2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678572

A Compositional Deadlock Detector for Android Java

James Brotherston*, Paul Brunet*, Nikos Gorogiannis† and Max Kanovich*
*Dept. of Computer Science, University College London, UK
†Facebook, UK
Email: J.Brotherston@ucl.ac.uk, nikos.gorogiannis@gmail.com, Paul@Brunet-Zamansky.fr, M.Kanovich@ucl.ac.uk

***Abstract*—We develop a static deadlock analysis for commercial Android Java applications, of sizes in the tens of millions of LoC, under active development at Facebook. The analysis runs primarily at code-review time, on only the modified code and its dependents; we aim at reporting to developers in under 15 minutes.**

To detect deadlocks in this setting, we first model the real language as an abstract language with balanced re-entrant locks, nondeterministic iteration and branching, and non-recursive procedure calls. We show that the existence of a deadlock in this abstract language is equivalent to a certain condition over the sets of *critical pairs* of each program thread; these record, for all possible executions of the thread, which locks are currently held at the point when a fresh lock is acquired. Since the critical pairs of any program thread is finite and computable, the deadlock detection problem for our language is decidable, and in NP.

We then leverage these results to develop an open-source implementation of our analysis adapted to deal with real Java code. The core of the implementation is an algorithm which computes critical pairs in a compositional, abstract interpretation style, running in quasi-exponential time. Our analyser is built in the INFER verification framework and has been in industrial deployment for over two years; it has seen over two hundred fixed deadlock reports with a report fix rate of $\sim$54%.

***Index Terms*—deadlocks, concurrency, program analysis**

I. INTRODUCTION

The avoidance and detection of *deadlocks* in a system is one of the most fundamental problems in concurrency. Deadlocking is classically exemplified by Dijkstra's "Five Dining Philosophers" [10]: Five philosophers sit around a table, with a fork between each pair of philosophers and a bowl of "a very difficult kind of spaghetti" in the centre, so that each philosopher requires both their left and right forks in order to eat. Without any communication between the philosophers, they will generally enter a deadlocked situation in which it is impossible for any of them to eat (for example if each of them immediately takes the fork to their left). More generally, in a concurrent program, a deadlock describes a situation in which, for some subset of that program's threads, it is impossible that any thread can eventually execute its next command.

In this paper, we attack the problem of detecting deadlocks in Android Java applications under continuous development at Facebook. These applications are typically in the tens of millions of LoC, and undergo thousands of revisions per day. Our main aim is to assist the developers in finding bugs introduced by their code revisions, and thus there are two principal desiderata of our deadlock analysis. The first is that it must return reports to developers relatively quickly, the target being under 15 minutes. It is impossible to analyse whole, large programs in their entirety within such a timeframe, which means that we need a method that works *compositionally*, enabling us to focus only on the changed files and their dependents. The second is that reports from our analysis must actually be *useful* to developers, meaning that we focus on minimising the number of false positive reports, as opposed to providing cast-iron guarantees of deadlock absence.

Our first step is to model the real programming language as an abstract programming language in which we can decide the presence of deadlocks within a reasonable complexity bound. The abstract programming language we use is based on scoped (a.k.a. "nested") re-entrant locks, nondeterministic iteration and branching, and nonrecursive procedure calls; it can be seen as an overapproximate model of Java, with all information about variable and memory assignment abstracted away.

We show that the existence of a deadlock in our abstract programs can be precisely characterised as a condition on the *critical pairs* of their (sequential) threads. Roughly speaking, 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. For the case of two threads, we establish that $C_1 \mid\mid C_2$ deadlocks if and only if there are critical pairs (X_1, ℓ_1) and (X_2, ℓ_2) of C_1 and C_2 respectively such that $\ell_1 \in X_2$ and $\ell_2 \in X_1$, with $X_1 \cap X_2 = \emptyset$; this condition can be generalised to the case of arbitrarily many threads (cf. Theorem 4.4). Similar to other deadlock conditions in the literature for communicating pushdown automata [24], [25], its correctness is crucially dependent on the fact that locking is *balanced* in our language, in that any thread must release locks in the reverse of the order in which they are acquired, i.e. "last in, first out". This is true of real programming languages whenever scoped-locking constructs are used, such as Java's `synchronized` keyword or C++'s `std::lock_guard`. Since the set of critical pairs of any thread in our language is in fact finite and computable, the existence of deadlocks in our abstract programs becomes decidable, and in NP.

Example 1.1. Consider a two-threaded program $C_1 \mid\mid C_2$, where C_1 and C_2 are sequential programs acquiring locks

DOI 10.1109/ASE51524.2021.00088

($\texttt{acq}(-)$) and releasing them ($\texttt{rel}(-)$) in reverse order:

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

C_1 has two critical pairs, $(\emptyset, x)$ and $(\{x\}, y)$, and similarly C_2 has two critical pairs $(\emptyset, y)$ and $(\{y\}, x)$. By taking $(X_1, \ell_1) = (\{x\}, y)$ and $(X_2, \ell_2) = (\{y\}, x)$, we can see that the condition above is met, and indeed $C_1 \mid\mid C_2$ deadlocks, because there is an execution in which, simultaneously, C_1 holds x while waiting for y, and C_2 holds y while waiting for x. Now consider the modified program $C_1' \mid\mid C_2'$, where $C_1' = \texttt{acq}(z); C_1; \texttt{rel}(z)$ and $C_2' = \texttt{acq}(z); C_2; \texttt{rel}(z)$. C_1' now has three critical pairs $(\emptyset, z)$, $(\{z\}, x)$ and $(\{z, x\}, y)$, and C_2' has critical pairs $(\emptyset, z)$, $(\{z\}, y)$ and $(\{z, y\}, x)$. In this case, the condition above is *not* met, and indeed $C_1' \mid\mid C_2'$ does not deadlock, because z acts as a "guard lock" preventing x and y from being accessed by C_1' and C_2' simultaneously.

We then leverage these theoretical results to design an automatic deadlock analyser geared specifically towards code changes in Android Java applications (in Section VI-B we outline the Android-specific features of the analysis). The core of our implementation is a context-insensitive program analysis that computes critical pairs in abstract interpretation style, running in quasi-exponential time in the syntactic size of the program. Crucially, this analysis is *compositional* in that the critical pairs of a procedure call depend only on the current state of the caller and the critical pairs of the procedure itself, which can be computed in advance. This means that, when analysing a code revision, we do not need to re-analyse the unchanged procedures in the program (i.e., the overwhelming majority). These properties enable our analysis to detect deadlocks in programs ranging in the tens of millions of LoC: two orders of magnitude larger than the largest programs handled by state-of-the-art static deadlock detectors.

We provide an open-source implementation of our analyser — named `starvation` and included as part of the INFER static analysis framework [1] — and describe its deployment and impact at Facebook, where it has analysed many hundreds of thousands of code revisions and seen over two hundred deadlock reports fixed in the last two years.

Important note. Contrary to what many readers may well expect, we do not undertake an experimental comparison of our tool with others in the literature, due to two major divergences between our tool and others. The first and most important point of divergence is that most — if not all — of the tools in the literature require the *whole program* in order to run. Since the programs we target are so large, these analysers might take hours or days to return a result, or simply run out of memory. In contrast, our tool can run relatively quickly on these programs because it only analyses *code changes*, relying on compositionality of the analysis with respect to the unchanged portions of the program. The second, less important divergence is that, since our priority is to avoid wasting developers' time, we prioritise actionability of the tool's deadlock reports, which means that we generally prefer to admit false negative than false positive deadlock reports. In contrast, most tools that we know of do just the opposite.

The remainder of this paper is structured as follows. Section II introduces our abstract concurrent programs. In Section III we develop the key connections between sequential program executions and their *traces* (of lock acquisitions and releases). Then, in Section IV, we establish the soundness and completeness of our deadlock condition based on critical pairs. In Section V we show that the critical pairs of any sequential program are computable, and so establish upper complexity bounds on the deadlock problem. Section VI describes our implementation of the deadlock analysis for Android Java, and its deployment impact at Facebook. Section VII surveys the related work, and Section VIII concludes.

For space reasons, the proofs of our theoretical results are only sketched in the present paper. However, the entire development has also been fully formalised in the Coq proof assistant, in roughly 8.7K lines of code.

II. PROGRAM SYNTAX AND SEMANTICS

Syntax. We let Locks be a finite set of *global lock names* and Procs a set of *procedure names*. We define *statements* C as follows, where ℓ ranges over Locks and p over Procs:

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

We assume there is a function $\mathsf{body}() : \mathsf{Procs} \to \mathsf{Stmt}$ that sends every procedure name to a statement, its body. A function computing the callees of a statement $\mathsf{callees}(\cdot) : \mathsf{Stmt} \to \mathcal{P}(\mathsf{Procs})$ can be easily defined. We forbid recursion in statements; i.e., for all $p \in \mathsf{Procs}$, $p \notin \mathsf{callees}(\mathsf{body}(p))$.

A statement is called *balanced* if it is generated by the following grammar, which ensures that $\texttt{acq}(\ell)$ and $\texttt{rel}(\ell)$ only appear in balanced pairs:

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

Moreover, balanced statements must call only balanced procedures: if C is balanced and $p \in \mathsf{callees}(C)$, then $\mathsf{body}(p)$ must be balanced as well. We note that our balanced statements are similar to those produced by compiling scope-based constructs like Java's `synchronized` keyword, or C++'s `std::lock_guard`.

We will frequently need to reason by structural induction over (balanced) statements. To account for procedure calls in such proofs, we employ an extended notion of "substructure" for statements, given as the reflexive-transitive closure of the following condition: any sub-statement of C (according to the grammar above) is a substructure of C, and $\mathsf{body}(p)$ is a substructure of $p()$. Since our procedures are non-recursive, this ordering is still well-founded.

Finally, a *parallel program* is an n-tuple of balanced statements written $C_1 \mid\mid \ldots \mid\mid C_n$.

Semantics. As our programs employ only lock guards and non-deterministic control, our program states record only

information about locks. We treat locks as *re-entrant* in that a thread already holding a lock can re-acquire it without deadlock.

A *lock state* is a function $L : \mathsf{Locks} \to \mathbb{N}$, recording how many times each lock has been acquired. We use the notation $\lfloor L \rfloor$ for $\{\ell \in \mathsf{Locks} \mid L(\ell) > 0\}$. If L_1 and L_2 are lock states then we write $L_1 \# L_2$ to mean that $\lfloor L_1 \rfloor \cap \lfloor L_2 \rfloor = \emptyset$. We write $\varnothing$ for the lock state sending all locks to 0. We write $L[\ell{+}{+}]$ and $L[\ell{-}{-}]$ for the lock states defined as L, except that $L[\ell{+}{+}](\ell) = L(\ell) + 1$ and $L[\ell{-}{-}](\ell) = L(\ell) - 1$.

A *configuration* is a pair $\langle C, L\rangle$, where C is a statement and L is a lock state. A *concurrent configuration* is then a pair of the form $\langle C_1 \mid\mid \ldots \mid\mid C_n, (L_1, \ldots, L_n)\rangle$, where $C_1 \mid\mid \ldots \mid\mid C_n$ is a parallel program and $L_1, \ldots, L_n$ are lock states. We may also write concurrent configurations as $\langle C_1, L_1\rangle \mid\mid \ldots \mid\mid \langle C_n, L_n\rangle$, or, using a "$\Sigma$-like" notation, as $||_{1\le i\le n} \langle C_i, L_i\rangle$. We write $\langle C_i, L_i\rangle \# \langle C_j, L_j\rangle$ to mean that $L_i \# L_j$ and, if X is a set of lock states, $L \# X$ to mean that $L \# L'$ for all $L' \in X$.

In Figure 1 we define the operational semantics of our programs by giving the small-step relations for statements on ordinary configurations, $\to$, and for parallel programs on concurrent configurations, $\rightsquigarrow$. An *execution* (of statement C) is then as usual a possibly infinite sequence of configurations $\pi = (\gamma_i)_{i\ge 0}$ (with $\gamma_0 = \langle C, \_\rangle$) such that $\gamma_i \to \gamma_{i+1}$ for all $i \ge 0$. A *concurrent execution* is defined analogously to an execution, by substituting concurrent configurations for configurations and $\rightsquigarrow$ for $\to$.

We often represent executions $(\gamma_i)_{i\ge 0}$ as $\gamma_0 \to^* \gamma_n$, where $\to^*$ is the reflexive-transitive closure of $\to$, and similarly using $\rightsquigarrow^*$ for concurrent executions.

Remark 2.1. *For any concurrent execution $\gamma_1 \mid\mid \ldots \mid\mid \gamma_n \rightsquigarrow^* \gamma'_1 \mid\mid \ldots \mid\mid \gamma'_n$ there exist standard executions $\gamma_i \to^* \gamma'_i$ for each $1 \le i \le n$. Furthermore, if $\gamma_i \# \gamma_j$, then $\gamma'_i \# \gamma'_j$; i.e., no two threads can acquire the same lock simultaneously.*

We define deadlock of a program to mean that at least two of its threads are deadlocked. For this, we introduce a notation for projecting a concurrent configuration onto a subset of its threads. If $\sigma = \langle C_1 \mid\mid \ldots \mid\mid C_n, (L_1, \ldots, L_n)\rangle$ is a concurrent configuration and $I = \{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$ is a set of "thread indices", we write σ_I to mean the concurrent configuration $\langle C_{i_1}, L_{i_1}\rangle \mid\mid \ldots\ldots \mid\mid \langle C_{i_m}, L_{i_m}\rangle$.

Definition 2.2 (Deadlock)**.** *A concurrent configuration, say $\sigma = \langle C_1 \mid\mid \ldots \mid\mid C_n, (L_1, \ldots, L_n)\rangle$, is* deadlocked *if there is a sequential transition $\langle C_i, L_i\rangle \to \langle C'_i, L'_i\rangle$ for each thread (i.e. for all $1 \le i \le n$) but there is no concurrent transition $\sigma \rightsquigarrow \sigma'$.*

A parallel program $C_1 \mid\mid \ldots \mid\mid C_n$ is said to deadlock *if we have that $\langle C_1 \mid\mid \ldots \mid\mid C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^* \sigma$ and σ_I is deadlocked for some $I \subseteq \{1, \ldots, n\}$.*

An immediate consequence is that if $C_1 \mid\mid \ldots \mid\mid C_m$ deadlocks and $m \le n$, then $C_1 \mid\mid \ldots \mid\mid C_n$ also deadlocks.

Proposition 2.3. *Let $\sigma = \langle C_1 \mid\mid \ldots \mid\mid C_n, (L_1, \ldots, L_n)\rangle$ be a concurrent configuration such that $L_i \# L_j$ for all $i \ne j$. The configuration σ is deadlocked iff there are statements $D_1, \ldots, D_n$ and locks $\ell_1, \ldots, \ell_n$ such that, for all $1 \le i \le n$*

$$\langle C_i, L_i\rangle \to \langle D_i, L_i[\ell_i{+}{+}]\rangle \quad \text{and} \quad \ell_i \in \textstyle\bigcup_{j\ne i} \lfloor L_j \rfloor \ .$$

Proof. Follows from the operational semantics. □

III. Executions and traces

In this section, we develop a key technical idea: any execution of a statement (in an arbitrary lock state) can be viewed simply as a sequence of lock acquisitions ℓ and releases $\overline{\ell}$, which we call the execution's *trace*. Thus, for example, the two possible executions of the statement

$$\begin{aligned}&\mathtt{acq}(\ell); \mathtt{if}(*)\ \mathtt{then}\ (\mathtt{acq}(j); \mathtt{skip}; \mathtt{rel}(j))\\ &\qquad\qquad \mathtt{else}\ (\mathtt{acq}(k); \mathtt{skip}; \mathtt{rel}(k)); \mathtt{rel}(\ell)\end{aligned}$$

have respective traces $\ell\, j\, \overline{j}\, \overline{\ell}$ and $\ell\, k\, \overline{k}\, \overline{\ell}$, depending on which branch of the `if` statement is chosen.

Traces preserve the essential information about executions, in that the effect of an execution on any given initial lock state can be computed from its trace. Moreover, executions of *balanced* statements have traces that are essentially well-parenthesized strings of lock acquisitions and releases; in fact they are *Dyck words* in formal language theory [21].

Definition 3.1. *The* lock alphabet Σ *is defined as the union of two disjoint copies of* Locks*:*

$$\Sigma := \{\ell \mid \ell \in \mathsf{Locks}\} \cup \{\overline{\ell} \mid \overline{\ell} \in \mathsf{Locks}\} \ .$$

A quasi-lock state *is a function in $\mathsf{Locks} \to \mathbb{Z}$. We lift the notations $[\ell{+}{+}]$ and $[\ell{-}{-}]$ from lock states to quasi-lock states in the obvious way, and write $+$ on quasi-lock states to denote the pointwise sum of functions, i.e. $(f+g)(x) = f(x) + g(x)$. We define the function $\langle\cdot\rangle$ from Σ-words to quasi-lock states inductively, as follows:*

$$\langle\varepsilon\rangle := \varnothing \qquad \langle u\,\ell\rangle := \langle u\rangle[\ell{+}{+}] \qquad \langle u\,\overline{\ell}\rangle := \langle u\rangle[\ell{-}{-}] \ .$$

Lemma 3.2. *For all $u, v \in \Sigma^\star$ we have $\langle u\, v\rangle = \langle u\rangle + \langle v\rangle$.*

Proof. By structural induction on v. □

In any sequential execution step $\langle C, L\rangle \to \langle C', L'\rangle$ we have $L' = L$, or $L' = L[\ell{+}{+}]$ or $L' = L[\ell{-}{-}]$ for some lock ℓ. This justifies the following definition.

Definition 3.3. *Given a transition $\langle C, L\rangle \to \langle C', L'\rangle$, we define its* trace *$u \in \Sigma \cup \{\varepsilon\}$ as follows:*

$$u = \begin{cases} \varepsilon & \text{if } L' = L \\ \ell & \text{if } L' = L[\ell{+}{+}] \\ \overline{\ell} & \text{if } L' = L[\ell{-}{-}]. \end{cases}$$

The trace of an execution is then defined as the concatenation of the traces of its individual transitions. We often write transitions and executions with their trace above the arrow, as in $\langle C, L\rangle \xrightarrow{u} \langle C', L'\rangle$ and $\langle C, L\rangle \xrightarrow{u}{}^* \langle C', L'\rangle$.

$$\langle \texttt{skip}; C, L\rangle \to \langle C, L\rangle \quad (\texttt{skip}) \qquad \langle \texttt{if}(*)\ \texttt{then}\ C_a\ \texttt{else}\ C_b, L\rangle \to \langle C_a, L\rangle \quad (\texttt{if1})$$
$$\langle p(), L\rangle \to \langle \texttt{body}(p), L\rangle \quad (\texttt{proc}) \qquad \langle \texttt{if}(*)\ \texttt{then}\ C_a\ \texttt{else}\ C_b, L\rangle \to \langle C_b, L\rangle \quad (\texttt{if2})$$
$$\langle \texttt{acq}(\ell), L\rangle \to \langle \texttt{skip}, L[\ell{+}{+}]\rangle \quad (\texttt{acq}) \qquad \langle \texttt{while}(*)\ \texttt{do}\ C, L\rangle \to \langle \texttt{skip}, L\rangle \quad (\texttt{while1})$$
$$\langle \texttt{rel}(\ell), L\rangle \to \langle \texttt{skip}, L[\ell{-}{-}]\rangle\ (L(\ell) > 0) \quad (\texttt{rel}) \qquad \langle \texttt{while}(*)\ \texttt{do}\ C, L\rangle \to \langle C; \texttt{while}(*)\ \texttt{do}\ C, L\rangle \quad (\texttt{while2})$$

$$\frac{\langle C_1, L\rangle \to \langle C_1', L'\rangle}{\langle C_1; C_2, L\rangle \to \langle C_1'; C_2, L'\rangle}\ (\texttt{seq}) \qquad \frac{\langle C_i, L_i\rangle \to \langle C_i', L_i'\rangle \qquad L_i' \,\#\, \{L_j \mid j \neq i\}}{\langle C_1 \,||\ldots||\, C_n, (L_1, \ldots, L_n)\rangle \rightsquigarrow \langle C_1 \,||\ldots||\, C_i' \,||\ldots||\, C_n, (L_1, \ldots, L_i', \ldots, L_n)\rangle}\ (\texttt{par}\ i)$$

Fig. 1. Small-step semantics for statements ($\to$) and parallel programs ($\rightsquigarrow$).

Proposition 3.4. *For any execution $\langle C_0, L_0\rangle \xrightarrow{u}{}^{*} \langle C_n, L_n\rangle$ and statement C, we can obtain an execution $\langle C_0; C, L_0\rangle \xrightarrow{u}{}^{*} \langle C_n; C, L_n\rangle$ with the same trace.*

Proof. By inductively applying the semantic rule (seq). □

We define the language of a statement, roughly speaking, as the set of traces of its possible executions. Subsequent technical results will make this correspondence precise.

Definition 3.5. *The* language $\mathcal{L}(C)$ *of a statement C is defined inductively as follows:*

$$\begin{aligned} \mathcal{L}(\texttt{skip}) &\coloneqq \{\varepsilon\} & \mathcal{L}(p()) &\coloneqq \mathcal{L}(\texttt{body}(p)) \\ \mathcal{L}(\texttt{acq}(\ell)) &\coloneqq \{\ell\} & \mathcal{L}(\texttt{rel}(\ell)) &\coloneqq \{\overline{\ell}\} \\ & & \mathcal{L}(C_1; C_2) &\coloneqq \mathcal{L}(C_1) \cdot \mathcal{L}(C_2) \\ & & \mathcal{L}(\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2) &\coloneqq \mathcal{L}(C_1) \cup \mathcal{L}(C_2) \\ & & \mathcal{L}(\texttt{while}(*)\ \texttt{do}\ C) &\coloneqq \mathcal{L}(C)^{\star} \end{aligned}$$

Remark 3.6. *$\mathcal{L}(C)$ is in fact a* regular language *over Σ.*

Our next lemma establishes that the effect of an execution is essentially determined by its trace.

Lemma 3.7. *For any execution $\pi : \langle C, L\rangle \xrightarrow{u}{}^{*} \langle C', L'\rangle$ we have $L' = L + \langle u\rangle$ and $u \cdot \mathcal{L}(C') \subseteq \mathcal{L}(C)$.*

Proof. We first prove the lemma for a single $\to$-step by rule induction on the semantics. The general result then follows by reflexive-transitive induction on $\to^{*}$. □

Next, we recall the notion of *Dyck words* over our lock alphabet Σ — essentially the well-parenthesized words of opening and closing "parentheses" ℓ and $\overline{\ell}$ — and relate them to executions of *balanced* statements.

Definition 3.8. *The language $\mathcal{D}$ of* Dyck words *over Σ is generated by the following grammar:*

$$D := \varepsilon \mid D\,D \mid \ell\,D\,\overline{\ell}.$$

The following lemmas establish basic properties of our Dyck words and their relation to our mapping $\langle\cdot\rangle$, and are all proven by structural induction (using Lemma 3.2).

Lemma 3.9. *If C is a balanced statement, $\mathcal{L}(C) \subseteq \mathcal{D}$.*

Lemma 3.10. *For any $u\,\overline{\ell}\,v \in \mathcal{D}$, there exist words $u_1 \in \Sigma^{\star}$ and $u_2 \in \mathcal{D}$ such that $u = u_1\,\ell\,u_2$.*

Lemma 3.11. *For any $u \in \mathcal{D}$ we have $\langle u\rangle = \varnothing$.*

Lemma 3.12. *For any $u\,v \in \mathcal{D}$ we have $\langle u\rangle \in \mathsf{Locks} \to \mathbb{N}$.*

We now establish an analogue of Lemma 3.10 for executions of balanced statements, which will be essential later for "disentangling" concurrent executions (see Lemma 4.3).

Lemma 3.13. *Let C be a balanced statement. For any execution of the form*

$$\langle C, \varnothing\rangle = \langle C_0, L_0\rangle \to^{*} \langle C_n, L_n\rangle \to \langle C_{n+1}, L_n[\ell{-}{-}]\rangle\ ,$$

there exists $j < n$ such that $L_j = L_n[\ell{-}{-}]$ and $L_{j+1} = L_n$.

Proof. Follows from Lemmas 3.2, 3.7, 3.9, 3.10 and 3.11. □

The final main result in this section is a kind of converse to Lemma 3.7, albeit for balanced statements only.

Lemma 3.14. *Let C be a balanced statement, and let $u\,v \in \mathcal{L}(C)$. For any lock state L, there is a statement D and an execution $\pi : \langle C, L\rangle \xrightarrow{u}{}^{*} \langle D, L + \langle u\rangle\rangle$ such that $v \in \mathcal{L}(D)$. If $v = \varepsilon$, then this statement also holds when $D = \texttt{skip}$.*

Proof. By structural induction on C, making use of earlier results. The main idea is to analyse the trace $u\,v \in \mathcal{L}(C)$ in order to inductively build an execution of C with trace u. □

Note that Lemma 3.14 does not hold for non-balanced statements. E.g., $\overline{\ell} \in \mathcal{L}(\texttt{rel}(\ell))$, but there are no executions of $\langle \texttt{rel}(\ell), \varnothing\rangle$.

Corollary 3.15. *For any balanced statement C, we have*

$$\mathcal{L}(C) = \{u \mid \langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle \texttt{skip}, \varnothing\rangle\}\ .$$

Proof. The $\supseteq$ inclusion follows from Lemma 3.7 and the $\subseteq$ inclusion from Lemma 3.14 (with $v = \varepsilon$). □

Example 3.16. Let C be the statement from the beginning of this section:

$$\begin{aligned} &\texttt{acq}(\ell); \texttt{if}(*)\ \texttt{then}\ (\texttt{acq}(j); \texttt{skip}; \texttt{rel}(j)) \\ &\qquad\qquad \texttt{else}\ (\texttt{acq}(k); \texttt{skip}; \texttt{rel}(k)); \texttt{rel}(\ell) \end{aligned}$$

From Definition 3.5 we have $\mathcal{L}(C) = \{\ell\,j\,\overline{j}\,\overline{\ell}, \ell\,k\,\overline{k}\,\overline{\ell}\}$, and there are exactly two possible executions of C from the empty lock state $\varnothing$ (we omit the intermediate commands and the skip steps):

$$\langle C, \varnothing\rangle \xrightarrow{\ell} \langle \_, \{\ell\}\rangle \xrightarrow{\varepsilon} \langle \_, \{\ell\}\rangle \xrightarrow{j} \langle \_, \{\ell, j\}\rangle \xrightarrow{\overline{j}} \langle \_, \{\ell\}\rangle \xrightarrow{\overline{\ell}} \langle \texttt{skip}, \varnothing\rangle$$
$$\langle C, \varnothing\rangle \xrightarrow{\ell} \langle \_, \{\ell\}\rangle \xrightarrow{\varepsilon} \langle \_, \{\ell\}\rangle \xrightarrow{k} \langle \_, \{\ell, k\}\rangle \xrightarrow{\overline{k}} \langle \_, \{\ell\}\rangle \xrightarrow{\overline{\ell}} \langle \texttt{skip}, \varnothing\rangle$$

The first execution has trace $\ell\, j\, \overline{j}\, \overline{\ell}$, and the second has trace $\ell\, k\, \overline{k}\, \overline{\ell}$. Thus indeed $\mathcal{L}(C) = \{u \mid \langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle \mathtt{skip}, \varnothing\rangle\}$.

Remark 3.17. *Statements can be viewed as string accepters on Σ-words, where C accepts u iff $\langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle \mathtt{skip}, \varnothing\rangle$. If C is balanced then, by Corollary 3.15, it accepts exactly $\mathcal{L}(C)$. Since $\mathcal{L}(C)$ is a regular language, this means that balanced statements can be viewed as* finite automata*.*

IV. Characterisation of deadlock existence

Here, we obtain our main theoretical result: the existence of a deadlock in a parallel program amounts to the existence of a (certain kind of) conflict between individual "summaries" of its threads, called their (sets of) *critical pairs*. Roughly speaking, a critical pair of a statement C is a pair (X, ℓ) such that some execution of C acquires the lock ℓ while holding the set of locks X (which cannot already include ℓ). Our main correctness result is stated as Theorem 4.4.

Definition 4.1. *The set* $\mathbf{Crit}(C)$ *of* critical pairs *of a statement C is defined as:*

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

The reason for our language-based definition of $\mathbf{Crit}(C)$, as opposed to an execution-based one, is that it turns out to be easy to compute (see Section V). The following lemma gives an equivalent formulation in terms of executions.

Lemma 4.2. *Let C be a balanced statement. We have that $(X, \ell) \in \mathbf{Crit}(C)$ iff there exist statements C', C'' and a lock state L such that $\langle C, \varnothing\rangle \to^{*} \langle C', L\rangle \to \langle C'', L[\ell{+}{+}]\rangle$, with $X = \lfloor L\rfloor$ and $\ell \notin X$.*

Proof. The ($\Leftarrow$) direction follows from Lemma 3.7, and the ($\Rightarrow$) direction from Lemma 3.14. □

Our final and most crucial lemma shows essentially that, for balanced statements, considerations of reachability on the concurrent transition relation $\rightsquigarrow$ can be reduced to reachability on the sequential relation $\to$.

Lemma 4.3. *Suppose $C_1 \,||\ldots||\, C_n$ does not deadlock. Then $\langle C_1 \,||\ldots||\, C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^{*} \gamma_1 \,||\ldots||\, \gamma_n$ iff, for each $1 \le i \le n$, we have $\langle C_i, \varnothing\rangle \to^{*} \gamma_i$ with $\gamma_i \# \{\gamma_j \mid j \ne i\}$.*

Proof. (Sketch) The ($\Rightarrow$) direction is immediate from Remark 2.1. For the ($\Leftarrow$) direction, we write $\gamma_{i,j} = \langle C_{i,j}, L_{i,j}\rangle$ for the jth configuration in the execution $\pi_i : \langle C_i, \varnothing\rangle \to^{*} \gamma_i$. An arbitrary concurrent configuration given by an interleaving from these n executions is given by $||_{1\le i\le n}\ \gamma_{i,j_i}$. We call such a configuration *compatible* when $\gamma_{i,j_i} \# \gamma_{k,j_k}$ for all $k \ne i$, and *reachable* when $\langle C_1 \,||\ldots||\, C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^{*} ||_{1\le i\le n}\ \gamma_{i,j_i}$. It then suffices to show that if $||_{1\le i\le n}\ \gamma_{i,j_i}$ is compatible then it is also reachable.

We proceed by induction on $J = \Sigma_{1\le i\le n}\ j_i$. The case $J = 0$ is trivial. Otherwise, $J > 0$ and we consider the compatibility of the "preceding" configurations $\gamma_{k,j_k-1} \,||\, ||_{1\le i\le n, i\ne k}\ \gamma_{i,j_i}$, where $1 \le k \le n$. We then consider two main subcases.

The first subcase is that some $\gamma_{k,j_k-1} \,||\, ||_{1\le i\le n, i\ne k}\ \gamma_{i,j_i}$ is compatible. In that case, using the induction hypothesis and the assumption that $||_{1\le i\le n}\ \gamma_{i,j_i}$ is compatible, we have that $||_{1\le i\le n}\ \gamma_{i,j_i}$ becomes reachable by applying $(\mathtt{par}\ i)$.

The remaining subcase is then that *no* configuration of the form $\gamma_{k,j_k-1} \,||\, ||_{1\le i\le n, i\ne k}\ \gamma_{i,j_i}$ is compatible; we can assume without loss of generality that these configurations are defined for the first $m \ge 2$ threads and undefined otherwise. Now, letting $1 \le k \le m$, we must have $L_{k,j_k} = L_{k,j_k-1}[\ell_k{-}{-}]$ for some lock ℓ_k, by the subcase assumption. By Lemma 3.10, we can find $h_k < j_k$ such that $L_{k,h_k} = L_{k,j_k}$ and $L_{k,h_k+1} = L_{k,j_k-1}$. It follows that $||_{1\le k\le m}\ \gamma_{k,h_k} \,||\, ||_{m+1\le i\le n}\ \gamma_{i,j_i}$ is also compatible. Thus, by the induction hypothesis that $||_{1\le k\le m}\ \gamma_{k,h_k} \,||\, ||_{m+1\le i\le n}\ \gamma_{i,j_i}$ is reachable. To conclude the proof, we show that $||_{1\le k\le m}\ \gamma_{k,h_k}$ is deadlocked, and thus $C_1 \,||\ldots||\, C_n$ deadlocks, a contradiction.

If not, then $||_{1\le k\le m}\ \gamma_{k,h_k} \rightsquigarrow \sigma$ for some σ via an application of the rule $(\mathtt{par}\ i)$. In that case, we can deduce from the lock state information that C_{i,h_i} must begin with the command $\mathtt{acq}(\ell_i)$ and so $L_{i,h_i+1} = L_{i,j_i-1}$. Thus $L_{i,j_i-1} \# \{L_{k,h_k} \mid k \ne i\}$. Since $L_{k,h_k} = L_{k,j_k}$ for each k, this means that $\gamma_{i,j_i-1} \,||\, ||_{1\le k\le m, k\ne i}\ \gamma_{i,j_i}$ is compatible, which contradicts the subcase assumption and so completes the proof. □

We are now finally in a position to characterise deadlock existence as a "conflict condition" on the critical pairs of its sequential components.

Theorem 4.4 (Deadlock characterisation)**.** *A parallel program $C_1 \,||\ldots||\, C_n$ deadlocks iff, for some $I \subseteq \{1, \ldots, n\}$ with cardinality ≥ 2, there are critical pairs (X_i, ℓ_i) for each $i \in I$ such that $X_i \cap \bigcup_{j\ne i} X_j = \emptyset$ and $\ell_i \in \bigcup_{j\ne i}\ X_j$.*

Proof. **Case** ($\Rightarrow$)**:** Suppose $C_1 \,||\ldots||\, C_n$ deadlocks, meaning that $\langle C_1 \,||\ldots||\, C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^{*} \sigma$ and σ_I is deadlocked, for some index set I; without loss of generality, we assume that $I = \{1, \ldots, m\}$, and thus $\sigma_I = \langle D_1 \,||\ldots||\, D_m, (L_1, \ldots, L_m)\rangle$. By Remark 2.1, we have for each $1 \le i \le m$ a sequential execution $\langle C_i, \varnothing\rangle \to^{*} \langle D_i, L_i\rangle$ with $L_i \# \{L_j \mid j \ne i\}$.

Since σ_I is deadlocked and $L_i \# L_j$ for all $i \ne j$, we have $\langle D_i, L_i\rangle \to \langle D_i', L_i[\ell_i{+}{+}]\rangle$ with $\ell_i \in \bigcup_{j\ne i} \lfloor L_j\rfloor$ for all $1 \le i \le m$, by Proposition 2.3. Thus we have executions $\langle C_i, \varnothing\rangle \to^{*} \langle D_i, L_i\rangle \to \langle D_i', L_i[\ell_i{+}{+}]\rangle$, with $\ell_i \notin \lfloor L_i\rfloor$, because $\ell_i \in \bigcup_{j\ne i} \lfloor L_j\rfloor$ and $L_i \# \{L_j \mid j \ne i\}$. By Lemma 4.2, we obtain $(\lfloor L_i\rfloor, \ell_i) \in \mathbf{Crit}(C_i)$. Taking $X_i = \lfloor L_i\rfloor$ for each i, all required conditions are satisfied.

Case ($\Leftarrow$)**:** We assume w.l.o.g. that $I = \{1, \ldots, m\}$, where $m \ge 2$. Let $1 \le i \le m$; using $(X_i, \ell_i) \in \mathbf{Crit}(C_i)$ we have by Lemma 4.2 that $\langle C_i, \varnothing\rangle \to^{*} \langle C_i', L_i\rangle \to \langle C_i'', L_i[\ell_i{+}{+}]\rangle$ and $X_i = \lfloor L_i\rfloor$ and $\ell_i \notin X_i$. Moreover, since $X_i \cap \bigcup_{j\ne i} X_j = \emptyset$, we have $L_i \# \{L_j \mid j \ne i\}$.

Now, suppose that $C_1 \,||\ldots||\, C_m$ does not deadlock. In that case Lemma 4.3 yields $\langle C_1 \,||\ldots||\, C_m, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^{*} \langle C_1' \,||\ldots||\, C_m', (L_1, \ldots, L_m)\rangle$, and hence the latter configuration cannot be deadlocked. But because $\langle C_i', L_i\rangle \to$

$\langle C_i'', L_i[\ell_i{+}{+}]\rangle$ and $\ell_i \in \bigcup_{j\neq i} \lfloor L_j \rfloor$ for each i (using the assumption that $\ell_i \in \bigcup_{j\neq i} X_j$), this configuration actually *is* deadlocked, by Proposition 2.3. Thus, $C_1 \mid\mid \ldots \mid\mid C_m$ deadlocks after all, a contradiction. □

The following example illustrates our deadlock condition.

Example 4.5. Define statements $C_1, \ldots, C_n$ as follows:

$$\begin{aligned}
C_1 &:= \texttt{acq}(\ell_2);\ \texttt{acq}(\ell_1);\ \texttt{skip};\texttt{rel}(\ell_1);\ \texttt{rel}(\ell_2);\\
C_2 &:= \texttt{acq}(\ell_3);\ \texttt{acq}(\ell_2);\ \texttt{skip};\texttt{rel}(\ell_2);\ \texttt{rel}(\ell_3);\\
&\vdots\\
C_{n-1} &:= \texttt{acq}(\ell_n);\ \texttt{acq}(\ell_{n-1});\ \texttt{skip};\texttt{rel}(\ell_{n-1});\ \texttt{rel}(\ell_n);\\
C_n &:= \texttt{acq}(\ell_1);\ \texttt{acq}(\ell_n);\ \texttt{skip};\texttt{rel}(\ell_n);\ \texttt{rel}(\ell_1);
\end{aligned}$$

We have $(\{\ell_{(i+1) \bmod n}\}, \ell_i) \in \mathbf{Crit}(C_i)$ for each $1 \leq i \leq n$. These critical pairs satisfy the deadlock condition of Theorem 4.4, and indeed $C_1 \mid\mid \ldots \mid\mid C_n$ deadlocks, by executing the first $\texttt{acq}(-)$ command in each thread. Conversely, any smaller subset of these threads, e.g. $C_1 \mid\mid \ldots \mid\mid C_{n-1}$, does not satisfy the deadlock condition, and indeed we can observe that $C_1 \mid\mid \ldots \mid\mid C_{n-1}$ does not deadlock.

V. Computing critical pairs

Having established that the existence of a deadlock in a parallel program reduces to a condition over the critical pairs of its threads (Theorem 4.4), our next order of business is to show that $\mathbf{Crit}(C)$ is in fact computable for any balanced statement C — something that is perhaps not immediately obvious from Definition 4.1. Here we establish a set of useful identities enabling us to compute $\mathbf{Crit}(C)$ inductively, with the consequence that the deadlock problem for our language is decidable and in NP (Theorem 5.5).

Proposition 5.1. *The following identities hold for all balanced statements C, C' and locks ℓ:*

$$\mathbf{Crit}(\texttt{skip}) = \emptyset \tag{C1}$$

$$\mathbf{Crit}(p()) = \mathbf{Crit}(\texttt{body}(p)) \tag{C2}$$

$$\mathbf{Crit}(\texttt{if}(*)\ \texttt{then}\ C\ \texttt{else}\ C') = \mathbf{Crit}(C) \cup \mathbf{Crit}(C') \tag{C3}$$

$$\mathbf{Crit}(C; C') = \mathbf{Crit}(C) \cup \mathbf{Crit}(C') \tag{C4}$$

$$\mathbf{Crit}(\texttt{while}(*)\ \texttt{do}\ C) = \mathbf{Crit}(C) \tag{C5}$$

$$\begin{aligned}\mathbf{Crit}(\texttt{acq}(\ell); C; \texttt{rel}(\ell)) &= \{(\emptyset, \ell)\} \cup\\ &\{(X \cup \{\ell\}, \ell') \mid (X, \ell') \in \mathbf{Crit}(C) \text{ and } \ell \neq \ell'\}\end{aligned} \tag{C6}$$

Proof. We show each identity directly from Definition 4.1, making use of auxiliary lemmas from section III. □

Example 5.2. We continue Example 3.16, where C is the statement:

$$\begin{aligned}&\texttt{acq}(\ell); \texttt{if}(*)\ \texttt{then}\ (\texttt{acq}(j); \texttt{skip}; \texttt{rel}(j))\\ &\qquad\qquad \texttt{else}\ (\texttt{acq}(k); \texttt{skip}; \texttt{rel}(k)); \texttt{rel}(\ell)\ .\end{aligned}$$

First, using equations (C1) and (C6), we get

$$\begin{aligned}\mathbf{Crit}(\texttt{acq}(j); \texttt{skip}; \texttt{rel}(j)) &= \{(\emptyset, j)\}\\ \text{and}\quad \mathbf{Crit}(\texttt{acq}(k); \texttt{skip}; \texttt{rel}(k)) &= \{(\emptyset, k)\}.\end{aligned}$$

Thus, writing $C = \texttt{acq}(\ell); C'; \texttt{rel}(\ell)$, equation (C3) gives us $\mathbf{Crit}(C') = \{(\emptyset, j), (\emptyset, k)\}$. Finally, applying (C6) once again and observing that $j, k \neq \ell$, we get

$$\begin{aligned}\mathbf{Crit}(C) &= \mathbf{Crit}(\texttt{acq}(\ell); C'; \texttt{rel}(\ell))\\ &= \{(\emptyset, \ell)\} \cup \{(\{\ell\}, j), (\{\ell\}, k)\}\\ &= \{(\emptyset, \ell), (\{\ell\}, j), (\{\ell\}, k)\}\ .\end{aligned}$$

Next we undertake a brief complexity analysis. We write $\#X$ for the cardinality of a finite set X.

Definition 5.3. *For any statement C, define its* size $\|C\|$ *by*

$$\|C\| \coloneqq |C| + \textstyle\sum_{p \in \mathsf{callees}(C)} |\texttt{body}(p)|\ ,$$

where $|C|$ is defined inductively as follows:

$$\begin{aligned}|\texttt{skip}| = |\texttt{acq}(\ell)| = |\texttt{rel}(\ell)| = |p()| &= 1\\ |\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2| = |C_1; C_2| &= |C_1| + |C_2|\\ |\texttt{while}(*)\ \texttt{do}\ C| &= |C|\end{aligned}$$

Proposition 5.4. *$\mathbf{Crit}(C)$ is finite and computable for any balanced statement C, with $\#\mathbf{Crit}(C) \leq \|C\|^{1+\#\mathsf{callees}(C)}$ and $\#X < \|C\|$ for all $(X, \ell) \in \mathbf{Crit}(C)$. If C does not contain any procedure calls, then $\#\mathbf{Crit}(C) \leq |C|$.*

Proof. By structural induction on C, using Prop. 5.1. □

To precisely state complexity bounds on the deadlock problem, we define the size of a parallel program as the sum of the sizes of its threads: $\|C_1 \mid\mid \ldots \mid\mid C_n\| = \sum_{1 \leq i \leq n} \|C_i\|$.

Theorem 5.5. *Whether a given parallel program deadlocks or not is decidable, and in* NP.

Proof. We just need to show how to check the deadlock condition of Theorem 4.4 in nondeterministic polynomial time. The NP procedure runs in three stages: (i) nondeterministically select an index set $I \subseteq \{1, \ldots, n\}$ of size ≥ 2; (ii) nondeterministically compute a critical pair (X_i, ℓ_i) for each $i \in I$, by recursing on the structure of C_i and using the equations (C1)–(C6) in Proposition 5.1; (iii) verify that $X_i \cap \bigcup_{j\neq i} X_j = \emptyset$ and $\ell_i \in \bigcup_{j\neq i} X_j$ for all $i, j \in I$. The last step can be done in polynomial time in $\|P\|$ because, by Proposition 5.4, each X_i is of size bounded by $\|C_i\|$. □

Remark 5.6. *An immediate consequence of Proposition 5.1 is that, for any balanced statement C, its critical pairs $\mathbf{Crit}(C)$ and size $\|C\|$ both remain unchanged under applications of the following rewrite rules to substatements of C:*

$$\begin{aligned}\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2 &\mapsto C_1; C_2\\ \textit{and}\qquad \texttt{while}(*)\ \texttt{do}\ C' &\mapsto C'\ .\end{aligned}$$

Therefore, the deadlock problem for our language reduces (polynomially) to the case where statements are restricted to the 'deterministic' grammar:

$$C :=\quad \texttt{skip} \mid p() \mid \texttt{acq}(\ell); C; \texttt{rel}(\ell) \mid C; C\ .$$

VI. Implementation and Impact

In this section we describe our implementation of a compositional deadlock analysis tool for code changes in Android Java applications. The core of the tool, described in Section VI-A, is an abstract interpretation analysis for computing the critical pairs of a method. In Section VI-B we describe our extension of this core procedure to an interprocedural analysis for Android Java code, and in Section VI-C we discuss its deployment and impact at Facebook.

A. Core analysis in abstract interpretation style

The core of our implementation is an analysis that computes the critical pairs of a statement in abstract interpretation style. Given any statement C, we define an analysis function $[\![C]\!](\cdot)$ on *abstract states*, which track the lock state and the set of critical pairs accumulated during the possible executions of C.

Definition 6.1. *An* abstract state *is a pair $\langle L, Z\rangle$, where L is a lock state and $Z \subseteq 2^{\mathsf{Locks}} \times \mathsf{Locks}$. We define a partial* join *operation $\sqcup$ on abstract states by $\langle L, Z_1\rangle \sqcup \langle L, Z_2\rangle = \langle L, Z_1 \cup Z_2\rangle$. We often write α for abstract states, and $\alpha_\bot$ for the "empty" abstract state $\langle \varnothing, \emptyset\rangle$.*

The function $[\![C]\!](\cdot)$ is then defined by structural induction on C in Figure 2. The clauses for the control flow statements are generic to abstract interpretation (given a suitable join operation $\sqcup$), which is why we do not simply define, e.g., $[\![\texttt{while}(*)\ \texttt{do}\ C]\!]\alpha = \alpha \sqcup [\![C]\!]\alpha$ as would intuitively be implied by equation (C5). However, this identity and similar ones can be inferred from our correctness proof.

We draw particular attention to the clause for procedure calls: computing the critical pairs of a procedure call $p()$ depends only on the current abstract state $\langle L, Z\rangle$ and the critical pairs of the procedure body in the empty state, $[\![\mathsf{body}(p)]\!]\alpha_\bot$, which can be computed "once and for all" in advance. This means that our analysis is *compositional* in that, when a procedure is changed, we only require to re-analyse that procedure and its dependents, not the whole program.

Proposition 6.2. *For any balanced statement C and abstract state $\alpha = \langle L, Z\rangle$, the result $[\![C]\!]\alpha$ of the analysis is given by*

$$\langle L, Z \cup \{(\lfloor L\rfloor \cup X, \ell) \mid (X, \ell) \in \mathbf{Crit}(C) \text{ and } L(\ell) = 0\}\rangle \ .$$

Thus $[\![C]\!]\alpha_\bot = \langle \varnothing, \mathbf{Crit}(C)\rangle$. Moreover, $[\![C]\!]\alpha$ is computable.

Proof. By structural induction on C, making use of the equations (C1)–(C6) in Proposition 5.4. □

Example 6.3. Continuing Example 5.2, statement C is:

$$\begin{aligned}&\texttt{acq}(\ell); \texttt{if}(*)\ \texttt{then}\ (\texttt{acq}(j); \texttt{skip}; \texttt{rel}(j))\\ &\qquad\qquad \texttt{else}\ (\texttt{acq}(k); \texttt{skip}; \texttt{rel}(k)); \texttt{rel}(\ell)\ .\end{aligned}$$

$$\begin{aligned}
[\![\texttt{acq}(\ell)]\!]\langle L, Z\rangle &= \langle L[\ell{+}{+}], Z \cup Z'\rangle \\
\text{where } Z' &= \begin{cases}\{(\lfloor L\rfloor, \ell)\} & \text{if } L(\ell) = 0\\ \emptyset & \text{if } L(\ell) > 0\end{cases}\\
[\![\texttt{rel}(\ell)]\!]\langle L, Z\rangle &= \langle L[\ell{-}{-}], Z\rangle\\
[\![p()]\!]\langle L, Z\rangle &= \langle L, Z \cup Z'\rangle\\
&\text{where } \langle _, Z''\rangle = [\![\mathsf{body}(p)]\!]\alpha_\bot \text{ in}\\
&Z' = \{(\lfloor L\rfloor \cup M, \ell) \mid (M, \ell) \in Z'' \wedge L(\ell) = 0\}\\
[\![\texttt{skip}]\!]\alpha &= \alpha\\
[\![C_1; C_2]\!]\alpha &= [\![C_2]\!]([\![C_1]\!]\alpha)\\
[\![\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\alpha &= ([\![C_1]\!]\alpha) \sqcup ([\![C_2]\!]\alpha)\\
[\![\texttt{while}(*)\ \texttt{do}\ C]\!]\alpha &= \bigsqcup_{n=0}^{\infty} [\![C]\!]^n\alpha
\end{aligned}$$

Fig. 2. Abstract analysis definition.

Writing abbreviations $C = \texttt{acq}(\ell); C'; \texttt{rel}(\ell)$ and $C' = \texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2$, we have by the clauses for `if` and for sequential composition (;) in Figure 2:

$$\begin{aligned}
[\![C]\!]\alpha_\bot &= [\![\texttt{acq}(\ell); C'; \texttt{rel}(\ell)]\!]\alpha_\bot\\
&= [\![C'; \texttt{rel}(\ell)]\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot\\
&= [\![\texttt{rel}(\ell)]\!][\![C']\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot\\
&= [\![\texttt{rel}(\ell)]\!][\![\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot\\
&= [\![\texttt{rel}(\ell)]\!]([\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot \sqcup [\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot)\ .
\end{aligned}$$

This gives us the basic structure of the computation. Next, using the rule for $\texttt{acq}(\ell)$ in Figure 2), we have

$$\begin{aligned}
[\![\texttt{acq}(\ell)]\!]\alpha_\bot &= [\![\texttt{acq}(\ell)]\!]\langle \varnothing, \emptyset\rangle\\
&= \langle \varnothing[\ell{+}{+}], \{(\emptyset, \ell)\}\rangle\ .
\end{aligned}$$

We write L_ℓ for the lock state $\varnothing[\ell{+}{+}]$ sending lock ℓ to 1 (and all other locks to 0). Next, we have

$$\begin{aligned}
&[\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot\\
&= [\![C_1]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle\\
&= [\![\texttt{acq}(j); \texttt{skip}; \texttt{rel}(j)]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle\\
&= [\![\texttt{skip}; \texttt{rel}(j)]\!][\![\texttt{acq}(j)]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle\\
&= [\![\texttt{rel}(j)]\!][\![\texttt{skip}]\!][\![\texttt{acq}(j)]\!]\langle L_\ell, \{(\emptyset, \ell)\}\rangle\\
&= [\![\texttt{rel}(j)]\!][\![\texttt{skip}]\!]\langle L_\ell[j{+}{+}], \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle\\
&= [\![\texttt{rel}(j)]\!]\langle L_\ell[j{+}{+}], \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle\\
&= \langle L_\ell[j{+}{+}][j{-}{-}], \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle\\
&= \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, j)\}\rangle\ .
\end{aligned}$$

And, by a similar calculation,

$$[\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\bot = \langle L_\ell, \{(\emptyset, \ell), (\{\ell\}, k)\}\rangle\ .$$

Now, using the definition of our abstract join $\sqcup$, we have

$$\begin{aligned}&[\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \sqcup [\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp\\ &= \langle L_\ell, \{(\emptyset,\ell),(\{\ell\},j)\}\rangle \sqcup \langle L_\ell, \{(\emptyset,\ell),(\{\ell\},k)\}\rangle\\ &= \langle L_\ell, \{(\emptyset,\ell),(\{\ell\},j),(\{\ell\},k)\}\rangle\ .\end{aligned}$$

Thus, putting everything together, we get

$$\begin{aligned}[\![C]\!]\alpha_\perp &= [\![\texttt{rel}(\ell)]\!]([\![C_1]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp \sqcup [\![C_2]\!][\![\texttt{acq}(\ell)]\!]\alpha_\perp)\\ &= [\![\texttt{rel}(\ell)]\!]\langle L_\ell, \{(\emptyset,\ell),(\{\ell\},j),(\{\ell\},k)\}\rangle\\ &= \langle L_\ell[\ell{--}], \{(\emptyset,\ell),(\{\ell\},j),(\{\ell\},k)\}\rangle\\ &= \langle \varnothing, \{(\emptyset,\ell),(\{\ell\},j),(\{\ell\},k)\}\rangle\ .\end{aligned}$$

Thus, recalling our recursive computation of $\mathbf{Crit}(C)$ in Example 5.2, we can see that indeed $[\![C]\!]\alpha_\perp = \langle \varnothing, \mathbf{Crit}(C)\rangle$.

Lemma 6.4. *Given a balanced statement C, the computation $[\![C]\!]\alpha_\perp$ requires at most quasi-exponential time in $\|C\|$. If C does not contain any procedure calls, the computation requires at most quadratic time in $|C|$.*

Proof. Follows from the fact that the computation of $[\![C]\!]\alpha_\perp$ is linear in the size of $\mathbf{Crit}(C)$ (its result) and the bounds on $\mathbf{Crit}(C)$ given by Proposition 5.4. □

Theorem 6.5. *The problem of checking whether a parallel program $P = C_1 \,||\, \ldots \,||\, C_n$ deadlocks can be solved in time exponential in $\|P\|$ and n. If the program does not contain any procedure calls, checking for deadlocks can be solved in time polynomial in $\|P\|$ and exponential in n.*

Proof. Computing $\mathbf{Crit}(C_1), \ldots, \mathbf{Crit}(C_n)$ can be performed in exponential time in $\|P\|$, by Lemma 6.4, and they contain at most exponentially many critical pairs, by Proposition 5.4. Then, checking the deadlock condition of Theorem 4.4 (over all possible index sets I) can be performed within a time bound exponential in $\|P\|$ and n. If P does not contain any procedure calls, then Lemma 6.4 and Prop. 5.4 instead yield a time bound polynomial in $\|P\|$ (but still exponential in n). □

B. Analysing Android Java code changes

The requirement that the analysis targets code changes leads to a number of design constraints, chief amongst which is that the analysis does not have the runtime envelope to analyse the whole program at hand; thus the analysis must work by analysing a modest superset of the modified code in a commit. Such a constraint can be problematic in looking for concurrency bugs due to their global nature. We note here techniques for addressing those difficulties as well as differences between implementation and theory. We also outline features specific to Android Java which we leverage in our analysis (cf. paragraphs on *Non-deterministic control* and *Concurrency inference*).

Balanced locking. The correctness of our analysis relies on balanced locking. As a general rule, this is good programming practice, supported in Java via the `synchronized` keyword, and we have found very few instances of unbalanced locking in the codebases targeted by our tool. This means that analysis precision does not suffer and that no changes to the analysis are needed, and is one reason we opted for a balanced-locking language model in the first place (the other reason being decidability). It is of course quite possible that unbalanced locking might well be more prevalent in other domains, but that is a matter of speculation. (Incidentally, our analyser actually *will* return a result when run on non-balanced code, but its correctness is not then guaranteed.)

Non-deterministic control. Control in Java is mostly deterministic, so our abstract semantics is over-approximate. In early trials of the tool, the majority of false positives we observed in practice stemmed from insensitivity to two conditions: firstly, whether a lock acquisition succeeded (e.g., via `Lock.trylock`); and secondly, whether the current thread is the UI thread. Therefore, we specialised the implementation domain to introduce partial path sensitivity on these conditions; this eliminated most of the false positives due to control abstraction.

Lock names. The set Locks must approximate the set of Java objects that can be used as monitors. Rather than use an expensive (and typically whole-program, which would run against our main design constraint) pointer analysis, we use *access paths*: syntactic expressions built with a program variable root and iteration of field- or array-dereferencing [22]. For example, `this.f.g` represents an object accessed through dereferencing the field `f` of the object `this`. Such a domain of abstract addresses has several trade-offs with respect to false positives and negatives, but that is beyond the scope of this paper.

We also classify objects into globally referenced or objects referenced through method parameters. Objects referenced through local variables are ignored. For globally referenced objects, the rule for method calls in Figure 2 applies unchanged. For parameter-referenced objects we apply a substitution of argument expressions over parameters on the callee summary before applying the procedure call rule. For instance, if the summary of method `foo(x)` involves the lock `x.f`, then applying the procedure call rule on `foo(h.g)` will result in the substitution `[h.g/x]` and the resulting critical pair at the callsite will involve the monitor `h.g.f`.

Concurrency inference. Since whole-program analysis is impracticable in our setting, we cannot always observe the spawning of execution threads, for these may happen in methods that are unmodified and unrelated via the call graph. As such we use an abstract domain for thread identity, where each method can be: of unknown identity; the UI thread; some background thread; or both (it may be executed on the UI thread as well as background threads). We extract this information from (a) thread annotations in Android code such as `@UiThread` and `@WorkerThread`; (b) Android method calls that test whether the current thread is the UI thread; (c) upward propagation through the call graph. Every critical pair in a method summary is decorated with the inferred thread identity, and this information is used to determine whether

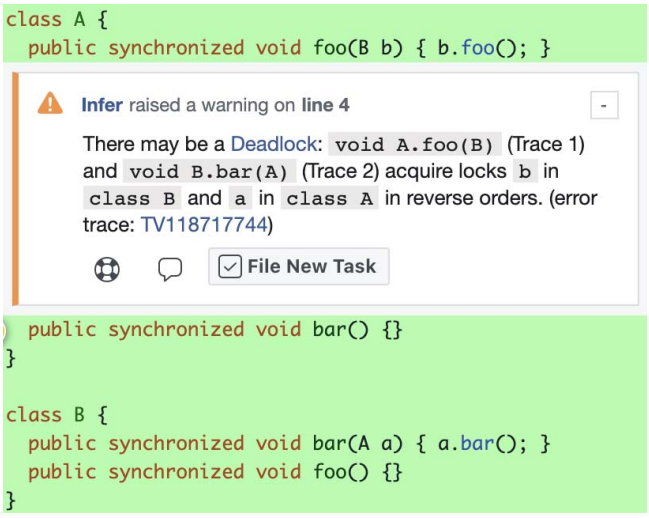

Fig. 3. Analysis report on textbook deadlock across two Java classes.

two critical pairs can occur concurrently (two UI-thread pairs cannot be concurrent, though any other combination can).

Detecting deadlocks non-globally. As the analysis targets code changes, it begins by summarising all methods in the set of changed files in a commit. By the procedure call rule, this leads to analysing all methods transitively called by the modified files. If we restrict deadlock detection to this set of summaries, we will miss deadlocks due to lock acquisitions performed by methods outside the call graph rooted in modified files. Thus, the analysis selects additional methods to summarise using the following heuristic.

For every method `M` summarised and every critical pair of the form $(L, \texttt{root}.f_1.....f_n)$ in the summary of `M`, where `root` is of class `C`, all methods of class `C` are also analysed (in search of a critical pair (L', ℓ') where $\texttt{root}.f_1.....f_n \in L'$). The analyser continues this process until the set of analysed methods reaches a fixpoint.

This heuristic works well when certain Java idioms are observed, namely when the monitors used are the `this` object (for example, in `synchronized` methods), or they are immutable private objects stored in object fields. For instance, this heuristic will catch the deadlock between classes `A` and `B` in Figure 3 (where we also illustrate a sample report to developers) even in a commit where only `A.foo` is modified. It's worth noting that our heuristic admits false negatives, e.g. when global locks are acquired in methods that reside in classes not containing the globals. However, in the code we usually analyse global locks are significantly less commonly used than non-global objects.

C. Industrial deployment and impact

Our flow-sensitive, context-insensitive analysis is implemented in OCaml (around 3kLoC) within the INFER static analysis framework [1], [11], and is specifically targeted at detecting *2-thread deadlocks in code changes (commits) of Android apps* within a continuous integration environment (CI).

Deployment. INFER is deployed at Facebook through a CI system which launches an analysis job whenever a commit is submitted for code review. This job concurrently runs multiple analysers on the submitted code changes and appears to the authors of the commit as yet another reviewer commenting on the code, based on the potential bugs found. *The deadlock analysis has been deployed on all Android code commits at Facebook for about two years.*

Fixed reports. In a non-safety-critical context such as Facebook, an analysis engineer's time is better spent developing analysis features than triaging reports for false positives. Thus, we track fixed reports (reports that code authors addressed by submitting a new version of a commit) rather than true positives. *Since it was deployed, the deadlock analyser has processed hundreds of thousands of commits, has issued more than 500 deadlock reports with a fix rate of 54%.*

As for the 46% of non-fixed reports, these fall into three broad categories. First, true positives that were in fact fixed by the developer, but not picked up by our fix tracking (e.g. because the bug was fixed in a new commit rather than a new version of an existing commit). Second, true positives that the developer decided not to address (e.g. because the bug would occur only in very rare circumstances). Third, false positives (e.g. because the tool reports a deadlock between two methods that will never in fact be run concurrently). Unfortunately, we have absolutely no way of knowing how many reports fall into each of these categories.

Analysis performance. The architecture of INFER means per-analyser runtime is not recorded. For this reason, we report only the total analysis time (including various other analysers), which provides an upper bound for our analysis. *Runtime for all analysers in the last 100 days to submission had a median of 90 seconds and an average of 213 seconds per commit. In the same time period,* INFER *analysed a median of 2k methods and on average 5k methods per commit.*

VII. RELATED WORK

In this section we compare our contribution to related work, first, on theoretical deadlock detection results based on automata, and second, on automated deadlock detection tools. We note that such tools are naturally classified as either dynamic, static or hybrid, depending on whether they operate primarily on program executions, program text, or both. Analysers that target Java programs typically rely on balanced locking and must accurately model re-entrant locks, whereas analysers for C code do not expect balanced locking and assume non-reentrant locks. In addition, deadlock analysers can be categorised according to whether they detect deadlocks involving only two threads, or more, and whether they produce false positives on guarded cycles.

A. Automata-theoretical deadlock detection

There is a substantial body of work on the literature on analysing general safety and liveness properties for communicating systems of pushdown automata, e.g. [34], [5], [20], [17], [14], [25], [26], [23], [24], [15]. Our abstract concurrent programs fall under this general umbrella since, as we have already observed (cf. Remark 3.17), they can be seen as collections of finite automata that synchronise via their shared locks. Indeed, our model can be polynomially encoded as a communicating pushdown system as considered e.g. in [5]. However, while arbitrary dataflow properties of even two-threaded pushdown systems are undecidable in general [31], our deadlock problem is decidable and thus represents a special case, which relies crucially on the fact that locking in our language is balanced.

The papers in this area most directly relevant to our own work are probably [25] and [24], which both consider comuunicating pushdown systems in which locks are also balanced (there "nested"). In particular, [25] reports decidability of the deadlocking problem for such systems, based on a cycle-checking condition in a graph obtained by augmenting the automaton with lock acquisition histories. This condition is somewhat similar to (though arguably more complicated than) our condition based on critical pairs, and the central theoretical result on which it relies (Theorem 5) has the same essential flavour as our similarly crucial Lemma 4.3, although the technical machinery is quite different. In [24] this result is strengthened to the decidability of any LTL property of communicating nested-lock pushdown systems, in exponential time in the number of locks. While these papers do not explicitly treat reentrant locks, it is shown in [26] that reentrant locks can be modelled in such systems using only non-reentrant locks, so this is not a serious point of difference.

Thus, although our deadlock characterisation result (Theorem 4.4) is new in itself, it probably *could* have alternatively been arrived at via a combination of the results in [25], [24] and [26], by encoding our language as a communicating nested-lock pushdown system. However, such an approach would almost certainly be messy, unsatisfying, and no less work than our own proof. Compared to these works, the novelties of our approach are as follows: we characterise deadlocks in a (simple) concurrent programming language rather than a system of communicating automata; our deadlock condition based on critical pairs is simpler than the automata-theoretic condition; and we give a proof for this setting that is entirely direct, self-contained and, we believe, quite elegant.

In a different direction, it is also worth mentioning [6] and [16]. These both aim at deadlock *avoidance* (rather than detection) in a concurrent functional language with re-entrant locks, by developing a type system for the language such that typable expressions are guaranteed not to deadlock. While the earlier [6] requires locks to be balanced, [16] lifts this restriction, and provides a prototype implementation.

B. Static analyses

Static deadlock detectors typically require a complete program in order to run. Most are focused on soundness, where the absence of reports implies deadlock-freedom. All analyses discussed are interprocedural, top-down, context-sensitive and typically non-compositional.

RACERX [12] is a path-insensitive analysis for C programs which does not use a pointer analysis, instead using syntactic information and types about variables, ignoring locks in local variables. Heavy use of caching transfer functions on statements is made, to improve runtimes due to context sensitivity. The search for cycles is up to a user specified number of threads. Many heuristics and techniques are employed to reduce false positive reports.

[32] reports on a Java analysis which targets libraries, thus partly dealing with the problem of identifying program entrypoints. As such, the analysis cannot see global aliasing information and uses a coarse, type-based memory domain. It can detect cycles of more than two threads, up to a pre-specified bound. The pure analysis reports too many false positives, therefore several unsound heuristics are used.

JADE [29] is a path-insensitive Java analysis which breaks down the problem into several sub-analyses, including reachability, aliasing/escaping, reentrancy and guarded-ness. It focuses on two-thread deadlocks, and has explicit mechanisms for rejecting guarded deadlock reports. It is expressed in Datalog and uses an iterative refinement scheme to increase precision, where the degree of sensitivity is increased based on the reports found in the last iteration.

[30] reports on an analysis for an abstract language, which reduces detection of deadlocks into race detection. A type system captures lock dependencies, and the inferred types are used to detect program points where a nested lock acquisition may occur. These points are instrumented with code mutating "race" variables. A data race detector then finds possible deadlocks. Too many false positives are reported for deadlocks among more than two threads, and additional checks are made to improve precision and to filter guarded cycles. No implementation is reported.

THREADSAFE [3] is a commercial, flow- and path-sensitive, per-class analysis for Java. Little detail is reported on the foundations of the analysis. It uses as entry points the public methods of each class, or modelled Android lifecycle methods. Only calls to private and protected methods are followed.

[27] reports on an analysis targeting C code with Posix threads. It infers concurrency on spawn/join points through the program graph, and contexts represent call- and thread-creation- sites. A must-lock analysis is employed to deal with guarded locks. Function pointer calls are inlined into case distinctions over the calls they might resolve to.

JADA [28] is a Java bytecode analysis that uses behavioural type rules to compositionally extract an infinite-state abstract model from bytecode. This model is then analysed using a context-sensitive fixpoint computation, generating reports of cyclic dependencies. The main strength of the approach seems

to be the ability to analyse recursive functions that spawn an unbounded number of threads.

C. Dynamic and hybrid analyses

Analyses that work with program traces usually require the whole program as well as appropriate test input, and tend to be focused on completeness (most reports are true positives).

GOODLOCK [18] is an analysis for Java programs implemented in Java PathFinder (JPF) [19] which maintains a lock-tree for each execution thread, where each node represents the lifetime of a lock acquisition and children nodes represent acquisitions wholly contained within the parent. A warning is reported whenever two threads have lock-trees which may request the same pair of locks in opposite orders. Since the whole lock-tree is available, gate locks can be detected and the warning suppressed. MAGICFUZZER [7], MAGICLOCK [8], UNDEAD [33] and AIRLOCK [9] all adopt and improve on this basic approach by applying various optimisations to the extracted lock order graph, with UNDEAD also attempting to keep traces in memory rather than external storage as far as possible, and AIRLOCK operating on-the-fly, running a polynomial-time algorithm on the lock graph to eliminate parts with no cycles before running the higher-cost algorithm to detect actual lock cycles.

[4] describes an analysis for Java programs, also implemented in JPF, that constructs a lock-order graph from an execution trace of an instrumented program. Although the graph edges denote dependencies between only a pair of locks, they are also labelled by the complete lock-set and the thread acquiring the lock. These labels are used to detect deadlocks between more than two threads and to filter out gated cycles.

[2] presents a sound type inference mechanism for types that ensure deadlock freedom for Java programs. Appropriate instrumentation for the untyped parts of the program is then used to feed an extension of the GOODLOCK algorithm to the unbounded thread case, yielding a hybrid analysis. Further filtering is then used to exclude gated cycles.

SHERLOCK [13] is an analysis for Java programs which uses GOODLOCK to get a set of deadlock candidates. The program is run on given inputs, producing an initial schedule which is then concolically executed and permuted in repeated steps, in search of witnessing schedules. The GOODLOCK-based algorithm can deal with more than two threads, and the original version can deal with gated cycles.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we provide a highly scalable, open-source deadlock analyser for Android Java, based on a sound and complete deadlock analysis (in NP) for an abstract concurrent programming language. Our deadlock analyser has been deployed at Facebook as part of the INFER framework for the last two years and has resulted in hundred of potential bugs being flagged, with an actual developer fix rate of over 50% (and we note that this does not imply a false positive rate of nearly 50%). The abstract programming language on which the analysis is based contains balanced (or nested) reentrant locks, nondeterministic control flow commands and nonrecursive procedure calls, but with all other features — in particular variable assignment — abstracted away. An abstraction of this sort is of course necessary to obtain decidability, for fundamental computability reasons. However, our overapproximation of real concurrent programs turns out to be sufficiently faithful to detect deadlock bugs in practice, and sufficiently scalable to run on commercial Android applications.

It is natural to wonder whether and how our abstract language might be extended while preserving the decidability of deadlock existence. Unfortunately, it seems to us that almost any nontrivial extension presents significant obstacles. For example, allowing procedure calls to be recursive causes problems for our approach since it is then possible to create statements with infinite non-balanced traces (e.g. by the procedure definition $p : \texttt{acq}(\ell); p(); \texttt{rel}(\ell)$); a second difficulty is that we also cannot straightforwardly reason by induction over the structure of statements. Allowing control flow to be deterministic, e.g. by allowing guards to query the lock state, is similarly problematic since the critical pairs of a statement are then dependent on the lock state in which it is executed, meaning that at the very least we would require a finer abstraction in order to avoid false positives. Finally, modelling forking and joining by allowing parallel compositions to appear nested within statements makes the problem much more complicated since, conceptually, it would require that we construct abstractions of all subthreads as well as determining which of them can run in parallel with each other. We nevertheless consider these (and other) extensions to be interesting potential directions for future work. It would also be nice to establish a lower complexity bound on our deadlock problem; we speculate that the problem is likely NP-complete, but unfortunately we have so far failed to find a suitable reduction.

REFERENCES

[1] Facebook INFER static analysis framework. https://fbinfer.com.

[2] R. Agarwal, L. Wang, and S. D. Stoller, "Detecting potential deadlocks with static analysis and run-time monitoring," in *Hardware and Software, Verification and Testing*. Springer, 2006, pp. 191–207.

[3] R. Atkey and D. Sannella, "ThreadSafe: Static analysis for java concurrency," *ECEASST*, vol. 72, 2015.

[4] S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Hardware and Software, Verification and Testing*. Springer, 2006, pp. 208–223.

[5] A. Bouajjani, J. Esparza, and T. Touili, "A generic approach to the static analysis of concurrent programs with procedures," in *Proceedings of POPL-30*. ACM, 2003, p. 62–73.

[6] G. Boudol, "A deadlock-free semantics for shared memory concurrency," in *Proceedings of ICTAC-6*. Springer, 2009, pp. 140–154.

[7] Y. Cai and W. Chan, "MagicFuzzer: scalable deadlock detection for large-scale applications," in *Proceedings of ICSE-34*. ACM, 2012, pp. 606–616.

[8] ——, "MagicLock: Scalable detection of potential deadlocks in large-scale multithreaded programs," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 266–281, 2014.

[9] Y. Cai, R. Meng, and J. Palsberg, "Low-overhead deadlock prediction," in *Proceedings of ICSE-42*. ACM, 2020, pp. 1298–1309.

[10] E. Dijkstra, "Hierarchical ordering of sequential processes," *Acta Informatica*, vol. 1, no. 2, pp. 115–138, 1971.

[11] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at Facebook," *Comm. ACM*, vol. 62, no. 8, p. 62–70, Jul. 2019.

[12] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of SOSP-19*. ACM, 2003, p. 237–252.

[13] M. Eslamimehr and J. Palsberg, "Sherlock: Scalable deadlock detection for concurrent programs," in *Proceedings of FSE-22*. ACM, 2014, p. 353–365.

[14] J. Esparza, P. Ganty, and R. Majumdar, "Parameterized verification of asynchronous shared-memory systems," in *Proceedings of CAV*. Springer, 2013, pp. 124–140.

[15] A. Farzan and Z. Kincaid, "Compositional bitvector analysis for concurrent programs with nested locks," in *Proceedings of SAS-17*. Springer, 2010, pp. 253–270.

[16] P. Gerakios, N. Papaspyrou, and K. Sagonas, "A type and effect system for deadlock avoidance in low-level languages," in *Proceedings of TLDI-7*. ACM, 2011, pp. 15–28.

[17] M. Hague, "Parameterised pushdown systems with non-atomic writes," *LIPIcs*, vol. 13, 09 2011.

[18] K. Havelund, "Using runtime analysis to guide model checking of Java programs," in *SPIN Model Checking and Software Verification*. Springer, 2000, pp. 245–264.

[19] K. Havelund and T. Pressburger, "Model checking Java programs using Java PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[20] A. Heußner, J. Leroux, A. Muscholl, and G. Sutre, "Reachability analysis of communicating pushdown systems," in *Proceedings of FoSSaCS*. Springer, 2010, pp. 267–281.

[21] J. E. Hopcroft and J. D. Ullman, *Formal Languages and Their Relation to Automata*. Addison-Wesley, Jan. 1969.

[22] N. D. Jones and S. S. Muchnick, "Flow analysis and optimization of lisp-like structures," in *Proceedings of POPL-6*. ACM, 1979, p. 244–256.

[23] V. Kahlon, "Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise CFL-reachability for threads communicating via locks," in *Proceedings of LICS-24*. IEEE, 2009, pp. 27–36.

[24] V. Kahlon and A. Gupta, "An automata-theoretic approach for model checking threads for LTL properties," in *Proceedings of LICS-21*. IEEE, 2006, pp. 101–110.

[25] V. Kahlon, F. Ivancic, and A. Gupta, "Reasoning about threads communicating via locks," in *Proceedings of CAV-17*. Springer, 2005, pp. 505–518.

[26] N. Kidd, A. Lal, and T. Reps, "Language strength reduction," in *Proceedings of SAS-15*. Springer, 2008, pp. 283–298.

[27] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter, "Sound static deadlock analysis for C/Pthreads," in *Proceedings of ASE-31*, 2016, pp. 379–390.

[28] C. Laneve and A. Garcia, "Deadlock detection of Java bytecode," in *Logic-Based Program Synthesis and Transformation*. Springer, 2018, pp. 37–53.

[29] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proceedings of ICSE-31*. IEEE Computer Society, 2009, p. 386–396.

[30] K. I. Pun, M. Steffen, and V. Stolz, "Deadlock checking by data race detection," *Journal of Logical and Algebraic Methods in Programming*, vol. 83, no. 5, pp. 400 – 426, 2014, the 24th Nordic Workshop on Programming Theory (NWPT 2012).

[31] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, p. 416–430, 2000.

[32] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for Java libraries," in *ECOOP 2005 - Object-Oriented Programming*, A. P. Black, Ed. Springer, 2005, pp. 602–629.

[33] J. Zhou, S. Silvestro, H. Liu, Y. Cai, and T. Liu, "UnDead: detecting and preventing deadlocks in production software," in *Proceedings of ASE-32*. ACM, 2017, pp. 729–740.

[34] W. Zielonka, "Notes on finite asynchronous automata," *RAIRO - Theoretical Informatics and Applications*, vol. 21, no. 2, pp. 99–135, 1987.