Static Analysis for Understanding Shared Objects in Open Concurrent Java Programs

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

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

***Abstract*—Concurrent programming with shared memory in an object-oriented language such as Java is notoriously difficult. Therefore, it is important to study new program understanding techniques for concurrent object-oriented languages.**

This paper studies shared objects in open concurrent Java programs. First, it proposes a classification of shared objects into three categories: *central*, *owned* and *distributed*. Second, it presents a new static analysis that infers central, owned and distributed objects, as well as experiments with the analysis.

I. Introduction

Concurrent programming in an object-oriented language with shared memory such as Java, is complex and challenging. This is due to the complex structure of shared objects, as well as the complex dynamics of interactions between different threads as they access these shared objects.

Somewhat surprisingly, little has been done on analysis and tools for understanding concurrent Java programs, or more specifically on analysis and tools for understanding the complex structure and behavior of shared objects in concurrent Java programs. Such analysis and tools can enhance program understanding, a task notoriously difficult and challenging; furthermore, they can lead to more effective language-based techniques for concurrency control. Therefore, it is important to develop and study new techniques in this direction.

This paper studies shared objects in open concurrent Java programs (i.e., Java libraries, or other incomplete Java programs, designed for use by multithreaded clients).

The first contribution of the paper is a classification of the 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, accesses to it are distributed through multiple objects.

The second contribution of the paper is a novel static analysis that infers central, owned and distributed objects. The analysis builds upon a general-purpose ownership inference analysis [11].

An important novelty of our program understanding technique is that it is *data-centric* — it emphasizes objects and the structure of these objects, rather than control flow. Similarly to data-centric synchronization [22], which argues that synchronization is more naturally attached to data (i.e., an object viewed as a set of data fields), rather than to individual methods, we argue that program understanding of object-oriented programs is more natural if it is driven by the objects and their structure, rather than by the method invocations.

Our technique classifies shared objects into two categories: *easy to reason about* and *difficult to reason about*. We conjecture that central and owned objects are easy to reason about — that is, it is easy to understand the behavior of a central or an owned object. On the other hand, distributed objects are difficult to reason about — that is, it is difficult to understand the behavior of a distributed object. Ownership, which is a relatively strong form of encapsulation, is important and desirable in data-centric program understanding. It confines reasoning about an object's behavior. Informally, if an object is owned, then reasoning is confined to a small set of objects, namely the boundary of the owner. In contrast, if an object is distributed, reasoning is not confined; in fact, it often spans the entire set of objects in the program.

We have implemented the analysis and present preliminary experiments on 4 relatively large publicly available open concurrent Java programs, as well as a case study on one of the programs, `jdbm`. The experiments and the case study are summarized by the following findings:

(1) The analysis infers central, owned and distributed objects precisely and efficiently. Therefore, we conjecture that the analysis can be incorporated in reverse-engineering and program understanding tools.
(2) Central and owned objects indeed appear to be easy to reason about, while distributed objects appear to be difficult to reason about. Therefore, we conjecture that the analysis can enhance program understanding tasks.
(3) Ownership appears to play an important role in real-world concurrent programs. This finding supports the idea of data-centric program understanding.

The paper has the following contributions:

- It proposes a classification of shared objects into *central*, *owned*, and *distributed*.
- It proposes a novel static analysis that classifies shared objects into central, owned and distributed.

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

```
class ClientThread extends Thread {
  static BaseRecordManager brm;
  static TransactionManager tm;
 public static void main() {
  brm = new BaseRecordManager();
  tm = brm.getTxnManager();
  (new ClientThread()).start();
  (new ClientThread()).start();
 }
 public void run() {
  brm.close()
  tm.synchronizeLog();
 }
}
```

Fig. 1. A single-threaded client and the corresponding multithreaded client.

- It presents preliminary experiments with the analysis on several publicly available open concurrent Java programs.

II. Problem Setting

We consider a set *Cls* of interacting classes that constitute an open concurrent program (i.e., a Java library of classes intended for use by multithreaded clients). 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, excluding dynamically loaded classes). 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 2 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 1 (class `Client`).

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. This is illustrated in Figure 1 which juxtaposes a single-threaded client and its corresponding multithreaded client. The `BaseRecordManager` objects and the `TransactionManager` object accessed in `main` in the single-threaded client, are written into static fields in the multithreaded client. Methods `close` and `synchronizeLog` called in `main` in the single-threaded client are directly called by each of the two threads spawned in the multithreaded client.

We classify the run-time objects created during an execution of a client $C \in AllClients(Int)$ into central, owned and distributed. The static analysis represents the run-time objects with a finite set of analysis objects O; each run-time object in an execution of some C corresponds to an object in O (roughly, each run-time object is represented by its allocation site). The goal of the static analysis is to classify the analysis objects in set O into central, owned and distributed.

III. Views of Program Execution

This section presents two views of a program execution, the *run-time object graph* (Section III-A), and the *run-time method sequence graph* (Section III-B). Section IV uses these views to define the classification of shared objects into central, owned and distributed.

The object graph represents a structural view of the execution — it shows the access relationships between objects and is used to reason about structural 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.

The views of program execution are necessary in order to define precisely the dynamic semantics of central, owned and distributed. Our static analysis, presented later in the paper, infers approximations of these graphs, and proceeds to infer central, owned and distributed analysis 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.

As mentioned earlier, the analysis represents objects by their allocation site: for example, all run-time objects allocated at allocation 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.

A. 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 $Og^r_{E_C}$ be the run-time object graph for this execution. $Og^r_{E_C}$ is constructed as follows:

- There is a field edge $o^r \xrightarrow{f} o^r_1$ in $Og^r_{E_C}$ if at some point of the execution E_C, field f of object o^r refers to object o^r_1.
- Similarly, there is an array edge $o^r \xrightarrow{[]} o^r_1$ in $Og^r_{E_C}$ if o^r is an array object, and at some point of the execution E_C, o^r has element o^r_1.
- There is an unlabeled edge $o^r \rightarrow o^r_1$ in $Og^r_{E_C}$ if at some point of the execution E_C, an instance method invoked on receiver object o^r has local variable l, $l \neq$ `this`, that refers to object o^r_1 [1].

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^r_1, then there is an edge `root` $\rightarrow o^r_1 \in Og^r_{E_C}$.

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

B. 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, $p = q.f$, $q \neq$ `this` and $p.f = q$, $p \neq$ `this`), and array accesses (i.e., $p = q[i]$ and $p[i] = q$) as special method calls. Notation $o^r_2.rd$ denotes the execution of a read $p = q.f$, $q \neq$ `this` where q refers to o^r_2. Similarly, $o^r_2.wr$ denotes the execution of a write $p.f = q, p \neq$ `this` where p refers to o^r_2.

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

- There is an edge $o^r_1.m_1() \rightarrow o^r_2.m_2() \in Og^{r+}_{E_C}$ if at some point of the execution E_C, method m_1 invoked on receiver o^r_1 executes a call $p.m_2(), p \neq$ `this` in m_1, where p refers to o^r_2, and the call leads to the invocation of method m_2 on receiver o^r_2.
- There is an edge $o^r_1.m'_1() \rightarrow o^r_2.m_2() \in Og^{r+}_{E_C}$ if at some point of the execution E_C, method m'_1 invoked on receiver o^r_1 has executed a call `this`.$m_1()$ in m'_1, and the call `this`.$m_1()$ has resulted in edge $o^r_1.m_1() \rightarrow o^r_2.m_2() \in Og^{r+}_{E_C}$.

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^r_1 directly calls m_2 on receiver o^r_2 through call site $p.m_2()$, which directly results in edge $o^r_1.m_1() \rightarrow o^r_2.m_2() \in Og^{r+}_{E_C}$. In the second case, method m_1 "jumps through" calls through `this` until it reaches a call not through `this`: for example, if there is a method m'_1 invoked on receiver o^r_1 and m'_1 executes a call `this`.$m_1()$, then in turn m_1 executes a call $p.m_2()$, $p \neq$ `this`, where p refers to o^r_2, then there is an edge $o^r_1.m'_1() \rightarrow o^r_2.m_2()$ in the method sequence graph. Skipping through calls through `this` is needed in order to emphasize transfer of control between *distinct* objects.

Throughout the paper we use the standard notation for reachability. E.g., $o^r_1.m_1() \rightarrow^* o^r_2.m_2() \in Og^{r+}_{E_C}$ denotes a path of zero or more edges in $Og^{r+}_{E_C}$ from $o^r_1.m_1()$ to $o^r_2.m_2()$; similarly, $o^r_1.m_1() \rightarrow^+ o^r_2.m_2() \in Og^{r+}_{E_C}$ denotes a path of one or more edges.

IV. Classification of Shared Objects

A. Defining the Classification

Let $C \in AllClients(Int)$ be a client of *Cls* as described in Section II, 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 $\mathcal{S}_{E_C}$ is the union of the set of *directly shared* objects denoted by $\mathcal{DS}_{E_C}$, and the set of *indirectly shared* objects denoted by $\mathcal{IS}_{E_C}$.

The set of directly shared objects is defined as follows:

$$\mathcal{DS}_{E_C} = \{o^r \mid \exists \texttt{root.main()} \rightarrow o^r.m() \in Og^{r+}_{E_C}\}.$$

The definition states that there exists an edge in $Og^{r+}_{E_C}$ from `root.main()` to a tuple $o^r.m()$ (i.e., `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:

$$\mathcal{IS}_{E_C} = \{o^r \mid o^r \notin \mathcal{DS}_{E_C} \wedge \exists o^r_j \text{ s.t. } \texttt{root} \rightarrow o^r_j \xrightarrow{f}{}^{+} o^r \in Og^r_{E_C} \wedge \exists p : \texttt{root.main()} \rightarrow^+ o^r.m() \in Og^{r+}_{E_C}\}.$$

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

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

The second condition, $\exists o_j^r$ s.t. root $\rightarrow o_j^r \xrightarrow{f}{}^{+} o^r \in Og_{E_C}^r$, states that o^r must be a transitively reachable field of some directly object o_j^r directly accessible to root. 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, $\exists p$: 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 root.main() to $o^r.m()$ (i.e., the execution of the client's 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:

$$\mathcal{S}_{E_C} = \mathcal{DS}_{E_C} \cup \mathcal{IS}_{E_C}.$$

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 $\mathcal{C}_{E_C}$, is the set of directly shared objects:

$$\mathcal{C}_{E_C} = \mathcal{DS}_{E_C}.$$

The set of *owned* objects in E_C, denoted by $\mathcal{O}_{E_C}$ is defined as follows:

$$\begin{aligned}\mathcal{O}_{E_C} = \{o^r \mid o^r \in \mathcal{IS}_{E_C} \wedge \\ \exists o_1^r \text{ s.t. } \forall p : \texttt{root.main()} \rightarrow^+ o^r.m() \in Og_{E_C}^{r+} \\ p \text{ is } \texttt{root.main()} \rightarrow^+ o_1^r.m_1() \rightarrow^+ o^r.m()\}.\end{aligned}$$

The definition states that an object $o^r \in \mathcal{S}_{E_C}$ 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.

Note that this definition of *owned* only requires that there exists a dominating owner object o_1^r; it is not concerned with who this owner is (in general, an owned object o^r has an immediate dominator and several other dominators, including root). Although the definition can be easily refined to designate the immediate dominator as the owner of o^r, we have chosen not to do so in order to keep the definition of owned as simple as possible.

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.

```
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();
  }
}
```

Fig. 2. BaseRecordManager from jdmb.

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

Fig. 3. Client of BaseRecordManager.

B. Example

We illustrate the classification with jdbm. Figure 2 shows excerpts from classes BaseRecordManager, RecordFile and TransactionManager in jdbm.

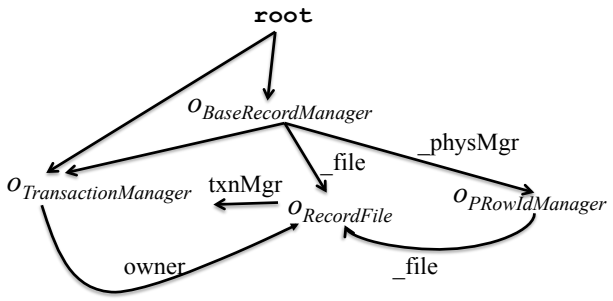

Fig. 4. Object graph.

Figure 3 shows a client of these classes which meets the constraints outlined in Section II.

Figure 4 shows the object graph corresponding to Figures 2 and 3. 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 3) accesses objects $o_{RecordFile}$ (created at site 1 in Figure 2), $o_{PRowIdManager}$ (created at site 2 in Figure 2), and $o_{TransactionManager}$ (created at site 3 in Figure 2); 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 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()` $\rightarrow o_{BaseRecordManager}$`.getTxnManager()`.

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

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

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

`root.main()` $\rightarrow o_{BaseRecordManager}$`.close()`
$\rightarrow o_{PRowIdManager}$`.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 central objects:

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

`root.main()` $\rightarrow o_{BaseRecordManager}$`.close()`
$\rightarrow o_{RecordFile}$`.close()`.

C. Discussion

Central objects are the objects potentially directly accessed by multiple threads (e.g., $o_{BaseRecordManager}$ 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., $o_{PRowIdManager}$ is hidden behind central object $o_{BaseRecordManager}$). Other objects become accessible to other central objects (e.g., $o_{RecordFile}$ is created by central object $o_{BaseRecordManager}$ but eventually it becomes accessible to central object $o_{TransactionManager}$ 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.

The important benefit from this classification is that it provides a simple and intuitive classification of the shared objects (often these objects are part of complex structures). It classifies the shared objects into two categories: *easy to reason about* and *difficult to reason about* objects; a program understanding task, manual or automatic, can examine easy objects quickly, and focus on difficult objects thereafter.

Consider the task of reasoning about object races [23].[2]

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 in `main`, say $o_c.m_1()$ and $o_c.m_2()$, where one of m_1 or m_2 is unsynchronized, creates an 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 $o_{PRowIdManager}$ is protected by synchronization on its owner $o_{BaseRecordManager}$; every thread holds the lock on owner $o_{BaseRecordManager}$ when accessing $o_{PRowIdManager}$, and therefore, it is guaranteed that no object race would occur on $o_{PRowIdManager}$. Second, even if the owned object is not protected by synchronization on an owner o_c, reasoning about object races is simplified by ownership: the task 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 *excelusively* within the ownership boundary of o_c. Such structured reasoning 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, reasoning about object races on o_d will need to traverse the entire set of objects which could be quite large. In our example, $o_{RecordFile}$ is a distributed object — it can be accessed along several paths: e.g., `root` $\rightarrow o_{BaseRecordManager} \rightarrow o_{PRowIdManager} \rightarrow o_{RecordFile}$.

[2]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.

V. Preliminary Analyses

Recall that the goal of our static analysis is to classify analysis objects in O into central, owned and distributed. Recall also that the dynamic semantics of central, owned and distributed is defined in terms of the run-time object graph and the run-time method sequence graph.

This section briefly describes the analyses that infer conservative approximations of the these graphs. Section VI presents the classification inference analysis which uses these approximations to infer central, owned and distributed analysis objects.

There are five preliminary analyses — fragment analysis (Section V-A), points-to analysis (Section V-B), object graph analysis (Section V-C), ownership analysis (Section V-D), and method sequence analysis (Section V-E). This section only outlines the analyses and states analyses properties important for the classification inference analysis; the analyses are described in detail elsewhere [18], [17], [11].

A. 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 3 instead of the conservative client. Our implementation uses conservative clients created according to the rules of the fragment analysis [18].

B. 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].[3]

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. 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].

C. Object Graph Analysis

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

Recall from Section III-A 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 $Og^r_{E_C}$ 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 2 and its client in Figure 3, is shown in Figure 4.

D. Ownership Analysis

The ownership analysis consists of two parts: dominance boundary analysis (Section V-D1), and minimal boundary analysis (Section V-D2).

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 $Og^r_{E_C}$ be the run-time object graph for E_C. Let $o_i^r \in Og^r_{E_C}$ be any run-time object represented by $o_i \in Og$.

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

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 2 and 3. 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 3, 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. For example, $Boundary(o_{TransactionManager})$=$\{o_{TransactionManager}\}$

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

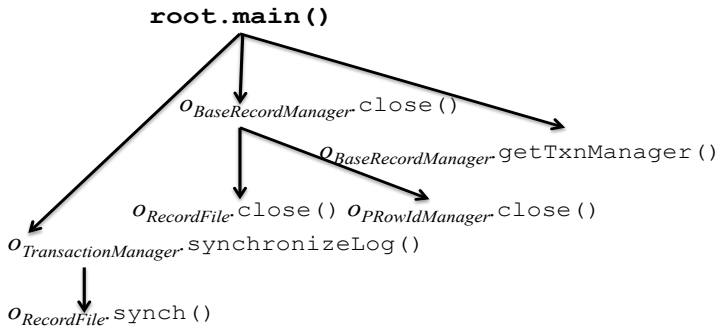

Fig. 5. Method sequence graph.

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 4. There are two boundaries that include this edge, $Boundary(o_{BaseRecordManager})$ and $Boundary(\texttt{root})$. $Boundary(o_{BaseRecordManager})$ is the minimal boundary ($Boundary(\texttt{root})$ 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 $Og^r_{E_C}$. Let $minBoundaries(o_i \rightarrow o_j)$ be the set computed by our analysis. There exists $o_k^r \in Og^r_{E_C}$ 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.

E. Method Sequence Analysis

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

Recall from Section III-B 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 $o_i^r.m_i() \rightarrow o_j^r.m_j() \in Og^{r+}_{E_C}$, 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 2 and 3 is shown in Figure 5.Constructors are omitted for brevity.

VI. 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 $C \in AllClients(Int)$.

A. Analysis Description

The analysis is shown in Figure 6. It is a breadth-first search on Og^+ starting at tuple $\texttt{root.main()}$. The analysis examines all objects o such that a tuple with receiver o, say $o.m()$, is reachable in Og^+ from $\texttt{root.main()}$ (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 IV-A: 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 $\texttt{root}$ 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 $\texttt{root}$ 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 $o''_i \rightarrow o_j$.

B. Example

Consider our running example and the method sequence graph in Figure 5. The analysis examines edge

$\texttt{root.main()} \rightarrow o_{TransactionManager}.\texttt{synchronizeLog()}$.

procedure *inferClassification*
globals Og^+, $Boundary$, $minBoundaries$
output sets $\mathcal{C} \subseteq O, \mathcal{O} \subseteq O$ and $\mathcal{D} \subseteq O$
[1] $WL = \{$`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 $o_i ==$ `root`
[6] add o_j to $\mathcal{C}$
[7] else if $o_j \notin \mathcal{C} \wedge \exists o_k$ s.t. `root` $\stackrel{f}{\rightarrow}^+ o_j \in Og$
[8] if `root` $\notin minBoundaries(o_i \rightarrow o_j)$
[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 it at end of WL

Fig. 6. Classification inference analysis.

The edge originates at `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

`root.main()` $\rightarrow o_{BaseRecordManager}$`.close()` and

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

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

Eventually, $o_{TransactionManager}$`.synchronizeLog()` is taken from the worklist. The analysis examines edge

$o_{TransactionManager}$`.synchronizeLog()` $\rightarrow o_{RecordFile}$`.synch()`.

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

Eventually, $o_{BaseRecordManager}$`.close()` is taken from the worklist. As a result, the analysis examines edge

$o_{BaseRecordManager}$`close()` $\rightarrow o_{PRowIdManager}$`.close()`.

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

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

$\mathcal{C} = \{o_{BaseRecordManager}, o_{TransactionManager}\}$,
$\mathcal{O} = \{o_{PRowIdManager}\}, \mathcal{D} = \{o_{RecordFile}\}$.

C. 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 6 and o_j is added to $\mathcal{O}$ at line 9. For every client $C \in AllClients(Int)$, execution E_C of C and edge $o_i^r \rightarrow o_j^r \in Og^r_{E_C}$ (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 $o_k^r \in Og^r_{E_C}$ such that (1) its representative o_k is in $minBoundaries(o_i \rightarrow o_j)$ and (2) the representative of every path from o_k^r to $o_i^r \rightarrow o_j^r$ is in $Boundary(o_k)$. Clearly, o_k^r is not `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 $Og^r_{E_C}$.

Suppose that there exists a path `root.main()` $\rightarrow^+ o_j^r.m_j() \in Og^{r+}_{E_C}$ 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 `root` $\rightarrow^+ o_j^r \in Og^r_{E_C}$ 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 `root.main()` $\rightarrow^+ o_j^r.m_j() \in Og^{r+}_{E_C}$ goes through some o_k^r. Since $o_j^r \in \mathcal{IS}_{E_C}$, then $o_j^r \in \mathcal{O}_{E_C}$.

VII. Empirical Results

We have implemented a prototype of the classification inference analysis. The prototype is implemented in Java using Soot 2.2.3 [20] and Spark [9]. It uses the Andersen-style points-to analysis provided by Spark and 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`, `jtds` 1.2 and `pool` 1.2. Information about these programs is given in Table I. 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 are the open concurrent programs used in previous work on static race detection [12].

A. Results

Table I 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 benchmarks. If an analysis object was found to be both owned and distributed, it was counted as a member of both $\mathcal{O}$ and

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 I
BENCHMARKS AND ANALYSIS RESULTS.

$\mathcal{D}$ (it is trivial to include a warning about such objects in the results; at this point our analysis does not include such a warning; we plan to incorporate it in the future). 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.[4]

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.

Our results show that a significant percentage, on average 25% of all objects, are owned. In addition, we looked at what fraction of the owned objects were protected by synchronization on an owner. The results show that, on average almost 50% of all owned objects are protected by synchronization on an owner (Column Protected $\mathcal{O}$ in Table I).[5]Therefore, ownership does play an important role in concurrent Java programs. This finding supports the idea of data-centric program understanding: since ownership happens frequently, programmers will benefit from the structured reasoning that ownership entails.

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

B. Case Study: `jdbm`

We performed a 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 central and owned objects indeed easy to reason about and distributed objects indeed 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 $C \in \mathit{AllClients}(\mathit{Int})$ such that a run-time object o^r represented by o is in $\mathcal{C}_{E_C}$ for some execution of $C\ E_C$. 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 $C \in \mathit{AllClients}(\mathit{Int})$ such that a run-time object o^r represented by o is in $\mathcal{D}_{E_C}$ for some execution of $C\ E_C$. 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 considered the task of reasoning about object races that may lead to unsafe behavior.[6] 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. 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. To the best of our understanding of `jdbm`, several of these races, including the data races on `BaseRecordManager` reported in [12], can happen in realistic clients.

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 4 hours 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 promising. The classification inference analysis is practical and precise and therefore it can be integrated in program understanding tools. Also, it can lead to better techniques for program understanding due to the search space reduction due to ownership.

VIII. RELATED WORK

Concurrency is a very active area of research. The vast majority of this work falls into one of two categories: (1)

[4]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.

[5]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.

[6]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 $o^r.f.g$).

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., [24], [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 infers ownership. Also, it focuses on understanding concurrent Java programs while [21], [22] focuses on types for data-centric synchronization.

IX. Conclusion

We presented a novel program understanding technique that studies shared objects in open concurrent Java programs. Our technique classifies shared objects into central, owned and distributed. We have presented a novel static analysis that infers central, owned and distributed objects precisely and efficiently.

There are many opportunities for future work. First, we plan to incorporate the prototype analysis into an Eclipse-based tool that could help support program understanding of concurrent programs. Second, we plan to extend the analysis to the whole-program case, which will enable experimentation on larger code bases. The experimentation will bring valuable insights into the structure of shared objects in concurrent programs, which may lead to novel techniques for concurrency control in object-oriented languages as well as novel techniques for error detection.

References

[1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 211–230, 2002.

[2] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, 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 *ACM Conference on Programming Language Design and Implementation*, pages 12–21, 2007.

[4] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, 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. Racerx: Effective, static detection of race conditions and deadlocks. In *ACM Symposium on Operating Systems Principles*, pages 237–253, 2003.

[7] C. Flanagan and S. Freund. Type-based race detection for java. In *ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.

[8] C. Flanagan, S. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *ACM Conference on Programming Language Design and Implementation*, pages 293–303, 2008.

[9] O. Lhotak and L. Hendren. Scaling Java points-to analysis using Spark. In *Conference on Compiler Construction*, pages 153–169, 2003.

[10] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ACM/IEEE International Conference on Software Engineering*, 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 *ACM Conference on Programming Language Design and Implementation*, 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 *ACM Conference on Programming Language Design and Implementation*, pages 320–331, 2006.

[15] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *ACM Conference on Programming Language Design and Implementation*, pages 14–24, 2004.

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

[17] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 43–55, 2001.

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

[19] K. Sen. Race directed random testing of concurrent programs. In *ACM Conference on Programming Language Design and Implementation*, 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 *Conference on Compiler Construction*, 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 *ACM Symposium on Principles of Programming Languages*, pages 334–345, 2006.

[22] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *European Conference on Object-oriented Programming*, 2010.

[23] C. von Praun and T. Gross. Object race detection. In *ACM Conference on Object-oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

[24] C. von Praun and T. Gross. Static conflict analysis for multithreaded object-oriented programs. In *ACM Conference on Programming Language Design and Implementation*, pages 115–128, 2003.