Decidable Deadlock Detection for an Abstract Scoped-Locking Language

JAMES BROTHERSTON, University College London, UK
PAUL BRUNET, University College London, UK
NIKOS GOROGIANNIS, Facebook London and Middlesex University, UK
MAX KANOVICH, University College London, UK

We study the problem of deadlock detection for an abstract programming language with balanced re-entrant locks, nondeterministic iteration and branching, and non-recursive procedure calls.

First, we show that the existence of a deadlock in a concurrent program is equivalent to a certain condition over the sets of so-called *critical pairs* of each of its threads. The critical pairs of a thread record, for all possible executions of the thread, which locks are currently held at the point when a fresh lock is acquired.

Second, we show that the set of critical pairs of any program thread is finite and computable. As a consequence, the deadlock detection problem for our abstract language is decidable, and in NP. We also present an algorithm which computes critical pairs in a compositional, abstract interpretation style, running in quasi-exponential time. All of our proof developments have been formalised in the Coq proof assistant.

Third, we provide an open-source implementation of a version of our analysis adapted to Java. Our analyser is built in the Infer verification framework and has been in deployment at Facebook for over two years; it has seen over two hundred fixed deadlock reports with a report fix rate of approximately 54%.

Additional Key Words and Phrases: deadlocks, concurrency, program analysis

1 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" [Dijkstra 1971]: 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 fork 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 consider the problem of detecting deadlocks in an abstract concurrent programming language featuring scoped re-entrant locks, nondeterministic iteration and branching, and nonrecursive procedure calls. This language can be seen as an overapproximate model of real-world programming languages such as Java, with all information about variable and memory assignment abstracted away.

We make three principal contributions to the problem, two theoretical and one practical. Our first contribution is to show that the existence of a deadlock in our abstract programs can be precisely characterised as a condition on the *critical pairs* of each of its (sequential) threads. Roughly speaking, a critical pair of a thread is a pair (X, ℓ) such that some execution of the thread acquires

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

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

an unheld lock ℓ while already holding the set of locks X. For the case of two threads, we establish that $C_1 \,||\, C_2$ 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 $X_1 \cap X_2 = \emptyset$ (cf. Theorem 4.4). This condition can be generalised to the case of arbitrarily many threads (cf. Theorem 5.5). 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`.

Example 1.1. Consider a two-threaded program $C_1 \,||\, C_2$, where C_1 and C_2 are the following sequential programs acquiring locks ($\mathtt{acq}(-)$) and releasing them ($\mathtt{rel}(-)$) in reverse order:

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

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 $(X_2, \ell_2) = (\{x\}, y)$, we can see that the condition above is met, and indeed $C_1 \,||\, C_2$ 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 $C_1' \,||\, C_2'$, where $C_1' = \mathtt{acq}(z); C_1; \mathtt{rel}(z)$ and $C_2' = \mathtt{acq}(z); C_2; \mathtt{rel}(z)$. 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 $C_1' \,||\, C_2'$ does not deadlock, because z acts as a "guard lock" preventing x and y from being accessed by C_1' and C_2' simultaneously.

Our second contribution is to show that the set of critical pairs of any thread in our language is in fact finite and computable. Consequently, due to the above characterisation of deadlocks, the existence of deadlocks in our abstract programs becomes decidable (and in NP). We present both a direct inductive computation of critical pairs, and a context-insensitive, flow-sensitive program analysis that computes them in abstract interpretation style, running in quasi-exponential time in the syntactic size of the program.

Our third contribution is an adaptation of our analysis to Java, and an open-source implementation within the Infer static analysis framework, aimed at finding deadlocks in code changes in Android applications. We describe its deployment and impact at Facebook, where it has seen over two hundred deadlock reports fixed in the last two years.

All of our theoretical results have also been proved mechanically in the Coq proof assistant[1]. The formalisation occupies roughly 8.7K lines of code, and follows fairly closely the pen-and-paper proofs in this paper. However, whereas for pedagogical reasons we shall begin here by considering two-threaded programs and then generalise to the case of n threads, the mechanised proofs deal directly with the general case.

The remainder of this paper is structured as follows. First, Section 2 introduces the syntax and semantics of our abstract concurrent programs (restricted initially to the two-threaded case). In Section 3 we develop the notion of a (sequential) program execution's *trace*, i.e. the sequence of lock acquisitions and releases it makes, and establish the key technical relationships between traces and executions. Then, in Section 4, we establish the soundness and completeness of our deadlock condition based on critical pairs, for two-threaded programs (as above). Section 5 generalises this result to the case of programs with $n \geq 2$ threads. In Section 6 we show that the set of critical pairs of any sequential program is finite and computable, and establish complexity bounds on the problem. Section 7 describes our implementation of the deadlock analysis and its deployment impact at Facebook. Section 8 surveys the related work, and Section 9 concludes.

[1] Made available as supplementary material for the referees.

2 PROGRAM SYNTAX AND SEMANTICS

Syntax. Locks is a finite set of *global lock names* and Procs is a set of *procedure names.*

We define *statements* C by the following grammar, where ℓ ranges over Locks and p over Procs:

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

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

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

Moreover, balanced statements must call only balanced procedures: if C is balanced and $p \in$ callees(C), then 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 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 ordered pair of balanced statements written $C_1 \mid\mid C_2$.

Semantics. Since our programs employ only non-deterministic control flow and lock guards, 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 : Locks $\to \mathbb{N}$, recording how many times each lock has been acquired. We use the notation $\lfloor L \rfloor$ for $\{\ell \in \text{Locks} \mid L(\ell) > 0\}$. 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 a pair $\langle C_1 \mid\mid C_2, (L_1, L_2)\rangle$, where $C_1 \mid\mid C_2$ is a parallel program and L_1, L_2 are lock states. We will also denote this concurrent configuration as $\langle C_1, L_1\rangle \parallel \langle C_2, L_2\rangle$. We write $\langle C_1, L_1\rangle \# \langle C_2, L_2\rangle$ to mean that $L_1 \# L_2$.

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, $\rightsquigarrow$. A configuration $\langle C, L\rangle$ is called *live* if there exists a transition $\langle C, L\rangle \to \langle C', L'\rangle$.

Definition 2.1. *An* execution *(of statement C) is a possibly infinite sequence of configurations $\pi = (\gamma_i)_{i\geq 0}$ (with $\gamma_0 = \langle C, \_\rangle$) such that $\gamma_i \to \gamma_{i+1}$ for all $i \geq 0$.*

A concurrent execution *is defined analogously to an execution, by replacing concurrent configurations for configurations and $\rightsquigarrow$ for $\to$ in the above.*

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

We make the following simple but useful observation on our semantics:

$$\langle \texttt{skip}; C, L\rangle \to \langle C, L\rangle \qquad (\texttt{skip})$$

$$\langle p(), L\rangle \to \langle \texttt{body}(p), L\rangle \qquad (\texttt{proc})$$

$$\langle \texttt{acq}(\ell), L\rangle \to \langle \texttt{skip}, L[\ell{+}{+}]\rangle \qquad (\texttt{acq})$$

$$\langle \texttt{rel}(\ell), L\rangle \to \langle \texttt{skip}, L[\ell{-}{-}]\rangle \ (L(\ell) > 0) \qquad (\texttt{rel})$$

$$\langle \texttt{if}(*) \texttt{ then } C_a \texttt{ else } C_b, L\rangle \to \langle C_a, L\rangle \qquad (\texttt{if1})$$

$$\langle \texttt{if}(*) \texttt{ then } C_a \texttt{ else } C_b, L\rangle \to \langle C_b, L\rangle \qquad (\texttt{if2})$$

$$\langle \texttt{while}(*) \texttt{ do } C, L\rangle \to \langle \texttt{skip}, L\rangle \qquad (\texttt{while1})$$

$$\langle \texttt{while}(*) \texttt{ do } C, L\rangle \to \langle C; \texttt{while}(*) \texttt{ do } C, L\rangle \qquad (\texttt{while2})$$

$$\frac{\langle C_1, L\rangle \to \langle C_1', L'\rangle}{\langle C_1; C_2, L\rangle \to \langle C_1'; C_2, L'\rangle} \qquad (\texttt{seq})$$

$$\frac{\langle C_1, L_1\rangle \to \langle C_1', L_1'\rangle \quad L_1' \# L_2}{\langle C_1 \mid\mid C_2, (L_1, L_2)\rangle \rightsquigarrow \langle C_1' \mid\mid C_2, (L_1', L_2)\rangle} \qquad (\texttt{par1})$$

$$\frac{\langle C_2, L_2\rangle \to \langle C_2', L_2'\rangle \quad L_1 \# L_2'}{\langle C_1 \mid\mid C_2, (L_1, L_2)\rangle \rightsquigarrow \langle C_1 \mid\mid C_2', (L_1, L_2')\rangle} \qquad (\texttt{par2})$$

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

Remark 2.2. *For any concurrent execution $\gamma_1 \mid\mid \gamma_2 \rightsquigarrow^* \gamma_1' \mid\mid \gamma_2'$, there exist standard executions $\gamma_1 \to^* \gamma_1'$ and $\gamma_2 \to^* \gamma_2'$. Furthermore, if $\gamma_1 \# \gamma_2$, then $\gamma_1' \# \gamma_2'$; i.e., the two threads cannot acquire the same lock simultaneously.*

Definition 2.3. *A concurrent configuration $\sigma = \langle C_1' \mid\mid C_2', (L_1, L_2)\rangle$ is* deadlocked *if both $\langle C_1', L_1\rangle$ and $\langle C_2', L_2\rangle$ are live, and there is no σ' such that $\sigma \rightsquigarrow \sigma'$. The parallel program $C_1 \mid\mid C_2$ is said to* deadlock *if there exists an execution $\langle C_1 \mid\mid C_2, (\varnothing, \varnothing)\rangle \rightsquigarrow^* \sigma$ such that σ is deadlocked.*

Deadlocked configurations can be characterised using the following observation.

Proposition 2.4. *Let $\sigma = \langle C_1 \mid\mid C_2, (L_1, L_2)\rangle$ be a concurrent configuration such that $L_1 \# L_2$. The configuration σ is deadlocked iff there are statements D_1, D_2 and locks ℓ_1, ℓ_2 such that*

$$\langle C_1, L_1\rangle \to \langle D_1, L_1[\ell_1{+}{+}]\rangle \ , \ \langle C_2, L_2\rangle \to \langle D_2, L_2[\ell_2{+}{+}]\rangle \ , \ \ell_1 \in \lfloor L_2 \rfloor \ \text{ and } \ \ell_2 \in \lfloor L_1 \rfloor \ .$$

Proof. **Case ($\Rightarrow$):** By assumption, $\langle C_1, L_1\rangle$ and $\langle C_2, L_2\rangle$ are live, with $L_1 \# L_2$, but there is no configuration σ' such that $\sigma \rightsquigarrow \sigma'$. Since $\langle C_1, L_1\rangle$ is live, we have $\langle C_1, L_1\rangle \to \langle D_1, L_1'\rangle$. We show that $L_1' = L_1[\ell_1{+}{+}]$ for some lock ℓ_1. Otherwise, by inspection of the semantics for $\to$ (Figure 1), the only other possibilities are either $L_1' = L_1$ or $L_1' = L_1[\ell_1{-}{-}]$. In either case, $\lfloor L_1' \rfloor \subseteq \lfloor L_1 \rfloor$, and since $L_1 \# L_2$ by assumption, we have $L_1' \# L_2$ and thus a concurrent transition using the rule par1:

$$\frac{\langle C_1, L_1\rangle \to \langle D_1, L_1'\rangle \quad L_1' \# L_2}{\langle C_1 \mid\mid C_2, (L_1, L_2)\rangle \rightsquigarrow \langle D_1 \mid\mid C_2, (L_1', L_2)\rangle}$$

which contradicts the fact that σ is deadlocked. For the same reason, we must have $\ell_1 \in \lfloor L_2 \rfloor$, since $\lfloor L_1' \rfloor = \lfloor L_1 \rfloor \cup \{\ell_1\}$ and $L_1 \# L_2$. By a symmetric argument, we also have $\langle C_2, L_2\rangle \to \langle D_2, L_2[\ell_2{+}{+}]\rangle$ and $\ell_2 \in \lfloor L_1 \rfloor$ for some lock ℓ_2.

Case ($\Leftarrow$)**:** Assume $\langle C_1, L_1\rangle \to \langle D_1, L_1[\ell_1{+}{+}]\rangle$ and $\langle C_2, L_2\rangle \to \langle D_2, L_2[\ell_2{+}{+}]\rangle$ with $\ell_1 \in \lfloor L_2 \rfloor$ and $\ell_2 \in \lfloor L_1 \rfloor$. Thus $\langle C_1, L_1\rangle$ and $\langle C_2, L_2\rangle$ are both live. Assume for contradiction that $\sigma \rightsquigarrow \sigma'$. Without loss of generality, we assume that this transition occurs via the rule par1:

$$\frac{\langle C_1, L_1\rangle \to \langle C_1', L_1'\rangle \quad L_1' \# L_2}{\langle C_1 \mid\mid C_2, (L_1, L_2)\rangle \rightsquigarrow \langle C_1' \mid\mid C_2, (L_1', L_2)\rangle}$$

Since $\langle C_1, L_1\rangle \to \langle D_1, L_1[\ell_1{+}{+}]\rangle$, it is clear by inspection of the semantics for $\to$ that we must have $L_1' = L_1[\ell_1{+}{+}]$; only the command $\mathsf{acq}(\ell_1)$ (possibly suffixed by some other statement) can increment ℓ_1. Thus $L_1[\ell_1{+}{+}] \# L_2$, which is a contradiction since $\ell_1 \in \lfloor L_2 \rfloor$ by assumption and $\ell_1 \in \lfloor L_1[\ell_1{+}{+}] \rfloor$. We conclude that σ' cannot exist, and so σ is deadlocked as required. □

3 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

$$\mathsf{acq}(\ell); \mathtt{if}(*)\ \mathtt{then}\ (\mathsf{acq}(j); \mathtt{skip}; \mathtt{rel}(j))\ \mathtt{else}\ (\mathsf{acq}(k); \mathtt{skip}; \mathtt{rel}(k)); \mathtt{rel}(\ell)$$

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.

From our point of view, 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 can be seen as *Dyck words* [Hopcroft and Ullman 1969] in formal language theory, as most notably used in the Chomsky–Schützenberger representation theorem [Chomsky and Schützenberger 1963].

3.1 From executions to traces

In this section, we show how to map executions of our statements to *traces*, which are words over a suitable "lock alphabet", and statements to languages of such traces, in a consistent way.

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:*

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

We immediately notice the following relation between $\langle\cdot\rangle$ and concatenation.

Lemma 3.2. *For any* Σ*-words* u *and* v *we have* $\langle u\, v\rangle = \langle u\rangle + \langle v\rangle$*.*

Proof. A straightforward induction on v. □

Next, we map executions of our statements to words over Σ. We can observe by inspecting our semantics (Figure 1) that in any execution step $\langle C, L\rangle \to \langle D, M\rangle$ we have $M = L$, or $M = L[\ell{+}{+}]$ or $M = L[\ell{-}{-}]$ for some lock ℓ. This justifies the following definition.

Definition 3.3. *Given a transition $\tau : \langle C, L\rangle \to \langle C', L'\rangle$ in our operational semantics, we define its* trace $\lambda(\tau) \in \Sigma \cup \{\varepsilon\}$ *as follows:*

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

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 $\tau : \langle C, L\rangle \xrightarrow{u} \langle C', L'\rangle$ and $\pi : \langle C, L\rangle \xrightarrow{u}{}^{*} \langle D, M\rangle$.

The trace of an execution is preserved under suffixing statements.

Proposition 3.4. *For any execution $\pi : \langle C_0, L_0\rangle \xrightarrow{u}{}^{*} \langle C_n, L_n\rangle$ and statement C, there is also an execution $\pi' : \langle C_0; C, L_0\rangle \xrightarrow{u}{}^{*} \langle C_n; C, L_n\rangle$.*

Proof. The statement is an inductive consequence of the fact that for any step $\langle C_i, L_i\rangle \to \langle C_{i+1}, L_{i+1}\rangle$ in the original execution π we can apply the rule (seq) in Figure 1 to obtain:

$$\frac{\pi_i : \langle C_i, L_i\rangle \to \langle C_{i+1}, L_{i+1}\rangle}{\pi_i' : \langle C_i; C, L_i\rangle \to \langle C_{i+1}; C, L_{i+1}\rangle}.$$

Since this new transition has the same input-output lock states, it has the same trace. □

We now define the language of a statement, roughly speaking, as the set of traces generated by 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:*

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

Remark 3.6. *For any statement C we have that $\mathcal{L}(C)$ is in fact a* regular language *over Σ: it is obtained from $\{\varepsilon\}$, $\{\ell\}$ and $\{\overline{\ell}\}$ by applying concatenation, union, and Kleene star. Furthermore, by construction, $\mathcal{L}(C)$ is never empty.*

Our next lemma establishes that the trace of an execution of statement C determines its effect on the lock state, and is a prefix of some word in $\mathcal{L}(C)$.

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 case where π is a single transition step, by rule induction on the transition relation (cf. Figure 1). The base cases — (skip), (proc), (acq), (rel), (if1), (if2), (while1) and (while2) — are all easy verifications. In the inductive case, (seq), we have:

$$\frac{\langle C_1, L\rangle \xrightarrow{u} \langle C_1', L'\rangle}{\langle C_1; C_2, L\rangle \xrightarrow{u} \langle C_1'; C_2, L'\rangle}$$

By the induction hypothesis we have $L = L + \langle u\rangle$ and $u \cdot \mathcal{L}(C_1') \subseteq \mathcal{L}(C_1)$. Thus we get $u \cdot \mathcal{L}(C_1'; C_2) = u \cdot \mathcal{L}(C_1') \cdot \mathcal{L}(C_2) \subseteq \mathcal{L}(C_1) \cdot \mathcal{L}(C_2) = \mathcal{L}(C_1; C_2)$ as required. This completes the single-step case.

For an arbitrary execution, the result then follows by reflexive-transitive induction on the transition relation. In the reflexive case, we have $u = \varepsilon$, $C' = C$ and $L' = L$, and so the statement

holds trivially. Otherwise we have $\pi : \langle C, L\rangle \xrightarrow{v} \langle C'', L''\rangle \xrightarrow{w}{}^{*} \langle C', L'\rangle$, with $u = v\,w$. By our result on single transitions, we get $L'' = L + \langle v\rangle$ and $v \cdot \mathcal{L}(C'') \subseteq \mathcal{L}(C)$. By induction hypothesis, we have $L' = L'' + \langle w\rangle$ and $w \cdot \mathcal{L}(C)' \subseteq \mathcal{L}(C'')$. Therefore, using Lemma 3.2, we have $L' = L + \langle w\rangle + \langle v\rangle = L + \langle v\,w\rangle$, so the first property holds. For the second property, we have $(v\,w) \cdot \mathcal{L}(C') = v \cdot (w \cdot \mathcal{L}(C')) \subseteq v \cdot \mathcal{L}(C'') \subseteq \mathcal{L}(C)$ and are done. □

3.2 Dyck words and balanced executions

Here, we recall the notion of *Dyck words* over our lock alphabet Σ and relate them to executions of *balanced* statements. If we view ℓ and $\overline{\ell}$ respectively as opening and closing "parentheses", then Dyck words are essentially the well-parenthesized words over Σ, which can be thought of as "balanced traces". These are exactly the traces generated by executing balanced statements.

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

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

It is immediate that the traces generated by executing balanced statements are Dyck words.

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

Proof. A straightforward structural induction on C. □

The key property of Dyck words we rely on is that any occurrence of a $\overline{\ell}$ letter in a Dyck word must be matched by an earlier occurrence of ℓ, with the intervening word also a Dyck word.

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 $u = u_1\,\ell\,u_2$.*

Proof. We proceed by structural induction (over Dyck words) on $u\,\overline{\ell}\,v$.

Case $u\,\overline{\ell}\,v = \varepsilon$: This case is clearly impossible.

Case $u\,\overline{\ell}\,v = d_1\,d_2$: We distinguish two subcases. First suppose d_1 is a prefix of u. Then $u = d_1\,w$ and $d_2 = w\,\overline{\ell}\,v$ for some $w \in \Sigma^\star$. Since $w\,\overline{\ell}\,v \in \mathcal{D}$, by the induction hypothesis we obtain w_1, w_2 such that $w = w_1\,\ell\,w_2$ and $w_2 \in \mathcal{D}$. We choose $u_1 = d_1\,w_1$ and $u_2 = w_2$, and check that $u = d_1\,w = d_1\,w_1\,\ell\,w_2 = u_1\,\ell\,u_2$.

Otherwise, if d_1 is *not* a prefix of u, then it must be that u is a prefix of d_1, i.e. $d_1 = u\,w$ and $\overline{\ell}\,v = w\,d_2$ for some $w \in \Sigma^\star$. We can assume that $w \neq \varepsilon$ (since this is covered by the first case), so we have $w = \overline{\ell}\,w'$ and $v = w'\,d_2$. Since $u\,w = u\,\overline{\ell}\,w' = d_1 \in \mathcal{D}$, by the induction hypothesis we obtain u_1, u_2 such that $u = u_1\,\ell\,u_2$ and $u_2 \in \mathcal{D}$.

Case $u\,\overline{\ell}\,v = k\,d\,\overline{k}$: We observe that $u = \varepsilon$ is not possible, as it would entail $\overline{\ell}\,v = k\,d\,\overline{k}$. We distinguish two further subcases on the length of u. First, if $u = k\,d$, then we have $u\,\overline{\ell}\,v = k\,d\,\overline{\ell}\,v = k\,d\,\overline{k}$. Thus $\ell = k$ and $v = \varepsilon$, so we can choose $u_1 = \varepsilon$ and $u_2 = d$, with $u_2 \in \mathcal{D}$.

Otherwise, there must be words u', v' such that $u = k\,u'$, $v = v'\,\overline{k}$ and $d = u'\,\overline{\ell}\,v'$. In this case, by the induction hypothesis on $u'\,\overline{\ell}\,v' = d \in \mathcal{D}$ to get u'_1, u'_2 such that $u' = u'_1\,\ell\,u'_2$ and $u'_2 \in \mathcal{D}$. We then choose $u_1 = k\,u'_1$ and $u_2 = u'_2$, and check that $u = k\,u' = k\,u'_1\ell\,u'_2 = u_1\,\ell\,u_2$. □

Our mapping $\langle\cdot\rangle$ from Definition 3.1 sends all Dyck words to the empty lock state $\varnothing$, and all prefixes of Dyck words to *bona fide* lock states (as opposed to quasi-lock states).

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

Proof. A straightforward structural induction on u, using Lemma 3.2. □

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

Proof. By structural induction on uv.

Case $u\,v = \varepsilon$**:** We must have $u = \varepsilon$, and so by definition $\langle u\rangle = \varnothing \in \mathsf{Locks} \to \mathbb{N}$.

Case $u\,v = d_1\,d_2$**:** By Lemma 3.11, $\langle d_1\rangle = \langle d_2\rangle = \varnothing$. We distinguish two subcases. First suppose that d_1 is a prefix of u. Then $u = d_1\,w$ and $d_2 = w\,v$ for some $w \in \Sigma^\star$. Therefore, using Lemma 3.2, $\langle u\rangle = \langle d_1\,w\rangle = \langle d_1\rangle + \langle w\rangle = \varnothing + \langle w\rangle = \langle w\rangle$. By the induction hypothesis on $d_2 = w\,v$, we have $\langle w\rangle \in \mathsf{Locks} \to \mathbb{N}$ and are done. Otherwise, u must be a prefix of d_1, i.e. $d_1 = u\,w$ and $v = w\,d_2$ for some w, and we are done immediately by induction hypothesis on d_1.

Case $u\,v = \ell\,d\,\overline{\ell}$**:** First, if $u = \varepsilon$ then $\langle u\rangle = \varnothing$ and we are done. Next, if $v = \varepsilon$ then $u = \ell\,d\,\overline{\ell}$, and since $\langle d\rangle = \varnothing$ by Lemma 3.11, we have $\langle u\rangle = \varnothing[\ell{+}{+}][\ell{-}{-}] = \varnothing$ and are also done. Otherwise, there must be words u', v' such that $u = \ell\,u'$ and $v = v'\,\overline{\ell}$ and $d = u'\,v'$. By induction hypothesis on d, we have $\langle u'\rangle \in \mathsf{Locks} \to \mathbb{N}$. Using Lemma 3.2, we have

$$\langle u\rangle = \langle \ell\,u'\rangle = \langle \ell\rangle + \langle u'\rangle = \langle u'\rangle + \langle \ell\rangle = \langle u'\,\ell\rangle = \langle u'\rangle[\ell{+}{+}]\,.$$

Since $\langle u'\rangle \in \mathsf{Locks} \to \mathbb{N}$, so is $\langle u'\rangle[\ell{+}{+}]$. This completes the proof. □

We can now establish an analogue of Lemma 3.10 for executions of balanced statements, which will play a crucial role later on in "disentangling" concurrent executions (see Lemma 4.3).

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

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

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

Proof. Let C be a balanced statement, and $\pi = (\langle C_i, L_i\rangle)_{0\leqslant i\leqslant n+1}$ be a non-trivial execution such that $\langle C_0, L_0\rangle = \langle C, \varnothing\rangle$ and $L_{n+1} = L_n[\ell{-}{-}]$. Necessarily, $\ell \in \lfloor L_n \rfloor$. We write $\lambda(i)$ for the trace of the transition $\langle C_i, L_i\rangle \to \langle C_{i+1}, L_{i+1}\rangle$; thus clearly $\lambda(n) = \overline{\ell}$. We also write $\lambda_{i,j}$, where $j \geq i$, for the word $\lambda(i)\ldots\lambda(j-1)$, i.e. the trace of the sub-execution $\langle C_i, L_i\rangle \to^* \langle C_j, L_j\rangle$. We write $\lambda(\pi)$ for the trace of the whole execution π, meaning that $\lambda(\pi) = \lambda_{0,n+1} = \lambda_{0,n}\,\overline{\ell}$.

Since $\pi : \langle C, \varnothing\rangle \xrightarrow{\lambda(\pi)}{}^* \langle C_{n+1}, L_n[\ell{-}{-}]\rangle$, we can apply Lemma 3.7 to obtain that $\lambda(\pi)\cdot\mathcal{L}(C_{n+1}) \subseteq \mathcal{L}(C)$. Recalling that the language of a statement is never empty, let $w \in \mathcal{L}(C_{n+1})$. Since C is balanced, $\mathcal{L}(C) \subseteq \mathcal{D}$ by Lemma 3.9, meaning that $\lambda(\pi)\,w = \lambda_{0,n}\,\overline{\ell}\,w \in \mathcal{D}$. Thus by Lemma 3.10 there are words $u \in \Sigma^\star$ and $v \in \mathcal{D}$ such that $\lambda_{0,n} = \lambda(0)\ldots\lambda(n-1) = u\,\ell\,v$. This means there is an index $0 \leq j \leq n-1$ such that $\lambda_{0,j} = u$ and $\lambda(j) = \ell$ and $\lambda_{j+1,n} = v$. By Lemma 3.11, we have $\langle\lambda_{j+1,n}\rangle = \langle v\rangle = \varnothing$.

We just need to check that our j satisfies the conditions of the lemma. Clearly $j < n$. To see that $L_{j+1} = L_n$, we use Lemma 3.7 again to see that for any index i we have $L_i = \varnothing + \langle\lambda_{0,i}\rangle = \langle\lambda_{0,i}\rangle$. Therefore, using Lemma 3.2:

$$L_n = \langle\lambda_{0,n}\rangle = \langle\lambda_{0,j+1}\cdot\lambda_{j+1,n}\rangle = \langle\lambda_{0,j+1}\rangle + \langle\lambda_{j+1,n}\rangle = L_{j+1} + \varnothing = L_{j+1}\,.$$

Finally, since $\lambda(j) = \ell$, we have $L_{j+1} = L_j[\ell{+}{+}]$, so $L_n[\ell{-}{-}] = L_{j+1}[\ell{-}{-}] = L_j[\ell{+}{+}][\ell{-}{-}] = L_j$. □

The final main technical result in this section is a kind of converse to Lemma 3.7 for the case of balanced statements: for any prefix u of a word in $\mathcal{L}(C)$, we can find a corresponding execution of C with trace u.

Lemma 3.14. *Let C be a balanced statement, and let $u, v \in \Sigma^\star$ such that $u\,v \in \mathcal{L}(C)$. 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 $v \in \mathcal{L}(D)$. Furthermore, if $v = \varepsilon$, then this statement also holds when $D = \mathtt{skip}$.*

Proof. Let $u\,v \in \mathcal{L}(C)$; we proceed by structural induction on the balanced statement C. We just show the case $C = \texttt{while}(*)\ \texttt{do}\ C'$, which is the most complex since it entails constructing an execution involving arbitrarily many executions of the loop body C'.

In this case, we have by assumption $u\,v \in \mathcal{L}(C')^\star$, and consider two main subcases. First suppose $u = \varepsilon$, meaning that $v \in \mathcal{L}(C')^\star$. If also $v = \varepsilon$, then we may take as our execution $\langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \xrightarrow{\varepsilon}{}^* \langle \texttt{skip}, L\rangle$ using rule while1 and noting that $L = L + \langle\varepsilon\rangle = L + \langle u\rangle$. If $v \neq \varepsilon$, then we take instead the trivial (0-step) execution $\langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \xrightarrow{\varepsilon}{}^* \langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle$, taking $D = \texttt{while}(*)\ \texttt{do}\ C'$ and noting that $v \in \mathcal{L}(D)$ by assumption.

Otherwise, $u \neq \varepsilon$, meaning that $u = u_1 \cdots u_n\, u'$ and $v = v_1\, v_2$, where each $u_i \in \mathcal{L}(C')$ and $u'\, v_1 \in \mathcal{L}(C')$ and $v_2 \in \mathcal{L}(C')^\star$. (In other words, u is the trace of n complete iterations of the loop body C' plus some portion of the $(n+1)$th iteration.) Moreover, since C' is a balanced statement, for each i we have $u_i \in \mathcal{D}$ by Lemma 3.9 and thus $\langle u_i\rangle = \varnothing$ by Lemma 3.11. Then, for each i, by induction hypothesis on C' and $u_i\,\varepsilon \in \mathcal{L}(C')$ we obtain an execution $\langle C', L\rangle \xrightarrow{u_i}{}^* \langle \texttt{skip}, L\rangle$. We suffix these using Proposition 3.4 to modify these executions to $\langle C'; \texttt{while}(*)\ \texttt{do}\ C', L\rangle \xrightarrow{u_i}{}^* \langle \texttt{skip}; \texttt{while}(*)\ \texttt{do}\ C', L\rangle$, and add an initial while2 step and a final skip step to modify them again to $\langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \xrightarrow{u_i}{}^* \langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle$.

Next, by induction hypothesis on C' with $u'\, v_1 \in \mathcal{L}(C')$, and using Proposition 3.4 to add the suffix $\texttt{while}(*)\ \texttt{do}\ C'$, we obtain an execution $\langle C'; \texttt{while}(*)\ \texttt{do}\ C', L\rangle \xrightarrow{u'}{}^* \langle D'; \texttt{while}(*)\ \texttt{do}\ C', L + \langle u'\rangle\rangle$ where $v_1 \in \mathcal{L}(D')$, and we may assume $D' = \texttt{skip}$ if $v_1 = \varepsilon$. We then build our required execution, using the sub-executions above and an intermediate while2 step, as follows:

$$
\begin{aligned}
\pi : \langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \quad &\xrightarrow{u_1}{}^* \quad \langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \\
&\xrightarrow{u_2}{}^* \quad \langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \\
&\quad\vdots \\
&\xrightarrow{u_n}{}^* \quad \langle \texttt{while}(*)\ \texttt{do}\ C', L\rangle \\
&\xrightarrow{\varepsilon} \quad \langle C'; \texttt{while}(*)\ \texttt{do}\ C', L\rangle \\
&\xrightarrow{u'}{}^* \quad \langle D'; \texttt{while}(*)\ \texttt{do}\ C', L + \langle u'\rangle\rangle
\end{aligned}
$$

This execution indeed has trace $u = u_1\, u_2 \ldots u_n\, u'$ and, using Lemma 3.2, we have

$$L + \langle u\rangle = L + \langle u_1\rangle + \cdots + \langle u_n\rangle + \langle u'\rangle = L + \varnothing + \cdots + \varnothing + \langle u'\rangle = L + \langle u'\rangle\ .$$

Moreover, taking $D = D'; \texttt{while}(*)\ \texttt{do}\ C'$, we have

$$v = v_1\, v_2 \in \mathcal{L}(D') \cdot \mathcal{L}(C')^\star = \mathcal{L}(D'; \texttt{while}(*)\ \texttt{do}\ C') = \mathcal{L}(D)\ .$$

It just remains to treat the special case where $v = \varepsilon$. In that case $v_1 = v_2 = \varepsilon$, and by the induction hypothesis on C_1 we may take $D' = \texttt{skip}$. We may then suffix π by the following execution steps using skip and while1:

$$\langle \texttt{skip}; \texttt{while}(*)\ \texttt{do}\ C', L + \langle u\rangle\rangle \xrightarrow{\varepsilon}{}^* \langle \texttt{while}(*)\ \texttt{do}\ C', L + \langle u\rangle\rangle \xrightarrow{\varepsilon}{}^* \langle \texttt{skip}, L + \langle u\rangle\rangle\ .$$

This modified execution still has trace u and ends in skip with lock state $L + \langle u\rangle$ as required. □

Observe that Lemma 3.14 does not hold for non-balanced statements. For example, we have $\overline{\ell} \in \mathcal{L}(\texttt{rel}(\ell))$, but there are no possible executions of $\langle \texttt{rel}(\ell), \varnothing\rangle$.

A simple corollary of Lemmas 3.7 and 3.14 is that the language of a balanced statement is exactly the set of traces of its "complete" executions to skip, starting from the empty lock state.

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. For the ($\supseteq$) inclusion, suppose that $\langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle \mathtt{skip}, \varnothing\rangle$. By Lemma 3.7 we have $\{u\} \cdot \mathcal{L}(\mathtt{skip}) \subseteq \mathcal{L}(C)$. Since $\mathcal{L}(\mathtt{skip}) = \{\varepsilon\}$, this means $\{u\} \subseteq \mathcal{L}(C)$, i.e. $u \in \mathcal{L}(C)$ as required.

For the ($\subseteq$) inclusion, suppose that $u \in \mathcal{L}(C)$, i.e. $u\,\varepsilon \in \mathcal{L}(C)$. By Lemma 3.14, with $L = \varnothing$ and $v = \varepsilon$, there is an execution $\langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle \mathtt{skip}, \varnothing + \langle u\rangle\rangle$. By Lemma 3.11, $\langle u\rangle = \varnothing$, and so we have $\langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle \mathtt{skip}, \varnothing\rangle$ as required. □

Remark 3.16. *A statement C in our language can be viewed as a string acceptor 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 the strings $\mathcal{L}(C)$. Since $\mathcal{L}(C)$ is a regular language (cf. Remark 3.6), this means that our balanced statements can be viewed as* nondeterministic finite automata *(over Σ).*

4 CHARACTERISATION OF DEADLOCK EXISTENCE

In this section, we obtain our main theoretical result: the existence of a deadlock in a parallel program $C_1 \parallel C_2$ amounts to the existence of a (certain kind of) conflict between individual "summaries" of C_1 and C_2, 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, relating deadlocks to the conflict condition on critical pairs, is stated as Theorem 4.4. In the subsequent Section 6, we show how to actually compute the critical pair summary of a statement and check the conflict condition.

First, we define the critical pairs of a statement in terms of its traces.

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 trace-based definition of $\mathbf{Crit}(C)$, as opposed to an execution-based one, is that it depends only on the language $\mathcal{L}(C)$, which turns out to be easy to compute (see Section 6). The following lemma, which relies on our technical results from Section 3, gives an equivalent formulation of critical pairs in terms of executions.

Lemma 4.2. *If C is a balanced statement, then $(X, \ell) \in \mathbf{Crit}(C)$ iff there exist statements C', C'' and 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. **Case ($\Rightarrow$):** Let $(X, \ell) \in \mathbf{Crit}(C)$. We have $X = \lfloor \langle u\rangle \rfloor$ and $\ell \notin X$, where $u\,\ell\,v \in \mathcal{L}(C)$. Thus we choose $L = \langle u\rangle$, and just require to build the needed execution. By Lemma 3.14 there is a statement D and an execution

$$\langle C, \varnothing\rangle \xrightarrow{u\,\ell}{}^{*} \langle D, \varnothing + \langle u\,\ell\rangle\rangle = \langle D, \langle u\rangle[\ell{+}{+}]\rangle\ .$$

By a simple inductive analysis, we can decompose this execution as:

$$\langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle C', \langle u\rangle\rangle \xrightarrow{\ell} \langle C'', \langle u\rangle[\ell{+}{+}]\rangle \xrightarrow{\varepsilon}{}^{*} \langle D, \langle u\rangle[\ell{+}{+}]\rangle$$

which completes the case.

Case ($\Leftarrow$): Adding trace labels to the execution, we have $\langle C, \varnothing\rangle \xrightarrow{u}{}^{*} \langle C', L\rangle \xrightarrow{\ell} \langle C'', L[\ell{+}{+}]\rangle$, where $\ell \notin \lfloor L \rfloor$. By Lemma 3.7 we have $L = \varnothing + \langle u\rangle = \langle u\rangle$ and $u \cdot \mathcal{L}(C') \subseteq \mathcal{L}(C)$, and $\ell \cdot \mathcal{L}(C'') \subseteq \mathcal{L}(C')$. Therefore, picking any $v \in \mathcal{L}(C'')$, we have $u\,\ell\,v \in u \cdot (\ell \cdot \mathcal{L}(C'')) \subseteq u \cdot \mathcal{L}(C') \subseteq \mathcal{L}(C)$. Since $\ell \notin \lfloor L \rfloor$, we have $(\lfloor L \rfloor, \ell) \in \mathbf{Crit}(C)$ as required. □

Before we can characterise deadlocks in terms of critical pairs, we require one more crucial lemma. This is to "disentangle" concurrent executions by showing that it suffices to consider their sequential components individually, without having to account for all their possible interleavings.

Lemma 4.3. *Let C, C' be balanced statements such that $C \parallel C'$ does not deadlock. Then we have $\langle C \parallel C', (\varnothing, \varnothing)\rangle \rightsquigarrow^* \gamma \parallel \gamma'$ iff $\langle C, \varnothing\rangle \to^* \gamma$ and $\langle C', \varnothing\rangle \to^* \gamma'$ with $\gamma \# \gamma'$.*

Proof. The ($\Rightarrow$) direction follows immediately from Remark 2.2.

For the ($\Leftarrow$) direction, we begin by assuming executions $\pi : \langle C, \varnothing\rangle \to^* \gamma$ and $\pi' : \langle C', \varnothing\rangle \to^* \gamma'$, with $\gamma \# \gamma'$. We use the following notations for the intermediate configurations of π and π':

$$\pi = \gamma_0 \to \ldots \gamma_i \to \cdots \to \gamma_n \qquad \gamma_i = \langle C_i, L_i\rangle$$
$$\pi' = \gamma'_0 \to \ldots \gamma'_{i'} \to \cdots \to \gamma'_{n'} \qquad \gamma_{i'} = \langle C'_{i'}, L'_{i'}\rangle$$

We have $\gamma_0 = \langle C, \varnothing\rangle$ and $\gamma'_0 = \langle C', \varnothing\rangle$ and $\gamma_n = \gamma$ and $\gamma'_{n'} = \gamma'$.

In the following we call the concurrent configuration $\gamma_i \parallel \gamma'_{i'}$ *compatible* whenever $\gamma_i \# \gamma'_{i'}$, and *reachable* whenever $\gamma_0 \parallel \gamma'_0 \rightsquigarrow^* \gamma_i \parallel \gamma'_{i'}$. We shall show by induction on $i + i'$ that if $\gamma_i \parallel \gamma'_{i'}$ is compatible then it is also reachable.

If $i + i' = 0$, then $i = i' = 0$, so $\gamma_i \parallel \gamma'_{i'}$ is trivially reachable. Otherwise, $i + i' > 0$, and we consider the compatibility of the concurrent configurations $\gamma_{i-1} \parallel \gamma'_{i'}$ and $\gamma_i \parallel \gamma'_{i'-1}$. If one of these does not exist, i.e. either $i = 0$ or $i' = 0$, then the other one does exist and is compatible; e.g. if $i = 0$ then $L_i = \varnothing$ and $i' > 0$, and trivially $\varnothing \# L_{i'-1}$, i.e. $\gamma_0 \# \gamma_{i'-1}$.

Suppose first that one of these configurations is compatible, say $\gamma_i \parallel \gamma'_{i'-1}$ (the other case being symmetric). By the induction hypothesis, $\gamma_i \parallel \gamma'_{i'-1}$ is reachable. Since $\gamma'_{i'-1} \to \gamma'_{i'}$ and $\gamma_i \# \gamma'_{i'}$ by the assumption that $\gamma_i \parallel \gamma'_{i'}$ is compatible, we have $\gamma_i \parallel \gamma'_{i'-1} \rightsquigarrow \gamma_i \parallel \gamma'_{i'}$ by the rule par2 (Figure 1). Thus $\gamma_0 \parallel \gamma'_0 \rightsquigarrow^* \gamma_i \parallel \gamma'_{i'-1} \rightsquigarrow \gamma_i \parallel \gamma'_{i'}$, meaning that $\gamma_i \parallel \gamma'_{i'}$ is reachable as required.

The remaining subcase is that $i, i' > 0$ and neither $\gamma_{i-1} \parallel \gamma'_{i'}$ nor $\gamma_i \parallel \gamma'_{i'-1}$ is compatible. We will show that $C \parallel C'$ must deadlock, thus contradicting the lemma assumption. We know that $\gamma_{i-1} \to \gamma_i$ and $\gamma'_{i'-1} \to \gamma'_{i'}$. We can deduce that $L_i = L_{i-1}[\ell{--}]$ for some lock ℓ, for if not, then $\lfloor L_{i-1}\rfloor \subseteq \lfloor L_i\rfloor$ and thus $\gamma_i \parallel \gamma'_{i'}$ compatible implies $\gamma_{i-1} \parallel \gamma'_{i'}$ compatible, contradicting the subcase assumption. For a similar reason, $L'_{i'} = L'_{i'-1}[\ell'{--}]$ for some lock ℓ'. As C and C' are balanced, we can apply Lemma 3.13 (twice) to obtain $j < i$ such that $L_j = L_i$ and $L_{j+1} = L_{i-1}$, and $j' < i'$ such that $L'_{j'} = L'_{i'}$ and $L'_{j'+1} = L'_{i'-1}$. Since $\gamma_i \parallel \gamma'_{i'}$ is compatible and $L_j = L_i$ and $L_{j'} = L_{i'}$, the configuration $\gamma_j \parallel \gamma'_{j'}$ is also compatible. We have $j + j' < i + i'$, so by the induction hypothesis $\gamma_j \parallel \gamma'_{j'}$ is reachable. To complete the proof, we show that $\gamma_j \parallel \gamma'_{j'}$ is deadlocked.

Assume for contradiction that $\gamma_j \parallel \gamma'_{j'} \rightsquigarrow \sigma'$. This must be a consequence of applying one of the rules par1 and par2 from Figure 1; we assume the former, with the other case being symmetric. In that case we have $\gamma_j \to \langle D, L\rangle$ with $L \# L'_{j'}$, and $\sigma' = \langle D \parallel C'_{j'}, (L, L'_{j'})\rangle$. We know that there is a transition $\gamma_j \to \gamma_{j+1}$, and that $L_j = L_i = L_{i-1}[\ell{--}] = L_{j+1}[\ell{--}]$, meaning that $L_{j+1} = L_j[\ell{+}{+}]$. Hence C_j must be the command $\mathsf{acq}(\ell)$, and thus this transition is unique, meaning that $\langle D, L\rangle = \gamma_{j+1}$ and in particular $L = L_{j+1}$. Thus $L_{j+1} \# L'_{j'}$, and so also $L_{i-1} \# L'_{i'}$. This means that $\gamma_{i-1} \parallel \gamma'_{i'}$ is compatible. This contradicts the subcase assumption. We conclude that $\gamma_j \parallel \gamma'_{j'}$ is deadlocked after all, and since it is also reachable, the program $C \parallel C'$ deadlocks, a contradiction. □

Essentially, Lemma 4.3 implies that, when only balanced statements are involved, considerations of reachability on the concurrent transition relation $\rightsquigarrow$ can be reduced to reachability on the sequential relation $\to$.

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 $C_1 \parallel C_2$ deadlocks if and only if there are $(X_1, \ell_1) \in \mathbf{Crit}(C_1)$ and $(X_2, \ell_2) \in \mathbf{Crit}(C_2)$ such that $\ell_1 \in X_2$ and $\ell_2 \in X_1$ with $X_1 \cap X_2 = \emptyset$.*

Proof. **Case (⇒):** Suppose $C_1 \,||\, C_2$ deadlocks, meaning that some concurrent configuration $\sigma = \langle D_1 \,||\, D_2, (L_1, L_2)\rangle$ is deadlocked and $\langle C_1 \,||\, C_2, (\varnothing, \varnothing)\rangle \rightsquigarrow^* \sigma$. By Remark 2.2, we have sequential executions $\langle C_1, \varnothing\rangle \to^* \langle D_1, L_1\rangle$ and $\langle C_2, \varnothing\rangle \to^* \langle D_2, L_2\rangle$ with $L_1 \# L_2$. Since σ is deadlocked and $L_1 \# L_2$, we can apply Proposition 2.4 to obtain statements D_1', D_2' and locks ℓ_1, ℓ_2 such that

$$\langle D_1, L_1\rangle \to \langle D_1', L_1[\ell_1{+}{+}]\rangle\ ,\ \langle D_2, L_2\rangle \to \langle D_2', L_2[\ell_2{+}{+}]\rangle\ ,\ \ell_1 \in \lfloor L_2 \rfloor \ \text{ and } \ \ell_2 \in \lfloor L_1 \rfloor\ .$$

Note that we have an execution $\langle C_1, \varnothing\rangle \to^* \langle D_1, L_1\rangle \to \langle D_1', L_1[\ell_1{+}{+}]\rangle$, and moreover $\ell_1 \notin \lfloor L_1 \rfloor$, because $\ell_1 \in \lfloor L_2 \rfloor$ and $L_1 \# L_2$. Thus, by Lemma 4.2, we obtain $(\lfloor L_1 \rfloor, \ell_1) \in \mathbf{Crit}(C_1)$. A symmetric argument yields $(\lfloor L_2 \rfloor, \ell_2) \in \mathbf{Crit}(C_2)$, completing the case.

Case (⇐): Let $(X_1, \ell_1) \in \mathbf{Crit}(C_1)$ and $(X_2, \ell_2) \in \mathbf{Crit}(C_2)$, with $\ell_1 \in X_2$ and $\ell_2 \in X_1$ and $X_1 \cap X_2 = \emptyset$. By Lemma 4.2, for each $i \in \{1, 2\}$ there exist statements C_i', C_i'' and lock states L_i such 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 $\ell_i \notin X_i$.

Suppose for contradiction that $C_1 \,||\, C_2$ does *not* deadlock. Since we have $\langle C_1, \varnothing\rangle \to^* \langle C_1', L_1\rangle$ and $\langle C_2, \varnothing\rangle \to^* \langle C_2', L_2\rangle$ with $L_1 \# L_2$, we have by Lemma 4.3 that $\langle C_1 \,||\, C_2, (\varnothing, \varnothing)\rangle \rightsquigarrow^* \langle C_1' \,||\, C_2', (L_1, L_2)\rangle$. Hence $\langle C_1' \,||\, C_2', (L_1, L_2)\rangle$ cannot be deadlocked. However, because we also have

$$\langle C_1', L_1\rangle \to \langle C_1'', L_1[\ell_1{+}{+}]\rangle\ ,\ \langle C_2', L_2\rangle \to \langle C_2'', L_2[\ell_2{+}{+}]\rangle\ ,\ \ell_1 \in X_2 = \lfloor L_2 \rfloor \ \text{ and } \ \ell_2 \in X_1 = \lfloor L_1 \rfloor\ ,$$

the configuration $\langle C_1' \,||\, C_2', (L_1, L_2)\rangle$ *is* deadlocked, by Proposition 2.4. We conclude by contradiction that $C_1 \,||\, C_2$ deadlocks after all. □

We note that Theorem 4.4 immediately implies that the existence of deadlocks in our setting is decidable provided that we can compute the critical pairs of any balanced statement. In Section 6 we show that this is indeed the case.

5 GENERALISING FROM 2 TO n THREADS

In this section, we generalise our critical pair condition for deadlock existence (Theorem 4.4) to the case of parallel programs with an arbitrary (finite) number of threads.

First, we make the necessary generalisation of our two-threaded parallel programs (cf. Section 2) to $n \geq 2$ threads. That is, a *parallel program* is now an n-tuple of balanced statements written $C_1 \,||\, \ldots \,||\, C_n$, and a *concurrent configuration* is now a pair $\langle C_1 \,||\, \ldots \,||\, C_n, (L_1, \ldots, L_n)\rangle$, where $L_1, \ldots, L_n$ are lock states. We may also write concurrent configurations as $\langle C_1, L_1\rangle \,||\, \ldots \,||\, \langle C_n, L_n\rangle$, or, using a "Σ-like" notation, as $||_{1 \leq i \leq n} \langle C_i, L_i\rangle$. 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 $L' \in X$. Finally, the transition relation $\rightsquigarrow$ on concurrent configurations is now given by the following general rule for a step performed by the ith thread:

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

Concurrent executions on our n-ary concurrent configurations are then defined as before, using the above generalised version of $\rightsquigarrow$.

We have the n-ary analogues of Remark 2.2 and Proposition 2.4:

Remark 5.1. *For any concurrent execution $\gamma_1 \,||\, \ldots \,||\, \gamma_n \rightsquigarrow^* \gamma_1' \,||\, \ldots \,||\, \gamma_n'$, there exist standard executions $\gamma_i \to^* \gamma_i'$ for each $1 \leq i \leq n$. Furthermore, if $\gamma_i \# \gamma_j$, then $\gamma_i' \# \gamma_j'$; i.e., no two threads can acquire the same lock simultaneously.*

In the n-ary case, we define deadlock of a program as meaning that at least two of its threads are deadlocked. For this, it is helpful to introduce a notation for projecting a concurrent configuration onto a subset of its threads. If $\sigma = \langle C_1 \,||\, \ldots \,||\, C_n, (L_1, \ldots, L_n)\rangle$ 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 5.2 (n-ary deadlock). *A concurrent configuration $\sigma' = \langle C'_1 \mid\mid \ldots \mid\mid C'_n, (L_1, \ldots, L_n)\rangle$ is* deadlocked *if $\langle C'_i, L_i\rangle$ is live for all $1 \le i \le n$ but there is no σ'' such that $\sigma' \rightsquigarrow \sigma''$.*

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

An immediate consequence of the above definition 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 (simply by ignoring transitions in the extra threads).

Proposition 5.3. *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 $i \neq j$. 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 $1 \le i \le n$*

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

Proof. **Case ($\Rightarrow$):** By assumption, $\langle C_i, L_i\rangle$ is live for all $1 \le i \le n$, with $L_i \# L_j$ for all $j \neq i$, but there is no configuration σ' such that $\sigma \rightsquigarrow \sigma'$. Let $1 \le i \le n$; since $\langle C_i, L_i\rangle$ is live, $\langle C_i, L_i\rangle \to \langle D_i, L'_i\rangle$ for some D_i and L'_i. We show that $L'_i = L_i[\ell_1{+}{+}]$ for some lock ℓ_i. Otherwise, either $L'_i = L_i$ or $L'_i = L_i[\ell_1{-}{-}]$, and in either case, $\lfloor L'_i \rfloor \subseteq \lfloor L_i \rfloor$. Then, since $L_i \# L_j$ for all $j \neq i$ by assumption, we have $L'_i \# L_j$ for all $j \neq i$ and thus there is a concurrent transition from σ using the rule par i above, which contradicts the fact that σ is deadlocked. Thus $\langle C_i, L_i\rangle \to \langle D_i, L_i[\ell_i{+}{+}]\rangle$ as required. For the same reason, we must have $\ell_i \in \bigcup_{j\neq i} \lfloor L_j \rfloor$; otherwise, $L'_i \# \{L_j \mid j \neq i\}$ and, again, there is a concurrent transition from σ using rule par i.

Case ($\Leftarrow$): Assume that for all $1 \le i \le n$ we have $\langle C_i, L_i\rangle \to \langle D_i, L_i[\ell_1{+}{+}]\rangle$ for some D_i and ℓ_i, and $\ell_i \in \bigcup_{j\neq i} \lfloor L_j \rfloor$. We immediately have that $\langle C_i, L_i\rangle$ is live for all $1 \le i \le n$. Assume for contradiction that $\sigma \rightsquigarrow \sigma'$, say via the rule par i above. Since $\langle C_i, L_i\rangle \to \langle D_i, L_i[\ell_i{+}{+}]\rangle$, it is clear by inspection of the semantics for $\to$ that we must have $L'_i = L_i[\ell_1{+}{+}]$; only the command $\mathsf{acq}(\ell_i)$ (possibly suffixed by some other statement) can increment ℓ_i. Thus $L_i[\ell_i{+}{+}] \# \{L_j \mid j \neq i\}$, which is a contradiction since $\ell_i \in \bigcup_{j\neq i} \lfloor L_j \rfloor$ by assumption and $\ell_i \in \lfloor L_i[\ell_i{+}{+}] \rfloor$. We conclude that σ' cannot exist, and so σ is deadlocked as required. □

The following lemma is the n-ary generalisation of the crucial "disentanglement" Lemma 4.3.

Lemma 5.4. *Suppose that $C_1 \mid\mid \ldots \mid\mid C_n$ does not deadlock. Then $\langle C_1 \mid\mid \ldots \mid\mid C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^* \gamma_1 \mid\mid \ldots \mid\mid \gamma_n$ iff, for each $1 \le i \le n$, we have $\langle C_i, \varnothing\rangle \to^* \gamma_i$ with $\gamma_i \# \{\gamma_j \mid j \neq i\}$.*

Proof. The ($\Rightarrow$) direction follows immediately from Remark 5.1.

For the ($\Leftarrow$) direction, we assume for each $1 \le i \le n$ an execution $\pi_i : \langle C_i, \varnothing\rangle \to^* \gamma_i$, with $\gamma_i \# \{\gamma_j \mid j \neq i\}$. We write $\gamma_{i,j} = \langle C_{i,j}, L_{i,j}\rangle$ for the jth configuration in the execution π_i, so that in particular $\gamma_{i,0} = \langle C_i, \varnothing\rangle$.

An arbitrary concurrent configuration given by an interleaving from these n executions is then given by $\|_{1\le i\le n}\ \gamma_{i,j_i}$. We call such a configuration *compatible* to mean that $\gamma_{i,j_i} \# \gamma_{k,j_k}$ for all $k \neq i$, and *reachable* to mean that $\langle C_1 \mid\mid \ldots \mid\mid C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^* \|_{1\le i\le n}\ \gamma_{i,j_i}$. We shall show by induction on $\Sigma_{1\le i\le n}\ j_i$ that if $\|_{1\le i\le n}\ \gamma_{i,j_i}$ is compatible then it is also reachable. It then follows that $\langle C_1 \mid\mid \ldots \mid\mid C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^* \gamma_1 \mid\mid \ldots \mid\mid \gamma_n$ as required.

If $\Sigma_{1\le i\le n}\ j_i = 0$, then $j_i = 0$ for all i, so $\|_{1\le i\le n}\ \gamma_{i,0} = \langle C_1 \mid\mid \ldots \mid\mid C_n, (\varnothing, \ldots, \varnothing)\rangle$ is trivially reachable. Otherwise, when $\Sigma_{1\le i\le n} j_i > 0$, we consider the compatibility of the "preceding" configurations $\gamma_{k,j_k-1} \mid\mid \|_{1\le i\le n, i\neq k}\ \gamma_{i,j_i}$, where $1 \le k \le n$. At least one such configuration must exist, because $\Sigma_{1\le i\le n} j_i > 0$.

Suppose first that some such configuration $\gamma_{k,j_k-1} \mid\mid \mid\mid_{1\le i\le n, i\neq k} \gamma_{i,j_i}$ is compatible. By the induction hypothesis, it is also reachable. We have $\gamma_{k,j_k-1} \to \gamma_{k,j_k}$ by assumption, and $\gamma_{j,j_k} \# \bigcup_{1\le i\le n, i\neq k} \gamma_{i,j_i}$ by the assumption that $\mid\mid_{1\le i\le n} \gamma_{i,j_i}$ is compatible. Thus $\mid\mid_{1\le i\le n} \gamma_{i,j_i}$ is reachable by applying the rule par i.

The remaining subcase is that no configuration of the form $\gamma_{k,j_k-1} \mid\mid \mid\mid_{1\le i\le n, i\neq k} \gamma_{i,j_i}$ is compatible. We assume without loss of generality that these configurations are defined for the first m threads, i.e. $j_k > 0$ for $1 \le k \le m$, and undefined otherwise, meaning that $\gamma_{k,j_k} = \gamma_{k,0} = \langle C_k, \varnothing\rangle$ for all $m+1 \le k \le n$. Note that $m \ge 2$, otherwise the only configuration of the above form becomes $\gamma_{1,j_1-1} \mid\mid \mid\mid_{2\le i\le n} \langle C_i, \varnothing\rangle$, which is compatible, contrary to assumption. 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. Otherwise, $\lfloor L_{k,j_k-1}\rfloor \subseteq \lfloor L_k, j_k\rfloor$, and since $\mid\mid_{1\le i\le n} \gamma_{i,j_i}$ is compatible, so is $\gamma_{k,j_k-1} \mid\mid \mid\mid_{1\le i\le n, i\neq k} \gamma_{i,j_i}$, contradiction. By Lemma 3.10, we can find $h_k < j_k$ such that $L_{k,h_k} = L_{k,j_k}$ and $L_{k,h_k+1} = L_{k,j_k-1}$. Since $\mid\mid_{1\le i\le n} \gamma_{i,j_i}$ is compatible, and we can find such an h_k for each $1 \le k \le m$, it follows that $\mid\mid_{1\le k\le m} \gamma_{k,h_k} \mid\mid \mid\mid_{m+1\le i\le n} \gamma_{i,j_i}$ is also compatible. Since $\Sigma_{1\le k\le m} h_k + \Sigma_{m+1\le i\le n} j_i < \Sigma_{1\le i\le n} j_i$, we then have by the induction hypothesis that $\mid\mid_{1\le k\le m} \gamma_{k,h_k} \mid\mid \mid\mid_{m+1\le i\le n} \gamma_{i,j_i}$ is reachable. To conclude the proof, we shall show that $\mid\mid_{1\le k\le m} \gamma_{k,h_k}$ is deadlocked, and thus $C_1 \mid\mid \ldots \mid\mid C_n$ deadlocks with index set $I = \{1, \ldots, m\}$, which is a contradiction.

Assume for contradiction that $\mid\mid_{1\le k\le m} \gamma_{k,h_k}$ is *not* deadlocked, meaning that $\mid\mid_{1\le k\le m} \gamma_{k,h_k} \rightsquigarrow \sigma$ for some σ. This must happen via an application of the rule par i, where $i = k$ and $1 \le k \le m$. In that case we have $\gamma_{i,h_i} \to \langle D, L\rangle$ with $L \# \{L_{k,h_k} \mid k \neq i\}$. We already know that $\gamma_{i,h_i} \to \gamma_{i,h_i+1}$ and $L_{i,h_i} = L_{i,j_i} = L_{i,j_i-1}[\ell_i{-}{-}] = L_{i,h_i+1}[\ell_i{-}{-}]$, meaning that $L_{i,h_i+1} = L_{i,h_i}[\ell_i{+}{+}]$. Hence C_{i,h_i} must begin with the command $\mathsf{acq}(\ell_i)$ and so the transition $\gamma_{i,h_i} \to \gamma_{i,h_i+1}$ is unique, meaning that $\langle D, L\rangle = \gamma_{i,h_i+1}$ and in particular $L = L_{i,h_i+1} = L_{i,j_i-1}$. Thus $L_{i,j_i-1} \# \{L_{k,h_k} \mid k \neq i\}$. This means that $\gamma_{i,j_i-1} \mid\mid \mid\mid_{1\le k\le m, k\neq i} \gamma_{i,j_i}$ is compatible, which contradicts the subcase assumption. We conclude that $\mid\mid_{1\le k\le m} \gamma_{k,h_k}$ must be deadlocked after all. This completes the proof. □

Theorem 5.5 (n-ary deadlock characterisation). *A parallel program $C_1 \mid\mid \ldots \mid\mid C_n$ deadlocks if and only if, for some index set $I \subseteq \{1, \ldots, n\}$ with cardinality ≥ 2, there are critical pairs (X_i, ℓ_i) for each $i \in I$ such that $X_i \cap \bigcup_{j\neq i} X_j = \emptyset$ and $\ell_i \in \bigcup_{j\neq i} X_j$.*

Proof. **Case ($\Rightarrow$):** Suppose $C_1 \mid\mid \ldots \mid\mid C_n$ deadlocks, meaning that $\langle C_1 \mid\mid \ldots \mid\mid C_n, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^* \sigma$ for some configuration σ and, for some index set $I \subseteq \{1, \ldots, n\}$, the "projection" σ_I is deadlocked. Without loss of generality, we assume that I refers to the first m threads, i.e. $I = \{1, \ldots, m\}$, where we must have $m \ge 2$. Thus we may write $\sigma_I = \langle D_1 \mid\mid \ldots \mid\mid D_m, (L_1, \ldots, L_m)\rangle$. By Remark 5.1, 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 $L_i \# \{L_j \mid j \neq i\}$.

Since σ_I is deadlocked and $L_i \# L_j$ for all $i \neq j$, we can apply Proposition 5.3 to obtain for all $1 \le i \le m$ statements D_i' and locks ℓ_i such that $\langle D_i, L_i\rangle \to \langle D_i', L_i[\ell_i{+}{+}]\rangle$ and $\ell_i \in \bigcup_{j\neq i} \lfloor L_j\rfloor$. Thus we have executions $\langle C_i, \varnothing\rangle \to^* \langle D_i, L_i\rangle \to \langle D_i', L_i[\ell_i{+}{+}]\rangle$, and moreover $\ell_i \notin \lfloor L_i\rfloor$, because $\ell_i \in \bigcup_{j\neq i} \lfloor L_j\rfloor$ and $L_i \# \{L_j \mid j \neq i\}$. 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 $1 \le i \le m$, it is clear that all conditions are satisfied: $(X_i, \ell_i) \in \mathbf{Crit}(C_i)$ by construction; $X_i \cap \bigcup_{j\neq i} X_j = \emptyset$ because $L_i \# \{L_j \mid j \neq i\}$; and $\ell_i \in \bigcup_{j\neq i} X_j$ because $\ell_i \in \bigcup_{j\neq i} \lfloor L_j\rfloor$. This completes the case.

Case ($\Leftarrow$): We assume without loss of generality that $I = \{1, \ldots, m\}$, where $m \ge 2$. By assumption we have for each $1 \le i \le m$ a critical pair $(X_i, \ell_i) \in \mathbf{Crit}(C_i)$, with $X_i \cap \bigcup_{j\neq i} X_j = \emptyset$ and $\ell_i \in \bigcup_{j\neq i} X_j$. By Lemma 4.2, there exists for each i statements C_i', C_i'' and a lock state L_i such 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 $\ell_i \notin X_i$. Thus, in particular, $L_i \# \{L_j \mid j \neq i\}$ for each i.

We show that the program $C_1 \,||\, \ldots \,||\, C_m$ deadlocks. Suppose not, for contradiction. Since for each $1 \le i \le m$ we have $\langle C_i, \varnothing\rangle \to^* \langle C_i', L_i\rangle$ with $L_i \# \{L_j \mid j \neq i\}$, we have by Lemma 5.4 that $\langle C_1 \,||\, \ldots \,||\, C_m, (\varnothing, \ldots, \varnothing)\rangle \rightsquigarrow^* \langle C_1' \,||\, \ldots \,||\, C_m', (L_1, \ldots, L_m)\rangle$. Hence $\langle C_1' \,||\, \ldots \,||\, C_m', (L_1, \ldots, L_m)\rangle$ cannot be deadlocked. However, because we also have $\langle C_i', L_i\rangle \to \langle C_i'', L_i[\ell_i{+}{+}]\rangle$ and $\ell_i \in \bigcup_{j\neq i} \lfloor L_j \rfloor$ for each i, the configuration $\langle C_1' \,||\, \ldots \,||\, C_m', (L_1, \ldots, L_m)\rangle$ *is* deadlocked, by Proposition 5.3. We conclude by contradiction that $C_1 \,||\, C_m$ deadlocks after all, and therefore so does $C_1 \,||\, \ldots \,||\, C_n$ (because its first m threads deadlock). □

The following example illustrates the basic intuition behind our n-ary deadlock condition.

Example 5.6. We construct n sequential statements $C_1, C_2, \ldots, C_n$, as follows:

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

Observe that we have an n-ary "cycle" of critical pairs over the threads C_i, namely $(\{\ell_{(i+1) \bmod n}\}, \ell_i) \in \mathbf{Crit}(C_i)$ for each $1 \le i \le n$. It is clear that these critical pairs collectively satisfy the deadlock condition of Theorem 5.5, and indeed $C_1 \,||\, \ldots \,||\, C_n$ deadlocks, by executing the first $\texttt{acq}(-)$ command in each thread.

Conversely, any smaller collection of threads, e.g. $C_1 \,||\, \ldots \,||\, C_{n-1}$, does not satisfy the deadlock condition: the only other critical pairs of any C_i have empty LHSs, and we do not have $\ell_1 \in \{\ell_2, \ldots, \ell_n\}$. Indeed, $C_1 \,||\, \ldots \,||\, C_{n-1}$ does not deadlock, because even when $C_1, \ldots, C_{n-1}$ have all executed their first $\texttt{acq}(-)$ command, it is still possible for C_1 to acquire ℓ_1 and subsequently release ℓ_1 and ℓ_2, at which point C_2 can acquire ℓ_2 and release ℓ_2 and ℓ_3, and so on.

6 COMPUTING CRITICAL PAIRS

Having established in the previous section that the existence of a deadlock in a parallel program $C_1 \,||\, C_2$ reduces to checking a condition on $\mathbf{Crit}(C_1)$ and $\mathbf{Crit}(C_2)$ (Theorem 4.4), our first order of business in this section is to show that $\mathbf{Crit}(C)$ is in fact computable for any balanced statement C. This is not immediately obvious from Definition 4.1, since $\mathbf{Crit}(C)$ is defined there in terms of the language $\mathcal{L}(C)$, which in general is infinite.

In Section 6.1 we show that $\mathbf{Crit}(C)$ can be computed inductively, with the immediate consequence that the deadlock problem for our language is decidable and in NP (Theorem 6.6). In Section 6.2 we show that $\mathbf{Crit}(C)$ can also be computed by an abstract interpretation-style analysis, which forms the basis of our implementation and runs in worst-case exponential time (Theorem 6.11). For programs without procedure calls, the procedure runs in polynomial time.

6.1 Inductive computation

Proposition 6.1. *The following identities hold for all balanced statements C, C' and locks ℓ:*

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

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

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

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

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

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

Proof. We establish each identity in turn. We omit the proofs of the cases C1, C2 and C3 here, since they are straightforward.

Case C4**:** We have $\mathcal{L}(C;C') = \mathcal{L}(C) \cdot \mathcal{L}(C')$. We prove both inclusions separately.

($\subseteq$) Let $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C;C')$. By definition, we have $u\,\ell\,v \in \mathcal{L}(C) \cdot \mathcal{L}(C')$ and $\ell \notin \lfloor\langle u\rangle\rfloor$. We distinguish two subcases. First, suppose that $u\,\ell$ is a prefix of some word in $\mathcal{L}(C)$, meaning that $v = v_1\,v_2$ with $u\,\ell\,v_1 \in \mathcal{L}(C)$ and $v_2 \in \mathcal{L}(C')$. This immediately entails that $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C)$. Otherwise, if $u\,\ell$ is not a prefix of any word in $\mathcal{L}(C)$, we must instead have $u = u_1\,u_2$ with $u_1 \in \mathcal{L}(C)$ and $u_2\,\ell\,v \in \mathcal{L}(C')$. Since $u_1 \in \mathcal{L}(C) \subseteq \mathcal{D}$, we have $\langle u_1\rangle = \varnothing$ by Lemma 3.11. Using Lemma 3.2, we have $\langle u\rangle = \langle u_1\,u_2\rangle = \langle u_1\rangle + \langle u_2\rangle = \varnothing + \langle u_2\rangle = \langle u_2\rangle$, and thus $\ell \notin \lfloor\langle u_2\rangle\rfloor$. This entails that $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C')$. In either case, we have $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C) \cup \mathbf{Crit}(C')$ as required.

($\supseteq$) Let $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C) \cup \mathbf{Crit}(C')$. First suppose $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C)$, meaning that $u\,\ell\,v \in \mathcal{L}(C)$ for some v, and $\ell \notin \lfloor\langle u\rangle\rfloor$. Recalling that $\mathcal{L}(C')$ is nonempty, let $v' \in \mathcal{L}(C')$, so that $u\,\ell\,v\,v' \in \mathcal{L}(C) \cdot \mathcal{L}(C')$. This immediately yields $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C;C')$. Otherwise, $(\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C')$, meaning that $u\,\ell\,v \in \mathcal{L}(C')$ for some v, and $\ell \notin \lfloor\langle u\rangle\rfloor$. Let $u' \in \mathcal{L}(C)$, so that $u'\,u\,\ell\,v \in \mathcal{L}(C) \cdot \mathcal{L}(C')$. Since $u' \in \mathcal{L}(C) \subseteq \mathcal{D}$ by Lemma 3.9, we have $\langle u_1\rangle = \varnothing$ by Lemma 3.11. Using Lemma 3.2, we have $\langle u'\,u\rangle = \langle u'\rangle + \langle u\rangle = \varnothing + \langle u\rangle = \langle u\rangle$. Thus $\ell \notin \lfloor\langle u'\,u\rangle\rfloor$, and so again $(\lfloor\langle u'\,u\rangle\rfloor, \ell) = (\lfloor\langle u\rangle\rfloor, \ell) \in \mathbf{Crit}(C;C')$. This completes the case.

Case C5**:** Let us write C^n to denote n copies of C in sequence $(C;\ldots;C)$. A simple induction shows that $\mathcal{L}(C^n) = \mathcal{L}(C)^n$ for all $n > 0$. Thus

$$\mathcal{L}(\texttt{while}(*)\ \texttt{do}\ C) = \mathcal{L}(C)^\star = \bigcup_{n\geq 0} \mathcal{L}(C)^n = \{\varepsilon\} \cup \bigcup_{n>0} \mathcal{L}(C)^n = \{\varepsilon\} \cup \bigcup_{n>0} \mathcal{L}(C^n)\,.$$

Using the above, we have

$$\begin{array}{rcl} \mathbf{Crit}(\texttt{while}(*)\ \texttt{do}\ C) & = & \{(\lfloor\langle u\rangle\rfloor, \ell) \mid \exists v.\, u\,\ell\,v \in \{\varepsilon\} \cup \bigcup_{n>0} \mathcal{L}(C^n) \text{ and } \ell \notin \lfloor\langle u\rangle\rfloor\} \\ & = & \{(\lfloor\langle u\rangle\rfloor, \ell) \mid \exists v.\, u\,\ell\,v \in \bigcup_{n>0} \mathcal{L}(C^n) \text{ and } \ell \notin \lfloor\langle u\rangle\rfloor\} \\ & = & \bigcup_{n>0} \mathbf{Crit}(C^n)\,. \end{array}$$

Since $\mathbf{Crit}(C;C) = \mathbf{Crit}(C) \cup \mathbf{Crit}(C) = \mathbf{Crit}(C)$ by equation C4, it follows by induction that $\mathbf{Crit}(C^n) = \mathbf{Crit}(C)$ for all $n > 0$. Thus $\mathbf{Crit}(\texttt{while}(*)\ \texttt{do}\ C) = \bigcup_{n>0} \mathbf{Crit}(C) = \mathbf{Crit}(C)$.

Case C6**:** We have $\mathcal{L}(\texttt{acq}(\ell);C;\texttt{rel}(\ell)) = \{\ell\} \cdot \mathcal{L}(C) \cdot \{\overline{\ell}\}$. We show both inclusions.

($\subseteq$) Let $(X, \ell') \in \mathbf{Crit}(\texttt{acq}(\ell);C;\texttt{rel}(\ell))$. Thus $X = \lfloor\langle u\rangle\rfloor$ for some u, where $u\,\ell'\,v \in \{\ell\} \cdot \mathcal{L}(C) \cdot \{\overline{\ell}\}$ for some v and $\ell' \notin \lfloor\langle u\rangle\rfloor$. We distinguish two subcases. First, if $u = \varepsilon$ then $\ell' = \ell$ and $\lfloor\langle u\rangle\rfloor = \lfloor\varnothing\rfloor = \emptyset$, so $(X, \ell') = (\emptyset, \ell)$ and we are done.

Otherwise, we have $u = \ell\,u'$ and $v = v'\,\overline{\ell}$ with $u'\,\ell'\,v' \in \mathcal{L}(C)$. First, notice that by Lemma 3.12 we have $\langle u'\rangle(\ell) \geq 0$. Thus $\lfloor u\rfloor = \lfloor\langle \ell\,u'\rangle\rfloor = \lfloor\langle u'\rangle[\ell{+}{+}]\rfloor = \lfloor\langle u'\rangle\rfloor \cup \{\ell\}$. Since $\ell' \notin \lfloor\langle u\rangle\rfloor$, we have $\ell' \notin \lfloor\langle u'\rangle\rfloor$ and $\ell' \neq \ell$. Since $u'\,\ell'\,v' \in \mathcal{L}(C)$ and $\ell' \notin \lfloor\langle u'\rangle\rfloor$, we have $(\lfloor\langle u'\rangle\rfloor, \ell') \in \mathbf{Crit}(C)$. Putting everything together, we have $(X, \ell') = (\lfloor\langle u'\rangle\rfloor \cup \{\ell\}, \ell')$ with $(\lfloor\langle u'\rangle\rfloor, \ell') \in \mathbf{Crit}(C)$ and $\ell' \neq \ell$. This completes the inclusion.

($\supseteq$) Let $(X, \ell') \in \{(\emptyset, \ell)\} \cup \{(X \cup \ell, \ell') \mid (X, \ell') \in \mathbf{Crit}(C) \text{ and } \ell \neq \ell'\}$. First suppose that $(X, \ell') = (\emptyset, \ell)$. Let $v \in \mathcal{L}(C)$, so that $\varepsilon\,\ell\,v\,\overline{\ell} \in \{\ell\} \cdot \mathcal{L}(C) \cdot \{\overline{\ell}\}$. Trivially, $\ell \notin \lfloor\langle\varepsilon\rangle\rfloor = \emptyset$, and thus $(\emptyset, \ell) \in \mathbf{Crit}(\texttt{acq}(\ell);C;\texttt{rel}(\ell))$ as required.

Otherwise, we have $X = X' \cup \{\ell\}$ where $(X', \ell') \in \mathbf{Crit}(C)$ and $\ell \neq \ell'$. In this case, we have words u', v' such that $X' = \lfloor\langle u'\rangle\rfloor$ and $u'\,\ell'\,v' \in \mathcal{L}(C)$ and $\ell' \notin X'$. We observe that $\ell\,u'\,\ell'\,v'\,\overline{\ell} \in \{\ell\} \cdot \mathcal{L}(C) \cdot \{\overline{\ell}\}$, and, by Lemma 3.12, we have $\langle u'\rangle(\ell) \geq 0$. Thus, using Lemma 3.2, $\lfloor\langle \ell\,u'\rangle\rfloor = \lfloor\varnothing[\ell{+}{+}] + \langle u'\rangle\rfloor = \lfloor\langle u'\rangle\rfloor \cup \{\ell\} = X$. Since $\ell' \notin X'$ and $\ell' \neq \ell$, we also have $\ell' \notin X$, and so again $(X, \ell') \in \mathbf{Crit}(\texttt{acq}(\ell);C;\texttt{rel}(\ell))$. □

We begin our complexity analysis by defining suitable notions of *size* for statements and sets of critical pairs. We write $\#X$ for the cardinality of a finite set X.

Definition 6.2. *For any statement C we define its* size $\|C\|$ *by*

$$\|C\| := |C| + \textstyle\sum_{p \in \mathrm{callees}(C)} |\mathrm{body}(p)|,$$

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

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

Note that, if C does not make any procedure calls, i.e. $\mathrm{callees}(C) = \emptyset$, then $\|C\| = |C|$. Moreover, since every statement has non-zero size (by definition), $\|C\| \geq 1 + \#\mathrm{callees}(C)$.

Proposition 6.3. *For any balanced statement C, the set $\mathbf{Crit}(C)$ is finite and computable, with $\#\mathbf{Crit}(C) \leq \|C\|^{1+\#\mathrm{callees}(C)}$ and $\#X < \|C\|$ for all $(X, \ell) \in \mathbf{Crit}(C)$. In particular, if C does not contain any procedure calls, then $\#\mathbf{Crit}(C) \leq |C|$.*

Proof. We proceed by structural induction on C, making use of the equations in Proposition 6.1..

Case $C = \texttt{skip}$**:** Trivial, since $\mathbf{Crit}(C) = \emptyset$ by C1.

Case $C = p()$**:** For the first property, using C2 and the induction hypothesis, we have

$$\#\mathbf{Crit}(p()) = \#\mathbf{Crit}(\mathrm{body}(p)) \leq \|\mathrm{body}(p)\|^{1+\#\mathrm{callees}(\mathrm{body}(p))} = (\|p()\|-1)^{\#\mathrm{callees}(p())} \leq \|p()\|^{1+\#\mathrm{callees}(p())}.$$

For the second property, letting $(X, \ell) \in \mathbf{Crit}(C)$, we have by induction hypothesis $\#X < \|\mathrm{body}(p)\| < \|p()\|$ as required.

Case $C \in \{C_1; C_2, \texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2\}$**:**. In both these cases we have $\mathrm{callees}(C) = \mathrm{callees}(C_1) \cup \mathrm{callees}(C_2)$, and $\mathbf{Crit}(C) = \mathbf{Crit}(C_1) \cup \mathbf{Crit}(C_2)$ by C3/C4. Thus, for the first property, we have

$$\begin{aligned}\#\mathbf{Crit}(C) &\leq \#\mathbf{Crit}(C_1) + \#\mathbf{Crit}(C_2)\\ &\leq \|C_1\|^{1+\#\mathrm{callees}(C_1)} + \|C_2\|^{1+\#\mathrm{callees}(C_2)}\\ &\leq \|C_1\|^{1+\#\mathrm{callees}(C)} + \|C_2\|^{1+\#\mathrm{callees}(C)} \leq \|C\|^{1+\#\mathrm{callees}(C)}.\end{aligned}$$

For the second property, letting $(X, \ell) \in \mathbf{Crit}(C)$, we have $(X, \ell) \in \mathbf{Crit}(C_i)$ for some $i \in \{1, 2\}$, and thus by induction hypothesis $\#X < \|C_i\| < \|C\|$.

Case $C = \texttt{while}(*)\ \texttt{do}\ C'$**:** This case holds immediately by induction hypothesis, since $\|C\| = \|C'\|$, $\mathrm{callees}(C) = \mathrm{callees}(C')$ and $\mathbf{Crit}(C) = \mathbf{Crit}(C')$ by C5.

Case $C = \texttt{acq}(\ell); C'; \texttt{rel}(\ell)$**:** We have $|C| = |C'| + 2$ and $\mathrm{callees}(C) = \mathrm{callees}(C')$, and, by C6:

$$\mathbf{Crit}(C) = \{(\emptyset, \ell)\} \cup \{(X \cup \{\ell\}, \ell') \mid (X, \ell') \in \mathbf{Crit}(C) \text{ and } \ell \neq \ell'\}.$$

For the first property, we have

$$\#\mathbf{Crit}(C) \leq 1 + \#\mathbf{Crit}(C') \leq 1 + \|C'\|^{1+\#\mathrm{callees}(C')} \leq \|C\|^{1+\#\mathrm{callees}(C')} = \|C\|^{1+\#\mathrm{callees}(C)}.$$

For the second property, let $(X, \ell') \in \mathbf{Crit}(C)$. If $X = \emptyset$ then the bound holds trivially. Otherwise $X = X' \cup \{\ell\}$ with $(X', \ell') \in \mathbf{Crit}(C')$, and using the induction hypothesis we have $\#X \leq 1 + \#X' < 1 + \|C'\| < \|C\|$.

This completes the induction. The case when C does not contain any procedure calls then follows immediately from the general case by taking $\mathrm{callees}(C) = \emptyset$ and $\|C\| = |C|$. □

The following examples illustrate these bounds.

Example 6.4. Let $\ell_1, \ldots, \ell_n$ be distinct locks, and consider the following balanced statement C:

$$\mathtt{acq}(\ell_1); \mathtt{acq}(\ell_2); \ldots \mathtt{acq}(\ell_n); \mathtt{skip}; \mathtt{rel}(\ell_n) \ldots; \mathtt{rel}(\ell_2); \mathtt{rel}(\ell_1)$$

We have $|C| = 2n + 1$ and, using equations C6 and C1, a simple induction establishes that

$$\mathbf{Crit}(C) = \textstyle\bigcup_{1 \leq i \leq n} \{(\{\ell_1, \ldots, \ell_{i-1}\}, \ell_i)\} \ .$$

Thus $\mathbf{Crit}(C)$ contains n elements, with the largest being $(\{\ell_1, \ldots, \ell_{n-1}\}, \ell_n)$.

Example 6.5. Let $\ell_1, \ldots, \ell_n$ be distinct locks, and consider the following procedures $p_1, \ldots, p_n$:

$$\mathrm{body}(p_1) = \mathtt{if}(*)\ \mathtt{then}\ \mathtt{acq}(\ell_i); \mathtt{skip}; \mathtt{rel}(\ell_i)\ \mathtt{else}\ \mathtt{skip}$$
$$\mathrm{body}(p_i) = \mathtt{if}(*)\ \mathtt{then}\ \mathtt{acq}(\ell_i); p_{i-1}(); \mathtt{rel}(\ell_i)\ \mathtt{else}\ p_{i-1}() \quad \text{for each } 2 \leq_< i \leq n \ .$$

For each $1 \leq i \leq n$, we have that $\mathrm{callees}(p_i) = \{p_j \mid 1 \leq j < i\}$, and a straightforward induction on i establishes that:

$$\mathbf{Crit}(\mathrm{body}(p_i)) = \{(X, \ell_k) \mid 1 \leq k \leq i \text{ and } X \subseteq \{\ell_{k+1}, \ldots, \ell_i\}\} \ .$$

The size of $C = \mathrm{body}(p_n)$ can be computed as follows:

$$\|C\| = |\mathrm{body}(p_n)| + \textstyle\sum_{p \in \mathrm{callees}(C)} |\mathrm{body}(p)| = \sum_{1 \leq i \leq n} |\mathrm{body}(p_i)| = 4n.$$

Now $\mathbf{Crit}(C) = \{(X, \ell_k) \mid 1 \leq k \leq n \text{ and } X \subseteq \{\ell_{k+1}, \ldots, \ell_n\}\}$ is in bijection with the set of non-empty subsets of $\{1, \ldots, n\}$. Given $S \subseteq \{1, \ldots, n\}$, let m be its minimum; then the critical pair $(\{\ell_i \mid i \in S \setminus \{m\}\}, \ell_m)$ belongs to $\mathbf{Crit}(C)$ (and vice-versa). Therefore, C has $2^{\frac{1}{4}\|C\|} - 1$ critical pairs, and some of them, e.g. the pair $(\{\ell_2, \ldots, \ell_n\}, \ell_1)$, are of linear size in $\|C\|$.

In order 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 constituent threads: $\|(C_1 \,||\, \ldots \,||\, C_n)\| = \sum_{1 \leq i \leq n} \|C_i\|$.

Theorem 6.6. *Whether a given parallel program deadlocks or not is decidable, and in* NP.

Proof. First we establish decidability. By Theorem 4.4, $C_1 \,||\, \ldots \,||\, C_n$ deadlocks iff, for some index set $I \subseteq \{1, \ldots, n\}$ with $|I| \geq 2$, there are critical pairs $(X_i, \ell_i) \in \mathbf{Crit}(C_i)$ for each $i \in I$ such that $X_i \cap \bigcup_{j \neq i} X_j = \emptyset$ and $\ell_i \in \bigcup_{j \neq i} X_j$. By Proposition 6.3, $\mathbf{Crit}(C_i)$ is finite and computable for any C_i. Therefore, deciding the latter condition can be done by checking all possible sets of critical pairs for all possible index sets.

The NP upper bound relies on the observation that we can use the equations in Proposition 6.1 to nondeterministically compute an arbitrary critical pair of any C_i in polynomial time (in $\|C_i\| \leq \|P\|$). Specifically, we can write a program that recurses on the structure of C_i and selects a critical pair by nondeterministically deciding which "branch" of the computation given by C1-C6 to follow at each stage. Such a program clearly runs in polynomial time in $\|C_i\|$.

Therefore, the NP procedure runs in three stages: (i) nondeterministically select an index set $I \subseteq \{1, \ldots, n\}$ of size ≥ 2; (ii) nondeterministically select a critical pair (X_i, ℓ_i) for each $i \in I$, as above; (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 6.3, each X_i is of size bounded by $\|C_i\|$. □

Remark 6.7. *An immediate consequence of Proposition 6.1 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:*

$$\mathtt{if}(*)\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2 \ \mapsto\ C_1; C_2 \quad \textit{and} \quad \mathtt{while}(*)\ \mathtt{do}\ C' \ \mapsto\ C' \ .$$

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

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

$$
\begin{array}{rcll}
[\![\texttt{acq}(\ell)]\!]\langle L, Z\rangle & = & \langle L[\ell{+}{+}], Z \cup Z'\rangle & \text{where } Z' = \begin{cases} \{(\lfloor L \rfloor, \ell)\} & \text{if } L(\ell) = 0 \\ \emptyset & \text{if } L(\ell) > 0 \end{cases} \\
[\![\texttt{rel}(\ell)]\!]\langle L, Z\rangle & = & \langle L[\ell{-}{-}], Z\rangle & \\
[\![p()]\!]\langle L, Z\rangle & = & \langle L, Z \cup Z'\rangle & \text{where } \langle \_, Z''\rangle = [\![\text{body}(p)]\!]\alpha_\perp \text{ in} \\
 & & & Z' = \{(\lfloor L \rfloor \cup M, \ell) \mid (M, \ell) \in Z'' \wedge L(\ell) = 0\}
\end{array}
$$

$$
\begin{array}{rcllrcl}
[\![\texttt{skip}]\!]\alpha & = & \alpha & & [\![C_1; C_2]\!]\alpha & = & [\![C_2]\!]([\![C_1]\!]\alpha) \\
[\![\texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2]\!]\alpha & = & ([\![C_1]\!]\alpha) \sqcup ([\![C_2]\!]\alpha) & & [\![\texttt{while}(*)\ \texttt{do}\ C]\!]\alpha & = & \bigsqcup_{n=0}^{\infty} [\![C]\!]^n \alpha
\end{array}
$$

Fig. 2. Abstract analysis definition.

6.2 Abstract interpretation-style computation

We now define an alternative way to compute the critical pairs of a statement, in abstract interpretation style. The rationale is that this style of computation, rather than the direct inductive computation given in the previous section, is the one that forms the basis of our implementation. The main idea is that, given any statement C, we define an analysis function $[\![C]\!](\cdot)$ on *abstract states*, which essentially track the lock state and the set of critical pairs accumulated during the possible executions of C.

Definition 6.8. *An* abstract state *of our analysis is a pair $\langle L, Z\rangle$, where L is a lock state and $Z \subseteq 2^{\text{Locks}} \times \text{Locks}$ (i.e. a set of pairs each comprising a set of locks and a single lock). 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 to range over abstract states, and $\alpha_\perp$ for the "empty" abstract state $\langle \varnothing, \emptyset\rangle$.

The function $[\![C]\!](\cdot)$ is then defined by structural induction on C in Figure 2. We remark that the clauses for the control flow statement (`if`, `while` and sequencing) 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.

Proposition 6.9. *For any balanced statement C and abstract state $\alpha = \langle L, Z\rangle$,*

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

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

Proof. We proceed by structural induction on C. The cases $C = \texttt{skip}$, $C = p()$, $C = C_1; C_2$ and $C = \texttt{if}(*)\ \texttt{then}\ C_1\ \texttt{else}\ C_2$ are straightforward using the induction hypothesis and the equations C1–C4. The case $C = \texttt{while}(*)\ \texttt{do}\ D$ is similarly straightforward once one notices that, writing D^n for $n > 0$ copies of D in sequence $(D; \ldots; D)$, we have $[\![D]\!]^n\alpha = [\![D^n]\!]\alpha$ by definition and $\mathbf{Crit}(D^n) = \mathbf{Crit}(\texttt{while}(*)\ \texttt{do}\ D)$ by equations C4 and C5. As a consequence, we in fact have $[\![\texttt{while}(*)\ \texttt{do}\ D]\!]\alpha = \alpha \sqcup [\![D]\!]\alpha$, meaning that a computation of $\bigsqcup_{n=0}^{\infty} [\![D]\!]^n\alpha$ will reach a fixed point after a single application of $[\![D]\!]$.

We show only the locking case, $C = \mathtt{acq}(\ell); C'; \mathtt{rel}(\ell)$, in detail. First notice that $\lfloor L[\ell{+}{+}] \rfloor = \lfloor L \rfloor \cup \{\ell\}$, and that $L[\ell{+}{+}](\ell') = 0$ iff $L(\ell') = 0$ and $\ell \neq \ell'$. Then, using the induction hypothesis on C', we have:

$$
\begin{aligned}
&[\![\mathtt{acq}(\ell); C'; \mathtt{rel}(\ell)]\!]\alpha \\
&= [\![\mathtt{rel}(\ell)]\!][\![C']\!][\![\mathtt{acq}(\ell)]\!]\alpha \\
&= [\![\mathtt{rel}(\ell)]\!][\![C']\!]\langle L[\ell{+}{+}], Z \cup Z' \rangle \qquad \text{with } Z' = \begin{cases} \{(\lfloor L \rfloor, \ell)\} & \text{if } L(\ell) = 0 \\ \emptyset & \text{if } L(\ell) > 0 \end{cases} \\
&= [\![\mathtt{rel}(\ell)]\!]\langle L[\ell{+}{+}], Z \cup Z' \cup \{(\lfloor L[\ell{+}{+}] \rfloor \cup X, \ell') \mid (X, \ell') \in \mathbf{Crit}(C') \text{ and } L[\ell{+}{+}](\ell') = 0\} \rangle \\
&= \langle L[\ell{+}{+}][\ell{-}{-}], Z \cup Z' \cup \{(\lfloor L[\ell{+}{+}] \rfloor \cup X, \ell') \mid (X, \ell') \in \mathbf{Crit}(C') \text{ and } L[\ell{+}{+}](\ell') = 0\} \rangle \\
&= \langle L, Z \cup Z' \cup \{(\lfloor L \rfloor \cup X \cup \{\ell\}, \ell') \mid (X, \ell') \in \mathbf{Crit}(C') \text{ and } L(\ell') = 0 \text{ and } \ell' \neq \ell\} \rangle
\end{aligned}
$$

Note that Z' can be re-expressed as $\{(\lfloor L \rfloor \cup \emptyset, \ell) \mid L(\ell) = 0\}$. Recall identity C6:

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

We can thus conclude this case, and the induction, by rewriting the last set expression above:

$$
\begin{aligned}
&Z \cup Z' \cup \{(\lfloor L \rfloor \cup X \cup \{\ell\}, \ell') \mid (X, \ell') \in \mathbf{Crit}(C') \text{ and } L(\ell') = 0 \text{ and } \ell \neq \ell'\} \\
&= Z \cup \{(\lfloor L \rfloor \cup \emptyset, \ell) \mid L(\ell) = 0\} \cup \{(\lfloor L \rfloor \cup X \cup \{\ell\}, \ell') \mid (X, \ell') \in \mathbf{Crit}(C') \text{ and } L(\ell') = 0 \text{ and } \ell \neq \ell'\} \\
&= Z \cup \{(\lfloor L \rfloor \cup X, \ell') \mid (X, \ell') \in \mathbf{Crit}(\mathtt{acq}(\ell); C'; \mathtt{rel}(\ell)) \text{ and } L(\ell') = 0\} \ .
\end{aligned}
$$

Finally, for the case $\alpha = \alpha_\perp = \langle \varnothing, \emptyset \rangle$, recalling that $\lfloor \varnothing \rfloor = \emptyset$ and $\varnothing(\ell) = 0$ for all locks ℓ, we obtain

$$[\![C]\!]\alpha_\perp = \langle \varnothing, \emptyset \cup \{(\lfloor \varnothing \rfloor \cup X, \ell) \mid (X, \ell) \in \mathbf{Crit}(C) \text{ and } \varnothing(\ell) = 0\} \rangle = \langle \varnothing, \mathbf{Crit}(C) \rangle \ . \qquad \square$$

Lemma 6.10. *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. Let us define the size of a (finite) set P of critical pairs as $|P| = \#P + \sum_{(X,\ell)\in P} \#X$. From the definition of the analysis given in Figure 2, and the remarks made in the proof of Proposition 6.9, it is apparent that the computation of $[\![C]\!]\alpha_\perp$ is linear in the size of $\mathbf{Crit}(C)$, its result. Therefore, we can obtain the time bound from the bounds on $\mathbf{Crit}(C)$ given by Proposition 6.3. In the general case, we have:

$$
\begin{aligned}
|\mathbf{Crit}(C)| &= \#\mathbf{Crit}(C) + \textstyle\sum_{(X,\ell)\in\mathbf{Crit}(C)} \#X \\
&\leq \#\mathbf{Crit}(C) \times (1 + \max\{\#X \mid (X, \ell) \in \mathbf{Crit}(C)\}) \\
&\leq \|C\|^{1+\#\mathrm{callees}(C)} \times (1 + \|C\| - 1) \qquad \text{(by Prop.6.3)} \\
&\leq \|C\|^{2+\#\mathrm{callees}(C)} \\
&\leq \|C\|^{1+\|C\|}.
\end{aligned}
$$

In the procedure-free case, we get:

$$
\begin{aligned}
|\mathbf{Crit}(C)| &= \#\mathbf{Crit}(C) + \textstyle\sum_{(X,\ell)\in\mathbf{Crit}(C)} \#X \\
&\leq \#\mathbf{Crit}(C) \times (1 + \max\{\#X \mid (X, \ell) \in \mathbf{Crit}(C)\}) \\
&\leq |C|^2 \ .
\end{aligned}
$$

$\square$

Theorem 6.11. *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. The decision algorithm consists in computing all critical pairs of the threads C_i and checking, for each possible index set $I \subseteq \{1, \ldots, n\}$ with $\#I \geq 2$, there are critical pairs (X_i, ℓ_i) for each $i \in I$ that collectively satisfy the deadlock condition of Theorem 5.5, namely that $X_i \cap \bigcup_{j \neq i} X_j = \emptyset$ and $\ell_i \in \bigcup_{j \neq i} X_j$ for all $i \in I$.

The first stage of the algorithm, i.e. computing $\mathbf{Crit}(C_1), \ldots, \mathbf{Crit}(C_n)$, can be performed in exponential time in $\|P\|$, by Lemma 6.10. Noting that for any critical pair $(X_i, \ell_i) \in \mathbf{Crit}(C_i)$ we have $\#X \leq \|C_i\|$ by Proposition 6.3, it is clear that any given set of $\leq n$ critical pairs can be checked in polynomial time in $\|P\|$. There are roughly 2^n possible index sets I, and in the worst case, the sets $\mathbf{Crit}(C_1), \ldots, \mathbf{Crit}(C_n)$ contain exponentially many critical pairs in $\|C_1\|, \ldots, \|C_n\|$ respectively, by Proposition 6.3. Therefore the number of critical pair sets to be checked is exponential in $\|P\|$ and n. This yields an overall time bound exponential in $\|C_1\|, \ldots, \|C_n\|$ and n.

However, if P does not contain any procedure calls, then Lemma 6.10 and Proposition 6.3 instead tells us that $\mathbf{Crit}(C_1), \ldots, \mathbf{Crit}(C_n)$ can be computed in *quadratic* time and contain at most a *linear* number of critical pairs, in $\|C_1\|, \ldots, \|C_n\|$ respectively. In that case, the argument above instead yields a time bound polynomial in $\|P\|$ (but still exponential in n). □

7 IMPLEMENTATION AND IMPACT

We have implemented a compositional program analyser based on the analysis presented in Figure 2 for the Java language. The flow-sensitive, context-insensitive analysis is developed in OCaml (around 3kLoC) within the Infer static analysis framework [Distefano et al. 2019], and is specifically targeted at detecting *2-thread deadlocks in code changes (commits) of Android apps* within a continuous integration environment.[2] In this section we discuss this implementation and its impact at Facebook. We note that we do not attempt an experimental evaluation with other tools, since the deployment constraints (completeness versus soundness, code changes versus batch mode) lead to very different design trade-offs, making a comparison on previously used benchmarks unhelpful.

7.1 Deployment and impact

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 inserting comments on the code, based on the potential bugs found. *The implementation of 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. In addition, theoretical veracity is not always correlated with actionability. For example, a report is sometimes rendered an effective false positive by un-written invariants. For these reasons, fixed reports (reports that code authors addressed by submitting a new version of a commit) rather than true positives are tracked. *Since it was deployed, the deadlock analyser has processed a total of 667k commits, has issued a total of 479 deadlock reports, and has seen a total of 260 fixes, yielding a fix rate of 54%.*

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 has seen a median of 90 seconds and an average of 213 seconds per commit. In the same time period, Infer analysed a median of 1.9k methods and on average 4.5k methods per commit.*

[2]The analyser (named `starvation`) is open-source code included in Infer, see https://fbinfer.com/.

7.2 Issues and differences between implementation and theory

The requirement that the analysis targets code changes leads to a number of design decisions, 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.

Balanced locking. Our analysis relies on balanced locking while Java allows unbalanced lock operations. Language support for balanced locking (via the **`synchronized`** keyword) and good-programming practices mean that the code-base targeted by our implementation has very few instances of unbalanced locking, and thus analysis precision does not suffer.

Non-deterministic control. Control in Java is mostly deterministic, so our abstract semantics is over-approximate. Most of the imprecision we observed in practice came from branching over two conditions: whether a lock acquisition succeeded (e.g., via `Lock.trylock`), or whether the current thread is the UI thread. We specialised the domain, introducing partial path sensitivity on these conditions, thus removing the vast majority of false alarms 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 [Jones and Muchnick 1979]. 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 we cannot do a whole-program analysis, 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 is executed on the UI thread as well as background threads). This information comes from (a) thread annotations used in Android code such as `@UiThread` and `@WorkerThread`; (b) Android method calls that determine whether the current thread is the UI thread; (c) upward propagation through the call graph. Each critical pair in a method summary is decorated with the inferred thread identity, and this information is used to determine whether 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

```
class A {
  public synchronized void foo(B b) {
    b.foo();
  }

  public synchronized void bar() {}
}
```

```
class B {
  public synchronized void bar(A a) {
    a.bar();
  }

  public synchronized void foo() {}
}
```

Fig. 3. Textbook deadlock across two Java classes.

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 $(L, \texttt{root}.f_1.\ldots.f_n)$ 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 $\texttt{root}.f_1.\ldots.f_n \in L'$). 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 (i) the **this** object, such as when using synchronised methods, or, (ii) immutable private objects stored in object fields. For instance, this heuristic will catch the deadlock between classes A and B in Figure 3 even in a commit where only A.foo is modified.

This clearly introduces the possibility of a false negative, e.g. when global locks are acquired in methods that reside in classes not containing the globals. The use of locks in global variables is, relatively, much rarer than locking non-global objects in the code we usually analyse. The heuristic also allows for false positives, since it does not check for evidence of thread spawning. To counter this effect, we use thread identity information as detailed above, and this reduces the incidence of such false positives in correlation with developer use of such mechanisms.

8 RELATED WORK

Deadlock detectors are naturally distinguished into dynamic, static and 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 analyses for C code do not expect balanced locking and assume non-reentrant locks. In addition, deadlock analyses can be categorised according to whether they detect deadlocks involving two, or more threads, and whether they produce false positives on guarded cycles.

8.1 Static analyses

Deadlock detectors operating on program sources typically require a complete program, though they can operate without test inputs. Most are focused on soundness (where absence of reports implies deadlock freedom). All analyses discussed are interprocedural, top-down, context-sensitive and typically non-compositional.

RacerX [Engler and Ashcraft 2003] 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.

[Williams et al. 2005] 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 [Naik et al. 2009] 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.

[Pun et al. 2014] 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 [Atkey and Sannella 2015] 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, for scalability.

[Kroening et al. 2016] 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 [Laneve and Garcia 2018] reports on a Java bytecode analysis which uses behavioural type rules for compositionally extracting an infinite-state abstract model from bytecode, and then analysing that with a context-sensitive fixpoint computation, generating reports of cyclic dependencies. The strength of the approach seems to be the ability to analyse recursive functions that spawn an unbounded number of threads.

8.2 Dynamic and hybrid analyses

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

GoodLock [Havelund 2000] is an analysis for Java programs implemented in Java PathFinder (JPF) [Havelund and Pressburger 2000] 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.

[Bensalem and Havelund 2006] 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.

[Agarwal et al. 2006] 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 [Eslamimehr and Palsberg 2014] is an analysis for Java programs which uses GoodLock to get a set of deadlock candidates. Using given program inputs, the program is then run, 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.

9 CONCLUSIONS AND FUTURE WORK

In this paper we establish the decidability of the deadlock problem, and provide an open-source deadlock analyser, for an abstract programming language with scoped (or "balanced") locking, nondeterministic control and nonrecursive procedure calls, but in which all other features — in particular variable assignment — are abstracted away. Such an abstraction is of course necessary to obtain decidability, for fundamental computability reasons. However, this overapproximation of real concurrent programs turns out to be sufficiently faithful to detect deadlock bugs in practice, and sufficiently scalable to run on real-world industrial codebases. 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 fix rate of over 50% (and we note that this does not imply a false positive rate of nearly 50%).

One interesting connection that could benefit from further elucidation is that of our work to automata-theoretic work on concurrent verification. We have already observed (cf. Remark 3.16) that our parallel programs can be seen as collections of (particular kinds of) nondeterministic finite automata that synchronise via their shared locks. Other work on automata-based concurrent systems, e.g. [Bouajjani et al. 2003; Esparza et al. 2013; Hague 2011; Heußner et al. 2010; Zielonka 1987] typically uses a slightly different synchronisation mechanism based on communication "visible", or external actions must be fired simultaneously by two automata. Nevertheless, our model can be polynomially encoded as a *communicating pushdown system*, as considered e.g. in [Bouajjani et al. 2003], which presents a high-level approach to analysing general safety properties of such systems. Thus our deadlock problem can be seen as an instance of the general class of problems considered there, and we cannot rule out the possibility that the decidability of our deadlock problem follows from some more general automata-theoretic result in the literature. However, while typical dataflow properties of arbitrary communicating pushdown systems are undecidable in general, the deadlock property we consider for our particular class of programs is decidable and thus represents a special case, which relies crucially on the fact that locking in our language is balanced. Our proof also has the added advantage of being direct. That is, we treat (abstract) programs rather than automata, our critical pair abstraction is specialised to the deadlock problem and our computation of this abstraction forms the basis of our automated deadlock detection tool.

It is natural to wonder whether and how our abstract programming 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 does not seem to drastically alter our language, since we already allow iteration, but it causes technical problems for our approach since we cannot then 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.

REFERENCES

Rahul Agarwal, Liqiang Wang, and Scott D. Stoller. 2006. Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In *Hardware and Software, Verification and Testing*, Shmuel Ur, Eyal Bin, and Yaron Wolfsthal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 191–207.

Robert Atkey and Donald Sannella. 2015. ThreadSafe: Static Analysis for Java Concurrency. *ECEASST* 72 (2015). https://doi.org/10.14279/tuj.eceasst.72.1025

Saddek Bensalem and Klaus Havelund. 2006. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Hardware and Software, Verification and Testing*, Shmuel Ur, Eyal Bin, and Yaron Wolfsthal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 208–223.

Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. 2003. A Generic Approach to the Static Analysis of Concurrent Programs with Procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/604131.604137

N. Chomsky and M.P. Schützenberger. 1963. The Algebraic Theory of Context-Free Languages. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 35. Elsevier, 118 – 161. https://doi.org/10.1016/S0049-237X(08)72023-8

E.W. Dijkstra. 1971. Hierarchical ordering of sequential processes. *Acta Informatica* 1, 2 (1971), 115–138.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70. https://doi.org/10.1145/3338112

Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: Scalable Deadlock Detection for Concurrent Programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 353–365. https://doi.org/10.1145/2635868.2635918

Javier Esparza, Pierre Ganty, and Rupak Majumdar. 2013. Parameterized Verification of Asynchronous Shared-Memory Systems. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–140.

Matthew Hague. 2011. Parameterised Pushdown Systems with Non-Atomic Writes. *Leibniz International Proceedings in Informatics, LIPIcs* 13 (09 2011). https://doi.org/10.4230/LIPIcs.FSTTCS.2011.457

Klaus Havelund. 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 245–264.

Klaus Havelund and Thomas Pressburger. 2000. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer* 2, 4 (2000), 366–381. https://doi.org/10.1007/s100090050043

Alexander Heußner, Jérôme Leroux, Anca Muscholl, and Grégoire Sutre. 2010. Reachability Analysis of Communicating Pushdown Systems. In *Foundations of Software Science and Computational Structures*, Luke Ong (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 267–281.

John E. Hopcroft and Jeffrey D. Ullman. 1969. *Formal Languages and Their Relation to Automata.* Addison-Wesley Longman Publishing Co., Inc., USA.

Neil D. Jones and Steven S. Muchnick. 1979. Flow Analysis and Optimization of LISP-like Structures. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*. Association for Computing Machinery, New York, NY, USA, 244–256. https://doi.org/10.1145/567752.567776

D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. 2016. Sound static deadlock analysis for C/Pthreads. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 379–390.

Cosimo Laneve and Abel Garcia. 2018. Deadlock Detection of Java Bytecode. In *Logic-Based Program Synthesis and Transformation*, Fabio Fioravanti and John P. Gallagher (Eds.). Springer International Publishing, Cham, 37–53.

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective Static Deadlock Detection. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

Ka I Pun, Martin Steffen, and Volker Stolz. 2014. Deadlock checking by data race detection. *Journal of Logical and Algebraic Methods in Programming* 83, 5 (2014), 400 – 426. https://doi.org/10.1016/j.jlamp.2014.07.003 The 24th Nordic Workshop on Programming Theory (NWPT 2012).

Amy Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *ECOOP 2005 - Object-Oriented Programming*, Andrew P. Black (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 602–629.

Wieslaw Zielonka. 1987. Notes on finite asynchronous automata. *RAIRO - Theoretical Informatics and Applications* 21, 2 (1987), 99–135. https://doi.org/10.1051/ita/1987210200991