Static Deadlock Detection for Java Libraries

Amy Williams, William Thies, and Michael D. Ernst

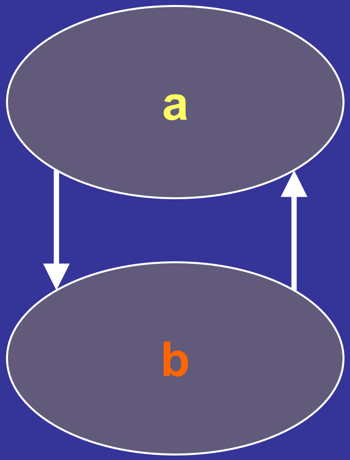
Massachusetts Institute of Technology

Deadlock

- Each deadlocked thread attempts to acquire a lock held by another thread
  - Can halt entire program execution
  - Difficult to expose during testing
  - Once exposed, difficult to replicate
- Example:

StringBuffer a, b;

Thread 1:	Thread 2:
a.append(b);	b.append(a);
locks a, b	*locks b, a*



Deadlock in Libraries

- Library writers may wish to provide guarantees
  - JDK’s StringBuffer documentation says class is thread-safe
- Goal: find client calls that deadlock library or verify that none exist

Analyzing Programs / Libraries

	For Programs:	**For Libraries:**
Method Calls	Fixed	Consider all calling patterns
Aliasing Possibilities	Fixed	Consider aliasing induced by any program
Number of Threads	Might be known	Unbounded

Deadlock from Sun’s JDK

```
import java.beans.beancontext.*;

BeanContextSupport support = new BeanContextSupport();
Object source = new Object();
PropertyChangeEvent event =
    new PropertyChangeEvent(source, "beanContext", ...);
support.add(source);
support.vetoableChange(event);
```

Thread 1:	**Thread 2:**
support.propertyChange(event);	support.remove(source);
locks global, field	*locks field, global*

Also found 13 other deadlocks

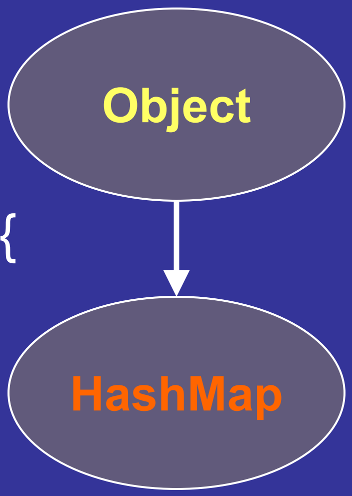
Analysis Overview

1. Build lock-order graph representing locking behavior of each method in library
2. Combine graphs for all public methods into single graph
3. Detect cycles in this graph, which indicate deadlock possibilities

- Analysis properties: reports all deadlocks, context-sensitive, flow-sensitive

JDK Source (simplified)

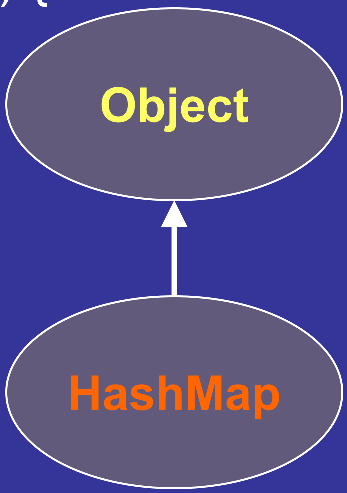
```
interface BeanContext {
    public static final Object globalHierarchyLock;
}
class BeanContextSupport {
    protected HashMap children;

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



Continued...

JDK Source (simplified), cont.

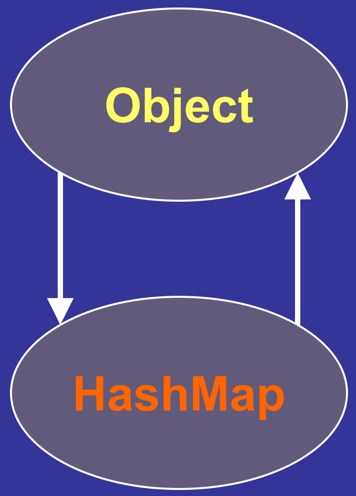
```
class BeanContextSupport {
    protected HashMap children;
    public void propertyChange(PropertyChangeEvent pce) {
      ...
      Object source = pce.getSource();
      synchronized(children) {
        if (...) {
          ...
          remove(source);
          ...
        }
      }
    }
}
```




```
public boolean remove(Object targetChild) {
  synchronized (BeanContext.globalHierarchyLock) {
    ...
  }
}
```

Merged Graph

- When merged, graphs indicate possible locking orders of all methods
- Cycles indicate possible deadlock
  - Expose cases in which threads lock set of locks in different (conflicting) orders



Outline

- Introduction
- Deadlock Detection Algorithm
- Results
- Related Work and Conclusions

Synchronization in Java

- Locking is hierarchical, performed using synchronized statement
  - Multiple locks acquired via nested synchronized statements

```
synchronized (lock1) {
  synchronized (lock2) {
    ...
  }
}
```

- Synchronizing on previously acquired lock always succeeds
  - Considered a no-op for our analysis
- Synchronized methods sugar for synchronizing on this

Synchronization in Java

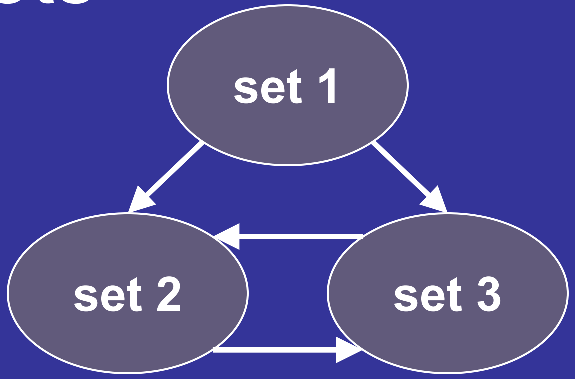
- wait() and notify() methods described in paper
- Java 1.5's non-hierarchical primitives (in java.concurrent package) not covered by analysis
  - Usage rare; recommended only for expert programmers

Analysis Overview

1. Build lock-order graph representing locking behavior of each method in library
   - Callee graphs integrated into caller
   - Iterate to fixed point; termination guaranteed
2. Combine graphs for all public methods into single graph
3. Detect cycles in this graph, which indicate deadlock possibilities

Lock-order Graph

- Directed graph that represents the order in which locks are acquired
- Nodes represent may-alias sets
  - Allows graphs from different methods to be combined



- Edges mean the source lock held while destination lock acquired
- Cycles indicate possibility of deadlock

Example Library

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:

Ordered list of locks held:

[]

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:

Ordered list of locks held:

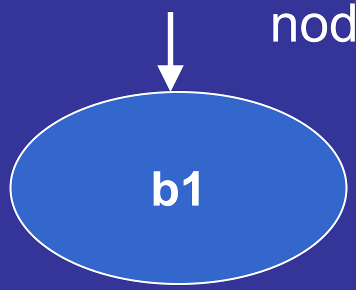
[]

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:

No locks held, so node is root



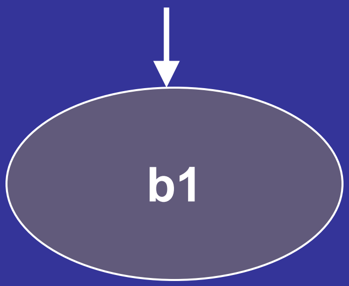
Ordered list of locks held:

[**b1**]

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



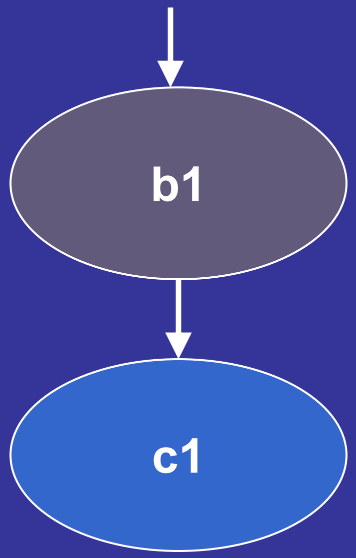
Ordered list of locks held:

[b1]

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



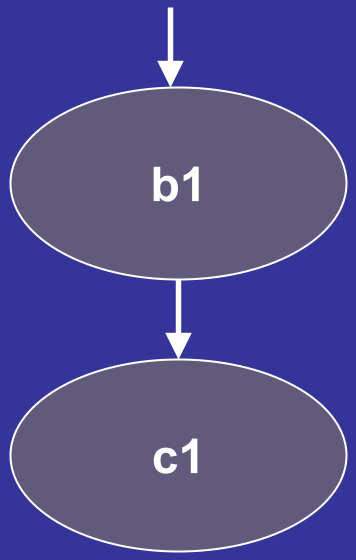
Ordered list of locks held:

[b1, **c1**]

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



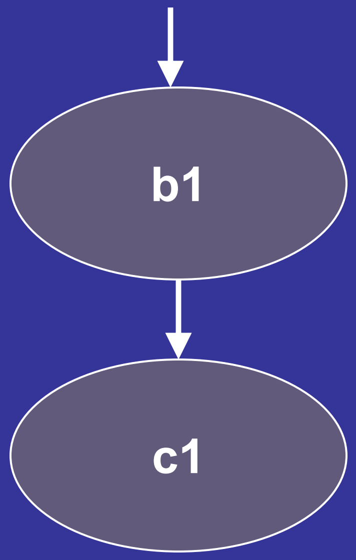
Ordered list of locks held:

[b1, c1]

Example Analysis: deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



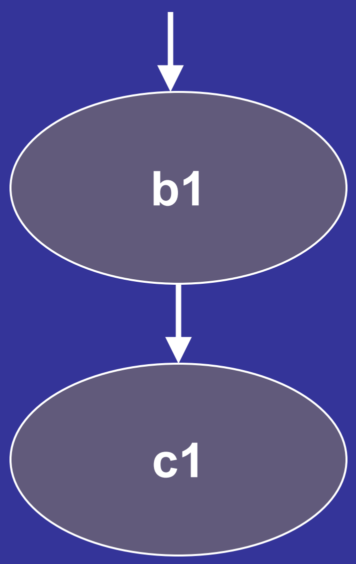
Ordered list of locks held:

[b1]

Lock-order graph for deposit()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:

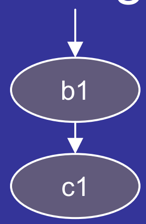


Example Analysis: openAccount()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:

deposit's graph:



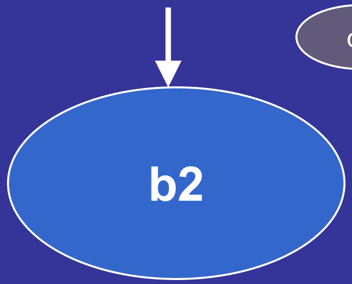
Ordered list of locks held:

[]

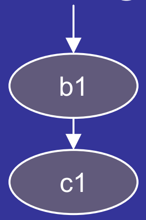
Example Analysis: openAccount()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



deposit's graph:



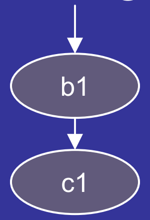
Ordered list of locks held:

[**b2**]

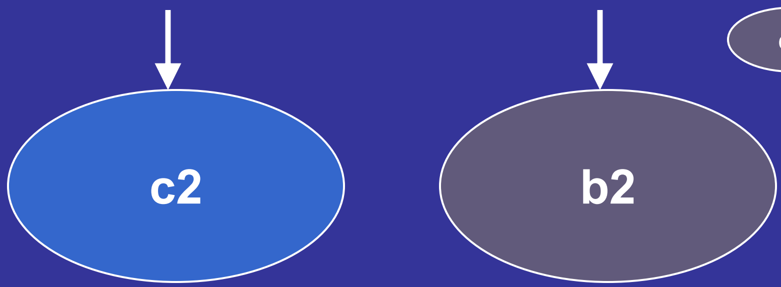
Example Analysis: openAccount()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

deposit's graph:



Graph:



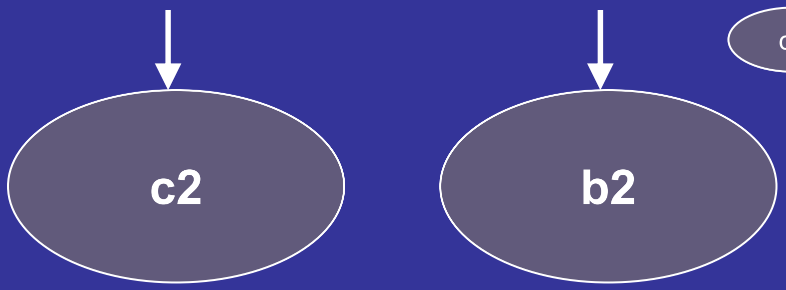
Ordered list of locks held:

[c2]

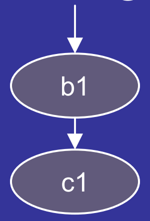
Example Analysis: openAccount()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



deposit’s graph:



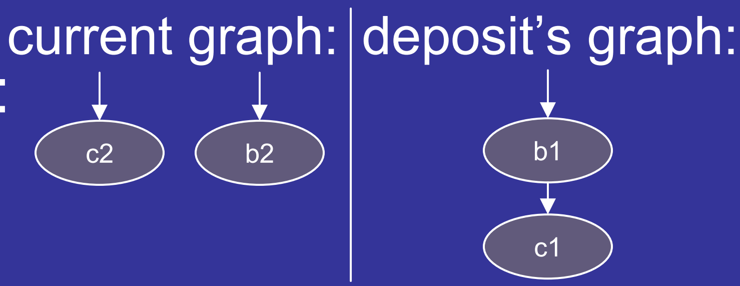
Ordered list of locks held:

[c2]

Example Analysis: openAccount()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:

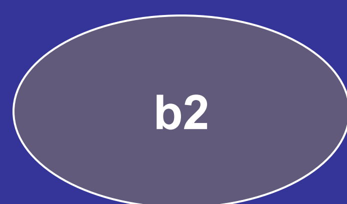


Ordered list of locks held:

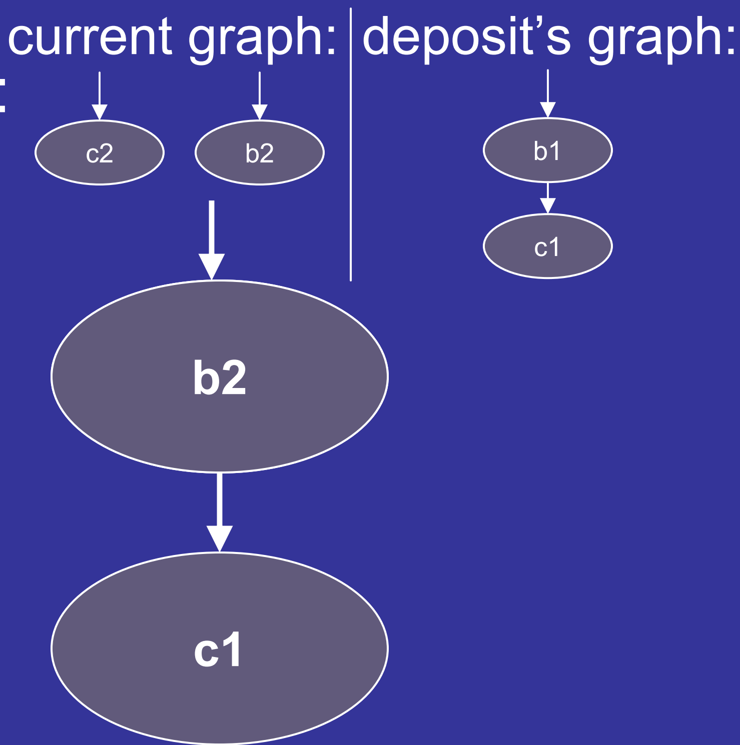
[c2]

Call to deposit(): update copy of deposit's graph

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```



Graph:

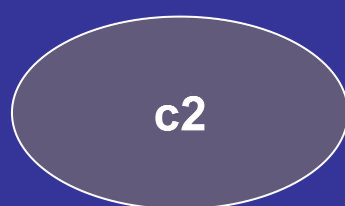


Ordered list of locks held:

[c2]

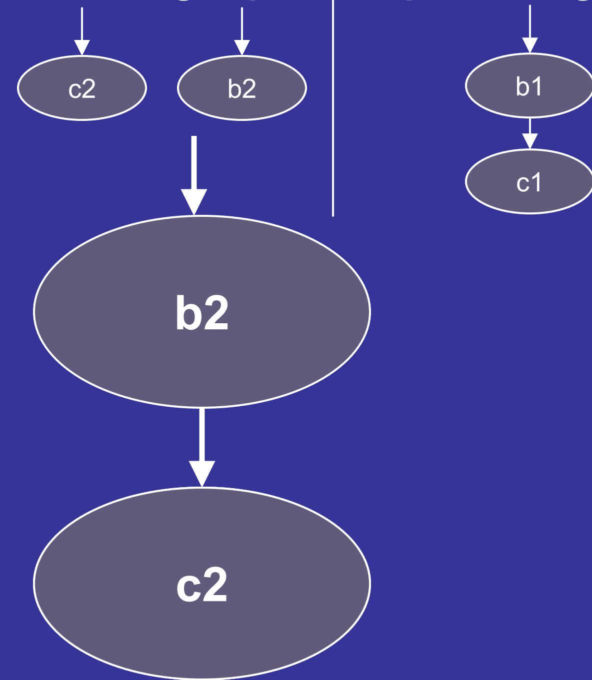
Call to deposit(): update copy of deposit's graph

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```



Graph:

current graph: | deposit's graph:



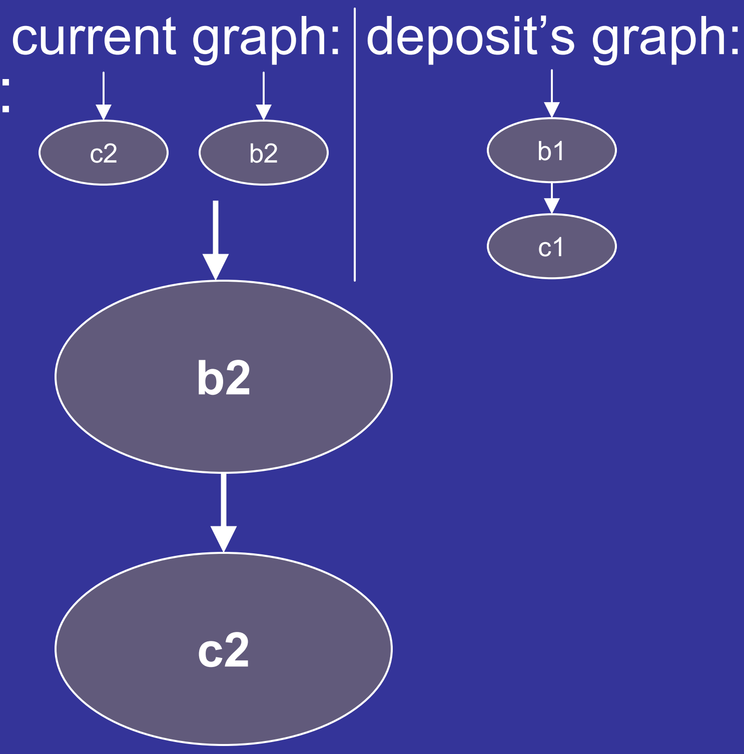
Ordered list of locks held:

[c2]

Call to deposit(): update copy of deposit's graph

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



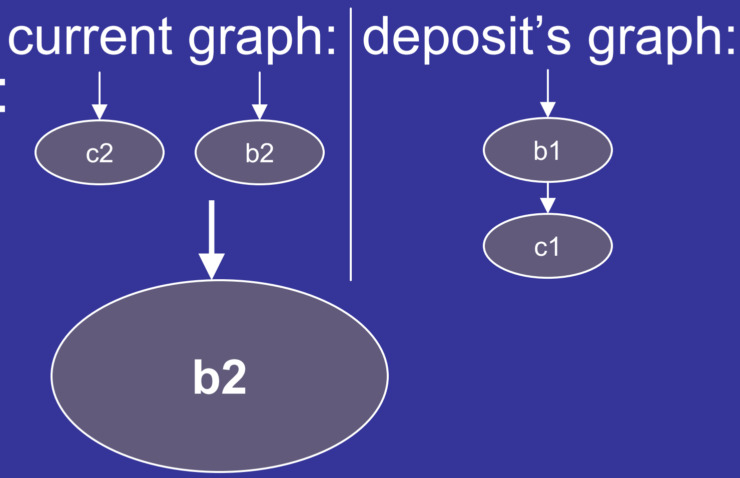
Ordered list of locks held:

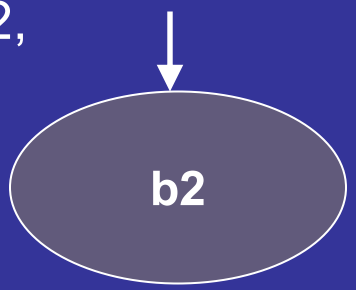
[c2]

Call to deposit(): update copy of deposit's graph

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:





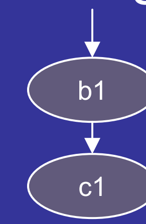
Ordered list of locks held:

[c2]

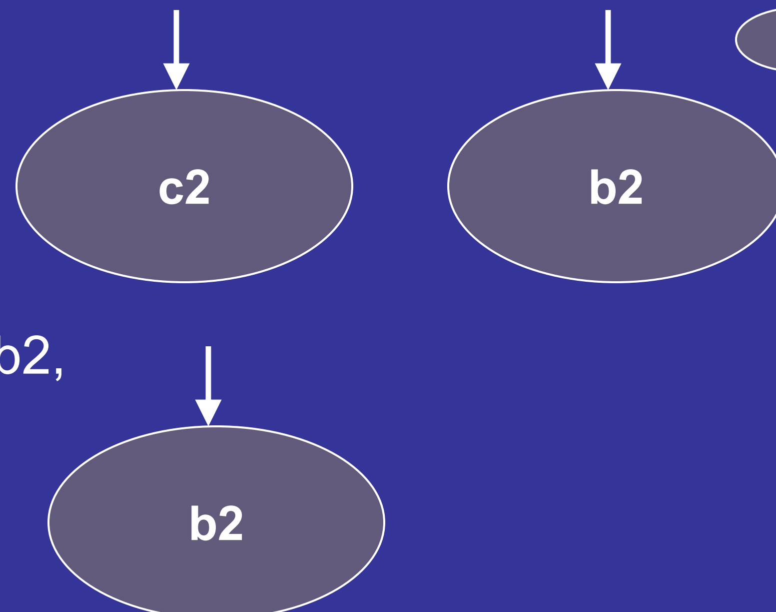
Call to deposit(): insert deposit's graph

```
public void deposit(Bank b1,
                   Client c1) {
   synchronized (b1) {
      synchronized (c1) {
         ...
      }
   }
}
public void openAccount(Bank b2,
                        Client c2) {
   synchronized (b2) {
      ...
   }
   synchronized (c2) {
      deposit(b2, c2);
   }
}
```

deposit's graph:



Graph:



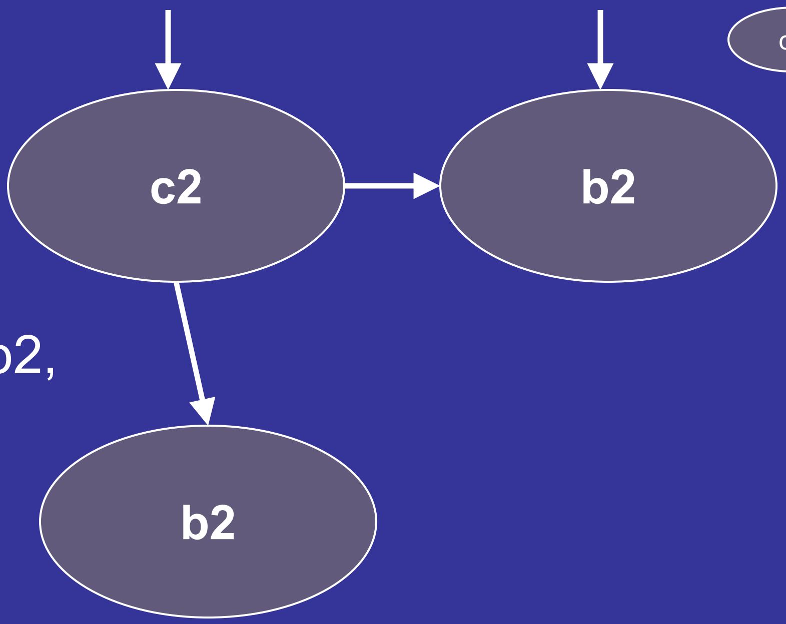
Ordered list of locks held:

[c2]

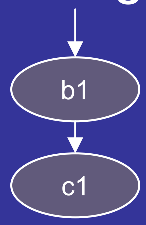
Call to deposit(): insert deposit's graph

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



deposit's graph:



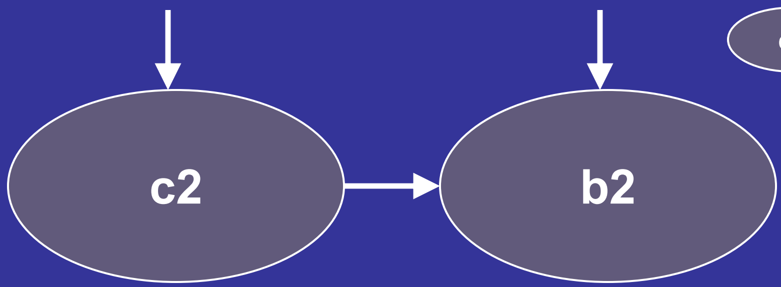
Ordered list of locks held:

[c2]

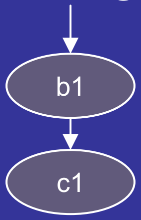
Lock-order graph for openAccount()

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Graph:



deposit's graph:

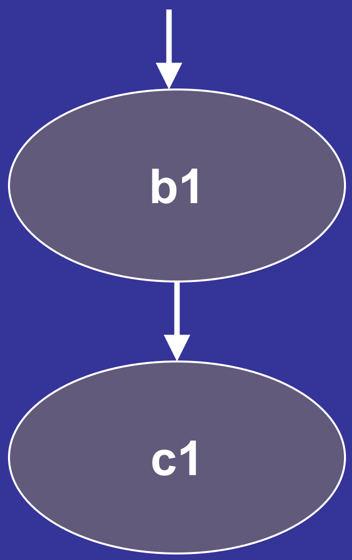


Analysis Overview

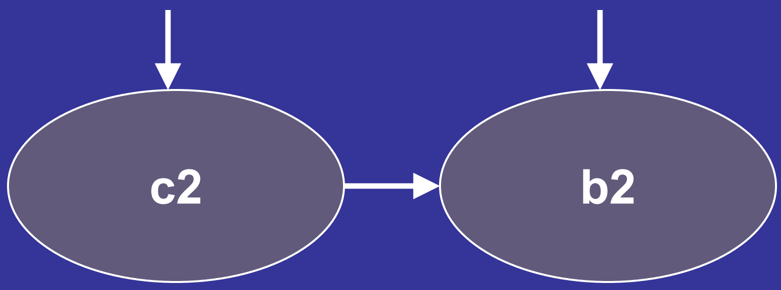
1. Build lock-order graph representing locking behavior of each method in library
   - Callee graphs integrated into caller
   - Iterate to fixed point; termination guaranteed
2. Combine graphs for all public methods into single graph
3. Detect cycles in this graph, which indicate deadlock possibilities

Combine Graphs

Graph for deposit():

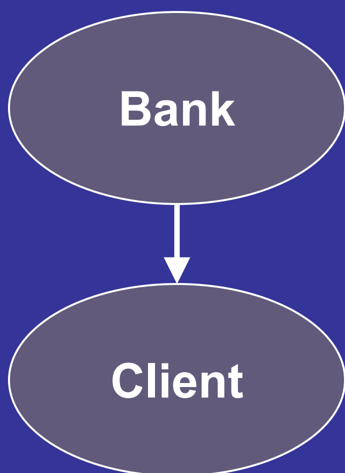


Graph for openAccount():

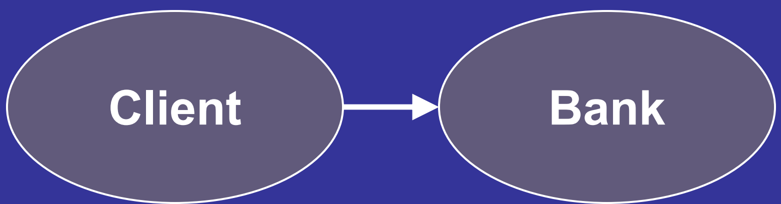


Combine Graphs

Graph for deposit():

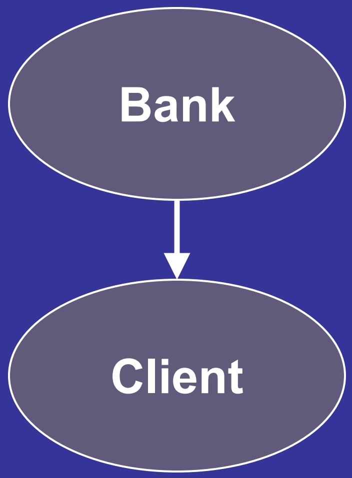


Graph for openAccount():



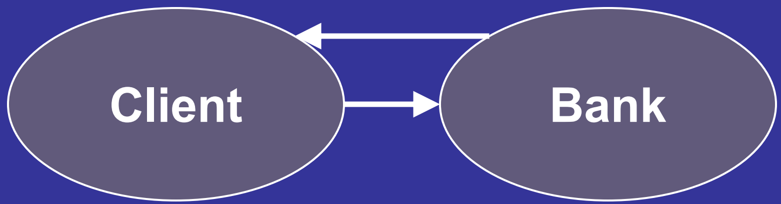
Combine Graphs

Graph for deposit():



Graph for openAccount():

Final graph:



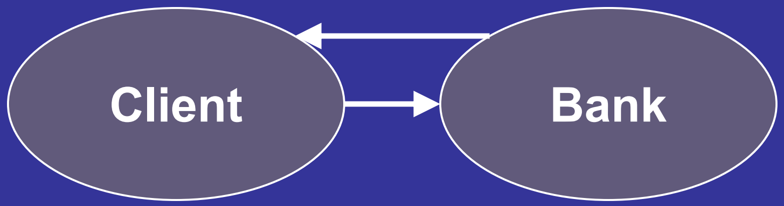
Analysis Overview

1. Build lock-order graph representing locking behavior of each method in library
    - Callee graphs integrated into caller
    - Iterate to fixed point; termination guaranteed
2. Combine graphs for all public methods into single graph
3. Detect cycles in this graph, which indicate deadlock possibilities

Cycle in Combined Graph

Cycles indicate possibility of deadlock, and deadlock is possible

Final graph:



Code that Deadlocks Library

```
public void deposit(Bank b1,
                    Client c1) {
  synchronized (b1) {
    synchronized (c1) {
      ...
    }
  }
}
public void openAccount(Bank b2,
                        Client c2) {
  synchronized (b2) {
    ...
  }
  synchronized (c2) {
    deposit(b2, c2);
  }
}
```

Bank b; Client c;

Thread 1:	**Thread 2:**
openAccount(b, c);	deposit(b, c);
locks c, b	*locks b, c*

Improving Precision

- We further refine may-alias sets and type information in certain cases (see paper)
  - Unaliased fields
  - Caller / callee type resolution
  - Final and effectively-final fields
- These optimizations prove very effective: one library went from 909 reports to only 1
- Context-sensitivity (integrating callee graphs) greatly improved precision

Outline

- Introduction
- Deadlock Detection Algorithm
- Results
- Related Work and Conclusions

Deadlocks Detected

- Analysis is sound: detects all deadlocks in library under analysis
- Assumptions:
  - Clients assumed to respect lock order of library for any shared locks
  - Callbacks are not modeled
    - The client code may call any public method
    - Would introduce many locking orders which are unlikely in practice
  - Reflection not handled

Deadlock Reports

- Each report: set of variables possibly involved in deadlock
- Also provided: set of methods possibly deadlocking using those variables
  - Sometimes many call sequences per report

Results: Overview

- Analyzed 18 libraries
- 13 libraries verified to be deadlock-free
  - Each library analyzed in under 3 minutes
- 5 libraries not verified
  - Exhibited 14 distinct deadlocks
  - Each library analyzed in under 3 minutes employing filtering heuristics

Deadlock-Free Libraries

Library	`sync`	kLOC	Reports
jcurzez	24	4	1
httpunit	17	23	0
jasperreports	11	67	0
croftsoft	11	14	2
dom4j	6	41	1
cewolf	6	7	0
jfreechart	5	125	0
htmlparser	5	22	0
jpcap	4	8	0
treemap	4	7	0
PDFBox	2	28	0
UJAC	1	63	0
JOscarLib	1	6	0

Deadlock-Free Libraries

Library	`sync`	kLOC	Reports
jcurzez	24	4	1
httpunit	17	23	0
jasperreports	11	67	0
croftsoft	11	14	2
dom4j	6	41	1
cewolf	6	7	0
jfreechart	5	125	0
htmlparser	5	22	0
jpcap	4	8	0
treemap	4	7	0
PDFBox	2	28	0
UJAC	1	63	0
JOscarLib	1	6	0

Deadlock-Free Libraries

Library	`sync`	kLOC	Reports
jcurzez	24	4	1
httpunit	17	23	0
jasperreports	11	67	0
croftsoft	11	14	2
dom4j	6	41	1
cewolf	6	7	0
jfreechart	5	125	0
htmlp			
jpcap			
treemap	4	7	0
PDFBox	2	28	0
UJAC	1	63	0
JOscarLib	1	6	0

Manually verified 4 reports to be false positives

Non-verified Libraries

Library	**`sync`**	**kLOC**	**Reports**	**Deadlocks Found**
JDK	1458	419	Out of Memory	7
Classpath	754	295	Out of Memory	5
ProActive	199	63	≥ 196	2
Jess	111	27	≥ 269	0
sdsu	69	26	≥ 20,479	0

Deadlocked JVM for all 14 cases

Filtering Heuristics

- Full analysis can yield too many reports
- Cycle length
  - Do not report cycles longer than 2 nodes
- Assume runtime type same as declared type
  - Lock declared as Object cannot alias with subclasses
- May filter out real deadlocks

Non-verified Libraries

Library	**`sync`**	**kLOC**	**Reports**	**Reports (Filtered)**	**Deadlocks Found**
JDK	1458	419	Out of Memory	70	7
Classpath	754	295	Out of Memory	32	5
ProActive	199	63	≥ 196	3	2
Jess	111	27	≥ 269	23	0
sdsu	69	26	≥ 20,479	3	0

Deadlocked JVM for all 14 cases

Non-verified Libraries

Library	`sync`	kLOC	Reports	Reports (Filtered)	Deadlocks Found
JDK	1458	419	Out of Memory	70	7
Classpath	754	295	Out of Memory	32	5
ProActive	199	63	≥ 196	3	2
Jess	111	27	≥ 269	23	0
sdsu	69	26	≥ 20,479	3	0

Deadlocked JVM for all 14 cases

Deadlocks Found

	JDK	Classpath
BeanContextSupport	×	
StringBuffer	×	×
synchronized Collections	×	×
PrintWriter/CharArrayWriter	×	
java.awt.dnd.DropTarget	×	
java.awt.EventQueue	×	×
java.awt.Menu	×	
java.util.SimpleTimeZone		×
java.util.logging.Logger		×

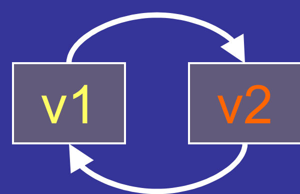
ProActive: ProxyForGroup, AbstractDataObject

ProActive's ProxyForGroup

- ProxyForGroup method asynchronousCallOnGroup() can be made to lock both this and any other ProxyForGroup object
  - Complicated state required to produce this scenario

Cyclic Deadlocks

- java.util.Vector can be deadlocked by forming a cycle with two Vector instances




```
Vector v1, v2; Object o;
v1.add(v2);
v2.add(v1);
```

Thread 1:	**Thread 2:**
v1.contains(o);	v2.contains(o);
locks v1, v2	*locks v2, v1*

- Similar deadlock in
  - All other synchronized Collections
  - Combinations of those Collections
- This deadlock only counted once for JDK and Classpath
  - 5 other deadlocks

Outline

- Introduction
- Deadlock Detection Algorithm
- Results
- Related Work and Conclusions

Related Work

- Using lock-order graphs:
  - Jlint [Artho, Biere 2001]; von Praun 2004
  - For programs, do not detect all deadlocks
- RacerX [Engler, Ashcraft 2003]
  - Non-hierarchical locking (for C), requires annotations, does not detect all deadlocks
- Model Checking:
  - Demartini, Iosif, Sisto 1999
  - Java Pathfinder: Havelund, Pressburger 2000
  - For programs, not scalable
- Ownership Types:
  - Boyapati, Lee, Rinard 2002
  - Requires annotations, restricts programming model

Conclusions

- Our analysis is effective at
  - Verifying libraries to be free from deadlock
  - Finding deadlocks
- Analysis of libraries can be effective at finding library specific defects

Sources of Imprecision

- Consider infeasible aliasing / sharing across threads
  - Do not track flow of values through fields
- Consider infeasible paths of control

Resolving Deadlocks

- Two possible solutions:
  - Rewrite methods to acquire locks in set order
  - Extend Java with synchronization primitive to atomically acquire multiple locks (can also write this as a library method)
- Issue: must know locks
  - Can sometimes write helper methods to determine locks
  - Locks may change while being determined
    - Global lock or transactions are alternatives