Static Deadlock Detection for Java Libraries

Amy Williams, William Thies, and Michael D. Ernst

Computer Science and Artificial Intelligence Laboratory,
Massachusetts Institute of Technology,
Cambridge, MA 02139 USA
{amy, thies, mernst}@csail.mit.edu

Abstract. Library writers wish to provide a guarantee not only that each procedure in the library performs correctly in isolation, but also that the procedures perform correctly when run in conjunction. To this end, we propose a method for static detection of deadlock in Java libraries. Our goal is to determine whether client code exists that may deadlock a library, and, if so, to enable the library writer to discover the calling patterns that can lead to deadlock.

Our flow-sensitive, context-sensitive analysis determines possible deadlock configurations using a lock-order graph. This graph represents the order in which locks are acquired by the library. Cycles in the graph indicate deadlock possibilities, and our tool reports all such possibilities. We implemented our analysis and evaluated it on 18 libraries comprising 1245 kLOC. We verified 13 libraries to be free from deadlock, and found 14 distinct deadlocks in 3 libraries.

1 Introduction

Deadlock is a condition under which the progress of a program is halted as each thread in a set attempts to acquire a lock already held by another thread in the set. Because deadlock prevents an entire program from working, it is a serious problem.

Finding and fixing deadlock is difficult. Testing does not always expose deadlock because it is infeasible to test all possible interleavings of a program's threads. In addition, once deadlock is exhibited by a program, reproducing the deadlock scenario can be troublesome, thus making the source of the deadlock difficult to determine. One must know how the threads were interleaved to know which set of locks are in contention.

We propose a method for static deadlock detection in Java libraries. Our method determines whether it is possible to deadlock the library by calling some set of its public methods. If deadlock is possible, it provides the names of the methods and variables involved.

To our knowledge, the problem of detecting deadlock in libraries has not been investigated previously. This problem is important because library writers may wish to guarantee their library is deadlock-free for any calling pattern. For example, the specification for `java.lang.StringBuffer` in Sun's Java Development Kit (JDK) states:

A.P. Black (Ed.): ECOOP 2005, LNCS 3586, pp. 602–629, 2005.

```
class BeanContextSupport {
  protected HashMap children;

  public boolean remove(Object targetChild) {
    synchronized(BeanContext.
                 globalHierarchyLock) {
      ...
      synchronized(targetChild) {
        ...
        synchronized (children) {
          children.remove(targetChild);
        }
        ...
      }
    }
    return true;
  }

  public void
  propertyChange(PropertyChangeEvent pce) {
    ...
    Object source = pce.getSource();
    synchronized(children) {
      if ("beanContext".equals(propertyName)
          && containsKey(source)
          && ((BCSChild)children.get(source)).
             isRemovePending()) {
        BeanContext bc = getBeanContextPeer();
        if (bc.equals(pce.getOldValue())
            && !bc.equals(pce.getNewValue())) {
          remove(source);
        } else {
          ...
}}}}}
```

Fig. 1. Simplified code excerpt from the `BeanContextSupport` class in the `java.beans.beancontext` package of Sun's JDK

```
Object source
  = new Object();

BeanContextSupport support
  = new BeanContextSupport();

BeanContext oldValue
  = support.getBeanContextPeer();

Object newValue
 = new Object();

PropertyChangeEvent event
 = new PropertyChangeEvent(source,
                           "beanContext",
                           oldValue,
                           newValue);

support.add(source);
support.vetoableChange(event);

thread 1:  support.propertyChange(event);
thread 2:  support.remove(source);
```

Fig. 2. Client code that can cause deadlock in methods from Figure 1. In thread 1, `children` is locked, then `BeanContext.globalHierarchyLock` is locked (via a call to `remove`) while in thread 2, the ordering is reversed. Deadlock occurs under some thread interleavings. The initialization code shown above is designed to elicit the relevant path of control flow within the library

> The [`StringBuffer`] methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

If the operations are to behave as if they occurred in some serial order, deadlock between `StringBuffer` methods should not be possible. No serial ordering over the `StringBuffer` methods could lead to deadlock because locks acquired by Java's `synchronized` construct (which `StringBuffer` uses) cannot be held between method calls. Nonetheless, our tool reports a calling pattern that causes deadlock in `StringBuffer`.

Libraries are often vulnerable to deadlock. We have induced 14 distinct instances of deadlock in 3 libraries (for detailed results, see Section 6). Simplified code for one of the deadlocks found in Sun's JDK is shown in Figure 1. In the `BeanContextSupport` class of the `java.beans.beancontext` package, the `remove()` and `propertyChange()` methods obtain locks in a different order. The client code shown in Figure 2 can induce deadlock using these methods. Several other methods in the same package use the same locking order as `remove()` and thus exhibit the same deadlock vulnerability.

This deadlock has a simple solution: the `propertyChange()` method can synchronize on `BeanContext.globalHierarchyLock` before `children`, or it could lock only `globalHierarchyLock`. Section 6.1 describes solutions for other deadlocks.

An overview of our analysis is given in Section 3. We have implemented our technique and analyzed 18 libraries consisting of 1245k lines of code, obtained from SourceForge, Savannah, and other open source resources. Using our tool, we verified 13 of these libraries to be free of deadlock, and confirmed 14 distinct instances of deadlock in 3 libraries.

Detecting deadlock across all possible calls to a library is different than detecting deadlock in a whole program. Concrete aliasing relationships exist and can be determined for a whole program, whereas the analysis of a library must consider all possible calls into the library, which includes a large number of aliasing possibilities. In a program, the number of threads can often be determined, but a client may call into a library from any number of threads, so our analysis must model an unbounded number of threads. These differences combine to yield a much larger number of reports than would be present in a program, which makes it important to suppress false reports.

The remainder of this paper is organized as follows. Section 2 explains the semantics of locks in the Java programming language. Section 3 discusses our analysis at a high level, and Section 4 provides a more detailed description of the analysis. Section 5 describes techniques for reducing the number of spurious reports. Section 6 gives our experimental results. Related work is given in Section 7, and Section 8 concludes.

2 Locks in Java

In Java, each object conceptually has an associated lock; for brevity, we will sometimes speak of an object as being a lock. The Java "`synchronized (`*expr*`) {` *statements* `}`" statement evaluates the expression to an object reference, acquires the lock, evaluates the statements in the block, and releases the lock when the block is exited, whether normally or because of an exception. This design causes locks to be acquired in some order and then released in reverse (that is, in LIFO order), a fact that our analysis takes advantage of. A Java method can be declared `synchronized`, which is syntactic sugar for wrapping the body in `synchronized (this) { ... }` for instance methods, or `synchronized (`*C*`.class) { ... }`, where *C* is the class containing the method, for static methods.

A lock that is held by one thread cannot be acquired by another thread until the first one releases it. A thread blocks if it attempts to acquire a lock that is held by another thread, and does not continue processing until it successfully acquires the lock.

A lock is held per-thread; if a given thread attempts to re-acquire a lock, then the acquisition always succeeds without blocking.[1] The lock is released when exiting the `synchronized` statement that acquired it.

The `wait()`, `notify()`, and `notifyAll()` methods operate on receivers whose locks are held. An exception is thrown if the receiver's lock is not held. The `wait()` method releases the lock on the receiver object and places the calling thread in that object's wait set. While a thread is in an object's wait set, it is not scheduled for processing. Threads are reenabled for processing via the `notify()` and `notifyAll()` methods, which, respectively, remove one or all the threads from the receiver object's wait set. Once a thread is removed from an object's wait set, the `wait()` method attempts to reacquire the lock for the object it was invoked on. The `wait()` method returns only after the lock is reacquired. Thus, a thread may block inside `wait()` as it attempts to reacquire the lock for the receiver object.

Java 1.5 introduces new synchronization mechanisms in the `java.util.concurrent` package that allow a programmer to acquire and release locks without using the `synchronized` keyword. These mechanisms make it possible to acquire and release locks in any order (in particular, acquires and releases need not be in LIFO order). Our tool does not handle these new capabilities in the Java language. However, most synchronization can be expressed using the primitives from Java 1.4, and we therefore expect that our technique will be applicable under current and future releases of Java.

3 Analysis Synopsis

We consider a *deadlock* to be the condition in which a set of threads cannot make progress because each is attempting to acquire a lock that is held by another member of the set. Our deadlock detector uses an interprocedural analysis to track possible sequences of lock acquisitions within a Java library. It represents possible locking patterns using a graph structure—the *lock-order graph*, described below. Cycles in this graph indicate possibilities of deadlock.

For each cycle, our tool reports the variable names of the locks involved in the deadlock as well as the methods that acquire those locks (see Section 4.4). Our tool is conservative and reports all deadlock possibilities. However, the conservative approximations cause the tool to consider infeasible paths and impossible alias relationships, resulting in false positives (spurious reports).

3.1 Lock-Order Graph

The analysis builds a single lock-order graph that captures locking information for an entire library. This graph represents the order in which locks are acquired

[1] For our purposes, it is sufficient to consider multiple synchronized statements over the same object in one thread as a no-op. A Java virtual machine tracks the number of *lock*/*unlock* actions (entrance and exit of a `synchronized` block) for each object. A counter is updated for each `synchronized` statement, but if the current thread already holds the target lock, no change is made to the thread's lock set.

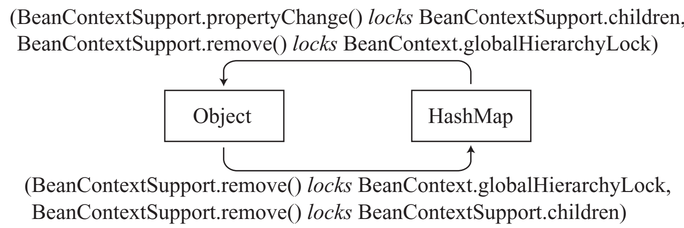


Fig. 3. Relevant portion of the lock-order graph for the code in Figure 1. The nodes represent the set of all `Object`s and `HashMap`s, respectively. Each edge is annotated by the sequence of methods (and corresponding variable names) that acquire first a lock from the source set, then a lock from the destination set

via calls to the library's public methods. Combining information about the locking behavior of each public method into one graph allows us to represent any calling pattern of these methods across any number of threads.

Each node of the lock-order graph represents a set of objects that may be aliased. (Types are an approximation to may-alias information; Section 5.1 gives a finer but still lightweight approximation applicable to fields.) An edge in the graph indicates nested locking of objects along some code path. That is, it indicates the possibility of locking first an object from the source node, then an object from the destination node.

A cycle consisting of nodes N_1 and N_2 means that along some code path, an object $o_1 \in N_1$ may be locked before some object $o_2 \in N_2$, and along another (or the same) path, o_2 may be locked before o_1. In general, a cycle exposes code paths leading to cyclic lock orders, and, when the corresponding paths are run in separate threads, deadlock may occur. Figure 3 shows the lock-order graph for the code in Figure 1.

To build the graph, the analysis iterates over the methods in the library, building a lock-order graph for each of them. All possible locking configurations of a method are modeled, including locks acquired transitively via calls to other methods. At a call site, the callee's graph is inserted into the caller. After each method's lock-order graph has reached a fixed point, the public methods' lock-order graphs are merged into a single graph for the library. Cycles are then detected, and reports are generated.

3.2 Deadlocks Detected by Our Technique

Our goal is to detect cases in which a sequence of client calls can cause deadlock in a library, or to verify that no such sequence exists. Our tool reports deadlock possibilities in which all deadlocked threads are blocked within a single library, attempting to acquire locks via Java `synchronized` statements or `wait()` calls. Under certain assumptions about the client and the library, our tool reports all such possibilities.

Our analysis focuses on deadlocks due to lock acquisitions via Java `synchronized` statements and `wait()` calls: progress of a program is halted as each thread in a set attempts to acquire a lock already held by another thread in the set. We are not concerned with other ways in which a program may fail to make progress. A thread might hang forever while waiting for input, enter an infinite loop, suffer livelock, or fail to call `notify()` or to release a user- or library-defined lock (that is, using a locking mechanism not built into Java). These problems in one thread can prevent another thread or the whole program from making progress: consider a call to `Thread.join()` (which waits for a given thread to terminate) on a thread that does not terminate. Detecting all of these problems is outside the scope of this paper.

Assumptions About Client Code. We make three assumptions about client code. If a client deviates from these assumptions, our tool is still useful for detecting deadlock, but it cannot detect deadlocks introduced by the deviant behavior. First, we assume that the client does not include a class that extends a library class or belongs to a library package. If such a class exists, it needs to be inspected by our analysis and treated as part of the library. Second, we assume that that the client does not invoke library methods within callbacks from the library; that is, all client methods M are either unreachable from the library, or the library is unreachable from M. For example, if a client class overrides `Object.hashCode()` such that it calls a synchronized method in the library, then any library method calling `hashCode()` should model that synchronization. The class therefore needs to be analyzed as though it is part of the library. Third, we assume that the client code is *well-behaved*: either it does not lock any objects locked by the library, or it does so in disciplined ways (as explained below).

Without the assumption of well-behavedness, it is difficult or impossible to guarantee deadlock freedom for a library without examining client code. An adversarial client can induce deadlock if it has access to two objects locked by a library. For example, suppose that a library has a synchronized method:

```
class A {
  synchronized void foo() { ... }
}
```

Then a client could cause deadlock in the following way:

```
A a1 = new A(), a2 = new A();
thread 1:  synchronized(a1) { a2.foo(); }
thread 2:  synchronized(a2) { a1.foo(); }
```

A client that locks a different set of objects than those locked by the library is always well-behaved. This is the case for arbitrary clients if the locks used by the library do not escape it; that is, if they are inaccessible to the client. Section 5.1 describes a method for detecting some inaccessible locks.

Even if the client and the library share a set of locks, the client can be well-behaved if it acquires those locks in a restricted pattern. These restrictions could be part of the library's specification—and such documentation could even be automatically generated for the library by a tool like ours. As above, one sufficient restriction is that clients do not lock objects that the library may lock; this requires the library to specify the set of objects that it will lock. A more liberal but sufficient restriction is that the client acquires locks in an order compatible with the library. In this scenario, the library specifies the order of lock acquisitions (say, as a lock-order graph), and clients are forbidden from acquiring locks in an order that introduces cycles into the graph. We believe that these restrictions are quite reasonable, and that information about the locks acquired by a library are a desirable part of its specification.

Assumptions About Library Code. In practice, libraries do not exist in isolation. Rather, each library uses additional libraries (e.g., the JDK) to help it accomplish its task. One approach to analyzing such cascaded libraries is to consider all of the libraries together, as if they were a single library. However, this hampers modularity, as the guarantees offered for one library depend on the implementation of other libraries. It also hampers scalability, as the effective library size can grow unwieldy for the analysis. For these reasons, our analysis considers each library independently. Consider that the "main" library under consideration relies on several "auxiliary" libraries. Under certain assumptions about the main library, our analysis detects all deadlock possibilities in which all threads are blocked within the main library. It does not report cases in which some threads are blocked in the main library and other threads are blocked in auxiliary libraries.

We make the following assumptions about library code. First, as the library under consideration (the main library) may be a client of some auxiliary libraries, it must satisfy the client assumptions (described previously) to guarantee deadlock freedom for its own users. Second, the main library cannot perform any synchronization in methods that are reachable via callbacks from auxiliary libraries (e.g., in `Object.hashCode()`). Callbacks through the auxiliary libraries are inaccessible to the analysis. Third, the library cannot use reflection. Reflection can introduce opaque calling sequences that impact the lock ordering. As with the client code, our analysis operates as usual even if these assumptions are broken, but it can no longer guarantee that all deadlock possibilities are reported.

4 Algorithm Details

The deadlock detector employs an interprocedural dataflow analysis for constructing lock-order graphs. The analysis is flow-sensitive and context-sensitive. At each program point, the analysis computes a symbolic state modeling the library's execution state. The symbolic state at the end of a method serves as a method summary. The analysis is run repeatedly over all methods until a fixed point is reached; termination of the analysis is guaranteed.

The type domains for the analysis are given in Figure 4. For simplicity, we present the algorithm for a language that models the subset of Java relevant to our analysis. The language omits field assignments; they are not relevant because

$$
\begin{array}{rl}
T \in & \text{Type} \\
v \in & \text{LocalVar} \\
\mathit{method} \in & \text{MethodDecl} = T_r\ m(T_1\ v_1,\, T_2\ v_2,\, \ldots,\, T_n\ v_n)\ \{\ \mathit{stmt}\ \} \\
 & \qquad\qquad\quad \text{where } v_1 = \texttt{this} \text{ if } m \text{ is instance method} \\
\mathit{library} \in & \text{Library} = \text{set-of MethodDecls} \\
\mathit{stmt} \in & \text{Statement} = \begin{array}{ll} \ \ T\ v & \mid \texttt{branch}\ \mathit{stmt}_1\ \mathit{stmt}_2 \\ \mid v := \texttt{new}\ T & \mid \texttt{synchronized}\ (v)\ \{\ \mathit{stmt}\ \} \\ \mid v_1 := v_2 & \mid v := m(v_1, \ldots, v_n) \\ \mid v_1 := v_2.f & \mid \texttt{wait}(v) \\ \mid \mathit{stmt}_1;\ \mathit{stmt}_2 & \end{array} \\
\mathit{pp} \in & \text{ProgramPoint}_\perp \\
o = \langle \text{pp}, \text{T} \rangle \in & \text{HeapObject} = \text{ProgramPoint} \times \text{Type} \\
g \in & \text{Graph} = \text{directed-graph-of HeapObjects} \\
\mathit{roots} \in & \text{Roots} = \text{set-of HeapObjects} \\
\mathit{env} \in & \text{Environment} = \text{LocalVar} \rightarrow \text{HeapObject} \\
s = \langle \text{g}, \text{roots}, \text{locks}, \in & \text{State} = \text{Graph} \times \text{Roots} \times \text{list-of HeapObjects} \times \\
\text{env}, \text{wait} \rangle & \qquad\qquad \text{Environment} \times \text{set-of HeapObjects}
\end{array}
$$

Fig. 4. Type domains for the lock-order dataflow analysis. Parameters are considered to be created at unique points before the beginning of a method. The "`branch` stmt_1 stmt_2" statement is a non-deterministic branch to either stmt_1 or stmt_2

our analysis does not track the flow of values through fields. Synchronized methods are modeled in this language using their desugaring (see Section 2) and loops are supported via recursion. Our implementation handles the full Java language.

Our analysis operates on symbolic heap objects. Each symbolic heap object represents the set of objects created at a given program point [6]; it also contains their type. For convenience, we say that a symbolic heap object o is *locked* when a particular concrete object drawn from o is locked.

The **state** is a 5-tuple consisting of:

- The current lock-order **graph**. Each node in the graph is a symbolic heap object. The graph represents possible locking behavior for concrete heap objects drawn from the sets modeled by the symbolic heap objects. A path of nodes $o_1 \ldots o_k$ in the graph corresponds to a potential program path in which o_1 is locked, then o_2 is locked (before o_1 is released), and so on.
- The **roots** of the graph. The roots represent objects that are locked at some point during execution of a given method when no other lock is held.
- The list of **locks** that are currently held, in the order in which they were obtained.
- An **environment** mapping local variables to symbolic heap objects. The environment is an important component of the interprocedural analysis, as it allows information to propagate between callers and callees. It also improves precision by tracking the flow of values between local variables.
- A set of objects that have had **wait** called on them without an enclosing `synchronized` statement in the current method.

4.1 Dataflow Rules

The dataflow rules for the analysis are presented in Figure 5. Helper functions appear in Figure 6, and mathematical operators (including the join operator) are defined in Figure 7. Throughout the following explanation, we define the *current lock* as the most recently locked object whose lock remains held; it is the last object in the list of currently held locks, or tail(s.locks).

The symbolic state is updated in the **visit_stmt** procedure (in Figure 5) which visits each statement in a method. A variable declaration or initialization introduces a fresh heap object. An assignment between locals copies an object within the local environment. A field reference introduces a fresh object (the analysis does not model the flow of values through fields). A `branch` models divergent paths and is handled by the join operator below. Calls to `wait()` are described in Section 4.2.

The rule for `synchronized` statements handles lock acquires; there are two cases. First, if the target object o is not currently locked (i.e., if $o \notin s$.locks), then an edge is added to the lock-order graph from the current lock to o, and o is appended to s.locks. If no objects were locked before the `synchronized` statement, o becomes a root in the graph (roots are important at a call site, as discussed below). Next, the analysis descends into the body of the `synchronized` block. Upon completion, the analysis continues to the next statement, preserving the lock-order graph from the `synchronized` block but restoring the list of locked objects valid before the `synchronized` statement. This is correct, since Java's syntax guarantees that any objects locked within the `synchronized` block are also released within the block.

In the second case for `synchronized` statements, the target is currently locked. Though the body is analyzed as before, the synchronization is a no-op and does not warrant an edge in the lock-order graph. To exploit this fact, the analysis needs to determine whether nested `synchronized` statements are locking the same concrete object. Though symbolic heap objects represent *sets* of concrete objects, they nonetheless can be used for this determination: if nested `synchronized` statements lock variables that are mapped to the same heap object (during analysis), then they always lock the same concrete object (during execution). This is true within a method because each heap object is associated with a single program point; as this simplified language contains no loops, any execution will visit that point at most once and hence create at most one concrete instance of the heap object. This notion also extends across methods, as both heap objects and concrete objects are directly mapped from caller arguments into callee parameters as described below. Thus, repeated synchronization on a given heap object is safely ignored, significantly improving the precision of the analysis.

Method calls are handled by integrating the graph for the callee into the caller as follows. In the case of overridden methods, each candidate implementation's graph is integrated. The analysis uses the most recent lock-order graph that has been calculated for the callee. Recursive sequences are iterated until reaching a fixed point. The calling context is first incorporated into a copy of the callee's graph either by removing the formal parameters (if the corresponding argument

```
visit_stmt(stmt, s) returns State s′
  s′ ← s
  switch(stmt)
    case T v  |  v := new T
      s′.env ← s.env[v := ⟨ program_point(stmt), T ⟩]
    case v1 := v2
      s′.env ← s.env[v1 := s.env[v2]]
    case v1 := v2.f
      s′.env ← s.env[v1 := ⟨ program_point(stmt), declared_type(v2.f) ⟩]
    case stmt1; stmt2
      s1 ← visit_stmt(stmt1, s)
      s′ ← visit_stmt(stmt2, s1)
    case branch stmt1 stmt2
      s′ ← visit_stmt(stmt1, s) ⊔ visit_stmt(stmt2, s)
    case synchronized (v) { stmt }
      o ← s.env[v]
      if o ∈ s.locks then
        // already locked o, so synchronized statement is a no-op
        s1 ← s
      else
        // add o to g under current lock, or as root if no locks held
        if s.locks is empty  // below, • denotes list concatenation
          then s1 ← ⟨s.g ∪ o, s.roots ∪ o, s.locks • o, s.env, s.wait⟩
          else s1 ← ⟨s.g ∪ o ∪ edge(tail(s.locks) → o), s.roots, s.locks • o,
                     s.env, s.wait⟩
      s2 ← visit_stmt(stmt, s1)
      s′ ← ⟨s2.g, s2.roots, s.locks, s2.env, s2.wait⟩
    case v := m(v1, . . . , vn)
      s′.env ← s.env[v := ⟨ program_point(stmt), return_type(m) ⟩]
      ∀ versions of m in subclasses of env[v1].T:
        sm ← visit_method(method_decl(m))
        s′m ← rename_from_callee_to_caller_context(sm, s, n)
        // connect the two graphs, including roots
        s′.g ← s′.g ∪ s′m.g
        if s.locks is empty then  // connect current lock to roots of s′m
          s′.roots ← s′.roots ∪ s′m.roots
          s′.wait ← s′.wait ∪ s′m.wait
        else
          ∀ root ∈ s′m.roots:
            s′.g ← s′.g ∪ edge(tail(s.locks) → root)
          ∀ o ∈ s′m.wait:  if tail(s.locks) ≠ o then
            s′.g ← s.g ∪ o ∪ edge(tail(s.locks) → o)
    case wait(v)
      o ← s.env[v]
      if s.locks is empty then
        s′.wait ← s.wait ∪ o
      else if tail(s.locks) ≠ o then
        // wait releases then reacquires o: new lock ordering
        s′.g ← s.g ∪ o ∪ edge(tail(s.locks) → o)
```

Fig. 5. Dataflow rules for the lock-order data-flow analysis

program_point(*stmt*) returns the program point for statement *stmt*
visit_method($T_r\ m(T_1\ v_1, \ldots, T_n\ v_n)$ { *stmt* }) returns State s'
  $s' \leftarrow$ empty State
  $\forall$ parameters $T_i\ v_i$ (including `this`):
    $s' \leftarrow$ **visit_stmt**($T_i\ v_i, s'$) // process formals via "$T\ v$" rule
  $s' \leftarrow$ **visit_stmt**($stmt, s'$)

rename_from_callee_to_caller_context(s_m, s, n) returns State s'_m
  $s'_m \leftarrow s_m$
  $\forall j \in [1, n] : formal_j \leftarrow s_m$.env$[v_j]$ // formal parameter
  $\forall j \in [1, n] : actual_j \leftarrow s$.env$[v_j]$ // actual argument
  $\forall o \in s_m$.g : // for all objects o locked by the callee
    if $\exists\ j\ \ s.t.\ \ o = formal_j$
      // o is formal parameter j of callee method
      then if $actual_j \in s$.locks
        // caller locked o, remove o from callee graph
        then s'_m.g, s'_m.roots $\leftarrow$ **splice_out_node**(s_m.g, s_m.roots, o)
        // caller did not lock o, rename o to actual arg
        else s'_m.g, s'_m.roots $\leftarrow$ **replace_node**(s_m.g, s_m.roots, o, $actual_j$)
      // o is not from caller, rename o to bottom program point $pp_\perp$
      else s'_m.g, s'_m.roots $\leftarrow$ **replace_node**(s_m.g, s_m.roots, o, $\langle pp_\perp, o.\mathrm{T}\rangle$)
  s'_m.wait $\leftarrow \emptyset$
  $\forall o \in s_m$.wait // for all objects in wait set
    if $\exists\ j\ \ s.t.\ \ o = formal_j$
      then s'_m.wait $\leftarrow s'_m$.wait $\cup\ actual_j$
      else s'_m.wait $\leftarrow s'_m$.wait $\cup\ \langle pp_\perp, o.\mathrm{T}\rangle$

splice_out_node($g, roots, o$) returns Graph g', Roots $roots'$
  $g' \leftarrow g \setminus o$
  $\forall$ edges$(src \rightarrow o) \in g\ \ s.t.\ \ o \neq src$:
    $\forall$ edges$(o \rightarrow dst) \in g\ \ s.t.\ \ o \neq dst$:
      $g' \leftarrow g'\ \cup$ edge$(src \rightarrow dst)$
  $roots' \leftarrow roots \setminus o$
  if $o \in roots$ then
    $\forall$ edges$(o \rightarrow dst) \in g\ \ s.t.\ \ o \neq dst$:
      $roots' \leftarrow roots' \cup dst$

replace_node($g, roots, o_{old}, o_{new}$) returns Graph g', Roots $roots'$
  $g' \leftarrow (g \setminus o_{old})\ \cup\ o_{new}$
  $\forall$ edges$(src \rightarrow o_{old}) \in g : g' \leftarrow g'\ \cup$ edge$(src \rightarrow o_{new})$
  $\forall$ edges$(o_{old} \rightarrow dst) \in g : g' \leftarrow g'\ \cup$ edge$(o_{new} \rightarrow dst)$
  if $o_{old} \in roots$
    then $roots' \leftarrow (roots \setminus o_{old})\ \cup\ o_{new}$
    else $roots' \leftarrow roots$

Fig. 6. Helper functions for the lock-order dataflow analysis

$actual_j$ is locked at the call site, in which case the lock acquire is a no-op from the caller's perspective) or by replacing them with the caller's actual arguments (if $actual_j$ is not locked at the call site). The non-formal parameter nodes are

$g_1 \ \cup \ g_2$ returns Graph g'
  // nodes are HeapObjects: equivalent values are collapsed
  $\text{nodes}(g') = \text{nodes}(g_1) \cup \text{nodes}(g_2)$
  // edges are pairs of HeapObjects: equivalent pairs are collapsed
  $\text{edges}(g') = \text{edges}(g_1) \cup \text{edges}(g_2)$

$g \ \setminus \ o$ returns Graph g'
  $\text{nodes}(g') = \text{nodes}(g) \setminus o$
  $\text{edges}(g') = \text{edges}(src \rightarrow dst) \in g \ \ s.t. \ \ o \neq src \ \wedge \ o \neq dst$

$s_1 \sqcup s_2$ returns State s'
  $s'.\text{g} \leftarrow s_1.\text{g} \cup s_2.\text{g}$
  $s'.\text{roots} \leftarrow s_1.\text{roots} \cup s_2.\text{roots}$
  $s'.\text{locks} \leftarrow s_1.\text{locks}$ // $s_1.\text{locks} = s_2.\text{locks}$
  $\forall v \in \{v' \mid v' \in s_1.\text{env} \ \vee \ v' \in s_2.\text{env}\}$:
    if $s_1.\text{env}[v] = s_2.\text{env}[v]$
      then $s'.\text{env} \leftarrow s'.\text{env}[v := s_1.\text{env}(v)]$
      else $s'.\text{env} \leftarrow s'.\text{env}[v := \langle \textbf{program\_point}(\text{join\_point}(v)),\ T_1 \sqcup T_2 \ \rangle]$
  $s'.\text{wait} \leftarrow s_1.\text{wait} \cup s_2.\text{wait}$
$T_1 \sqcup T_2$ returns lowest common superclass of T_1 and T_2

Fig. 7. Union and difference operators for graphs, and join operator for symbolic state

then replaced with nodes of the same type and with a special program point of $pp_\perp$, indicating that they originated at an unknown program point (bottom). The callee's wait set is adjusted in a similar fashion. At this point, an edge is added from the current lock in the caller to each of the roots of the modified callee graph. Finally, the two graphs are merged, collapsing identical nodes and edges.

The **join** operator ($\sqcup$) in Figure 7 is used to combine states along confluent paths of the program (e.g., `if` statements). We are interested in locking patterns along any possible path, which, for the graphs, roots, and wait sets, is simply the union of the two incoming states' values. The list of current locks does not need to be reconciled between two paths, as the hierarchy of `synchronized` blocks in Java guarantees that both incoming states will be the same. The new environment remains the same for mappings common to both paths. If the mappings differ for a given variable then a fresh heap object must be introduced for that variable. The fresh object is assigned a program point corresponding to the join point for the variable (each variable is considered to join at a separate location). The strongest type constraint for the fresh object is the join of the variables' types along each path—their lowest common superclass.

The algorithm for constructing the entire library's lock-order graph is given in Figure 8. The **top_level** procedure first computes a fixed point state value for each method in the library. Termination is guaranteed since there can be at most $|\text{PP}| \cdot |\text{Type}|$ heap objects in a method and the analysis only adds objects to the graph at a given stage. After computing the fixed points, the procedure performs a post-processing step to account for subclassing. Because the analysis

top_level(*library*) returns Graph g
  $s_1, \ldots, s_n \leftarrow$ dataflow fixed points over public methods in *library*
  $g \leftarrow$ **post_process**$(s_1, \ldots, s_n)$

post_process$(s_1, \ldots, s_n)$ returns Graph g
  $g \leftarrow$ empty Graph
  $\forall i \in [1, n]$:
    $\forall$ edges $(o_1 \rightarrow o_2) \in s_i$.g:
      // Add edges between all possible subclasses of locked objects.
      // All heap objects now have bottom program point $pp_\perp$.
      $\forall$ subclasses T_1 of $o_1.T$, $\forall$ subclasses T_2 of $o_2.T$:
        $o_{T_1} \leftarrow \langle pp_\perp, T_1 \rangle$
        $o_{T_2} \leftarrow \langle pp_\perp, T_2 \rangle$
        $g \;\leftarrow\; g \;\cup\; o_{T_1} \;\cup\; o_{T_2} \;\cup\;$ edge$(o_{T_1} \rightarrow o_{T_2})$

Fig. 8. Top-level routine for constructing a lock-order graph for a library of methods

for each method was based on the declared type of locks, extra edges must be added for all possible concrete types that a given heap object could assume. While it is also possible to modify the dataflow analysis to deal with subclassing at each step, it is simpler and more efficient to use post-processing.

4.2 Calls to `wait()`

A call to `wait()` on object o causes the lock on o to be released and subsequently reacquired, which is modeled by adding an edge in the lock-order graph from the most recently acquired lock to o. However, this edge can be omitted if o is also the most recently acquired lock, as releasing and reacquiring this lock has no effect on the lock ordering. In contrast to `synchronized` statements, `wait()` can influence the lock-order graph even though its receiver is locked at the time of the call. For example, before the `wait()` call in Figure 9, `a` is locked before `b`. However, during the call to `wait()`, `a`'s lock is released and later acquired while `b`'s lock remains held, so `a` is also locked after `b`. Deadlock is therefore possible.

It is illegal to call `wait()` on an object whose lock is not held; if this happens during program execution, Java throws a runtime exception. Even so, it is possible for a method to call `wait()` outside any `synchronized` statement, since the receiver could be locked in the caller. When a method calls `wait()` outside any `synchronized` statement, our analysis needs to consider the calling context to determine the effects of the `wait()` call on the lock-order graph. For this reason, when no locks are held and `wait()` is called, the receiver object is stored in the wait set and later accounted for in a caller method.

None of the libraries we analyzed reported any potential deadlocks due to `wait()`. This suggests that programmers most often call `wait()` on the most recently acquired lock.

```
void m1(Object a, Object b) {
  synchronized(a) {
    synchronized (b) {
      a.wait();
      ...
}}}
```

```
void m2(Object a, Object b) {
  synchronized(a) {
    a.notify();
    synchronized (b) {
      ...
}}}
```

```
Object a = new Object();
Object b = new Object();

thread 1:  m1(a, b);
thread 2:  m2(a, b);
```

Fig. 9. Method `m1()` imposes both lock orderings `a`→`b` and `b`→`a`, due to the call to `a.wait()`. Method `m2()`, which imposes the lock ordering `a`→`b`, can cause deadlock when run in parallel with `m1()`, as illustrated in the third column

4.3 Dataflow Example

An example of the dataflow analysis appears in Figure 10. The example contains a class `A` with two methods, `foo()` and `bar()`. The symbolic state s_{foo} represents the method summary for `foo()`. Program points are represented as a variable name and a line number corresponding to the variable's assignment. For example, $\langle \mathrm{pp}_{\mathtt{b1:5}}, \mathtt{B}\rangle$ is a symbolic heap object, of type `B`, for parameter `b1` on line 5 of `foo()`; $\langle \mathrm{pp}_{\mathtt{lock:11}}, \mathtt{B}\rangle$ is a symbolic heap object, also of type `B`, for the field `lock` as referenced on line 11 of `foo()` (though `lock` is declared on line 2, each field reference creates a fresh heap object). The lock-order graph for `foo()` illustrates that parameters `b1` and `c1` can each be locked in sequence, with `lock` locked separately. Note that the graph contains two separate nodes for `b1` and `lock`—both of type `B`—in case one of them can be pruned when integrating into the graph of a caller.

The symbolic state in `bar()` immediately before the call to `foo()` is represented by s_{bar1}. Since `bar()` is a synchronized method, a heap object for `this` appears as a root of the graph. The graph illustrates that parameters `b2` and `c2` can be locked while the lock for `this` is held. The list of locks held at the point of the call is given by s_{bar1}.locks; it contains `this` and `c2`.

The most interesting aspect of the example is the method call from `bar()` to `foo()`. This causes the graph of s_{foo} to be adjusted for the calling context and then integrated into the graph of s_{bar1} with edges added from the node for the current lock, `c2`. The calling context begins with the actual parameter `b2`. Since `b2` is not locked in s_{bar1} at the point of the call, the formal parameter `b1` is replaced by `b2` throughout the graph of s_{foo}. However, the actual parameter `c2` is locked in s_{bar1}, so the corresponding formal parameter `c1` is removed from the graph of s_{foo}. The last node in `foo()` corresponds to `lock`, which is a field reference rather than a formal parameter; thus, its program point is replaced with $pp_\perp$ before integrating into `bar()`. The result, s_{bar2}, has one new node ($pp_\perp$) and two new edges (from `c2` to both `b2` and $pp_\perp$). The other state components in s_{bar2} are unchanged from s_{bar1}.

The last component of Figure 10 gives the overall lock-order graph, treating `foo()` and `bar()` as a library of methods. As there is no subclassing in this example, the final lock-order graph can be obtained simply by taking the union of graphs from s_{foo} and s_{bar2}, setting all program points to $pp_\perp$. The cycle in the lock-order graph corresponds to a real deadlock possibility in which `foo()` and `bar()` are called concurrently with the same arguments.

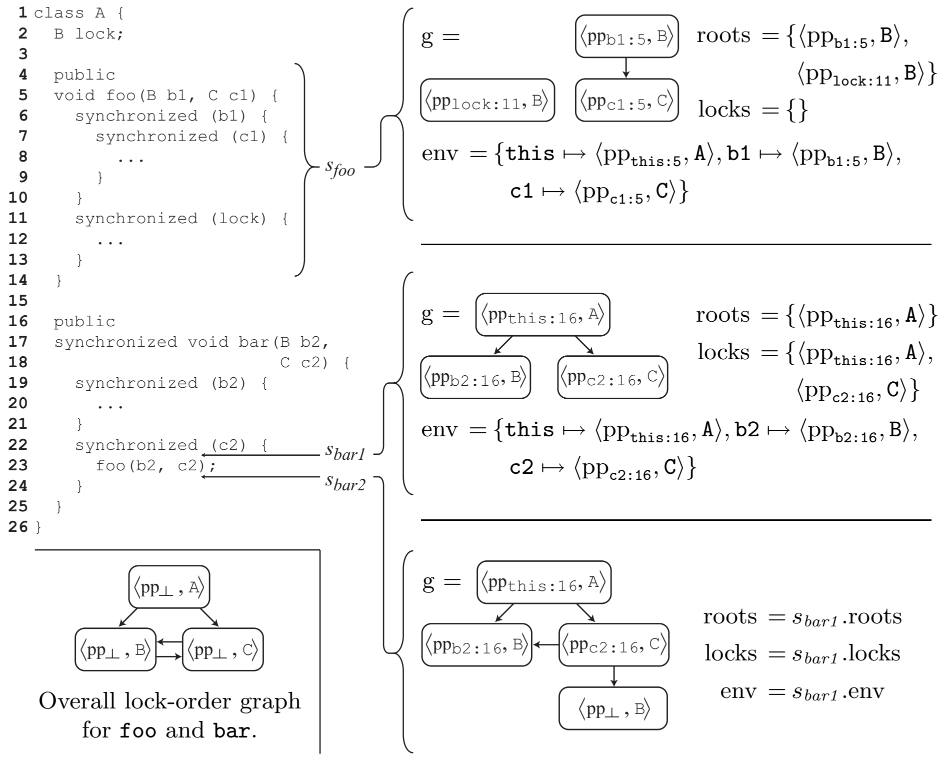


Fig. 10. Example operation of the dataflow analysis. The symbolic state is shown for the method summary of `foo`, as well as for two points in `bar` (before and after a call to `foo`). The `wait` sets (not shown) are empty in each case. The top-level lock-order graph for this library of methods is shown at bottom left

4.4 Reporting Possible Deadlock

To report deadlock possibilities, the analysis finds each cycle in the lock-order graph, using a modified depth-first search algorithm. Once a cycle is found, a report is constructed using its edge annotations. Each edge in the lock-order graph has a pair of annotations, one for the source lock and one for the destination lock. Each annotation consists of the variable name of the lock and the method that acquires it. As graphs are combined, edges may come to have multiple annotations.

A report is given for each distinct set of lock variables. These reports include each of the sets of methods that acquire that set of locks. In this way, methods with the same or similar locking behavior are presented to the user together. In our experience with the tool, most of the grouped method sets constitute the same locking pattern, so this style can save significant user effort.

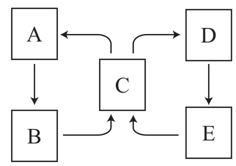


Fig. 11. The path {C, A, B, C, D, E} is a non-simple cycle: it visits node C twice

```
public void println(String s)
{
  synchronized (this) {
    print(s);
    newLine();
  }
}
```

```
public void print(String s) {
  if (s == null) {
    s = "null";
  }
  write(s);
}
```

```
private void write(String s) {
  try {
    synchronized (this) {
      ...
    }
  }
  ...
}
```

Fig. 12. Code excerpt from Sun's `java.io.PrintStream` class. Due to the repeated synchronization on `this`, an intraprocedural analysis reports a spurious deadlock possibility while an interprocedural analysis does not

The analysis reports every *simple* cycle (also known as an elementary circuit) in a given graph. A cycle is simple if it does not visit any node more than once. Given a node that is involved in more than one simple cycle, one can construct a non-simple cycle by traversing each cycle in sequence (see Figure 11). It is possible to construct cases where a non-simple cycle causes deadlock even though the component cycles do not [26]. However, as we have never observed such a case in practice, the analysis reports only the simple cycles as a way of compressing the results. For completeness, the user should consider these cycles in combination.

4.5 Intraprocedural Weaknesses

Our analysis is interprocedural, because our experience is that an intraprocedural analysis produces too many false reports. For example, Figure 12 illustrates part of Sun's `java.io.PrintStream` class, in which both `println()` and `write()` attempt to lock `this`. An intraprocedural analysis cannot prove that the same object is locked in both methods. Thus, it reports a deadlock possibility corresponding to the case when two concurrent calls to `println()` result in different locking orders on a pair of `PrintStream` objects. However, because the objects locked are always equivalent, the second synchronization does not affect the set of locks held. This spurious report is omitted by our interprocedural analysis.

5 Reducing False Positives

Like many static analyses, our tool reports false positives. A false positive is a report that cannot possibly lead to deadlock, because (for example) it requires an infeasible aliasing relationship or an infeasible set of paths through the program. False positives reduce the usability of the tool because verifying that the report is spurious can be tedious. We have implemented sound optimizations that reduce the number of false reports without eliminating any true reports. This section describes some of the optimizations; two additional implemented optimizations handle synchronization over an object and one of its internal fields, and synchronization over method call return values [26].

5.1 Unaliased Fields

An unaliased field is one that always points to an object that is not pointed to by another variable in the program. As an optimization, our analysis detects these fields, and assigns a unique type to each of them. This can decrease the number of deadlock reports by disambiguating unaliased field references from other nodes in the lock-order graph. (It is necessary to create a node for these fields, rather than discarding information about synchronization over them. Although they have no aliases, they may still be involved in deadlock.)

The following analysis is used to discover unaliased fields. Initially, all non-public fields are assumed to be unaliased. As the analysis visits each statement in the library, that assumption is nullified for a field `f` if any of the following patterns apply:

1. `f` is assigned a non-null, non-newly-allocated expression.
2. `f` appears on the right-hand side of an assignment expression.
3. `f` appears outside any of the following expressions: a `synchronized` statement target, a comparison expression (e.g., `foo == bar`), an array access or array length expression, or as an argument to a method that does not allow it to escape.

A simple iterative escape analysis determines which arguments escape a method. Calls from the library to its methods as well as calls to the JDK are checked; arguments are assumed to escape methods where no source is available.

The analysis presented in Section 4.1 introduces a new symbolic heap object for every reference to a field. This is necessary because the analysis does not model the possible values of fields. Unaliased fields are restricted in the possible values they may hold. In particular, they are always assigned new objects, and, if they are reassigned, their old objects cannot be accessed. Because of this property, nested synchronization over the same field of a given object can be treated as a no-op (thereby eliminating spurious reports), since only one of the values locked is accessible. That is, one of the two synchronized statements is on a lock that no longer exists and should therefore be ignored. The analysis uses the same heap object for all references to the same unaliased field within a given object, thereby regarding nested synchronizations as no-ops as desired. This heap object propagates across call sites rather than being mapped to $pp_\perp$.

In addition to detecting unaliased fields, our analysis stores the set of possible runtime types of these fields. This information is readily available for unaliased fields, as they are only assigned fresh objects (created with the `new` keyword).

With this information, the analysis can determine a more precise set of possible callee methods when an unaliased field is used as a receiver.

Detecting and utilizing unaliased fields can be very beneficial. For example, this optimization reduces the number of reports from over 909 to only 1 for the jcurzez library, and from 66 to 0 for the httpunit library.

5.2 Callee/Caller Type Resolution

Accurate knowledge about dynamic types prevents locks on one object from being conservatively assumed to apply to other objects. In general, the dynamic types of arguments are a subclass of the declared parameter types; likewise, the dynamic type of the receiver is a subclass of its declared type in the caller. Callee/caller type resolution collects extra type information by leveraging the fact that the declared types of objects in callees and callers sometimes differ.

To understand the benefits of type resolution, consider the following:

```
Object o;
o.hashCode();
```

When analyzing a particular implementation of `hashCode()`, say, in class `Date`, the receiver is known to be of type `Date`, not `Object` as it was declared in the above code. The callee/caller type resolution optimization takes advantage of this information when integrating the lock-order graph for a callee such as `Date.hashCode()` into that of the caller. Instead of using the callee or caller type exclusively, the more specific type is used. This results in more precise type information in the overall lock-order graph, thereby decreasing the size of the alias sets. Type resolution can have a large impact on spurious reports: reports for the croftsoft library decrease from 1837 to 2, and reports for the jasperreports library decrease from 28 to 0.

5.3 Final and Effectively-Final Fields

For `final` fields, all references are to the same object. Our analysis takes advantage of this fact by using the same heap object for each of the references to the same `final` field within a given object. The analysis also detects fields that are *effectively-final*: non-public fields that are not assigned a value (except null) outside their constructor. Exploiting `final` fields reduces the number of reports from 46 to 32 for the Classpath library.

6 Results

We implemented our deadlock detector in the Kopi Java compiler [9], which inputs Java source code. Our benchmarks consist of 18 libraries, most of which we obtained from SourceForge and Savannah[2]. The results appear in Figure 13.

[2] ProActive [16], Jess [12], SDSU [20], and Sun's JDK [23] are not from SourceForge or Savannah, but are freely available online.

Library	Code size			Graph size		Reports	Deadlocks
	`sync`	Classes	kLOC	Nodes	Edges		
JDK 1.4	1458	1180	419	65	278	70 ∗	≥7
Classpath 0.15	754	1074	295	15	22	32 ∗	≥5
ProActive 1.0.3	199	407	63	3	3	3 ∗	≥2
Jess 6.1p6	111	125	27	12	30	23 ∗	≥0
sdsu (1 Oct 2002)	69	139	26	2	2	3 ∗	≥0
jcurzez (12 Dec 2001)	24	27	4	1	1	1	0
httpunit 1.5.4	17	117	23	0	0	0	0
jasperreports 0.5.2	11	271	67	0	0	0	0
croftsoft (09 Nov 2003)	11	108	14	1	1	2	0
dom4j 1.4	6	155	41	1	1	1	0
cewolf 0.9.8	6	98	7	0	0	0	0
jfreechart 0.9.17	5	396	125	0	0	0	0
htmlparser 1.4	5	111	22	1	1	0	0
jpcap 0.01.15	4	58	8	0	0	0	0
treemap 2.5.1	4	47	7	0	0	0	0
PDFBox 0.6.5	2	127	28	0	0	0	0
UJAC 0.9.9	1	255	63	0	0	0	0
JOscarLib 0.3beta1	1	77	6	0	0	0	0

∗ Unsound filtering heuristics used (see Section 6.3)

Fig. 13. Number of deadlock reports for each library. The table indicates the size of each library in terms of number of `synchronized` statements (given in the column labeled `sync`), number of classes (source files), and number of lines of code (in thousands). The size of the lock-order graph is measured after pruning nodes and edges that are not part of a strongly connected component. "Deadlocks" shows the numbers of confirmed deadlock cases in each library. The JDK and Classpath results are for packages in java.*. We were unable to compile 6 source files in JDK due to bugs in our research compiler

The analysis ran in less than 3 minutes per library on a 3.60GHz Pentium 4 machine. For the larger libraries, it is prohibitively expensive to compute all possible deadlock reports, so we implemented a set of unsound heuristics to filter them (see Section 6.3).

6.1 Deadlocks Found

We invoked 14 deadlocks in 3 libraries; 12 of these deadlocks were previously unknown to us. We verified each instance by writing client code that causes deadlock in the library. There are at least 7 deadlocks in the JDK, 5 in GNU Classpath, and 2 in ProActive.

As described in Section 4.4, our analysis groups reports based on the lock variables involved. Some of the deadlocks described below can be induced through calls to any of a number of different methods with the same locking pattern; we only describe a single case, and report the number of deadlocks in this conservative fashion.

Deadlocks Due to Cyclic Data Structures. Of the 14 deadlocks we found, 7 are the result of cycles in the underlying data structures. As an example, consider `java.util.Hashtable`. This class can be deadlocked by creating two `Hashtable` objects and adding each as an element of the other, i.e., by forming a cyclic relationship between the instances. In this circumstance, calling the synchronized `equals()` method on both objects in different threads can yield deadlock. The `equals()` method locks its receiver and calls `equals()` on its members, thus locking any of its internal `Hashtable` objects. When run in two threads, each of the calls to `equals()` has a different lock ordering, so deadlock can result.

Although this example may seem degenerate, the JDK `Hashtable` implementation attempts to support this cyclic structure: the `hashCode()` method prevents a potential infinite loop in such cases by preventing recursive calls from executing the hash value computation. A comment within `hashCode()` says, "This code detects the recursion caused by computing the hash code of a self-referential hash table and prevents the stack overflow that would otherwise result."

In addition to `Hashtable`, all synchronized `Collection`s and combinations of such `Collections` (e.g., a `Vector` in a cyclic relationship with a `Hashtable`) can be deadlocked in a similar fashion. This includes `Collection`s produced via calls to `Collections.synchronizedCollection()`, `Collections.synchronizedList()`, `Collections.synchronizedSortedMap()`, etc. For the purposes of reporting, all these cases are counted as a single deadlock in both the JDK and Classpath.

Deadlock resulting from cyclic data structures is quite difficult to correct. Locks must be acquired in a consistent order, or they must be acquired simultaneously. To do either of these things requires knowing which objects will be locked by calling a given method. Determining this information without first locking the container object is problematic since its internals may change during inspection. It appears that the only solution is to use a global lock for synchronizing instances of all `Collection` classes. This solution is undesirable, however, because it prevents multi-threaded uses of different `Collection` objects. Library writers may instead choose to leave these deadlock cases in place, but document their existence and describe how to appropriately use the class.

Not only do these cyclic data structures lead to deadlock, but they may also result in a stack overflow due to infinite recursion. A number of the classes having this kind of deadlock also have methods that produce unbounded recursion for the case of cyclic data structures. It seems that these deadlock cases reveal intended structural invariants (i.e., that a parent object is not reachable through its children) about the classes they involve.

The remaining 5 cyclic deadlocks are similar to that described above. Deadlock can be induced in `java.awt.EventQueue` from both JDK and Classpath, in `java.awt.Menu` from JDK, in `java.util.logging.Logger` from Classpath, and in `AbstractDataObject` from Proactive. Each class has a method that allows a cyclic relationship to be formed, and another method (or set of methods) that locks the containing object and the internal one.

Other Deadlock Cases. In addition to the cyclic case described above, ProActive exhibits a subtle deadlock in the `ProxyForGroup` class. Through a sequence of calls, the `asynchronousCallOnGroup()` method of `ProxyForGroup` can be made to lock both `this` and any other `ProxyForGroup`. Instantiating two or more `ProxyForGroup` objects and forcing each to lock the other induces deadlock. The state necessary to produce this scenario is relatively complex. The offending method contains, within four nested levels of control flow, a method call that returns an `Object`; under certain circumstances, the object returned is a `ProxyForGroup`, as needed to produce deadlock. We would not expect a library writer to notice this deadlock possibility without using a tool like ours.

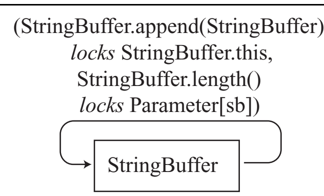
We invoked 4 additional deadlocks in the JDK. One deadlock is in `BeanContextSupport` as described in Section 1. A second deadlock is in `StringBuffer.append(StringBuffer)`, as illustrated in Figure 14. This deadlock occurs because `append()` is a synchronized method (i.e., it locks `this`), and it locks its argument. Thus, using the client code in Figure 14, if `a` is locked in thread 1, and `b` is locked in thread 2 before it is in thread 1, deadlock results. Note that this is an example of a case where only a single method is used to cause deadlock.

Another deadlock from the JDK occurs in `java.io.PrintWriter` and `java.io.CharArrayWriter`. Simplified code for this deadlock is shown in Figure 15. The `PrintWriter` and `CharArrayWriter` classes both contain a `lock` field for synchronizing I/O operations. In `PrintWriter`, the lock is set to the output stream `out`, while in `CharArrayWriter`, the lock is set to `this`.

The last deadlock in the JDK is located in `java.awt.dnd.DropTarget`. This class can be deadlocked by calling `setComponent()` with an argument (of type `Component`) having a valid `DropTarget` set. When this call is made, the receiver is locked followed by the argument's `DropTarget`. Thus, the code in Figure 16 can lead to deadlock.

GNU Classpath exhibits 2 deadlocks besides those described so far. The first is in `StringBuffer`, and is analogous to the JDK bug described above. The second is in `java.util.SimpleTimeZone`. The `SimpleTimeZone.equals(Object)` method is synchronized and locks its argument; it is therefore susceptible to the same style of deadlock as that of `StringBuffer.append()`.

```
class StringBuffer {
 synchronized StringBuffer
   append(StringBuffer sb) {
     ...
     // length() is synchronized
     int len = sb.length();
     ...
 }
}
```




```
StringBuffer a =
  new StringBuffer();
StringBuffer b =
  new StringBuffer();

thread 1:  a.append(b);
thread 2:  b.append(a);
```

Fig. 14. Library code, lock-order graph, and client code that deadlocks JDK's `StringBuffer` class. This deadlock is also present in Classpath

```
class PrintWriter {
  PrintWriter(OutputStream o) {
    lock = o;
    out = o;
  }

  void write(char buf[],
      int off, int len) {
    synchronized (lock) {
      out.write(buf, off, len);
    }
  }
}
```

```
class CharArrayWriter {
  CharArrayWriter() {
    lock = this;
  }

  void writeTo(Writer out) {
    synchronized (lock) {
      out.write(buf, 0,
                count);
    }
  }
}
```

```
// c.lock = c
c = new CharArrayWriter();
// p1.lock = c
p1 = new PrintWriter(c);
// p2.lock = p1
p2 = new PrintWriter(p1);

thread 1:  p2.write("x",0,1);
thread 2:  c.writeTo(p2);
```

Fig. 15. Simplified library code from `PrintWriter` and `CharArrayWriter` from Sun's JDK, and, on the right, client code that causes deadlock in the methods. In thread 1, `p1` is locked first, then `c`; in thread 2, `c` is locked, then `p1`. Because the locks are acquired in different orders, deadlock occurs under some thread interleavings

```
DropTarget a = new DropTarget(), b = new DropTarget();
Component aComp = new Button(), bComp = new Button();

aComp.setDropTarget(a);
bComp.setDropTarget(b);

thread 1:  a.setComponent(bComp);
thread 2:  b.setComponent(aComp);
```

Fig. 16. Client code that induces deadlock in the JDK's `DropTarget` class

It is interesting to note that JDK and Classpath implementations of `SimpleTimeZone` and `Logger` differ in their locking behavior: it is not possible to invoke deadlock in these classes using the JDK. Similarly, the Classpath implementations of `PrintWriter` and `CharArrayWriter` do not deadlock; other relevant portions of Classpath are not fully implemented.

Fixing Deadlocks. There are a number of viable solutions to the deadlocks presented above. The methods performing synchronization could be written to acquire the needed locks in a set order. Java could be extended with a synchronization primitive to atomically acquire multiple locks. A utility routine could be written to accomplish the same effect as this primitive, taking as arguments a list of locks to acquire and a thunk to execute, then acquiring the locks in a fixed order. These solutions require knowledge of the set of locks to be acquired. Sometimes this is immediately apparent from the code; otherwise, a method that determines the locks required for an operation could be added to an interface. In all these cases, the implementation could order the locks using `System.identityHashCode()`, breaking ties arbitrarily but consistently. Note however, that these solutions assume that the needed locks will not change while they are being determined. If they might change, it may be necessary to use a global lock for the classes involved in the deadlock.

6.2 Verifying Deadlock Freedom

Using our tool, we verified 13 libraries to be free from the class of deadlocks described in Section 3.2. Note that these libraries may perform callbacks to client code, some extend the JDK, and most perform reflection; our technique does not model synchronization resulting from these behaviors. For 10 of these libraries, the verification is fully automatic, with 0 reports from our tool. Across the other 3 libraries, our tool reports a total of 4 deadlocks, which we manually verified to be false positives.

The false report in jcurzez is for a scenario in which an internal field `f` of the same type as its containing class is set to a parameter of the constructor. To eliminate this report, the analysis would have to combine several facts and additional optimizations. Croftsoft gives two spurious reports because an object involved in the synchronization cannot have the runtime type that our tool conservatively assumes to be possible. The final report is for dom4j, and is spurious because of infeasible control flow.

6.3 Unsound Filtering Heuristics

For the larger libraries, the number of reports given by our algorithm is too high (more than 100,000 for the JDK) for each to be considered by hand. In addition, it is computationally demanding to report every deadlock possibility. In order to make the tool more usable for large libraries (both in terms of number of reports and time needed to gather them) our tool uses unsound filtering heuristics. These heuristics aim to identify reports that have the greatest likelihood of representing a true deadlock. However, as unsound heuristics, they also have the potential to eliminate true deadlock cases from consideration.

Our tool applies two filtering heuristics on certain of the libraries in Figure 13. One heuristic is to restrict attention to cycles in the lock-order graph that are shorter than a given length. For the filtered libraries, only cycles with two or fewer nodes were reported. Shorter cycles contain fewer locks, and are easier to examine manually. In addition, shorter cycles might be more likely to correspond to actual deadlocks, as each edge in a cycle represents a pair of lock acquisitions that has some chance of being infeasible (due to infeasible control flow or aliasing relationships).

The second filtering heuristic is to assume that the runtime type of each object is the same as its declared type. This reduces the number of reports in two ways. First, the analysis ceases to account for dynamic dispatch, as it assumes that there is exactly one target of each method call. This causes the lock-order graph for a given method to be integrated at fewer call-sites, thereby decreasing the number of edges in the overall graph. Second, this heuristic causes the **top_level** routine (Figure 8) to forgo expansion of each edge into edges between all possible subclasses. This heuristic has some intuitive merit because it restricts attention to code that operates on a specific type, rather than a more general type. For example, it considers the effects of all synchronized methods of a given class, but it eliminates the assumption that all objects could be aliased with a field of type `Object` that may be locked elsewhere.

7 Related Work

The long-standing goal of ensuring that concurrent programs are free of deadlock remains an active research focus. Mukesh reviews the various approaches [22].

Several researchers have developed static deadlock detection tools for Java using lock-order graphs [17, 1, 24]. To the best of our knowledge, the Jlint static checker [17] is the first to use a lock-order graph. The original implementation of Jlint considers only `synchronized` methods; it does not model `synchronized` statements. Artho and Biere [1] augment Jlint with limited support for `synchronized` statements. However, their analysis does not report all deadlock possibilities. It only considers cases they reason are most fruitful for finding bugs: 1) all fields and local variables are assumed to be unaliased, meaning that two threads must lock exactly the same variable to elicit a deadlock report, 2) nested `synchronized` blocks are tracked only within a single class, not across methods in different classes, and 3) inheritance is not fully considered.

von Praun detects deadlock possibilities in Java programs using a lock-order graph and context-sensitive lock sets [24–pp.105–110]. Our analysis was developed independently [25]. While von Praun's alias analysis is more sophisticated than ours, it is unclear how to adapt it to model all possible calls to a library. Also, in an effort to reduce false positives, the analysis suppresses reports in which all locks belong to the same alias set; as a consequence, it does not find 12 of the 14 deadlocks exposed by our tool. While von Praun's analysis could be trivially modified to report such cases, it would then report, in addition, all of the benign cases that repeatedly lock a single object (as in Figure 12). Suppressing these reports is the motivation for the flow-sensitive and interprocedural aspects of our analysis: our analysis can recognize that two object references are identical, thereby qualifying repeated synchronizations on a given object as benign. von Praun's analysis does not offer this benefit, in part because it is flow-insensitive and unification-based. Also, it does not consider that `wait()` can introduce a cyclic locking pattern (as in Figure 9). Our tool reports all deadlock possibilities.

RacerX [10] is a flow-sensitive, context-sensitive tool for detecting deadlocks and race conditions in C systems code. Because our tool analyzes Java instead of C, it operates under a different set of constraints. We fully account for objects and inheritance, reporting all deadlock possibilities; RacerX operates on a procedural language, and might fail to report every deadlock case due to function pointers and high-overhead functions. Our tool analyzes unmodified Java code, while RacerX requires annotations to indicate the locking behavior of system-specific C functions. Our tool exploits the hierarchical synchronization primitives in Java; in C, precision is sacrificed due to the decoupling of lock and unlock operations (sometimes on different paths of the same function, as noted by the authors).

Several groups have taken a model-checking approach to finding deadlock in Java programs. Demartini, Iosif, and Sisto [8] translate Java into the Promela language, for which the SPIN model checker verifies deadlock freedom. Their verification reports all deadlock possibilities so long as the program does not exceed the maximum number of modeled objects or threads.

Java Pathfinder also performs model checking by translating Java to Promela, including support for exceptions and polymorphism [13]. It has also been used to analyze execution traces; a deadlock vulnerability is reported if two threads obtain locks in a different order at runtime [14]. This approach can detect "gate locks": a shared lock that guards each thread's entry into a hazardous out-of-order locking sequence, thereby preventing deadlock. The technique has evolved into a general online monitoring environment called Java PathExplorer [15].

Breuer and Valls describe static detection of deadlock in the Linux kernel [3]. They target deadlocks caused by threads that call `sleep` while still holding a spinlock. Chaki et al. [4] use counterexample-guided abstraction refinement and the MAGIC verification tool [5] to detect deadlock in message-passing C programs. The technique is compositional and efficient (compared to traditional model checking) because the abstraction for each thread can be refined independently until the overall system exhibits a bug or is proven free of deadlock. However, the number of threads and locks (and their interaction) must be known statically.

The Ada programming language allows rendezvous communication between a call statement in one task and an accept statement in another. Most analyses for Ada aim to verify that rendezvous communication succeeds, rather than considering the order of synchronization on shared resources (locks). For example, Masticola and Ryder [19] give a polynomial-time algorithm for reporting all possible rendezvous deadlocks for a subset of Ada (they also report false positives). Corbett [7] evaluates three methods for finding deadlock in Ada programs. Many analyses rely on the common case where Ada tasks are fixed and initiated together, in contrast to Java threads which are always created dynamically.

Boyapati, Lee, and Rinard [2] augment Java with ownership types to ensure deadlock freedom at compile time. While this is an elegant solution, it requires translating existing programs to use new type annotations, and some computations might be hard to express. Flanagan and Qadeer describe a type and effect system for atomicity [11]. In this system, a method is atomic if it appears to execute serially, without interleaving of other threads. They identify an atomicity violation in `StringBuffer.append`, providing part of the impetus for our work.

Zeng and Martin augment a Java Virtual Machine with a deadlock avoidance mechanism [28]. This technique constructs a lock-order graph dynamically, tracking the actual objects that are locked during execution. As cycles form in the graph, "ghost locks" are introduced to prevent multiple threads from entering the cyclic regions. While this avoids deadlock later in the execution, deadlock could still occur while the graph is being built.

Zeng describes a system that uses exceptions to indicate various kinds of deadlock in a Java Virtual Machine [27]. Such a mechanism allows a client to intelligently respond to deadlock in a library component. Pulse [18] is an operating system mechanism that detects general deadlocks via speculative execution of blocked processes. There is also a large body of work on dynamically detecting deadlock in the context of databases and distributed systems [21, 22].

8 Conclusions

Library writers wish to ensure their libraries are free of deadlock. Because this assurance is difficult to obtain by testing or by hand, a tool for identifying possible deadlock (or verifying freedom from deadlock) is desirable. Model checking is a possible approach to the problem, but the well-known state explosion problem makes it impractical for most libraries.

We have presented a flow-sensitive, context-sensitive analysis for static detection of deadlock in Java libraries. Out of 18 libraries, we verified 13 to be free of deadlock, and found 14 reproducible deadlocks in 3 libraries. The analysis uses lock-order graphs to represent locking configurations extracted from libraries. Nodes in these graphs represent alias sets, edges represent possible lock orderings, and cycles indicate possible deadlocks.

Our analysis is quite effective at verifying deadlock freedom and finding deadlock, but it still produces a sizable number of false reports. Rather than asking the user to investigate these reports, the reports could be dispatched to a model checker which could automatically check for deadlock. In this framework, our tool would serve to limit the search space of the model checker, possibly allowing sound verification of large libraries.

Just as static verification of all possible program executions offers stronger guarantees than dynamic analysis of one or a few executions, verification that a library cannot deadlock is preferable to checking that a particular client program does not deadlock while using the library. To our knowledge, our tool is the first to address the problem of deadlock detection in libraries. However, the technique is also applicable to whole programs, and may prove to be effective in that context.

Acknowledgments

We thank Viktor Kuncak, Manu Sridharan, Huu Hai Nguyen, Wilson Hsieh, and Stephen McCamant for their feedback and suggestions on this work, and Martin Lackner for support with Kopi. The second author thanks Saman Amarasinghe for supporting his participation in this project. We also thank the anonymous reviewers for their comments. This work is supported in part by NSF grant CCR-0133580, the MIT-Oxygen Project, and an NSF Graduate Research Fellowship.

References

1. Artho, C., Biere, A.: Applying static analysis to large-scale, multi-threaded Java programs. In: ASWEC. (2001) 68–75
2. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA. (2002) 211–230
3. Breuer, P.T., Garcia-Valls, M.: Static deadlock detection in the Linux kernel. In: Ada-Europe. (2004) 52–64
4. Chaki, S., Clarke, E., Ouaknine, J., Sharygina, N.: Automated, compositional and iterative deadlock detection. In: MEMOCODE. (2004)
5. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE TSE **30** (2004) 388–402
6. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: PLDI. (1990)
7. Corbett, J.C.: Evaluating deadlock detection methods for concurrent software. IEEE TSE **22** (1996) 161–180
8. Demartini, C., Iosif, R., Sisto, R.: A deadlock detection tool for concurrent Java programs. Software: Practice and Experience **29** (1999) 577–603
9. DMS Decision Management Systems GmbH: The Kopi Project (2004) `http://www.dms.at/kopi/`.
10. Engler, D., Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks. In: SOSP. (2003) 237–252
11. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: POPL. (2003) 338–349
12. Friedman-Hill, E.: Jess, the Java expert system shell (2004) `http://herzberg.ca.sandia.gov/jess/`.
13. Havelund, K., Pressburger, T.: Model checking Java programs using Java PathFinder. STTT **2** (2000) 366–381
14. Havelund, K.: Using runtime analysis to guide model checking of Java programs. In: SPIN. (2000) 245–264
15. Havelund, K., Roşu, G.: Monitoring Java programs with Java PathExplorer. In: RV. (2001)
16. INRIA: Proactive (2004) `http://www-sop.inria.fr/oasis/ProActive/`.
17. Knizhnik, K., Artho, C.: Jlint (2005) `http://jlint.sourceforge.net/`.
18. Li, T., Ellis, C.S., Lebeck, A.R., Sorin, D.J.: Pulse: A dynamic deadlock detection mechanism using speculative execution. In: USENIX Technical Conference. (2005) 31–44
19. Masticola, S.P., Ryder, B.G.: A model of Ada programs for static deadlock detection in polynomial time. Workshop on Parallel and Distributed Debugging (1991)
20. San Diego State University: SDSU Java library (2004) `http://www.eli.sdsu.edu/java-SDSU/`.
21. Shih, C.S., Stankovic, J.A.: Survey of deadlock detection in distributed concurrent programming environments and its application to real-time systems. Technical report, UMass UM-CS-1990-069 (1990)
22. Singhal, M.: Deadlock detection in distributed systems. IEEE Computer **22** (1989) 37–48
23. Sun Microsystems, Inc.: Java Development Kit (2004) `http://java.sun.com/`.
24. von Praun, C.: Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs. PhD thesis, Swiss Federal Institute of Technology, Zurich (2004)

25. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection in Java libraries. Research Abstract #102, MIT Computer Science and Artificial Intelligence Laboratory (February, 2004)
26. Williams, A.L.: Static detection of deadlock for Java libraries. Master's thesis, MIT Dept. of EECS (2005)
27. Zeng, F.: Deadlock resolution via exceptions for dependable Java applications. In: DSN. (2003) 731–740
28. Zeng, F., Martin, R.P.: Ghost locks: Deadlock prevention for Java. In: MASPLAS. (2004)