A Compositional Deadlock Detector for Android Java

Darius Mureșan Henrique Luz Rui Xavier

November 12, 2025

Aarhus University
Department of Computer Science

Overview

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.

Classical Example

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

The Research Gap

- 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**?

Research Question

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.

Critical Pairs and Deadlock Detection

Critical Pairs

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

Definition

A **critical pair** of a thread is a pair (X, ℓ) such that some execution of the thread acquires an unheld lock ℓ while already holding the set of locks X.

Deadlock Condition

For the case of two threads:

$$C1 \mid\mid C2 \quad \textit{deadlocks} \iff \text{there are critical pairs}$$
$$(X1,\ \ell 1) \text{ and } (X2,\ \ell 2) \text{ of C1 and C2 respectively, such that:}$$

$$\ell 1 \in X2 \quad \textit{and} \quad \ell 2 \in X1 \quad \textit{and} \quad X1 \cap X2 = \emptyset$$

Example of Critical Pairs

Example

Consider the two threads:

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

Their critical pairs are:

$$Crit(C1) = \{(\emptyset, x), (\{x\}, y)\}$$
$$Crit(C2) = \{(\emptyset, y), (\{y\}, x)\}$$

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

Example of Critical Pairs

Solution

Consider the two threads:

$$C1' : acq(z); C1; rel(z)$$
$$C2' : acq(z); C2; rel(z)$$

Their critical pairs are:

$$Crit(C1') = \{(\emptyset, z), (\{z\}, x)\}, (\{z, x\}, y)\}$$
$$Crit(C2') = \{(\emptyset, z), (\{z\}, y)\}, (\{z, y\}, x)\}$$

Since $\{z, x\} \cap \{z, y\} = z$, the deadlock condition does not hold — no deadlock!

z acts as a guard lock, preventing x and y from being accessed by C1' and C2' simultaneously.

Concurrent Programs

Abstract Language

- Simplified model of Java concurrency:

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

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

Program Execution Traces

Lock States

- We treat locks as **re-entrant and scoped**.
- A *lock state* is a function that maps each lock to its acquisition count.
- A *configuration* is a pair $\langle C, L \rangle$, where C is a statement and L is the lock state.
- A *concurrent configuration* is then, a pair of the form $\langle C_1 || ... || C_n, (L_1, ..., L_n) \rangle$.

Executions as 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 Trace

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.

Traces as Finite Automata

- 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, ℓ): some execution acquires lock ℓ while holding all locks in X.

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

Key Theorem (Simplified)

Theorem 4.4 — Deadlock Condition

Program $C_1||...||C_n$ deadlocks iff there exist critical pairs (X_i, ℓ_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)`

$$\mathrm{Crit}(\mathrm{C1}) = \{(\emptyset, x), (\{x\}, y)\}$$
$$\mathrm{Crit}(\mathrm{C2}) = \{(\emptyset, y), (\{y\}, x)\}$$

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

Proof Structure

- 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

Computing Critical Pairs

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

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

Complexity Results

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

Implementation

Implementation Overview

- 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

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

- 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.

Compositionality

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

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

Analysis for Android Java

- **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

Industrial Deployment

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

Results and Impact

- 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

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.

Conclusion

- 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?