actually occur at. So we lose no information by dropping control chains that are identical to the last control chain recorded.

We now only record a chain in the trace if it differs from the last chain recorded for the target object. To accomplish this, we cache an integer hash of the last chain recorded for each object. Since we need to keep this information for *every object created by the analysand*, using the standard Java map implementations would be too much overhead. The GNU Trove library provides an int to int hash map that uses open addressing for collision resolution.

The third column of the table in Figure 9 shows the size of the trace collected with the new cache. The size of the chains, and the trace, is one third the size of the original.

Instrumentation overhead. Our original instrumenter slowed TSafe down from about 1 minute to around 20 minutes. With the cache and improved I/O code, we trimmed this time to about 3 minutes.

Since TSafe is an interactive program it is not possible to take precise timing measurements.

5.2 Object Control Dominators

Even with the reduced trace size and refactored analysis code, our Crocopat [8] script still could not compute dominators for the captured object control graph from TSafe. Our script implemented a conventional worklist algorithm for computing dominators.

Figure 10 characterizes the steps of the dominator algorithm before it runs out of memory. It took two hours of computation on a 2GHz G5, with 1gb of memory allocated to the process, to produce this table. It shouldn't be that computationally demanding to compute a dominators for a graph of only 80,000 nodes (Figure 11), so we decided to re-implement the dominator computation in Java.

We think that the reason the Crocopat script is so inefficient at computing dominators is that each iteration requires two existential quantifiers, which are $> O(n^2)$ [7]. The existential quantifiers are used to compute the intersection of the dominator sets for the predecessors of a node.

Figure 10 Computing object control dominators for TSafe

Iteration	Size of Worklist	Size of Dominator Relation
1	1	0
2	5	1
3	8	11
4	72,269	31
5	12,206	289,046
6	75,491	324,647

We found Figure 10 very surprising, and it caused us to investigate the topology of the graph (Figure 11).

Figure 11 characterizes the topology of the object control graph for TSafe running the 450kb sample input. There is one node, the TSafe ConfigConsole, with a fan-out of almost 70,000. 98% of the nodes are leaf nodes (have no successors; fan-out is zero).

Figure 11 Topology characteristics of object control graph for 450kb TSafe example. 81,809 edges. 80,259 nodes.

Fan-In	Count
1	78,734
2	1,508
3	12
4	3
5	1
6	1

Fan-Out	Count
0	78,777
1	893
$1 < f \leq 10$	569
$10 < f \leq 100$	15
$100 < f < 69,270$	4
69,270	1

Figure 11 informed the design decisions for our Java re-implementation of the dominator computation. The vast majority of nodes have a single predecessor and no successors. We name these *tree-leaf* nodes. Similarly, we refer to nodes with a single predecessor as *tree-nodes*. The set of tree-leaf nodes is a subset of the tree-nodes. We name nodes with multiple predecessors *join-nodes*.

Tree-nodes require only a single pointer to represent their predecessor relation — there is no need to allocate a dynamic data structure. Also, tree-nodes do not need to explicitly represent their dominator sets: it is simply the dominator set of the predecessor plus self. The immediate dominator of a tree-node is simply the predecessor. Figure 10 shows that the explicit representation of the global dominator relation can grow quite large; implicitly representing this information for 98% of the nodes should save a substantial amount of space.

Computing dominators is a well studied problem. Cooper et al. [15] provide a recent survey, and argue that a carefull implementation of an iterative algorithm can be practically competitive with more sophisticated algorithms that have lower asymptotic bounds (eg, Lengauer-Tarjan). Georgiadis et al. [20] provide a more recent study that shows the Lengauer-Tarjan is competitive with the Cooper et al. [15] algorithm for common control-flow graphs.

Cooper et al. [15] note the following problems with naïve implementations of the iterative algorithm:

- Bit vectors are space efficient but are too slow to compute intersections. (Computing the intersections is essentially where Crocopat is failing.)
- Sparse sets compute intersections quickly, but take up too much space.

They note that using properly ordered lists to represent the dominator sets enables efficient intersection: the intersection of two lists is their common prefix. This ordering of the dominator sets also allows trivial computation of the immediate dominator: it's the penultimate element in the list.

However, the naïve representation of a list, like sparse sets, takes too much space. They note that these lists form a tree (the dominator tree), and this tree exploits the redundancy in the lists.

We think we can get adequate space savings by simply not explicitly representing the dominator set for the tree-nodes, which are about 98% of the nodes. Our conjecture is that explicitly storing lists of dominators for the remaining 2% of the nodes (the join-nodes) will not consume an overwhelming amount of space. It is easier to simply keep a list for each of these join-nodes than to construct the tree that Cooper et al. [15] suggest to exploit the common structure of the lists.

With both the Cooper et al. [15] tree representation and our implicit representation for tree-leaf nodes, it is much easier to iterate dominator sets from the node backwards to the root, rather than forwards from the root to the node. This means that the intersection computation iterates backwards over the two lists being intersected and stops when they are the same. As Cooper et al. [15] describe, this requires a 'two-finger' approach, and knowing the preorder number of the nodes (or, in our case, the shortest depth from the root).

The Cooper et al. [15] iterative algorithm initializes the dominator set of each node to be the entire graph, and then starts from the leaves and works backwards, computing intersections that reduce the size of the sets. This only makes sense when using the tree to exploit sharing. Since we are not using this data structure, we start with empty sets and work forwards from the root (as the idea of dominators is often described pedagogically).

Cooper et al. [15] compare their algorithm empirically against the almost linear time Lengauer-Tarjan algorithm, and find that their iterative algorithm performs 2.5 times better for realistic control flow graphs (this figure has been disputed by Georgiadis et al. [20], who claim the approaches are equal). They also construct 'unrealistically large graphs' of around 30,000 nodes, and note that the iterative algorithm is still competitive at this size. They suspect that the asymptotic lower bound of the Lengauer-Tarjan algorithm will start to come into play for larger graphs. While our graphs are larger, if we subtract the 98% of nodes that are tree-nodes, we are left with a tractable graph of a few thousand nodes.

It is possible that this TSafe object control graph is not topologically representative of the object control graphs that will be generated from other programs. However, we suspect that having many leaf objects with single predecessors will be common.

Our Java re-implementation computes dominators and immediate dominators for this TSafe object control graph in under 7 seconds.

5.3 Object Control Properties

Computing the control property (eg, unique, linear, shared) of each object with our Crocopat implementation was also not scaling to TSafe. We now believe this is because the computation of non-linear parents requires three existential quantifiers (there exists a time$_2$ between time$_1$ and time$_3$). As discussed above, existential quantifiers cost $> O(n^2)$ in Crocopat.

We have augmented our dominator computation to also compute the control property of each object as it constructs the graph. Construction follows the trace: edges that occur earlier in the trace are added sooner. We store the predecessors of each node in the order that they are added. When a new predecessor is added, we check if it is already in the list of predecessors for that node. If it is already in the list, and not the last element of that list, then we know the node is shared. Tree-nodes (ie, only one predecessor) have a unique controller. Join-nodes (ie, multiple predecessors) are controlled linearly if we do not detect while constructing the graph that they are shared.

5.4 Messages

The Crocopat script that computes the message properties of each object completed the 450kb TSafe example in 57 minutes. This is a simple analysis that basically just counts the number of readers and writers for each object: it should take no more than linear time, and could be implemented with gawk.

Again, we think it is the (necessary) use of existential quantifiers in the Crocopat formulation that is causing the problems.

5.5 Groups

The Crocopat script that generalizes properties of objects to properties of groups of objects cannot complete the 450kb TSafe example. The BDD package runs out of memory. This script contains snippets such as the following, that identifies all fields that refer to only shared objects:

```
FieldGroupShared(c) := EX(o, FieldGroup(c,o) & Shared(o)) &
!EX(o, FieldGroup(c,o) & !Shared(o));
PRINT ["FieldGroupShared"] FieldGroupShared(c) TO "out.rsf";
```

This should be a fairly fast and easy computation, but expressing it in Crocopat requires these existential quantifiers. We will re-implement this computation as well.

6 Related Work

The problem of non-local mutation of shared data making programs difficult to understand and reason about has been addressed in many ways. At one extreme, 'pure' functional programmers disallow it entirely. Most people most of the time find that shared mutable data is useful, and much research has gone into finding ways to control and reason about it. In the words of Noble et al. [30]:

> Any attempt to address the aliasing problem for practical object-oriented programming must be evaluated as an engineering compromise: how much safety does it provide, at what cost, and, most importantly, how usable are the mechanisms by typical programmers doing general purpose programming. The crucial question is how natural (or how contrived) a programming style is required by the proposed aliasing mode checking.

The most common approach is to develop a type system where the programmer provides annotations that are checked in a modular fashion. These type systems are generally designed around either a notion of encapsulation, often called 'ownership', or around alias control.

All of these static systems emphasize safety and strive for usability. By contrast, our dynamic system emphasizes usability (we place no extra restrictions on the programmer), and stives for safety. Which produces higher quality software is an empirical question that has as of yet to be answered.

6.1 Static Specification and Verification

Uniqueness. A unique object may have at most one incoming reference at a time [11, 22, 23]. Uniqueness type systems achieve encapsulation by ensuring that the whole is the only object with a reference to its part. Thus, the object that references a unique type fully encalsulates that type. By preventing data sharing, one can reason modularly about unique types. However, unique types make data communication much more difficult.

Alias transfer occurs via *destructive reads*, in which a reference becomes null at the same time it is read [22]. In addition to requiring modified assignment statements, these systems require a programming style that trades aliases back and forth. Invoking a purely functional method on a unique variable requires returning that parameter alias in addition to the normal return value of the method. Furthermore, it is unclear how programmers should handle unique objects that have fields that sometimes may be null. In this case, reducing complexity with a uniqueness invariant increases the complexity of the representation invariant, even though uniqueness can identify an object's representation.

The problem with most approaches to unique types is that they are too restrictive. More advanced research has tried to make these systems less restrictive while still retaining some control.

[22, 26] weaken the uniqueness invariant to permit short-term borrowing, or *dynamic aliases* (ie, local variables), that are never assigned to fields. Dynamic aliases make it more difficult to reason about sharing. Weakening an impractical invariant is less desirable than finding the right invariant.

Instead of mandatory destructive reads, *alias burying* permits non-destructive reads in cases where destruction is "unnecessary" [11]. A conservative use of unnecessary means that a unique reference may be read if the original reference is never used after the read. If the behavior is the same as doing a destructive read, then there is no need to actually do one. Alias burying can therefore capture control transfer (factories and listeners), but not borrowing (iterators, containers, intentionally shared data). The necessarily conservative type system results rely on an expensive static analysis.

One way in which our work differs from much of the work just described is that we focus on the objects that are *in control* rather than on the aliasing in the program. We are not concerned with who has a reference or how they got it, only with what happens in the program execution on account of them having it. So, for example, we do not distinguish between aliases attained through local variables or through fields – which is an important distinction for much of the above work.

Ownership. Ownership type systems explicitly encapsulate owned objects. In the most conservative approach, *owner as dominator*, an object may only be referenced through its owner [14]. The owner is said to dominate because all paths in the heap go through the owner. Only the owned objects are encapsulated; the owner may be aliased by any object. Because no object necessarily has one alias, control transfer is non-local; thus, an object's owner is typically invariant across execution.

External uniqueness relaxes uniqueness such that internal objects may alias the unique type [13], much like how owned objects may alias their parents. To determine the unique object's internal state, external uniqueness uses an owner as dominator type system.

Dominator ownership systems prevent useful program idioms, some of which use readonly aliasing, eg, iterators, factories, containers, observers. Modifier as owner type systems enforce that only mutating paths in the heap must go through the owner, which permits modular verification and more implementations than owner as dominator. [17, 28, 29] have a static notion of owner as mutator using the Universes type system and JML. However, modifier as owner as a program invariant prevents intentional sharing, eg, iterators that remove elements.

Read-only annotations. Read-only annotations on pointer variables provide some control over where data is mutated from. This idea exists in some form in C/C++ in the `const` keyword. Recent proposals have described adding this functionality to Java [9, 33]. The primary difference between this approach and approaches based on uniqueness and ownership is that reference immutability is not centred on a notion of encapsulation. While reference immutability helps the programmer manage mutable data, it does not help with modular reasoning.

6.2 Inference

There has been relatively little work on ownership inference, compared with the work on static specification and verification.

Static. The most elaborate static inference system is that of Heine and Lam [21]. They define owner as allocator/de-allocator, and consider that the variables that allocate and de-allocate an object own it. They do a static analysis that attempts to push the ownership bit through all the assignment statements between the allocator and the de-allocator. If their system can connect the dots, it proves that the object is disposed of exactly once. They use their system to detect potential memory leaks, and do not communicate the ownership information to the programmer, nor do they attempt to address the matter of encapsulation.

Boyapati [10] has an intra-procedural inference system to accompany his type system.

Aldrich et al. [3] has an inference system that the author's web-page seems to claim is not quite as satisfactory as one may be led to believe from reading the paper.

Dynamic. Wren [34] provides a theoretical foundation for inferring Clarke/Noble [14] ownership types by finding invariant dominators in runtime samples and then relaxing owners to contruct consistent ownership type annotations. This is a theoretical framework that was never implemented.

Wren [34] is similar to our work in that it describes an analysis that looks for dominators in graphs that are captured dynamically. The difference is in the graphs: Wren [34] proposes to capture entire heap graphs, wheres we capture mutation chains. Mutation chains have the practical advantage of being *much* smaller than heap graphs, and we argue that object control invariants get more at the core of encapsulation than ownership does.

7 Discussion

7.1 Scalability

Crocopat. Much to our surprise, in this project Crocopat proved to be very slow and memory intensive. This was not our experience in previous work [31]. Crocopat is based on BDDs so that it can scale [8]. In this work, all of the analyses that we originally implemented in Crocopat and then re-implemented in another (imperative) language were much faster in the imperative incarnations. We think this is because of the cost of existential quantifiers in Crocopat, and their necessity for the properties we are computing.

Dominators. We have implemented an iterative graph dominator algorithm customized for the kinds of graphs that we expect to be analyzing. These differ significantly from common control flow graphs, because they have many tree-leaf nodes (ie, nodes with one predecessor and no successors), and because a small minority of nodes have a massive number of successors.

Trace compression. We will need to store traces more efficiently to do real case studies. We are currently generating traces that are hundreds of megabytes for TSafe, which is not that large of a program.

7.2 Evaluation

Our aspiration is that this technique will help programmers understand and reason about sharing in their programs. Ideally this will reduce unnecessary and erroneous sharing, and provide better ways to think about the essential sharing.

7.3 Future Work

Simple annotations. Our tool generates a large list of Eclipse bookmarks, even for toy programs. We currently prioritize those bookmarks that reveal sharing. Allowing the user to write simple annotations would give us a good prioritization mechanism: highlight those bookmarks where the program's behaviour deviated from the programmer's intent.

Static analysis. We are interested in developing a static analysis to compute this same information, possibly using the new context-sensitive points-to analysis of Sridharan and Bodik [32]. We have a hunch that the static and dynamic analyses will compute similar values — unlike, for example, in points-to analysis where the static and dynamic approaches tend to produce very different results [27].

Multi-threaded programs. In this paper we have only considered single-threaded programs. However, our instrumenter collects information about threads, and we believe the instrumenter is thread-safe. So we just need to think about what to do with this data about threading that we are collecting.

Program evolution. We think that the evolution of the ownership structure of a program may reveal design violations introduced during maintenance activities.

References

[1] Rahul Agarwal and Scott D. Stoller. Type inference for parameterized race-free Java. In Giorgio Levi and Bernhard Steffen, editors, *Proc. 5th VMCAI*, volume 2937 of *LNCS*, pages 149–160, Venice, Italy, January 2004.

[2] Jonathan Aldrich. *Using Types to Enforce Architectural Structure.* PhD thesis, U. Washington, August 2003.

[3] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In Satoshi Matsuoka, editor, *Proc. 17th OOPSLA*, pages 311–330, Seattle, WA, October 2002.

[4] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. 11th ECOOP*, volume 1241 of *LNCS*, Jyväskylä, Finland, June 1997. ISBN 3-540-63089-9.

[5] David F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Berkeley, December 1997. UCB/CSD-98-1017.

[6] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In James Coplien, editor, *Proc. 11th OOPSLA*, pages 324 – 341, San Jose, CA, October 1996.

[7] Dirk Beyer and Andreas Noack. Crocopat 2.1 introduction and reference manual. Technical Report UCB/CSD-04-1338, Berkeley, 2004.

[8] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient relational calculation for software analysis. *IEEE Transactions on Software Engineering*, 31 (2):137–149, 2005. URL `http://dx.doi.org/10.1109/TSE.2005.23`.

[9] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In Doug Schmidt, editor, *Proc. 19th OOPSLA*, pages 35–49, Vancouver, British Columbia, Canada, October 2004.

[10] Chandrasekhar Boyapati. *SafeJava: A Unified Type System for Safe Programming.* PhD thesis, MIT, 2004.

[11] John Boyland. Alias burying: unique variables without destructive reads. *Software—Practice & Experience*, 31(6):533–553, May 2001.

[12] Luca Cardelli, editor. *Proc. 17th ECOOP*, volume 2743 of *LNCS*, Darmstadt, Germany, July 2003. ISBN 3-540-40531-3.

[13] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In Cardelli [12], pages 176–200. ISBN 3-540-40531-3.

[14] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In Craig Chambers, editor, *Proc. 13th OOPSLA*, Vancouver, British Columbia, Canada, October 1998.

[15] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software—Practice & Experience*, (4), 2001.

[16] Greg Dennis. TSAFE: building a trusted computing base for air traffic control software. Master's thesis, MIT, January 2003.

[17] Werner Dietl and Peter Müller. Universes: Lightweight ownership for jml. *Journal of Object Technology*, 6(8):5–32, 2005.

[18] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[20] L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. Finding dominators in practice. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA 2004)*, volume 3221 of *LNCS*, pages 677–688, 2004.

[21] David L. Heine and Monica S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In Rajiv Gupta, editor, *Proc. PLDI*, June 2003.

[22] John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Proc. 6th OOPSLA*, pages 271–285, Phoenix, October 1991.

[23] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Messenger*, 3(2):11–16, 1992. ISSN 1055-6400.

[24] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In John Mitchell, editor, *29th POPL*, Portland, Oregon, January 2002.

[25] Patrick Lam and Martin Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In Cardelli [12], pages 275–302. ISBN 3-540-40531-3.

[26] Naftaly H. Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *Proc. 10th ECOOP*, volume 1098 of *LNCS*, pages 189–209, Linz, Austria, July 1996. ISBN 3-540-61439-7.

[27] Markus Mock, Manuvir Das, Craig Chambers, and Susan J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In John Field and Gregor Snelting, editors, *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Snowbird, UT, June 2001.

[28] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001.

[29] ———. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. 2002.

[30] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *Proc. 12th ECOOP*, volume 1445 of *LNCS*, Brussels, Belgium, July 1998. ISBN 3-540-64737-6.

[31] Derek Rayside, Lucy Mendel, Robert Seater, and Daniel Jackson. An analysis and visuzliation for revealing object sharing. In Margaret-Anne Story and Li-Te Cheng, editors, *Eclipse Technology Exchange (ETX)*, San Diego, CA, October 2005.

[32] Manu Sridharan and Ratislav Bodik. Refinement-Based Context-Sensitive Points-To Analysis for Java. Technical Report UCB/EECS-2005-13, Berkeley, November 2005. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2005/EECS-2005-13.html`.

[33] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In Richard P. Gabriel, editor, *Proc. 20th OOPSLA*, pages 211–230, San Diego, CA, October 2005. ISBN 1-59593-031-0.

[34] Alisdair Wren. Ownership type inference. Master's thesis, Department of Computing, Imperial College, 2003. URL `http://www.cl.cam.ac.uk/users/aw345/writings/`.