A Compositional Static Deadlock Detector for Android Code Revisions

Nikos Gorogiannis (Facebook London)

Joint work with James Brotherston, Paul Brunet, Max Kanovich (UCL)

Contents

- Motivation & Problem Statement
- An Analysis for Deadlocks
- Adaptation to Java & Implementation
- Impact & Future Work

Problem Statement

Find ***deadlocks*** introduced by revisions,
during ***code review*** (in <15min),
on app code in the ***10s of MLoC***,
running on 1000s of revisions/day.

- Deadlock analyses are whole-program.
- A deadlock involves two traces.
  - Often only one trace is affected by a revision.
- We can't afford to analyse the whole program here.

Approach

- ***Partial-program*** analysis of modified files in revision.
- Compositional summarisation of each method.
  - Sequential analysis of ***lock behaviour***.
- Concurrency check:
  - What methods may run in parallel to Foo?
    Use locks acquired by Foo to find these methods.
  - Collect static information on thread identity.

An Analysis for Deadlocks

Abstract Language

$\mathscr{L}$: set of global lock names for recursive/reentrant locks

$$\begin{aligned} C := \; & \texttt{skip} \mid p() \mid \texttt{acq}(\ell) \mid \texttt{rel}(\ell) \mid C; C \\ & \mid \texttt{if}(*) \; \texttt{then} \; C \; \texttt{else} \; C \mid \texttt{while}(*) \; \texttt{do} \; C \end{aligned}$$

Non-deterministic control; no recursion.

Top-level programs must be ***balanced*** wrt locking (**synchronized**).

Op. semantics via tracking lock states $L : \mathscr{L} \to \mathbb{N}$.

Deadlock = absence of transitions to a next state.

Critical Pairs

$(X, y) \in \mathrm{Crit}(C)$

where X is a set of locks and y is a lock such that $y \notin X$

Intuitively (definition in the paper):

C ***acquires*** *lock* y *(which it does not hold)*

while it holds precisely the locks in X

Thm. $C_1 \mid\mid C_2$ deadlocks iff there are critical pairs $(X_1, \ell_1) \in \mathrm{Crit}(C_1)$ and $(X_2, \ell_2) \in \mathrm{Crit}(C_2)$ such that

$$X_1 \cap X_2 = \emptyset \text{ and } \ell_1 \in X_2 \text{ and } \ell_2 \in X_1$$

A Static Analysis for Critical Pairs

Abstract States:

$$\alpha = \langle L, Z \rangle$$

$Z \subseteq 2^{\mathscr{L}} \times \mathscr{L}$ is a set of critical pairs

$L : \mathscr{L} \to \mathbb{N}$ is a thread-local lock state

Prop. For any balanced C: $$[\![C]\!]\alpha_\perp = \langle \varnothing, \mathrm{Crit}(C) \rangle$$

Thm. Checking $P = C_1 \,||\, \ldots \,||\, C_n$ for deadlock can be done in exponential time in $|P|$ and is in NP.

Example Deadlock

a . foo(b) || b . bar(a) deadlocks:

```
class A {
  public synchronized void foo(B b) { b.foo(); }
  public synchronized void bar() {}
}
```

```
class B {
  public synchronized void bar(A a) { a.bar(); }
  public synchronized void foo() {}
}
```

Critical Pairs

$\{(\varnothing, a), (\{a\}, b)\}$

$\{(\varnothing, a)\}$

$\{(\varnothing, b), (\{b\}, a)\}$

$\{(\varnothing, b)\}$

Pairs $(\{a\}, b)$ and $(\{b\}, a)$ satisfy the deadlock conditions:

- $\{a\} \cap \{b\} = \varnothing$
- $b \in \{b\}$ and $a \in \{a\}$

Adaptation to Java & Implementation

Implementation and Adaptations

- Implemented in ***Infer*** (open source, OCaml, ~3kLoC).
- Locks represented as ***access paths*** ($\text{this}.f.g.h$).
- ***Thread identity***: main-thread, worker, both, neither.
  - Android lifecycle, annotations, assertions.
  - + Class hierarchy + back-propagation over calls.

Analysis applied to Code Revisions

```
class A {
  public synchronized void foo(B b) { b.foo(); }
  public synchronized void bar() {}
}

class B {
  public synchronized void bar(A a) { a.bar(); }
  public synchronized void foo() {}
}
```

(modification)

1. $a.\mathrm{foo}(b)$ is analysed; it has the pair $(\{a : A\}, b : B)$.
2. Since $a.\mathrm{foo}(b)$ takes a lock in class B, analyse all methods in B (which may run in parallel with $a.\mathrm{foo}(b)$).
3. Does any pair of $a.\mathrm{foo}(b)$ satisfy the deadlock conditions against any pair from methods in B?

Impact & Future Work

Impact

- Analysed ***>100k of revisions*** in >2 years.
- Issued ***>500 reports***, with long traces.
- Fix rate is ***>50%***.
- In last 100 days,
  - Infer analysis runtime on average=***~200sec***.
  - #methods/revision analysed on average=***~5k***.

Future Work

- Is the problem NP-complete?
- Which adaptations admit further study?
  - Treatment of access paths.
- Can we modestly enlarge the set of dependencies?
  - Eg, by precomputing the locks used by classes.

Thanks!

https://fbinfer.com