A Compositional Deadlock Detector for Android Java

Darius Mureșan Henrique Luz Rui Xavier

November 19, 2025

Aarhus University
Department of Computer Science

Motivation

Deadlocks: A Refresher

- A **deadlock** occurs when two or more threads are each waiting for the other to release a lock, so none can make progress.

Example (two threads, two locks)

```
Thread 1:  acq(x); acq(y);
Thread 2:  acq(y); acq(x);
```

- One possible execution:
  1. Thread 1 acquires x
  2. Thread 2 acquires y
  3. Thread 1 now waits for y, Thread 2 waits for x
- Neither thread can continue $\Longrightarrow$ deadlock.

Why Deadlocks Matter

- Concurrency is everywhere in Android apps:
    - UI thread + many background threads
    - Callback-driven, event-based execution
    - Shared mutable state (objects used as locks)
- A deadlock freezes part (or all) of the app:
    - UI becomes unresponsive
    - Background tasks never finish
    - Users perceive the app as *broken*
- Avoiding and detecting deadlocks is a core reliability problem.

Industrial Setting: Facebook Android Apps

- Target: large Android apps under continuous development:
  - Tens of millions of lines of code (LoC)
  - Thousands of revisions per day
- Goal of the analysis:
  - Run at code-review time on every commit
  - Give feedback to developers **within minutes**
- This rules out whole-program, from-scratch analyses for every change.

Key Challenges

- **Scalability**:
  - Cannot re-analyse the whole app for each commit
  - Need a compositional, incremental analysis
- **Usefulness for developers**:
  - Too many false positives $\Rightarrow$ warnings are ignored
  - The focus is on actionable reports, not proving absence of deadlocks
- **The research question**: can we get both **theoretical guarantees** and **practical performance** at this scale?

Research Gap and Question

Limitations of Existing Approaches

- Many existing deadlock analyses:
  - Assume access to the **whole program**
  - Do not support compositional, change-focused analysis
- Tools often prioritise:
  - Strong soundness guarantees
  - But with high false-positive rates in practice
- For huge codebases, this is misaligned with developer needs:
  - Developers prefer fewer, more trustworthy warnings
  - It is acceptable to miss some rare deadlocks

Research Question

Main Question

Can we design a **compositional** static analysis for **Android Java** that detects deadlocks in large codebases, with:

- A clean **theoretical characterisation** of deadlocks
- A **decidable** and **tractable** core problem
- A practical implementation integrated in CI

High-Level Contributions

- Abstract language with balanced re-entrant locks and nondeterministic control
- New deadlock condition based on critical pairs of threads
- Proof that deadlock detection in this language is decidable and in NP
- Compositional implementation in Facebook's `INFER`, deployed at scale

Concurrent Programs

Abstract Language for Concurrency

- Abstracts away data and heap; focuses only on locks and control.
- Statements C are built from:

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

- **Balanced statements** enforce lock discipline:
  - Locks must be released in LIFO order
  - Models scoped constructs like `synchronized` and `std::lock_guard`
- A parallel program is an n-tuple $C_1 \mid\mid \ldots \mid\mid C_n$.

Example Program in the Abstract Language

Example

$$C = \mathit{acq}(x); (\texttt{if(*) then } \mathit{acq}(y); \mathit{rel}(y) \texttt{ else skip}); \mathit{rel}(x)$$

- This program:
  - Always acquires and releases x
  - Nondeterministically either:
    - acquires and releases y, or
    - does nothing in the branch
- Shows how nested acquisitions map to lock scopes in Java.

Lock States and Configurations

- Locks are **re-entrant**:
  - A thread can acquire the same lock multiple times
  - Lock state $L : \text{Locks} \to \mathbb{N}$ counts acquisitions
- We write $\lfloor L \rfloor = \{\ell \mid L(\ell) > 0\}$ for held locks.
- A configuration is a pair $\langle C, L \rangle$:
  - C: current statement
  - L: lock state
- A concurrent configuration:

$$\langle C_1 || \ldots || C_n, (L_1, \ldots, L_n) \rangle$$

where each thread has its own lock state.

Deadlock Definition (1)

- Sequential step: $\langle C, L\rangle \to \langle C', L'\rangle$.

Example

$$\langle \texttt{acq(x);skip;rel(x)}, L\rangle \;\to\; \langle \texttt{skip;rel(x)}, L[x{+}{+}]\rangle$$

(acquiring lock x increases its count)

- Parallel step: advance one thread, provided it does not grab a lock already held by another thread.
- Intuitively, we are deadlocked when:
  - Each thread can individually make a step (sequential)
  - But no *parallel* step is possible any more

Deadlock Definition (2)

- Formal definition (simplified):
    - A concurrent configuration is deadlocked if at least two threads are stuck in this way.
    - A program deadlocks if some reachable configuration is deadlocked.

Critical Pairs and Deadlock Detection

Intuition: Critical Pairs

Critical Pair (Informal)

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 exactly the set of locks X.

- Captures which locks are held when a new lock is acquired.
- Abstracts away the concrete control-flow and interleavings.
- Each thread has a **finite** set of critical pairs.

Critical Pairs: A Simple Example

Statement

$$C = acq(x);\ acq(y);\ rel(y);\ rel(x)$$

Computing Crit(C)

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

- At the first acquisition of x: no locks held $\Rightarrow (\emptyset, x)$
- Later, the thread holds $\{x\}$ when acquiring $y \Rightarrow (\{x\}, y)$

Two-Thread Deadlock Condition

Deadlock Condition (2 Threads)

For two statements C_1 and C_2 running concurrently:

$$C_1 \,||\, C_2 \text{ deadlocks } \iff \exists (X_1, \ell_1) \in \mathrm{Crit}(C_1), (X_2, \ell_2) \in \mathrm{Crit}(C_2)$$

such that

$$\ell_1 \in X_2, \quad \ell_2 \in X_1, \quad X_1 \cap X_2 = \emptyset.$$

- Each thread holds a lock the other is trying to acquire.
- The currently held lock sets do not overlap.
- Generalises to arbitrarily many threads.

Example: Classic Two-Thread Deadlock

Two Threads

$$C_1 : \ \mathit{acq}(x); \mathit{acq}(y); \mathit{skip}; \mathit{rel}(y); \mathit{rel}(x)$$
$$C_2 : \ \mathit{acq}(y); \mathit{acq}(x); \mathit{skip}; \mathit{rel}(x); \mathit{rel}(y)$$

- Critical pairs:

$$\mathrm{Crit}(C_1) = \{(\emptyset, x), (\{x\}, y)\}$$
$$\mathrm{Crit}(C_2) = \{(\emptyset, y), (\{y\}, x)\}$$

- Take $(X_1, \ell_1) = (\{x\}, y)$ and $(X_2, \ell_2) = (\{y\}, x)$:
  - $\ell_1 = y \in X_2 = \{y\}$
  - $\ell_2 = x \in X_1 = \{x\}$
  - $X_1 \cap X_2 = \emptyset$
- Condition holds $\Rightarrow$ deadlock is possible.

Guard Locks: Breaking the Deadlock

Adding a Guard Lock

$$C_1' = acq(z); C_1; rel(z)$$
$$C_2' = acq(z); C_2; rel(z)$$

- Now:

$$\mathrm{Crit}(C_1') = \{(\emptyset, z), (\{z\}, x), (\{z, x\}, y)\}$$
$$\mathrm{Crit}(C_2') = \{(\emptyset, z), (\{z\}, y), (\{z, y\}, x)\}$$

- Any potentially conflicting critical pairs share lock z:

$$\{z, x\} \cap \{z, y\} = \{z\} \neq \emptyset$$

- Deadlock condition fails $\Rightarrow$ no deadlock.
- Intuition: z acts as a *guard lock* protecting the region.

Program Execution Traces

Executions as Traces

- Each step either:
  - Leaves the lock state unchanged
  - Acquires a lock ℓ
  - Releases a lock ℓ
- We record only lock actions as a **trace** over alphabet Σ:

$$\Sigma = \{\ell \mid \ell \in \mathsf{Locks}\} \cup \{\bar{\ell} \mid \ell \in \mathsf{Locks}\}$$

Example

- Statement: `acq(x); if(*) then acq(y); rel(y) else acq(z); rel(z); rel(x)`
- Possible traces: $xy\overline{yx}$ and $xz\overline{zx}$

Executions as Traces (Examples)

Example Statement

$$C = acq(x); acq(y); rel(y); rel(x)$$

- Valid trace:

$$x\, y\, \overline{y}\, \overline{x}$$

  Let us now consider:

$$C' = acq(x); acq(y); rel(x); rel(y)$$

- Invalid trace (bad nesting):

$$x\, y\, \overline{x}\, \overline{y} \quad \Rightarrow \quad \text{lock } x \text{ released while } y \text{ still held}$$

Dyck Words and Balanced Locking

- For balanced statements, traces are **Dyck words**:
  - Well-parenthesised strings of opens and closes
  - Locks always released in reverse order of acquisition
- Key property:
  - For any balanced statement C, all traces in $L(C)$ are Dyck words.
  - Executions never underflow the lock stack.
- This structure is crucial for the decidability and the critical-pair characterisation of deadlocks.

Dyck Words and Balanced Locking (Example)

- For balanced statements, traces form **Dyck words**:
  - Perfectly nested lock acquisitions / releases
  - The lock stack never underflows

Valid Dyck Word

$$x\, y\, \overline{y}\, z\, \overline{z}\, \overline{x}$$

- Corresponds to nested Acquire–Release behaviour:

$$acq(x);\ acq(y);\ rel(y);\ acq(z);\ rel(z);\ rel(x)$$

Invalid Word

$$x\, y\, \overline{x}\, \overline{y}$$

- Releases x before y: violates LIFO discipline.

Languages of Statements

- Each statement C defines a language $L(C) \subseteq \Sigma^\star$:
  - All traces of possible executions of C
- Defined inductively:

$$\mathsf{L}(\texttt{skip}) = \{\varepsilon\} \qquad \mathsf{L}(\mathit{rel}(\ell)) = \{\bar{\ell}\}$$

$$\mathsf{L}(p()) = \mathsf{L}(\mathsf{body}(p)) \qquad \mathsf{L}(C_1; C_2) = \mathsf{L}(C_1) \cdot \mathsf{L}(C_2)$$

$$\mathsf{L}(\mathit{acq}(\ell)) = \{\ell\} \qquad \mathsf{L}(\texttt{while(*) do } C) = \mathsf{L}(C)^*$$

$$\mathsf{L}(\texttt{if(*) then } C_1 \texttt{ else } C_2) = \mathsf{L}(C_1) \cup \mathsf{L}(C_2)$$

- For balanced C, $L(C)$ is regular and consists of Dyck words.

Soundness and Completeness

Cumulative Lock Effect of a Trace

- For a trace $u \in \Sigma^*$, $\langle u \rangle$ denotes its **cumulative lock effect**:
  - Start from the empty lock multiset
  - Read actions in u left to right
  - Each x increments the count of lock x
  - Each $\overline{x}$ decrements the count of lock x
- The set of *currently held locks* after reading u is:

$$\lfloor \langle u \rangle \rfloor = \{\, \ell \mid \langle u \rangle(\ell) > 0 \,\}.$$

Example

Trace prefix: $u = x\, y\, \overline{y}$

$$\begin{aligned} \langle \varepsilon \rangle &= \emptyset \\ \langle x \rangle &= \{x\} \\ \langle xy \rangle &= \{x, y\} \\ \langle xy\overline{y} \rangle &= \{x\} \end{aligned}$$

So:

$$\lfloor \langle u \rangle \rfloor = \{x\}.$$

Critical Pairs: Formal Definition

Critical Pairs of a Statement

For a balanced statement C:

$$\mathrm{Crit}(C) = \{(\lfloor\langle u\rangle\rfloor, \ell) \mid \exists v.\ u\ell v \in L(C) \text{ and } \ell \notin \lfloor\langle u\rangle\rfloor\}$$

where $\langle u\rangle$ is the cumulative lock effect of trace u.

- This is equivalent to: *there exists an execution that acquires ℓ while holding exactly X.*
- The paper shows the execution-based and language-based definitions coincide.

Critical Pairs: Formal Definition (Example)

Example Trace

$$u\ell v = x\, y\, z\, \overline{z}\, \overline{y}\, \overline{x}$$

- Take $u = x\, y$, $\ell = z$:

$$\langle u \rangle = \{x, y\}$$

- Since $z \notin \{x, y\}$, acquiring z yields:

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

- Meaning: During some execution, the thread acquires z while holding exactly $\{x, y\}$.

Deadlock Characterisation Theorem

Theorem 4.4 (Simplified)

A parallel program $C_1 || \dots || C_n$ deadlocks iff there exists an index set $I \subseteq \{1, \dots, n\}$ with $|I| \geq 2$ and critical pairs $(X_i, \ell_i) \in \mathrm{Crit}(C_i)$ for each $i \in I$ such that:

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

- Each thread holds locks needed by the others.
- Held-lock sets are pairwise disjoint.
- This condition is both **sound** and **complete**.

Complexity

Computing Critical Pairs Compositionally

- The paper gives 6 equations describing $\mathsf{Crit}(C)$ *by syntax* of C.
- Examples:

$$\mathsf{Crit}(\texttt{skip}) = \emptyset$$

$$\mathsf{Crit}(p()) = \mathsf{Crit}(\mathsf{body}(p))$$

$$\mathsf{Crit}(C; C') = \mathsf{Crit}(C) \cup \mathsf{Crit}(C')$$

$$\mathsf{Crit}(\texttt{if}\,(*)\ \texttt{then}\ C\ \texttt{else}\ C') = \mathsf{Crit}(C) \cup \mathsf{Crit}(C')$$

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

- These identities allow a bottom-up computation of $\mathsf{Crit}(C)$.

Complexity Bounds

- **Finite** and **computable**:
  - For any balanced C, $\mathrm{Crit}(C)$ is finite.
- Complexity of deadlock detection:
  - The deadlock problem for this language is **decidable** and lies in **NP**.
  - Idea: nondeterministically guess a set of threads and critical pairs, then check the deadlock condition in polynomial time.
- For programs without procedure calls:
  - Computing $\mathrm{Crit}(C)$ is polynomial in program size.
- Lower bounds (e.g. NP-completeness) are left as future work.

Implementation

Core Abstract Interpretation

- Implementation is an abstract interpretation that computes critical pairs.
- Abstract state:

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

  where

  - L: abstract lock state
  - Z: set of (approximate) critical pairs
- Each command C defines a transformer $[\![C]\!](\alpha)$.
- The join operation on states:

$$\langle L, Z_1 \rangle \sqcup \langle L, Z_2 \rangle = \langle L, Z_1 \cup Z_2 \rangle$$

Compositionality of the Analysis

- Procedure calls are handled via **summaries**:
  - For each procedure p we precompute $\mathrm{Crit}(\mathrm{body}(p))$.
  - At a call $p()$, we combine the current state with p's summary.
- Consequences:
  - When code changes, only affected procedures and their callers need re-analysis.
  - Most of the program can be reused from previous runs.
  - This is essential for deployment in continuous integration.

Adapting to Android Java

- The abstract language is mapped to real Java/Android code:
  - `synchronized` methods/blocks $\Rightarrow$ balanced lock regions
  - Re-entrant monitors modelled as nested acquisitions
- Android-specific refinements:
  - Partial path-sensitivity for methods like `tryLock()` and UI-thread checks
  - Lock naming via access-paths (`this.f.g`, etc.)
  - Thread identity domain (`@UiThread`, `@WorkerThread`, background)
- Implemented as the `starvation` analyser inside `INFER`.

Deployment and Results

Integration in Facebook's CI

- `INFER` is part of Facebook's continuous integration:
  - Every Android commit triggers static analyses, including deadlock analysis.
  - The analyser appears as an automated reviewer on code reviews.
- The deadlock analysis targets **code changes**, not whole apps:
  - Summarise modified methods and their dependents
  - Use heuristics to find relevant methods that share locks

Quantitative Impact

- Deployed on all Android commits for $\sim$2 years.
- Scale:
  - Hundreds of thousands of commits analysed
  - Typically 2k – 5k methods per commit
- Performance:
  - Median analysis time of 90 seconds per commit
  - Average analysis time of 213 seconds per commit
- Effectiveness:
  - 500+ deadlock reports issued
  - $\sim$ 54% of these reports were fixed by developers

Practical Considerations

- The tool optimises for **actionability**, not pure soundness:
  - Prefers fewer, high-quality warnings
  - Accepts some false negatives to keep the noise low
- Some non-fixed reports:
  - May still be real bugs (fixed elsewhere or considered low priority)
  - Some are false positives (e.g. infeasible concurrency patterns)
- Overall: evidence that the analysis finds real, impactful bugs at scale.

Conclusion and Related Work

Related Work

- **Automata-theoretic** approaches
- **Static analyses** for deadlocks
- **Dynamic and hybrid** techniques
- This paper:
  - Places a compositional static analysis with strong theory into this landscape
  - Focuses on large-scale industrial deployment

Takeaways

- A new, **critical-pair based** characterisation of deadlocks for a balanced lock language.
- Deadlock detection in this setting is **decidable** and in **NP**.
- A compositional implementation scaled to tens of millions of LoC in production.
- **Formalisation**: full development mechanised in Coq ($\sim$ 8.7k LOC).

Future Work

Future Directions

- Extend the theory to richer languages (recursion, deterministic guards, nested parallelism)
- Sharpen complexity bounds (e.g. NP-completeness)
- Explore similar compositional ideas for other concurrency bugs

Pros and Cons

Pros

- Achieves a balance between solid theory and real industrial deployment.
- Compositionality is not only formal but practical in CI workflows.
- Critical pairs provide compact, human-interpretable summaries of thread behaviour.
- Abstraction is stable across code evolution, making the analysis maintainable long-term.

Cons

- Compositionality may miss multi-layer deadlocks spread across the call graph (false negatives).
- Strong reliance on balanced locking assumptions; custom sync patterns break the model.
- The first full-program analysis remains expensive despite good per-commit performance.
- The reported statistics aggregate all categories of deadlock reports; separating them would clarify how many were real bugs, low-priority issues, or false positives.
- The evaluation reports total CI analysis time, but does not isolate the cost of *this* analyser, making its standalone overhead harder to assess.

Thank you!

Questions?