Oral Exam Notes: A Compositional Deadlock Detector for Android Java

Based on Brotherston et al. (ASE 2021)

1. Motivation & Context (Approx. 1 Minute)

Goal: Establish why this research exists and the specific industrial constraints.

- **The Problem:** Concurrency is ubiquitous in Android (UI thread + background threads), making deadlocks a core reliability issue that freezes apps.
- **The Industrial Setting:** The authors target Facebook's massive Android codebases (tens of millions of LoC) with thousands of daily commits.
- **The Challenge:** Standard whole-program analysis is too slow for Continuous Integration (CI). They need a tool that runs at **code-review time** (minutes, not hours) and focuses on *actionable* reports over theoretical perfection.
- **Research Gap:** Existing tools either require the whole program or prioritize soundness (proving absence of deadlocks) at the cost of high false-positive rates, which developers ignore.

2. Theoretical Framework: Critical Pairs (Approx. 1.5 Minutes)

Goal: Explain the "Core Contribution" — the mathematical model.

- **Abstract Language:** The paper models Android Java as an abstract language with *balanced re-entrant locks* (like `synchronized` blocks) and non-deterministic control.
- **Critical Pairs:** The central innovation is the concept of a **Critical Pair**.
- **Definition:** For a thread T, a critical pair is a tuple (X, l), meaning: "There exists an execution where the thread attempts to acquire lock l while *already holding* the set of locks X".
- **The Deadlock Theorem:** The authors prove that a deadlock occurs if and only if two threads have conflicting critical pairs:

  $$(X_1, l_1) \in \mathrm{Crit}(C_1) \quad \text{and} \quad (X_2, l_2) \in \mathrm{Crit}(C_2)$$

  Such that each thread holds the lock the other needs ($l_1 \in X_2$ and $l_2 \in X_1$) and their held locks do not overlap ($X_1 \cap X_2 = \emptyset$).
- **Complexity:** This problem is proven to be decidable and in **NP**.

3. Implementation: Compositionality (Approx. 1.5 Minutes)

Goal: Explain how the theory becomes a tool that scales.

- **Abstract Interpretation:** The tool calculates critical pairs using abstract interpretation. It computes a summary for each method.
- **Compositionality is Key:** Because the analysis is compositional, they do not need to re-analyze the whole app. When a developer submits a code change, the tool only analyzes the changed methods and their dependencies.
- **Handling Android Specifics:**
  - They map Java `synchronized` blocks to the balanced locks in the abstract language.
  - They use heuristics for thread identity (e.g., `@UiThread`, `@WorkerThread`) to reduce false positives.
- **Tooling:** The implementation is called `Starvation`, integrated into the **INFER** static analysis framework.

4. Results & Evaluation (Approx. 0.5 Minutes)

Goal: Prove it actually works in the real world.

- **Deployment:** Deployed on all Android commits at Facebook for over 2 years.
- **Performance:** Fast analysis times—median of 90 seconds per commit.
- **Impact:** Detected over 500 deadlock reports. Crucially, developers fixed $\approx 54\%$ of these reports, indicating high trust and "actionability".

5. Conclusion & Critique (Approx. 0.5 Minutes)

Goal: Summarize and show critical thinking.

- **Pros:** Successfully bridges the gap between theoretical guarantees (decidability) and industrial scale. The use of summaries makes it maintainable.
- **Limitations/Cons:**
  - It relies heavily on "balanced locking" (structured locking); unstructured locking breaks the model.
  - Compositionality can lead to false negatives if a deadlock spans across parts of the call graph not currently being analyzed.