Detecting Deadlock in Programs with Data-Centric Synchronization

Daniel Marino*, Christian Hammer†, Julian Dolby‡, Mandana Vaziri‡, Frank Tip§, Jan Vitek¶

*Symantec Research Labs, USA
danlmarino@yahoo.com

†Saarland University, Germany
c.hammer@acm.org

‡IBM T.J. Watson Research Center, USA
{dolby,mvaziri}@us.ibm.com

§University of Waterloo, Canada
ftip@uwaterloo.ca

¶Purdue University, USA
jv@cs.purdue.edu

***Abstract*—Previously, we developed a data-centric approach to concurrency control in which programmers specify synchronization constraints declaratively, by grouping shared locations into *atomic sets*. We implemented our ideas in a Java extension called AJ, using Java locks to implement synchronization. We proved that atomicity violations are prevented by construction, and demonstrated that realistic Java programs can be refactored into AJ without significant loss of performance.**

This paper presents an algorithm for detecting possible deadlock in AJ programs by ordering the locks associated with atomic sets. In our approach, a type-based static analysis is extended to handle recursive data structures by considering programmer-supplied, compiler-verified lock ordering annotations. In an evaluation of the algorithm, all 10 AJ programs under consideration were shown to be deadlock-free. One program needed 4 ordering annotations and 2 others required minor refactorings. For the remaining 7 programs, no programmer intervention of any kind was required.

I. Introduction

Writing concurrent programs that operate on shared memory is error-prone as it requires reasoning about the possible interleavings of threads that access shared locations. If programmers make mistakes, two kinds of software faults may occur. *Data races* and *atomicity violations* may arise when shared locations are not consistently protected by locks. *Deadlock* may occur as the result of undisciplined lock acquisition, preventing an application from making progress. Previously [1–3], we proposed a data-centric approach to synchronization to raise the level of abstraction in concurrent object-oriented programming and prevent concurrency-related errors.

In our approach, fields of classes are grouped into *atomic sets*. Each atomic set has associated *units of work*, code fragments that preserve the consistency of their atomic sets. Our compiler inserts synchronization that is sufficient to guarantee that, for each atomic set, the associated units of work are serializable [4], thus preventing data races and atomicity violations *by construction*. Our previous work reported on the implementation of atomic sets as an extension of Java called AJ: we demonstrated that atomic sets enjoy low annotation overhead and that realistic Java programs can be refactored into AJ without significant loss of performance [3].

However, our previous work did not address the problem of deadlock, which may arise in AJ when two threads attempt to execute the units of work associated with different atomic sets in different orders. This paper presents a static analysis for detecting possible deadlock in AJ programs. The analysis is a variation on existing deadlock-prevention strategies [5, 6] that impose a global order on locks and check that all locks are acquired in accordance with that order. However, we benefit from the declarative nature of data-centric synchronization in AJ to infer the locks that threads may acquire: (i) all locks are associated with atomic sets, and (ii) the memory locations associated with different atomic sets will be disjoint unless they are explicitly merged by the programmer. Our algorithm computes a partial order on atomic sets which is consistent with lock acquisition order. If such an order can be found, a program is deadlock-free. For programs that use recursive data structures, the approach is soundly extended to take into account a programmer-specified ordering between different instances of an atomic set.

We implemented this analysis and evaluated it on 10 AJ programs. These programs were converted from Java as part of our previous work [3], and cover a range of programming styles. The analysis was able to prove all 10 programs deadlock-free. Minor refactorings were needed in 2 cases, and a total of 4 ordering annotations were needed, all in 1 program.

In summary, this paper makes the following contributions:

- We present a static analysis for detecting possible deadlock in AJ programs. It leverages the declarative nature of atomic sets to check that locks are acquired in a consistent order. If so, the program is guaranteed to be deadlock-free. Otherwise, possible deadlock is reported.
- To handle recursive data structures, we extend AJ with ordering annotations that are enforced by a small extension of AJ's type system. We show how these annotations are integrated with our analysis in a straightforward manner.
- We implement the analysis and evaluate it on a set of 10 AJ programs. The analysis establishes deadlock-freedom of each of these, requiring minor refactorings in 2 cases. Only 4 ordering annotations were needed, in 1 program.

II. Data-Centric Synchronization with AJ

AJ [2] extends Java with the syntax of Fig. 1. An AJ class can have zero or more atomicset declarations. Each atomic set

978-1-4673-3076-3/13 © 2013 IEEE

ICSE 2013, San Francisco, CA, USA

has a symbolic name and intuitively corresponds to a logical lock protecting a set of memory locations. Each atomic set has associated *units of work*, code fragments that preserve the consistency of their associated atomic sets. These units of work are the only code permitted to access the atomic set's fields, so only this code needs to be synchronized to ensure its consistency. By default, the units of work for an atomic set declared in a class C consist of all non-private methods in C and its subclasses. Given data-centric synchronization annotations, the AJ compiler inserts concurrency control operations that are sufficient to guarantee that any execution is *atomic-set serializable* [4], i.e., equivalent to one in which, for each atomic set, its units of work occur in some serial order. One may think of a unit of work as an atomic section [7] that is only atomic with respect to a particular set of memory locations. Accesses to locations not in the set are visible to other threads. Methods that do not operate on locations within atomic sets will not be synchronized.

We illustrate the discussion with a binary tree example. Fig. 2 shows a class Tree with fields root and size; root points to the Node that is the root of the tree. Each node has left and right fields pointing to its children, as well as a value and a weight. Class Tree has methods size(), which returns the number of nodes in the tree, find(), for finding a node with a given value, and insert() for inserting a value into the tree. The latter two methods rely on methods Node.find() and Node.insert(). Tree also has methods compute(), which returns the weighted sum of its nodes' values, and copyRoot(), which inserts the root's value into another tree passed as an argument.

We assume that the programmer wants to ensure that concurrent calls to incWeight() and compute() on the same tree never interleave, as this might trigger a race condition that causes Tree.compute() to return a stale value. We now discuss how this can be achieved in AJ.

Tree declares an atomic set t (line 2). The annotations on lines 3–4 have the effect of including root and size in this atomic set. At run time, each Tree object has an *atomic-set instance* t containing the corresponding fields. The AJ compiler inserts locks to ensure that the units of work for t execute atomically.

atomicset a	Declaration of an atomic set in a class or interface.
atomic(a)	Annotation on instance fields and classes. A field can belong to at most one atomic set. Annotated fields can only be accessed from the this reference.
unitfor(a)	Annotation on method arguments. This declares the method to be an additional unit of work for the specified atomic set in the argument object.
notunitfor	Annotation to indicate that a method is not a unit of work for atomic sets in its declaring class.
|a=this.b|	Annotation on variable declarations and constructor calls. This unifies the atomic set a in the annotated variable or constructed object with the current object's atomic set b.

Fig. 1. Data-centric annotations.

```
1  class Tree {
2    atomicset(t);
3    private atomic(t) Node root|n=this.t|;
4    private atomic(t) int  size = 1;
5    Tree(int v)           { root=new Node|n=this.t|(v); }
6    int size()            { return size; }
7    INode find(int v)     { return root.find(v); }
8    void insert(int v)    { root.insert(v); size++; }
9    int compute()         { return root.compute(); }
10   void copyRoot(Tree tree) { tree.insert(root.getValue()); }
11 }
12
13 interface INode { void incWeight(int n); }
14
15 class Node implements INode {
16   atomicset(n);
17   private atomic(n) Node left|n=this.n|;
18   private atomic(n) Node right|n=this.n|;
19   private atomic(n) int  value, weight = 1;
20
21   Node(int v)    { value = v; }
22   int getValue() { return value; }
23   void insert(int v) {
24     if (value==v) weight++;
25     else if (v < value) {
26       if (left==null)  left  = new Node|n=this.n|(v);
27       else             left.insert(v);
28     } else {
29       if (right==null) right = new Node|n=this.n|(v);
30       else             right.insert(v);
31     }
32   }
33   public void incWeight(int n){ weight += n; }
34   INode find(int v) {
35     if (value == v)   return this;
36     else if (v<value) return left==null? null : left.find(v);
37     else              return right==null? null : right.find(v);
38   }
39   int compute(){
40     int result = value * weight;
41     result += (left == null)? 0 : left.compute();
42     return result + (right == null)? 0 : right.compute();
43   }
44 }
```

Fig. 2. AJ Tree example.

Preserving the consistency of complex data structures typically requires protecting multiple objects (e.g., all of a Tree's nodes) with a single lock. This can be achieved using *aliasing annotations*, which unify the atomic sets of a Tree and the different Node objects into one larger atomic set. Aliasing annotations are type qualifiers, so the declaration Node left|n=this.n| on line 17 specifies that the atomic set instance n of the object referenced by left is unified with that of the current object. Likewise, atomic set instance n in the Node allocated on line 5 is unified with atomic set instance t in its enclosing Tree object. AJ's type system enforces the consistency of such aliasing annotations to prevent synchronization errors.

Together, the aliasing annotations on Tree and Node ensure that all locations in a Tree object are protected by the same lock. Fig. 3(a) shows a client where two threads insert concurrently into a tree. Such operations will execute correctly, as

```
45 class T extends Thread {
46   T(Tree t0, int v)  { tree=t0; value=v; }
47   public void run() { tree.insert(value); }
48   Tree tree; int value;
49 }
50
51 public static void main(String[] args) throws ... {
52   Tree tree  = new Tree(10);
53   Thread T1 = new T(tree, 12);
54   Thread T2 = new T(tree, 5);
55   T1.start (); T2.start (); T1.join (); T2.join ();
56 }
```

(a)

```
57 class U extends Thread {
58   U(Tree t1, Tree t2) { tree1=t1; tree2=t2; }
59   public void run()   { tree1.copyRoot(tree2); }
60   Tree tree1, tree2;
61 }
62
63 public static void main(String[] args) throws ... {
64   Tree tree1 = new Tree(1), tree2 = new Tree(2);
65   Thread T3 = new U(tree1, tree2);
66   Thread T4 = new U(tree2, tree1);
67   T3.start (); T4.start (); T3.join (); T4.join ();
68 }
```

(b)

Fig. 3. Two clients of the Tree class of Fig. 2.

AJ ensures mutual exclusion. Note that the client code does not refer to atomic sets at all, as is typical in our approach.

III. Deadlock Detection in AJ

A. Execution of the Example

Recall that for any object o created at runtime that is of a type that declares an atomic set t, there will be an *atomic set instance* $o.t$ that protects the fields in o that are declared to be in t. Atomic set instances can be thought of as resources that are acquired when an associated unit of work is executed. As we shall see shortly, deadlock may arise if two threads concurrently attempt to acquire such resources out of order.

Consider the program of Fig. 3(a), which creates a tree and two threads that work on it. Execution proceeds as follows:

1) When a Tree object is created and assigned to variable tree on line 52, its corresponding atomic set instance, tree.t, protects the root and size fields of the new object.
2) Tree's constructor on line 5 creates a Node object. The alias declaration on line 3 causes its left, right, value and weight fields to be included in atomic set instance tree.t.
3) The object creations for T1 and T2 on lines 53–54 are standard, with no special operations for atomic sets.
4) Once the workers start (line 55), both threads attempt to invoke insert() on tree. Since insert() is a unit of work for t and both threads operate on the same Tree object, AJ's runtime system enforces mutual exclusion, by taking a lock upon calling insert() (see Sec. V). Thus, the two operations execute serially.
5) The join() calls on line 55 wait for the workers to finish.

Now consider the code in Fig. 3(b), which is similar except that two Tree objects are created and assigned to variables tree1 and tree2 (line 64). Then, two worker threads, T3 and T4, are created on lines 65–66. Note that each worker is passed references to both tree1 and tree2 in the constructor calls, but *in a different order*. Then, each worker calls copyRoot() on one tree, which in turn calls insert() on the other. These methods are both units of work for atomic set t, so T3 attempts to acquire the lock for tree1.t upon calling copyRoot() and then the lock for tree2.t when it calls insert(). T4 attempts precisely the reverse: it acquires the lock for tree2.t when calling copyRoot() and then the lock for tree1.t when calling insert(). This is a classic situation where deadlock may arise when threads acquire multiple locks in different orders.

B. Preventing Deadlock

Deadlock can be prevented by totally ordering all possible locks, and always acquiring locks in that order. Our algorithm attempts to find a partial order $<$ on atomic sets, where $a < b$ means that threads never attempt to acquire a lock on an a while holding a lock on a b. That is, any thread that needs both locks simultaneously must acquire a first. If no such order can be found, deadlock is deemed possible. The ordering $<$ between atomic sets reflects transitive calling relationships between their units of work. For each path in the call graph from a method m that is a unit of work for atomic set a to a method n that is a unit of work for atomic set b, we create an ordering constraint $a < b$. However, if $a = b$ and we can determine that both methods are units of work on the *same atomic-set instance*, then no ordering constraint needs to be generated, as locks are reentrant. Possible deadlock is reported if, after generating all such constraints, $<$ is not a partial order. While this algorithm is conceptually simple, some complications arise in the presence of atomic set aliasing, when multiple names may refer to the same atomic set. This will be discussed further in Sec. IV.

For Fig. 3(a), the algorithm infers that atomic sets t and n are unordered and declares the program deadlock-free, since due to aliasing annotations it can show that all transitive calls between units of work simply result in lock re-entry. For Fig. 3(b), a constraint t $<$ t is inferred, indicating that deadlock may occur, as we have already seen.

C. Refactoring against Deadlocks

In our experience, many cases of deadlock can be avoided by simple refactorings that order lock acquisition. This can be accomplished using AJ's unitfor construct, which declares a method to be an additional unit of work for an atomic set in one of its parameters. For example, deadlock can be prevented in Fig. 3(b) by placing a unitfor annotation on the parameter tree of the copyRoot() method as follows:

```
void copyRoot(unitfor(t) Tree tree){
  tree.insert(root.getValue());
}
```

This declares copyRoot() to be a unit of work for atomic set instance tree.t, as well as this.t. When a method is a unit of work for multiple atomic set instances, AJ's semantics guarantees that the corresponding resources are acquired atomically, thus preventing deadlock in Fig. 3(b). Sometimes, deeper code restructuring is needed before the unitfor construct can be used; Sec. VI gives some examples.

D. Recursive Data Structures

The basic algorithm sketched above can fail to prove the absence of deadlock in programs that use recursive data structures. Fig. 4 illustrates this with a variant of our binary tree that allows concurrent updates to the weight of different nodes in the same tree. However, insert() should still ensure mutual exclusion to avoid corruption of the tree's structure.

This synchronization policy is implemented by keeping the atomic sets of the tree and of its nodes distinct: the atomic set instances of different Node objects must *not* be aliased with each other as this would preclude concurrent access to different nodes. In Fig. 4, once a thread has a reference to an INode, it can invoke incWeight() on it. As Node.incWeight() is a unit of work for the node's atomic set n, no other thread can concurrently access that node. However, since different nodes no longer share the same atomic set instance, incWeight() can be called concurrently on different nodes, as desired. Note that invoking Tree.insert() involves acquiring the lock associated with the tree's atomic set instance t, thus ensuring the desired mutual exclusion behavior.

E. Analyzing the Modified Tree Example

Now consider using the tree of Fig. 4 with the client program of Fig. 5. The basic algorithm discussed above would compute an ordering constraint n < n for this program, because Node.insert() recursively invokes itself on the children of the current node. Given the absence of aliasing annotations, these nodes now have distinct atomic set instances, and the basic algorithm concludes that deadlock is possible since it cannot rule out that two threads may access the atomic set instances of different Node objects in different orders. However, it is easy to see that this particular program is deadlock-free, as the recursive calls to insert() traverse the tree in top-down order. Hence, the locks associated with the instances of atomic set n in the traversed nodes are always acquired in a consistent order, precluding deadlock.

```
72 class Tree {
73   atomicset(t);
74   private atomic(t) Node root;
75   Tree(int v){ root = new Node(v); }
76    ...
77 }
78 class Node implements INode {
79   atomicset(n);
80   private atomic(n) Node left;
81   private atomic(n) Node right;
82    ...
83   void insert(int v){
84      ... left = new Node(v); ...
85      ... right = new Node(v); ...
86   }
87 }
```

Fig. 4. A tree that permits concurrent access to its nodes. Unmodified code fragments have been elided.

```
88 class V extends Thread {
89   V(Tree t, int v){ tree=t; val=v; }
90   public void run(){ tree.insert(val); }
91   Tree tree; int val;
92 }
93 ...
94 public static void main(String[] args)
95           throws InterruptedException{
96   Tree tree = new Tree(10);
97   Thread T5 = new V(tree, 3);
98   Thread T6 = new V(tree, 4);
99   T5.start (); T6.start (); T5.join (); T6.join ();
100 }
```

Fig. 5. Client program for the example of Fig. 4.

|this.a<a| Annotation on variables and constructors. Specifies the order between atomic set a in the annotated variable or constructed object, and the atomic set a in the current object.

Fig. 6. Extending AJ with ordering annotations.

F. Ordering Annotations

To handle recursive data structures, we extend AJ with *ordering annotations* as shown in Fig. 6. This lets programmers specify an ordering between instances of the same atomic set. The deadlock analysis can then avoid generating constraints of the form $a < a$ when the user-provided ordering indicates that a call cannot contribute to deadlock. Fig. 7 shows how to express an ordering between an atomic set n in a given node, and in each of its children. Given these annotations, our enhanced algorithm (see Sec. V) confirms that the program of Fig. 5 is indeed deadlock-free. Note that programmer-provided ordering annotations are not blindly trusted. The type-checker ensures that the specified order is acyclic while the analysis verifies that it is consistent with lock acquisition order.

IV. Algorithm

A. Auxiliary Definitions

Fig. 8 defines auxiliary concepts upon which our algorithm relies. We assume that a call graph of the program has been constructed and that $\rightarrow$ denotes the calling relationship

```
101 class Node implements INode {
102   atomicset(n);
103   private atomic(n) Node left|this.n<n|;
104   private atomic(n) Node right|this.n<n|;
105    ...
106   void insert(int v){
107      ... left = new Node|this.n<n|(v); ...
108      ... right = new Node|this.n<n|(v); ...
109   }
110 }
```

Fig. 7. Adding ordering annotations to the example of Fig. 4. Unmodified code fragments have been elided.

$\mathcal{M}$:= *set of methods in program*
$\mathcal{V}$:= *set of final method params plus a special ? symbol*
$\mathcal{A}$:= *set of atomic sets*
$\mathcal{N}$:= $\{=, <\} \times \mathcal{V} \times \mathcal{A}$ *set of lock identifiers*
$\mathcal{L}$:= $2^{\mathcal{N}}$ *set of atomic-set instances (i.e., locks)*
$\mathcal{D}$:= $2^{\mathcal{L}}$ *set of locksets*

$\texttt{uow} : \mathcal{M} \to \mathcal{D}$:= *returns the set of locks that a method grabs*
$\texttt{padaptName} : (\mathcal{M} \times \mathcal{V} \times \mathcal{M}) \to \mathcal{V}$:= *renames a variable from the perspective of caller to callee*
$\texttt{padaptLock} : (\mathcal{M} \times \mathcal{L} \times \mathcal{M}) \to \mathcal{L}$:= *adapts all names identifying a lock from the perspective of caller to callee*
$\texttt{addNames} : (\mathcal{M} \times \mathcal{L}) \to \mathcal{L}$:= *consults annotations in scope to add other names for a lock to its representation.*

$$\texttt{uow}(m) = \{\ \{\ v.A\ \} |\ m \text{ is a unit-of-work for } v.A\ \}$$

$$\texttt{addNames}(m, l) = l \cup \{\ v.A \mid w.B \in l \text{ and } v.A \text{ is annotated to be an alias for } w.B \text{ in } m\text{'s scope}\ \}$$

$$\texttt{padaptName}(m_s, v, m_t) = \begin{cases} \mathsf{this} & \text{if } m_s \text{ contains the call } v.m_t(...) \\ w & \text{if } m_s \text{ passes } v \text{ as the actual argument for the formal parameter } w \text{ of } m_t \\ ? & \text{otherwise} \end{cases}$$

$$\texttt{padaptLock}(m_s, l, m_t) = \{\ *v.A |\ *w.A \in \texttt{addNames}(m_s, l) \wedge \texttt{padaptName}(m_s, w, m_t) = v\ \}$$

$$\frac{m \text{ is an entry point}}{\emptyset \in \mathit{LBE}(m)}\ \text{(LBE-ENTRY)} \qquad \frac{n \to m \qquad d \in \mathit{LBE}(n)}{\{\ \texttt{padaptLock}(n, l, m) |\ l \in (d \cup \texttt{uow}(n))\ \} \in \mathit{LBE}(m)}\ \text{(LBE-CALL)}$$

Fig. 8. Auxiliary definitions.

between methods[1]. Function uow associates each method with the atomic-set instances for which it is a unit of work, including those due to unitfor constructs. Intuitively, $\texttt{uow}(m)$ identifies the set of locks that m acquires (or re-enters) in the current AJ implementation. A lock is an element of $\mathcal{L}$, and is represented as a *set of names* since locks may have many names due to aliasing annotations. Names (elements of $\mathcal{N}$) are notated as $*v.A$ where $*$ is either $=$ or $<$, v is a final method parameter or variable, and A is the name of an atomic set. If neither $=$ or $<$ is specified, then $=$ is assumed. Names of the form $< v.A$ are not considered until Sec. IV-C.

Fig. 8 also defines $\mathit{LBE}(m)$ (*locks before entry*), denoting the sets of locks that may be held just before entering method m. In general, different sets of locks may be held when m is invoked by different callers. It is important to keep these sets of locks distinct, to avoid imprecision in the analysis that could give rise to false positives. Our algorithm effectively performs a context-sensitive analysis by computing a separate set of locks (lockset) for each path in the call graph[2], where locksets are propagated from callers to callees and augmented with locally acquired locks. When locks are passed from caller to callee, names are *adapted* to the callee, to account for the fact that different name(s) now represent the same lock (see functions padaptName and padaptLock in Fig. 8). Note that padaptName and padaptLock use a special symbol '?' to handle cases where a lock cannot be named by a variable in the scope of the callee, and that padaptLock relies on function addNames to gather additional names that must refer to the same lock due to aliasing annotations.[3] The definition of $\mathit{LBE}(m)$ consists of two rules:

- Rule LBE-ENTRY adds the empty lockset to $\mathit{LBE}(m)$ if m is an entry point, indicating that no locks are held before the program begins.
- Rule LBE-CALL takes each lockset that may be held before entering a caller, augments it with the locks that the caller acquires, and then adapts the lockset to the perspective of the callee using padaptLock.

These rules are iterated to a fixed point in order to determine all of the locksets that may be held before entering a method.

B. Core Algorithm

Fig. 9 defines an ordering '$<$' on atomic sets using $\mathit{LBE}(m)$. Intuitively, for atomic sets A and B we have $A < B$ if a lock associated with an instance of atomic set A may be acquired before a lock that is associated with an instance of atomic set B. Rule UOW states that this is the case if there is a method m and some lockset $d \in \mathit{LBE}(m)$ that contains a lock named $v.A$, and we have some $w.B$ that names a lock in $\texttt{uow}(m)$ that is not already held in d.[4]

When atomic sets are aliased, we must account for the fact that multiple names may refer to the same lock. In general, generating an ordering constraint $A < B$ can be avoided when encountering a unit of work for atomic-set instance $w.B$ if a lock corresponding to atomic-set instance $v.A$ is already held, and if it can be determined that $v.A$ and $w.B$ must refer to the same lock, (in that case the lock is simply re-entered). Two key steps enable us to do this: (i) by keeping locksets separate

[1] To simplify the presentation, we assume that a method m calls another method n at most once, and that the same variable is not passed for multiple parameters. Our implementation, of course, does not have these restrictions.

[2] Note that $\mathit{LBE}(m)$ could conservatively contain a lockset that is never held before entering method m if the call graph contains infeasible paths. However, because AJ inserts the necessary lock acquisitions and uow reflects this knowledge, the locksets themselves are precise and represent exactly the locks that are held if a particular path in the call graph is traversed.

[3] This is not necessary for soundness, but allows the algorithm to more precisely identify lock re-entry.

[4] Note that UOW subtly relies on the fact that uow never returns a lock named using ?, since atomic-set instances for which a method is a unit-of-work are always nameable from that method's scope. Hence, there is no danger of failing to generate an ordering constraint because we are re-entering '?.B'.

$$\frac{\begin{array}{c} d \in \mathit{LBE}(m) \quad l_1 \in d \quad l_2 \in \mathtt{uow}(m) \\ v.A \in l_1 \quad w.B \in l_2 \quad l_3 \in d \Rightarrow w.B \notin l_3 \end{array}}{A < B}\ (\textsc{uow}) \qquad \frac{\begin{array}{c}\text{object creation with alias annotation} \\ \texttt{|b=this.a|}\ \text{is reachable in code}\end{array}}{A \rightsquigarrow B}\ (\textsc{gives})$$

$$\frac{A \rightsquigarrow B}{A \sim B}\ (\textsc{share-lock-1}) \qquad \frac{A \rightsquigarrow B \quad A \rightsquigarrow C}{B \sim C}\ (\textsc{share-lock-2}) \qquad \frac{A \sim B}{B \sim A}\ (\textsc{share-sym})$$

$$\frac{A < B \quad B < C}{A < C}\ (\textsc{trans}) \qquad \frac{A < B \quad B \sim C}{A < C}\ (\textsc{share-1}) \qquad \frac{A \sim B \quad B < C}{A < C}\ (\textsc{share-2}) \qquad \frac{A \rightsquigarrow B \quad B \rightsquigarrow C}{A \rightsquigarrow C}\ (\textsc{gives-trans})$$

Fig. 9. Definition of the ordering relation '$<$' between atomic sets.

for each path in the call graph, we can determine when locks must be held, and (ii) the representation of a lock maintains all its known names (*i.e.,* must-aliases), allowing us to identify situations where locks are re-entered.

To be sound, when the analysis generates ordering constraints due to lock acquisition, it must do so for all atomic sets that *may* be used to name the locks involved. Because alias annotations can be cast away, we cannot rely on local annotations to provide the analysis with all possible may-aliases for a given lock. Therefore, rules SHARE-1 and SHARE-2 conservatively generate additional orderings to account for any annotated constructors in the whole program that could cause instances of two atomic sets to be implemented using the same lock. Rather than naively merging atomic sets that have instances that may be aliased, our analysis uses a transitive '$\rightsquigarrow$' (gives) relation and a symmetric '$\sim$' (shares) relation. This avoids generating spurious ordering constraints and deadlock reports. The code in Fig. 10 demonstrates why this is needed. Two classes C and D use a utility class List, and each uses an alias annotation that causes the List's atomic set to be implemented using the lock for its own atomic set. The result is that, although a List may share a lock with either a C or a D, C objects never share locks with D objects. By maintaining this level of precision, we avoid generating a spurious deadlock report at line 127.

Lastly, rule TRANS defines '$<$' to be transitive. Now, deadlock may occur if '$<$' is not a valid partial order. Conversely, if there is no atomic set A such that $A < A$, then the program is deadlock-free: we have found a valid partial order on atomic sets that is consistent with the order in which new locks are acquired by transitively called units of work.

C. Accounting for Ordering Annotations

The basic algorithm is unable to infer a partial order among atomic sets in some programs that manipulate recursive data structures. For the program of Fig. 4, the rules of Fig. 9 infer n $<$ n, leading to the conclusion that deadlock might occur. However, as discussed in Sec. III-E, deadlock is impossible in this case because locks are always acquired in a consistent order that reflects how trees are always traversed in the same direction. Intuitively, tracking ordering constraints at the atomic-set level is insufficient in cases where threads recursively execute units of work associated with multiple instances of the same atomic set.

```
111 class List{ atomicset(l);
112   ...
113 }
114
115 class C{ atomicset(c);
116   List x =
117     new List|l=this.c |();
118   ...
119   void foo (){...}
120 }
```

```
121 class D{ atomicset(d);
122   List y =
123     new List|l=this.d |();
124   C myC = new C();
125   ...
126   void bar() {
127     myC.foo();
128     ...
129   }
130 }
```

Fig. 10. Having a separate (transitive) gives relation and (symmetric) shares relation allows us to correctly derive c$\sim$l, l$\sim$c, d$\sim$l and l$\sim$d, but *not* c$\sim$d or d$\sim$c. This precision prevents generating a spurious deadlock report at line 127.

Our solution involves having programmers specify ordering annotations that indicate a finer-grained partial order between different instances of the same atomic set, as was illustrated in Fig. 7. We extended the AJ type system to allow an atomic set instance to be ordered relative to *exactly one* other atomic set instance when it is constructed. The type system ensures that the object to which the newly constructed object is being related is already completely constructed, preventing objects that are being constructed simultaneously from specifying conflicting orders relative to one another.

Since the programmer is restricted to giving a single constraint at object creation time, with respect to a completely constructed object, a cycle in the specified order is impossible. The type system then ensures that this order is respected by any dataflow that carries the ordering annotation. Finally, the analysis verifies that the programmer-specified, acyclic ordering is consistent with lock acquisition order, signaling potential deadlock if units of work for different instances of an atomic set may be entered out of the specified order.

$$\begin{array}{l}\mathtt{addNames}(m, l) = \{\, l \cup \\ \quad \{\, *w.B \mid *v.A \in l \text{ and } w.B \text{ is annotated to be an} \\ \qquad\qquad \text{alias for } v.A \text{ in } m \text{ 's scope} \,\} \cup \\ \quad \{\, < x.A \mid *v.A \in l \text{ and } x.A \text{ is annotated to be} \\ \qquad\qquad \text{greater than } v.A \text{ in } m \text{ 's scope} \,\}\}\end{array}$$

$$\frac{\begin{array}{c} d \in \mathit{LBE}(m) \quad l_1 \in d \quad l_2 \in \mathit{uow}(m) \quad v.A \in l_1 \\ w.B \in l_2 \quad l_3 \in d \Rightarrow w.B \notin l_3 \quad < w.B \notin l_1 \end{array}}{A < B}\ (\textsc{uow})$$

Fig. 11. Changes to the algorithm to support ordering annotations between instances of an atomic set.

Fact	Derivation
A1) $\emptyset \in$ *LBE*(T.run)	LBE-ENTRY
A2) $\emptyset \in$ *LBE*(Tree.insert)	(A1), LBE-CALL
A3) { { this.n } } $\in$ *LBE*(Node.insert)	(A2), LBE-CALL

(a)

Fact	Derivation
B1) $\emptyset \in$ *LBE*(U.run)	LBE-ENTRY
B2) $\emptyset \in$ *LBE*(Tree.copyRoot)	(B1), LBE-CALL
B3) { { ?.t } } $\in$ *LBE*(Tree.insert)	(B2), LBE-CALL
B4) **t < t**	(B3), ORDER-UOW

(b)

	Fact	Derivation
C1)	$\emptyset \in$ *LBE*(V.run)	LBE-ENTRY
C2)	$\emptyset \in$ *LBE*(Tree.insert)	(C1), LBE-CALL
C3)	{ { ?.t } } $\in$ *LBE*(Node.insert)	(C2), LBE-CALL
C4)	{ { ?.t }, { ?.n } } $\in$ *LBE*(Node.insert)	(C3), LBE-CALL
C5)	t < n	(C3) or (C4), ORDER-UOW
C6)	**n < n**	(C4), ORDER-UOW

(c)

	Fact	Derivation
D1)	$\emptyset \in$ *LBE*(V.run)	LBE-ENTRY
D2)	$\emptyset \in$ *LBE*(Tree.insert)	(D1), LBE-CALL
D3)	{ { ?.t } } $\in$ *LBE*(Node.insert)	(D2), LBE-CALL
D4)	{ { ?.t }, { ?.n, < this.n } } $\in$ *LBE*(Node.insert)	(D3), LBE-CALL
D5)	t < n	(D3) or (D4), and ORDER-UOW

(d)

Fig. 12. Functioning of the algorithm on binary tree example. Relevant facts that are derivable are shown for (a) client code in Fig. 3(a) which is deadlock-free; (b) client code in Fig. 3(b) which may deadlock; (c) client code in Fig. 5, which the algorithm conservatively reports may deadlock; and (d) client code in Fig. 5 after adding ordering annotations. Several derivable facts are not shown in the figure, including $t \rightsquigarrow n$, $t \sim n$, $n \sim t$ for (a) and (b), and $t < n$, $n < n$, and $n < t$ for (b).

Fig. 11 updates our analysis to soundly accommodate untrusted, user-specified orderings between atomic set instances. Function `addNames` now consults the ordering annotations available within a method and its enclosing class. Any atomic-set instance specified to be greater than a given instance is added to the lock's representation and prefixed with a '<' to indicate that it is not a must-alias, but rather a lock that is safe to enter after the represented lock. Rule UOW now avoids generating an ordering constraint due to one lock being held when another is acquired if the former is "less" than the latter.

If the analysis indicates deadlock-freedom, then it has found a valid partial order on all atomic set instances in the program that is consistent with the order in which threads acquire them. The ordering is made up of a coarse-grained ordering on atomic sets that indicate ordering between all instances of two atomic sets, and a fine-grained ordering among instances of a single atomic set as indicated by the user's annotations. An informal correctness argument can be found in [8].

D. Example

Let us consider the behavior of our analysis on the example program in Fig. 2 and its client in Fig. 3(a). The relevant facts discovered by our analysis are shown in Fig. 12(a) along with an indication of the rules and facts used to derive them. Note that the facts shown in the figure incorporate an optimization where names of form ?.a are dropped from a lock's set representation if it also contains a must-alias not involving ?. See Sec. V for why this is safe.

From LBE-ENTRY, we know that *LBE*(T.run) contains the empty lockset. Using this fact in the premise of LBE-CALL, we derive $\emptyset \in$ *LBE*(Tree.insert). For the call from Tree.insert() to Node.insert(), LBE-CALL makes the following calculations:

- $\emptyset \in$ *LBE*(Tree.insert), `uow`(Node.insert) = { { this.t } }
- { this.t } $\in \emptyset \cup$ { { this.t } }
- `addNames`(Tree.insert, { this.t }) = { this.t, root.n }
- `padaptName`(Tree.insert, this, Node.insert) = ?
- `padaptName`(Tree.insert, root, Node.insert) = this
- `padaptLock`(Tree.insert, { this.t }, Node.insert) = { ?.t, this.n }

After removing the unnecessary name involving ?, we get { { this.n } } $\in$ *LBE*(Node.insert). Note that ?.t can be dropped because the must-alias this.n is a more exact name for the lock in this context. The recursive calls to Node.insert() result in the same lockset, so no additional facts are derived using LBE-CALL. Furthermore, no ordering facts can be derived: the only method with a non-empty lockset upon entry is Node.insert(), and that lockset already contains the lock for which the method is a unit of work, preventing rule UOW from generating an ordering constraint. Since the empty ordering relation is a valid partial order, the program is declared deadlock-free. The remainder of Fig. 12 shows the relevant facts derived for the other examples from Figs. 3(b) and 5.

V. Implementation

We implemented the deadlock analysis as an extension of our existing proof-of-concept AJ-to-Java compiler [3], which is an Eclipse plugin project. In this implementation, data-centric synchronization annotations are given as special Java comments. These comments are parsed and given to the type checker and deadlock analysis. Type errors such as the use of inconsistent ordering annotations are reported using markers in the Eclipse editor. If type-checking and the deadlock analysis succeed, the AJ source is translated to Java, and written into a new project that holds the transformed code. This project can then be compiled to bytecode, and executed using a standard JVM. More details on the implementation can be found in [3].

The deadlock analysis relies on the WALA program analysis framework[5] for the construction of a call graph. The analysis first determines all entry points to the program (e.g., main() methods and the run() methods of threads), and then builds a conservative approximation of the program's call graph.[6] The propagation of atomic sets in our analysis is essentially a distributive data flow problem, so we are able to leverage WALA's efficient Interprocedural Finite Distributive Subset solver [9]. Our actual implementation works slightly harder than the formal rules of Sec. IV in gathering and propagating information gleaned from aliasing and ordering annotations, allowing, e.g., final fields of method parameters to be included in lock names. As mentioned, lock identifiers involving ? are

[5]See wala.sourceforge.net.

[6]Reflection must be approximated as with most static program analyses.

TABLE I
AJ SUBJECT PROGRAMS. THE TABLE SHOWS, FOR EACH SUBJECT PROGRAM, THE NUMBER OF LINES OF SOURCE CODE (INCLUDING WHITE SPACE AND COMMENTS), FILES AND DATA-CENTRIC ANNOTATIONS (ONE SUB-COLUMN FOR EACH TYPE OF ANNOTATION).

benchmark	LOC		files	data-centric annotations						
	program	collections		atomic-set	atomic (class)	atomic (field)	unitfor	alias	notunitfor	total
collections	0	10846	63	5	0	53	40	330	0	428
elevator	609	yes	6	1	1	0	0	6	0	8
tsp	754	no	6	2	2	0	0	0	0	4
weblech	1971	no	14	2	0	4	0	0	0	6
jcurzez1	6639	no	49	5	2	7	15	24	0	53
jcurzez2	6633	no	49	4	3	2	6	4	0	19
tuplesoup	7217	yes	40	7	5	11	12	0	46	81
cewolf	14002	yes	129	6	6	0	0	2	0	14
mailpuccino	14519	yes	135	14	13	1	0	0	0	28
jphonelite	16484	yes	105	14	10	26	0	8	0	58
specjbb	17730	yes	64	18	15	34	1	24	4	80

discarded if an exact name for the lock is known (*i.e.,* one not including $<$ or ?). This allows the analysis to converge more quickly, and is sound since the algorithm conservatively generates additional ordering constraints from existing ones for any atomic sets that globally may have instances implemented by the same lock (see rules SHARE-1, SHARE-2).

Soundness of the analysis relies on AJ's type checker to verify that ordering annotations reflect a valid partial order. This involves checking that ordering annotations are preserved by assignment, parameter passing, and redeclaration. Casts may discard annotations but cannot manufacture them from unannotated types. A newly constructed object can be ordered with respect to at most one existing object by annotating the instance creation or a constructor parameter. Details about the changes to AJ's type system and compiler can be found in [8].

VI. EVALUATION

We analyzed a collection of AJ programs with our implementation in order to answer the following research questions:

- RQ1 How successful is the analysis in demonstrating the absence of deadlock in AJ programs?
- RQ2 How often are program transformations and ordering annotations necessary to prove the absence of deadlock?
- RQ3 What is the running time of the analysis?

A. Subject Programs

The subject AJ programs used in this evaluation are shown in Table I. These programs were created in the context of a previous project that focused on evaluating the annotation overhead and performance of AJ [3], by manually converting a number of existing multi-threaded Java programs into AJ. Details about this conversion effort are discussed in [3].

The programs were obtained from several different sources and reflect a variety of programming styles. Elevator and tsp have been used by several other researchers (e.g., [10]) in projects related to data race detection. Weblech is a web crawler that recursively downloads all pages from a web site. Jcurzez allows building text-based user interfaces for simple terminals. The original jcurzez code did not support for multi-threading, and we created two versions with well-defined behavior in the presence of concurrency: jcurzez1 achieves this behavior in a coarse-grained fashion while jcurzez2 does so using more fine-grained synchronization. Cewolf is a framework for creating graphical charts. Jphonelite is a Java SIP voice over IP SoftPhone for computers. Tuplesoup is a small Java-based framework for storing and retrieving simple hashes. Mailpuccino is a Java email client. Finally, specjbb is a widely used multi-threaded performance benchmark.All subject programs except tsp, weblech, and jcurzez rely on AJ versions of Java collections (e.g, TreeMap, ArrayList), which therefore must be analyzed as well in those cases.

Table I shows some key characteristics of the subject programs, including the number of lines of source code, the number of files, and the number of data-centric synchronization constructs. The row labeled "collections" is not a stand-alone subject program but rather displays the characteristics of the collection classes from the java.util package that we converted to AJ. The actual subject programs report only "yes" or "no" in this LOC column for collections to indicate whether they use these classes or not and thus whether the collection code was examined by the analysis.

As is apparent from the data, the number of atomic sets in the subject programs is small, ranging from 1 to 18. specjbb includes the largest number of fields in atomic sets (34 fields, and 15 entire classes). This is the case because a complex web of data structures is accessed and updated by multiple threads in this benchmark. unitfor annotations and aliasing are limited in application code but plentiful in the library classes.

B. Deadlock Analysis

In the absence of ordering annotations, our analysis guarantees the absence of deadlock in all but one of the subject programs (jcurzez2). Demonstrating the absence of deadlock in that program required 4 ordering annotations. Table II also shows the number of locksets that the algorithm generates during its analysis (i.e., the size of set $\mathcal{D}$ in Fig. 8) as well as the running time of the analysis on each subject program. Experiments were run on a MacBook Air with a 1.8 GHz Intel Core i5 processor and 4GB of RAM. Even in its current unoptimized state, the analysis takes at most 75 seconds.

TABLE II
ANALYSIS RESULTS. THE TABLE SHOWS, FOR EACH SUBJECT PROGRAM, THE NUMBER OF ORDERING ANNOTATIONS REQUIRED TO GUARANTEE THE ABSENCE OF DEADLOCK, AND THE RUNNING TIME OF OUR ANALYSIS.

	Ordering annotations	|locksets|	Time [s]
elevator	0	39	1.0
tsp	0	33	1.4
weblech	0	39	4.6
jcurzez1	0	409	10.3
jcurzez2	**4**	541	9.4
tuplesoup	0	785	8.8
cewolf	0	25	19.7
mailpuccino	0	205	48.2
jphonelite	0	34	7.2
specjbb	0	414	75.1

```
1  public abstract class AbstractWindow {
2    atomicset b;
3    protected final AbstractWindow parent|this.b<b|;
4    protected
5    AbstractWindow(AbstractWindow|this.b<b| parent, ...) {
6      this.parent = parent;
7    }
8    public |this.b<b| AbstractWindow getParent() {
9      return parent;
10   }
11 }
```

Fig. 13. Excerpt from jcurzez2 requiring ordering annotations.

For the majority of our subject programs (7 out of 10), deadlock-freedom could be demonstrated without any programmer intervention. Both specjbb and tuplesoup required some slight refactoring in order to eliminate spurious deadlock reports. In both cases, component objects of a parent object kept a reference to their parent object in a field. Later, the analysis was unable to infer the equality of the parent that called a method in a child object and the object stored in the child's parent field. We refactored the problematic calls to pass an instance of the parent as a parameter to the child's method. *Cewolf* is a J2EE servlet that does not provide a main method; its methods are invoked by an application server that we modeled with mock classes from WALA's J2EE package.

Only one subject program, jcurzez2, required ordering annotations to be proven deadlock-free. Fig. 13 shows an excerpt of the problematic methods. Class AbstractWindow contains a recursive reference to a parent window on which it sometimes makes calls. The annotation on the constructor's parent parameter causes the atomic-set instance b of a newly constructed AbstractWindow to be placed in the lock order before parent.b. The type system allows this ordering information to be propagated to the field the parameter is stored in and the return value of this field's getter method. After adding ordering annotations, our analysis can rule out deadlock.

In summary, the research questions posed at the beginning of this section can be answered as follows:

RQ1: The analysis was able to prove the absence of deadlock in all 10 of the subject programs that we considered.

RQ2: Two programs required minor refactorings before the absence of deadlock could be demonstrated. One program relied on recursive data structures that necessitated the introduction of 4 ordering annotations. For the remaining 7 programs, no programmer intervention was needed.

RQ3: The running time of the analysis is at most 75 seconds in all cases.

C. Threats to Validity

A critical reader might argue that the subject programs are small, and that they do not adequately represent concurrent programming styles that occur in practice. Obtaining suitable subject programs is a challenge for us, because AJ is a research language without real users. The AJ programs used in this evaluation were converted from Java as part of our previous work on evaluating the annotation overhead and performance of AJ [3]. Their construction predates this work on deadlock analysis and we used all AJ programs that were available. The analyzed code includes AJ versions of collections such as TreeMap and ArrayList and all of their associated auxiliary data structures (e.g., map entries and iterators), which are quite complex. Furthermore, our subject programs include specjbb, a widely-used performance benchmark, and several programs that other researchers used in research on concurrency errors. Therefore, based on the current results, we are optimistic that the proposed deadlock analysis will scale to bigger programs.

VII. RELATED WORK

Deadlock detection, prevention and avoidance is well trodden ground. In this section, we focus on static techniques.

Static analysis. At heart, all static analysis techniques attempt to detect cyclic waits-on relationships between tasks. To this end, they construct abstractions of the program's control flow, tasking and synchronization behavior. Cycles in these graphs correspond to possible deadlock. The precision of the analysis depends on ruling out cycles that cannot happen in practice. Masticola's work [5] is one example, and includes an extensive discussion of prior work.

To prove the absence of deadlocks caused by resource acquisition, a common approach is to statically look for an order on resources such that no task ever holds a resource while requesting a lesser one. Saxena [11] explored this approach in the context of concurrent Pascal code where all shared resources can be enumerated. Engler and Ashcraft [6] apply this approach to the analysis of large C programs, but abstract any non-global lock resource by the name of the type in which it is stored. Williams et al. [12] propose a lock-ordering based deadlock analysis for Java, focusing on analyzing libraries in the absence of client code. Our analysis follows this traditional approach of finding an order for resources, leveraging the declarative nature of AJ by using atomic set instances as a sound and effective abstraction for locks.

Generating deadlock-free code. Golan-Gueta et al. [13] demonstrate a technique for generating fine-grained, deadlock-free locking code for tree- and forest-based data structures. They introduce a strategy called domination locking to achieve this. AJ cannot support domination locking, but it provides a declarative way to write deadlock- and race-free code for

general-purpose programs. Emmi et al. [14] use integer linear programming (ILP) to infer a locking strategy for programs written with atomic blocks in versions of C and Java. They impose ordering constraints on lock acquisition in order to avoid generating programs that can deadlock. AJ provides more programmer control over the level of concurrency and the desired behavior than this approach.

Type systems. Type-based approaches that address deadlock typically rely on an underlying type and effect system that exposes the locking behavior in type signatures and provides some mechanism to control aliasing. Boudol's work is a good example [15]: It defines a deadlock-free semantics for an imperative language and a type and effect system for deadlock avoidance. In his work, singleton reference types allow reasoning about precise aliasing relationships between pointers and their locks. Geriakos et al. [16] extend this approach to unstructured locking and report low runtime overhead. Boyapati et al. [17] describe another such system where the notion of ownership [18] is used to restrict aliasing. In their work, a Java-like language is extended with ownership annotations and lock levels. Each lock has an associated lock level, and methods are annotated with the keyword `locks` to indicate they acquire locks at a given level. The type system ensures that locks are acquired in descending order. Gordon et al. [19] focus on fine-grained locking scenarios that involve concurrent data structures such as circular lists and mutable trees, where it is difficult to impose a strict total order on the locks held simultaneously by a thread. The approach relies on a notion of *lock capabilities*: Associated with each lock is a set of capabilities to acquire further locks, and deadlock-freedom is demonstrated by proving acyclicity of the capability-granting relation. Inference algorithms have been proposed to reduce the annotation burden. Agarwal et al. [20] present a type inference algorithm that infers locks-clauses for Boyapati's type system. In programs that cannot be typed, a generalization of GoodLock [21] is used for runtime detection. Vasconcelos et al. [22] define a type inference system for a typed assembly language that defines a partial order in which locks have to be acquired. Their system supports non-structured locks in a cooperative multi-threading environment where threads may be suspended while holding locks.

Our approach relies on a static analysis that leverages the declarative nature of synchronization in AJ to prove deadlock-freedom. Programmer-supplied ordering annotations are required only in relatively rare cases when a recursive data structure with fine-grained synchronization is manipulated concurrently. Our results suggest that this hybrid approach successfully avoids common pitfalls, such as the false positives reported by some static analyses, and the heavy notational burden of some type-based approaches.

VIII. Conclusions

We presented an analysis for detecting possible deadlock in AJ programs. The analysis is a variation on existing deadlock-prevention strategies [5, 6] that impose a global order on locks and check that locks are always acquired in accordance with that order. The declarative nature of synchronization in AJ enables us to compute an analogous ordering on atomic sets that reflects the invocations from units of work on one atomic to units of work on another. For recursive data structures, this coarse-grained ordering sometimes does not suffice. Therefore, we added ordering annotations to AJ that enable programmers to specify an order between different instances of an atomic set, and we extend our analysis to soundly take these untrusted ordering annotations into account. We extended our AJ implementation to type-check ordering annotations, and incorporated the deadlock analysis in the type checker.

In an evaluation of the algorithm, all 10 AJ programs under consideration were shown to be deadlock-free. One program needed 4 ordering annotations and 2 others required minor refactorings. For the remaining 7 programs, no programmer intervention of any kind was required.

References

[1] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *POPL*, 2006, pp. 334–345.

[2] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek, "A type system for data-centric synchronization," in *ECOOP*, 2010, pp. 304–328.

[3] J. Dolby, C. Hammer, D. Marino, F. Tip, M. Vaziri, and J. Vitek, "A data-centric approach to synchronization," *ACM TOPLAS* 34(1):4, 2012.

[4] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *ICSE*, 2008, pp. 231–240.

[5] S. P. Masticola, "Static detection of deadlocks in polynomial time," Ph.D. dissertation, Rutgers University, 1993.

[6] D. R. Engler and K. Ashcraft, "Racerx: effective, static detection of race conditions and deadlocks," in *SOSP*, 2003, pp. 237–252.

[7] T. L. Harris and K. Fraser, "Language support for lightweight transactions," in *OOPSLA*, 2003, pp. 388–402.

[8] D. Marino, C. Hammer, J. Dolby, M. Vaziri, F. Tip, and J. Vitek, "Detecting deadlock in programs with data-centric synchronization," IBM, Tech. Rep. RC25300, 2012.

[9] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *POPL*, 1995, pp. 49–61.

[10] C. von Praun and T. R. Gross, "Atomicity violations in object-oriented programs," *J. Object Technology* 6(3), pp. 103–122, 2004.

[11] A. Saxena, "Static detection of deadlocks," University of Colorado at Boulder, Tech. Rep. CU-CS-122-77, 1977.

[12] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *ECOOP*, 2005, pp. 602–629.

[13] G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav, "Automatic fine-grain locking using shape properties," in *OOPSLA*, 2011, pp. 225–242.

[14] M. Emmi, J. S. Fischer, R. Jhala, and R. Majumdar, "Lock allocation," in *POPL*, 2007, pp. 291–296.

[15] G. Boudol, "A deadlock-free semantics for shared memory concurrency," in *Theoretical Aspects of Computing - ICTAC 2009*, 2009, pp. 140–154.

[16] P. Gerakios, N. Papaspyrou, and K. Sagonas, "A type and effect system for deadlock avoidance in low-level languages," in *TLDI*, 2011.

[17] C. Boyapati, R. Lee, and M. C. Rinard, "Ownership types for safe programming: preventing data races and deadlocks," in *OOPSLA*, 2002, pp. 211–230.

[18] J. Noble, J. Potter, and J. Vitek, "Flexible alias protection," in *ECOOP*, 1998, pp. 158–185.

[19] C. S. Gordon, M. D. Ernst, and D. Grossman, "Static lock capabilities for deadlock freedom," in *TLDI '12*, 2012, pp. 67–78.

[20] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Haifa Verification Conference*, 2005, pp. 191–207.

[21] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *SPIN*, 2000, pp. 245–264.

[22] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro, "Type inference for deadlock detection in a multithreaded polymorphic typed assembly language," in *EPTCS*, vol. 17, 2010, pp. 95–109.