Sound Deadlock Prediction

CHRISTIAN GRAM KALHAUGE and JENS PALSBERG, UCLA, USA

For a concurrent program, a prediction tool maps the history of a single run to a prediction of bugs in an exponential number of other runs. If all those bugs can occur, then the tool is sound. This is the case for some data race tools like RVPredict, but was, until now, not the case for deadlock tools. We present the first sound tool for predicting deadlocks in Java. Unlike previous work, we use request events and a novel form of executability constraints that enable sound and effective deadlock prediction. We model prediction as a general decision problem, which we show is decidable and can be instantiated to both deadlocks and data races. Our proof of decidability maps the decision problem to an equivalent constraint problem that we solve using an SMT-solver. Our experiments show that our tool finds real deadlocks effectively, including some missed by DeadlockFuzzer, which verifies each deadlock candidate by re-executing the input program. Our experiments also show that our tool can be used to predict more, real data races than RVPredict.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Software defect analysis**; *Software testing and debugging*;

Additional Key Words and Phrases: Deadlocks, Prediction

ACM Reference Format:
Christian Gram Kalhauge and Jens Palsberg. 2018. Sound Deadlock Prediction. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 146 (November 2018), 29 pages. https://doi.org/10.1145/3276516

1 INTRODUCTION

A concurrent program can execute in different ways due to interleaving of multiple threads. Even a terminating program has an exponential number of interleavings, which is a challenge for any testing effort. The problem is that a single run represents one interleaving among many, which leaves many other runs to test. The idea of predictive program analysis is to draw conclusions from one run of a concurrent program about an exponential number of other runs. Thus, predictive analysis has the potential to save vast amounts of testing time.

A typical predictive analysis, in this case for deadlocks, proceeds as follows:

(1) run the program, record an execution history, and pick a deadlock candidate in that history;
(2) derive logical constraints from the history and the deadlock candidate; and
(3) solve the constraints.

In the context of predictive analyses, soundness means that a bug predicted by the analysis corresponds to a real bug that can appear in an interleaving of the events in the original, recorded history. Thus, if the analysis is sound and the constraints are satisfiable, then a deadlock is possible with some interleaving of the events in the recorded history. Notice how a single constraint system characterizes an exponential number of executable interleavings.

We draw a distinction between sound deadlock *predictors*, as described above, and deadlock *detectors* that re-execute an input program to confirm a deadlock candidate. DeadLockFuzzer [Joshi et al. 2009] and ConLock [Cai and Lu 2016; Cai et al. 2014] are examples of deadlock detectores that

Authors' address: Christian Gram Kalhauge, kalhauge@cs.ucla.edu; Jens Palsberg, palsberg@ucla.edu, Computer Science Department, UCLA, University of California, Los Angeles, 486 Engineering VI, Los Angeles, CA, 90095, USA.

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution.
© 2018 IW3C2 (International World Wide Web Conference Committee), published under Creative Commons CC-BY 4.0 License.
2475-1421/2018/11-ART146
https://doi.org/10.1145/3276516

each first runs an *unsound* predictive analysis and then excludes false positives via re-execution. While deadlock detectors have been successful in finding real deadlocks, the re-execution of the input programs has drawbacks. Most significantly, re-execution requires access to the inputs of the first run. Access to the inputs may be hard for systems where the inputs may be random, transient, or user-provided, and not stored. However, re-execution could be avoided entirely if the predictive analysis was sound. Thus, one of the benefits of sound prediction is that it avoids re-execution.

Sound predictive analyses have been highly successful when employed to find data races [Chen et al. 2008; Huang et al. 2014; Said et al. 2011; Şerbănuţă et al. 2012] and assertion failures [Huang et al. 2015; Wang et al. 2009]. Additionally, GPredict [Huang et al. 2015] is a state-of-the-art sound predictor of a variety of bugs, but not deadlocks. The reason that GPredict is unable to predict deadlocks is subtle and only became clear after correspondence with one of the GPredict authors. As we will discuss in Section 2, the problem is that GPredict uses constraints that inherently disregard interleavings that can result in a deadlock. Thus, sound deadlock prediction has remained an open problem, until now.

The challenge: Can we design a sound deadlock predictor?

Our results: We present the first sound deadlock predictor for Java. It uses request events and a novel form of executability constraints. We prove correctness of our constraints and show experimentally that our tool, Dirk, does well compared to previous work.

In contrast to previous work, our candidates are based on requests events rather than acquire events. Our insight is that the constraints on request events differ in an essential way from the constraints on acquire events. When a thread acquires a lock, the history contains a request event before the acquisition and an acquire event after the acquisition. The key difference is that the request event is executed even if the lock isn't available. In particular, if the program is about to enter a deadlock and the lock isn't available, the request event is executed while the acquire event isn't executable. Thus, our constraints overcome the problem of disregarding interleavings that can result in a deadlock.

In the spirit of GPredict, our model is built on a general notion of a *candidate*, that is, a set of events in the history that could potentially be a bug in some interleaving. Such candidates can be found for deadlocks, data races, and beyond. We model prediction for candidates as a general decision problem. Our proof of decidability maps the decision problem to an equivalent constraint satisfaction problem that we can give to an off-the-shelf SMT-solver.

We have experimented with a variety of Java programs. We execute each program 1,000 times, instrumented to log the events that we include in histories. After the runs, we identify the deadlock candidates and run our deadlock predictor. In our experiments, our tool found twelve deadlocks, including four from the JaConTeBe [Lin et al. 2015] benchmark suite, while successfully rejecting all unreachable deadlocks.

We did an experimental comparison with the sound deadlock detector DeadlockFuzzer [Joshi et al. 2009] that re-executes a program to confirm a deadlock candidate. Out of 11 benchmarks with known deadlocks, DeadlockFuzzer crashed on the four largest benchmarks (while our tool found deadlocks in all four). Additionally, DeadlockFuzzer matched our tool on four small benchmarks, and found two deadlocks in a fifth small benchmark (while our tool found only one), but found no deadlock in a sixth small benchmark (while our tool found a deadlock). We were able to get DeadlockFuzzer to work on 4 out of 11 DaCpao benchmarks, Dirk worked on all, but neither DeadlockFuzzer nor Dirk find any deadlocks in these benchmarks.

In addition to prediction of deadlocks, we have also experimented with predicting data races and found that our tool predicts more true data races than RVPredict [Huang et al. 2014]. This

confirms that our histories and constraints are more powerful than those of RVPredict, which is a state-of-the-art sound data-race predictor, based on the same concepts as GPredict.

Like previous work, we assume that the memory model is sequential consistency, that is, an execution interleaves atomic actions such as reads and writes. The Java memory model is weaker so any deadlocks we find are real.

The rest of the paper. In section 2 we give an example. We then introduce formalism (Section 3) and show that deadlock prediction is decidable (Section 4). In Section 5 we present our experimental results, in Section 6 we discuss related work, and finally in Section 7 we provide our conclusion. An extended version of this paper has an appendix with proofs of our theorems.

2 EXAMPLE

We will use the example in Figure 1 to illustrate three central points of the paper. First, we will show why previous sound predictors cannot find deadlocks. Second, we will show how our executability constraints help to overcome this problem. Third, we will show how modeling data-flow enables us to predict more deadlocks than previous techniques.

The bank account example. Figure 1a shows a Java class Account with code that can deadlock. Each Account object represents the balance of an account and has two methods transfer and deposit. The transfer method extracts money from the current account and deposits it into another. Both methods are synchronized, so while a deadlock is possible, data races on the balance are impossible.

Figure 1b shows a partial history of an execution that doesn't contain a deadlock. A history is a sequence of events of the form $(o)_t^n$, which says that operation o was executed as the nth operation in thread t.

Initially, the bank has two accounts A and B, each with a balance. The history represents a transfer from A to B and then a transfer from B to A. To start, the first thread begins a transfer by requesting and acquiring the lock for A. Next, it updates the balance for A and then begins the deposit by requesting and acquiring the lock for B. The left column of Figure 1b ends with the update to the balance of B and the release of both locks. The right column of Figure 1b shows how the second thread proceeds in a similar manner to execute a transfer from B to A.

Figure 1c shows a different interleaving that contains a deadlock. As before, the first thread requests and acquires the lock for A and it updates the balance of A, but now the second thread requests and acquires the lock of B and it updates the balance of B. At this point, the second thread requests the lock for A and finds that the first thread already holds that lock. So, the second thread has no choice but to wait for the first thread to release the lock. Meanwhile, the first thread requests the lock for B and finds that the second thread already holds that lock. Thus, this interleaving contains a deadlock.

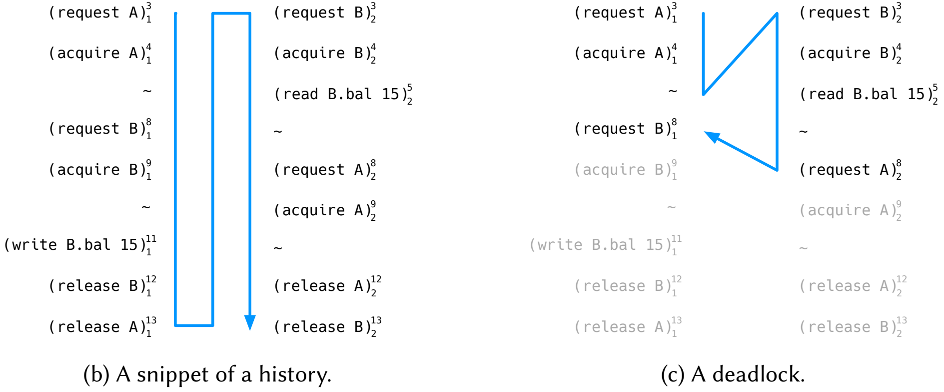
Previous sound predictors disallow deadlocks. A critical question is: can we use a modest modification to previous work on sound data-race prediction that will allow us to predict the deadlock in Figure 1c from the history in Figure 1b? In three steps, we will show that the answer is "no".

Our first step is to consider the GPredict paper [Huang et al. 2015], which describes a state-of-the-art predictor. The supplementary material of the GPredict paper gives a deadlock specification that includes a pattern of lock operations. We will describe that pattern in English, rather than in the GPredict constraint notation.

> A deadlock needs three subsequences of events to match in a history. The first subsequence matches when thread 1 acquires A, thread 2 acquires B, thread 2

```
public class Account {
    int bal = 0;
    synchronized void transfer(int x, Account trg) { this.bal -= x; trg.deposit(x) }
    synchronized void deposit(int x) { this.bal += x; }
}
```

(a) The Account class.



(b) A snippet of a history.

(c) A deadlock.

Fig. 1. An example program and two histories.

acquires A, and thread 1 acquires B. The other two subsequences match when the locks are nested as required for a deadlock.

The intent is for this pattern to match histories like the one in Figure 1c except that it can't! The third and fourth acquire operations won't be executed because of the deadlock and thus they should never appear in the history. In terms of Figure 1c, we can say that the two acquire events, $(\text{acquire B})_1^9$ and $(\text{acquire A})_2^9$, in gray, won't be executed if the corresponding program deadlocks.

Our second step is to consider a modest modification of the GPredict paper. While the histories of GPredict only contain acquire and release events for locks, we could introduce request events, as illustrated in Figure 1b and Figure 1c. This represents substantial progress, because when a dealock occurs the two acquire events won't be executed, but the two associated request events will be.

Indeed, those two request events will be the last events that will be executed in the two deadlocked threads. Now we can change the GPredict specification to match against request events. However, even this improved pattern will never predict any deadlocks.

The problem is that the constraints generated by GPredict also disallow deadlocks. To see why, let's consider the lock constraints used in both GPredict [Huang et al. 2015] and RVPredict [Huang et al. 2014]; they are of the following form:

$$\bigwedge_{(a,r)\neq(a',r')\in S_l} (r' \prec a \;\vee\; r \prec a')$$

Here a, a' and r, r' represent acquire and release events respectively. The "happens before" relation, $\prec$, denotes a constraint on the ordering of events in histories. S_l is a set of paired acquire and release events for a lock l, where each pair denote the boundaries of a critical section. The goal of the constraints is to maintain lock mutual exclusion, namely that two critical sections do not overlap. We can derive these constraints for the history in Figure 1b:

$$\begin{array}{ccc} (\text{release A})_1^{13} \prec (\text{acquire A})_2^9 & \vee & (\text{release A})_2^{12} \prec (\text{acquire A})_1^4 \\ (\text{release B})_2^{13} \prec (\text{acquire B})_1^9 & \vee & (\text{release B})_1^{12} \prec (\text{acquire B})_2^4 \end{array}$$

Here and in GPredict, events are assumed to happen in program order, so all events remain in the per-thread order recorded in the original history. To demonstrate a deadlock, we must find an ordering that solves these constraints and places the deadlocking request events in the very end of each of their threads.

Let first assume that $(\text{release A})_1^{13} \prec (\text{acquire A})_2^{9}$ is true, satisfying the first set of constraints. In this case, we can then satisfy the other constraint with $(\text{release B})_1^{12} \prec (\text{acquire B})_2^{4}$, but that completely avoids the deadlock because it execute all the events of thread 1 before thread 2 even reaches the critical section. Alternatively we can try to satisfy the second part of the second constraint $(\text{release B})_2^{13} \prec (\text{acquire B})_1^{9}$. When taking the program order into account (here marked with "po"), the constraints cannot be solved because they create a cycle in the happens before relation:

$$(\text{release A})_1^{13} \prec (\text{acquire A})_2^{9} \underset{\text{po}}{\prec} (\text{release B})_2^{13} \prec (\text{acquire B})_1^{9} \underset{\text{po}}{\prec} (\text{release A})_1^{13}$$

Assuming that $(\text{release A})_2^{12} \prec (\text{acquire A})_1^{4}$ is true yields a similar result, and thus, the constraints would simply disallow the deadlock in Figure 1c.

Our third step is to analyze why the GPredict paper [Huang et al. 2015] reported that experiments with GPredict actually found a deadlock. After personal communication with one of the GPredict authors, we found out that the implementation deviates from the paper. Specifically, GPredict first removes the acquire and release events of the deadlock, then it generates constraints and solves the constraints. Removing events without careful though is dangerous and can lead to unsoundness, and since this variation comes without any claim about soundness, we conjecture that it is unsound.

In summary, existing sound predictors disallow deadlocks, even after modest modifications.

Executability constraints. The observations above about previous work raise the question: how do we specify sound constraints that allow deadlocks? Intuitively, we need to weaken the constraints used in previous work, but we must be careful. Weakening the constraints means allowing more interleavings and we must avoid impossible interleavings which would make the analysis unsound.

We observe that the acquire events that must be executed to cause a deadlock are also the ones that must be ordered after release events to avoid a deadlock. However, not all acquire events have to, or can, be executed in a history with a deadlock, so they should be disregarded. Finding out exactly which events have to be executable dependents on a given interleaving. Thus, we will specify constraints on the *executability* of events.

For comparison, notice that the constraints defined in the GPredict paper [Huang et al. 2015] require that all acquire events be executed, while the GPredict implementation unconditionally removes some events from a history. By contrast, our constraints represent a sound compromise that encodes whether events should be executed.

Our executability constraints ensure three properties:

(1) if an event is executable, then every earlier event in the thread is also executable;
(2) if an acquire event is executable, then any release event of the same lock ordered before it has to be executable; and
(3) if a read event is executable (and it's value is needed), then the previous write event to the same location, writing the read value, should be executable.

We will now sketch how this works.

We begin with considering the first and second property. Our executablility constraints for an acquire event e look like this:

$$\bigwedge_{a \in \mathrm{A}_l \setminus \{e\}} e \prec a \;\vee\; \bigvee_{r \in \mathrm{R}_a} r \prec e \wedge \Phi^{exec}_{\mathcal{H}}(\emptyset, r)$$

These constraints ensure that all acquire events e that has to be executable, have to for all other acquire events of the same lock (A_l), to either be ordered before that acquire event a or it has to be ordered after the corresponding release event (R_a) of the acquire event. All acquire events are paired with *at most* one release event: the first release event of that lock after the acquire event. Since a history can come from a abruptly terminated program (e.g. a deadlock happened), not all acquire events have a corresponding release event. The constraints $\Phi^{exec}_{\mathcal{H}}(\emptyset, r)$ require that the release event is executable. This ensures that the thread of the release event does not deadlock before releasing the lock.

Returning to Figure 1a, if we wish to detect the deadlock, we may start by requiring that $(\text{request B})^8_1$ and $(\text{request A})^8_2$ are executable. Due to property 1, they will also require that $(\text{acquire A})^4_1$ and $(\text{acquire B})^4_2$ are executable. Applying our constraints above to the two events we obtain:

$$(\text{acquire A})^4_1 \prec (\text{acquire A})^9_2 \;\vee\; ((\text{release A})^{12}_2 \prec (\text{acquire A})^4_1 \wedge \Phi^{exec}_{\mathcal{H}}(\emptyset, (\text{release A})^{12}_2))$$

$$(\text{acquire B})^4_2 \prec (\text{acquire B})^9_1 \;\vee\; ((\text{release B})^{12}_1 \prec (\text{acquire B})^4_2 \wedge \Phi^{exec}_{\mathcal{H}}(\emptyset, (\text{release B})^{12}_1))$$

If the release events are ordered before the acquire events, then there is naturally no deadlock so we take the left side of both disjunctions to get:

$$(\text{acquire A})^4_1 \prec (\text{acquire A})^9_2 \qquad (\text{acquire B})^4_2 \prec (\text{acquire B})^9_1$$

We can see that there is no conflict in these constraints, and we can safely order the two acquire events after the events. The only reason we can do this is because we have not required that $(\text{acquire A})^9_2$ or $(\text{acquire B})^9_1$ are executable.

We are now able to find the cycle in Figure 1c, but the careful reader may be concerned. Since we have allowed the cycle, we have also allowed the program to go into a deadlock. This is critical, as the program might end in a deadlock before reaching the deadlock we are checking. Let us assume that the deadlock we want to find is after the possible deadlock in Figure 1. In this case, the acquire events $(\text{acquire A})^9_2$ and $(\text{acquire B})^9_1$ are executable. We can reuse the constraints from before, because they are the only ones that will get us into the deadlock. Naturally if any of the acquire events were executable, it would not be a deadlock. Let's therefore just inspect $(\text{acquire A})^9_2$.

$$(\text{acquire A})^9_2 \prec (\text{acquire A})^4_1 \;\vee\; ((\text{release A})^{13}_1 \prec (\text{acquire A})^9_2 \wedge \Phi^{exec}_{\mathcal{H}}(\emptyset, (\text{release A})^{13}_1))$$

It is clear that $(\text{acquire A})^9_2 \prec (\text{acquire A})^4_1$ cannot be true, because we already know the opposite. Thus, we focus on the other direction: $(\text{release A})^{13}_1 \prec (\text{acquire A})^9_2$. In this case the new acquire event would be executed after the release of A, which would avoid the deadlock. Our constraints therefore successfully reject histories with deadlocks, except the ones we are looking for.

Data-flow dependency. Now let us consider the third property; "if a read event is executable (and it's value is needed), then the previous write event to the same location, writing the read value, should be executable.".

In the history of Figure 1b, $(\text{read B.bal } 15)^5_2$ sees the \$15 balance of B due to the transfer of 5\$ dollars from A to B preformed with $(\text{write B.bal } 15)^{11}_1$. Without extra work, those two events would always have to be in that order. Importantly, this means that $(\text{write B.bal } 15)^{11}_1$ and $(\text{request A})^8_2$ would have to be executed after the inner critical section of thread 1, and we would not be able to predict the deadlock.

To address this problem, the RVPredict paper [Huang et al. 2014] introduced the notion of feasibility. It stated that the values of a history do not have to be exact for a history to be executable; the values can change as long as it does not affect the executability of future events. This insight gave RVPredict a substantial advantage over Said et al. [2011]. On our example, it does however fall short. RVPredict conjectures that the value read from $(\text{write B.bal } 15)_1^{11}$, could have been used to compute A, which is used by the request event $(\text{request A})_2^8$, it conservatively does not allow us to reorder the events, and would not be able to detect the deadlock.

This feels highly unsatisfactory, so we want to explore this space more. We can reduce the problem to a question of approximating the data-flow from read events to other events. The key is that RVPredict determines that there exist a data-flow from 15$ to A, even though no such data-flow exist. Explained in terms of data-flow, we can say that Said et al. [2011] conclude there could be a data-flow from any read event to any future event, and RVPredict improves on this by only finding a data-flow from read events to future events that uses the local state of the thread.

We could take this data-flow idea to the extreme, and precisely model the data-flow by symbolically logging all values and operations in the program. It would require an extreme amount of instrumentation, and a full simulation of the byte code in events, but then we would be able to have the maximal possible prediction power. We suggest a less invasive data-flow model which is just precise enough to correctly conclude that there is no data-flow from the read event to the request event in our example.

In Java, it is impossible to create a reference to an object from a primitive value. This is one of the hall-mark features in Java compared to C. Since programmers do not have access to the object space, there is a whole range of problems that cannot occur. We exploit this fact, and claim that there is no direct data-flow from primitive values to objects references. Of course, primitive values can still affect the creation and loading of objects, but only through control-flow events which we model. Like RVPredict, we introduce special (branch) events to indicate that any previous read value may have affected the control-flow and therefore all future events. Tracking objects also turns out to be extremely useful, as dynamic dispatching is a big source of control flow in object-oriented programs. Because dynamic dispatching uses the local state of thread, RVPredict would have to insert a (branch) event to indicate that all previous read events could have affected the control-flow of continuation of the history. Here, instead, we can insert an $(\text{enter } r\ m)$ event which only depends on the reference r, and not all values. This data-flow model enables us to find more deadlocks, and on average to out-perform RVPredict on finding data races.

Now, using our new data-flow model, we can conclude that there is no data flow from the read event to the request event and therefore that the value read by the event is unimportant for the execution of the request event. This allows us to reorder the events and show that Figure 1c, in fact, is an executable deadlock.

3 FORMALIZATION

In this section, we will formalize all the components that we need to model deadlock prediction as a decision problem. Specifically, we will formalize histories, executability, and deadlocks, and we will show how those concepts apply to the example in Section 2.

3.1 Histories

A *history* is a sequence of events of the form $(o)_t^n$, which is the nth operation o in the thread t. We define eleven kinds of operations, see fig. 2a.

The eleven operations are the nine operations used by RVPredict [Huang et al. 2014] plus $(\text{request } r)$ and $(\text{enter } r\ m)$. The request event is crucial for predicting deadlocks, as explained earlier. The enter event enables us to distinguish between dynamic dispatch and branching, which

History	$h \coloneqq h; e \mid \epsilon$
Event	$e \coloneqq (o)^n_t$
Operation	$o \coloneqq \text{begin} \mid \text{end}$
	$\mid \text{fork } t \mid \text{join } t$
	$\mid \text{release } r \mid \text{acquire } r \mid \text{request } r$
	$\mid \text{read } l\ v \mid \text{write } l\ v$
	$\mid \text{enter } r\ m \mid \text{branch}$
Value	$v \coloneqq \text{ref}(r) \mid \text{data}_{type}(n)$
Location	$l \coloneqq \text{object}^r_n \mid \text{array}^r_n \mid \text{static}_n$
Thread	$t \coloneqq n$
Ref	$r \coloneqq \mathbf{r}_n \mid \text{null}$
Method	$m \coloneqq \mathbf{m}_n$
Nat	$n \coloneqq 0 \mid 1 \mid 2 \mid \ldots$

(a) History syntax.

begin	the beginning of a thread.
end	the end of a thread.
fork t	a thread forks another thread t.
join t	a thread is done waiting for another thread t to end.
release r	a thread releases a lock r (represented by a reference).
acquire r	a thread acquires a lock r.
read l v	a thread read a value v from location l.
write l v	a thread write a value v to location l.
branch	a thread chooses a conditional branch.
request r	a thread have requests a lock r, but have not yet acquired it.
enter r m	a thread dynamically dispatches on the reference r to m.

(b) Operation descriptions.

Fig. 2. History syntax and operation descriptions.

in turn enables us to model data flow more precisely. A good model of the data flow helps detect more deadlocks, as explained earlier.

The memory operations work on three different kinds of locations: object variables, static variables, and array positions. For simplicity of notation, we have converted all variable names to natural numbers.

Notation. A history is a list and we will use list notation for operations on histories. Specifically, an event can be contained in a history $e \in h$, a history has a length $|h|$, and a history can be the prefix of another history $h \sqsubseteq h'$. We can also append histories $h \cdot h'$. The set of all permutations $[\![h]\!]$ is the set of all reorderings of the events of h. The set of all prefixes of permutations of h is denoted $[h]$. We have a cut-filter syntax $h{\downarrow}^e_f$, which is cutting h to the first occurrence of e in h and then removing all events not matching f. Common filters are a thread id t (indicating that only events in that thread should be kept) and operations $\text{read } l\ *$ (indicating that only events that match those operations should be kept) We can also use a wildcard $(*)$ that matches everything. We allow the shorthand $e = \text{last}(h)$, which means that there exists a prefix h' such that $h = h'; e$. Moreover, $[t \mapsto V]$ gives a map from t to the set V, $M[t]$ gets the set associated with t, and $M \cup M'$ gives a map from each t to union of the sets for that t.

We use two functions on events: $\text{T}_{(o)^n_t} = t$ and $\text{O}_{(o)^n_t} = o$. We can also access the set of all threads in a set E of events: $\text{threads}(E)$.

3.2 Executability and Consistency

We formalize history executability as a notion of history *consistency*. Our starting point for defining consistency are the definitions in Said et. al. [Said et al. 2011] and Huang et al. [Huang et al. 2014]. Based on those, we added clauses about our two new kinds of events ($(\text{request } r)$ and $(\text{enter } r\ m)$) and we generalized the data flow model. We also changed the style of definition to one that uses inference rules, which greatly helped facilitate our proofs. In contrast to the RVPredict paper [Huang et al. 2014], we make no *formal* claims about whether consistency is an accurate model of executability in Java. Instead, we have confirmed *experimentally* that an executable history is consistent and, dually, that a consistent history is executable.

A history h is *consistent* if we can derive $\emptyset \vdash h$ using the rules in Figure 3. More generally, if M is a map of threads to *required values*, then h is consistent with respect to M if we can derive $M \vdash h$ using the rules in Figure 3. Intuitively, the required-value map M enables us to track the data flow through a history. We leave implicit that the definition of $M \vdash h$ is parameterized by a data-flow relation DF, which approximates whether a value is needed to make the rest of a history consistent. In the remainder of this section, we will explain the details of Figure 3, required values, and DF.

We begin with a summary of the definitions of consistency in Said et. al. [Said et al. 2011] and Huang et al. [Huang et al. 2014]. The definition of consistency in Said et. al. [Said et al. 2011] has three main requirements: Read-write consistency, lock consistency and the preservation of concurrency primitives. In more detail, read-write consistency requires each read event to read a value that was written by a write event. Lock consistency requires all locking events to be nonoverlapping. Finally, the preservation of concurrency primitives requires that the events of a thread have to be executed in order, that a thread cannot begin before it has been forked, and that a join event comes after the end of the target thread. The definition of consistency in Huang et al. [Huang et al. 2014] improved upon Said et. al. [Said et al. 2011] by introducing a notion of a *symbolic value*. The idea is that if a value is read by a read event but stays unused until after the candidate then the value is inconsequential. In support of this idea, Huang et al. [2014] also introduced a branch event, which helps model control flow. However, they stopped short of modeling data flow. Instead, they used the conservative approximation that all prior reads contributed to the value that decided the outcome of the branch.

Review of Figure 3. The figure embodies the previous work extended with clauses for the two new events and with use of DF. We will now explain each rule.

(a) defines history consistency using two rules. First, Rule (con-empty) says that an empty history is always consistent. Second, Rule (con-step) says that a nonempty history $(h; e)$ is consistent if the event e in thread t is numbered correctly in that thread, e is a valid extension to its own thread, e is consistent with the other threads in the history $(M, h, t \vdash o)$, and the rest of the history is consistent with respect to a required value map that contains the required values of e. Thus, history consistency relies on a notion of event consistency.

(b) defines event consistency using two rules. First, Rule (con-read) says that if a read event reads a value v that is in the required set $M[t]$, then the last write operation w earlier in the history must write v. This requires that the history up to w $(h{\downarrow}^{w})$ is consistent with the values needed by w (V_w) and $\{v\}$. The value that a write event writes is only important to the consistency of the history if the value is ever read. This requirement is enforced here. Intuitively, the condition $\mathrm{DF}(v, M[t])$ is true if the read event reads a value that is needed later. Notice that if the read does not read a value that is data-flow related to the required values of its own thread, then we require nothing. Second, Rule (con-event) delegates the rest of the definition to (d).

(c) defines valid thread extension using three rules that ensure that events are correctly placed in each thread. First, Rule (v-empty) says that a begin event is always at the beginning of a thread. Second, Rule (v-acq) says that each acquire event comes right after a request event. Finally, Rule (v-step) says that nothing comes after an end event.

(d) defines event consistency for events where M plays no role. In five cases, this imposes no further requirements; let us go through the other five cases. First, Rule (con-begin) says that a begin event must have a fork event before it, unless it is in the first thread 0. Second, Rule (con-fork) says that a fork event can never fork the first thread, and it needs to be the only fork of its thread in the history. Third, Rule (con-join) says that for a join event to be consistent, an end event must be the last event in the thread that it is joining. Fourth, Rule (con-acq) says that an acquire event requires

$$\frac{}{M \vdash \epsilon}\text{(con-empty)} \qquad \frac{e = (o)_t^{|h\downarrow_t|} \quad \text{valid}(h\downarrow_t, o) \quad M, h, t \vdash o \quad M \cup [t \mapsto \mathrm{V}_e] \vdash h}{M \vdash h; e}\text{(con-step)}$$

(a) $\boxed{M \vdash h}$ History consistency

$$\frac{\mathrm{DF}(v, M[t]) \implies \text{writes}(h, l, v, w) \land [\mathrm{T}_w \mapsto \{v\} \cup \mathrm{V}_w] \vdash h\downarrow^w}{M, h, t \vdash \text{read } l\ v}\text{(con-read)} \qquad \frac{h, t \vdash o}{M, h, t \vdash o}\text{(con-event)}$$

$$\text{writes}(h, l, v, w) \triangleq \text{last}\Big(h\downarrow_{\text{write } l\ *}\Big) = w \land \mathrm{O}_w = \text{write } l\ v$$

(b) $\boxed{M, h, t \vdash o}$ Event consistency

$$\frac{h = \epsilon \quad o = \text{begin}}{\text{valid}(h, o)}\text{(v-empty)} \qquad \frac{\text{last}(h) = (\text{request } l)_t^n \quad o = \text{acquire } l}{\text{valid}(h, o)}\text{(v-acq)}$$

$$\frac{\text{last}(h) = (o')_t^n \quad o' \neq \text{end} \quad o' \neq \text{request } l \quad o \neq \text{acquire } l}{\text{valid}(h, o)}\text{(v-step)}$$

(c) $\boxed{\text{valid}(h, o)}$ Intra-thread consistency

$$\frac{t \neq 0 \implies h\downarrow_{\text{fork } t} \neq \epsilon}{h, t \vdash \text{begin}}\text{(con-begin)} \qquad \frac{t' \neq 0 \quad h\downarrow_{\text{fork } t'} = \epsilon}{h, t \vdash \text{fork } t'}\text{(con-fork)} \qquad \frac{\text{last}(h\downarrow_{t'}) = (\text{end})_{t'}^n}{h, t \vdash \text{join } t'}\text{(con-join)}$$

$$\frac{}{h, t \vdash \text{request } l}\text{(con-req)} \qquad \frac{l \notin \mathrm{LS}(h)}{h, t \vdash \text{acquire } l}\text{(con-acq)} \qquad \frac{l \in \mathrm{LS}(h\downarrow_t)}{h, t \vdash \text{release } l}\text{(con-rel)}$$

$$\frac{}{h, t \vdash \text{end}}\text{(con-end)} \qquad \frac{}{h, t \vdash \text{branch}}\text{(con-branch)} \qquad \frac{}{h, t \vdash \text{enter } r\ m}\text{(con-enter)} \qquad \frac{}{h, t \vdash \text{write } l\ v}\text{(con-write)}$$

(d) $\boxed{h, t \vdash o}$ Inter-thread consistency

$$\mathrm{LS}(h) \triangleq \Big\{ l \;\Big|\; \exists t, n.\ (\text{acquire } l)_t^n = \text{last}\Big(h\downarrow_{\text{acquire } l, \text{release } l}\Big)\Big\}$$

(e) $\boxed{\mathrm{LS}(h)}$ Lock set

$$\mathrm{V}_e \triangleq \begin{cases} \{\bot_V\} & \mathrm{O}_e = \text{branch} \\ \{\text{ref}(r)\} & \mathrm{O}_e \in \left\{ \begin{array}{l} \text{request } r, \text{acquire } r, \\ \text{release } r, \text{enter } r\ m \end{array} \right\} \\ V_l & \mathrm{O}_e \in \{\text{read } l\ v, \text{write } l\ v\} \\ \emptyset & \text{otherwise} \end{cases} \qquad \mathrm{V}_l \triangleq \begin{cases} \{\text{ref}(r)\} & l = \text{object}_n^r \\ \{\text{ref}(r), \text{data}_{i32}(n)\} & l = \text{array}_n^r \\ \emptyset & l = \text{static}_n \end{cases}$$

(f) $\boxed{\mathrm{V}_e, \mathrm{V}_l}$ Required values of events and locations

Fig. 3. Consistency rules

that the lock is not held in the history, as indicated by that the lock set (see fig. 3e) of h does not contain l. Fifth, Rule (con-rel) says that a release event requires that the lock it is releasing is in the lockset of the current thread. Notice that this means that a lock cannot be released by other threads than the thread in which it was acquired; this is specific to Java.

(e) defines that the lock set of a history h consists of all the locks in h that have been acquired but not yet released. This enables us to know the locks held at any point of the execution.

(f) defines the values used by an event. The most obstructive event is the branch event. Branches are inserted before any control flow that we do not model with other events. In such cases, we add a $\bot_V$ event, which indicates that all values are required, as anything could happen. For request, acquire, release and enter events we require the reference they point to. This is a break from RVPredict [Huang et al. 2014], where they assume that the value of each lock does not change with the program. We require the references of lock events to be consistent for two reasons: first, synchronization on null throws an error and, second, ensuring that lock events always lock on the same reference enables significantly easier reasoning about deadlocks. For both a read and the write event, the values of the location V_l is required. All other events do not depend on the local state of the thread. Notice that write events do not have the values they write in their required value set. This is because the history determines whether those values are required to make the event consistent. A location can either be the field of an object, in which case the accessed object is required. If it is an array, both the reference to the array, and the integer used to access the array is required. Finally, the location can be a static field, which does not depend on the local state of the program.

The data-flow predicate. A use of data-flow predicate $\mathrm{DF}(v, V)$ expresses that we have a data flow from a read of a value v to a place where v is needed, expressed as a set of values V. We require that DF is monotone in its second argument:

$$V_1 \subseteq V_2 \implies (\forall v.\, \mathrm{DF}(v, V_1) \Rightarrow \mathrm{DF}(v, V_2))$$

We can define a range of data-flow predicates, such as the following:

$$\begin{aligned}
\mathrm{DF}_\top(v, V) &\triangleq \mathit{true} \\
\mathrm{DF}_{!\varnothing}(v, V) &\triangleq V \neq \emptyset \\
\mathrm{DF}_{\mathrm{obj}}(v, V) &\triangleq (\exists r. v = \mathrm{ref}(r) \land v \in V) \lor (\exists v' \in V. \forall r. v' \neq \mathrm{ref}(r)) \\
\mathrm{DF}_\bot(v, V) &\triangleq \mathit{false}
\end{aligned}$$

The predicates $\mathrm{DF}_\top$, $\mathrm{DF}_{!\varnothing}$, and $\mathrm{DF}_{\mathrm{obj}}$ each supports sound deadlock prediction, while $\mathrm{DF}_\bot$ leads to unsound deadlock prediction. The simplest sound variation is $\mathrm{DF}_\top$, which says that there is a data flow from all values to the required-value set. This is the data-flow model used in Said et al. [Said et al. 2011]. The predicate $\mathrm{DF}_{!\varnothing}$ is roughly the one used, implicitly, in RVPredict [Huang et al. 2014]. In more detail, $\mathrm{DF}_{!\varnothing}$ says that we only have to report a data flow if the required-value set actually contains any values. On top of those predicates, we also introduce a new data-flow predicate $\mathrm{DF}_{\mathrm{obj}}$. This new predicate exploits the fact that in Java, references can only be created by the new Object() construct and never by other effects of computation. This means that there is no direct data flow from any value, including references, to other references. So, $\mathrm{DF}_{\mathrm{obj}}(v, V)$ says that there is no data-flow from a value v to the to the required values V, if the required values are all references, and none of them are the value v itself. Those predicates are related as follows:

$$\mathrm{DF}_\bot(v, V) \implies \mathrm{DF}_{\mathrm{obj}}(v, V) \implies \mathrm{DF}_{!\varnothing}(v, V) \implies \mathrm{DF}_\top(v, V)$$

Intuitively, we defined $\mathrm{DF}_{\mathrm{obj}}$ as a predicate that leads to sound deadlock prediction and improves upon $\mathrm{DF}_{!\varnothing}$ and $\mathrm{DF}_\top$.

The strongest possible data-flow predicate is $\mathrm{DF}_\perp$; it says that no data flow is possible, which leads to unsound deadlock prediction. The minimal requirement for a sound data-flow predicate is that all values are data-flow dependent on themselves; if $v \in V$ then $\mathrm{DF}(v, V)$. Clearly, this is false for $\mathrm{DF}_\perp$. We use $\mathrm{DF}_\perp$ in experiments to determine a bound on the potential for our technique: if we cannot predict a deadlock using $\mathrm{DF}_\perp$, no predicate that leads to sound prediction can predict it, either.

3.3 Deadlocks

The literature describes two kinds of deadlocks, namely resource deadlocks and communication deadlocks. This paper focuses on resource deadlocks, like the one in Figure 1, while we leave to future work to predict communication deadlocks soundly. A communication deadlock happens when threads are stuck waiting on messages from other threads.

A history h contains a *deadlock* if there exists a nonempty set of events E such that E forms a deadlock candidate in h, every event in E is last in its thread, and h is a consistent history:

$$\begin{aligned}
\mathrm{deadlock}(E, h) &\triangleq \mathrm{candidate}(E, h) \ \land\ \mathrm{allLast}(E, h) \ \land\ \emptyset \vdash h \\
\mathrm{candidate}(E, h) &\triangleq r_1 \leadsto_h r_2 \leadsto_h \ldots \leadsto_h r_n \leadsto_h r_1 \ \land\ E = \{r_1, r_2, \ldots, r_n\} \neq \emptyset \\
e \leadsto_h r &\triangleq \exists l.\ e \in h \ \land\ r \in h{\downarrow}_{\mathrm{request}\ l} \ \land\ l \in \mathrm{LS}(h{\downarrow}^{e}_{\mathrm{T}_e}) \\
\mathrm{allLast}(E, h) &\triangleq \forall e \in E.\ e = \mathrm{last}\left(h{\downarrow}_{\mathrm{T}_e}\right)
\end{aligned}$$

A deadlock candidate is a cycle in the lock relation $e \leadsto_h r$. Here, e has a lock l in its lock set while r requests l. Intuitively, if e happens, then r has to wait until some release event happens in the future. Thus, e has to be executed before the thread can continue after r. For discovery of deadlock candidates in a history, our implementation uses cycle detection, akin to what is done by unsound deadlock predictors iGoodlock [Joshi et al. 2009] and, later, MagicLock [Cai and Chan 2012, 2014].

Deadlocked threads are stuck. As a sanity check of our definition of deadlock, we prove that if threads enter a deadlock, then they will not perform any more work. We can formalize the idea of "will not perform any more work" as the threads being *locally stuck.*

A history h is *locally stuck* in a set of threads T when all consistent continuations of h contains no events by a thread in T:

$$\mathrm{stuck}(T, h) \triangleq [\emptyset \vdash h \ \land\ \forall h'.\ \emptyset \vdash h \cdot h' \implies h'{\downarrow}_T = \epsilon]$$

Theorem 1 (Deadlocked threads are stuck). *For all histories h and set of events E,*

$$\mathrm{deadlock}(E, h) \implies \mathrm{stuck}(\mathrm{threads}(E), h)$$

We prove the theorem in the appendix of the extended version of this paper.

3.4 Example

We will show that Figure 1c contains a deadlock according to our definitions above. The first step is to identify a deadlock candidate.

The request of $(\text{request B})^8_1$ is lock related to $(\text{request A})^8_2$, denoted

$$(\text{request B})^8_1 \leadsto_{\mathcal{H}} (\text{request A})^8_2$$

Informally, this means that if $(\text{request B})^8_1$ is executed, then the thread of $(\text{request A})^8_2$ cannot proceed and acquire the lock A before the thread of $(\text{request B})^8_1$ has executed a release event. This happens because thread 1 holds the lock on A at the execution point of $(\text{request A})^8_1$.

Figure 4 shows the lock relation for the lock of A (shown in solid blue) and the lock of B (shown in dotted green) in the base history ($\mathcal{H}$) from fig. 1b. In this figure, the solid dark blue arrow corresponds to the lock relation of $(\text{request B})_1^8$ and $(\text{request A})_2^8$, and the dotted dark green arrow shows that $(\text{request A})_2^8$ and $(\text{request B})_1^8$ are lock related. These two relations form a cycle. This means that if both events are executed, then they both have to wait for the other threads thread to emit a release event. This is impossible and we have a deadlock. Formally, the set $E = \{(\text{request B})_1^8, (\text{request A})_2^8\}$ is a deadlock candidate.

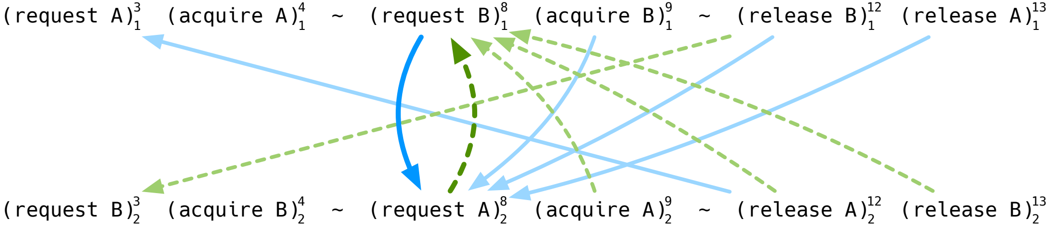


Fig. 4. Lock relation for the lock of A (solid blue) and the lock of B (dotted green).

We have seen that E is a deadlock candidate, now we just have to show that the elements are all last in their threads (allLast(E, h)), and that the history is consistent ($\emptyset \vdash h$).

We will use h to denote the history in Figure 1c:

$$\begin{aligned} h = \ldots; (\text{request A})_1^3; (\text{acquire A})_1^4; \\ \ldots; (\text{request B})_2^3; (\text{acquire B})_2^4; (\text{read B.bal } 15)_2^5; \\ \ldots; (\text{request A})_2^8; (\text{request B})_1^8 \end{aligned}$$

Looking at h we can see that the events in E are all last in their threads. So what remains is to show that the history is consistent. Consistency depends on the choice of data flow predicate (DF). Let us first try with $\text{DF}_{!\varnothing}(v, V)$, which is true for any value v if the required value set V is nonempty.

We want to show that $\emptyset \vdash h$. We use $h', h'', h''', \ldots$ to indicate parts of the history, so that we don't have to juggle the full history. Let $h = h'; (\text{request b})_1^8$. Since there is an event in the history we have to use (con-step). So we need to show that $8 = |h'{\downarrow}_1|$, valid$(h'{\downarrow}_1, \text{request B})$, $\emptyset, h', 1 \vdash \text{request B}$, and $\emptyset \cup [t \mapsto \text{V}_e] \vdash h$. Indeed, 8 events remain in thread 2, as part of "..." above. To show valid$(h'{\downarrow}_1, \text{request B})$, we use (v-step) since operation was not an acquire event, and the last event in the thread did not end it. Since a request event does not have cross thread effects we can use (con-req) with no further requirements to prove $\emptyset, h', 1 \vdash \text{request B}$. We will now continue with $\{(1, \text{B})\} \vdash h'$, because the local values used by $(\text{request b})_1^8$ (V_e) is $\{\text{B}\}$.

Proving consistency for $h' = h''; (\text{request A})_2^8$ follows the same approach, and it adds $(2, \text{A})$ to the required values map: $\{(1, \text{B}), (2, \text{A})\} \vdash h''$.

Now let $h'' = h'''; (\text{read B.bal } 15)_2^5; \ldots$, and fast forwarding over the events in $\ldots$, we are now trying to show that $h'''; (\text{read B.bal } 15)_2^5$ is consistent with the required values $\{(1, \text{B}), (2, \text{A})\}$. As part of that we have to prove $(\{(1, \text{B}), (2, \text{A})\}, h''', 1 \vdash \text{read B.bal } 15)$, and $(\{(1, \text{B}), (2, \text{A})\} \vdash h''')$. Inspecting (con-read) we see that we have to show:

$$\text{DF}(15, \{\text{A}\}) \implies \text{writes}(h''', \text{B.bal}, 15, w) \;\wedge\; [2 \mapsto \{15, \text{B}\}] \vdash h'''{\downarrow}^w$$

We can now see the effect that DF has on consistency. If $\text{DF}(15, \{\text{A}\}) = \text{DF}_{!\varnothing}(15, \{\text{A}\}) = \mathit{true}$, then the last write ($w$) to B.bal, in the history has to write 15 (formally, writes$(h''', \text{B.bal}, 15, w)$).

The initial values of the account is 10, so with no other write events, the last written value is 10. Thus, $\mathrm{writes}(h''', \mathrm{B.bal}, 15, w)$ is false and we can't show $\emptyset \vdash h$ with $\mathrm{DF}_{!\varnothing}(v, V)$.

Let us instead use $\mathrm{DF}_{\mathrm{obj}}(15, \{\mathrm{A}\})$. Since $v = 15$, and there is no non-ref values in V, we can conclude that there is no data-flow from 15 to $\{\mathrm{A}\}$. This makes $\mathrm{DF}_{\mathrm{obj}}(15, \{\mathrm{A}\}) = \mathit{false}$, which means that $\{(1, \mathrm{B}), (2, \mathrm{A})\}, h''', 1 \vdash \text{read B.bal } 15$ is true. Continuing this way, we will finally have to prove that $M \vdash \epsilon$, which we get from (con-empty). We conclude that Figure 1c contains a deadlock.

4 SOUND DEADLOCK PREDICTION IS DECIDABLE

The deadlock prediction problem is: given a consistent history $\mathcal{H}$ and a set of events E in $\mathcal{H}$, does there exist a history h, which is a prefix of a permutation of $\mathcal{H}$ such that $\mathrm{deadlock}(E, h)$? We will show that this problem is decidable. Our approach is to map the problem to an equivalent constraint problem and then show that the constraint problem is decidable.

4.1 Constraints and Satisfaction

A constraint is a proposition ϕ in this happens-before logic, where e_1, e_2 are events (Figure 2):

$$\phi \coloneqq e_1 \prec e_2 \mid e_1 \sim e_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2$$

$$\frac{e_1 \neq e_2 \qquad e_2 \in h \Rightarrow e_1 \in h{\downarrow}^{e_2}}{h \models e_1 \prec e_2}\text{(hbl-hb)}$$

$$\frac{\mathrm{T}_{e_1} \neq \mathrm{T}_{e_2} \qquad e_1 \in h \vee e_2 \in h \Rightarrow e_1 = \mathrm{last}\left(h{\downarrow}^{e_2}_{\mathrm{T}_{e_1},\mathrm{T}_{e_2}}\right) \vee e_2 = \mathrm{last}\left(h{\downarrow}^{e_1}_{\mathrm{T}_{e_1},\mathrm{T}_{e_2}}\right)}{h \models e_1 \sim e_2}\text{(hbl-con)}$$

$$\frac{h \models \phi_1}{h \models \phi_1 \vee \phi_2}\text{(hbl-disjl)} \qquad \frac{h \models \phi_2}{h \models \phi_1 \vee \phi_2}\text{(hbl-disjr)} \qquad \frac{h \models \phi_1 \quad h \models \phi_2}{h \models \phi_1 \wedge \phi_2}\text{(hbl-conj)}$$

A proposition ϕ is about the order of events in histories. Specifically, the $\prec$ operator says that some event happened before another, while $\sim$ says that two events are executed concurrently. Given a history h, the notation $h \models \phi$ means that h satisfies ϕ.

Rule (hbl-hb) says that for $h \models e_1 \prec e_2$ to be true, the two events must be different and if e_2 is in the history, then e_1 should be before e_2 in h. Rule (hbl-con) says that for $h \models e_1 \sim e_2$ to be true, two conditions must be satisfied. First, the two events must be in different threads. Second, e_1 or e_2 must be right before the other in the threads of e_1 and e_2. If one is in the history and the other is not then it should be the last event in the two threads.

We can see easily that the happens-before logic is *prefix-closed* in the sense that if h satisfies ϕ, then every prefix of h also satisfies ϕ.

The satisfiability problem is: given ϕ, does there exist h such that $h \models \phi$? This problem is decidable in exponential time and maps straightforwardly to theories known to SMT solvers, like QF_IDL and QF_RDL.

4.2 From Deadlock Prediction to Constraint Satisfaction

We will reduce the deadlock prediction problem to the constraint satisfaction problem. Given an instance $(\mathcal{H}, E)$ of the deadlock prediction problem, we define a constraint $\Phi_{\mathcal{H}}(E)$ as follows:

$$\Phi_{\mathcal{H}}(E) \triangleq \Phi^{all}_{\mathcal{H}}(E) \wedge \Phi^{con}(E)$$

$$\Phi^{all}_{\mathcal{H}}(E) \triangleq \bigwedge_{e \in E} \Phi^{exec}_{\mathcal{H}}(\emptyset, e) \qquad \Phi^{con}(E) \triangleq \bigwedge_{e, e' \in E, e \neq e'} e \sim e'$$

Here, $\Phi^{all}_{\mathcal{H}}(E)$ says that all events in E must be executable, while $\Phi^{con}(E)$ says that all events in E must be concurrent. Figure 6 defines $\Phi^{exec}_{\mathcal{H}}(\emptyset, e)$, which we explain below.

Figure 6 relies on the must happen before relation, defined in Figure 5, which covers three things. It maintains sequential consistency (mhb-po), it ensures that a thread does not start before it has been forked (mhb-fork), and it ensures that a thread does not continue after trying to join another thread, before the other thread has ended (mhb-join).

$$\frac{e', e \in h \quad e' = (o')^{n-1}_t \quad e = (o)^n_t}{e' \rightarrow_h e}\text{(mhb-po)}$$

$$\frac{e' \in h{\downarrow}_{\text{fork } t} \quad e \in h{\downarrow}_{t,\text{begin}}}{e' \rightarrow_h}\text{(mhb-fork)} \qquad \frac{e' \in h{\downarrow}_{t,\text{end}} \quad e \in h{\downarrow}_{\text{join } t}}{e' \rightarrow_h e}\text{(mhb-join)}$$

Fig. 5. $\boxed{e \rightarrow_h e}$ Must happens before relation

$$\Phi^{exec}_{\mathcal{H}}(V, e) \triangleq \Phi^{mhb}_{\mathcal{H}}(V, e) \wedge \Phi^{read}_{\mathcal{H}}(V, e) \wedge \Phi^{acq}_{\mathcal{H}}(e)$$

$$\Phi^{mhb}_{\mathcal{H}}(V, e) \triangleq \bigwedge_{e' \rightarrow_{\mathcal{H}} e} e' \prec e \wedge \begin{cases} \Phi^{exec}_{\mathcal{H}}(V \cup \mathrm{V}_e, e') & \textit{if } \mathrm{T}_e = \mathrm{T}_{e'} \\ \Phi^{exec}_{\mathcal{H}}(\emptyset, e') & \textit{otherwise} \end{cases}$$

$$\Phi^{read}_{\mathcal{H}}(V, e) \triangleq \bigvee_{w \in \mathrm{W}_{l,v}} \left(\begin{array}{c} w \prec e \wedge \Phi^{exec}_{\mathcal{H}}(\{v\}, w) \\ \wedge \bigwedge_{w' \in \mathrm{W}_{l,*} \setminus \{w\}} w' \prec w \vee e \prec w' \end{array} \right) \quad \Phi^{acq}_{\mathcal{H}}(e) \triangleq \bigwedge_{a \in \mathrm{A}_l \setminus \{e\}} e \prec a \vee \bigvee_{r \in \mathrm{R}_a} r \prec e \wedge \Phi^{exec}_{\mathcal{H}}(\emptyset, r)$$

$$\textit{if } \mathrm{O}_e = \text{read } l\ v \wedge \mathrm{DF}(v, V) \ \textit{else true} \qquad \textit{if } \mathrm{O}_e = \text{acquire } l \ \textit{else true}$$

$$\mathrm{W}_{l,v} \triangleq \mathcal{H}{\downarrow}_{\text{write } l\ v} \qquad \mathrm{A}_l \triangleq \mathcal{H}{\downarrow}_{\text{acquire } l} \qquad \mathrm{R}_a \triangleq \left\{ r \in \mathcal{H}{\downarrow}_{\text{release } l} \,\middle|\, a = \text{last}\left(\mathcal{H}{\downarrow}^{r}_{\text{acquire } l}\right) \right\}$$

Fig. 6. Executability constraints

Executability constraints. We denote the executability constraints like this $\Phi^{exec}_{\mathcal{H}}(V, e)$. The constraints are created over on a base history $\mathcal{H}$. We deem an event e executable if upholds the must-happen-before ($\Phi^{mhb}_{\mathcal{H}}(V, e)$), read ($\Phi^{read}_{\mathcal{H}}(V, e)$) and acquire ($\Phi^{acq}_{\mathcal{H}}(e)$) constraints, using the required values V (see fig. 6). Must-happen-before consistency ($\Phi^{mhb}_{\mathcal{H}}(V, e)$), require all events e' "Must happens before"-related to e in $\mathcal{H}$, to be executable and happens before e. If e' is in the same thread as e (when it is not a join or fork relation) then it also have to be executable with the union of the required values and the required values of e. Read consistency ($\Phi^{read}_{\mathcal{H}}(V, e)$) only applies to all read events whose values are data-flow related to the required value map. For each of those events we require that there exists at least one write w event to that location with that value, which is executed before r, and all other write events are either before w or after r. Acquire consistency ($\Phi^{acq}_{\mathcal{H}}(e)$) only applies to acquire events a. For every acquire event a' of the same lock, either it happens after a, or the corresponding release event is executed before a.

Our constraints are rather different from the ones used in the previous work on sound data-reace prediction, particularly by Said et al. [Said et al. 2011] and by Huang et al. [Huang et al. 2014]. The main difference stems from a key point in our definition of the deadlock prediction problem. Specifically, the deadlock prediction problem calls for a *prefix of a permutation* of a history and asks

only for that prefix to be consistent. Intuitively, we require consistency only from the beginning of the permutation to the place where we find the candidate. This creates a task for the constraints that is much bigger than if we simply require that the entire permutation must be consistent. Our constraints reflect the complexity of this task.

Notice the use of DF in the side condition of $\Phi^{read}_{\mathcal{H}}(V, e)$. We use DF in the process of the generating the constraints, while DF itself is absent from the generated constraints.

Properties. Our reduction of the deadlock prediction problem has the following property, which is the basis for our solution procedure.

Theorem 2 (Correctness). *For every consistent history $\emptyset \vdash \mathcal{H}$, and a set of events $E \subseteq \mathcal{H}$:*

$$\exists h \in [\mathcal{H}] \,.\, \text{deadlock}(E, h) \iff \text{candidate}(E, \mathcal{H}) \wedge \exists h' \in [\![\mathcal{H}]\!] \,.\, h' \models \Phi_{\mathcal{H}}(E)$$

Intuitively, the theorem says that deadlock prediction reduces to 1) a check that we have a candidate and 2) that a constraint derived from the history and candidate is satisfiable. The theorem is true for any DF. We prove the theorem in the appendix of the extended version of this paper; here we sketch key parts of the reasoning.

The direction from left to right is straightforward: from a history h that contains a deadlock E, we have that E is a deadlock candidate and that we can construct $h' \in [\![\mathcal{H}]\!]$ that satisfies $h' \models \Phi_{\mathcal{H}}(E)$ by adding the rest of the events from $\mathcal{H}$ to the end of h. The crucial lemma says that h is consistent if and only if all events h are executable, that is, $\forall e \in h.\ h \models \Phi^{exec}_{\mathcal{H}}(\emptyset, e)$. This lemma is also used in the second half of the proof.

The direction from right to left is more intricate. From a history h' that satisfies $h' \models \Phi_{\mathcal{H}}(E)$, we may need to remove some events to produce a consistent h. We do this with the function minimize:

$$\text{minimize}_{\mathcal{H}}(E, \epsilon) \triangleq \epsilon$$

$$\text{minimize}_{\mathcal{H}}(E, h'; e) \triangleq \begin{cases} h; e & \text{if } h; e \models \Phi^{exec}_{\mathcal{H}}(\emptyset, e) \wedge E \nsubseteq h \\ h & \text{otherwise} \end{cases}$$
$$\text{where } h = \text{minimize}_{\mathcal{H}}(E, h')$$

Notice that minimize runs in polynomial time in the size of its input. The call $\text{minimize}_{\mathcal{H}}(E, h')$ returns a consistent history h with only events from $\mathcal{H}$ and with the events in E all last in h. In particular, a careful examination of $h; e \models \Phi^{exec}_{\mathcal{H}}(\emptyset, e)$, which we skip here, shows that $E \subseteq h$. Additionally, notice that $h' \models \Phi_{\mathcal{H}}(E)$ forces all events to be in program order so, since we remove everything after E, the events in E must be the last events in each of their threads in h. In summary, we have $\text{deadlock}(E, \text{minimize}_{\mathcal{H}}(E, h'))$ if $h' \models \Phi_{\mathcal{H}}(E)$.

Deadlock prediction algorithm. For an instance $(\mathcal{H}, E)$ of the deadlock prediction problem, we decide $\exists h \in [\mathcal{H}] \,.\, \text{deadlock}(E, h)$ by first checking $\text{candidate}(E, \mathcal{H})$ and then checking $\exists h' \in [\![\mathcal{H}]\!] \,.\, h' \models \Phi_{\mathcal{H}}(E)$. Theorem 2 shows that this solution procedure is correct.

Complexity. We first show that we can represent $\Phi_{\mathcal{H}}(E)$ by a Boolean formula of size $O(|\mathcal{H}|^3)$.

We will do that by replacing each copy of a large Boolean expression such as $\Phi^{exec}_{H}(\emptyset, e)$ with a Boolean variable. We define the variables $\mathcal{R}^e_{\mathcal{H}}$, $\mathcal{A}^e_{\mathcal{H}}$ and $\mathcal{J}^e_{\mathcal{H}}$ for $e \in \mathcal{H}$.

$$\begin{aligned} \mathcal{R}^e_{\mathcal{H}} &\triangleq \Phi^{read}_{\mathcal{H}}(V, e) \quad \textit{if } \text{DF}(v, V) \wedge \text{O}_e = \text{read } l\ v \\ \mathcal{A}^e_{\mathcal{H}} &\triangleq \Phi^{acq}_{\mathcal{H}}(e) \\ \mathcal{J}^e_{\mathcal{H}} &\triangleq \Phi^{exec}_{\mathcal{H}}(\emptyset, e) \end{aligned}$$

First, in each use of $\Phi^{exec}_{\mathcal{H}}(V,e)$, $\Phi^{mhb}_{\mathcal{H}}(V,e)$ and $\Phi^{read}_{\mathcal{H}}(V,e)$, we have that V is a known set so we can evaluate the side condition of $\Phi^{read}_{\mathcal{H}}(V,e)$ during constraint generation. We can now translate all occurrences of $\Phi^{exec}_{\mathcal{H}}(\emptyset,e)$ into $\mathcal{J}^{e}_{\mathcal{H}}$. We translate all occurrences of $\Phi^{read}_{\mathcal{H}}(V,e)$ into $\mathcal{R}^{e}_{\mathcal{H}}$ if $\mathrm{DF}(v,V)$ and $\mathrm{O}_e = \mathrm{read}\ l\ v$ and true if it is not. Finally we translate $\Phi^{acq}_{\mathcal{H}}(e)$ into $\mathcal{A}^{e}_{\mathcal{H}}$. While encoding $\Phi^{exec}_{\mathcal{H}}(V,e)$, the number of recursive calls to $\Phi^{exec}_{\mathcal{H}}(V,e)$ is worst case the length of the history $|\mathcal{H}|$, as all cross thread calls, and calls to $\Phi^{acq}_{\mathcal{H}}(e)$ or $\Phi^{read}_{\mathcal{H}}(V,e)$ have been replaced with variables. Encoding $\Phi^{exec}_{\mathcal{H}}(V,e)$ can therefore be done in size $|\mathcal{H}|$.

Encoding $\mathcal{R}^{e}_{\mathcal{H}}$ and $\mathcal{A}^{e}_{\mathcal{H}}$ requires at most $|\mathcal{H}|$ uses of $\Phi^{exec}_{\mathcal{H}}(V,e)$. Therefore, we know that we can encode them in size $|\mathcal{H}| \times |\mathcal{H}|$. Encoding $\mathcal{J}^{e}_{\mathcal{H}}$ is in size $|\mathcal{H}|$. We have $3 \times |\mathcal{H}|$ variables of size $|\mathcal{H}| \times |\mathcal{H}|$. $\Phi_{\mathcal{H}}(E)$ itself uses no more than $|E|$ invocations of $\Phi^{exec}_{\mathcal{H}}(V,e)$ and encoding $\Phi^{con}(E)$ is in size $O(|E|^2)$. Assuming that $|E| \ll |\mathcal{H}|$, we can see that $\Phi_{\mathcal{H}}(E)$ can be encoded in size $O(|\mathcal{H}|^3)$ with all the variables. Because we have an encoding of $\Phi_{\mathcal{H}}(E)$ in size $O(|\mathcal{H}|^3)$, deciding $\exists h' \in [\![\mathcal{H}]\!] \,.\, h' \models \Phi_{\mathcal{H}}(E)$ is exponential in the size of the history.

The deadlock prediction algorithm uses polynomial time to find a candidate E in $\mathcal{H}$ such that $\mathrm{candidate}(E,\mathcal{H})$, and exponential time in the size of the history to decide $\exists h' \in [\![\mathcal{H}]\!] \,.\, h' \models \Phi_{\mathcal{H}}(E)$. In summary the algorithm is in worst case exponential time in the size of the history.

5 EVALUATION

We have implemented our technique in a tool called Dirk. In this section we report on experiments with Dirk aimed at answering these four research questions:

(1) **Can our technique find deadlocks?** We have proved that our technique is sound and now we want to show experimentally that it is effective. We also compare with DeadlockFuzzer [Joshi et al. 2009], which is a sound deadlock *detector*. DeadlockFuzzer uses re-execution to verify the results of the unsound deadlock predictor, iGoodlock, from the same paper.
(2) **Do better data-flow models make a difference for deadlock prediction?** We want to determine whether the additional complexity pays off.
(3) **Does our technique scale to big programs?** We want to determine if our technique can scale to histories greater than 1 billion events.
(4) **How does our technique compare with an existing sound data-race predictor?** We instantiate our notion of candidate with data races and compare with a sound state-of-the-art data-race predictor.

We begin with a summary of the implementation and the experimental setup, after which we report on experiments with deadlock prediction (questions 1 and 2), the experiments with benchmarks at scale (question 3), and then with data-race prediction (question 4).

5.1 Implementation and Experimental Setup

Implementation. We implemented an event logger and an offline solver for Java. The event logger, Wiretap, is implemented using ASM [Bruneton et al. 2002]. The logger writes the history to a file, which is then read by our solver. The solver generates constraints, parameterized by DF, and sends them to Z3 [De Moura and Bjørner 2008]. We filter the candidates on whether they have been solved before, whether they share locks, and whether they are must-happen-before related.

Our logger ignores any class with the org/mockito, org/powermock, wiretap, java, and the sun prefixes. This avoids that the logger instruments itself, though, in return, we cannot log interthread communication with those libraries.

For increased performance, we use a windowing strategy like the one proposed in [Huang et al. 2014]. The windowing strategy allows us to scale to big benchmarks, but the selection of window

size affect the precision but not the soundness of the results. When deriving the constraints we use a compressed values set, so instead of saving all values we only save those that can affect DF.

Solving the constraints can be done by several theories: QF_LIA, QF_IDL, QF_LRA, and QF_RDL. Those are all supersets of happens-before logic, and our experiments showed that QF_RDL was most efficient at solving the constraints.

Setup. The experiments were performed on a machine with 24 Intel(R) Xeon(R) Silver 4116 CPU cores at 2.10GHz and 188 Gb RAM. The sever was running NixOS [Dolstra and Löh 2008], and the tests were run using Nix scripts. We ran the tests using Java 6 (6u45), as this is the version for which most benchmark worked. The benchmarks in the scaling section were executed using Java 8 (build 25.102-b04). The tests were run in parallel batches of 10 to 24 depending on the memory usage.

5.2 Deadlock Prediction

In this section we answer questions 1 and 2.

Benchmarks. We have collected 11 benchmarks, see Table 1. For each benchmark we give the name, the lines of Bytecode (LOBC), which is the number of Bytecode instructions, the number of classes on the classpath, and number of bytes in those classes. Table 1 also presents the average dynamic statistic for 1,000 runs of the tools, which we will cover in the next section.

The first four benchmarks are our baseline that together tests hard cases. Specifically, *bensalem* is a variant of the example from Bensalem and Havelund [2006]; *notadeadlock* is, as the name suggests, not a deadlock, but it has been designed to look like one (see Figure 7 for the code); *picklock* is the motivating example from the PickLock paper [Sorrentino 2015]; and *transfer* is our own example (see Section 2). Note that *picklock* contains both a real deadlock (with ID 3) and something that is not a deadlock but looks like one (with ID 4).

The next three benchmarks (*account*, *deadlock*, *diningPhilosophers*) are from the Software-artifact Infrastructure Repository (SIR) [Do et al. 2005], which is a large collection of Java and C/C++ benchmarks. We focused on the Concurrency Testing Subjects[1] [Dwyer et al. 2006a,b; Visser et al. 2003]. From this collection we extracted all the benchmarks for which annotations said that they contain deadlocks, except the ones for which our inspection of the code revealed that all the deadlocks are communication deadlocks. Note: we removed the outer loop from *diningPhilosophers* to decrease the number of deadlock candidates.

The final four benchmarks (*dbcp1*, *dbcp2*, *derby2*, *log4j2*) are from the JaConTeBe benchmark suite [Lin et al. 2015], which is also available through SIR[2]. The JaConTeBe benchmark suite have been extracted from deadlocks found in real-life programs. The benchmarks have then been set up so that a single run will deadlock with high probability. In JaConTeBe, the deadlocks have been annotated with categories; we selected the ones that were annotated to be resource deadlocks and was not in the standard library. The four benchmarks reported here were the only benchmarks where we could find deadlock candidates, or the program went into a deadlock, in any of the 1,000 runs.

We experimented with many more benchmarks to find deadlocks but found few. We speculate that either deadlocks are fixed quickly after they are discovered, or they stay hidden in rarely executed code [Eslamimehr and Palsberg 2014].

Execution. The execution of a concurrent program can be nondeterministic even with a fixed input. Logging the events from a benchmark multiple times might give drastically different histories.

[1] http://sir.unl.edu/portal/bios/concurrency.php
[2] http://sir.unl.edu/portal/bios/JaConTeBe.php

```
synchronized (l1) { synchronized (l2) {} }
Object tmp = l1; l1 = l2; l2 = tmp;
```

Fig. 7. *notadeadlock*: running two copies of this procedure in parallel threads may seem like a recipe for a deadlock, because the first thread would see l1; l2 and the second would see l2; l1. However, knowledge about the data flow between the two threads enables us to determine that this is a false positive.

Table 1. List of benchmarks used with static data, and average statistics for Dirk and DeadlockFuzzer over 1,000 executions.

	static data			*Dirk*			*DeadlockFuzzer*		
name	LOBC	classes	size[kB]	$\|\mathcal{H}\|$	log[s]	solve[s]	succ	total[s]	cand.
bensalem	871	29	25.3	59.0	0.28	0.21	99	2.17	1
notadeadlock	871	29	25.3	48.0	0.30	0.22	100	1.75	2
picklock	871	29	25.3	49.7	0.37	0.23	100	2.18	3,4
transfer	871	29	25.3	61.6	0.43	0.24	92	2.42	5
account	333	3	3.2	747.1	0.33	0.82	93	3.87	6,7
deadlock	99	4	1.7	37.0	0.29	0.20	100	2.14	8
diningPhil.	123	3	1.4	126.0	0.30	0.22	100	2.33	9
dbcp1	125 590	993	2235.0	1873.6	1.96	0.51	100	-	10,11
dbcp2	125 620	991	2234.5	3162.9	1.76	1.14	100	-	12,13,14
derby2	781 108	2517	11 301.0	1931.2	2.56	0.69	100	-	15
log4j2	36 483	299	730.0	2230.4	7.27	2.22	100	-	16

Thus, we run the logger and the offline solvers 1,000 times on each benchmark, which enables us to gauge both the chance to predict a deadlock and the chance to experience a deadlock.

We ran a solver for each of the 4 different DF's. The solvers were given the same histories to solve. If the logger ended up in a deadlock, then it stopped the execution and noted the deadlock.

We have compared our implementation against DeadlockFuzzer [Joshi et al. 2009], a state-of-the-art deadlock detector. DeadlockFuzzer works by first predicting candidates, using their own predictor called iGoodlock, and then try to confirm them using re-execution. We also ran DeadlockFuzzer 1,000 times. We followed same setup as noted in the run file at their GitHub repository[3], except we did only a single execution of the re-player instead of three.

Results. In Table 1, we present the dynamic information of running the two deadlock detectors, *Dirk* and *DeadlockFuzzer*. In the *Dirk* part of the table, $|\mathcal{H}|$, indicates the length of the history, "log" column show the time to log the history (instrumentation is done on the fly so it is counted in this column), and the "solve" columns indicates the time solve the history with $\mathrm{DF}_{\mathrm{obj}}$. In the *DeadlockFuzzer* part of the table we have two columns. DeadlockFuzzer where run with timeout of 1800 s (30 minutes), and it was clear that the predictor (iGoodlock) either succeeded fast or timed out. So instead of reporting an average time we report a percentage of times that prediction terminated "succ". All results going forward is only over the runs that did not timeout. The next column is the average time to instrument, log, solve and re-execute in the benchmark. The fields left black didn't instrument correctly because of reflection. The final columns "cand." lists the identifiers for 16 deadlock candidates found by the tools, which we will cover in more detail in Table 2.

[3] https://github.com/ksen007/calfuzzer

Table 2. Findings for each candidate. Each column answers, for 1,000 history collected from the benchmark, how often in percentages is the candidate reported by each solver. The "cand." column indicate times the candidate where found. The "actual" indicates that the actual histories that was in a deadlock.

			Dirk						
ID	benchmark	locks	cand.	$DF_{\bot}$	DF_{obj}	$DF_{!\varnothing}$	$DF_{\top}$	*d.fuz.*	actual
1	bensalem	2	100.0	100.0	100.0	100.0	100.0	99.6	0.0
2	notadeadlock	2	98.3	98.3	0.0	0.0	0.0	0.0	0.0
3	picklock	2	100.0	100.0	100.0	100.0	100.0	100.0	1.1
4	picklock	2	98.9	0.0	0.0	0.0	0.0	0.0	0.0
5	transfer	2	100.0	100.0	100.0	2.1	2.1	100.0	2.1
6	account	4	100.0	100.0	0.0	0.0	0.0	84.0	0.0
7	account	3	100.0	100.0	0.1	0.1	0.1	95.4	0.0
8	deadlock	2	100.0	100.0	100.0	1.1	1.1	100.0	0.5
9	diningPhil.	5	100.0	100.0	100.0	100.0	100.0	0.0	0.0
10	dbcp1	2	100.0	100.0	99.3	99.3	99.3	-	74.7
11	dbcp1	2	25.3	25.3	24.6	24.6	24.6	-	2.3
12	dbcp2	2	4.0	4.0	4.0	4.0	4.0	-	4.0
13	dbcp2	2	55.4	55.4	0.0	0.0	0.0	-	0.0
14	dbcp2	2	96.0	96.0	40.6	40.6	40.6	-	40.6
15	derby2	2	100.0	100.0	100.0	100.0	100.0	-	58.7
16	log4j2	2	100.0	100.0	100.0	100.0	100.0	-	100.0

Table 2 lists our results for the 16 deadlock candidates in our 11 benchmarks. The first three columns are the candidate IDs, the benchmarks in which they were found, and the number of locks involved in the candidate. The next column indicates the frequency with which we found a candidate in the 1,000 runs. The next column describes how often a deadlock was found with the unsound $DF_{\bot}$, while the next three columns report on what we found with the three sound DF's. The next column, marked with "d.fuz." shows the results of running DeadlockFuzzer. Our table shows the count of deadlocks per method in which the deadlocking code appears. We made this happen for DeadlockFuzzer by removing some duplicates in DeadlockFuzzer's output. This was needed because DeadlockFuzzer counts deadlocks per call site, and hence may report a deadlock more than once. Each field marked with a dash indicates that the tool failed on the benchmark. The last column shows the percentage of Dirk's logged runs that actually ended up in the listed deadlock.

DeadlockFuzzer failed on *dbcp1*, *dbcp2*, *derby2*, and *log4j2* with a "NoClassDefFoundError". These errors happened when the tool tried to run the instrumented code. The instrumentation of those four benchmarks took 35.53, 29.22, 33.54 and 6.81 seconds, respectively. So, DeadlockFuzzer was unable to find candidates or deadlocks for those cases.

Evaluation. First we see that the sound DF's only report sound deadlocks. In particular, the two nondeadlocks with IDs 2, 4 were predicted by none of the sound predictors or DeadlockFuzzer.

Now we address research question (1). For most of the candidates (1,3,5,7,8,9,10,11,15), the chance that a single run helps predict that a candidate is an actual deadlock is higher than actually encountering the deadlock. For other candidates (6,12,13,14,16), we saw no difference. Thus our tool does much better than simply running the program.

Compared to the DF_{obj} data-flow predicate, DeadlockFuzzer did find an equivalent number of deadlocks in the benchmarks for which it works. An exception to this is 6 and 7, for which DeadlockFuzzer outperforms Dirk. Both of theses candidates stems from the *account* benchmark (which is different from the Account class shown in the example). A possible reason why Dirk does worse on 6 and 7 is that the benchmark uses many array accesses and if statements, both of which are considered branching events in our model. For such uses, our model ensures that all prior read events read the same value, which may be too strict for this case. Notice that our very strong (and unsound) data-flow predicate $DF_{\perp}$, do detect the deadlock. It might therefore be possible that a stronger sound data-flow predicate than DF_{obj} exists, which can detect this kind of deadlocks. We leave it to future work to design a better data-flow predicate. Dirk did outperform DeadlockFuzzer on 1 by a less than a percentage, which might be due to thrashing or an unlucky execution in the scheduler. Dirk did much better in 9, were DeadlockFuzzer was not able to detect the deadlock candidate at all.

DeadlockFuzzer did sometimes report the same deadlock under a different candidate, as it might run into another deadlock in the program while solving for the first. These experiments suggest that for small test cases Dirk perform similarly to DeadlockFuzzer and is, on average, 4 times faster. In this case using Dirk as the base predictor in a re-execution detector to quickly eliminate sound deadlocks, might make sense. Re-execution can then be done on the candidates not proven sound to improve the precision of the predictor.

DeadlockFuzzer did not find any of the last 7 candidates, this was due to a problem with the instrumentation in the presence of reflection. Reflection handling is notoriously hard when doing offline instrumentation. Dirk does not use offline instrumentation but instead instruments classes on the fly as they are loaded into the ClassPool. Online instrumentation is not always an option for re-executers as they require consistency between runs. Dirk also have the benefit of being able to run without all classes instrumented as long as no synchronization happens between the threads in the uninstrumented regions. DeadlockFuzzer on the contrary uses a system to keep track of objects between runs to faith-fully direct a replay towards the correct candidate. It is therefore a critical error if not all the code is instrumented. By having online instrumentation and being able to ignore some classes we outperform DeadlockFuzzer on all cases, where we are able to instrument, run and find the deadlocks before DeadlockFuzzer is done instrumenting. In summary, Dirk avoids re-execution of code so it handles reflection and missing code robustly and gives a speed advantage.

Now we address research question (2). Even though $DF_{!\varnothing}$ is based of RVPredict and $DF_{\top}$ is based of Said et. al., we saw no difference between the results of $DF_{!\varnothing}$ and $DF_{\top}$. This may be due to our stricter definition of V_e compared to the one used (implicitly) in RVPredict. In contrast, DF_{obj} dramatically improves the chance to predict the candidates 5 and 8, and we see that it has a strictly higher chance of finding bugs than $DF_{!\varnothing}$ and $DF_{\top}$. We note that all of the sound DF's have approximately the same solving time, so we see no drawback in using DF_{obj}. We made no special optimizations for any of the choices of DF.

Our unsound predictor based on $DF_{\perp}$ reports the false positive candidate 2, but not candidate 4. We do not know whether the candidate 13 is actually a real deadlock. However, notice that the unsound predictor based on $DF_{\perp}$ predicts almost all the actual deadlocks with a 100% probability, which indicates that it may be a good indicator for the *absence* of deadlocks, in case it rejects the candidate.

Candidates 8,9,10,11 depend heavily on the thread scheduling. We leave to future work to combine Dirk with a tool like MCR [Huang 2015] or Sherlock [Eslamimehr and Palsberg 2014] to improve the chance of predicting those deadlocks.

In the four JaConTeBe benchmarks, candidates 10 and 11 had been reported here[4], candidate 14 had been reported here[5], candidate 15 had been reported here[6], and candidate 16 had been reported here[7]. In contrast, to the best of our knowledge, candidate 12 has not been reported before, so we list it here:

```
org/apache/commons/dbcp/AbandonedTrace.clearTrace:()V!-1
org/apache/commons/pool/impl/GenericObjectPool.addObjectToPool:(Ljava/lang/Object;Z)V!59
```

5.3 Scaling to Billions of Events

In this section we answer question 3. We do that using two versions of the DaCapo benchmarks [Blackburn et al. 2006]. First, we evaluate our tool on the 2018 release of the DaCapo benchmarks, and second, we compare our tool with DeadlockFuzzer based on the 2009 release.

5.3.1 Evaluation Based on the 2018 DaCapo Benchmarks. In Dirk we have implemented a windowing technique like in RVPredict [Huang et al. 2014], which enables us to run in linear time of the history. This enables us to scale to very big benchmarks. To evaluate this we have run the DaCapo benchmarks [Blackburn et al. 2006]. We used the Bach 9.12 MR1 version of the benchmark suite, released in 2018. It contained a major revision that made it possible to run the benchmark suite using Java 8. We ran the benchmarks unmodified. The DaCapo benchmark suite contains 14 benchmarks, totaling 27,527 classes and 6,838,117 lines of Bytecode. As per the instructions of the benchmark-suite we used the newest version of lusearch (lusearch-fix). The benchmarks tomcat, tracebeans and tradesoap were discarded as they did not succeed in our test framework when running them without instrumentation.

Execution. Each benchmark was run in the small configuration and only run once and with no time limit. The tests were all run in parallel. Dirk were configured with windows of size 500,000 because the histories were big. The idea is to scan for candidates and then if a candidate is found, output the window that contains the candidate and then in a separate pass try to verify the deadlock with a smaller window size.

Results. We did not find deadlock candidates in any of the DaCapo benchmarks, so we cannot measure the time to verify any deadlocks. Table 3 shows the results of running each of the benchmarks with Dirk. The first column represents the name, and the second, the number of threads. The three next columns represent the running times in seconds of running the benchmark with no instrumentation "run", logging the history "log", and searching the history for deadlocks "solve". We have totaled the running time of all the benchmarks under these columns. For logging and solving we describe the relative times compared to running uninstrumented. The last four columns contains the number of millions of events logged in the histories, the $|\mathcal{H}|$ column describes the total number of events in the history, the "lock" the number of lock events, "R/W" the number of read and write events, and finally "E/B" the number of enter and branch events. In the bottom of the columns we list the total number of events and the percentages that each group of events contributed.

Evaluation. The windowing algorithm used in Dirk enable us to scale to big benchmarks. Dirk did in 15 hours manage to check a total of 5.5 billion events for the absence of deadlocks. The algorithm has to runningly maintain a lockset, and for each window it creates a lock graph. While the overhead of logging is not directly correlated with the running time, the algorithm roughly

[4]https://issues.apache.org/jira/browse/DBCP-65
[5]https://issues.apache.org/jira/browse/DBCP-270
[6]https://issues.apache.org/jira/browse/DERBY-5560
[7]https://bz.apache.org/bugzilla/show_bug.cgi?id=41214

Table 3. The result of running Dirk on the DaCapo benchmark suite (Bach 9.12 MR1 2018).

		running times (in seconds)			*number of events (in millions)*			
name	threads	run	log	solve	$\lvert\mathcal{H}\rvert$	lock	R/W	E/B
avrora	4	5.94	894.07	9928.14	1450.21	2.95	882.04	565.21
batik	2	7.09	25.47	152.80	26.73	0.07	13.01	13.65
eclipse	14	17.59	77.05	743.35	98.83	0.84	45.74	52.24
fop	1	2.71	20.49	178.48	24.47	0.00	11.61	12.86
h2	26	6.72	713.11	39 012.78	2623.81	13.71	1176.82	1433.28
jython	2	8.55	108.32	1806.33	241.83	12.31	86.05	143.48
luindex	1	2.48	15.49	201.95	26.74	0.00	16.66	10.08
lusearch-fix	9	0.79	248.45	2181.76	304.02	0.62	192.51	110.89
pmd	1	1.00	9.70	48.89	6.60	0.00	3.87	2.72
sunflow	49	1.57	221.88	3553.57	523.82	0.00	370.56	153.26
xalan	11	2.36	155.32	1710.97	203.76	1.34	119.38	83.04
		56.80	×44	×1048	5530.82	0.58%	52.76%	46.66%

scales linearly with between 1.7 - 3.6 million events per second, except "pmd" which only reports 0.7 million events per second (over 9 seconds). The search algorithm scales linearly with between 119 and 174 thousands events per second for all benchmarks except "h2" which only processed 67 thousand per second. The most time was used in "h2", where there was both a high number of lock events and number of threads. These numbers makes us comparable with tools like MagicLock [Cai and Chan 2014], which, in their paper, showed that they could handle 33 million lock events in 50s. Other tools like iGoodlock (the predictor in Deadlocks) [Joshi et al. 2009] and MulticoreSDK [Da Luo et al. 2011], did in the same evaluation did not scale above 0.5 million events.

Our logger is unoptimized and its logging overhead of x44 times is too much for production use, yet within the scope of integration testing. Except for "h2", the search time below 3 hours which means that it can be used in a nightly build cycle.

Interestingly, most of the events are read, write, enter and branch events: almost 99.42%. Since the search time is roughly correlated with the number of events in the history, future work could address how to remove unneeded events from the histories to speed up search. Also, techniques for finding candidates from deadlock predictors such as MagicLock [Cai and Chan 2014] might be used to improve the search time.

5.3.2 Comparison Based on the 2009 DaCapo Benchmarks. The DaCapo benchmarks all use reflection, which is unsupported by DeadlockFuzzer. We overcame this problem by running DeadlockFuzzer together with Tamiflex [Bodden et al. 2011] and by modifying DeadlockFuzzer modestly.

We found that DeadlockFuzzer fails to work on the 2018 release of the DaCapo benchmarks, likely due to changes in Java. Instead we compare our tool and DeadlockFuzzer based on the 2009 release of the DaCapo benchmarks.

Results. Our version of DeadlockFuzzer works for four of the 2009 DaCapo benchmarks, see Table 4. The first column shows the name of the benchmark, the second column shows the number of threads, and next three show the running times for running the benchmarks without transformations, running dirk (logging + solving) and running DeadlockFuzzer (tamiflex + inst + deadlock prediction + re-execution), respectively. DeadlockFuzzer found four candidates, all in *eclipse*, but its re-execution confirmed none of them.

Table 4. The result of running Dirk and DeadlockFuzzer on the DaCapo benchmark suite (Bach 9.12 2009). *Eclipse did not run to completion.

		time (in seconds)		
name	threads	run	*dirk*	*d.fuz*
eclipse	19	20.05	907.63	402.48*
luindex	1	2.55	241.32	166.55
lusearch	9	3.29	2276.97	204.39
sunflow	49	2.14	3576.64	181.17
		28.03	×249.8	×34.1

Evaluation. DeadlockFuzzer is 65 times faster than Dirk, on average. Thus, re-running the program is much more efficient than logging the nesseary events and then solving constraints for offline prediction. However, as we saw earlier, Dirk can find deadlocks that DeadlockFuzzer misses.

5.4 Data-Race Prediction

In this section we answer question 4.

Data-Race Candidates. We can instantiate our notion of candidate to the case of data races:

$$\text{data-race}(E, h) \triangleq \text{candidate}_{DR}(E, h) \land \text{allLast}(E, h) \land \emptyset \vdash h$$
$$\text{candidate}_{DR}(\{e_1, e_2\}, h) \triangleq e_1, e_2 \in h \ \land \ \mathrm{O}_{e_1} = \text{write}\ l\ v \ \land \ \mathrm{O}_{e_2} \in \{\text{write}\ l\ v'', \text{read}\ l\ v'\}$$

Benchmarks. We use the benchmarks from the RVPredict paper [Huang et al. 2014], plus one additional benchmark *co-example*, see Figure 8. We use *co-example* to illustrate a difference between RVPredict and Dirk, as follows. After communication with one of the authors of the RVPredict paper, we learned that to remain sound, when not modeling that acquire events uses the local state, RVPredict always orders events that share references as in the original history. In contrast, Dirk has no such limitation.

```
t1.1 |  b.i++;  // data race                       t2.1 |  synchronized (L) { Object a = o; }
t1.2 |  synchronized (L) { o = new Object(); }     t2.2 |  b.i++; // data race
```

Fig. 8. *co-example*: RVPredict requires that t2.1 always read the same object from t1.2. This makes the data race in t2.2 undetectable by RVPredict. Dirk can see that b is a different object than o, so we can ignore the read event in t2.1. In the benchmark, thread 1 is executed before thread 2 with a high probability.

Execution. Dirk records a superset of the events recorded by RVPredict. For fair comparison, we ran each tool 100 times and averaged the results. We run both RVPredict and Dirk with 60 seconds solver timeout and 30 minutes (1,800 seconds) total timeout. Additionally, for both tools we used windows of 10,000 events, although the difference in the number of events recorded makes this slightly uneven.

Evaluation. Table 5 list the results for the 12 benchmarks. The first column is the name of the benchmark, and the second and third column show the length of the histories for the two tools. The 4th, 5th, and 6th column show the number of candidates in Dirk histories, and the number of data races that each tool predicted, including the standard deviation (marked with ±). The remaining columns show the logging time and the solve time for each tool.

Table 5. Data-race prediction by Dirk and RVPredict across 100 executions.

	$\|\mathcal{H}\|$		*data-races*			*log [s]*		*solve [s]*	
program	dirk	rvp	cand.	dirk	rvp	dirk	rvp	dirk	rvp
co-example	2.70×10^1	1.70×10^1	3.0±0	3.0±0	0.0±0	0.4	0.7	0.2	0.6
critical	4.26×10^1	3.51×10^1	6.7±1	6.7±1	5.4±2	1.1	0.6	0.2	0.7
airline	2.01×10^2	2.14×10^2	9.0±0	9.0±0	9.0±0	2.3	0.6	0.3	0.8
account	1.40×10^2	1.26×10^2	5.0±0	5.0±0	5.0±0	0.8	1.1	0.2	0.7
bbuffer	2.31×10^3	2.23×10^3	15.0±0	10.9±1	9.5±3	2.4	1.1	483.9	79.5
pingpong	1.55×10^4	5.65×10^1	3.0±0	3.0±0	2.7±1	2.3	0.7	0.4	0.7
bubblesort	9.88×10^3	5.40×10^3	9.2±1	0.0±0	9.1±2	2.4	1.3	1800.0	249.7
bufwriter	1.63×10^3	2.78×10^2	4.0±0	0.7±1	2.0±0	0.3	0.6	1529.0	1.0
mergesort	1.51×10^3	1.77×10^3	4.0±2	4.0±2	4.0±2	2.3	0.8	41.1	2.4
raytracer	2.51×10^4	2.28×10^4	5.0±0	5.0±0	4.8±1	0.5	2.4	41.1	1.6
montecarlo	9.50×10^6	1.17×10^7	0.0±0	0.0±0	1.2±4	5.0	152.5	1800.0	58.2
moldyn	2.66×10^5	2.64×10^5	23.5±4	0.0±0	12.0±4	0.7	5.8	1800.0	158.6

The two tools produce histories of similar length, except for *bufwriter* and *pingpong*, where Dirk produces longer histories. Many factors can play a role here, including different scheduling in different runs, particularly due to the effect of logging on scheduling. In most cases, the logging times are similar.

Now we address research question (4). First, we consider cases where Dirk reports *more* data races than RVPredict (*co-example*, *critical*, *bbuffer*, *pingpong*). In all cases, except *co-example*, we manually verified this subset property: the data races predicted by Dirk in any run is a subset of those predicted by RVPredict in all runs. Inspecting these we can see that the standard derivation of finding data races with Dirk is lower than RVPredict. This tells us that Dirk are less subject to data races hiding in interleavings. Second, we consider cases where the numbers across the two tools are the same (*airline*, *account*, and *mergesort*). Especially interesting is *mergesort*, where a high standard derivation for merely detecting the candidates mean that the existence of data race candidates are heavily dependent on the scheduling. Third, we consider the case where Dirk reports *fewer* data races than RVPredict (*bubblesort*, *bufwriter*, *montecarlo*, *moldyn*), including three cases where Dirk times out. Dirk uses more complex constraints than RVPredict, which may be the reason Dirk times out more frequently. On *bufwriter*, Dirk times out on 67% of the runs, on the runs where it does not timeout we correctly report two data races like RVPredict. We note that RVPredict reports just a few data races that Dirk misses which we believe is due to the fortunes and misfortunes of scheduling.

In summary, Dirk tends to predict more data races than RVPredict, but times out in more cases, due to increased solve time.

6 RELATED WORK

In Section 2, we gave a comparison of our technique with the prediction tool GPredict [Huang et al. 2015], and in Section 5, and we gave an experimental comparison of our tool with Deadlock-Fuzzer [Joshi et al. 2009] and RVPredict [Huang et al. 2014]. In this section we compare our results with other related work.

Unsound deadlock prediction. Predictions tools make predictions about an exponential number of permutations of the history of a single execution. Among such tools, *unsound* prediction tools may produce false positives. The earliest deadlock prediction tools are GoodLock [Havelund 2000] and Visual Thread [Harrow 2000]. Essentially, both techniques approximated the lock acquisition history by building a graph, and then checking the graph for existence of deadlock candidates. Later, Bensalem and Havelund [Bensalem and Havelund 2006], formalized the appropriate notion of graph, which is also known as a lock-order graph. After those papers prediction techniques have improved by *storing more information* during a run, this way they can remove edges in the lock-order graph and therefore avoid some false positives.

MulticoreSDK [Da Luo et al. 2011] developed a layered approach that first finds candidates in a location graph and then in a traditional lock-order graph. iGoodlock [Joshi et al. 2009] introduced a lock-dependency relation that is similar to a lock-order graph and also contains the context of each lock, which is helpful for attempts to verify a deadlock. For comparison, our approach relies on request events, which are absent from both lock-order graphs and lock-dependency relations. MagicLock [Cai and Chan 2012, 2014] improved the performance of iGoodlock by clever optimizations that remove many false positives, including single-threaded deadlocks and deadlocks that have a shared guarding lock (a higher level lock). PickLock [Sorrentino 2015] uses a different approach: it includes information about the lock-acquisition history and happens-before information on top of the lock order. Using this information the tool can exclude some false positives that were left over by previous techniques. We asked the authors of PickLock for the implementation, but were unable to get it. Instead, for comparison, we implemented their motivating example and, as we discussed in Section 5, we successfully rejected their false positive (candidate 4 in Table 2). While our approach finds deadlock candidates in much the same way as MagicLock and iGoodlock, our constraints and decision procedure avoid false positives.

Sound deadlock detection, by re-execution. Instead of reducing the number of false positives, another direction aims at verifying the candidates produced by a deadlock predictor by *running the program again.* Prominent tools include DeadlockFuzzer [Joshi et al. 2009], ConLock [Cai et al. 2014], and ASN [Cai et al. 2015]. Those tools use randomized approaches that efficiently try out different schedules to see if they can recreate the deadlock in the program. Reproducing deadlocks this way, however, might result in thrashing. Thrashing happens when the program ends in an artificial deadlock that stems from barriers that the analysis inserted to force the execution in some direction. ConLock$^+$ [Cai and Lu 2016] uses constraints on the events while running the program, to decrease thrashing to a negligible amount, and the tool reproduces most deadlocks with >80% probability. However, the inner model of ConLock$^+$ does not take read-write induced control-flow changes into account, which may explain the modest level of thrashing they do experience. For comparison, our analysis is completely offline after the initial logging, and it cannot result in thrashing. Additionally, since our analysis is sound we avoid verifying the results by running the program again. This is important because the additional runs come with no guarantee of success.

Other deadlock detectors. Static analysis tends to be highly conservative at handling concurrency. Chord [Naik et al. 2009] and RacerX [Engler and Ashcraft 2003] are both static deadlock analyses, yet both of them produce many false positives, and even then they come with no guarantee of reporting all possible deadlocks. Any effort to investigate the many false positives is daunting for a developer. For comparison, while our tool don't report all possible deadlocks in a program from a single run, every deadlock that it does report is a real deadlock and should be taken seriously.

Working on the same premise as the dynamic deadlock fuzzers, SherLock [Eslamimehr and Palsberg 2014] tries to confirm deadlocks reported by a static analysis tool. It does this by synthesizing

histories in the program using a concolic engine. Sherlock also uses an SMT-solver to predict possible interleavings of threads, but its specification of a deadlock is unsound. We see potential in combining concolic execution with our technique, yet we leave that to future work.

Another direction is model checking, which checks all paths through a program. For example, the Java PathFinder [Havelund and Pressburger 2000] translates from Java to Promela, after which it analyzes Promela with the model checker SPIN [Holzmann 1990]. On top on SPIN, researchers have defined deadlock checkers [DeMartini et al. 1999; Visser et al. 2003]. This approach has the potential to detect all possible deadlocks, yet the approach tends to be slow for large programs.

Other predictive models. The first predictive model to use an SMT solver to solve for a permutation of a history was *concurrent trace programs* (CTP) [Wang et al. 2009]. CTP uses a model that looks like executability constraints, but is far more expressive. It models all events uniformly as symbolic assignments and conditions using the source code. The CTP model is more powerful than ours, but it requires the SMT solver to solve path constraints, which in general is more expensive. This extra power was needed to find assertion violation, which is the focus of the paper, but the extra expressiveness is not needed to find deadlocks or data races. The CTP model have not been used for finding deadlocks or data races.

Our approach is akin to sound data-race predictors. We build on the maximal causal model (MCM) [Serbanuta et al. 2008, 2012]. It maps any history to a sound and *maximal* set of predictions, relative to the chosen model of executions. Using this model, Said et al. [Said et al. 2011] used an SMT solver to do sound data-race prediction for Java programs. RVPredict [Huang et al. 2014] took this a step further by modeling the control-flow. We build on both of these papers by adding request events and executability constraints. In our experiments, our tool tends to find more data races than RVPredict, which is because we log more information.

Other models for finding data races includes the happens-before model (HB) [Lamport 1978]. This model cannot be used to find deadlocks (in the form in [Lamport 1978]) because it explicitly orders all locks in the order they where found. Causally Precedes (CP) [Smaragdakis et al. 2012] allows for reordering of locks, but only if they do not read or write from the same variables. Most recently, Weakly Casually Precedes (WCP) [Kini et al. 2017] is a weaker version of CP. Experiments are needed to determine whether CP and WCP would do a good job of finding deadlocks. Both of them claim that if they predict a data race it is either a data race or a deadlock. However, we see no easy way to distinguish, and if it is a deadlock, which locks or locations are involved.

7 CONCLUSION

We have shown how to do sound deadlock prediction for Java. The key insight is to use request events and to use executability constraints to model that once a thread is part of a deadlock, it executes no further events. Our technique uses a general notion of candidate that we have also used for data-race prediction. We have proved correctness and we have shown that our tool is both effective at predicting deadlocks and competitive with previous work at detecting deadlocks and predicting data races.

Our model of deadlock prediction has three key limitations. First, it models sequential consistency, which is stronger than most realistic memory models, including the Java memory model. We leave to future work to do sound deadlock prediction for weak memory models. Second, the release of a lock can only be of a lock that was acquired by that same thread. This is true in such languages as Java and C#, where locks are structured in blocks, while in such languages as C++, we cannot rely on this property. We leave to future work to do sound deadlock prediction for C++.

Finally, while we have added two kinds of events to our notion of history compared to what RVPredict uses, we believe that an even more detailed model can help improve the model of consistency, hence improve the predictions.

ACKNOWLEDGMENTS

We thank Jeff Huang for helpful discussions, and we thank John Bender and the anonymous OOPSLA reviewers for helpful comments. Our research was supported by DARPA under agreement FA8750-15-2-0009, subcontract RF228-G1:3.

REFERENCES

Saddek Bensalem and Klaus Havelund. 2006. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing (HVC'05)*. Springer-Verlag, Berlin, Heidelberg, 208–223. https://doi.org/10.1007/11678779_15

Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.

Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. 2011. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceeding of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 241–250. https://doi.org/10.1145/1985793.1985827

Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).

Yan Cai and WK Chan. 2012. MagicFuzzer: scalable deadlock detection for large-scale applications. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 606–616.

Yan Cai and WK Chan. 2014. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs. *IEEE Transactions on Software Engineering* 40, 3 (2014), 266–281.

Yan Cai, Changjiang Jia, Shangru Wu, Ke Zhai, and Wing Kwong Chan. 2015. ASN: a dynamic barrier-based approach to confirmation of deadlocks from warnings for large-scale multithreaded programs. *IEEE Transactions on Parallel and Distributed Systems* 26, 1 (2015), 13–23.

Y. Cai and Q. Lu. 2016. Dynamic Testing for Deadlocks via Constraints. *IEEE Transactions on Software Engineering* 42, 9 (Sept 2016), 825–842. https://doi.org/10.1109/TSE.2016.2537335

Yan Cai, Shangru Wu, and WK Chan. 2014. ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 491–502.

Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. 2008. jPredictor: A Predictive Runtime Analysis Tool for Java. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, New York, NY, USA, 221–230. https://doi.org/10.1145/1368088.1368119

Zhi Da Luo, Raja Das, and Yao Qi. 2011. Multicore SDK: A practical and efficient deadlock detector for real-world applications. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*. IEEE, 309–318.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

Claudio DeMartini, Radu Iosif, and Riccardo Sisto. 1999. A deadlock detection tool for concurrent Java programs. *Software: Practice and Experience* 29, 7 (1999), 577–603.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.

Eelco Dolstra and Andres Löh. 2008. NixOS: A purely functional Linux distribution. In *ACM Sigplan Notices*, Vol. 43. ACM, 367–378.

Matthew B Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Ranganath, Todd Wallentine, et al. 2006a. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 73–89.

Matthew B Dwyer, Suzette Person, and Sebastian Elbaum. 2006b. Controlling factors in evaluating path-sensitive error detection techniques. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 92–104.

Dawson Engler and Ken Ashcraft. 2003. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM, 237–252.

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*. ACM, 353–365.

Jerry J Harrow. 2000. Runtime checking of multithreaded applications with visual threads. In *International SPIN Workshop on Model Checking of Software*. Springer, 331–342.

Klaus Havelund. 2000. Using runtime analysis to guide model checking of Java programs. In *International SPIN Workshop on Model Checking of Software*. Springer, 245–264.

Klaus Havelund and Thomas Pressburger. 2000. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer (STTT)* 2, 4 (2000), 366–381.

Gerard J. Holzmann. 1990. Design and Validation of Protocols. *Tutorial Computer Networks and ISDN Systems* 25 (1990), 981–1017.

Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 165–174.

Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic predictive concurrency analysis. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 847–857.

Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal Sound Predictive Race Detection with Control Flow Abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM Sigplan Notices*, Vol. 44. ACM, 110–120.

Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 157–170.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. Jacontebe: A benchmark suite of real-world java concurrency bugs (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 178–189.

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 386–396.

Mahmoud Said, Chao Wang, Zijiang Yang, Karem Sakallah, and Karem Sakallahg. 2011. Generating data race witnesses by an SMT-based analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6617 LNCS (2011), 313–327. https://doi.org/10.1007/978-3-642-20398-5_23

Traian Florin Serbanuta, Feng Chen, and Grigore Rosu. 2008. *Maximal Causal Models for Multithreaded Systems*. Technical Report UIUCDCS–R–2008–3017. University of Illinois at Urbana-Champaign.

Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. 2012. Maximal causal models for sequentially consistent systems. In *International Conference on Runtime Verification*. Springer, 136–150.

Traian Florin Serbanuta, Feng Chen, and Grigore Rosu. 2012. Maximal causal models for sequentially consistent systems. In *International Conference on Runtime Verification*. Springer, 136–150.

Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound predictive race detection in polynomial time. In *ACM Sigplan Notices*, Vol. 47. ACM, 387–400.

Francesco Sorrentino. 2015. PickLock: A deadlock prediction approach under nested locking. In *Model Checking Software*. Springer, 179–199.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model checking programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–272.