Detecting Deadlock in Programs with Data-Centric Synchronization

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

*Symantec Research Labs
Email: danlmarino@yahoo.com

†Saarland University
Email: c.hammer@acm.org

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

§Purdue University
Email: 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, 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 simple type-based static analysis is extended to handle recursive data structures by considering programmer-supplied lock ordering annotations. In an evaluation of the algorithm on 10 AJ programs, all of these programs were shown to be deadlock-free. Only 4 ordering annotations were needed, in one program and 2 programs required some 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], [2], [3], we proposed a data-centric approach to synchronization that aims 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 also 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 yet 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. In languages like Java, where locks are allocated dynamically, reasoning about deadlock involves determining the values that references may point to and understanding sharing patterns in the heap. Solutions to this problem (discussed in Section VI) include static program analysis and type systems. Static analysis techniques compute an over-approximation of the objects that variables may point to and can, in some cases, rule out deadlock. However, in other cases, they may reject perfectly correct programs with little usable feedback for programmers. Type systems provide programmers with a clear set of rules for writing code that is guaranteed to be deadlock-free. However, type-based approaches can be notationally heavy and overly restrictive.

This paper presents a static analysis for detecting possible deadlock in programs with data-centric synchronization. The analysis can be seen as a variation on existing deadlock-prevention strategies [5], [6] that impose a global order on locks and check that all threads acquire locks in accordance with that order. However, our approach leverages the declarative nature of data-centric synchronization in AJ to infer the locks that different threads may acquire. In particular, we rely on the fact that, in AJ programs, (i) all locks are associated with atomic sets, and that (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 types that declare atomic sets. If such an order can be found, a program is deadlock-free. For programs that use recursive data structures, this simple type-based approach is extended to take into account a programmer-specified ordering between different instances of an atomic set.

We implemented these ideas and evaluated them 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 implemented the analysis and evaluated it on a set of AJ programs. The analysis found all programs to be deadlock-free, requiring minor refactorings in two 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 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. 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 ►cite◄ 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 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 that node's 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 copy(), which inserts the root's value into another tree passed as an argument.

atomicset a	A class or interface declaration may have multiple atomic set declarations.
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)	Each method argument can be annotated by one or more unitfor annotations, which has the effect of making the method an additional unit of work for the specified atomic sets in the argument object.
|a=this.b|	Annotation on variable declarations and in constructor expressions. The atomic set a of the type of the annotated variable or constructed object is aliased 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;
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 copy(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.

In this example, we assume 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. However, preserving the consistency of complex data structures typically requires multiple objects

```
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.copy(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.

(e.g., all of a Tree's nodes) to be protected by 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| (line 17) specifies that the atomic set instance n of the object referenced by left is unified with that of the current object. AJ's type system enforces the consistency of aliasing annotations to prevent synchronization errors. For example, the Node allocated on line 5 is annotated |n=this.t| to enable the type system to verify that it belongs to the same atomic set instance as the enclosing Tree object.

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 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

We will now discuss, using a motivating example, how deadlock may arise in AJ programs and how to prevent it.

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 Figure 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 **??**, 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 of T1 and T2 on lines **??**–**??** are standard, with no special operations for atomic sets.
4) Once the workers start (line **??**), 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 Section **??**). Thus, the two operations will execute serially.
5) The join() calls on line **??** wait for the workers to finish.

Now consider the code in Figure 3(b), which is similar except that two Tree objects are created and assigned to variables tree1 and tree2 (line **??**). Then, two worker threads, T3 and T4, are created on lines **??**–**??**. Note that each worker thread is passed references to both tree1 and tree2 in the constructor calls, but *in a different order*. Then, each worker calls copy() 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 and then the lock for tree2.t, and T4 attempts to acquire the lock for tree2.t and then tree1.t. This is a classical situation where deadlock may arise when threads acquire multiple locks in different orders.

B. Preventing Deadlock

Deadlock can be prevented by always acquiring locks in order. Our algorithm attempts to find a partial order $<$ on atomic sets, where $a < b$ means that units of work on atomic set a are always executed before units of work on atomic set b. 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 minor 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 Section 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 copy() method as follows:

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

This declares copy() 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; Section V 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. 6 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. 6, 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 desired mutual exclusion behavior.

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

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

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

Fig. 5. Client program for the example of Fig. 6.

|this.a<a| Annotation on variables and constructors. This 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.

E. Analyzing the modified tree example

Now consider Fig. 7. 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. In 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.

F. Ordering Annotations

To handle recursive data structures, we extend AJ with *ordering annotations* as shown in Fig. 8. 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

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

Fig. 7. Adding ordering annotations to the example of Fig. 6. 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 is a unit-of-work for $v.A \}$

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

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

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

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

Fig. 8. Auxiliary definitions.

that a call cannot contribute to deadlock. Fig. 9 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 confirms that the program of Fig. 7 is indeed deadlock-free. To ensure that it is sound for the analysis to rely on ordering annotations, AJ's type checker must verify that they are valid. This will be discussed in Section **??**.

IV. Algorithm

Sec. **??** presents auxiliary definitions. Sec. **??** presents our core algorithm for detecting possible deadlock. Then, Sec. **??** describes an extension that performs a more precise analysis of recursive data structures such as the one in Fig. 6.

A. Auxiliary Definitions

Fig. 10 defines auxiliary concepts upon which our algorithm relies. We assume that a call graph of the program has been constructed and that $\to$ denotes the calling relationship 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. **??**.

Fig. 10 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. 10). 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

[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.

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

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

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

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

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. **??** defines an ordering '$<$' on atomic sets using *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 $d \in$ *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, care must be taken to account for the fact that multiple names may refer to the same lock. In general, the generation of 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, because in that case the lock is simply re-entered. Two key steps enable us to do this:

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

However, we cannot rely on local annotations alone to give us all possible names for a given lock (*i.e.,* may-aliases) as aliasing annotations can be cast away. Therefore, rules SHARE-1 and SHARE-2 conservatively generate additional orderings to account for any aliasing annotations in the whole program that may cause instances of two atomic sets to be implemented using the same lock. To prevent generating spurious ordering constraints, we use a transitive '$\rightsquigarrow$' (gives) relation and a symmetric '$\sim$' (shares) relation instead of simply merging atomic sets when they may be aliased. To see why this is needed, consider a situation where two classes C and D both use a utility class List, and where each aliases List's atomic set to its own. Then while a C object or a D object may share a lock with a List object, C objects never share locks with D objects. 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 programs that manipulate recursive data structures. For the program of Fig. 6, the rules of Fig. **??** infer $\mathsf{n} < \mathsf{n}$, leading to the conclusion that deadlock might occur. However, as discussed in Sec. **??**, 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 execute units of work associated with multiple instances of the same atomic set.

Our solution involves having programmers specify ordering annotations that imply the existence of a finer-grained partial order between different instances of the same atomic set, as was illustrated in Fig. 9. 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

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

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

(UOW)

Fig. 10. Changes to the algorithm to support ordering annotations between instances of an atomic set.

[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'.

	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.copy)	(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. 11. 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. 7, which the algorithm conservatively reports may deadlock; and (d) client code in Fig. 7 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).

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.

Fig. 16 updates our analysis to accommodate user-specified orderings between instances of an atomic set. 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 guaranteed to be less than latter.

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 that are discovered by our analysis are shown in Fig. 15(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 has other, more exact names.

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). The recursive calls to Node.insert() result in the same lockset, so no additional facts are derived using the LBE-CALL. Furthermore, there are no ordering facts that 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. 15 shows the relevant facts derived for the other examples from Figs. 3(b) and 7.

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 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 original source is translated to Java, and written into a new project that holds the transformed code. This project can then be compiled to Java bytecode, and executed using a standard Java VM. For further details on the implementation of AJ, the reader is referred to [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 use WALA's efficient implementation of an Interprocedural Finite Distributive Subset solver to perform the analysis [7]. Our actual implementation works slightly harder than the formal rules of Sec. IV in gathering and propagating information gleaned from aliasing and ordering annotations, for example allowing final fields of method parameters to be included in lock names. As mentioned before, lock identifiers involving ? are 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

[5] See wala.sourceforge.net.

[6] Reflection must be approximated as with most static program analyses. In our case, the only reflective calls that matter are the ones that may call units of work.

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	330	40	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 (tuned)	17730	yes	64	18	15	34	1	24	4	80

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).

conservatively generates additional ordering constraints from existing ones for any atomic sets which globally may have instances implemented by the same lock (see rules SHARE-1, SHARE-2).

To ensure that it is sound for the analysis to rely on ordering annotations, AJ's type checker must verify that they are valid. 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. The ordering must also be consistent with the runtime ordering used to enforce orderly acquisition of locks when a unit of work takes multiple locks. Details about the changes to AJ's type system and compiler can be found in a technical report ►cite◄.

VI. EVALUATION

In order to gain some insight into the practical applicability of our deadlock analysis, we analyzed a collection of AJ programs with our implementation. The main purpose of this evaluation is 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 before the analysis could prove the absence of deadlock?

RQ3 What is the running time of the analysis?

A. Subject Programs

We applied the deadlock analysis to the AJ programs 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. For details about the specific steps involved in this conversion effort, the reader is referred to [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., [8]) 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 have clearly defined support for multi-threading, and we created two versions of the code 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 easy to use Java-based framework for storing and retrieving simple hashes. mailpuccino program 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 programs range from small to medium-sized. The number of atomic sets is small, ranging from 1 to 18. SpecJBB has the largest number of fields declared 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. The unitfor annotations are limited in application code but plentiful in the library classes. Aliasing is a sign of linked data structures that must be kept in sync, but that can be accessed from multiple entry points. Again, these are mostly found in library classes.

	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 (tuned)	0	414	7.1

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.

```
1  public abstract class AbstractWindow {
2    atomicset b;
3    protected final AbstractWindow parent|this.b<b|;
4
5    protected
6    AbstractWindow(AbstractWindow|this.b<b| parent, ...) {
7      this.parent = parent;
8    }
9    public |this.b<b| AbstractWindow getParent() {
10     return parent;
11   }
12 }
```

Fig. 12. Excerpt from jcurzez2 requiring ordering annotations.

B. Deadlock Analysis

In the absence of ordering annotations, our deadlock analysis is able to guarantee the absence of deadlock in all but one of the subject programs (jcurzez2). Demonstrating the absence of deadlock in that program required the insertion of 4 ordering annotations. Table II also shows the number of different locksets that the algorithm generates during its analysis (i.e., the size of the set $\mathcal{D}$ of locksets in our algorithm) as well as the running time of the deadlock analysis for each subject program. **►Dan: include details of machine on which experiments were performed◄** Even in its current unoptimized state, the analysis and the subsequent rewriting pass run in matter of seconds with no noticeable memory pressure.

For the majority of our subject programs (7 out of 10), the analysis could prove the absence of deadlock 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. The benchmark *cewolf* is a J2EE servlet that does not provide a central main method. Instead, its methods are resolved by an application server which we modeled with mock classes from WALA's J2EE package.

Only one subject program, jcurzez2, required ordering annotations to be proven deadlock-free. This program implements a fine-grained synchronization scheme that prevents fully automatic certification of deadlock freedom. Fig. 18 shows an excerpt of the problematic methods. The 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 . This fact can be exploited during deadlock analysis. 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, the deadlock analysis establishes that deadlock is impossible.

C. Threats to Validity

►discuss benchmark selection bias, etc.◄

VII. RELATED WORK

Deadlock detection, prevention and avoidance is well trodden ground. We restrict our focus to the most closely related static techniques.

a) Static analysis.: At heart, all static analysis techniques follow [5] and attempt to detect cyclic waits-on relationships between tasks. To this end, they construct abstractions, at various level of precision, of the program's control flow, tasking and synchronization behavior. Cycles that occur in these graphs correspond to possible deadlock. The precision of the analysis depends on ruling out cycles that cannot happen in practice. Engler and Ashcraft [6] choose a simple type-based approximation in which variables are represented by the name of their type, instead of a full-fledged alias analysis. In a language with subtyping, such an approximation would lump all objects together due to upcasts. Williams et al. [9] propose another flow-sensitive and context-sensitive static analysis for Java. They construct a lock order graph, which represents the order in which locks are acquired. Nodes of this graph correspond to sets of objects that may be aliased, and edges between nodes to precedence between lock operations. Again, cycles indicate deadlocks. Williams et al. present a number of unsound heuristics to reduce false positives and resort to well-behavedness assumptions about clients of libraries (e.g., clients do not make calls to library methods from within callbacks, and do not lock explicitly library objects). Finally, Wang et al. [10] build on Engler and Ashcraft's analysis and synthesize code that is guaranteed to prevent deadlocks while striving not to reduce concurrency too much. Overall, the appeal of these techniques is that they require no programmer intervention and their drawbacks are the number of false positives and the fact that if a deadlock is reported, the programmer is offered no real guidance on how to fix the code.

b) Type systems.: Type-based approaches that address deadlocks are often based 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 [11]: 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. [12] extend this approach to unstructured locking and report low runtime overhead. Boyapati et al. [13] describe another such system where the notion of ownership [14] 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. Methods have effect annotations as well. The type system is limited to tree-shaped recursive structures and hierarchical locks. A type system that ensures that locks are acquired in descending order is discussed, but no proofs or empirical results are presented. Gordon et al. [15] 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. [16] presented a type inference algorithm that can infer locks-clauses for Boyapati's type system. For programs that cannot be typed, they propose a generalization of GoodLock [17] for runtime detection. Vasconcelos et al. [18] define a type inference system for a typed assembly language. Their type system 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. Type-based techniques give programmers clear guidelines for writing deadlock free algorithms; the type system defines the rules precisely. But they have the disadvantage of being onerous in term of programmer effort and inference is usually imprecise due to its flow-insensitive nature. Type and effect systems are arguably impractical in object-oriented languages due to the presence of subtyping in those languages. Dealing cleanly with, for example, collection classes likely requires much more type machinery than most end users are willing to put up with.

VIII. Conclusions

This paper presented a whole-program analysis for deadlock detection in the context of the AJ language. The declarative nature of synchronization constructs in AJ allows for a simpler algorithm than previous deadlock detection techniques, in part because the annotations AJ uses to generate synchronization code naturally partition the locks used to protect shared memory, simplifying reasoning about aliasing. The presence of composite and recursive data structures requires that we extend AJ with a new annotation to enforce ordering constraints between distinct instances of atomic sets. The ordering annotations allow the deadlock analysis to differentiate between instances of the same type. However, our experiments show that such annotations are rarely needed in practice and that the analysis can validate most subject programs without any programmer intervention at all. Overall we have found that AJ presents fewer challenges for deadlock detection than plain Java code, and that the various object-oriented programming idioms found in our target programs can be supported in a straightforward manner. By leveraging AJ's declarative nature, and making careful tradeoffs between precision and abstraction, we are able to achieve a scalable and effective deadlock analysis.

References

[1] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *Conference record of the Symposium on Principles of Programming Languages (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 *Proc. of the European Conference on Object-Oriented Programming (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 Transactions on Programming Languages and Systems (To appear)*, 2012, a version appeared as IBM Research Report RC25106.

[4] C. Hammer, J. Dolby, M. Vaziri, and F. Tip, "Dynamic detection of atomic-set-serializability violations," in *Proceedings of the International Conference on Software Engineering (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 *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237–252.

[7] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the Symposium on Principles of programming languages (POPL)*, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[8] C. von Praun and T. R. Gross, "Atomicity violations in object-oriented programs," *Journal of Object Technology*, vol. 3, no. 6, pp. 103–122, June 2004.

[9] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 602–629.

[10] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke, "The theory of deadlock avoidance via discrete control," in *Proceedings of the Symposium on Principles of programming languages (POPL)*, 2009, pp. 252–263.

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

[12] P. Gerakios, N. Papaspyrou, and K. Sagonas, "A type and effect system for deadlock avoidance in low-level languages," in *Proceedings of the Workshop on Types in language design and implementation (TLDI)*, 2011, pp. 15–28.

[13] C. Boyapati, R. Lee, and M. C. Rinard, "Ownership types for safe programming: preventing data races and deadlocks," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2002, pp. 211–230.

[14] J. Noble, J. Potter, and J. Vitek, "Flexible alias protection," in *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, Jul. 1998.

[15] C. S. Gordon, M. D. Ernst, and D. Grossman, "Static lock capabilities for deadlock freedom," University of Washington Department of Computer Science and Engineering, Tech. Rep. UW-CSE-11-10-01, Oct. 2011.

[16] 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.

[17] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *SPIN*, 2000, pp. 245–264.
[18] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro, "Type inference for deadlock detection in a multithreaded polymorphic typed assembly language," in *Electronic Proceedings in Theoretical Computer Science*, vol. 17, 2010, pp. 95–109.

APPENDIX

A. Typing

The most complete description of AJ's type system is given in [3]. This section presents the small changes that are required to validate ordering annotations. A *class* definition C is well-typed if its fields are well-typed in the context of C. Furthermore, all methods (including non-overridden inherited methods) must be well-typed. In the definitions below, we use the notation C *has* a to indicate that class C declares or inherits an atomic set a. Checking a field declaration where τ is a type with an alias ordering annotation simply requires checking that all referenced atomic sets exist.

$$\frac{\begin{array}{c}(\tau \equiv \mathsf{D}|\mathsf{this.b}<\mathsf{a}| \ \mathit{implies} \ \ \mathsf{D} \ \mathit{has} \ \mathsf{a} \ \mathit{and} \ \mathsf{C} \ \mathit{has} \ \mathsf{b}) \\ (\alpha = \mathsf{atomic}\,(\mathsf{b}) \ \mathit{implies} \ \ \mathsf{C} \ \mathit{has} \ \mathsf{b})\end{array}}{\alpha\,\tau\,\mathsf{f} \quad \text{OK in C}} \ \text{(T-FIELD)}$$

Checking a method requires typing its body in an environment E constructed by composing the disjoint sets of parameters $\overline{\mathsf{x}}$, local variables $\overline{\mathsf{z}}$ and the distinguished variable this. If class C has an atomic set a, the type of this is $\mathsf{C}|\mathsf{a}<=\mathsf{this.a}|$. The type of the local variable y appearing in the return statement must match the return type of the method, and if the method overrides an inherited method, the signature must be unchanged.

$$\frac{\begin{array}{c}E \equiv \overline{\mathsf{x}:\tau_{\mathsf{x}}}, \overline{\mathsf{z}:\tau_{\mathsf{z}}}, \mathsf{this}:\tau_{\mathsf{this}} \quad E \ \vdash \ \mathsf{s}; \mathsf{return}\,\mathsf{y} \\ E(\mathsf{y}) = \tau \quad \mathsf{C} \ \mathsf{extends} \ \mathsf{D} \\ (\mathit{if} \ \ \mathsf{C} \ \mathit{has} \ \mathsf{a} \ \ \mathit{then} \ \ \tau_{\mathsf{this}} \equiv \mathsf{C}|\mathsf{this.a}<\mathsf{a}| \ \ \mathit{else} \ \ \tau_{\mathsf{this}} \equiv \mathsf{C}) \\ \mathit{override}(\mathsf{m}, \mathsf{D}, \overline{\tau_{\mathsf{x}}} \to \tau) \\ (\tau_{\mathsf{x}} \equiv \mathsf{E}|\mathsf{this.a}<\mathsf{b}| \ \mathit{implies} \\ \mathsf{E} \ \mathit{has} \ \mathsf{b} \ \mathit{and} \ \mathsf{C} \ \mathit{has} \ \mathsf{a} \ \mathit{and} \ \mathsf{m} \ \mathit{is\ constructor})\end{array}}{\tau\,\mathsf{m}(\overline{\tau_{\mathsf{x}}\,\mathsf{x}})\,\{\,\overline{\tau_{\mathsf{z}}\,\mathsf{z}};\ \mathsf{s}; \mathsf{return}\,\mathsf{y}\,\} \quad \text{OK in C}} \ \text{(T-METHOD)}$$

Type checking casts simply requires checking that when the source variable has an ordering annotation this ordering annotation not be modified by the cast.

$$\frac{\begin{array}{c}E(\mathsf{x}) = \mathsf{D}|\mathsf{this.b}<\mathsf{a}| \qquad E(\mathsf{y}) = \mathsf{C}|\mathsf{this.b}<\mathsf{a}| \\ \mathsf{C} \ \mathit{has} \ \mathsf{a} \qquad E(\mathsf{this}) \ \mathit{has} \ \mathsf{b} \qquad \mathsf{D} <: \mathsf{C}\end{array}}{E \ \vdash \ \mathsf{y} = (\mathsf{C}|\mathsf{this.b}<\mathsf{a}|)\mathsf{x}} \ \text{(T-CAST-ASET)}$$

It is possible to entirely discard any alias annotation, including ordering constraints.

$$\frac{E(\mathsf{x}) = \mathsf{C}|\mathsf{this.b}<\mathsf{a}| \qquad \mathsf{C} \ \mathit{not} \ \mathsf{internal} \qquad E(\mathsf{y}) = \mathsf{C}}{E \ \vdash \ \mathsf{y} = (\mathsf{C})\mathsf{x}} \ \text{(T-CAST-OFF)}$$

The rule for method calls checks the types of the arguments and the return type. Viewpoint adaption is necessary to ensure that the types of the arguments and the return value are visible from the viewpoint of the receiver. We do not detail viewpoint adaption here, the only important consideration is that it does not change ordering annotations.

$$\frac{\begin{array}{ccc}E(\mathsf{y}) = \tau_{\mathsf{y}} & \mathit{typeof}\,(\tau_{\mathsf{y}}.\mathsf{m}) = \overline{\tau} \to \tau & E(\overline{\mathsf{z}}) = \overline{\tau_{\mathsf{z}}} \\ \overline{\tau_{\mathsf{z}}} = \mathit{adapt}(\overline{\tau}, \tau_{\mathsf{y}}) & \tau' = \mathit{adapt}(\tau, \tau_{\mathsf{y}}) & E(\mathsf{x}) = \tau'\end{array}}{E \ \vdash \ \mathsf{x} = \mathsf{y}.\mathsf{m}(\overline{\mathsf{z}})} \ \text{(T-CALL)}$$

The rules for field selection and field update are unchanged. They already check that the type of the field matches exactly (including ordering annotations) that of the variable it is stored into.

$$\frac{E(\mathsf{this}) = \tau \quad E(\mathsf{x}) = \tau_{\mathsf{f}} \quad \mathit{typeof}\,(\tau.\mathsf{f}) = \tau_{\mathsf{f}}}{E \ \vdash \ \mathsf{x} = \mathsf{this.f}} \ \text{(T-SELECT)}$$

$$\frac{E(\mathsf{this}) = \tau \quad E(\mathsf{y}) = \tau_{\mathsf{f}} \quad \mathit{typeof}\,(\tau.\mathsf{f}) = \tau_{\mathsf{f}}}{E \ \vdash \ \mathsf{this.f} = \mathsf{y}} \ \text{(T-UPDATE)}$$

B. Compiler

To support ordering annotations, the AJ compiler must be modified to ensure that a programmer-specified ordering between different instances of an atomic set is consistent with the runtime ordering that is used to enforce orderly acquisition of locks when multiple locks are taken by one unit of work. This order is implemented by assigning a *lockId* to newly constructed objects.

We assign lockIds to newly constructed objects that are constrained by ordering annotations as follows. For instance creations of the form

```
new Node|this.n<n|()
```

the lockId of the newly created object is assigned such that it is larger than the object referred to by the this. In practice, we pick the first available lockId which is larger. Conversely for

```
new Node|n<this.n|()
```

the implementation would pick the first available lockId less than the lockId of this. Lastly, consider the case when the new object is constrained by an ordered parameter of its constructor:

```
class Person {
  atomicset a;
  final private Person|this.a<a| dad;
  Person(Person |this.a<a| other) {  dad = other;  }
}
```

In this example the ordering annotation tells us that we will always grab the lock on a child before locking its dad. When a Person is constructed, the compiler ensures that the lockId of the newly allocated object is set to the first available value smaller than dad's.