

# A Compositional Deadlock Detector for Android Java

---

Darius Mureşan   Henrique Luz   Rui Xavier

November 12, 2025

Aarhus University  
Department of Computer Science

# Overview

---

- We present a **compositional static analysis** for detecting deadlocks in Android Java.
- Designed for **industrial-scale** codebases (tens of millions of lines).
- Implemented in **INFER**, deployed at Facebook for 2+ years.
- Achieves a **54% developer fix rate**.

# Motivation

---

# Why Study Deadlocks?

- Deadlocks are a key challenge in concurrent programming.
- Occur when threads **cyclically wait on each other's locks**.
- Result: entire system halts — loss of responsiveness, reliability.
- Especially relevant for **Android apps** using Java's synchronized blocks.

## Dijkstra's Dining Philosophers

Five philosophers share forks and a bowl of spaghetti. Each must hold two forks to eat — potential for a circular wait.

- Illustrates the essential nature of deadlock: mutual waiting.
- Analogous to threads waiting for locks.

# Deadlocks in Industry

- At Facebook, Android apps exceed **10M+ lines of code**.
- Thousands of commits daily  $\Rightarrow$  rapid iteration.
- Developers need feedback in **under 15 minutes**.
- Whole-program analyses are too slow and memory-intensive.

## Goal

**Fast, scalable, accurate deadlock detection** integrated into CI pipelines.

# Challenges in Traditional Analyses

- Reanalyze entire program on each commit.
- Poor scalability and high false-positive rates.
- Lack **compositionality**.
- Need: an analysis that focuses only on changed code and its dependencies.



## Research Gap and Question

---

- Existing tools are **non-compositional**.
- They often trade soundness for scalability.
- We need a **mathematically grounded, incremental** analysis.

## Core Challenge

Can we design a deadlock detector that is both **sound in theory** and **scalable in practice**?

## Main Question

Can we develop a **compositional deadlock detector** for **Android Java** that is sound, complete, and efficient for large industrial codebases?

- Model Java concurrency with **balanced, re-entrant locks**.
- Detect deadlocks via **critical pairs**.
- Integrate into **INFER** for continuous integration.

# Concurrent Programs

---

**Rui** ► *Is this too much text for one slide?* ◀

- Simplified model of Java concurrency:

$$C ::= \text{skip} \mid p() \mid \text{acq}(l); C; \text{rel}(l) \mid C; C \\ \mid \text{if}(*) \text{ then } C \text{ else } C \mid \text{while}(*) \text{ do } C$$

- This grammar guarantees balanced pairs of lock acquisitions/releases.
- We consider only non-recursive procedures.
- If  $C$  is balanced, then so is  $\text{body}(p)$  with  $p \in \text{callees}(C)$ .
- A parallel program is an  $n$ -tuple of balanced statements  $C_1 \parallel C_2 \parallel \dots \parallel C_n$ .

- We treat locks as **re-entrant and scoped**.

- Each thread has a **lock state** which maps locks to acquisition counts.
- Deadlock occurs when every thread can take a local step, but no joint step is possible.

$$\langle C_1 || C_2, (L_1, L_2) \rangle$$

## Key Idea

Deadlock arises when threads hold disjoint locks yet each waits on a lock held by another.

# Program Execution Traces

---



- Executions can be represented as strings of lock acquisitions/releases — e.g. 'x y y x'.
- Balanced programs produce **Dyck words** (well-nested parentheses).
- Captures the essential locking behaviour.

$$L(C) = \{\text{all possible lock traces of } C\}$$

### Example

```
acq(x); if(*) then acq(y); rel(y); else acq(z);  
rel(z); rel(x)
```

$$L(C) = \{ xyyx, xzzx \}$$

- Balanced structure guarantees decidability.
- Enables abstract reasoning about possible interleavings.

- Each balanced statement can be viewed as a **finite automaton** over lock actions.
- Allows algorithmic computation of lock dependencies.
- Provides the foundation for critical-pair analysis.

# Soundness and Completeness

---

## Critical Pair

$(X, \ell)$ : some execution acquires lock  $\ell$  while holding all locks in  $X$ .

- Captures possible lock dependencies.
- Computed for each sequential thread.

### Theorem 4.4 — Deadlock Condition

Program  $C_1 || \dots || C_n$  deadlocks iff there exist critical pairs  $(X_i, \ell_i)$   
s.t.

$$\ell_i \in \bigcup_{j \neq i} X_j \quad \text{and} \quad X_i \cap \bigcup_{j \neq i} X_j = \emptyset$$

**Intuition:** Each thread holds a lock another needs.

# Illustrative Example

## Two-Thread Example

C1:  acq(x); acq(y); rel(y); rel(x) C2:  acq(y);  
acq(x); rel(x); rel(y)

$$\text{Crit}(C1) = \{(\emptyset, x), (\{x\}, y)\}$$

$$\text{Crit}(C2) = \{(\emptyset, y), (\{y\}, x)\}$$

Since  $x \in \{y\}$  and  $y \in \{x\}$ , the condition holds — **deadlock!**

- Define executions  $\rightarrow$  and parallel composition.
- Show equivalence between execution semantics and trace semantics.
- Prove critical-pair condition is **sound** (no missed deadlocks) and **complete** (no spurious ones).

## Result

Existence of deadlock  $\Leftrightarrow$  conflict between critical pairs.



# Complexity

---

- Recursive equations (C1–C6) compute  $\text{Crit}(C)$  compositionally.
- Each construct (if, while, seq) has a local combination rule.
- **Example:**

$$\text{Crit}(\text{acq}(\ell); C; \text{rel}(\ell)) = \{(\emptyset, \ell)\} \cup \{(X \cup \{\ell\}, \ell') \mid (X, \ell') \in \text{Crit}(C)\}$$

- **Finite and computable:**  $\text{Crit}(C)$  always finite.
- Deadlock detection problem is **decidable** and in NP.
- Non-recursive programs  $\Rightarrow$  quadratic time.
- With procedures  $\Rightarrow$  quasi-exponential.

# Implementation

---

- Implemented as an **abstract interpretation** within INFER.
- Computes method summaries: critical pairs + thread identity.
- Compositionally reuses summaries of unchanged methods.

## Core Idea

Analyse only modified methods and their dependents.

# Abstract State Representation

$\alpha = \langle L, Z \rangle$  where  $L = \text{lock state}$ ,  $Z = \text{set of critical pairs}$

- Join operation:  $\langle L, Z_1 \rangle \sqcup \langle L, Z_2 \rangle = \langle L, Z_1 \cup Z_2 \rangle$
- Each command updates this abstract state.

- Procedure call depends only on:
  1. Current abstract state.
  2. Precomputed summary of the callee.
- Enables incremental reanalysis — ideal for CI/CD.

$$Jp()K\langle L, Z \rangle = \langle L, Z \cup f(L, \text{Crit}(\text{body}(p))) \rangle$$

- **Balanced locking:** uses `synchronized`.
- **Partial path sensitivity:** e.g., for `tryLock()` and UI threads.
- **Lock naming:** access-path abstraction (`this.f.g`, etc.).
- **Thread inference:** uses annotations like `@UiThread`.



## Deployment and Results

---

- Deployed as part of Facebook's **continuous integration system**.
- Runs automatically on every Android commit.
- Appears as an automated “reviewer” commenting on potential deadlocks.

- Deployed for **2+ years** on all Android commits.
- **500+** deadlock reports issued.
- **54%** of reports fixed by developers.
- Median runtime (all analyses): **90 s per commit**.
- Analyses **2k–5k methods per commit**.

## Conclusion and Related Work

---

- Builds on automata-theoretic analyses of **pushdown systems**.
- Compared to prior tools:
  - Compositional, not whole-program.
  - Targets balanced re-entrant locks.
  - Prioritizes **actionable results** over completeness.

- Developed a **sound and complete** compositional analysis.
- Scales to tens of millions of lines.
- Successfully deployed in industry with tangible impact.
- Formalized and proven in **Coq (8.7k LOC)**.

## Future Work

Extend to recursive calls, deterministic control, and nested parallelism.

**Thank you!**

**Questions?**