

# On the Structure of Sharing in Open Concurrent Java Programs

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

*This paper studies the structure of shared objects in open concurrent Java programs. It proposes classification of shared objects into three categories: central, owned and distributed. This classification facilitates program understanding and error detection in concurrent programs.*

*The paper presents a new static analysis that infers central, owned and distributed objects in open concurrent Java programs. It presents experiments that reveal the structure of shared objects in real-world Java programs.*

## 1 Introduction

In recent years, research on concurrent programming with shared memory has gained considerable momentum. The vast majority of research falls into two categories (1) work on error prevention, and (2) work on error detection. Work on error prevention focuses on language-based techniques (i.e., type systems) that disallow concurrency errors such as data races and deadlocks. Work on error detection focuses predominantly on dynamic techniques that detect concurrency errors at runtime.

Surprisingly, little has been done on understanding concurrent programs, or more specifically on understanding the structure of shared objects in concurrent programs. Analysis of the structure of shared objects can enhance general-purpose program understanding, and can lead to more effective techniques for concurrency control and concurrency error detection.

This paper studies the structure of shared objects in *open concurrent* Java programs. It classifies shared objects as *central*, *owned* or *distributed*. Informally, a *central* object is an object *directly* accessed by client threads. An *owned* object is an object *indirectly* accessed by client threads; however, an owned object is dominated by an owner object, meaning (informally) that each access to that object goes through the owner object. A *distributed* object is indirectly accessed by client threads as well; however, a distributed object is not dominated by an "owner" object; instead, ac-

cesses to it are distributed through multiple objects. The paper posits that central and owned objects are "easy to reason about", while distributed objects are "difficult to reason about"; this means that it is easy to understand the behavior of central and owned objects and prove those objects safe or unsafe, while it is difficult to understand the behavior of distributed objects and prove those objects safe or unsafe. The paper also posits that well-structured concurrent programs should emphasize the role of ownership — they should strive for a maximal number of owned objects and for a minimal number of distributed objects.

This classification can improve general-purpose program understanding and can lead to more effective techniques for concurrency error detection.

This paper presents a novel static analysis that infers central, owned and distributed objects in open concurrent Java programs. It presents experiments with the analysis on publicly available open concurrent Java programs, and a detailed case study on one of the programs, `jdbm`. The experiments and the case study indicate that (1) the analysis infers central, owned and distributed objects precisely and efficiently, (2) ownership plays an important role, and (3) although the majority of objects, 60% on average, are central and owned (and easy to reason about), a large percentage, 40%, are distributed (and difficult to reason about).

The paper has the following contributions:

- It proposes a classification of shared objects into *central*, *owned*, and *distributed*, and novel static analysis that infers central, owned and distributed objects.
- It presents experiments with the analysis on several publicly available open concurrent Java programs.

## 2 Problem Setting

We consider a set  $Cls$  of interacting classes that constitute an open concurrent program. The classes in  $Cls$  are intended to be thread-safe and typically include multiple synchronized methods. In addition, we consider a set  $Int$  of methods and fields from the classes in  $Cls$ . The methods

and fields in *Int* define the interface to the client. In general, *Int* could contain a subset of all public methods and fields in *Cls* or it may contain the entire set of public methods and fields in *Cls*. For our purposes, *Int* contains the entire set of public methods and fields in *Cls*.

A client of *Cls* is any arbitrary Java class that calls methods from *Int* and reads/writes fields from *Int*, and does not access any methods/fields from *Cls* that are not in *Int*.  $AllClients(Int)$  denotes the set of all possible clients for *Int*; clearly, this set is infinite. We assume that *Cls* is closed with respect to *Int*: that is, for any arbitrary client  $C \in AllClients(Int)$ , any class that could be referenced during the execution of *C* is included in *Cls*. In other words, we assume that clients of *Cls* exercise interactions only among classes from the given set *Cls* (i.e., the currently available program). These constraints on clients are consistent with previous work on analysis of open programs [18, 12].

We illustrate the problem setting with one of our benchmarks, `jdbm`, an open concurrent program which implements a transactional persistence engine. Figure 1 shows excerpts from classes `BaseRecordManager`, `RecordFile` and `TransactionManager`; these classes and their methods are public. *Int* consists of the public methods in classes `BaseRecordManager`, `TransactionManager` and `RecordFile`. An arbitrary client that meets the constraints outlined above is shown in Figure 2.

Note that the clients in  $AllClients(Int)$  are single-threaded — essentially, a client is a single method `main`. However, this is sufficient to enable reasoning about the interactions between actually multithreaded clients and *Cls*. Clearly, every object accessed in the client’s `main` method is an object that can be directly shared by multiple threads: every method/field accessed in `main` can be directly called/accessed by multiple threads.

We classify the run-time objects during executions of clients  $C \in AllClients(Int)$  into central, owned and distributed. Our goal is to reason about all clients and infer central, owned and distributed objects across all executions of clients in  $AllClients(Int)$ .

### 3 Views of Program Execution

This section presents two views of a program execution, the *run-time object graph* (Section 3.1), and the *run-time method sequence graph* (Section 3.2). Section 4 uses these views to define the classification of shared objects.

Throughout this paper, run-time objects are denoted using superscript *r*: e.g.,  $o^r$ ,  $o_i^r$ ,  $o_1^r$ , etc. Analysis objects (i.e., abstract objects that represent the run-time objects) are denoted using exactly the same notation but without the superscript *r*: e.g.,  $o$ ,  $o_i$ ,  $o_1$ , represent objects  $o^r$ ,  $o_i^r$ ,  $o_1^r$ , respectively. Our analysis represents objects by their allocation site: for example, all run-time objects allocated at alloca-

tion site  $s_i$  are represented by analysis object  $o_i \in O$  (here  $O$  denotes the set of all analysis objects).

For brevity, we do not discuss static fields and static methods. They are handled in the implementation.

#### 3.1 Run-time Object Graph

A *run-time object graph* represents a view of a program execution. The nodes in the run-time object graph are the run-time objects, and the edges represent the access relationships between these run-time objects.

Let  $C \in AllClients(Int)$  be a client of *Cls*, and let  $E_C$  be an execution of *C*. Let  $Og_{E_C}^r$  be the run-time object graph for this execution.  $Og_{E_C}^r$  is constructed as follows:

- There is a field edge  $o^r \xrightarrow{f} o_1^r$  in  $Og_{E_C}^r$  if at some point of the execution  $E_C$ , field *f* of object  $o^r$  refers to object  $o_1^r$ .
- Similarly, there is an array edge  $o^r \Downarrow o_1^r$  in  $Og_{E_C}^r$  if  $o^r$  is an array object, and at some point of the execution  $E_C$ ,  $o^r$  has element  $o_1^r$ .
- There is an unlabeled edge  $o^r \rightarrow o_1^r$  in  $Og_{E_C}^r$  if at some point of the execution  $E_C$ , an instance method invoked on receiver object  $o^r$  has local variable *l*,  $l \neq \text{this}$ , that refers to object  $o_1^r$ <sup>1</sup>.

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

This definition of the run-time object graph is consistent with earlier definitions [4, 13]. Note that  $Og_{E_C}^r$  accumulates edges as the program executes and never deletes edges; at the end of the execution,  $Og_{E_C}^r$  contains all edges that have been active during the program run.

#### 3.2 Run-time Method Sequence Graph

A *run-time method sequence graph* represents another view of a program execution. The nodes in the run-time object graph are the run-time tuples  $o^r.m()$ , where  $o^r.m()$  denotes that instance method *m* is invoked on receiver  $o^r$ . The edges represent the calling relationships between these run-time tuples.

For convenience, we denote field accesses not through `this` (i.e.,  $p = q.f$ ,  $q \neq \text{this}$  and  $p.f = q$ ,  $p \neq \text{this}$ ), and array accesses (i.e.,  $p = q[i]$  and  $p[i] = q$ ) as special method calls. Notation  $o_2^r.rd$  denotes the execution of a read  $p = q.f$ ,  $q \neq \text{this}$  where *q* refers to  $o_2^r$ . Similarly,  $o_2^r.wr$  denotes the execution a write  $p.f = q$ ,  $p \neq \text{this}$  where *p* refers to  $o_2^r$ .

<sup>1</sup>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  $r_1=r.m(); r_1.n()$ ).

Again, let  $C \in AllClients(Int)$  be a client of  $Cls$ , and let  $E_C$  be an execution of this client. Let  $Og_{E_C}^{r+}$  be the run-time method sequence graph for this execution.  $Og_{E_C}^{r+}$  is constructed as follows:

- There is an edge  $o_1^r.m_1() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$  if at some point of the execution  $E_C$ , method  $m_1$  invoked on receiver  $o_1^r$  executes a call  $p.m_2(), p \neq \text{this}$  in  $m_1$ , where  $p$  refers to  $o_2^r$ , and the call leads to the invocation of method  $m_2$  on receiver  $o_2^r$ .
- There is an edge  $o_1^r.m_1'() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$  if at some point of the execution  $E_C$ , method  $m_1'$  invoked on receiver  $o_1^r$  has executed a call  $\text{this}.m_1()$  in  $m_1'$ , and the call  $\text{this}.m_1()$  has resulted in edge  $o_1^r.m_1() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$ .

As mentioned earlier, the method sequence graph shows the calling relationships between run-time tuples  $o^r.m()$ . The two bullets capture two cases. In the first case, method  $m_1$  executing on receiver  $o_1^r$  directly calls  $m_2$  on receiver  $o_2^r$  through call site  $p.m_2()$ , which directly results in edge  $o_1^r.m_1() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$ . In the second case, method  $m_1$  "jumps through" calls through  $\text{this}$  until it reaches a call not through  $\text{this}$ : for example, if there is a method  $m_1'$  invoked on receiver  $o_1^r$  and  $m_1'$  executes a call  $\text{this}.m_1()$ , then in turn  $m_1$  executes a call  $p.m_2(), p \neq \text{this}$ , where  $p$  refers to  $o_2^r$ , then there is an edge  $o_1^r.m_1'() \rightarrow o_2^r.m_2() \in Og_{E_C}^{r+}$  in the method sequence graph. The method sequence graph reflects transfer of control between distinct objects.

Throughout the paper we use the standard notation for reachability. E.g.,  $o_1^r.m_1() \rightarrow^* o_2^r.m_2() \in Og_{E_C}^{r+}$  denotes a path of zero or more edges in  $Og_{E_C}^{r+}$  from  $o_1^r.m_1()$  to  $o_2^r.m_2()$ ; similarly,  $o_1^r.m_1() \rightarrow^+ o_2^r.m_2() \in Og_{E_C}^{r+}$  denotes a path of one or more edges.

### 3.3 Discussion

The object graph and the method sequence graph represent two different views of a program execution. The object graph represents a structural (static) view of the execution — it shows the access relationships between objects and is used to reason about structural (static) properties such as ownership. On the other hand, the method sequence graph represents a dynamic view of the execution — it shows the transfer of control between distinct objects.

We use the object graph and the method sequence graph to define the classification of shared objects. Our analysis, presented later in the paper, infers static approximations of these graphs, and proceeds to reason about the structure of shared objects using these static approximations.

## 4 Classification of Shared Objects

### 4.1 Defining the Classification

Let  $C \in AllClients(Int)$  be a client of  $Cls$  as described in Section 2, and let  $E_C$  be an execution of this client. We

define the notion of *shared* objects, and proceed to classify the *shared* objects into *central*, *owned* and *distributed*.

The set of *shared* objects in  $E_C$ , denoted by  $S_{E_C}$  is the union of the set of *directly shared* objects denoted by  $DS_{E_C}$ , and the set of *indirectly shared* objects denoted by  $IS_{E_C}$ .

The set of directly shared objects is defined as follows:

$$DS_{E_C} = \{o^r \mid \exists \text{root.main}() \rightarrow o^r.m() \in Og_{E_C}^{r+}\}.$$

The definition states that there exists an edge in  $Og_{E_C}^{r+}$  from  $\text{root.main}()$  to a tuple  $o^r.m()$  (i.e.,  $\text{main}$  directly calls a method on receiver  $o^r$ ). The objects  $o^r$  are potentially directly accessed by multiple threads —  $o^r$ 's methods/fields can be called/accessed directly by multiple threads.

The set of indirectly shared objects is defined as follows:

$$IS_{E_C} = \{o^r \mid o^r \notin DS_{E_C} \wedge \exists o_j^r \in DS_{E_C} \text{ s.t. } o_j^r \xrightarrow{f} o^r \in Og_{E_C}^{r+} \wedge \exists p : \text{root.main}() \rightarrow^+ o^r.m() \in Og_{E_C}^{r+}\}.$$

The definition states that an object  $o^r$  is indirectly shared, if the following three conditions are met. The first condition,  $o^r \notin DS_{E_C}$ , states that  $o^r$  must not be a directly shared

object. The second condition,  $\exists o_j^r \in DS_{E_C} \text{ s.t. } o_j^r \xrightarrow{f} o^r \in Og_{E_C}^{r+}$ , states that  $o^r$  must be a transitively reachable field of some directly shared object  $o_j^r$ . This condition excludes "temporary" objects from consideration: if an object  $o_1^r$  is not reachable from a directly shared object on a sequence of field accesses, then  $o_1^r$  is local to the execution of a particular method, and  $o_1^r$  cannot be shared. The third condition,  $\exists p : \text{root.main}() \rightarrow^+ o^r.m() \in Og_{E_C}^{r+}$ , states that there must exist a path  $p$  in  $Og_{E_C}^{r+}$  from  $\text{root.main}()$  to  $o^r.m()$  (i.e., the execution of the client's  $\text{main}$  leads to access of object  $o^r$ ). Note that this access happens indirectly, that is, through one or more intermediate objects.

Thus, the set of shared object is as follows:

$$S_{E_C} = DS_{E_C} \cup IS_{E_C}.$$

We are ready to define the set of *central* objects, *owned* objects and *distributed* objects in  $E_C$ .

The set of *central* objects in  $E_C$ , which we denote by  $C_{E_C}$ , is the set of directly shared objects:

$$C_{E_C} = DS_{E_C}.$$

The set of *owned* objects in  $E_C$ , denoted by  $O_{E_C}$  is defined as follows:

$$O_{E_C} = \{o^r \mid o^r \in IS_{E_C} \wedge \exists o_1^r \text{ s.t. } \forall p : \text{root.main}() \rightarrow^+ o^r.m() \in Og_{E_C}^{r+} \wedge p \text{ is } \text{root.main}() \rightarrow^+ o_1^r.m_1() \rightarrow^+ o^r.m()\}.$$

The definition states that an object  $o^r \in S_{E_C}$  is *owned*, if the following two conditions are met. The first condition,

$o^r \in \mathcal{IS}_{E_C}$ , states that  $o^r$  must be an indirectly shared object. The second condition states that there must exist an object  $o_1^r$ , such that every path from `root.main()` to  $o^r$ , goes through  $o_1^r$ . In other words, an owned object is potentially indirectly accessed by multiple client threads; however, each access goes through the same owner object.

Finally, the set of *distributed* objects in  $E_C$ , which we denote by  $\mathcal{D}_{E_C}$  is defined as follows:

$$\mathcal{D}_{E_C} = \{o^r \mid o^r \in \mathcal{IS}_{E_C} \wedge o^r \notin \mathcal{O}_{E_C}\}.$$

The definition states that an object  $o^r \in \mathcal{S}_{E_C}$  is *distributed* if it is indirectly shared and it is not owned. That is, a distributed object is an object potentially indirectly accessed by multiple client threads; however, unlike an owned object, these accesses happen in a distributed manner, through multiple distinct objects.

Note that this classification ignores synchronization and object immutability. It intends to classify objects into "easy to reason about" and "difficult to reason about" not into safe and unsafe from concurrency errors. Therefore, objects of each kind can be safe or unsafe. For example, an owned object can be safe if it is immutable, or if it is protected by synchronization (e.g., on itself, or on an owner); conversely, an owned object can be unsafe if it is mutated and it is not protected by synchronization. Similarly, a distributed object can be safe if it is immutable, or if it is protected by synchronization; it can also be unsafe. It turns out however, that owned objects tend to be safe, while distributed objects tend to be unsafe.

## 4.2 Example

We illustrate the classification with `jdbm`. Figure 1 shows excerpts from classes `BaseRecordManager`, `RecordFile` and `TransactionManager` in `jdbm`. Figure 2 shows a client of these classes which meets the constraints outlined in Section 2.

Figure 3 shows the object graph corresponding to Figures 1 and 2. In this example, which is intentionally simplified, the set of run-time objects coincides with the set of analysis objects. We use notation  $o$  without superscript  $r$  to denote the objects. In general, there are more than one run-time objects mapped to a single analysis object.

Recall that an edge from object  $o_1$  to object  $o_2$  means that  $o_1$  has access to  $o_2$ , or in other words  $o_1$  holds a reference to  $o_2$  and consequently, methods invoked on receiver  $o_1$  may call methods on receiver  $o_2$ . Object  $o_{BaseRecordManager}$  (created at creation site 4 in Figure 2) accesses objects  $o_{RecordFile}$  (created at site 1 in Figure 1),  $o_{PRowIdManager}$  (created at site 2 in Figure 1), and  $o_{TransactionManager}$  (created at site 3 in Figure 1); the references to the first two are through fields `_file` and `_physMgr`; the reference to the last is through local variable `txnMgr` in method `getTxnManager`. Method `main` of the client calls methods on  $o_{BaseRecordManager}$  and  $o_{TransactionManager}$ ; the

```
public class BaseRecordManager {
    private RecordFile _file;
    private PhysicalRowIdManager _physMgr;
    BaseRecordManager(RecordFile file) {
1   _file = new RecordFile();
2   _physMgr = new PhysicalRowIdManager(_file);
    }
    public synchronized TransactionManager
        getTxnManager() {
        TransactionManager txnMgr = _file.txnMgr;
        return txnMgr;
    }
    public synchronized void close() {
        _file.close();
        _physMgr.close();
    }
}

public final class RecordFile {
    final TransactionManager txnMgr;
    RecordFile() {
3   txnMgr = new TransactionManager(this);
    }
    ...
}

public class TransactionManager {
    private RecordFile owner;
    TransactionManger(RecordFile file) {
        owner = file;
    }
    public void synchronizeLog() {
        ...
        owner.synch();
    }
}
```

**Figure 1.** `BaseRecordManager` from `jdbm`.

first object is created in `main`; the second object is returned to `main` by method `getTxnManager`.

In this example, object  $o_{BaseRecordManager}$  is a directly shared object, and therefore it is a central object:

`root.main() →  $o_{BaseRecordManager}.getTxnManager()$ .`

Object  $o_{TransactionManager}$  is central as well:

`root.main() →  $o_{TransactionManager}.synchronizeLog()$ .`

Object  $o_{PRowIdManager}$  is an owned object because all accesses from `root.main()` to it go through  $o_{BaseRecordManager}$ . For example:

`root.main() →  $o_{BaseRecordManager}.close()$   
→  $o_{PRowIdManager}.close()$ .`

Finally,  $o_{RecordFile}$  is a distributed object. It is an indirectly shared object; however, unlike the accesses to owned object  $o_{PRowIdManager}$ , the accesses to  $o_{RecordFile}$  are distributed through two distinct objects:

`root.main() →  $o_{TransactionManager}.synchronizeLog()$   
→  $o_{RecordFile}.synch()$  and`

`root.main() →  $o_{BaseRecordManager}.close()$`

```

class Client {
  public static void main() {
4 BaseRecordManager brm =
      new BaseRecordManager();
  TransactionManager tm = brm.getTxnManager();
  brm.close();
  tm.synchronizeLog();
}
}

```

Figure 2. Client of `BaseRecordManager`.

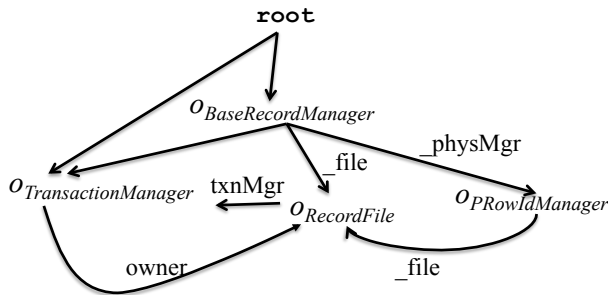


Figure 3. Object graph.

→ `ORecordFile.close()`.

### 4.3 Discussion

Central objects are the objects potentially directly accessed by multiple threads (e.g., `OBaseRecordManager` is a central object). Therefore, their classes are typically synchronized with the intention to be thread-safe.

Typically, a central object creates and accesses a large number of objects during its lifetime. Some of these objects remain “hidden” behind the creating central object and all accesses to them go through their creating central object (e.g., `OPRowIdManager` is “hidden” behind central object `OBaseRecordManager`). Other objects become accessible to other central objects (e.g., `ORecordFile` is created by central object `OBaseRecordManager` but eventually it becomes accessible to central object `OTransactionManager` as well).

Owned objects are hidden behind an owner object, typically one of the central objects. Distributed objects are the shared objects not hidden behind an owner object — they can be accessed in a distributed manner, through several central objects.

There are two important benefits from the classification. First, the classification facilitates general-purpose program understanding. It provides a simple and intuitive classification of the shared objects in concurrent programs with complex shared structures. Second, it facilitates concurrency error detection. It classifies the shared objects into two categories: “easy to reason about” and “difficult to reason about” objects; an error detection technique, manual or automatic, can examine “easy” objects quickly, and focus on “difficult” objects thereafter. We believe that future work

should focus on design patterns that maximize the number of “easy” objects and minimize the number of “difficult” objects — this will facilitate both general-purpose program understanding and error detection.

Consider the problem of reasoning about *object races*.<sup>2</sup>

Central objects are “easy to reason about”. A central object  $o_c$  is accessed directly in `main`. Any pair of methods called on  $o_c$ , say  $o_c.m_1()$  and  $o_c.m_2()$ , where one of  $m_1$  or  $m_2$  is unsynchronized, create a potential object race on  $o_c$ .

Owned objects are “easy to reason about” as well. An owned object  $o_o$  has an owner object  $o_c$ . There are two important benefits from ownership. First, often an owned object is protected by synchronization on its owner. In our running example `OPRowIdManager` is protected by synchronization on its owner `OBaseRecordManager`; every thread holds the lock on owner `OBaseRecordManager` when accessing `OPRowIdManager`, and thus, it is guaranteed that no object race occurs on `OPRowIdManager`. Second, even if the owned object is not protected by synchronization on its owner  $o_c$ , reasoning about object races is simplified by ownership: the detection technique must first identify an object race on  $o_c$ , and if such a race exists, proceed to search for an object race on  $o_o$  *exclusively* within the ownership boundary of  $o_c$ . Such structured detection can reduce search space and search time significantly.

On the other hand, distributed objects are “difficult to reason about”. A distributed object  $o_d$  may be accessed through multiple objects — generally, a search for object races on  $o_d$  will need to traverse the entire set of objects which could be quite large. In our example, `ORecordFile` is a distributed object — it can be accessed along several paths: e.g., `root` → `OBaseRecordManager` → `OPRowIdManager` → `ORecordFile`.

## 5 Preliminary Analysis

We perform five preliminary analyses — fragment analysis (Section 5.1), points-to analysis (Section 5.2), object graph analysis (Section 5.3), ownership analysis (Section 5.4), and method sequence analysis (Section 5.5).

### 5.1 Fragment Analysis

Points-to, object graph, ownership, and method sequence analyses are designed as *whole-program analyses*. However, the problem considered in this paper requires analysis of an open program: the input is a set of classes  $Cls$  and the interface to  $Cls$   $Int$ . We use a general technique called *fragment analysis* [18] which constructs a conservative client that approximates all clients  $C \in AllClients(Int)$ .

For brevity, in further examples and figures, we use the client in Figure 2 instead of the conservative client. Our

<sup>2</sup>Informally, an object race on an object  $o$  occurs when two threads access  $o$  “simultaneously”. An object race may or may not trigger a data race.

implementation uses conservative clients created according to the rules of the fragment analysis [18].

## 5.2 Points-to Analysis

Points-to analysis determines the set of objects that a given reference variable or a reference field may point to. In this paper, we consider the Andersen-style flow- and context-insensitive points-to analysis for Java [17, 9].<sup>3</sup>

The analysis distinguishes objects per allocation sites — each allocation site  $s_i$  corresponds to analysis object  $o_i \in O$ . The analysis computes the points-to graph,  $Pt$ , of the program. We extend the  $Pt$  notation to denote points-to sets:  $Pt(l)$  denotes the points-to set of variable  $l$ . The semantics of the analysis is well-known [17].

When the points-to analysis is applied on the completed program (i.e.,  $Cls$  and the conservative client), the properties of the fragment analysis guarantee that the constructed points-to graph  $Pt$  approximates the points-to graph of each  $C \in AllClients(Int)$  [18, 16].

## 5.3 Object Graph Analysis

Object graph analysis approximates the run-time object graphs over executions of clients  $C \in AllClients(Int)$ .

Recall from Section 3.1 the definition of the run-time object graph for an execution of a client of  $C \in AllClients(Int)$ . The object graph analysis constructs the static object graph,  $Og$ . The nodes in  $Og$  are taken from the set of analysis objects  $O$ , and the edges represent the access relationships.  $Og$  approximates the run-time object graphs over all executions of clients  $C \in AllClients(Int)$ : if there is a run-time access edge  $o_1^r \rightarrow o_2^r$  for some execution of some  $C$ , then there is an edge in  $Og$  from the representative of  $o_1^r$  to the representative of  $o_2^r$  (i.e., each  $Og_{EC}^r$  is represented by  $Og$ ).

The object graph analysis uses the points-to graph  $Pt$  computed by the points-to analysis and computes the object graph  $Og$ . For brevity, we do not present this analysis; it is beyond the scope of this paper. The analysis is presented in our previous work [10] and in our technical report [11].

The object graph constructed for the program consisting of Figure 1 and its client in Figure 2, is shown in Figure 3.

## 5.4 Ownership Analysis

The ownership analysis consists of two parts: dominance boundary analysis (Section 5.4.1), and minimal boundary analysis (Section 5.4.2).

<sup>3</sup>Flow-insensitive analyses do not take into account the flow of control between program points and are less precise and less expensive than flow-sensitive analyses. Context-sensitive analyses distinguish between different calling contexts of a method and are more precise and more expensive than context-insensitive ones.

### 5.4.1 Dominance Boundary Analysis

The dominance boundary analysis infers the *dominance boundary* of an object  $o_i \in O$ . Informally, the dominance boundary of an object  $o_i \in O$  is the subgraph of the object graph  $Og$  that is dominated by  $o_i$  (i.e., all accesses to objects in the boundary go through  $o_i$ ).

The dominance boundary analysis uses the object graph  $Og$ , takes as input  $o_i$ , and computes  $Boundary(o_i)$ , the dominance boundary of  $o_i$ . The ownership analysis is not shown here; it is beyond the scope of this paper. The analysis is presented and proven correct in [11].

The following lemma holds for  $Boundary(o_i)$ :

**Lemma 5.1** *Let  $o_i \in O$  be any analysis object and let  $Boundary(o_i)$  be the dominance boundary of  $o_i$  computed by our ownership analysis. Let  $C$  be any client of  $Cls$ ,  $C \in AllClients(Int)$ , let  $E_C$  be any execution of  $C$ , and let  $Og_{EC}^r$  be the run-time object graph for  $E_C$ . Let  $o_i^r \in Og_{EC}^r$  be any run-time object represented by  $o_i \in Og$ .*

*For every path  $p: o_i^r \rightarrow \dots \rightarrow o_j^r \in Og_{EC}^r$ , such that the representative of  $p$  is in  $Boundary(o_i)$ , we have that  $o_i^r$  dominates  $o_j^r$  in  $Og_{EC}^r$ .*

Informally, the lemma states that the computed static boundary of  $o_i$  (under) approximates the run-time dominance boundary of every object  $o_i^r$  represented by  $o_i$ .

Consider our running example in Figures 1 and 2. The dominance boundary of object  $o_{BaseRecordManager}$ , denoted by  $Boundary(o_{BaseRecordManager})$ , includes  $o_{BaseRecordManager}$  and  $o_{PRowIdManager}$ , and the edge between them. Clearly, for every execution of the client in Figure 2, the  $o_{BaseRecordManager}$  object created in `main` dominates the  $o_{PRowIdManager}$  object.

The dominance boundary of `root` includes the entire  $Og$ . Clearly, for every execution of the client, `root` dominates the objects created during the execution. The dominance boundaries of the other objects are singleton nodes.

### 5.4.2 Minimal Boundary Analysis

In addition to the dominance boundary analysis, we employ minimal boundary analysis which computes *minimal boundary* information. Intuitively, given an edge  $o_i \rightarrow o_j \in Og$  the minimal boundary analysis finds the minimal dominance boundaries that enclose the edge — roughly, if a boundary  $Boundary(o_k)$  is a minimal boundary that includes  $o_i \rightarrow o_j$ , then every other boundary that includes this edge, say  $Boundary(o'_k)$ , is larger than  $Boundary(o_k)$  (i.e., we have  $Boundary(o'_k) \supset Boundary(o_k)$ ). For example, consider edge  $o_{BaseRecordManager} \rightarrow o_{PRowIdManager}$  in the object graph in Figure 3. There are two boundaries that include this edge,  $Boundary(o_{BaseRecordManager})$  and

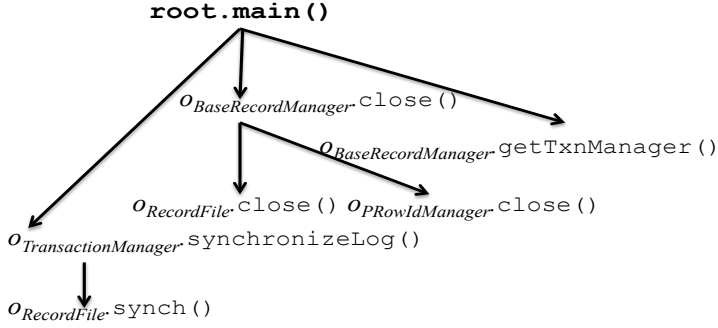


Figure 4. Method sequence graph.

$Boundary(\text{root})$ .  $Boundary(o_{BaseRecordManager})$  is the minimal boundary ( $Boundary(\text{root})$  is larger).

The minimal boundary analysis uses  $Og$  and boundary information, takes as input edge  $o_i \rightarrow o_j$  and computes set  $minBoundaries(o_i \rightarrow o_j)$  where  $minBoundaries(o_i \rightarrow o_j)$  contains the roots of the actual minimal boundaries. In our example,  $minBoundaries(o_{BaseRecordManager} \rightarrow o_{PRowIDManager}) = \{o_{BaseRecordManager}\}$ .

The following lemma holds:

**Lemma 5.2** Let  $o_i^r \rightarrow o_j^r$ , represented by  $o_i \rightarrow o_j$ , be an edge in some  $Og_{EC}^r$ . Let  $minBoundaries(o_i \rightarrow o_j)$  be the set computed by our analysis. There exists  $o_k^r \in Og_{EC}^r$  represented by  $o_k \in Og$ , such that (1)  $o_k \in minBoundaries(o_i \rightarrow o_j)$  and (2) the representative of every path  $o_k^r \rightarrow^* o_i^r \rightarrow o_j^r$  is in  $Boundary(o_k)$ .

Informally, the lemma states that set  $minBoundaries(o_i \rightarrow o_j)$  "covers" each run-time edge  $o_i^r \rightarrow o_j^r$  — that is, we consider at least one boundary that covers this edge. The proof of this lemma is presented in [11].

The two analyses are needed by the classification inference analysis. The correctness results help establish the correctness result for the classification inference analysis.

## 5.5 Method Sequence Analysis

Method sequence analysis approximates the run-time method sequence graphs over executions of clients  $C \in AllClients(Int)$ .

Recall from Section 3.2 the definition of the run-time method sequence graph for an execution of a client  $C \in AllClients(Int)$ . The method sequence analysis constructs the static method sequence graph,  $Og^+$ . The nodes in  $Og^+$  are tuples  $o_i.m_i()$  (the tuples are formed with analysis objects), and the edges represent the transfer of control between distinct objects.  $Og^+$  is a safe approximation of the run-time method sequence graphs over all executions of all clients  $C \in AllClients(Int)$ : if there is an execution that exhibits edge  $o_i^r.m_i() \rightarrow o_j^r.m_j() \in Og_{EC}^r$ , then there is a representative edge  $o_i.m_i() \rightarrow o_j.m_j() \in Og^+$ .

The method sequence analysis uses  $Og$  and outputs the method sequence graph  $Og^+$ . Again, for brevity, we do not present this analysis; it is presented in [11].

The method sequence graph  $Og^+$  for the running example in Figures 1 and 2 is shown in Figure 4. Constructors are omitted for brevity.

## 6 Classification Inference Analysis

We are now ready to define our classification inference analysis. The analysis uses the method sequence graph  $Og^+$ , dominance boundary information,  $Boundary$ , and minimal boundary information,  $minBoundaries$ . It outputs sets  $\mathcal{C}$ ,  $\mathcal{O}$  and  $\mathcal{D}$  which approximate central, owned and distributed objects over all executions of all clients  $C \in AllClients(Int)$ .

### 6.1 Analysis Description

The analysis is shown in Figure 5. It is a breadth-first search on  $Og^+$  starting at tuple  $\text{root.main}()$ . The analysis examines all objects  $o$  such that a tuple with receiver  $o$ , say  $o.m()$ , is reachable in  $Og^+$  from  $\text{root.main}()$  (in other words, the analysis examines the representatives of all shared objects). Lines 5-11 represent the "visit" functionality of the breadth-first-search. Lines 12-13 mark the new tuple as visited and add it at the end of worklist  $WL$ .

Lines 5-6 identify the directly shared objects (i.e., the central objects). Since this is a bread-first-search, it is guaranteed that set  $\mathcal{C}$  will be filled before the search considers objects that fall into  $\mathcal{O}$  and  $\mathcal{D}$ . Lines 7-11 identify the indirectly shared objects (i.e., the owned and distributed objects). The analysis checks the conditions for indirectly shared objects specified in Section 4.1:  $o_j$  must not be a directly shared object (i.e.,  $o_j \notin \mathcal{C}$ ) and  $o_j$  must be a transitively reachable field of a directly shared object  $o_k$ . Subsequently, the analysis groups the indirectly shared objects into owned and distributed: if  $\text{root}$  is not in the minimal boundary of the incoming edge  $o_i \rightarrow o_j$ , then  $o_j$  represents owned objects; otherwise, that is, if  $\text{root}$  is in the minimal boundary of the incoming edge  $o_i \rightarrow o_j$ , then  $o_j$  represents distributed objects. Note that one object  $o_j$  may represent both owned and distributed objects — it may be owned on an incoming edge  $o_i' \rightarrow o_j$ , and distributed on a different incoming edge  $o_i'' \rightarrow o_j$ .

### 6.2 Example

Consider our running example and the method sequence graph in Figure 4. The analysis examines edge

$\text{root.main}() \rightarrow O_{TransactionManager}.synchronizeLog()$ .

The edge originates at  $\text{root.main}()$  and at line 6  $O_{TransactionManager}$  is added to  $\mathcal{C}$ . Subsequently (at lines 12-13), tuple  $O_{TransactionManager}.synchronizeLog()$  is marked as visited and is added at the end of the worklist. Next, the analysis examines edges

$\text{root.main}() \rightarrow O_{BaseRecordManager}.close()$  and

procedure *inferClassification*

**globals**  $Og^+$ , *Boundary*, *minBoundaries*

**output** sets  $\mathcal{C} \subseteq O$ ,  $\mathcal{O} \subseteq O$  and  $\mathcal{D} \subseteq O$

```

[1]  $WL = \{\text{root.main}()\}$ 
[2] while  $WL$  is not empty
[3]   take  $o_i.m_i()$  from  $WL$ 
[4]   foreach  $o_i.m_i() \rightarrow o_j.m_j() \in Og^+$ 
[5]     if  $o_i == \text{root}$ 
[6]       add  $o_j$  to  $\mathcal{C}$ 
[7]     else if  $o_j \notin \mathcal{C} \wedge \exists o_k \in \mathcal{C}$  s.t.  $o_k \xrightarrow{f} o_j \in Og$ 
[8]       if  $\text{root} \notin \text{minBoundaries}(o_i \rightarrow o_j)$ 
[9]         add  $o_j$  to  $\mathcal{O}$ 
[10]      else
[11]        add  $o_j$  to  $\mathcal{D}$ 
[12]      if  $o_j.m_j()$  not visited
[13]        mark  $o_j.m_j()$  as visited, add  $o_j.m_j()$  at end of  $WL$ 

```

**Figure 5. Classification inference analysis.**

$\text{root.main}() \rightarrow OBaseRecordManager.getTxnManager()$ .

Object  $OBaseRecordManager$  is added to  $\mathcal{C}$ ; tuples  $OBaseRecordManager.close()$  and  $OBaseRecordManager.getTxnManager()$  are marked as visited and added at the end of the worklist.

Eventually,  $OTransactionManager.synchronizeLog()$  is taken from the worklist. The analysis examines edge

$OTransactionManager.synchronizeLog() \rightarrow ORecordFile.synch()$ .

At line 7 the analysis examines object  $ORecordFile$ : it is not in  $\mathcal{C}$  and it is a field of two central objects,  $OBaseRecordManager$  and  $OTransactionManager$ . The analysis proceeds to determine that  $\text{minBoundaries}(OTransactionManager \rightarrow ORecordFile)$  equals  $\{\text{root}\}$ . Thus, the "closest owner" of  $ORecordFile$  is  $\text{root}$  (i.e., there is no owner  $o_k$  that could protect  $ORecordFile$ ). The analysis determines that  $ORecordFile$  is distributed and adds it to  $\mathcal{D}$ .

Eventually,  $OBaseRecordManager.close()$  is taken from the worklist. As a result, the analysis examines edge

$OBaseRecordManager.close() \rightarrow ORowIdManager.close()$ .

Again, at line 7 the analysis examines  $ORowIdManager$  and determines that it is not in  $\mathcal{C}$  and it is a field of one central object, namely  $OBaseRecordManager$ . The analysis proceeds to determine that  $\text{minBoundaries}(OBaseRecordManager \rightarrow ORowIdManager)$  equals  $\{OBaseRecordManager\}$ . Clearly, the edge is in the dominance boundary of  $OBaseRecordManager$  and the owner object  $OBaseRecordManager$  can protect  $ORowIdManager$ . The algorithm determines that  $ORowIdManager$  is owned and adds it to  $\mathcal{O}$ .

The analysis proceeds until the worklist  $WL$  is empty. The final result is:

$\mathcal{C} = \{OBaseRecordManager, OTransactionManager\}$ ,

$\mathcal{O} = \{ORowIdManager\}$ ,  $\mathcal{D} = \{ORecordFile\}$ .

### 6.3 Analysis Correctness

The analysis overapproximates central and distributed objects, and underapproximates owned objects. Informally, if the analysis determines that an object  $o$  is distributed, then this means that some of the run-time objects  $o^r$  represented by  $o$  may be distributed; however, it may be the case that all  $o^r$  are owned. Again informally, if the analysis determines that an object  $o$  is owned, this essentially means that all run-time objects  $o^r$  represented by  $o$  are owned.

More formally, the following theorem holds:

**Theorem 6.1** *Let  $o_i \rightarrow o_j$  be any edge in  $Og$  s.t.  $o_i \rightarrow o_j$  is examined at line 8 in Figure 5 and  $o_j$  is added to  $\mathcal{O}$  at line 9. For every client  $C \in \text{AllClients}(\text{Int})$ , execution  $E_C$  of  $C$  and edge  $o_i^r \rightarrow o_j^r \in Og_{E_C}^r$  (i.e.,  $o_i^r \rightarrow o_j^r$  is represented by  $o_i \rightarrow o_j$ ), we have that if  $o_j^r \in \mathcal{IS}_{E_C}$ , then  $o_j^r \in \mathcal{O}_{E_C}$ .*

We provide a sketch of the proof. By Lemma 5.2 we have that there exists  $o_k^r \in Og_{E_C}^r$  such that (1) its representative  $o_k$  is in  $\text{minBoundaries}(o_i \rightarrow o_j)$  and (2) the representative of every path from  $o_k^r$  to  $o_i^r \rightarrow o_j^r$  is in  $\text{Boundary}(o_k)$ . Clearly,  $o_k^r$  is not  $\text{root}$  (or the test at line 8 would have failed). By (2) and Lemma 5.1 we have that  $o_k^r$  dominates  $o_i^r$  and  $o_j^r$  in  $Og_{E_C}^r$ .

Suppose that there exists a path  $\text{root.main}() \rightarrow^+ o_j^r.m_j() \in Og_{E_C}^{r+}$  such that the path does not go through some  $o_k^r.m_k()$  (i.e., it does not go through a method executed on  $o_k^r$ ). Then there exists a path  $\text{root} \rightarrow^+ o_j^r \in Og_{E_C}^r$  such that the path does not go through  $o_k^r$ . This contradicts the established fact that  $o_k^r$  dominates  $o_j^r$ .

Thus, every path  $\text{root.main}() \rightarrow^+ o_j^r.m_j() \in Og_{E_C}^{r+}$  goes through some  $o_k^r$ . Since  $o_j^r \in \mathcal{IS}_{E_C}$ , then  $o_j^r \in \mathcal{O}_{E_C}$ .

## 7 Empirical Results

The analysis is implemented in Java using Soot 2.2.3 [20] and Spark [9]. It uses the Andersen-style points-to analysis provided by Spark. We performed the 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 1400MB.

We used several publicly available open concurrent Java programs: `jdbm 1.0`, `jdbf`, `jtlds 1.2` and `pool 1.2`. Information about these programs is given in Table 1. Column 2 describes the benchmark, and Column 3 gives the number of methods determined as reachable by the points-to analysis in Spark. These programs were used in previous work on static race detection [12].

### 7.1 Results

Table 1 shows the results of our inference. Columns 4, 5 and 6 list the sizes of sets  $\mathcal{C}$ ,  $\mathcal{O}$  and  $\mathcal{D}$  for each of our

Program	Description	#Methods	$\mathcal{C}$	$\mathcal{O}$	$\mathcal{D}$	Protected $\mathcal{O}$	Analysis time
jdbm	persistence engine	4904	21(32%)	16(23%)	32(46%)	12(75%)	177 seconds
jdbf	mapping system	6383	15(23%)	30(47%)	19(30%)	18(60%)	449 seconds
pool	Apache Commons pool	3982	35(70%)	3(6%)	12(24%)	0(0%)	95 seconds
jtds	JDBC driver	5980	51(15%)	81(24%)	210(61%)	43(53%)	1097 seconds
Average			35%	25%	40%	47%	

**Table 1. Benchmarks and analysis results.**

benchmarks. On average 35%, of the analysis objects are found to be central, 25% are found to be owned and 40%, are found to be distributed.<sup>4</sup>

Our results show that the majority of analysis objects, about 60%, are central or owned, and therefore “easy to reason about”. However, a significant percentage, 40%, are distributed, and therefore “difficult to reason about”.

In addition, we looked at what fraction of the owned objects were protected by synchronization on an owner. The results show that a significant percentage, on average almost 50% of all owned objects are protected by synchronization on an owner (Column “Protected  $\mathcal{O}$ ” in Table 1).<sup>5</sup>Therefore, ownership does play a significant role in concurrent programs. We claim that this role should be enhanced — well-written multithreaded programs should strive for a maximal number of owned objects and a minimal number of distributed objects.

The running time of the analysis is shown in the last column of Table 1; it includes the times for Soot and Spark, as well as the times for the analyses from Sections 5 and 6.

## 7.2 Case Study: jdbm

We performed a detailed case study on one of our benchmarks, jdbm. The case study addresses two questions. First, what is the precision of the analysis — that is, are the reported central and distributed objects indeed central and distributed? Second, are the central, owned and distributed objects safe or unsafe from object and data races; are central and owned objects “easy to reason about” and distributed objects “difficult to reason about”?

In order to address the first question, we examined the set of objects reported as central, and the set of objects reported as distributed. For each object  $o \in \mathcal{C}$  we attempted to construct a client  $C \in AllClients(Int)$  such that a run-time object  $o^r$  represented by  $o$  is in  $\mathcal{C}_{EC}$  for some execution of

<sup>4</sup>Note that in terms of absolute numbers, our analysis is very conservative — that is, it creates a large number of central objects, and in turn these central objects create a large number of owned and distributed objects (recall that each public class in  $Cls$  is instantiated in the conservative client created by the fragment analysis). In typical clients, there would be a smaller number of central objects, and in turn there would be a smaller number of owned and distributed objects.

<sup>5</sup>This experiment underestimates protection; it considers protection due to synchronized methods (e.g., `synchronized void close() { . . . }`) but ignores protection due to synchronized blocks. The role of ownership is in fact even greater than reported.

$C_{EC}$ . For each of the 21 objects in  $\mathcal{C}$ , we were able to find such a client  $C$ . Similarly, for each object  $o \in \mathcal{D}$ , we attempted to construct a client  $C \in AllClients(Int)$  such that a run-time object  $o^r$  represented by  $o$  is in  $\mathcal{D}_{EC}$  for some execution of  $C_{EC}$ . For each of the 32 objects in  $\mathcal{D}$ , we were able to find such a client  $C$ . Therefore, the analysis captures the classification precisely.

In order to address the second question, we examined the set of objects reported as central, the set of objects reported as distributed, and the set of objects reported as unprotected owned. For each object  $o \in \mathcal{C}$  we attempted to construct a multithreaded client  $C'$  such that a run-time object  $o^r$  represented by  $o$  is accessed by multiple threads, and this access can lead to unsafe behavior.<sup>6</sup> For 19 out of the 21 objects, we were able to construct such a client  $C'$ . Similarly, for each  $o \in \mathcal{D}$  we attempted to construct a multithreaded client  $C'$  such that a run-time object  $o^r$  represented by  $o$ , is accessed by multiple threads and the access can lead to unsafe behavior. For 20 out of the 32 objects in  $\mathcal{D}$  we were able to construct a client which exposes unsafe behavior. Finally, for each unprotected owned object  $o$  we attempted to construct a client that exposes unsafe behavior. For each of the 4 objects, we were able to construct such a client. The examination revealed many potential data races, including the data race on `BaseRecordManager` reported in [12].

The case study indicates that owned objects tend to be safe (12 out of 16 objects were protected by synchronization on an owner object), while distributed objects tend to be unsafe (20 out of 32 objects were unsafe).

The case study confirms our hypothesis that central and owned objects are “easy to reason about”, while distributed objects are “difficult to reason about”. It took approximately 20 minutes to examine and construct clients for the 25 central and owned objects. In contrast, it took approximately 5 hours, on two different days, to examine and construct clients for the 32 distributed objects; it was rarely obvious how the client can access a distributed object, and a detailed examination of almost the entire code base of jdbm was needed.

We view these results as very promising. The classifica-

<sup>6</sup>By unsafe behavior we mean that there is an execution of  $C', E_{C'}$ , such that with appropriate thread scheduling,  $E_{C'}$  leads to an object race on  $o^r$ , and the object race triggers a data race on  $o^r$  (i.e., data race on location  $o^r.f$ ), or a data race on a field of  $o^r$  (i.e., on location  $o^r.f.g$ ).

tion inference analysis is practical and precise and therefore it can be integrated in program understanding tools. Also, it can lead to better techniques for concurrency error detection due to the search space reduction due to ownership.

## 8 Related Work

Concurrency is a very active area of research. The vast majority of this work falls into one of two categories: (1) *prevention* and (2) *detection* of concurrency errors.

Work on prevention focuses on type systems (e.g., [2, 1, 7, 14]). These type systems disallow concurrency errors such as data races and deadlocks at the language level. They typically require extension of the language, compiler and run-time, as well as significant amount of annotations. Our approach works on existing languages and does not require annotations by the programmer. It is focused on uncovering the structure of shared objects and on understanding concurrency and sharing in real-world Java programs.

There is a large body of work on detection of concurrency errors. Work on detection includes dynamic approaches (e.g., [23, 19, 8]), and static approaches (e.g., [15, 3, 6, 12]). Our analysis does not focus on error detection, but on the problem of uncovering the structure of shared objects in concurrent programs. To the best of our knowledge, this is the first work that characterizes sharing in concurrent programs.

Ownership types prevent certain aliasing; there are many proposals in the literature [4, 2, 5], including ones that exploit ownership for prevention of concurrency errors [2, 1]. Our ownership analysis focuses on inference and the study of real world programs, not on type systems.

Recent work by Vaziri et al. [21, 22] focuses on data-centric synchronization; it argues that in object-oriented languages synchronization is naturally attached to atomic sets of objects and fields. Similarly to our work, this work argues that ownership plays an important role. Unlike our work, it studies language-based approaches to data-centric synchronization — i.e., programmers must provide atomicity and ownership annotations. Our work focuses on understanding and classification of shared objects.

## References

- [1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
- [2] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, pages 56–69, 2001.
- [3] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.
- [4] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
- [5] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.
- [6] D. Engler and K. Ashcraft. Racex: Effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–253, 2003.
- [7] C. Flanagan and S. Freund. Type-based race detection for java. In *PLDI*, pages 219–232, 2000.
- [8] C. Flanagan, S. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, pages 293–303, 2008.
- [9] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, pages 153–169, 2003.
- [10] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.
- [11] A. Milanova and Y. Liu. Practical static ownership inference. Technical Report RPI/DCS-09-04, Rensselaer Polytechnic Institute, Dec. 2009.
- [12] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI*, 2006.
- [13] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, pages 80–89, 1998.
- [14] P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *PLDI*, pages 320–331, 2006.
- [15] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.
- [16] A. Rountev. Precise identification of side-effect free methods. In *ICSM*, pages 82–91, 2004.
- [17] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.
- [18] 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.
- [19] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, 2008.
- [20] 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.
- [21] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, pages 334–345, 2006.
- [22] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, 2010.
- [23] C. von Praun and T. Gross. Static conflict analysis for multithreaded object-oriented programs. In *PLDI*, pages 115–128, 2003.