Symbolic Deadlock Analysis for Concurrent Libraries and their Clients

Jyotirmoy V. Deshmukh[1] E. Allen Emerson[1] Sriram Sankaranarayanan[2]

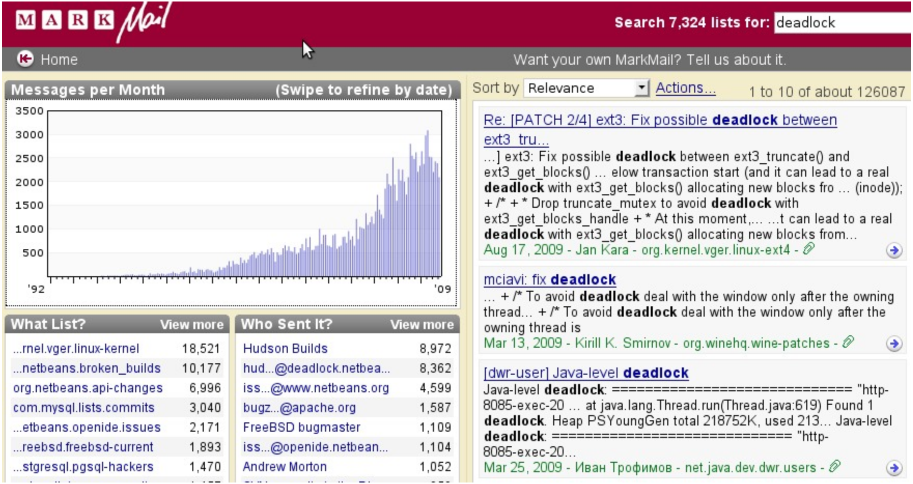
{deshmukh,emerson}@cs.utexas.edu, srirams@colorado.edu

[1]University of Texas at Austin

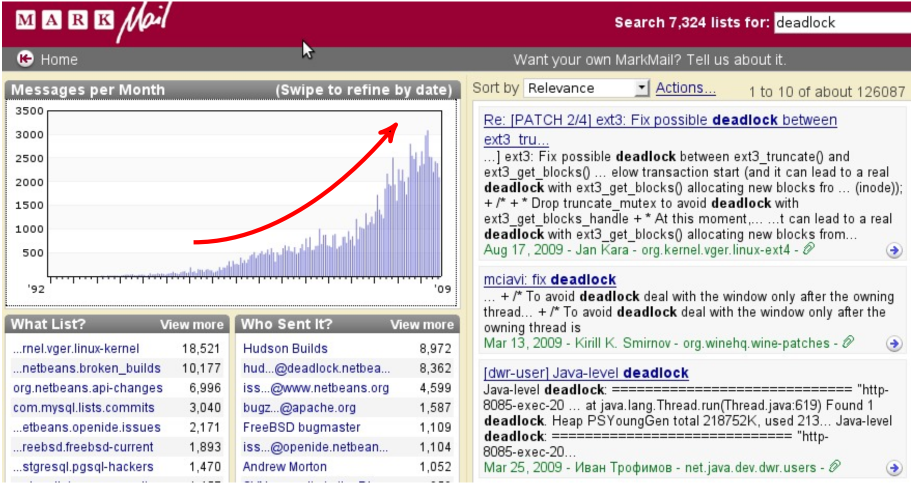
[2]University of Colorado at Boulder

Automated Software Engineering 2009

Thread Safety



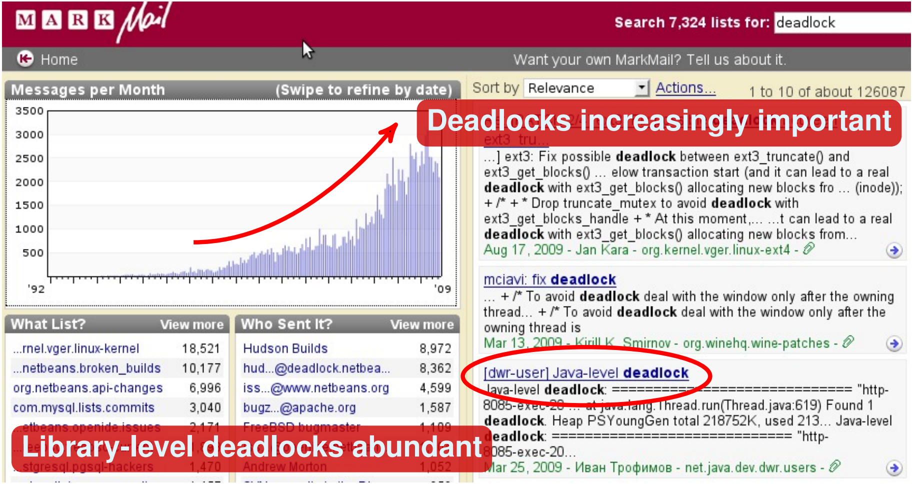
Thread Safety



Thread Safety

Thread Safety

Thread Safety



Concurrent Software is Modular

- Concurrent Library: methods concurrently invokable.
- Multi-threaded Client: each thread invokes library methods.
- “Whole-program approach” too expensive.

Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP '05]

Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP '05]
- Aliasing information for improved accuracy.

Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP '05]
- Aliasing information for improved accuracy.
- Interface Contracts on methods to ensure deadlock-freedom.

Deadlockability Analysis: Goals

- Predict concurrent method invocations potentially leading to deadlock. [Williams et. al, ECOOP '05]
- Aliasing information for improved accuracy.
- Interface Contracts on methods to ensure deadlock-freedom.
- Use interface contracts when analyzing Client code.

Outline

Outline

1. Deadlockability Analysis
2. Problem Size Reduction
3. Symbolic Computation
4. Results

java.awt.EventQueue

```
EventQueue nextQueue;
:
void postEventPrivate (Event e) {
    ...
    synchronized (this) {
        ...
        nextQueue.postEventPrivate(e);
        ...
    }
    ...
}
```

||

```
void wakeup(boolean f) {
    ...
    synchronized (this) {
        ...
        nextQueue.wakeup(f);
        ...
    }
    ...
}
```

Lock-Order Graphs

```
void postEventPrivate (Event e) {
  ...
  synchronized (this) {
    nextQueue.postEventPrivate(e);
    ...
  }
  ...
}
```

lg(postEventPrivate)

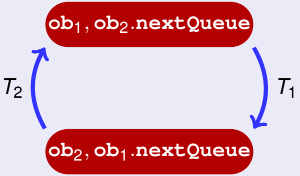
```
void wakeup (boolean f) {
  ...
  synchronized (this) {
    nextQueue.wakeup(f);
    ...
  }
  ...
}
```

lg(wakeup)

Aliasing Pattern leading to Deadlock?

ob1 = ob2.nextQueue, ob2 = ob1.nextQueue

```
void postEventPrivate (Event e) {
    . . .
    synchronized (this) {
        nextQueue.postEventPrivate(e);
        . . .
    }
    . . .
}
void wakeup (boolean f) {
    . . .
    synchronized (this) {
        nextQueue.wakeup(f);
        . . .
    }
    . . .
}
```



lg(wakeup) $\bigcup$ lg(postEventPrivate)

Such weird aliasing comes from...

```
EventQueue nextQueue;
void push (EventQueue eq) {
  ...
  nextQueue = eq;
  ...
}
```

Sequence of method calls

`eq1.push(eq2);`	`eq2.push(eq1);`
⋮	⋮
`eq1.wakeup(...);`	`eq2.postEventPrivate(...);`
⋮	⋮

Deadlock-causing Aliasing Pattern

Aliasing Pattern between *lg*(`postEventPrivate`), *lg*(`wakeup`)

$$\alpha = \text{isAliased}(\texttt{ob1}, \texttt{ob2.nextQueue}) \;\wedge\; \text{isAliased}(\texttt{ob2}, \texttt{ob2.nextQueue})$$

Interface Contract

```
void postEventPrivate (Event e) {            ||  void wakeup(boolean f) {
  ...                                        ||    ...
  synchronized (this) {                      ||    synchronized (this) {
    ...                                      ||      ...
    nextQueue.postEventPrivate(e);           ||      nextQueue.wakeup(f);
    ...                                      ||      ...
  }                                          ||    }
  ...                                        ||    ...
}                                            ||  }
```

For postEventPrivate, wakeup

$$\neg \text{isAliased}(\texttt{ob1}, \texttt{ob2.nextQueue}) \vee$$
$$\neg \text{isAliased}(\texttt{ob2}, \texttt{ob1.nextQueue})$$

Call-site $\models \mathcal{I} \Rightarrow$ **postEventPrivate** $\|$ **wakeup** is deadlock-free.

Approach: View from 10,000 feet

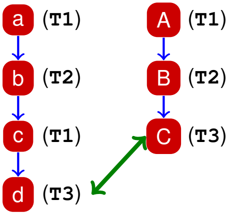
Compute:

- Lock-graphs for library methods (static analysis)
- DL-causing patterns for combinations of 2 or more methods.
- Derive Interface Contracts.

Outline

Prune Lock-graphs:

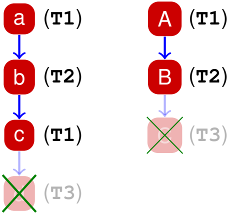
Remove nodes that cannot be part of cycle



- Terminal nodes that may alias only to other terminal nodes.

Prune Lock-graphs:

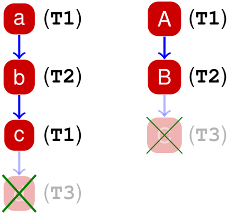
Remove nodes that cannot be part of cycle



- Terminal nodes that may alias only to other terminal nodes.

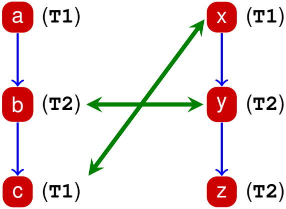
Prune Lock-graphs:

Remove nodes that cannot be part of cycle

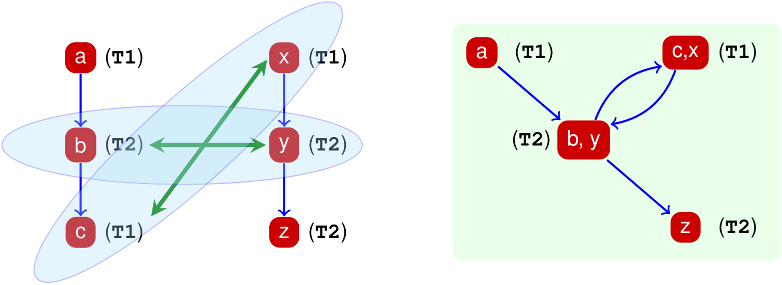


- Terminal nodes that may alias only to other terminal nodes.
- Initial nodes that may alias only to other initial nodes.

Smarter Enumeration by Subsumption



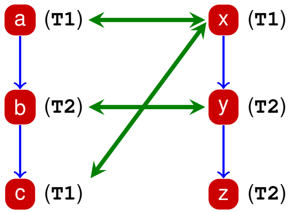
Smarter Enumeration by Subsumption

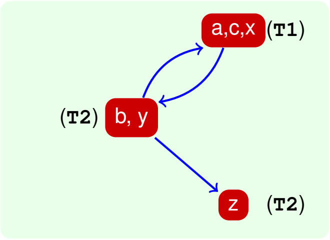


Deadlock-causing Aliasing Pattern (α_1)

$$\text{isAliased}(\mathbf{b}, \mathbf{y}) \wedge$$
$$\text{isAliased}(\mathbf{c}, \mathbf{x})$$

Smarter Enumeration by Subsumption





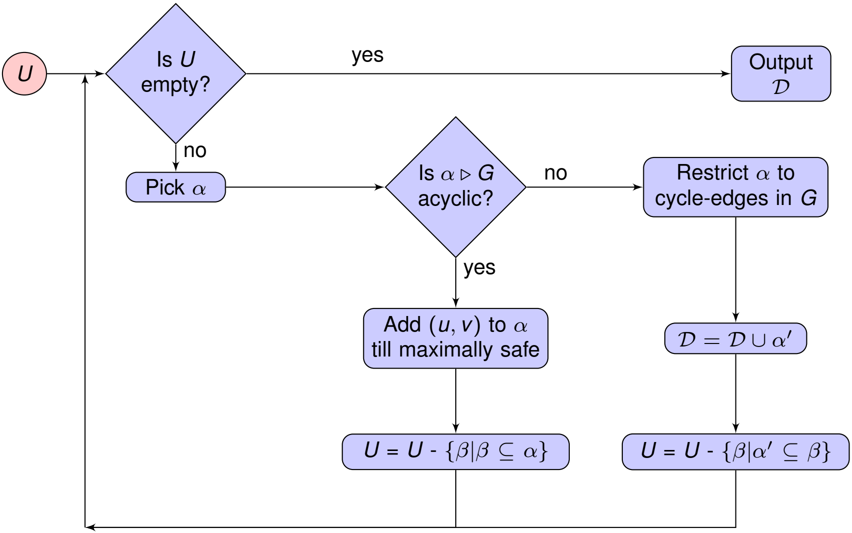
Deadlock-causing Aliasing Pattern (α_2)

$$\text{isAliased}(\mathbf{b}, \mathbf{y}) \wedge$$
$$\text{isAliased}(\mathbf{c}, \mathbf{x}) \wedge$$
$$\text{isAliased}(\mathbf{a}, \mathbf{x})$$

Subsumption

- α_2 subsumes α_1 : α_2 has more aliasing.
- DL with lesser aliasing $\Rightarrow$ DL with more aliasing.
- Only enumerate "minimally" unsafe patterns.
- Disregard subsuming patterns.

Explicit Enumeration



Outline

Aliasing Pattern Enumeration with SMT

Theorem

Enumerating all deadlock-causing aliasing patterns is NP-complete.

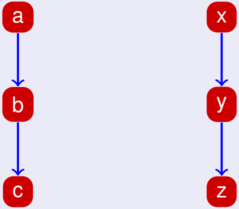
Symbolic Computation

- Encode Lock-Order Graphs as Inequality Constraints.
- Encode Aliasing as Equality Constraints.
- Transform Cycle Detection in a Graph to SAT of a Constraint.
- Use SMT solvers to check SAT.

Symbolic Encoding

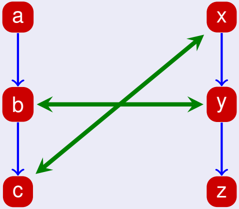
$$(\mathbf{a} < \mathbf{b} \quad \wedge \quad \mathbf{b} < \mathbf{c})$$

Symbolic Encoding



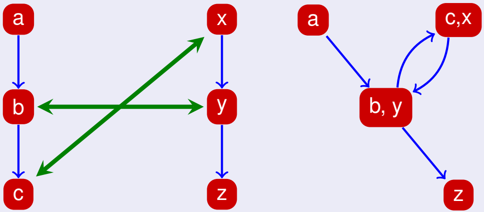
$$
\begin{array}{ll}
 & (\mathbf{a} < \mathbf{b} \quad \wedge \quad \mathbf{b} < \mathbf{c}) \\
\& & (\mathbf{x} < \mathbf{y} \quad \wedge \quad \mathbf{y} < \mathbf{z})
\end{array}
$$

Symbolic Encoding



$$
\begin{aligned}
& (\mathbf{a} < \mathbf{b} \quad \wedge \quad \mathbf{b} < \mathbf{c}) \\
\&\ & (\mathbf{x} < \mathbf{y} \quad \wedge \quad \mathbf{y} < \mathbf{z}) \\
\&\ & (\mathbf{c} = \mathbf{x} \quad \wedge \quad \mathbf{b} = \mathbf{y})
\end{aligned}
$$

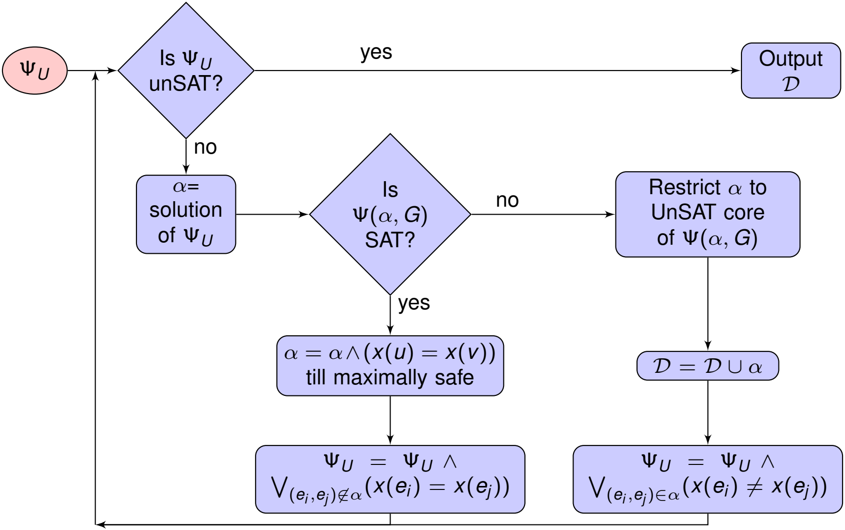
Symbolic Encoding



$$
\begin{aligned}
& (\mathbf{a} < \mathbf{b} \quad \wedge \quad \mathbf{b} < \mathbf{c}) \\
\& \ & (\mathbf{x} < \mathbf{y} \quad \wedge \quad \mathbf{y} < \mathbf{z}) \\
\& \ & (\mathbf{c} = \mathbf{x} \quad \wedge \quad \mathbf{b} = \mathbf{y})
\end{aligned}
$$

Cycle $\equiv$ UNSAT $\equiv$ deadlock!

Symbolic Algorithm



Outline

Experimental Results

Library Name		LOC	Time Taken (secs)	False + ves	Potential Deadlocks
ftpproxy	(ftp proxy)	1.0K	13.0	-	-
JavaFTP	(ftp client)	2.6K	9.0	-	-
cache4j	(object cache)	2.6K	15.0	-	-
netty	(network app f/w)	11.0K	14.0	-	-
apache-log4j	(logging service)	33.3K	130.1	1	1
oddjob	(job scheduler)	41.3K	250.0	-	-
hsqldb	(database engine)	157.6K	806.8	3	3
javax 1.6 sdk		534.3K	629.0	6	2
java 1.6 sdk		551.8K	1011.6	14	12
		> 1.3M	< 2880	24	18

Vindication

Most deadlocks identified correspond to real, live bug reports by developers!

Library Name	Method names	Bug Report
java.awt (EventQueue)	postEventPrivate, wakeup	Sun Bug DB ids: 4913324, 6424157, 6542185.
java.awt (Container)	removeAll, addPropertyChangeListener	OS-dir mail archive.
java.util (LogManager) (Logger)	addLogger getLogger	Sun Bug DB id: 6487638.
javax.swing (JComponent)	setFont paintChildren	Bug in Jajuk player
hsqldb (Session)	isAutoCommit close	OS-dir mail archive

With Interface Contracts, we get . . .

- better specification of (deadlock-free) thread-safe behavior,
- useful documentation for client developers,
- plug-in for statically analyzing existing client code, and,
- compositional flavor in reasoning about deadlocks.

Thank You!

Lock-order Graphs

Definition

Access Expression (a.e.): **ob** or sequence of nested fields of **ob**.

Definition (Lock-order Graph $G(V, E)$ for method m)

$(v_1, v_2) \in E \Leftrightarrow$:

- v_1 aliased to some a.e. x,
- v_2 aliased to some a.e. y,
- Path **lock(x)** $\rightarrow \ldots \rightarrow$ **lock(y)** in $cfg(m)$

Computing Lock-order Graphs

Summary = State after each program statement

- which locks currently held (*ls*)
- lock-order graph (*lg*)
- root nodes (*rs*), and,
- aliasing information,

Computing Lock-order Graphs

- Standard interprocedural summary-based forward static analysis.
- **`lock(x)`** = add x to ls, $\forall y \in ls$ add (y, x) to lg.
- **`unlock(x)`** = remove x from ls.
- Branch merge = union of summaries.
- Invocation of m = concatenate $lg(m)$ to current lg.

Deadlockability Analysis

Given library $\mathcal{L} = \{C_1, \ldots, C_m\}$

Methods $m_1, \ldots, m_k$ spread across classes $C_1, \ldots, C_m$.

Compute for all $m_1, \ldots, m_k$

Lock-order graphs $lg(m_1), \ldots, lg(m_k)$.

Check for each pair m_i, m_j

Is there any aliasing pattern s.t. $lg(m_i) \cup lg(m_j)$ has cycles?

Compute

$\mathcal{D}$: set of all *deadlock-causing aliasing patterns*.

So far . . .

Model Checking

- Generate global state graph.
- Explore all possible interleavings.

But...

May not scale after abstraction and partial order reduction.

So far . . .

Static Analysis

- Lock-acquisition order graph (*lg*) for each thread.
- Conservatively merge *lg* for concurrent threads.
- Cycle in merged graph $\Rightarrow$ possible deadlock.

But...

Too many false positives if analysis coarse, unscalable otherwise.

Deadlock-causing Aliasing Pattern Enumeration

Definition (Subsumption)

α_2 subsumes α_1 ($\alpha_1 \subseteq \alpha_2$) iff $\forall(u, v) : (u, v) \in \alpha_1 \Rightarrow (u, v) \in \alpha_2$.

Lemma (Given $\alpha_1 \subseteq \alpha_2$)

α_1 *is deadlock-causing* $\Rightarrow$ α_2 *is deadlock-causing.*

Definition (Minimally Unsafe)

α minimally unsafe iff for any (u, v), $\alpha - (u, v)$ is safe.

We only need to consider minimally unsafe patterns.

Deadlock-causing Aliasing Pattern Enumeration

Subsumption

- α_2 subsumes $\alpha_1 \Rightarrow \alpha_2$ has more aliasing than α_1.
- $\alpha_1 \subseteq \alpha_2$: α_1 is deadlock-causing $\Rightarrow \alpha_2$ is deadlock-causing.
- α minimally unsafe if removing any aliasing makes it safe.

We only need to enumerate minimally unsafe patterns!

Encoding Lock-Graph $G(V, E)$

- $x(v_i)$: topological rank of $v_i \in V$.
- $\Psi(G) = \bigwedge_{(v_i,v_j)\in E} (x(v_i) < x(v_j))$.

Encoding Aliasing Pattern α

$$\Psi(\alpha) = \bigwedge_{(v_i,v_j)\in\alpha} (x(v_i) = x(v_j))$$

Reduction to SAT

$\alpha \triangleright G$ has a cycle iff $\Psi(\alpha, G) = \Psi(G) \wedge \Psi(\alpha)$ is unsatisfiable.

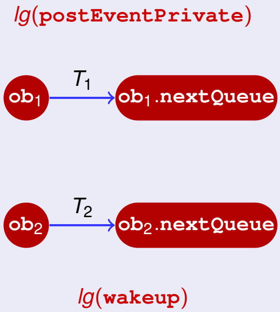
A few more (sound) filters...

Prune ...

- locks corresponding to **`final`** fields.
- **`private`** fields not accessed outside constructor/finalizer.
- immutable constants.
- **`private`** objects that cannot escape scope of methods.

Joint Lock-Order Graph without Aliasing

```
void postEventPrivate (Event e) {
  . . .
  synchronized (this) {
    nextQueue.postEventPrivate(e);
    . . .
  }
  . . .
}
void wakeup (boolean f) {
  . . .
  synchronized (this) {
    nextQueue.wakeup(f);
    . . .
  }
  . . .
}
```



Derive Interface Contracts

Definition (Interface Contract)

Compute $\mathcal{D}$: all deadlock-causing aliasing patterns.

$$\mathcal{I}(m_i, m_j)\text{: } \bigwedge_{\alpha \in \mathcal{D}} \bigvee_{(e_i, e_j) \in \alpha} \neg\text{isAliased}(e_i, e_j).$$

Call-site of m_i, m_j satisfies $\mathcal{I} \Rightarrow m_i \parallel m_j$ is deadlock-free.