An Effective Dynamic Analysis for Detecting Generalized Deadlocks

Pallavi Joshi
EECS, UC Berkeley, CA, USA
pallavi@cs.berkeley.edu

Mayur Naik
Intel Labs Berkeley, CA, USA
mayur.naik@intel.com

Koushik Sen
EECS, UC Berkeley, CA, USA
ksen@cs.berkeley.edu

David Gay
Intel Labs Berkeley, CA, USA
dgay@acm.org

ABSTRACT

We present an effective dynamic analysis for finding a broad class of deadlocks, including the well-studied lock-only deadlocks as well as the less-studied, but no less widespread or insidious, deadlocks involving condition variables. Our analysis consists of two stages. In the first stage, our analysis observes a multi-threaded program execution and generates a simple multi-threaded program, called a *trace program*, that only records operations observed during the execution that are deemed relevant to finding deadlocks. Such operations include lock acquire and release, wait and notify, thread start and join, and change of values of user-identified synchronization predicates associated with condition variables. In the second stage, our analysis uses an off-the-shelf model checker to explore all possible thread interleavings of the trace program and check if any of them deadlocks. A key advantage of our technique is that it discards most of the program logic which usually causes state-space explosion in model checking, and retains only the relevant synchronization logic in the trace program, which is sufficient for finding deadlocks. We have implemented our analysis for Java, and have applied it to twelve real-world multi-threaded Java programs. Our analysis is effective in practice, finding thirteen previously known as well as four new deadlocks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Verification

Keywords

deadlock detection, dynamic program analysis, concurrency

1. INTRODUCTION

A deadlock in a multi-threaded program is an unintended condition in which one or more threads block forever waiting for a synchronization event that will never happen. Deadlocks are a common problem in real-world multi-threaded programs. For instance, 6,500/198,000 ($\sim$ 3%) of the bug reports in the bug database at `http://bugs.sun.com` for Sun's Java products involve the keyword "deadlock" [11]. Moreover, deadlocks often occur non-deterministically, under very specific thread schedules, making them harder to detect and reproduce using conventional testing approaches. Finally, extending existing multi-threaded programs or fixing other concurrency bugs like races often involves introducing new synchronization, which, in turn, can introduce new deadlocks. Therefore, deadlock detection tools are important for developing and testing multi-threaded programs.

There is a large body of work on deadlock detection in multi-threaded programs, including both dynamic analyses [2, 3, 7, 8, 11] and static analyses [4, 5, 10, 14, 15, 18, 19]. Most of these approaches exclusively detect *resource deadlocks*, a common kind of deadlock in which a set of threads blocks forever because each thread in the set is waiting to acquire a lock held by another thread in the set. Specifically, these techniques check if the program follows a common idiom, namely, that there is no cycle in the lock-order graph consisting of nodes corresponding to each lock and edges from l_1 to l_2 where lock l_2 could be acquired by a thread while holding lock l_1. Dynamic analyses predict violations of this idiom by analyzing multi-threaded executions of the program that do not necessarily deadlock, whereas static analyses do so by analyzing the source code or an intermediate representation of the source code of the program.

However, deadlocks besides resource deadlocks, namely *communication deadlocks* that result from incorrect use of condition variables (i.e. wait-notify synchronization), as well as deadlocks resulting from unintended interaction between locks and condition variables, are no less widespread or insidious than resource deadlocks. From the user perspective, deadlocks are harmful regardless of the reason that causes the involved set of threads to block forever. Yet, in the extensive literature on deadlock detection only little work (e.g. [1, 10]) addresses communication deadlocks.

We initially set out to devise a dynamic analysis to predict communication deadlocks by checking an idiom analogous to that for resource deadlocks. However, after studying a large number of communication deadlocks in real-world programs, we realized that there is no single idiom that programmers follow when writing code using condition variables. Therefore, any such dynamic analysis based on checking idioms would give many false positives and false negatives. The study also suggested that finding communication deadlocks

FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.

is a hard problem as a communication deadlock can be non-trivially dependent on aspects of the underlying synchronization logic that vary from program to program.

In this paper, we present a novel dynamic analysis, called CheckMate, to predict a broad class of deadlocks subsuming resource deadlocks, communication deadlocks, and deadlocks involving both locks and condition variables. CheckMate, instead of checking conformance to a particular idiom, creates a simple multi-threaded program, called a *trace program*, by observing an execution of the original program, and model checks the *trace program* to discover potential deadlocks. The trace program for a given multi-threaded execution creates an explicit thread for each dynamic thread created by the execution. The code for each thread in the trace program consists of the sequence of lock acquires and releases, wait and notify calls, and thread start and join calls by the corresponding thread in the original execution. However, this is not enough: the state of the *synchronization predicates* associated with condition variables must also be tracked and checked. In the trace programs these synchronization predicates are represented as boolean variables with explicit assignments where the original program performed an assignment that caused the predicate's value to change, and explicit checks before uses of wait and notify. All other operations performed by a thread in the original execution, such as method calls, assignments, and expression evaluations, are not incorporated in the trace program. In summary, a trace program captures the dynamic synchronization pattern exhibited by the original program execution. CheckMate then model checks, i.e. explores all interleavings of, the trace program and checks if it could deadlock. If a deadlock is discovered, CheckMate maps the interleaving back to the original program, and reports it as a potential deadlock in the original program.

CheckMate analyzes Java programs, but Java does not make synchronization predicates explicit: any predicate could be the synchronization predicate for a condition variable, making it difficult to automatically identify them. Synchronization predicates could be inferred statically; however, we do not focus on such analysis in this paper. Instead, CheckMate requires that programmers identify synchronization predicates using a lightweight annotation mechanism (Section 4.1). CheckMate uses these annotations to observe all changes to values of synchronization predicates. Note that manual annotations are not necessary if the language makes synchronization predicates explicit.

In summary, the key contributions of CheckMate are:

- A key insight underlying our approach is that model checking applied to the trace program is much more likely to scale than if it were applied to the original program. This is because the trace program discards all data operations and computations performed by the program and only retains operations that are deemed relevant to finding deadlocks, namely synchronization operations and changes to values of synchronization predicates. Moreover, it is generated by observing a single finite execution of the original program. Therefore, a trace program has a much smaller (and finite) state space than the original program and is more tractable to model check.
- A key benefit of our approach is that it can find any kind of deadlock irrespective of whether the program follows a recommended idiom or not.
- Another benefit of our model checking-based approach over idiom checking-based approaches is that our approach provides detailed counterexamples explaining the deadlocks it reports—a feature we have found particularly useful for debugging communication deadlock.
- We believe that our idea of capturing the essential synchronization skeleton of a program execution using a trace program is novel, and could be applied to discover other kinds of concurrency bugs, although such applications might need to extend a trace program with more kinds of operations.

Our analysis is not sound because a deadlock in the trace program may not be feasible in the original program. Our analysis is also not complete as we construct the trace program by observing a single execution. Note that this is true of most predictive dynamic analyses such as Eraser [16] and Goodlock [8]. However, we have found our analysis to be effective in practice. We have implemented it and applied it to twelve real-world multi-threaded Java benchmarks. It has low run-time overhead, and generates trace programs that are model checked efficiently using Java PathFinder (JPF) [17, 9], an explicit state model checker for Java bytecode. It also has low false positive and false negative rates, and detects thirteen previously known as well as four new deadlocks in our benchmarks.

2. RATIONALE

In this section, we explain the rationale behind our approach after describing the recommended usage pattern for condition variables.

Recommended Usage

Figure 1 shows the recommended pattern for using condition variables in Java.[1] The condition variable in the figure is associated with a synchronization predicate b (e.g. "FIFO X is non-empty"). Any code like $F1$ that requires b to be true (e.g. "dequeuing an element from FIFO X") must hold the condition variable's lock l, and repeatedly wait (l.wait()) on the condition variable until b is true. Any code $F2$ that might make b true (e.g. "enqueue an element in FIFO X") must make these modifications while holding the condition variable's lock l, and must notify all threads that might be waiting for b to become true (l.notifyAll()). After a waiting thread wakes up, it should again check to see if b indeed holds for two reasons. First, the notifying thread ($F2$) may notify when b is not necessarily true. Second, some other thread may have invalidated b before the woken thread was able to acquire lock l.

Our Initial Effort

We devised a dynamic analysis to predict communication deadlocks using thread interleavings that did not exhibit the deadlocks. We checked to see if all condition variables used in such interleavings followed the recommended usage pattern shown in Figure 1. Consider the real-world code fragment in Figure 2. Any of its interleavings violates two aspects of that pattern: first, thread 1 uses an `if` instead of a `while` to check predicate b, and secondly, neither thread

[1]Strictly speaking, Java uses monitors that combine a lock and a condition variable. We use lock and condition variable to clarify to which aspect of a monitor we are referring; this also makes clearer how CheckMate would apply to languages with separate locks and condition variables.

```
   // F1                        // F2
synch (l) {                synch (l) {
  while (!b)                 <change in value
    l.wait();                of b that could
  <do something that         make b true>
  requires b = true>         l.notifyAll();
}                          }
```

Figure 1: Recommended condition variable usage. We use synch to abbreviate synchronized.

```
// Thread 1                // Thread 2
if (!b)

                           b = true;
                           synch (l)
                             l.notifyAll();
  synch (l)
    l.wait();
```

Figure 2: Deadlock due to missed notification.

accesses b in the same synchronized block as the one containing l.`wait()` or l.`notifyAll()`. The analysis we devised thus reports a possible deadlock in this code fragment regardless of the interleaving it observes. Indeed, the shown interleaving of this code fragment exhibits a deadlock. In this interleaving, thread 1 first finds that boolean b is false. Thread 2 then sets b to true, notifies all threads in the wait set of l (i.e. the threads that are waiting on l), and releases l. The wait set of l, however, is empty; in particular, thread 1 is not (yet) waiting on l. Finally, thread 1 resumes and waits on l, assuming incorrectly that b is still false, and blocks forever as thread 2—the only thread that could have notified it—has already sent a notification and terminated. This is a classic kind of communication deadlock called a *missed notification* in Java.

Limitations of Pattern Enforcement

We found pattern-based enforcement to be of limited value, for two reasons. First, programmers often optimize the recommended pattern based on their knowledge about the code. For instance, if the synchronization predicate b is always true when a thread is woken up, then the thread may not need to check b again, i.e. the `while` in $F1$ can be an `if`. Or, if a number of threads are woken up by a notifying thread, but the first thread that acquires lock l always falsifies the predicate b, then waking up the other threads is pointless. In this case, the notifying thread ($F2$) can use `notify` to wake a single thread. A real-world example violating the pattern is found in `lucene` (version 2.3.0), a text search engine library by Apache: in some cases, a thread may change the value of a synchronization predicate in one synchronized block and invoke `notifyAll()` in another synchronized block. Although the notification happens in a different synchronized block, it always follows the change in value of the predicate, and hence there is no deadlock because of this invariant which the recommended usage pattern does not capture.

Second, code that respects the pattern can still deadlock because of interactions between other locks and condition variables. Consider the code in Figure 3 (based on a real example): the locks follow the cycle-free lock-order graph idiom for avoiding resource deadlocks, and the condition variables follow the recommended usage pattern in Figure 1 for avoiding communication deadlocks, yet their combined use causes a deadlock, exhibited by the shown interleaving. This code fragment can occur because a library uses the condition variable involving lock `l2`, and an application calls into the

```
// Thread 1          // Thread 2
synch (l1)
  synch (l2)
    while (!b)
      l2.wait();
                     synch (l1)
                       synch (l2)
                         l2.notifyAll();
```

Figure 3: Deadlock involving both locks and condition variables.

library while holding its own lock `l1`. In fact, this pattern occurs frequently enough in practice that FindBugs, a popular static bug-finding tool for Java that checks common bug patterns, reports that calls to `wait()` with two locks held may cause a deadlock [10].

In summary, we could not find an idiom for accurately predicting all deadlocks by observing interleavings that did not exhibit them. This motivated us to devise an analysis that uses a model checker to explore all possible interleavings. Model checking is difficult to scale to large programs. We chose to strike a trade-off between scalability, completeness, and soundness by model checking a *trace program* obtained from a single execution of the given program. In doing so, we sacrifice both completeness and soundness, but our analysis scales to large programs. This is not only because the trace program is generated from a single finite execution of the given program, but also because it only records operations that we deem are relevant to finding the above kinds of deadlocks. Not only does our analysis find both deadlocks discussed above, it does not report a false deadlock for the correct usage of notification in `lucene` described above.

3. OVERVIEW

In this section, we illustrate our analysis using the example Java program in Figure 4. Class `MyBuffer` is intended to implement a thread-safe bounded buffer that allows a producer thread to add elements to the buffer and a consumer thread to remove elements from it. List `buf` represents the buffer, `cursize` denotes the current number of elements in the buffer, and `maxsize` denotes the maximum number of elements allowed in the buffer at any instant. Ignore the underlined field `condition` and all operations on it for now. The program uses a condition variable with two associated synchronization predicates to synchronize the producer and consumer threads. The first predicate checks that the buffer is full (method `isFull()`), and the second that the buffer is empty (method `isEmpty()`).

A producer thread adds elements to the buffer using the `put()` method. If the buffer is full, it waits until it gets notified by a consumer thread. After adding an element to the buffer, it notifies any consumer thread that may be waiting for elements to be available in the buffer. Likewise, a consumer thread removes elements from the buffer using the `get()` method. If the buffer is empty, it waits until it gets notified by a producer thread. After removing an element from the buffer, if the buffer was full, it notifies any producer thread that may be waiting for space to be available in the buffer. Finally, the `resize()` method allows changing the maximum number of elements allowed in the buffer.

The `main()` method creates a `MyBuffer` object `bf` with a `maxsize` of 1. It also creates and spawns three threads that execute in parallel: a producer thread `p` that adds two integer elements to `bf`, a consumer thread `c` that removes

an element from bf, and a third thread r that resizes bf to have a maxsize of 10.

```
1 public class MyBuffer {
2   private List buf = new ArrayList();
3   private int cursize = 0, maxsize;
4   private ConditionAnnotation condition =
5     new ConditionAnnotation(this) {
6       public boolean isConditionTrue() {
7         return ((MyBuffer) o).isFull();
8       }
9     };
10  public MyBuffer(int max) {
11    maxsize = max;
12  }
13  public synch void put(Object elem) {
14    condition.waitBegin(this);
15    while (isFull())
16      wait();
17    condition.waitEnd();
18    buf.add(elem);
19    cursize++;
20    notify();
21  }
22  public Object get() {
23    Object elem;
24    synch (this) {
25      while (isEmpty())
26        wait();
27      elem = buf.remove(0);
28    }
29    synch (this) {
30      condition.notifyBegin(this);
31      if (isFull()) {
32        cursize--;
33        notify();
34      } else
35        cursize--;
36      condition.notifyEnd();
37    }
38    return elem;
39  }
40  public synch void resize(int max) {
41    maxsize = max;
42  }
43  public synch boolean isFull() {
44    return (cursize >= maxsize);
45  }
46  public synch boolean isEmpty() {
47    return (cursize == 0);
48  }
49  public static void main(String[] args) {
50    final MyBuffer bf = new MyBuffer(1);
51    Thread p = (new Thread() {
52      public void run() {
53        for (int i = 0; i < 2; i++)
54          bf.put(new Integer(i));
55      }
56    }).start();
57    Thread r = (new Thread() {
58      public void run() { bf.resize(10); }
59    }).start();
60    Thread c = (new Thread() {
61      public void run() { bf.get(); }
62    }).start();
63  }
64 }
```

Figure 4: Example with a communication deadlock.

Suppose we execute the program, and the three threads spawned by the main thread interleave as shown in Figure 5. In this interleaving, thread p first puts integer 0 into bf. Since the maxsize of bf is 1, bf is now full. But before p puts another integer into bf, thread r changes the maxsize of bf to 10. Thus, bf is not full any more. Thread p then puts integer 1 into bf. Finally, thread c removes integer 0 from bf. Note that neither of the two wait()'s in the program is executed in this interleaving. However, there is another interleaving of threads p, r, and c that deadlocks. This interleaving is shown in Figure 6. Thread p puts integer 0 into bf. Since the maxsize of bf is 1, bf gets full. When p tries to put another integer into bf, it executes the wait() in the put() method and blocks. Thread r then increases the maxsize of bf, and thus, bf is not full any more. Thread c then removes integer 0 from bf. Since bf is not full any more (as thread r grew its capacity), it does not notify thread p. Thus, p blocks forever.

Our analysis can predict the deadlock from the interleaving in Figure 5, although that interleaving does not exhibit the deadlock, and does not even execute any wait(). For this purpose, our analysis records three kinds of information during the execution of that interleaving. First, it records synchronization events that occur during the execution, like lock acquires and releases, calls to wait() and notify(), and calls to start() and join() threads. Secondly, it records changes to the value of any predicate associated with a condition variable during the execution. Since Java has no explicit synchronization predicates associated with condition variables, our analysis requires the user to explicitly identify each such predicate by defining an instance of class ConditionAnnotation (shown in Figure 8). In our example in Figure 4, there are two condition variables in class MyBuffer, one for predicate isFull(), and the other for predicate isEmpty(). We manually annotate the MyBuffer class with the underlined field condition defined on lines 4-9 to identify predicate isFull(). This field holds a ConditionAnnotation object that defines a method isConditionTrue() that can determine in any program state whether that predicate is true. Our analysis uses this method to determine if each write in the observed execution changes the value of predicate isFull(). We provide a similar annotation (not shown for brevity) for predicate isEmpty(). Note that our annotations are very simple to add if we know the implicit synchronization predicates associated with condition variables. These annotations can also be inferred automatically using static analysis, but we leave that to future work.

Thirdly, our analysis also records each wait() and notify() event that did not occur during the observed execution because the condition under which it would have occurred was false in that execution. Our analysis again relies on manual annotations for this purpose, this time in the form of calls to pre-defined methods waitBegin(), waitEnd(), notifyBegin(), and notifyEnd() on the ConditionAnnotation object corresponding to the predicate associated with the condition. The annotations on lines 14 and 17 denote that the execution of wait() in the put() method depends on the value of predicate isFull(). During execution, even if this predicate is false, these annotations enable our analysis to record that had it been true, the wait() would have executed. Likewise, the annotations on lines 30 and 36 denote that the execution of notify() in the get() method depends on the value of predicate isFull(). Similar annotations (not shown for brevity) are added to handle the use of the isEmpty() predicate.

Our analysis generates the Java program shown in Fig-

```
// Thread p    // Thread r    // Thread c
bf.put(0)
                bf.resize(10)
bf.put(1)
                                    bf.get()
```

Figure 5: Non-deadlocking interleaving for Figure 4.

```
// Thread p    // Thread r    // Thread c
bf.put(0)
bf.put(1)
                bf.resize(10)
                                    bf.get()
```

Figure 6: Deadlocked interleaving for Figure 4.

```
1 public class TraceProgram {
2   static Object bf = new Object();
3   static boolean isFull;
4   static Thread main = new Thread() {
5     public void run() {
6       isFull = false;
7       p.start();
8       r.start();
9       c.start();
10    }
11  };
12  static Thread p = new Thread() {
13    public void run() {
14      synch (bf) {     // enter bf.put(0)
15        if (isFull) {
16          synch (bf) { bf.wait(); }
17        }
18        isFull = true;
19        bf.notify();
20      }                // leave bf.put(0)
21      synch (bf) {     // enter bf.put(1)
22        if (isFull) {
23          synch (bf) { bf.wait(); }
24        }
25        bf.notify();
26      }                // leave bf.put(1)
27    }
28  };
29  static Thread r = new Thread() {
30    public void run() {
31      synch (bf) { // enter bf.resize(10)
32        isFull = false;
33      }              // leave bf.resize(10)
34    }
35  };
36  static Thread c = new Thread() {
37    public void run() {
38      synch (bf) {     // enter bf.get()
39        if (isFull) {
40          synch (bf) { bf.notify(); }
41        }
42      }                // leave bf.get()
43    }
44  };
45  public static void main(String[] args) {
46    main.start();
47  }
48 }
```

Figure 7: Trace program generated by observing the execution of the interleaving in Figure 5 of the example in Figure 4.

ure 7, which we call a *trace program*, by observing the execution of the interleaving in Figure 5 and with the help of the above annotations. Note that the trace program has excluded all the complex control structure (e.g. the for loop and method calls) and memory updates (e.g. changes in `cursize` and `buf`) and has retained the necessary synchronization operations that happened during the execution. This simple trace program without the complicated program logic of the original program is much more efficient to model check.

In the trace program, we have used descriptive identifier names and comments to help relate it to the original program. Such comments and identifier names help our analysis to map any error trace in the trace program to the original program, which could be used for debugging. The fact that our analysis can generate an informative error trace in the original program is a key advantage of our technique over other predictive dynamic analysis techniques. In the trace program, `bf` denotes the instance of `MyBuffer` created during the observed execution. Note that we make `bf` of type `Object`, instead of type `MyBuffer`, because we do not need to worry about the program logic in the trace program. `isFull` denotes predicate `bf.isFull()` upon which the `wait()` in the `put()` method and `notify()` in the `get()` method are control-dependent. The main thread `main` initializes `isFull` to false, and starts threads `p`, `r`, and `c` as in the observed execution. Note that although the `wait()` in the `put()` method is not executed in either of the two calls to `bf.put()` by thread `p` in that execution, the `run()` method of thread `p` in the trace program records that this `wait()` would have been executed in either call had `isFull` been true. Also, `isFull` is set to true on line 18 since the buffer becomes full after thread `p` puts the first integer 0 into it. Thread `r` is the thread that resizes the buffer and increases its `maxsize`. The `run()` method of thread `r` sets `isFull` to false on line 32 since the buffer is no longer full after its `maxsize` has been increased. Finally, although the `notify()` in the `get()` method is not executed in the call to `bf.get()` by thread `c` in the observed execution, the `run()` method of thread `c` in the trace program records that the `notify()` would have been executed had `isFull` been true. Thus, the trace program captures all synchronization events in the observed execution, any writes in that execution that change the value of any annotated predicate associated with a condition variable, as well as any annotated `wait()`'s and `notify()`'s that did not occur in that execution but could have occurred in a different execution. All other operations in the observed execution of the original program are not deemed relevant to finding deadlocks.

There exists an interleaving of the threads in this trace program that corresponds to the interleaving in Figure 6 that exhibits the deadlock. In this interleaving of the trace program, `p` executes its `run()` method till the `wait()` on line 23, where it gets blocked. Then, `r` completely executes its `run()` method and exits. Thereafter, `c` executes its `run()` method, but does not notify `p` because `isFull` is false. Thus, `p` blocks forever waiting to be notified by `c`. Our analysis uses an off-the-shelf model checker to explore all possible interleavings of the trace program and check if any of them deadlocks. In the process of model checking, it encounters this interleaving, and thus finds the deadlock in the original program.

4. ALGORITHM

In this section, we present our deadlock detection algorithm. We first describe the annotations our algorithm requires (Section 4.1). We then formalize the execution of a concurrent system that includes the operations that our al-

gorithm deems relevant to finding deadlocks (Section 4.2), and use that formalization to describe our trace program generation algorithm (Section 4.3). We then discuss how to model check the trace program to report possible deadlocks in the original program (Section 4.4).

4.1 Condition Annotations

```
1 public abstract class ConditionAnnotation{
2   protected static int counter = 0;
3   protected Object o;
4   protected int condId;
5   protected boolean curVal;
6   public ConditionAnnotation(Object o1) {
7     o = o1; condId = counter++;
8     associateWithObject(o1); initCond();
9   }
10  public abstract boolean isConditionTrue
        ();
11  public void waitBegin(Object lock) {
12    int lockId = getUniqueObjId(lock);
13    boolean val = isConditionTrue();
14    addLine("if (c"+condId+") {");
15    if (!val)
16      addLine("synchronized
            (l"+lockId+")"+"{l"+lockId+".wait();}");
17  }
18  public void waitEnd() { addLine("}"); }
19  public void notifyBegin(Object lock) {
20    int lockId = getUniqueObjId(lock);
21    boolean val = isConditionTrue();
22    addLine("if (c"+condId+") {");
23    if (!val)
24      addLine("synchronized (l"+lockId+")"+"{l"+
            lockId+".notify();}");
25  }
26  public void notifyEnd() { addLine("}"); }
27  public void logChange() {
28    boolean newVal = isConditionTrue();
29    if (newVal != curVal) {
30      addLine("c"+condId+"="+newVal+";");
31      curVal = newVal;
32    }
33  }
34  private void associateWithObject(Object
        o) {
35    ... associate this instance of ConditionAnnotation
          with the object o ...
36  }
37  private void initCond() {
38    curVal = isConditionTrue();
39    addLine("c"+condId+"="+curVal+";");
40  }
41 }
```

Figure 8: Definition of class `ConditionAnnotation`.

Our algorithm requires users to annotate the predicate associated with each condition variable in a Java program using class `ConditionAnnotation` (Figure 8). For brevity, we do not show the synchronization required to make `ConditionAnnotation` thread-safe. We describe here how programmers use ConditionAnnotation to annotate their programs; Section 4.3 shows how our algorithm uses these annotations to generate the trace program.

For each predicate associated with a given condition variable, the user subclasses `ConditionAnnotation`, implementing its abstract method `isConditionTrue()`. This method must evaluate to true if and only if the predicate evaluates to true. This predicate will depend on one or more Java objects or static fields. For simplicity, we describe here only the case where the predicate depends on a single object o, but our implementation handles the more general case. The object o is passed to the `ConditionAnnotation` constructor and accessed by the `isConditionTrue()` implementation. Finally, the user calls pre-defined methods `waitBegin()`, `waitEnd()`, `notifyBegin()`, and `notifyEnd()` on the created instance of `ConditionAnnotation` before and after any calls to `wait()`, `notify()`, and `notifyAll()` that are control-dependent on the predicate.

```
1 public class AddLinesToTraceProgram {
2   public Map thrToLines = new TreeMap();
3   public int getUniqueObjId(Object o) {
4     ... return unique integer ID for object o ...
5   }
6   public void addLine(String line) {
7     ... append line to list mapped to current thread in
          thrToLines ...
8   }
9 }
```

Figure 9: Definition of class `AddLinesToTraceProgram` used by class `ConditionAnnotation` and Algorithm 1.

Figure 4 shows the annotations required for the buffer-full predicate. The underlined field `condition` defined on lines 4-9 specifies the actual predicate (line 7, simply a call to method `isFull()` of class `MyBuffer`) and the object on which it depends (line 5, the `MyBuffer` instance). On lines 14, 17, 30 and 36 it specifies the predicate-dependent calls to `wait()` and `notify()`.

4.2 Concurrent System

In this section, we formalize the execution of a concurrent system in terms of the operations that our algorithm deems relevant to finding deadlocks. It is straightforward to express the synchronization logic of a multi-threaded Java program in this system.

A concurrent system consists of a finite number of threads that communicate with each other using shared objects. At any instant, the system is in a state s, in which each thread is at a statement. It transitions from one state to another with the execution of a statement by a thread. The initial state is denoted by s_0. We assume that locks are acquired and released by each thread in a nested manner, that is, if a thread acquires l_1 before l_2 then it releases l_2 before l_1. This is true for Java programs. Our algorithm can be extended to settings with arbitrary locking patterns. We also assume that each thread t is started only once. A statement may be of one of the following forms (we use "current thread" to denote the thread executing the statement):

1. `Acquire(`l`)`: the current thread acquires lock l.
2. `Release()`: the current thread releases the lock it last acquired.
3. `Wait(`l`)`: the current thread is waiting on the condition variable (monitor) l.
4. `Notify(`l`)`: the current thread notifies a thread (if any) waiting on the condition variable l.
5. `NotifyAll(`l`)`: the current thread notifies all threads waiting on the condition variable l.
6. `Start(`t`)`: the current thread starts a fresh thread t, that is, a thread that has not yet been started.
7. `Join(`t`)`: the current thread is waiting for thread t to finish executing.

Algorithm 1 TRACEPROGRAMGENERATOR(s_0)

```
1: s ⇐ s0
2: while Enabled(s) ≠ ∅ do
3:    t ⇐ a random thread in Enabled(s)
4:    stmt ⇐ next statement to be executed by t
5:    s ⇐ Execute(s,t)
6:    if stmt = Acquire(l) then
7:       lId ⇐ getUniqueObjId(l)
8:       addLine( "synchronized (l" + lId + ") {" )
9:    else if stmt = Release() then
10:      addLine( "}" )
11:   else if stmt = Wait(l) then
12:      lId ⇐ getUniqueObjId(l)
13:      addLine( "l" + lId + ".wait();" )
14:   else if stmt = Notify(l) then
15:      lId ⇐ getUniqueObjId(l)
16:      addLine( "l" + lId + ".notify();" )
17:   else if stmt = NotifyAll(l) then
18:      lId ⇐ getUniqueObjId(l)
19:      addLine( "l" + lId + ".notifyAll();" )
20:   else if stmt = Start(t) then
21:      tId ⇐ getUniqueObjId(t)
22:      addLine( "t" + tId + ".start();" )
23:   else if stmt = Join(t) then
24:      tId ⇐ getUniqueObjId(t)
25:      addLine( "t" + tId + ".join();" )
26:   else if stmt = Write(o) || Stmt = Call(o) then
27:      for each ConditionAnnotation c associated with o do
28:         c.logChange()
29:      end for
30:   end if
31: end while
32: if Active(s) ≠ ∅ then print 'System Stall!' endif
33: CreateTraceProgram(AddLinesToTraceProgram.thrToLines)
```

8. `Write(`o`)`: the current thread writes to (some field of) object o.
9. `Call(`o`)`: the current thread invokes a method on object o.

We also use the following definitions in our algorithm:

1. `Enabled(`s`)` denotes the set of all threads that are enabled in state s. A thread is disabled in the following situations: (i) it is waiting to acquire a lock currently held by another thread, (ii) it is waiting to be notified by another thread, or (iii) it is waiting for another thread to finish executing.
2. `Active(`s`)` denotes the set of all threads that have not finished executing in state s.
3. `Execute(`s`,`t`)` denotes the next state resulting from the execution of thread t's next statement in state s.

4.3 Generating the Trace Program

The first stage of CHECKMATE, Algorithm 1 generates a trace program by observing an execution of a program with synchronization predicate annotations. It populates global map `thrToLines` in class `AddLinesToTraceProgram` (Figure 9) while observing the execution. Map `thrToLines` maps each thread in the observed execution to a list of strings. Each string is a statement or a part of a statement, and the whole list is a legal block of statements that constitutes the body of the corresponding thread in the trace program. Whenever a synchronization statement is executed by a thread in the observed execution, the algorithm calls method `addLine()` in class `AddLinesToTraceProgram` to add the string it generates to the list mapped to that thread in `thrToLines`. The generated string depends on the kind of synchronization statement that was executed.

Algorithm 2 `CreateTraceProgram(thrToLines)`

```
1: lockIds ⇐ set of lock identifiers in thrToLines
2: predIds ⇐ set of synchronization predicate identifiers in
   thrToLines
3: thrIds ⇐ set of thread identifiers in thrToLines
4: print "public class TraceProgram {"
5: for all lId such that lId is in lockIds do
6:    print "static Object l" + lId + "= new Object();"
7: end for
8: for all pId such that pId is in predIds do
9:    print "static boolean p" + pId + ";"
10: end for
11: for all tId such that tId is in thrIds do
12:    print " static Thread t" + tId + "= new Thread() {"
13:    print " public void run() {"
14:    for all s such that s is in thrToLines[tId] do
15:       print s
16:    end for
17:    print "} };"
18: end for
19: print " public static void main(String[] args) {"
20: for all tId: tId+".start();" not in any list in thrToLines do
21:    print tId+".start();"
22: end for
23: print " } }"
```

If a lock acquire statement is executed, the algorithm begins a `synchronized` statement for the involved lock object. The lock object is uniquely identified in the trace program using method `getUniqueObjId()` in class `AddLinesToTraceProgram` which provides a unique integer for each object created in the observed execution. If a lock release statement is executed, the algorithm closes the last `synchronized` statement that it had started for the thread. We had earlier stated our assumption that locks are acquired and released in a nested fashion. Thus, a lock release statement always releases the lock that was most recently acquired by the thread.

If a `wait()` statement is executed, the algorithm generates a corresponding `wait()` statement. Similarly, when a `notify()`, `notifyAll()`, `start()`, or `join()` statement is executed, a corresponding statement is generated.

If a statement writing to (some field of) object o is executed, or a method is called on an object o, then the algorithm finds all `ConditionAnnotation` objects that are associated with object o (this association is setup in the `ConditionAnnotation` constructor). After the write or the method call, the state of o may have changed, and hence the predicates associated with those `ConditionAnnotation` objects may have also changed. The trace program needs to track changes to the values of predicates associated with condition variables. For this purpose, the algorithm calls method `logChange()` in class `ConditionAnnotation` to evaluate the predicate and check if its value has indeed changed, and if so, generates a statement writing the new value to the variable associated with the predicate.

When calls to the pre-defined methods `waitBegin()`, `waitEnd()`, `notifyBegin()`, and `notifyEnd()` (Figure 8) that have been added as annotations execute, statements are generated for the trace program that capture the control-dependence of the execution of the `wait()` or `notify()` or `notifyAll()` on the synchronization predicate that has been annotated. These statements are also added using the

method `addLine()` in class `AddLinesToTraceProgram`. We use *if* statements to capture the control dependence; with better annotations or program analysis, we can use *while* statements wherever appropriate to state the control dependence.

After observing the complete execution, the algorithm creates a legal Java program (trace program) by calling method `CreateTraceProgram()` defined in Algorithm 2. It creates an object for each lock object, a boolean variable for each predicate, and a thread object for each thread in the observed execution. For each created thread, it prints a `run()` method containing the list of strings generated for the corresponding thread in map `thrToLines`. Finally, the algorithm prints the `main()` method, and starts all those threads in it which are not started by any other thread[2]. The trace program for the example in Figure 4 is shown in Figure 7.

The algorithm does a couple of optimizations before it prints the body for each thread. Firstly, it does not print any `synchronized` statement that involves a lock that is local to the thread. Thread-local locks cannot be involved in a deadlock, and hence can be safely removed. Secondly, for any sequence of statements that consists only of `synchronized` statements, it does the following optimization. It finds the different nestings of lock acquires within the sequence. Instead of printing all `synchronized` statements present in the sequence, the algorithm prints one block of nested `synchronized` statements for each nesting of lock acquires. This removes a lot of redundancy in `synchronized` statements because of loops in the original program.

4.4 Model Checking the Trace Program

The second stage of our algorithm uses an off-the-shelf model checker to explore all possible thread interleavings of the trace program and check if any of them deadlocks. A deadlock in the trace program may or may not imply a deadlock in the original program. The counterexample provided by the model checker assists in determining whether a deadlock reported by our algorithm is real or false. Multiple counterexamples may denote the same deadlock. We group together counterexamples in which the same set of statements (either lock acquires or calls to `wait()`) is blocked and report each such group as a different possible deadlock. To increase readability, we map the statements in the counterexample back to the corresponding statements in the original (non-trace) program. The deadlock for the example in Figure 4 is shown in Figure 6.

5. EVALUATION

We have implemented our analysis in a prototype tool called CheckMate for Java programs. Given a Java program, we first use the `ConditionAnnotation` class (Figure 8) to manually annotate the predicate associated with each condition variable in the program. CheckMate then uses JChord [20], a program analysis framework for Java, to instrument lock acquires and releases, calls to `wait()`, `notify()`, and `notifyAll()`, calls to thread `start()` and `join()`, and all writes to objects and method calls in the program. It then executes the annotated, instrumented program on given input data and generates the trace program. Finally, it uses the JPF model checker to explore all possible executions of the trace program and report deadlocks.

[2]Normally just the main thread.

5.1 Experimental setup

We applied CheckMate to several Java libraries and applications. We ran all our experiments on a dual socket Intel Xeon 2GHz quad core server with 8GB RAM. The libraries include the Apache log4j logging library (`log4j`), the Apache Commons Pool object pooling library (`pool`), an implementation of the OSGi framework (`felix`), the Apache Lucene text search library (`lucene`), a reliable multicast communication library (`jgroups`), the JDK logging library (`java.util.logging`), the Apache Commons DBCP database connection pooling library (`dbcp`), and the JDK swing library (`javax.swing`). We used two different versions of `jgroups`. We wrote test harnesses exercising each library's API, including two different harnesses for each of `pool` and `lucene`, and a single harness for each of the remaining libraries.

The applications include Groovy, a Java implementation of a dynamic language that targets Java bytecode (`groovy`), JRuby, a Java implementation of the Ruby programming language (`jruby`), and a Java web server from W3C (`jigsaw`). For `jigsaw`, we wrote a harness to concurrently send multiple requests and administrative commands like "shutdown server" to the web server to simulate a concurrent environment.

5.2 Results

Table 1 summarizes our experimental results. The second column reports the number of `ConditionAnnotation`'s we had to provide, each annotating a different synchronization predicate in the benchmark. We report the number of `ConditionAnnotation`'s that we had to define, and not the total number of lines of code that we had to use to define the `ConditionAnnotation`'s and to invoke methods on those `ConditionAnnotation`'s. The numbers in this column show that the annotation burden of our approach is very small.

The third column shows the number of lines of Java code in methods that were executed in the original program. The fourth column shows the number of lines of Java code in the trace program. Notice that the trace programs are much smaller than (executed parts of) original programs although the trace program unrolls all loops and inlines all methods executed in the original program.

The fifth column gives the average runtime of the original program without any instrumentation. We do not report the runtime for the `jigsaw` webserver because of its interactive nature. The sixth column gives the average runtime of the original program with annotations and instrumentation; it includes the time to generate the trace program. Comparing these two columns shows that the runtime overhead of CheckMate is acceptable.

The seventh column gives the average runtime of JPF on the original program. We could not run JPF on eight of these programs because it does not support some JDK libraries, and has limited support for reflection. For the remaining six programs, JPF did not terminate within 1 hour nor did it report any error traces.

The eighth column shows the average runtime of JPF on the trace programs. It terminates within a few seconds on eleven of these programs. It does not terminate within 1 hour for `jgroups-2.5.1` and `jigsaw-2.2.6`, but it reports a number of error traces in that time. These benchmarks have a lot of threads (31 for `jgroups-2.5.1` and 12 for `jigsaw-2.2.6`), hence a huge number of thread interleavings, which

Program name	No. of cond annots	Orig prog LOC	Trace prog LOC	Orig prog time	Time to gen prog	JPF time on orig prog	JPF time on trace prog	No. of error traces	Potential errors	Confirmed errors	Known errors
groovy-1.1-B1	1	45,796	59	0.118s	1s	> 1h	1.3s	5	1/0	1/0	1/0
log4j-1.2.13	2	48,023	225	0.116s	1s	-	8.7s	167	2/0	1/0	1/0
pool-1.5 (harness 1)	4	48,024	136	0.116s	1s	> 1h	2.3s	41	1/0	1/0	1/0
pool-1.5 (harness 2)	4	48,024	191	0.123s	1s	> 1h	2.6s	36	1/0	1/0	1/0
felix-1.0.0	4	73,512	113	0.173s	2.8s	-	-	-	-	1/0	1/0
lucene-2.3.0 (harness 1)	9	68,311	298	0.230s	3s	> 1h	1s	0	0/0	0/0	1/0
lucene-2.3.0 (harness 2)	9	81,071	3,534	0.296s	3.6s	> 1h	20s	0	0/0	0/0	1/0
jgroups-2.6.1	12	92,934	118	0.228s	4s	-	3.4s	39	2/0	1/0	1/0
jigsaw-2.2.6	17	122,806	3,509	-	-	-	> 1h	7894	2/7	1/5	0/2
jruby-1.0.0RC3	16	136,479	966	1.1s	13.7s	-	3.9s	58	1/0	1/0	1/0
jgroups-2.5.1	15	160,644	2,545	9.89s	21s	-	> 1h	124	1/0	0/0	1/0
java logging (jdk-1.5.0)	0	43,795	131	0.177s	2s	> 1h	3.7s	96	0/2	0/1	0/1
dbcp-1.2.1	0	90,821	400	0.74s	3.3s	-	12s	320	0/2	0/2	0/2
java swing (jdk-1.5.0)	0	264,528	1,155	0.96s	17.6s	-	105s	685	3/1	0/1	0/1

Table 1: Experimental results

makes model checking slow. JPF crashes on the trace program for `felix-1.0.0`. Comparing the runtime of JPF on the original and trace programs shows that it is much more feasible to model check the trace programs.

The ninth column shows the number of error traces produced by JPF for the trace programs. An error trace is an interleaving of threads that leads to a deadlock. Not each error trace leads to a different deadlock, and thus, the number of error traces is not an indication of the number of different deadlocks in the program. Hence, CheckMate groups together error traces in which the same set of statements (either lock acquires or calls to `wait()`) is blocked, and reports each such group as a potential deadlock. The tenth column shows the number of these potential deadlocks reported by CheckMate. The eleventh column shows how many of these deadlocks we could manually confirm as real, and the final column shows the number of deadlocks that were previously known to us. The first number in each entry in the last three columns is the number of communication deadlocks including the deadlocks that involve both locks and condition variables. The second number is the number of resource deadlocks. In most of the benchmarks, we were able to find all previously known deadlocks. Since `JPF` crashed on the trace program for `felix-1.0.0`, we applied a randomized model checker (i.e. a model checker that tries out random thread schedules) to it. The randomized model checker reported a deadlock that was the same as it's previously known communication deadlock.

5.3 Deadlocks found

We found a number of previously known and unknown deadlocks in our experiments. We discuss some of them in detail below. Our running example in Figure 4 documents the previously known communication deadlock we found in `log4j` that is reported at `https://issues.apache.org/bugzilla/show_bug.cgi?id=38137`.

Figure 10 shows a previously known deadlock in `groovy` reported at `http://jira.codehaus.org/browse/GROOVY-1890`. It shows relevant code from `MemoryAwareConcurrentReadMap.java`. This deadlock is a hybrid between a communication and resource deadlock. Thread `T2` increments field `concurrentReads` of a `MemoryAwareConcurrentReadMap` object. Thread `T1` checks predicate `concurrentReads != 0`. Since this predicate is true, it executes the `wait()` on line 307. `T1` executes the `wait()` on `writeQueue`, but it also holds a lock on `writeLock`. Thread `T2` is the only thread that can wake it up, but before `T2` can reach the `notify()` on line 336, it needs to acquire the lock on `writeLock` to decrement the value of `concurrentReads`. Since the lock on `writeLock` is held by `T1`, it gets blocked. Thus, `T1` is waiting to be notified by `T2`, and `T2` is waiting for `T1` to release `writeLock`.

```
            T2
326:synch (writeLock) {
327:  concurrentReads++;
328:}
            T1
126:synch (writeLock) {
304:  synch (writeQueue) {
305:    while (concurrentReads != 0) {
307:      writeQueue.wait();
309:    }
310:  }
129:}
            T2
332:synch (writeLock) {
333:  concurrentReads--;
334:}
335:synch (writeQueue) {
336:  writeQueue.notify();
337:}
```

Figure 10: Deadlock in `groovy`.

We found a previously unknown communication deadlock in `jigsaw`. Figure 11 explains the deadlock. The line numbers in the figure are of statements in `ResourceStoreManager.java` in the benchmark. Thread `T1` is a `StoreManagerSweeper` thread that executes the `wait()` on line 406 after it has been started. But, before it can execute this `wait()`, the server receives a request to shut down. Thread `T2`, which is a `httpd` server thread, tries to shut down the `StoreMan-`

```
        T1                          T2
                  417:synch void shutdown() {
                  419:  notifyAll();
                  420:}
401:boolean done = false;
404:while(!done) {
406:  wait();
407:  done = true;
410:}
```

Figure 11: Deadlock in `jigsaw`.

`agerSweeper` thread, and invokes `notifyAll()` at line 419 during the process of shutting down. This `notifyAll()` is the notification that is meant to wake `T1` up when it waits at the `wait()` on line 406. Thus, when `T1` actually executes the `wait()`, it just gets hung there. It has already missed the notification that was supposed to wake it up.

Two resource deadlocks in `jigsaw` were known previously [11]. We not only found those two deadlocks, but we also found three other resource deadlocks in `jigsaw`. They are similar to the two previously known deadlocks in that they also involve locks on `SocketClientFactory` and `SocketClientState` objects, but they differ in the source line numbers where the locks are acquired.

6. OTHER RELATED WORK

There is little prior work on detecting communication deadlocks. Agarwal and Stoller [1] dynamically predict missed notification deadlocks, in particular they define a happens-before ordering between synchronization events, and use it to reason if a wait that was woken up by a notify could have happened after that notify. Farchi et al [6] describe several concurrency bug patterns that occur in practice including missed notification. They also describe a heuristic that can increase the probability of manifesting a missed notification during testing. Hovemeyer and Pugh [10] present several common deadlock patterns in Java programs that are checked by their static tool FindBugs, including many involving condition variables such as unconditional wait, wait with more than one lock held, etc. Their patterns cannot help to detect the deadlock in Figure 4 and missed notifications in general. Von Praun [18] also statically detects waits that may execute with more than one lock held, and waits that may be invoked on locks on which there is no invocation of a notify. His approach too cannot detect the deadlock in Figure 4 and missed notifications. Li et al [12] build a deadlock monitor that runs as a system daemon, and detects deadlocks that actually occur during the execution of systems with multiple processes or threads. The monitor can detect deadlocks involving semaphores and pipes in addition to locks. In CheckMate, we observe a deadlock-free program execution and predict deadlocks that could occur in a different execution of the program.

7. CONCLUSION AND FUTURE WORK

We have presented a novel dynamic analysis called CheckMate that predicts a broad class of deadlocks.

Like most predictive dynamic analyses [16, 8, 13], CheckMate is neither complete nor sound. Since it does not track all control and data dependencies observed during execution, it can miss deadlocks as well as report false deadlocks. One way to rectify that would be to use dynamic slicing to track not just the variables in synchronization predicates, but also other variables that can affect the values of variables in synchronization predicates.

Acknowledgments

We would like to thank our anonymous reviewers for their valuable comments. This work was supported in part by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227), and by NSF Grants CNS-0720906 and CCF-0747390.

8. REFERENCES

[1] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD*, 2006.

[2] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with static analysis and runtime monitoring. In *PADTAD*, 2005.

[3] S. Bensalem and K. Havelund. Scalable dynamic deadlock analysis of multi-threaded programs. In *PADTAD*, 2005.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[5] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP*, pages 237–252, 2003.

[6] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS*, 2003.

[7] J. Harrow. Runtime checking of multithreaded applications with visual threads. In *SPIN*, 2000.

[8] K. Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN*, 2000.

[9] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *Intl. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[10] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java. In *CSJP*, 2004.

[11] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, 2009.

[12] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Annual Technical Conference*, 2005.

[13] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, 2006.

[14] S. Masticola and B. Ryder. A model of Ada programs for static deadlock detection in polynomial time. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 97–107, 1991.

[15] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, 2009.

[16] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[17] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *ASE*, 2003.

[18] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[19] A. Williams, W. Thies, and M. Ernst. Static deadlock detection for Java libraries. In *ECOOP*, 2005.

[20] `http://code.google.com/p/jchord/`