Precise Race Detection and Efficient Model Checking Using Locksets

Tayfun Elmas
Koc Univ
telmas@ku.edu.tr

Shaz Qadeer
Microsoft
qadeer@microsoft.com

Serdar Tasiran
Koc Univ
stasiran@ku.edu.tr

March 2006

Technical Report
MSR-TR-2005-118

In this paper, we present a new algorithm for detecting data-races in an execution of a concurrent program. Our algorithm is sound and precise, that is, it reports a race in an execution iff there are two accesses to a shared variable along the execution that are not ordered by the happens-before relation. Previous algorithms for computing the happens-before relation are based on clock vectors. On the other hand, our algorithm is based solely on the concept of locksets and is able to capture all mutual-exclusion synchronization idioms uniformly with one mechanism. Our lockset algorithm could be very useful for improving the precision of flow-sensitive static analyses, particularly those for detecting data-races and atomicity violations in concurrent programs. We present one such analysis, a model checking algorithm that uses our lockset algorithm both to check for races exhaustively and perform partial-order reduction when races are absent. Our characterization of the happens-before relation in terms of locksets rather than clock vectors is crucial for the fixpoint computation inherent in model checking and other flow-sensitive analyses. We have implemented our algorithm and used it to prove the absence of data-races and assertion failures on a number of examples containing a variety of synchronization idioms.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Many software systems depend critically on concurrent components for performance. Such systems are used extensively, which makes their functional correctness very important. To improve the reliability of these systems, we need efficient and easy-to-use analysis and verification tools specific to concurrency-related problems. Detection of race conditions on shared data is a central issue in such tools. Even though some race conditions may be benign, awareness of race conditions or their absence allows programmers to optimize their programs and to make them safer. Numerous techniques and tools have been developed to analyze races and to guard against them [28, 32, 5, 3].

Race conditions on shared data variables are usually defined in terms of the happens-before relation. The Java memory model [18], for instance, defines the important notion of a race-free execution in terms of the happens-before relation. An action α_1 happens-before another action action α_2 in a concurrent execution if α_1 occurs before α_2 and either both α_1 and α_2 are performed by the same thread or α_1 is transitively connected to α_2 by a series of synchronization actions, such as acquire or release of a lock, or fork of a thread, or join with a thread.

The connection between races, the happens-before relation, and locks has given rise to two categories of race-detection algorithms: vector-clock based and lockset-based. Vector-clock based race detection algorithms are precise but, when a race is detected, they fail to provide easy-to-interpret information for understanding and fixing the race condition [32]. Moreover, as explained later in this paper, these algorithms can not be naturally used as the basis for flow-sensitive static analyses, such as stateful model checking.

Lockset-based race-detection algorithms are more intuitive and capture directly the locking discipline employed by the programmer, but existing lockset algorithms have other shortcomings. These algorithms are specific to a particular locking discipline. For instance, the classic lockset algorithm popularized by the Eraser tool [28], is based on the assumption that each potentially shared variable must be protected by a single lock throughout the whole computation. For many realistic programs this assumption is false and leads to the reporting of a false race. Other similar algorithms can handle more sophisticated locking mechanisms [2, 7] by incorporating knowledge of these mechanisms into the lockset inference rules. They may still report false races when the particular locking discipline they are tracking is violated.

In this paper we provide, for the first time, a precise characterization of the happens-before relation in terms of locksets. This characterization enables us to formulate a *necessary and sufficient* condition for race-freedom based solely on the well-understood and simple concept of locksets. Our method can uniformly handle a variety of synchronization idioms such as thread-local data that later becomes shared, shared data protected by different locks at different points in time, and data protected indirectly by locks on container objects. Moreover, this generality is accomplished without explicit reference to the particular locking discipline into the lockset inference rules.

The primary application of our lockset algorithm is flow-sensitive analyses for concurrent software. We present one such analysis based on systematic state exploration. Our lockset algorithm is used continuously during state exploration, serving the dual purposes of checking for races exhaustively and performing partial-order reduction when races are absent. In a spirit similar to the dynamic partial-order reduction technique of Flanagan and Godefroid [12], our algorithm performs partial-order reduction optimistically but achieves soundness by exploring more thread interleavings when a race is detected.

Our characterization of the happens-before relation in terms of locksets (as opposed to clock vectors) is crucial because it allows our model checking algorithm to cache visited states and perform fixpoint computation. State caching is a fundamental optimization for programs which naturally have cycles and reconvergence in their state transition graphs. When a state is reached during an execution, all the history information required for detecting races and performing partial-order reduction is captured concisely by the locksets associated with that state. When the same state is revisited via a different path, it is possible to decide by comparing the locksets in the two states whether the search needs to re-explore executions starting from that state. Thus, in contrast to the purely stateless approach of Flanagan and Godefroid [12] based on vector clocks, our algorithm can perform both stateless and stateful search efficiently.

We believe that our lockset algorithm is useful not just for model checking but also for other dataflow and type-based analyses for concurrent programs. Several static analysis techniques [5, 3, 11] already use locksets and inherit the imprecision of the lockset inference algorithms they are based on. Since our algorithm is precise, we think it, rather than the classic lockset algorithm, should form the basis of such analyses.

This paper is organized as the following. Section 2 introduces our lockset algorithm and its application to interesting scenarios in example programs. Concurrent programs are formally described in Section 3. Section 4 gives a formal description of the lockset algorithm. The model checking algorithm that leverages the lockset algorithm is presented in Section 5. Section 6 explains how the lockset algorithm can be extended for locking disciplines that allow concurrent reads. The related work on race detection and partial-order reduction algorithms are given in Section 8 and the paper is concluded with Section 9.

2 Overview

In this section, we present examples that illustrate our algorithm's novel features and contrast it with existing lockset algorithms. In the following, s denotes a program state reached during an execution of the program, q denotes a shared variable, and t denotes a thread. $LH_s(t)$ is the set of locks held by t at s, and $LS_s^{alg}(q)$ is the set of locks that algorithm alg believes protect access to q. Lockset-based race-detection algorithms declare the existence of a race condition when $LH_s(t) \cap LS_s^{alg}(q)$ is empty. They differ in how they infer locksets, i.e., how they compute $LS_s^{alg}(q)$.

Lockset algorithms in the literature are too conservative in how they update LS^{alg} during an execution. For instance, the standard lockset algorithm (denoted by std) is based on the assumption that each shared variable is protected by a fixed unique lock throughout the execution. It attempts to infer this lock by setting $LS^{std}(q)$ to the intersection $LH(t) \cap LS^{std}(q)$ at each access to q by thread t. If this intersection becomes empty, it reports a race. Clearly, std is too conservative since it reports a false race if the lock protecting a variable changes over time. A toy example illustrating this scenario is given below.

```
  T1                 T2                 T3
------             -------            ------
acq(m1)            acq(m2)            acq(m1)
acq(m2)            acq(m3)            acq(m3)
x++                x++                x++
rel(m1)            rel(m2)            rel(m1)
rel(m2)            rel(m3)            rel(m3)
```

The code executed by each thread `Ti` is listed underneath the heading `Ti`. In the interleaving in which all actions of `T1` are completed followed by all actions of `T2` followed by all actions of `T3`, when `T3` accesses `x`, the standard algorithm declares a race since $LS^{std}(\mathtt{x}) = \mathtt{m}2$ before this access and `T3` does not hold $\mathtt{m}2$.

A less conservative alternative, denoted by algorithm *lsa*, is to set $LS^{lsa}(q)$ to $LH(t)$ after a race-free access to q by a thread t. This choice results in a less pessimistic sufficient condition but is still too conservative. In the example above, *lsa* will not report a race, but it will report a false race in the example below.

```
Class IntBox {
    Int x;
}

IntBox a = new IntBox;  // IntBox object o1 created
IntBox b = new IntBox;  // IntBox object o2 created

Lock ma, mb;

  T1                 T2                 T3
------             -------            ------
acq(ma)            acq(ma);           acq(mb);
a.x++;             acq(mb);           b.x++;
rel(ma);           tmp = a;           rel(mb);
                   a = b;
                   b = tmp;
                   rel(ma);
                   rel(mb);
```

Consider again the interleaving in which all actions of `T1` are completed, followed by those of `T2` and `T3` as above. `T2` swaps the objects referred to by variables `a` and `b`, so that during `T3`'s actions, `b` refers to `o1`. `o1.x` is initially protected by `ma` but is protected by `mb` after `T2`'s actions. *lsa* is unable to infer the correct new lock for `o1.x` since `T2` makes no direct access to `o1.x` and $LS^{lsa}(\mathtt{o}1.x)$ is not modified by `T2`'s actions.

Our algorithm's lockset update rules allow $LS^{our}(q)$ to grow and change during the execution and, in this way, we are able avoid false alarms. For instance, in the example above, after `T1` accesses `a.x`, our algorithm would associate `ma` with `o1.x`. Then, when `T2` acquires `ma` and `mb`, our algorithm would grow the lockset associated with `o1.x` (and all other objects with non-empty locksets at that point) to include both `ma` and `mb`. As a result, during `T3`'s access to `o1.x`, $LS^{our}(\mathtt{b.x})$ would include `mb` as well and no race will be reported.

In the following subsection, we illustrate our algorithm's lockset update rules step by step on a task queue example.

2.1 The Task Queue

Consider the task queue example for which pseudocode is provided in Figure 1. This example demonstrates a program that schedules tasks through a queue named `taskQueue` and executes it one by one. Each task of instance `Task` contains an array `subTasks` of subtasks. The computation for a single subtask is represented by a function `Perform` that takes a subtask and produces an integer output. The sum of all the outputs are the final result of the task and is also stored in its `out` field.

`CreateTask`, given an array of subtasks, creates a new task and enqueues it in the task queue. `PerformNextTask` dequeue a task from `taskQueue` and calls `ParallelTaskHelper`, which actually performs the task. `ParallelTaskHelper` forks for each subtask a new thread that runs `PerformSubTask`. `PerformSubTask` computes the partial result for the given subtask and adds it to the final result of the task.

This example is interesting because it makes use of thread locality, dynamically changing locksets, fork and join operations to ensure mutually exclusive access.

Consider the following interleaving of actions during a temporal scenario, which begins with creation of two threads **`T1`** and **`T2`**:

1. A thread **`T1`**, by running `CreateTask` with two subtasks as input,
   (a) creates a new task `task` by calling its constructor (line 1),
   (b) acquires `Qlock` and calls `taskQueue.Enqueue` (`task`) (lines 2-4).
2. A second thread **`T2`**, by running `PerformNextTask`,
   (a) acquires `Qlock` and calls `taskQueue.Dequeue` () that returns `task` (lines 1-3), and
   (b) calls `ParallelTaskHelper` (`task`) (line 4). `ParallelTaskHelper` creates two threads **`T3`** and **`T4`**, each for one subtask (lines 1-2).
3. The first thread **`T3`**, by running `PerformSubTask` (`task`, 0),
   (a) calls `Perform` (`task.subTasks` [0]) (line 1), acquires `Tlock`, and
   (b) adds `subTaskResult` to `task.out` (lines 2-4).
4. The second thread **`T4`**, by running `PerformSubTask` (`task`, 1),
   (a) calls `Perform` (`task.subTasks` [1]) (line 1), acquires `Tlock`, and
   (b) adds `subTaskResult` to `task.out` (lines 2-4).
5. Thread **`T2`**, continuing running `ParallelTaskHelper`,
   (a) joins both threads **`T3`** and **`T4`** (lines 3-4),
   (b) prints `task.out`.

Let us focus on the shared variable **`task.out`** for a particular task `task`. In the execution described above, there is no race on **`task.out`** but the lock protecting it changes dynamically. For example, `task.out` is local to **`T1`** at the beginning and to **`T2`** at the end of the scenario. We now show how our lockset algorithm handles this execution. Each item below explains how $LS(\mathbf{task.out})$ changes after each action during the scenario. $LS(\mathbf{task.out})$ is initialized to the set of all locks and thread identifiers. Our algorithm handles thread-locality by treating thread identifiers similar to locks, and allowing LH and LS to contain thread identifiers.

1. (a) In the constructor of `Task`, `task.out` is first accessed by **`T1`**. At this point $LH(\mathbf{T1}) = \{\mathbf{T1}\}$ and $LS(\mathbf{task.out}) = Addr \cup Tid$. Then we check $LS(\mathbf{task.out}) \cap LH(\mathbf{T1}) = \{\mathbf{T1}\}$. Since the intersection is not empty, $LS(\mathbf{task.out})$ is assigned $LH(\mathbf{T1})$, such that $LS(\mathbf{task.out}) = \{\mathbf{T1}\}$.
   (b) After **`T1`** acquires `Qlock`, $LH(\mathbf{T1}) = \{\mathbf{T1}, \mathtt{Qlock}\}$. We check $LS(\mathbf{task.out}) \cap LH(\mathbf{T1}) = \{\mathbf{T1}\}$. Since the intersection is not empty, $LS(\mathbf{task.out})$ is added $LH(\mathbf{T1})$, such that $LS(\mathbf{task.out}) = \{\mathbf{T1}, \mathtt{Qlock}\}$.
2. (a) After **`T2`** acquires `Qlock` $LH(\mathbf{T2}) = \{\mathbf{T2}, \mathtt{Qlock}\}$. We check $LS(\mathbf{task.out}) \cap LH(\mathbf{T2}) = \{\mathtt{Qlock}\}$. Since the intersection is

```
class Task {
  SubTask[n] subTasks;
  int out;
  Task(SubTask[] sT) { subTasks = sT; out = 0; }
}

Queue<Task> taskQueue;

CreateTask(subTask[] sTs)
  oneTask = new Task(sTs);
  acquire(Qlock)
     taskQueue.Enqueue(oneTask);
  release(Qlock);

PerformNextTask()
  acquire(Qlock)
     oneTask = taskQueue.Dequeue();
  release(Qlock);
  ParallelTaskHelper(oneTask);

ParallelTaskHelper(oneTask)
  foreach (i < n)
    children[i] = fork(PerformSubTask, oneTask, i);
  foreach (i < n)
    join(children[i]);
  print(oneTask.out);

PerformSubTask(oneTask, i)
  subTaskResult = Perform(oneTask.subTasks[i]);
  acquire(Tlock);
     oneTask.out += subTaskResult;
  release(Tlock);
```

Figure 1. Pseudocode for the task queue example.

not empty, $LS(\texttt{task.out})$ is added $LH(\texttt{T1})$ such that $LS(\texttt{task.out}) = \{\texttt{T1}, \texttt{T2}, \texttt{Qlock}\}$.

(b) After a **T2** creates **T3**, we check $LS(\texttt{task.out}) \cap LH(\texttt{T2}) = \{\texttt{T2}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is added $\{\texttt{T3}\}$, such that $LS(\texttt{task.out}) = \{\texttt{T1}, \texttt{T2}, \texttt{Qlock}, \texttt{T3}\}$. The same update applies when **T2** creates **T4** such that $LS(\texttt{task.out}) = \{\texttt{T1}, \texttt{T2}, \texttt{Qlock}, \texttt{T3}, \texttt{T4}\}$.

3. (a) After **T3** acquires Tlock, $LH(\texttt{T3}) = \{\texttt{T3}, \texttt{Tlock}\}$. We check $LS(\texttt{task.out}) \cap LH(\texttt{T3}) = \{\texttt{T3}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is added $LH(\texttt{T3})$ such that $LS(\texttt{task.out}) = \{\texttt{T1}, \texttt{T2}, \texttt{Qlock}, \texttt{T3}, \texttt{T4}, \texttt{Tlock}\}$.

   (b) After task.out is written we check $LS(\texttt{task.out}) \cap LH(\texttt{T3}) = \{\texttt{T3}, \texttt{Tlock}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is assigned $LH(\texttt{T3})$ such that $LS(\texttt{task.out}) = \{\texttt{T3}, \texttt{Tlock}\}$.

4. (a) After **T4** acquires Tlock, $LH(\texttt{T4}) = \{\texttt{T4}, \texttt{Tlock}\}$. We check $LS(\texttt{task.out}) \cap LH(\texttt{T4}) = \{\texttt{Tlock}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is added $LH(\texttt{T4})$such that $LS(\texttt{task.out}) = \{\texttt{T3}, \texttt{Tlock}, \texttt{T4}\}$.

   (b) After task.out is written we check $LS(\texttt{task.out}) \cap LH(\texttt{T4}) = \{\texttt{T4}, \texttt{Tlock}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is assigned $LH(\texttt{T4})$ such that $LS(\texttt{task.out}) = \{\texttt{T4}, \texttt{Tlock}\}$.

5. (a) After a **T2** joins **T4**, we check $LS(\texttt{task.out}) \cap LH(\texttt{T4}) = \{\texttt{T4}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is added $\{\texttt{T2}\}$ such that $LS(\texttt{task.out}) = \{\texttt{T4}, \texttt{Tlock}, \texttt{T2}\}$.

   (b) After task.out is accessed by print, we check $LS(\texttt{task.out}) \cap LH(\texttt{T2}) = \{\texttt{T2}\}$. Since the intersection is not empty, $LS(\texttt{task.out})$ is assigned $LH(\texttt{T2})$ such that $LS(\texttt{task.out}) = \{\texttt{T2}\}$.

The description above illustrates although $LS(\texttt{task.out})$ shrinks at an access to $LS(\texttt{task.out})$, it can grow whenever a thread executes an acquire, fork, or join operation. It is this ability to grow the lockset that is fundamental to capturing dynamic locking idioms.

$$
\begin{array}{rcrcl}
pc & \in & PC & & \\
a & \in & Addr & & \\
t & \in & Tid & & \\
v & \in & Value & = & PC \cup Addr \cup Tid \cup Integer \\
f & \in & Field & & \\
(a,f) & \in & HeapVariable & = & Addr \times Field \\
x,y,z & \in & LocalVar & & \\
\alpha, \alpha_1 & & Action & = & x = new \mid y = x.f \mid x.f = y \\
 & & & & \mid x = op(y_1, \ldots, y_m) \\
 & & & & \mid acq(x) \mid rel(x) \\
 & & & & \mid x = fork \mid join(x) \\
h & \in & Heap & = & Addr \to Field \to Value \\
l & \in & LocalStore & = & LocalVar \to Value \\
\ell & \in & LocalState & = & PC \times LocalStore \\
\ell s & \in & LocalStates & = & Tid \to LocalState \\
(\ell s, h) & \in & State & = & Heap \times LocalStates
\end{array}
$$

Figure 2. Domains

3 Concurrent programs

In this section, we present a simple formalization of concurrent programs that will allow us to describe our algorithms precisely and succinctly. A concurrent program essentially consists of a set of threads, each of which executes a sequence of operations. These operations include local computation involving thread-local variables, reading and writing shared variables on the heap, and synchronization operations such as acquiring and releasing mutex locks, forking a thread, and joining with a thread. We give more details below.

Program state: A state of a program is a pair $(\ell s, h)$. The partial function $\ell s : Tid \to LocalState$ maps a thread identifier t to the local state of thread t. The set Tid is the set of thread identifiers. The local state $\ell s(t)$ is a pair $\langle pc, l \rangle$ consisting of the control location pc and a valuation l to the *local variables* of thread t. The heap h is a collection of cells each of which has a unique address and contains a finite set of fields. The set $Addr$ is the set of heap addresses. Formally, the heap h is a partial function mapping addresses to a function that maps fields to values. Given address $a \in Addr$ and field $f \in Field$, the value stored in the field f of cell with address a is denoted by $h(a, f)$. The pair (a, f) is called a *heap variable* of the program. Heap variables are shared among the threads of the program, and thus, operations on these are visible to all threads. Each local variable or field of a cell may contain values from the set $Tid \cup Addr \cup Integer$.

Actions: An action $\alpha \in Actions$ is an operation that is guaranteed to be performed atomically by the executing thread. The action $x = new$ allocates a new object on the heap and stores its address in the local variable x. The action $y = x.f$ reads into y the value contained in the f field of the object whose address is in x. If x does not contain the address of a heap object, this action goes wrong. Similarly, the action $x.f = y$ stores a value into a field of a heap object. The action $x = op(y_1, \ldots, y_n)$ models local computation where $op(y_1, \ldots, y_n)$ is either an arithmetic or boolean function over the local variables $y_1, \ldots, y_n$.

Every object on the heap has a lock associated with it. This lock is modeled using a special field *owner* that is accessible only by the *acq* and *rel* actions. The action $acq(x)$ acquires the lock on the object whose address is contained in x. This action is enabled only if $x.owner = 0$ and it sets $x.owner$ to the identifier of the executing thread. The action $rel(x)$ releases the lock on the object whose address is contained in x by setting $x.owner$ to 0. This action goes wrong if the value of $x.owner$ is different from the identifier of the executing thread.

The action $x = fork$ creates a new thread and stores its identifier into x. The local variables of the child thread are a copy of the local variables of the parent thread. The action $join(x)$ is enabled only if the thread whose identifier is contained in x has terminated.

Control flow graph: The behavior of the program is specified by a control flow graph over a set PC of control locations. A labeling function $Label : PC \to LocalVar$ labels each location with a local variable. The set of control flow edges are specified by two functions $Then : PC \to Action \times (PC \cup \{end, wrong\})$ and $Else : PC \to Action \times (PC \cup \{end, wrong\})$. Suppose $Label(pc) = x$, $Then(pc) = (\alpha_1, pc_1)$, and $Else(pc) = (\alpha_2, pc_2)$. When a thread is at the location pc, the next action executed by it depends on the value of x. If the value of x is nonzero, then it executes the action α_1 and goes to pc_1. If the value of x is zero, then it executes the action α_2 and goes to pc_2. A thread *terminates* and cannot perform any more actions if it reaches one of the special locations *end* or *wrong*. The location *end* indicates normal termination and *wrong* indicates erroneous termination. The control location *wrong* may be reached, for example, if the threads fails an assertion or if it attempts to access a field of non-address value.

Transition relation: We now formally define the semantics of the program as a transition relation $\xrightarrow{\alpha}_t \subseteq State \times State$, where $t \in Tid$ is a thread identifier and $\alpha \in Action$ is an action. This relation gives the transitions of thread t. Program execution starts with a single thread with identifier $t_I \in Tid$ at control location pc_I. The initial state of the program is $(\ell s_I, h_I)$, where $\ell s_I(t_I) = \langle pc_I, l_I \rangle$ and undefined elsewhere, and the heap h_I is not defined at any address. The initial local store l_I of thread t_I assigns 0 to each variable. In each step, a nondeterministically chosen thread t executes an action α and changes the state according to the transition relation $\xrightarrow{\alpha}_t$. Let $(\ell s, h)$ be a state such that $\ell s(t) = \langle pc, l \rangle$ and $Label(pc) = z$. Let $\langle \alpha, pc' \rangle = Then(pc)$ if $l(z) \neq 0$ and $Else(pc)$ otherwise. Then, the relation $\xrightarrow{\alpha}_t$ is given by the rules in Figure 3 where we do a case analysis on α. An execution σ of the program is a finite sequence $(\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell s_n, h_n) \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$ such that $(\ell s_1, h_1) = (\ell s_I, h_I)$ and $(\ell s_k, h_k) \xrightarrow{\alpha_k}_{t_k} (\ell s_{k+1}, h_{k+1})$ for all $1 \leq k \leq n$.

(ALLOCATE)
$$\frac{\alpha = (x = new) \quad h(a) = \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l[x := a] \rangle], h[a := \lambda_I])}$$

(READHEAP)
$$\frac{\alpha = (y = x.f) \quad h(l(x)) \neq \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l[y := h(l(x), f)] \rangle], h)}$$

(READHEAP FAIL)
$$\frac{\alpha = (y = x.f) \quad h(l(x)) = \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$$

(WRITEHEAP)
$$\frac{\alpha = (x.f = y) \quad h(l(x)) \neq \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle], h[(l(x), f) := l(y)])}$$

(WRITEHEAP FAIL)
$$\frac{\alpha = (x.f = y, pc') \quad h(l(x)) = \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$$

(OPERATION)
$$\frac{\alpha = (x = op(y_1, \ldots, y_m))}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l[x := op(l(y_1), \ldots, l(y_m))] \rangle], h)}$$

(ACQUIRE)
$$\frac{\alpha = acq(x) \quad h(l(x), owner) = 0}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle], h[(l(x), owner) := t])}$$

(ACQUIRE FAIL)
$$\frac{\alpha = acq(x) \quad h(l(x)) = \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$$

(RELEASE)
$$\frac{\alpha = rel(x) \quad h(l(x), owner) = t}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle], h[(l(x), owner) := 0])}$$

(RELEASE FAIL)
$$\frac{\alpha = rel(x) \quad (h(l(x)) = \bot \vee h(l(x), owner) \neq t)}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$$

(FORK)
$$\frac{\alpha = (x = fork) \quad \ell s(u) = \bot}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle][u := \langle pc_I, l \rangle], h)}$$

(JOIN)
$$\frac{\alpha = join(x) \quad \ell s(l(x)) = \langle end, l' \rangle}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle], h)}$$

Figure 3. Transition relation

4 Lockset algorithm

In this section, we describe our algorithm for checking whether a given execution σ has a data-race. We use the standard characterization of data-races based on the happens-before relation. Our algorithm is sound and precise, that is, it reports a data-race on an execution iff there is a data-race in that execution. The novelty of our algorithm is that it is based on locksets, in contrast with traditional algorithms that are based on clock vectors. We will show that this aspect of our algorithm gives it significant advantages over traditional approaches. We first present the definition of the happens-before relation.

DEFINITION 1. *Let* $\sigma = (\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$ *be an execution of the program. The happens-before relation* $\xrightarrow{hb}$ *for* σ *is the smallest transitively-closed relation on the set* $\{1, 2, \ldots, n\}$ *such that for any* k *and* l*, we have* $k \xrightarrow{hb} l$ *if* $1 \leq k \leq l \leq n$ *and one of the following holds:*

Initialization:
LS = $\lambda q \in \mathit{HeapVariable}.\ \mathit{Addr} \cup \mathit{Tid}$

Let $(\ell s, h) \xrightarrow{\alpha}_t (\ell s', h')$, $\ell s(t) = \langle pc, l \rangle$ and $\ell s'(t) = \langle pc', l' \rangle$.

1. $\alpha = (x = \mathit{new})$ or $\alpha = (x = \mathit{op}(y_1, \ldots, y_m))$:
   LS is not updated.
2. $\alpha = (y = x.f)$ or $\alpha = (x.f = y)$:
   let $lh = LH((\ell s, h), t)$ *in*
   $LS = LS[(l(x), f) := lh]$
3. $\alpha = \mathit{acq}(x)$:
   let $lh = LH((\ell s', h'), t)$ *in*
   $LS = \lambda q \in \mathit{HeapVariable}.\ (lh \cap LS(q) \neq \emptyset)\ ?\ lh \cup LS(q)\ :\ LS(q)$
4. $\alpha = \mathit{rel}(x)$:
   LS is not updated.
5. $\alpha = (x = \mathit{fork})$:
   let $lh = LH((\ell s, h), t)$ *in*
   $LS = \lambda q \in \mathit{HeapVariable}.\ (lh \cap LS(q) \neq \emptyset)\ ?\ \{l'(x)\} \cup LS(q)\ :\ LS(q)$
6. $\alpha = \mathit{join}(x)$:
   let $lh = LH((\ell s, h), l(x)),\ lh' = LH((\ell s, h), t)$ *in*
   $LS = \lambda q \in \mathit{HeapVariable}.\ (lh \cap LS(q) \neq \emptyset)\ ?\ lh' \cup LS(q)\ :\ LS(q)$

Figure 4. Update rules for the lockset algorithm

1. *$t_k = t_l$.*
2. *$\alpha_k = \mathit{rel}(x)$, $\alpha_l = \mathit{acq}(y)$, and $\ell s_k(t_k)(x) = \ell s_l(t_l)(y)$.*
3. *$\alpha_k = (x = \mathit{fork})$ and $t_l = \ell s_{k+1}(t_k)(x)$.*
4. *$\alpha_l = \mathit{join}(x)$ and $t_k = \ell s_l(t_l)(x)$.*

We use the happens-before relation to define data-race free executions as follows. Consider an action α_k in the execution σ and a heap variable $q = (\ell s_k(t_k)(x), f)$. The thread t_k *reads* q, if $\alpha_k = (x = y.f)$. The thread t_k *writes* q, if $\alpha_k = (x.f = y)$. The thread t_k *accesses* the variable q if it either reads or writes q. The execution σ is *race-free* on q if for all $k, l \in [1, n]$ such that α_k and α_l access q, we have $k \xrightarrow{hb} l$. For now, our definition does not distinguish between read and write accesses. In Section 6, we will refine our algorithm to make this distinction.

The Java memory model [18] also defines data-race free executions in a manner similar to us. However, their definition of a happens-before relation also includes all edges between accesses to a volatile variable. Although our programming language does not include volatile variables, their effect on the happens-before relation can be modeled easily by introducing for each volatile variable q a new lock p and inserting an acquire of p before and a release of p after each access to q.

Our algorithm for detecting data races in an execution σ uses two auxiliary functions, LH and LS. The function LH from Tid to $\mathit{Powerset}(\mathit{Addr} \cup \mathit{Tid})$ provides for each thread t the set of locks held by t. Apart from the locks present in the program, our algorithm also considers each thread identifier t to be a lock that is held by that thread for its lifetime. Given a state $(\ell s, h)$ and a thread t, we formally define $LH((\ell s, h), t) = \{t\} \cup \{a \in \mathit{Addr} \mid h(a, \mathit{owner}) = t\}$. We often write $LH(t)$ when the state $(\ell s, h)$ is clear from the context. The function LS from $\mathit{HeapVariable}$ to $\mathit{Powerset}(\mathit{Addr} \cup \mathit{Tid})$ provides for each variable q its lockset $LS(q)$ which contains the set of locks that potentially protect accesses to q. The algorithm updates LS with the execution of each transition in σ. These updates to LS maintain the invariant that if thread t holds at least one lock in $LS(q)$ at an access of q, then the previous access to q is related to this access by the happens-before relation.

Our algorithm consists of the set of rules in Figure 4. Initially $LS(q) = \mathit{Addr} \cup \mathit{Tid}$ for all $q \in \mathit{HeapVariable}$. Given as input a transition $(\ell s, h) \xrightarrow{\alpha}_t (\ell s, h)$, the rules in the figure show how to update LS by a case analysis on α. A race on the heap variable $q = (l(x), f))$ is reported in Rule 2, if $LS(q) \cap LH((\ell s, h), t) = \emptyset$ just before the update.

The computation of the function LH in any state requires a single scan of the heap. If that is too expensive, the function LH can be easily computed incrementally by the algorithm as follows. We initialize $LH(t) = \{t\}$ for all $t \in \mathit{Tid}$. At an acquire operation by thread t, we add the lock being acquired to $LH(t)$. At a release operation by thread t, we remove the lock being released from $LH(t)$.

To present the intuition behind our algorithm, let us consider the evolution of $LS(q)$ for a particular heap variable q starting from an access by thread t. According to Rule 2, this access sets $LS(q)$ to $LH(t)$. The other rules ensure that as the execution proceeds, the lockset $LS(q)$ grows or remains the same, until the next access to q is performed by a thread t', at which point $LS(q)$ is set to $LH(t')$. In other words, the invariant $LH(t) \subseteq LS(q)$ holds at the state after the access by t up to the state just before the next access by t'. Suppose $t' \neq t$. If $LS(q) \cap LH(t') \neq \emptyset$ just before the second access, then an argument based on the invariant shows that the two accesses are related by the happens-before relation. The real insight of our algorithm appears in ensuring the contrapositive, that is, in showing that if the first access happens before the second access, then $LS(q) \cap LH(t') \neq \emptyset$.

To illustrate how our algorithm ensures the contrapositive, consider the following scenario. Suppose $q = (o, f)$ and o is an object freshly allocated by t. Further, at the access of q by thread t no program locks were held so that $LH(t) = \{t\}$. Later on, thread t makes this object visible by acquiring the lock of a shared object o' and assigning the reference o to a field in o'. After t releases the lock o', thread t' acquires it, gets a reference to o, releases the lock o', and accesses the variable (o, f). In this case, there is a happens-before edge between the two accesses due to the release of o' by t and the acquire of o' by t'.

Our algorithm detects this happens-before edge by growing the lockset of q at each acquire operation. In Rule 3 for the acquire operation, the set lh of locks held by thread t after the acquire operation is added to the lockset $LS(q)$ of any variable q if there is a common lock between lh and $LS(q)$. As a consequence of this rule, when thread t acquires the lock o' in the example described above, the lock o' is added to $LS(q)$, updating it to $\{t, o'\}$. Similarly, when thread t' acquires the lock o', the lockset $LS(q)$ is updated to $\{t, o', t'\}$ and thus $LH(t') \cap LS(q) \neq \emptyset$ at the access of q by t'. The rationale for growing the locksets at fork and join operations in Rules 5 and 6 respectively is similar.

We have proved the following theorem about the correctness of our algorithm. This theorem shows that our algorithm is both sound

```
record Node {
  State state;
  (HeapVariable → Powerset(Tid ∪ Addr)) LS;
  (HeapVariable → Node) la;
  (Tid ∪ {0}) tid;
  Powerset(Tid) done;

    ((Addr ∪ Tid) → (Addr ∪ Tid)) f;
    Powerset(HeapVariable) races;
    Powerset(HeapVariable) va;
    boolean succOnStack;
}
    (State → HeapVariable → (Addr ∪ Tid)) table;
    (State → Powerset(HeapVariable)) rtable;
```

Figure 5. Record *Node*

and precise.

THEOREM 1 (CORRECTNESS). *Consider a program execution $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$. Let a heap variable q and $i \in [1, n-1]$ be such that α_i and α_n access q but α_j does not access q for all $j \in [i+1, n-1]$. Then $LS_n(q) \cap LH((\ell s_n, h_n), t_n) \neq \emptyset$ iff $i \xrightarrow{hb} n$.*

The proof of Theorem 1 depends on the following fundamental lemma that formally characterizes the relationship between the current lockset of each variable and the synchronization operations that occurred in the history of the execution.

LEMMA 1. *Let $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ be an execution of the program. Let q be a variable that was last accessed by action α_i in σ.*

1. *Let $m \in Addr$ be such that $m \notin LH((\ell s_{n+1}, h_{n+1}), t)$ for all $t \in Tid$. Then $m \in LS_{n+1}(q)$ iff there exists j such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$, $\alpha_j = rel(x)$, and $\ell s_j(t_j)(x) = m$.*

2. *Let $m \in Addr$ be such that $m \in LH((\ell s_{n+1}, h_{n+1}), t)$ for some t. Then $m \in LS_{n+1}(q)$ iff there exists j such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$ and $t_j = t$.*

3. *Let $t \in Tid$. Then $t \in LS_{n+1}(q)$ iff there exists j such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (x = fork)$ and $\ell s_{j+1}(t_j)(x) = t$.*

The proof of our correctness theorem appears in the appendix of the full version of our paper [1].

In the next section, we will utilize our lockset algorithm to devise an efficient model checking algorithm for concurrent programs. This algorithm can be used to find data-races and safety violations in the program by systematically exploring its state space.

5 Model checking using locksets

In this section, we present an application of the lockset algorithm described in the previous section. We develop an algorithm to systematically and efficiently explore the state space of a concurrent program. The main challenge in systematic exploration is to reduce the number of thread interleavings that need to be explored while maintaining soundness. Partial-order techniques [23] have employed the idea of *selective search* to achieve such a reduc-

```
Search() {
  Node curr = new Node;
  curr.state = (ℓs_I, h_I);
  curr.LS = λq ∈ HeapVariable. Addr ∪ Tid;
  curr.la = λq ∈ HeapVariable. null;
  curr.tid = 0;
  curr.done = ∅;

  Stack⟨Node⟩ stack = new Stack⟨Node⟩;
  stack.Push(curr);

    (Addr ∪ Tid) → (Addr ∪ Tid) f;
    curr.f = Canonize(curr.state);
    table(curr.f(curr.state)) = curr.f(curr.LS);
    curr.races = ∅;
    rtable(s) = ∅;
    curr.va = ∅;
    curr.succOnStack = false;

  while (¬stack.IsEmpty()) {
    Tid t;
    curr = stack.Peek();
    if (curr.tid = 0 ∧ done ⊂ enabled(curr.state))
      t = choose(enabled(curr.state) \ curr.done);
    elsif (curr.tid ≠ 0 ∧ curr.tid ∉ curr.done)
      t = curr.tid;
    else {
      stack.Pop();

        if (curr.tid = 0 ∧ curr.succOnStack) {
          foreach (HeapVariable q′. curr.la(q′) ≠ null)
            curr.la(q′).tid = 0;
          curr.races = HeapVariable;
        }
        rtable(curr.f(curr.state)) = curr.f(curr.races);
        Node prev = stack.Peek();
        prev.races = prev.races ∪ (curr.races \ curr.va);

      continue;
    }
    curr.done = curr.done ∪ {t};
    curr.tid = t;
    Node next = Successor(curr, t);

      ((Addr ∪ Tid) → (Addr ∪ Tid)) f = next.f;
      State s = f(next.state);
      if (table(s) exists) {
        (HeapVariable → (Addr ∪ Tid)) locksets;
        locksets = f^{-1}(table(s));
        if (locksets ⊆ next.LS) {
          if (∃n ∈ stack. s = n.f(n.state)) {
            curr.tid = 0;
            curr.succOnStack = true;
          }
          continue;
        }
        next.LS = locksets ∩ next.LS;
      }
      table(s) = f(next.LS);

    stack.Push(next);
  }
}
```

Figure 6. Procedure *Search*

tion. In each explored state s, these algorithms attempt to identify a thread t such that the operation of t enabled in s is independent of all operations in any execution from s consisting entirely of operations by threads other than t. If such a thread t is identified, then it suffices to schedule only t in s. The fundamental problem with these algorithms is that, since the executions in the future of s have not been explored, they are forced to make pessimistic guesses about independence. For example, if the operation of thread t is an access of a shared heap variable q, then a pessimistic analysis would declare it to be not independent (or dependent). But if this access by t and any future access by another thread consistently follow the locking discipline associated with q, then these two accesses are separated by the happens-before relation and consequently the access by thread t can be classified as an independent operation. The lockset algorithm described in the previous section is able to track the happens-before relation precisely and therefore gives us a powerful tool to identify such independent actions.

The model checking algorithm is implemented by the procedure *Search* in Figure 6 and procedure *Successor* in Figure 7. The procedure *Search* performs a depth-first search (DFS) of the state space using the *stack* variable declared on line 7. The DFS stack consists of a sequence of *Node* records each of which stores information associated with a state visited during the search. The state itself is stored in the field *state*. The search keeps track of the locksets for the heap variables in the field *LS* and executes the lockset algorithm along every execution generated by the search. The field *la* provides for each heap variable a reference to the node in the DFS stack from which the last access to that variable was performed. The fields *tid* and *done* determine the scheduling of threads from the node. The field *done* contains the identifiers of those threads that have already been scheduled from the node.

To schedule an action α of thread t from a node *curr* at the top of the stack, the field $curr.tid$ is set to t and the procedure *Successor* is invoked. This procedure returns the successor node *next*, which contains the new state and locksets. The value of $curr.la$ is copied over to $next.la$, except if α accesses a variable q in which case $next.la(q)$ is updated to point to *curr*. In the procedure *Search*, the action α is optimistically treated as an independent action. As the search proceeds, the value of $next.la(q)$ is copied to its successors on the stack. If a later action creates a data-race on q with α, then a reference to *curr* is retrieved using $la(q)$ and $curr.tid$ is set to 0. When *curr* is again at the top of the stack, the procedure *Search* observes that $curr.tid = 0$ and schedules other threads from *curr*. If, on the other hand, no race is discovered, then α is indeed an independent action and it is unnecessary to schedule other threads from *curr*.

The fields f, *races*, *va*, and *succOnStack* of *Node*, the variables *table* and *rtable*, lines 9–15, 25–32 and 38–52 of the procedure *Search*, and lines 21 and 27–34 of the procedure *Explore* are used to implement state caching in our algorithm. Indeed, by omitting these lines *Search* becomes a stateless model checking [24] algorithm which is sound but guaranteed to terminate only on finite acyclic state spaces. If these lines are included, then *Search* is a stateful model checking algorithm that is sound and guaranteed to terminate on all finite state spaces. Flanagan and Godefroid [12] earlier presented a stateless model checking algorithm, also based on optimistic partial-order reduction algorithm. To compute the happens-before relation, their algorithm augments the program state with clock vectors. Since the clock values in the vectors monotonically increase with the length of the execution, state caching would not be effective and their algorithm is limited to systematic but stateless execution. We significantly improve upon their work by giving the ability to perform both stateless and stateful model checking. As

```
Node Successor(Node curr, Tid t) {
1  Heap h, h′;
2  LocalStates ℓs, ℓs′;
3  Action α;
4  (h, ℓs) = curr.state;
5  let (h, ℓs) −α→_t (h′, ℓs′);
6  Node next = new Node;
7  next.state = (h′, ℓs′);
8  next.LS = Update(curr.LS, (h, ℓs) −α→_t (h′, ℓs′));
9  next.la = curr.la;
10 next.f = Canonize(next.state);
11 next.races = ∅;
12 next.va = ∅;

13 switch (α) {
14    case y = x.f :
15    case x.f = y :
16       HeapVariable q = (ℓs(t)(x), f);
17       next.va = {q};
18       next.la(q) = curr;
19       if (curr.LS(q) ∩ LH(curr.state, t) = ∅) {
20          curr.la(q).tid = 0;

21             curr.races = curr.races ∪ {q};

22       }

23    case acq(x) :
24    case join(x) :
25       curr.tid = 0;
26 }

27       ((Addr ∪ Tid) → (Addr ∪ Tid)) f = next.f;
28       State s = f(next.state);
29       if (rtable(s) exists) {
30          next.races = f^{-1}(rtable(s));
31          foreach (HeapVariable q′ ∈ next.races)
32             next.la(q′).tid = 0;
33          curr.races = curr.races ∪ (next.races \ next.va);
34       }

35 next.tid = (t ∈ enabled(next.state)) ? t : 0;
36 next.done = ∅;
37 next.succOnStack = false;
38 return next;
39}
```

Figure 7. Procedure *Successor*

described below, our characterization of the happens-before relation in terms of locksets is crucial for this improvement. Our algorithm, by virtue of being stateful, provides a guarantee of termination and the possibility of avoiding redundant state exploration.

The variable *table* is a map from states to locksets and is used to store the states together with the corresponding locksets explored by the algorithm. The variable *rtable* maps a state to the set of heap variables on which a race may occur in some execution starting from that state. An entry corresponding to state s is added to *table* when it is pushed on the stack (lines 11 and 52). Conversely, an entry corresponding to state s is added to *rtable* when it is popped from the stack (line 30).

The algorithm computes the canonical representatives of the initial state $(\ell s_I, h_I)$ and the initial locksets in lines 9–11. The canonical representatives capture symmetries in the state space due to the restricted operations allowed on the set *Addr* of heap addresses and the set *Tid* of thread identifiers. The canonical representatives are computed in two steps. First, the function *Canonize* is used to construct a *canonizer* f, a one-one onto function on $Addr \cup Tid$. Then, the states and the locksets are transformed by an application of this function. The canonizer is stored in the f field of *curr* and an entry from the representative of the initial state to the representative of the initial lockset is added to *table*. There are well-understood techniques for performing canonization [13, 29, 30] and we omit the details for lack of space.

The algorithm explores a transition on line 37 by calling the *Successor* procedure. This function returns the next state in the node *next*. If a race is detected on line 19 due to an access to a heap variable q, then the *tid* field of the node from which the last access to q was made is set to 0. In addition, lines 27–34 of *Successor* check if the future races from the successor state have already been computed. If they have, then those races are used to set the *tid* field of other stack nodes to 0.

After generating the successor node *next*, the *Search* procedure stores the canonizer of $next.state$ in $next.f$. If there is no entry corresponding to the canonical representative of $next.state$ in *table*, then it adds a new entry and pushes *next* on the stack. The most crucial insight of the algorithm appears in the case when an entry exists. In that case, the corresponding locksets are retrieved in the variable *locksets*. In line 43, the algorithm checks whether $locksets(q) \subseteq next.LS(q)$ for each heap variable q. If the check succeeds, then it is unnecessary to explore from $next.state$ since any state reachable from $next.state$ with locksets $next.LS$ is also reachable from $next.state$ with *locksets* and any race that happens from the state $next.state$ with locksets $next.LS$ also happens from $next.state$ with *locksets*.

Lines 44–47 take care of a well-known problem with partial-order techniques [23]. By setting $curr.tid$ to 0 in case $next.state$ is on the stack, the algorithm ensures that transitions of other threads get scheduled in the next iteration of the loop on line 19–20. In this case, the field $curr.succOnStack$ is also set to *true*. When a node is popped from the stack (line 24), if its *tid* field is 0 and *succOnStack* field is *true* (line 25–29), then the algorithm considers all races to be possible in the future and updates the *tid* fields of stack nodes appropriately.

Finally, if the subset check on line 43 fails, then the algorithm updates $next.LS$ to be the pointwise intersection of *locksets* and the old value of $next.LS$, updates *table* so it maps the canonical representative of $next.state$ to the canonical representative of the new value of $next.LS$, and finally pushes *next* on the stack.

Initialization:
$LSW = \lambda q \in HeapVariable.\ Addr \cup Tid$
$LSR = \lambda q \in HeapVariable.\ \lambda u \in Tid.\ Addr \cup Tid$

Let $(\ell s, h) \xrightarrow{\alpha}_t (\ell s', h')$, $\ell s(t) = \langle pc, l\rangle$ and $\ell s'(t) = \langle pc', l'\rangle$.

1. $\alpha = (x = new)$ or $\alpha = (x = op(y_1, \ldots, y_m))$:
   LSW and *LSR* are not updated.
2. $\alpha = (y = x.f)$:
   LSW is not updated.
   $let\ \ lh = LH((\ell s, h), t)\ in$
   $LSR = LSR[((l(x), f), t) := lh]$
3. $\alpha = (x.f = y)$:
   $let\ \ lh = LH((\ell s, h), t)\ in$
   $LSW = LSW[(l(x), f) := lh]$
   $LSR = LSR[(l(x), f) := \lambda u \in Tid.\ lh]$
4. $\alpha = acq(x)$:
   $let\ \ lh = LH((\ell s', h'), t)\ in$
   $LSW = \lambda q \in HeapVariable.\ (lh \cap LSW(q) \neq \emptyset)\ ?\ lh \cup LSW(q)\ :\ LSW(q)$
   $LSR = \lambda q \in HeapVariable. \lambda u \in Tid.\ (lh \cap LSR(q,u) \neq \emptyset)\ ?\ lh \cup LSR(q,u)\ :\ LSR(q,u)$
5. $\alpha = rel(x)$:
   LSW and *LSR* are not updated.
6. $\alpha = (x = fork)$:
   $let\ \ lh = LH((\ell s, h), t)\ in$
   $LSW = \lambda q \in HeapVariable.\ (lh \cap LSW(q) \neq \emptyset)\ ?\ \{l'(x)\} \cup LSW(q)\ :\ LSW(q)$
   $LSR = \lambda q \in HeapVariable. \lambda u \in Tid.\ (lh \cap LSR(q,u) \neq \emptyset)\ ?\ \{l'(x)\} \cup LSR(q,u)\ :\ LSR(q,u)$
7. $\alpha = join(x)$:
   $let\ \ lh = LH((\ell s, h), l(x)),\ lh' = LH((\ell s, h), t) in$
   $LSW = \lambda q \in HeapVariable.\ (lh \cap LSW(q) \neq \emptyset)\ ?\ lh' \cup LSW(q)\ :\ LSW(q)$
   $LSR = \lambda q \in HeapVariable. \lambda u \in Tid.\ (lh \cap LSR(q,u) \neq \emptyset)\ ?\ lh' \cup LSR(q,u)\ :\ LSR(q,u)$

Figure 8. Update rules for the extended lockset algorithm

The correctness of our algorithm is captured by the following theorem.

THEOREM 2 (SOUNDNESS). *Consider a program execution* $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ *such that* $\ell s_{n+1}(t) = \langle wrong, l\rangle$ *for some* $t \in Tid$ *and* $l \in LocalStore$. *Then the algorithm in Figure 6 explores a state* $(\ell s, h, LS)$ *such that* $\ell s(t) = \langle wrong, l\rangle$.

The proof of this theorem appears in the appendix of the full version of our paper [1].

6 Extending the lockset algorithm for concurrent reads

The lockset algorithm described in Section 4 does not distinguish between read and write accesses to a variable. To increase performance while still guaranteeing race-freedom, many programs rely on a locking discipline in which concurrent reads to a variable are allowed. In this section, we extend the lockset algorithm to allow for concurrent reads by treating reads and writes differently.

In the extended version of the algorithm, LS is divided into two separate maps LSR and LSW. The function LSW from $HeapVariable$ to $Powerset(Addr \cup Tid)$ is similar to the earlier LS and provides for each variable q the lockset $LSW(q)$ containing the set of locks that protect write accesses to q. The function LSR from $HeapVariable \times Tid$ to $Powerset(Addr \cup Tid)$ provides for each variable q and for each thread t the lockset $LSR(q,t)$ containing the set of locks that protect read accesses to q by t.

The update rules for the extended algorithm are given in Figure 8. Initially, we have $LSW(q) = Addr \cup Tid$ for all $q \in HeapVariable$, and $LSR(q,u) = Addr \cup Tid$ for all $q \in HeapVariable$ and for all $u \in Tid$. Given the maps LSW and LSR at state $(\ell s, h)$, we show how to compute the maps at state $(\ell s', h')$ by a case analysis on α. Let $q = (l(x), f))$ be a variable. If thread t performs a read access to q, Rule 2 only updates $LSR(q,t)$. But if thread t performs a write access to q, Rule 3 updates $LSW(q)$ and $LSR(q,u)$ for all $u \in Tid$. A race at a read access for q is reported in Rule 2 if $LH(t) \cap LSW(q) = \emptyset$ just before the access. A race at a write access for q is reported in Rule 3 if $LH(t) \cap LSR(q,u) = \emptyset$ for some $u \in Tid$.

7 Implementation and evaluation

We have implemented the algorithm described in Section 5 using the extended lockset algorithm described in Section 6. Our implementation is based on the Zing [30] model checking infrastructure. The most interesting aspect of the implementation is the management of locksets. Recall that the algorithm in Figure 6 augments the program state with a lockset for each heap variable and runs the lockset algorithm of Figure 4 as it explores the execution sequences of the program. The implementation is not straightforward because naively associating a lockset with each heap variable may increase the size of the state vector by a large factor. Moreover, acquire, fork, and join operations require the lockset of every variable to be updated. Again, a naive implementation would require a scan of the entire heap which can also be prohibitively expensive.

To solve these problems, we observed that although a typical program might have a large number of heap variables it uses only a small number of locks. Therefore the total number of distinct locksets in use would also be small and we expect that a large number of heap variables will have the same lockset associated with them. Therefore, we separated the lockset management into an abstract data type called the *lockset table*. All locksets for the program are stored in the lockset table, and the program refers to these locksets by integer indices. This strategy ensures that there is precisely one copy of each distinct lockset in the state and allows sharing of locksets among different heap variables. For each object created by the program, the implementation also create a shadow object of the same size. The *i*-th field of the shadow object contains the index of the lockset for the *i*-th field of the original object. Another advantage of this implementation is that for acquire, fork, and join operations, instead of iterating over the entire state vector, we only need to iterate over the locksets in the lockset table and update each distinct lockset according to the rules. Since we expect the lockset table to be much smaller than the state vector, this approach is much faster.

We have evaluated our implementation on a number of interesting examples that exhibit a variety of synchronization idioms. Existing static techniques, described in Section 8, would find it difficult to prove the absence of data-races on these examples. Our lockset algorithm can uniformly deal with all the synchronization idioms and thus enables our model checker to verify the absence of data-races. For lack of space, we only give brief descriptions of these examples below. We encourage the reader to examine the source code available at the web address: http://www.research.microsoft.com/˜qadeer/pldi06-examples.zip.

Indexer and **FileSystem**. These two examples were presented by Flanagan and Godefroid [12]. Both these examples have global variables that statically appear to be shared among the program threads but are dynamically thread-local. The dynamic partial-order reduction algorithm presented by them discovers the thread-locality and consequently schedules exactly one interleaving of the threads. Our algorithm works just as well and also schedules exactly one interleaving of the threads.

IndependentWork1 and **IndependentWork2**. These two examples were presented by Robby et al. [19] to illustrate the need for static and dynamic escape analysis for detecting independent actions in partial-order reduction. In IndependentWork1, the main thread creates two different threads each of which creates and initializes a list and then traverses it. In IndependentWork2, the main thread creates and initializes two lists and then creates two threads each of which traverses one of the lists. For both examples, once a thread starts accessing a list, the list becomes local to that thread. Our analysis discovers this property and consequently does not report any races. In addition, the model checking finishes after scheduling exactly one interleaving of the threads.

HaltException. This example was presented by Havelund and Presburger [16]. It contains a lock-protected buffer that is used by a producer thread and a consumer thread to share work items. Both during the initialization of the work item by the producer and its processing by the consumer, the work item is local to the respective thread. The ownership transfer happens when the item is queued into the buffer. Our analysis verified the absence of data-races and a variety of assertions.

BlinkTree. This example contains the implementation of a single level of a concurrent B-link tree [31]. There is a linked list of container nodes, each of which has references to a set of data nodes. Each data node contains a pair consisting of a key and a datum. The fields of the container node and the data nodes attached to it are protected by the lock of the container node. The data structure supports three operations—*Insert*, *Delete*, and *Lookup*, each of which is highly optimized to acquire as few locks as possible. To keep the tree balanced, the *Insert* and *Delete* operations may move data nodes from one container node to another. Thus, the lock protecting a data node may change dynamically. Our analysis verified the absence of data-races and a variety of assertions.

IOManager. This example is concerned with the lifecycle of an I/O Request Packet (IRP), a data structure that encapsulates a single request for I/O from an application to the kernel. An IRP passes through multiple ownership transfers from its creation to its completion. There could be as many as four threads potentially seeking access to a single IRP—a thread that creates the IRP, a thread that completes the IRP successfully, a thread that cancels the IRP, and a thread that performs post-processing on a successfully completed or canceled IRP. This example has the most sophisticated synchronization among our examples and involves two lock-protected queues and several volatile variables. Our tool proved the absence of race conditions and assertion violations.

8 Related Work

We present related work along two axis: static and dynamic race detection, and partial-order reduction in software model checking.

Race detection: Static approaches to race checking exploit compile-time analysis on the program source, and report potential sources of races. Warlock [22] and RacerX [11] use this approach. Another approach is to augment the programming language's type system to express common synchronization mechanisms so that any well-typed program is guaranteed to be race-free. This approach requires a considerable amount of annotation into the source code by the programmer and also restricts the kinds of synchronization idioms that can be employed. The formal type systems used by Flanagan et al. [5] and Boyapati et al. [3, 7] capture many common synchronization patterns including mutually exclusive locks, thread-local objects, objects with internal synchronization, objects with fields synchronized by external locks, etc. Inspired by [3], Grossman et al. [7] extend Cyclone's polymorphic type system with threads and locks. Then their notion of type safety implies absence of races. The main shortcoming of the static methods is the fact that they are rather restrictive, i.e., they report many false positives and require escape mechanisms to bypass benign race conditions.

Dynamic approaches aim to detect races at runtime by looking at the history of memory accesses and synchronization operations recorded along an execution of the program. Dynamic methods are more accurate for analyzing individual executions and do not suffer much from false positives as much as static methods. However, they are not exhaustive and thus cannot reason about race-freedom of a whole program. There are two main classes of dynamic techniques: lockset analysis and happens-before analysis. Lockset-based algorithms verify that the program execution conforms to a locking discipline – a programming methodology that ensures the absence of data races. Eraser [28] is a tool for detecting race conditions dynamically by enforcing the locking discipline that every shared variable is protected by a unique lock. It handles object initialization patterns using a state-based approach but can not handle dynamically changing locksets since it only allows a lockset to get smaller. There is much work [14, 6, 32] that refines the Eraser algorithm by improving the state machine it uses and the transitions to reduce the number of false positives. The approaches that check a happens-before relation [20, 21, 10] are based on Lamport's happens-before relation [17], which outputs a partial ordering on program statements. A data race occurs when there is no temporal ordering provided by the happens-before relation between two conflicting memory accesses. This technique is more general than lockset-based methods, and it can be applied to programs with fork/join or signal/wait synchronization in addition to locks. However, it is less efficient to implement than a lockset algorithm and imprecise computation of the relation might lead to false negatives. There are techniques [15, 9, 32] that combine lockset and happens-before analysis that get advantages of both approaches. Our technique, for the first time, computes a precise happens-before relation using an implementation that makes use of only locksets.

Partial-order reduction: Researchers have used synchronization mechanisms to do partial-order reduction [23] for model checking concurrent systems. Verisoft uses stateless search and partial order reduction on actual code written in a full-blown implementation language. Verisoft [24] introduces a stateless exploration technique that exploits persistent and sleep sets [23]. Stoller et al. [27, 26] consider various kinds of exclusive access predicates for shared variables that specify mutually-exclusive synchronization disciplines. These predicates are used to perform partial-order reduction on the state space, in the meanwhile inferring the assumptions on the predicates. The work in [26] is interesting in that their approach only requires checking if the reduced software obeys the synchronization discipline. Unless the exclusive access predicates are expressive enough, these techniques do not work well when the synchronization discipline, e.g. the locksets protecting a variable, changes over time along the execution. The Bogor model checker [19] detects thread-local objects at each state visited by performing a heap traversal and dynamic escape analysis, and exploits patterns of lock acquisitions and releases in order to find ample sets [8]. Transaction based dynamic partial-order reduction method by Flanagan and Qadeer [4] is based on the theory of reduction. One application of the lockset algorithm can be improving their technique by detecting race-free variable accesses to accurately infer transaction boundaries.

Flanagan and Godefroid [12] presents a stateless model checking algorithm that dynamically tracks a dependency relation between actions seen and computes an approximation of a persistent set at each state visited during the exploration. Since it is stateless, their algorithm runs only on acyclic state spaces. Their approach requires efficient implementation of vector clocks for computing a happens-before relation that captures the dependency relation. Furthermore the dependency relation is tracked along the execution paths. Therefore, their technique is hard to implement in a stateful setting, as pruning a state reached through a different path may cause losing part of the dependency relation.

9 Conclusions

In this paper, we present a new algorithm for detecting data-races in an execution of a concurrent program. Our algorithm is sound and precise, that is, it reports a race in an execution iff there are two accesses to a shared variable along the execution that are not ordered by the happens-before relation. Our algorithm is based solely on the concept of locksets and is able to capture all mutual-exclusion synchronization idioms uniformly with one mechanism. Our lockset algorithm can be used, both in the static or the dynamic context, to develop analyses for concurrent programs, particularly those for detecting data-races, atomicity violations, and failures of safety specifications.

We presented a model checking algorithm for concurrent software that uses our lockset algorithm both to check for races exhaustively and to perform partial-order reduction when races are absent. We have implemented our algorithm and evaluated it by verifying the absence of data-races and assertion failures on a number of examples exhibiting a variety of synchronization idioms. In future work, we would like to tackle more examples, especially from operating systems, which are notorious for having complicated synchronization idioms. We would also like to evaluate the efficacy of our lockset algorithm in the context of dynamic data-race detection.

10 References

[1] Tayfun Elmas, Shaz Qadeer, Serdar Tasiran. Precise Race Detection and Efficient Model Checking Using Locksets. The full version of the paper with appendices. http://www.research.microsoft.com/~qadeer/pldi06-submission-fullversion.ps

[2] C. Boyapati, R. Lee, M. Rinard. A type system for preventing data races and deadlocks in Java programs. In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, pages 211–230, 2002.

[3] C. Boyapati, M. Rinard. A parameterized type system for race-free Java programs. In Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages and Applications, Tampa Bay, FL, Oct. 2001.

[4] C. Flanagan and S. Qadeer. Transactions for Software Model Checking. In Proceedings of the Workshop on Software Model Checking, pages 338–349, June 2003.

[5] C. Flanagan and S. Freund. Type-based race detection for java. In Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation, Vancouver, Canada, June 2000.

[6] C. Praun and T. Gross. Object race detection. In Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA), pages 70-82, 2001.

[7] D. Grossman. Type-safe multithreading in Cyclone. In Workshop on Types in Language Design and Implementation (TLDI), January 2003.

[8] E. Clarke, O. Grumberg, and D. Peled. Model Checking. MIT Press, 2000.

[9] E. Pozniansky and A. Schuster. Efficient on-the-fly race detection in multithreaded C++ programs. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2003.

[10] E. Schonberg. On-the-fly detection of access anomalies. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 285-297, 1989.

[11] Engler, D. and Ashcraft, K. RacerX: Effective, static detection of race conditions and deadlocks. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles. SOSP'03. ACM Press, New York, NY, 237-252.

[12] Flanagan, C. and Godefroid, P. Dynamic partial-order reduction for model checking software. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'05. ACM Press, New York, NY, 110-121.

[13] Guillaume Brat, Klaus Havelund, Seung-Joon Park, and Willem Visser. Model Checking Programs. In IEEE International Conference on Automated Software Engineering (ASE), September 2000.

[14] H. Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM), May 2004.

[15] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification, pages 331342, London, UK, 2000. Springer-Verlag.

[16] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. In the International Journal on Software Tools for Technology Transfer (STTT), December 1998.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558565, July 1978.

[18] Manson, J., Pugh, W., and Adve, S. V. The Java memory model. In Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL'05. ACM Press, New York, NY, 378-391.

[19] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. Formal Methods in System Designs, 2004.

[20] M. Christiaens and K. De Bosschere. TRaDe, a topological approach to on-the-fly race detection in Java programs. In Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM), Apr. 2001.

[21] M. Ronsse and K. De Bosschere. RecPlay: A fully integrated practical record/replay system. ACM Transactions on Computer Systems, 17(2):133152, May 1999.

[22] Nicholas Sterling. Warlock: A static data race analysis tool. In USENIX Winter Technical Conference, January 1993.

[23] P. Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. volume 1032 of Lecture Notes in Computer Science. Springer-Verlag, January 1996.

[24] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174–186, Paris, January 1997.

[25] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In Proc. 5th Conference on Computer Aided Verification, volume 697 of Lecture Notes in Computer Science, pages 438-449, Elounda, June 1993. Springer-Verlag.

[26] Scott D. Stoller, Ernie Cohen. Optimistic Synchronization-Based State-Space Reduction. In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), volume 2619 of Lecture Notes in Computer Science, pages 489–504. Springer-Verlag, April 2003.

[27] Scott D. Stoller. Model-Checking Multi-Threaded Distributed Java Programs. International Journal on Software Tools for Technology Transfer, 4(1):71-91. Springer-Verlag. October 2002.

[28] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[29] Robby, Matthew B. Dwyer, John Hatcliff Bogor: An Extensible and Highly-Modular Model Checking Framework, March 2003. In the Proceedings of the Fourth Joint Meeting of the European Software Engineering Conference and ACM SIG-

SOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2003).

[30] T. Andrews, S. Qadeer, J. Rehof, S.K. Rajamani, and Y. Xie. Zing: A model checker for concurrent software. Proceedings of the 16th International Conference on Computer-Aided Verification, 2004.

[31] Y. Sagiv, Concurrent operations on B-trees with overtaking, Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems, p.28-37, March 25-27, 1985, Portland, Oregon, United States.

[32] Yu, Y., Rodeheffer, T., and Chen, W. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In Proceedings of the Twentieth ACM Symposium on Operating Systems Principles. SOSP'05. ACM Press, New York, NY, 221-234.

A Correctness proof of the lockset algorithm

LEMMA 1 *Let* $\sigma = (\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$ *be an execution of the program. For all* $1 \le j \le n+1$, *let* LH_j *and* LS_j *be the auxiliary variables associated with state* $(\ell s_j, h_j)$. *Let* q *be a variable that was last accessed by action* α_i *in* σ.

1. *Let* $m \in Addr$ *be such that* $m \notin LH_{n+1}(t)$ *for all* $t \in Tid$. *Then* $m \in LS_{n+1}(q)$ *iff there exists* j *such that* $1 \le j \le n$, $i \xrightarrow{hb} j$, $\alpha_j = rel(x)$, *and* $\ell s_j(t_j)(x) = m$.
2. *Let* $m \in Addr$ *be such that* $m \in LH_{n+1}(t)$ *for some* t. *Then* $m \in LS_{n+1}(q)$ *iff there exists* j *such that* $1 \le j \le n$, $i \xrightarrow{hb} j$ *and* $t_j = t$.
3. *Let* $t \in Tid$. *Then* $t \in LS_{n+1}(q)$ *iff there exists* j *such that* $1 \le j \le n$, $i \xrightarrow{hb} j$ *and either* $t_j = t$ *or* $\alpha_j = (x = fork)$ *and* $\ell s_{j+1}(t_j)(x) = t$.

PROOF. We prove the lemma by induction over the length of the execution. When the execution length is 0, the lemma holds trivially because there is no variable q that is accessed by an action in the execution. Suppose the lemma holds for $\sigma = (\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell s_n, h_n)$ and $(\ell s_n, h_n) \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$. Fix a variable q that was last accessed by α_i for some $1 \le i \le n$. We perform a case analysis on α_n.

1. $\alpha_n = (x = new)$ or $\alpha_n = (x = op(y_1, \ldots, y_m))$. Neither LH nor LS changes and no heap variable is accessed. Therefore the proof follows by a straightforward application of the inductive hypothesis.

2. $\alpha_n = (y = x.f)$ or $\alpha_n = (x.f = y)$. Let $q' = (\ell s_n(t_n)(x), f)$ be the variable accessed by α_n. We prove the two cases, $i = n$ and $i \ne n$, separately.

   First, suppose $i = n$. Then $q = q'$ and $LS_{n+1}(q) = LS_{n+1}(q') = LH_{n+1}(t_n)$.

   (a) Suppose $m \notin LH_{n+1}(t)$ for all $t \in Tid$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Since $LS_{n+1}(q) = LH_{n+1}(t_n)$, we get a contradiction. For the "only if" direction, suppose there exists j such that $1 \le j \le n$, $i \xrightarrow{hb} j$, $\alpha_j = rel(x)$, and $\ell s_j(t_j)(x) = m$. Since $i = n$ and $i \xrightarrow{hb} j$, we have $j = n$ and we arrive at the contradiction.

   (b) Suppose $m \in LH_{n+1}(t)$ for some $t \in Tid$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Since $LS_{n+1}(q) = LH_{n+1}(t_n)$ and $i = n$, we get $i \xrightarrow{hb} n$ and $t = t_n$. For the "only if" direction, suppose there exists j such that $1 \le j \le n$, $i \xrightarrow{hb} j$, and $t_j = t$. Since $i = n$, we get $j = n$ and $t = t_n$. Therefore $m \in LH_{n+1}(t) = LH_{n+1}(t_n) = LS_{n+1}(q)$.

   (c) Suppose $t \in Tid$. For the "if" direction, suppose $t \in LS_{n+1}(q)$. Then $t = t_n$ and $i = n \xrightarrow{hb} n$. For the "only if" direction, suppose there exists j such that $1 \le j \le n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (x = fork)$ and $\ell s_{j+1}(t_j)(x) = t$. Since $i = n$, we get $j = n$. Since $t_n \in LS_{n+1}(q)$, we are done.

   Second, suppose $i \ne n$. Then $q \ne q'$ and $LS_n(q) = LS_{n+1}(q)$.

   (a) Suppose $m \notin LH_{n+1}(t)$ for all $t \in Tid$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. We have $m \in LS_n(q)$ and we are done by the inductive hypothesis. For the "only if" direction, suppose there exists j such that $1 \le j \le n$, $i \xrightarrow{hb} j$, $\alpha_j = rel(x)$, and $\ell s_j(t_j)(x) = m$. Since α_n is not a release action, it must be that $j < n$. Then the inductive hypothesis gives us that $m \in LS_n(q)$ and we are done.

   (b) Suppose $m \in LH_{n+1}(t)$ for some $t \in Tid$. The proof is exactly the same as the case above.

   (c) Suppose $t \in Tid$. For the "if" direction, suppose $t \in LS_{n+1}(q)$. We have $t \in LS_n(q)$ and we are done by the inductive hypothesis. For the "only if" direction, let j be the least integer such that $1 \le j \le n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (y = fork)$ and $\ell s_{j+1}(t_j)(y) = t$. We argue that $j \ne n$. Since α_n is not a fork action, we only have to argue for the case $t_j = t$. Suppose α_j is the first action of thread t. Then there exists l such that $i \xrightarrow{hb} l < j$ and α_l forked thread t, i.e., $\alpha_l = (y = fork)$ and $\ell s_{l+1}(t_l)(y) = t$, which contradicts the minimality of j. Suppose α_j is not the first action of thread t. Since α_j is not a synchronization action, there exists l such that $i \xrightarrow{hb} l < j$ and $\alpha_l = \alpha_j$, which again contradicts the minimality of j. Thus, we conclude that $j < n$ and he inductive hypothesis gives us that $t \in LS_n(q) \subseteq LS_{n+1}(q)$.

3. $\alpha_n = acq(x)$. Let $p = \ell s_n(t_n)(x)$ be the lock acquired by α_n. We have that $LS_{n+1}(q) = LS_n(q)$ if $LH_{n+1}(t_n) \cap LS_n(q) = \emptyset$ and $LS_{n+1}(q) = LS_n(q) \cup LH_{n+1}(t_n)$ otherwise.

   (a) Suppose $m \notin LH_{n+1}(t)$ for all $t \in Tid$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Since $m \notin LH_{n+1}(t_n)$, we get that $m \in LS_n(q)$ and we are done by the inductive hypothesis. For the "only if" direction, suppose there exists j such that $1 \le j \le n$, $i \xrightarrow{hb} j$, $\alpha_j = rel(x)$, and $\ell s_j(t_j)(x) = m$. Since α_n is not a release action, it must be that $j < n$. Then the inductive hypothesis gives us that $m \in LS_n(q) \subseteq LS_{n+1}(q)$ and we are done.

   (b) Suppose $m \in LH_{n+1}(t)$ for some $t \in Tid$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. If $m \in LS_n(q)$, we are done by the inductive hypothesis. Otherwise, $m \in LH_{n+1}(t_n)$, $t = t_n$, and either $t_n \in LS_n(q)$ or there exists some $m' \in LH_{n+1}(t_n) \cap LS_n(q)$. If $t_n \in LS_n(q)$, then by the inductive hypothesis there exists j such that $1 \leq j \leq n-1$, $i \xrightarrow{hb} j$ and either $t_j = t_n$ or $\alpha_j = (x = \mathit{fork})$ and $\mathit{ls}_{j+1}(t_j)(x) = t_n$. Thus, we get $i \xrightarrow{hb} n$ and we are done. If there exists some $m' \in LH_{n+1}(t_n) \cap LS_n(q)$, there are two cases: either $m' \in LH_n(t_n)$ or $m' = \mathit{ls}_n(t_n)(x)$. If $m' \in LH_n(t_n)$, by the inductive hypothesis there exists j such that $1 \leq j \leq n-1$, $i \xrightarrow{hb} j$ and $t_j = t_n$. Since $t_n = t$, we are done. If $m' = \mathit{ls}_n(t_n)(x)$, then $m' \notin LH_n(u)$ for all $u \in \mathit{Tid}$. By the inductive hypothesis, there exists j such that $1 \leq j \leq n-1$, $i \xrightarrow{hb} j$, $\alpha_j = \mathit{rel}(y)$, and $\mathit{ls}_j(t_j)(y) = m'$. Since $i \xrightarrow{hb} j \xrightarrow{hb} n$ and $t_n = t$, we are done. For the "only if" direction, suppose j is the least integer such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$, and $t_j = t$. There are two cases: either $t \neq t_n$ or $t = t_n$. If $t \neq t_n$, we have $t_j \neq t_n$ and therefore $j < n$. By the inductive hypothesis, we have $m \in LS_n(q) \subseteq LS_{n+1}(q)$. If $t = t_n$, there are two cases: $j < n$ or $j = n$. If $j < n$, then by the inductive hypothesis we have $m \in LS_n(q) \subseteq LS_{n+1}(q)$. If $j = n$, then there is a $k < n$ such that $i \xrightarrow{hb} k$ and one of two cases hold: Either $\alpha_k = (y = \mathit{fork})$ and $\mathit{ls}_{k+1}(t_k)(y) = t_n$ or $\alpha_k = \mathit{rel}(y)$, and $\mathit{ls}_k(t_k)(y) = \mathit{ls}_n(t_n)(x)$. In the first case, the inductive hypothesis gives us $t_n \in LS_n(q)$. In the second case, the inductive hypothesis gives us $\mathit{ls}_n(t_n)(x) \in LS_n(q)$. Thus, in both cases we have $LS_n(q) \cap LH_{n+1}(t_n) \neq \emptyset$ and therefore $m \in LH_{n+1}(t_n) \subseteq LS_{n+1}(q)$.

(c) The proof for this case is very similar to the second case above.

4. $\alpha_n = \mathit{rel}(x)$. It must be the case that $\mathit{ls}_n(t_n)(x) \in LH_n(t_n)$, otherwise the action goes wrong.

(a) Suppose $m \notin LH_{n+1}(t)$ for all $t \in \mathit{Tid}$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Then $m \in LS_n(q)$. There are two cases: either $m \notin LH_n(t)$ for all $t \in \mathit{Tid}$ or $m = \mathit{ls}_n(t_n)(x)$. If $m \notin LH_n(t)$ for all $t \in \mathit{Tid}$, the inductive hypothesis gives us that there exists j such that $1 \leq j \leq n-1$, $i \xrightarrow{hb} j$, $\alpha_j = \mathit{rel}(y)$, and $\mathit{ls}_j(t_j)(y) = m$. If $m = \mathit{ls}_n(t_n)(x)$, then $m \in LH_n(t_n)$ and the inductive hypothesis gives us that there exists j such that $1 \leq j \leq n-1$, $i \xrightarrow{hb} j$ and $t_j = t_n$. Therefore, we get $i \xrightarrow{hb} n$, $\alpha_n = \mathit{rel}(x)$, and $m = \mathit{ls}_n(t_n)(x)$. For the "only if" direction, suppose j is such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$, $\alpha_j = \mathit{rel}(x)$, and $\mathit{ls}_j(t_j)(x) = m$. If $j < n$, then the inductive hypothesis gives us $m \in LS_n(q) = LS_{n+1}(q)$. If $j = n$, then $m = \mathit{ls}_n(t_n)(x)$ and therefore $m \in LH_n(t_n)$. Since t_n must have acquired m before releasing it, α_n is not the first action of t_n and there is $k < n$ such that $t_k = t_n$ and $i \xrightarrow{hb} k$. By the inductive hypothesis, we have $m \in LS_n(q) = LS_{n+1}(q)$ and we are done.

(b) Suppose $m \in LH_{n+1}(t)$ for some $t \in \mathit{Tid}$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Then $m \in LH_n(t)$ and $m \in LS_n(q)$, and we are done by the inductive hypothesis. For the "only if" direction, let j be the least integer such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$, and $t_j = t$. If $j = n$, $t = t_n$. But α_n is not the first action of t_n. So there must be $k < n$ such that $i \xrightarrow{hb} k$ and $t_k = t_n$ and we get a contradiction. Therefore $j < n$ and by the inductive hypothesis we get $m \in LS_n(t) = LS_{n+1}(t)$.

(c) Suppose $t \in \mathit{Tid}$. For the "if" direction, suppose $t \in LS_{n+1}(q)$. Then $t \in LS_n(q)$ and we are done by the inductive hypothesis. For the "only if" direction, let j be the least integer such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (x = \mathit{fork})$ and $\mathit{ls}_{j+1}(t_j)(x) = t$. If $j = n$, then $t = t_n$. But α_n is not the first action of t_n. So there must be $k < n$ such that $i \xrightarrow{hb} k$ and $t_k = t_n$ and we get a contradiction. Therefore $j < n$ and we get $t \in LS_n(q) = LS_{n+1}(q)$ from the inductive hypothesis.

5. $\alpha_n = (x = \mathit{fork})$. Let $u = \mathit{ls}_{n+1}(t_n)(x)$ be the thread forked by α_n. We have that $LS_{n+1}(q) = LS_n(q)$ if $LH_n(t_n) \cap LS_n(q) = \emptyset$ and $LS_{n+1}(q) = LS_n(q) \cup \{u\}$ otherwise.

(a) Suppose $m \notin LH_{n+1}(t)$ for all $t \in \mathit{Tid}$. Then $m \notin LH_n(t)$ for all $t \in \mathit{Tid}$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Then $m \in LS_n(q)$ and we are done by a straightforward application of the inductive hypothesis. For the "only if" direction, suppose j is such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$, $\alpha_j = \mathit{rel}(y)$, and $\mathit{ls}_j(t_j)(y) = m$. Since $\alpha_n = (x = \mathit{fork})$, we get $j < n$. By the inductive hypothesis, we get $m \in LS_n(q) \subseteq LS_{n+1}(q)$.

(b) Suppose $m \in LH_{n+1}(t)$ for some $t \in \mathit{Tid}$. Then $m \in LH_n(t)$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Then $m \in LS_n(q)$ and we are done by a straightforward application of the inductive hypothesis. For the "only if" direction, let j be the least integer such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$, and $t_j = t$. If $j = n$, then α_n must be the first action of t_n and therefore $LH_n(t_n) = \{t_n\}$. Since $t_n = t$, we get a contradiction. Therefore $j < n$ and by the inductive hypothesis, we get $m \in LS_n(q) \subseteq LS_{n+1}(q)$.

(c) Suppose $t \in \mathit{Tid}$. For the "if" direction, suppose $t \in LS_{n+1}(q)$. Then, either $t \in LS_n(q)$ or $t = u$ and $LS_n(q) \cap LH_n(t_n) \neq \emptyset$. In the first case, we are done by the inductive hypothesis. In the second case, by the inductive hypothesis, there exists j such that $1 \leq j < n$, $i \xrightarrow{hb} j$, and $t_j = t_n$. Therefore $i \xrightarrow{hb} n$ and we are done. For the "only if" direction, let j be the least integer such that $1 \leq j \leq n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (y = \mathit{fork})$ and $\mathit{ls}_{j+1}(t_j)(y) = t$. If $j < n$, then the inductive hypothesis gives us $t \in LS_n(q) \subseteq LS_{n+1}(q)$. If $j = n$, then α_n must be the first action of t_n and either $t = t_n$ or $t = u$. Therefore, there is k such that $1 \leq k < n$, $i \xrightarrow{hb} k$, $\alpha_k = (z = \mathit{fork})$, and $\mathit{ls}_{k+1}(t_k)(z) = t_n$. By the inductive hypothesis, we get that $t_n \in LS_n(q) \subseteq LS_{n+1}(q)$. Since $t_n \in LH_n(t_n)$, we have $LS_n(q) \cap LH_n(t_n) \neq \emptyset$. Therefore $u \in LS_{n+1}(q)$. Since either $t = t_n$ or $t = u$, we are done.

6. $\alpha_n = \mathit{join}(x)$. Let $u = \mathit{ls}_n(t_n)(x)$ be the thread joined by α_n. We have that $LS_{n+1}(q) = LS_n(q)$ if $LH_n(u) \cap LS_n(q) = \emptyset$ and $LS_{n+1}(q) = LS_n(q) \cup LH_n(t_n)$ otherwise.

(a) Suppose $m \notin LH_{n+1}(t)$ for all $t \in \mathit{Tid}$. Then $m \notin LH_n(t)$ for all $t \in \mathit{Tid}$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Then $m \in LS_n(q)$ and we are done by a straightforward application of the inductive hypothesis. For the "only if" direction, suppose j is such that

$1 \le j \le n$, $i \xrightarrow{hb} j$, $\alpha_j = rel(y)$, and $\ell s_j(t_j)(y) = m$. Since $\alpha_n = join(x)$, we get $j < n$. By the inductive hypothesis, we get $m \in LS_n(q) \subseteq LS_{n+1}(q)$.

(b) Suppose $m \in LH_{n+1}(t)$ for some $t \in Tid$. Then $m \in LH_n(t)$. For the "if" direction, suppose $m \in LS_{n+1}(q)$. Then, either $m \in LS_n(q)$ or $m \in LH_n(t_n)$, $t_n = t$, and $LH_n(u) \cap LS_n(q) \neq \emptyset$. In the first case, we are done by a straightforward application of the inductive hypothesis. In the second case, we know by the inductive hypothesis that there is k such that $1 \le k < n$, $i \xrightarrow{hb} k$, and $t_k = u$. Therefore $i \xrightarrow{hb} n$ and we are done. For the "only if" direction, let j be the least integer such that $1 \le j \le n$, $i \xrightarrow{hb} j$, and $t_j = t$. There are two cases, $j = n$ and $j < n$. If $j = n$, then either α_n is the first action of t_n or there is a k such that $1 \le k < n$, $i \xrightarrow{hb} k$, and $t_k = u$. If α_n is the first action of t_n, we get $LH_n(t_n) = \{t_n\}$. Since $t_n = t$ and $m \in LH_n(t)$ we get a contradiction. If there is a k such that $1 \le k < n$, $i \xrightarrow{hb} k$, and $t_k = u$, then by the inductive hypothesis $u \in LS_n(q)$. Since $u \in LH_n(u)$, we get $LH_n(u) \cap LS_n(q) \neq \emptyset$. Therefore $m \in LH_n(t) \subseteq LS_{n+1}(q)$ and we are done. If $j < n$ then the inductive hypothesis gives us $m \in LS_n(q) \subseteq LS_{n+1}(q)$.

(c) Suppose $t \in Tid$. We prove the two cases, $t = t_n$ and $t \neq t_n$, separately. First, suppose $t = t_n$. For the "if" direction, suppose $t \in LS_{n+1}(q)$. Then, either $t \in LS_n(q)$ or $LS_n(q) \cap LH_n(u) \neq \emptyset$. If $t \in LS_n(q)$, then we are done by the inductive hypothesis. If $LS_n(q) \cap LH_n(u) \neq \emptyset$, then by the inductive hypothesis there exists j such that $1 \le j < n$, $i \xrightarrow{hb} j$, and $t_j = u$. Therefore $i \xrightarrow{hb} n$ and we are done. For the "only if" direction, let j be the least integer such that $1 \le j \le n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (y = fork)$ and $\ell s_{j+1}(t_j)(y) = t$. If $j < n$, then the inductive hypothesis gives us that $t \in LS_n(q) \subseteq LS_{n+1}(q)$ and we are done. If $j = n$, then either there exists k such that $1 \le k < n$, $i \xrightarrow{hb} k$, and $t_k = u$ or α_n is the first action of t_n and there exists k such that $1 \le k < n$, $i \xrightarrow{hb} k$, $\alpha_k = (z = fork)$ and $\ell s_{k+1}(t_k)(z) = t_n$. In the first case, we have $u \in LS_n(q)$ by the inductive hypothesis. Since $u \in LH_n(u)$, we have $LS_n(q) \cap LH_n(u) \neq \emptyset$. Therefore, we get $t_n \in LH_n(t_n) \subseteq LS_{n+1}(q)$ and we are done. In the second case, we have $t_n \in LS_n(q)$ by the inductive hypothesis and we are done since $LS_n(q) \subseteq LS_{n+1}(q)$.

Second, suppose $t \neq t_n$. For the "if" direction, suppose $t \in LS_{n+1}(q)$. Then $t \in LS_n(q)$ and we are done by the inductive hypothesis. For the "only if" direction, let j be such that $1 \le j \le n$, $i \xrightarrow{hb} j$ and either $t_j = t$ or $\alpha_j = (y = fork)$ and $\ell s_{j+1}(t_j)(y) = t$. Then, it must be that $j < n$. By the inductive hypothesis, we have $t \in LS_n(q) \subseteq LS_{n+1}(q)$ and we are done.

□

THEOREM **1** **(Correctness).** *Let $\sigma = (\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$ be an execution of the program. Let q be a variable and $i \in [1, n-1]$ be such that α_i and α_n access q but α_j does not access q for all $j \in [i+1, n-1]$. Then $LS_n(q) \cap LH_n(t_n) \neq \emptyset$ iff $i \xrightarrow{hb} n$.*

PROOF. The proof follows easily from a simple application of Lemma 1. □

B Soundness proof of the stateful model checking algorithm

For the proof, we will find it convenient to add LS to the state which now becomes a triple $(\ell s, h, LS)$. A transition $(\ell s, h, LS) \xrightarrow{\alpha}_t (\ell s', h', LS')$ is a *mover* if one of the following conditions is satisfied:

1. $\alpha = rel(x)$.
2. $\alpha = (x = fork)$.
3. $\alpha = (y = x.f)$ or $\alpha = (x.f = y)$ and for any execution $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ where $(\ell s_1, h_1, LS_1) = (\ell s', h', LS')$ and $t_j \neq t$ for all $j \in [1, n]$, we have that α_j does not access the variable $(\ell s(t)(x), f)$ for all $j \in [1, n]$.

LEMMA 2. *Let $(\ell s_1, h_1, LS_1) \xrightarrow{\alpha}_t (\ell s'_1, h'_1, LS'_1)$ be a mover. Let $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ be an execution where $t_j \neq t$ for all $j \in [1, n]$. Then there is a mover $(\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha}_t (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1})$ and an execution $\sigma' = (\ell s'_1, h'_1, LS'_1) \xrightarrow{\alpha_1}_{t_1} (\ell s'_2, h'_2, LS'_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1})$ such that the following are true for all $j \in [1, n+1]$:*

1. *If $\alpha = rel(x)$, then $\ell s_j(u) = \ell s'_j(u)$ for all $u \neq t$ and $LS_j(q) = LS'_j(q)$ for all $q \in HeapVariable$.*
2. *If $\alpha = (x = fork)$, then $\ell s_j(u) = \ell s'_j(u)$ for all $u \notin \{t, \ell s'_1(t)(x)\}$ and $LS_j(q) = LS'_j(q)$ for all $q \in HeapVariable$.*
3. *If $\alpha = (y = x.f)$ or $\alpha = (x.f = y)$, then $\ell s_j(u) = \ell s'_j(u)$ for all $u \neq t$ and $LS_j(q) = LS'_j(q)$ for all $q \neq (\ell s_1(t)(x), f)$.*

PROOF. The proof is by induction over the number n. The base case $n = 0$ is trivial.

We now prove the inductive case. Suppose that the lemma is true for an execution $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell s_n, h_n, LS_n)$ where $t_j \neq t$ form $j \in [1, n-1]$. The inductive case involves a 3-way case analysis over α_n. Our assumptions due to the inductive hypothesis are the following:

- There is a mover $(\ell s_1, h_1, LS_1) \xrightarrow{\alpha}_t (\ell s'_1, h'_1, LS'_1)$
- There is an execution given by $\sigma' = (\ell s'_1, h'_1, LS'_1) \xrightarrow{\alpha_1}_{t_1} (\ell s'_2, h'_2, LS'_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell s'_n, h'_n, LS'_n)$ for which the three facts above are true for all $j \in [1, n]$.
- There is a mover $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha}_t (\ell s'_n, h'_n, LS'_n)$

1. $\alpha = rel(x)$ where $\ell s(t)(x) = a$ for some object a. Since lock release during the transition $(\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha}_t (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1})$ only changes program counter of t, and $a.owner$, $\ell s_{n+1}(u) = \ell s'_{n+1}(u)$ for all $u \neq t$. Since lock release

does not update any $LS(q)$ for any variable q, $LS_j(q) = LS'_j(q)$ for all $q \in \mathit{HeapVariable}$.

2. $\alpha_n = (x = \mathit{fork})$ where $\ell s(t)(x) = t'$ for a new thread t'. Since thread creation during $(\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha}_t (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1})$ only changes program counter of t, and $\ell s_{n+1}(t')$, $\ell s_j(u) = \ell s'_j(u)$ for all $u \notin \{t, \ell s'_1(t)(x)\}$. Because $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha}_t (\ell s'_n, h'_n, LS'_n)$ does not change $LS(q)$ for any $q \in \mathit{HeapVariable}$, $(\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha}_t (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1})$ does not change $LS(q)$, either. Then $LS_j(q) = LS'_j(q)$ for all $q \in \mathit{HeapVariable}$.

3. $\alpha_n = (y = x.f)$ or $\alpha = (x.f = y)$. An access to variable $(\ell s_1(t)(x), f))$ during $(\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha}_t$ $(\ell s'_{n+1}, h'_{n+1}, LS'_{n+1})$ only changes program counter of t, and either $\ell s_{n+1}(t)(y)$ or $(\ell s_{n+1}(t)(x), f)$, $\ell s_j(u) = \ell s'_j(u)$ for all $u \neq t$ and $LS_j(q) = LS'_j(q)$ for all variables $q \neq (\ell s_1(t)(x), f)$.

□

The following lemmas prove the correctness of the stateful version of the algorithm.

Let $\prec$ be the total order between the states explored by our model checking algorithm. $(\ell s, h, LS) \prec (\ell s', h', LS')$ iff a node z such that $z.\mathit{state} = (\ell s, h, LS)$ is popped from the stack before a node z' such that $z'.\mathit{state} = (\ell s', h', LS')$ is popped from the stack. In this order each state $(\ell s, h, LS)$ corresponds to a unique node z such that $z.\mathit{state} = (\ell s, h, LS)$ that is actually pushed on the stack. Note that the algorithm creates some temporary nodes that are never pushed on the stack. For example, a node that is created at line 37 is thrown away if control reaches line 48. Those temporary nodes do not participate in the induction.

We say a transition $(\ell s, h, LS) \xrightarrow{\alpha}_t (\ell s', h', LS')$ or just the state $(\ell s', h', LS')$ *hits the stack* if there is a node $z \in \mathit{stack}$ such that $z.\mathit{state} = (\ell s', h', LS')$.

LEMMA 3. *Let the stack in the algorithm contain the sequence $z_1, \ldots, z_n$ of nodes corresponding to the program execution $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell s_n, h_n, LS_n)$ at a time when z_n is about to be popped. Let $\sigma' = (\ell s_n, h_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha_{n+1}}_{t_{n+1}} \ldots \xrightarrow{\alpha_k}_{t_k} (\ell s_{k+1}, h_{k+1}, LS_{k+1})$ be an extension of σ. If q is a variable such that α_i accesses q for some $i \in [1, n-1]$, α_j does not access q for all $j \in [i+1, k-1]$, and α_k accesses q, then either $i \xrightarrow{hb} k$ or $z_i.tid = 0$.*

PROOF. We prove the lemma by induction over the total order $\prec$. Let z be the node being popped from the stack.

Base case: z_n is the first node to be popped from the stack. All the transitions from z_n are explored such that $z_n.\mathit{done} = \mathit{enabled}(z_n.\mathit{state})$. If $\mathit{enabled}(z_n.\mathit{state}) = \emptyset$, then we are done immediately. Otherwise, there is at least one transition that hits both on the stack and in the hashtable (otherwise, z_n is not the first node to be popped). Therefore, $z.\mathit{succOnStack}$ is true. The code between lines 25–30 in *Search* takes care of the proof.

Inductive case: There are two cases:

1. All the transitions from z_n are explored such that $z_n.\mathit{done} = \mathit{enabled}(z_n.\mathit{state})$. There are two sub-cases, some transition hits on the stack or no transition hits on the stack.

   (a) A transition $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ hits on the stack; there is another node z' in the stack such that $z'.\mathit{state} = (\ell s_{n+1}, h_{n+1}, LS_{n+1})$. Then, we have $z.\mathit{succOnStack} = \mathit{true}$ and the code between lines 25–30 in *Search* takes care of the proof.

   (b) No transitions from $z.\mathit{state}$ hits on the stack: Then for each transition $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$, there must be a node z' that must have been popped before z_n such that $z'.\mathit{state} = (\ell s_{n+1}, h_{n+1}, LS_{n+1})$. If $k > n$, we then invoke the inductive hypothesis because $z_n.\mathit{state} \prec z_{n+1}.\mathit{state}$. Otherwise, $k = n$ and the lockset algorithm whose correctness is given in Theorem 1 guarantees that either $i \xrightarrow{hb} n$ or $z_i = 0$ for all $i \in [1, n-1]$.

2. Not all the transitions from z_n are explored. In this case, we have $(\exists u \in \mathit{enabled}(z_n.\mathit{state}).\ u \notin z_n.\mathit{done})$ and $z_n.\mathit{tid} = t$ for some $t \in \mathit{Tid}$. $(\ell s_{n+1}, h_{n+1}, LS_{n+1})$ does not hit on the stack because otherwise line 45 in *Search* will set $z.\mathit{tid} = 0$. Therefore, $(\ell s_{n+1}, h_{n+1}, LS_{n+1}) \prec (\ell s_n, h_n, LS_n)$.

   Let us consider the transition given by $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha}_t (\ell s_{n+1}, h_{n+1}, LS_{n+1})$. Since $z_n.\mathit{tid} = t$, we know that the transition $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha}_t (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ is a mover by definition. Either $\alpha = \mathit{rel}(x)$ or $\alpha = (x = \mathit{fork})$ or $\alpha = (y = x.f)$ or $\alpha = (x.f = y)$.

   Let $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1}) \xrightarrow{\alpha_{n+1}}_{t_{n+1}} \ldots \xrightarrow{\alpha_l}_{t_l} (\ell s_{l+1}, h_{l+1} LS_{l+1})$ be the longest sub-execution of σ' such that $t_j \neq t$ for all $j \in [n, l]$. From Lemma 2, we get a transition $(\ell s_{l+1}, h_{l+1}, LS_{l+1}) \xrightarrow{\alpha}_t (\ell s'_{l+1}, h'_{l+1}, LS'_{l+1})$ and another execution $(\ell s'_n, h'_n, LS'_n) \xrightarrow{\alpha_n}_{t_n} (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1}) \xrightarrow{\alpha_{n+1}}_{t_{n+1}} \ldots \xrightarrow{\alpha_l}_{t_l} (\ell s'_{l+1}, h'_{l+1} LS'_{l+1})$ such that the following are true for all $j \in [n, l+1]$:

   (a) $\ell s_j(u) = \ell s'_j(u)$ for all $u \neq t$.

   (b) If $\alpha = \mathit{rel}(x)$ or $\alpha = (x = \mathit{fork})$, then $LS_j(q) = LS'_j(q)$ for all q.

   (c) If $\alpha = (y = x.f)$ or $\alpha = (x.f = y)$, then $LS_j(q) = LS'_j(q)$ for all $q \neq \ell s_1(t)(x)$.

   There are two cases: $l < k$ and $l = k$ (if $l > k$, it suffices to check the case $l = k$).

   (a) $l < k$: We have $t_{l+1} = t$, $\alpha_{l+1} = \alpha$, and $\sigma'' = (\ell s_{l+2}, h_{l+2}, LS_{l+2}) = (\ell s'_{l+1}, h'_{l+1}, LS'_{l+1})$. Thus, we get an execution $(\ell s_n, h_n, LS_n) \xrightarrow{\alpha}_t (\ell s'_n, h'_n, LS'_n) \xrightarrow{\alpha_n}_{t_n} (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1}) \ldots (\ell s'_{l+1}, h'_{l+1}, LS'_{l+1}) = (\ell s_{l+2}, h_{l+2}, LS_{l+2}) \xrightarrow{\alpha_{l+2}}_{t_{l+2}} (\ell s_{l+2}, h_{l+2}, LS_{l+2}) \cdots (\ell s'_k, h'_k, LS'_k) \xrightarrow{\alpha'_k}_{t'_k} (\ell s_{k+1}, h_{k+1}, LS_{k+1})$.

   (b) $l = k$: We get an execution $\sigma'' = (\ell s_n, h_n, LS_n) \xrightarrow{\alpha}_t (\ell s'_n, h'_n, LS'_n) \xrightarrow{\alpha_n}_{t_n} (\ell s'_{n+1}, h'_{n+1}, LS'_{n+1}) \cdots$

$(\ell s'_k, h'_k, LS'_k) \xrightarrow{\alpha_k}_{t_k} (\ell s'_{k+1}, h'_{k+1}, LS'_{k+1})$.

In both cases above, the inductive hypothesis is true for the extension $(\ell s'_n, h'_n, LS'_n) \xrightarrow{\alpha_n}_{t_n} \ldots$ due to $z_{n+1}.state = (\ell s'_n, h'_n, LS'_n)$ and $(\ell s'_{n+1}, h'_{n+1}, LS'_{n+1}) \prec (\ell s_n, h_n, LS_n)$.

There are two cases: α accesses q or α does not access q.

(a) Suppose α accesses q. Because $z_n.tid \neq 0$, we know that $n \xrightarrow{hb} k$ for σ'' by the inductive hypothesis for $(\ell s_{n+1}, h_{n+1}, LS_{n+1})$. By definition of mover, $LS_{k+1}(q) = LS'_{k+1}(q)$. If $i \xrightarrow{hb} n$ for σ', then $LS'_n(q) \cap LH_n(t) \neq \emptyset$. Then $LS'_{k+1}(q) \cap LH_{k+1}(t) \neq \emptyset$ and $i \xrightarrow{hb} n$ for σ'. $i \xrightarrow{hb} k$ for σ' by transitivity of $\xrightarrow{hb}$. Otherwise ($i \not\xrightarrow{hb} n$), due to Theorem 1, we get that the lockset algorithm sets $z_i.tid$ to 0.

(b) Suppose α does not access q. Now the inductive hypothesis for $(\ell s_{n+1}, h_{n+1}, LS_{n+1})$ applies as follows. By definition of mover, $LS_{k+1}(q) = LS'_{k+1}(q)$. If $LS'_{k+1}(q) \cap LH_{k+1}(t) = \emptyset$ then $i \not\xrightarrow{hb} k$ for σ'' and so $z_i.tid = 0$. Then $LS_{k+1}(q) \cap LH_{k+1}(t) = \emptyset$ and $z_i.tid = 0$ is obtained. If $LS'_{k+1}(q) \cap LH_{k+1}(t) \neq \emptyset$ then $i \xrightarrow{hb} k$ for both σ'' and σ'.

□

Let $len(\sigma)$ be the number of transitions in σ. We define a well-founded order $\ll$ over execution sequences starting from states explored by the algorithm as follows: For two executions $\sigma = (\ell s_n, h_n, LS_n) \xrightarrow{\alpha_n}_{t_n} \cdots$ and $\sigma' = (\ell s_m, h_m, LS_m) \xrightarrow{\alpha_m}_{t_m} \cdots$, $\sigma \ll \sigma'$ if either $len(\sigma) < len(\sigma')$ or $len(\sigma) = len(\sigma')$ and $(\ell s_n, h_n, LS_n) \prec (\ell s_m, h_m, LS_m)$.

LEMMA 4. *Suppose the algorithm explores a state* $(\ell s_1, h_1, LS_1)$ *and* $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2, LS_2) \xrightarrow{\alpha_2}_{t_2} \ldots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell s_n, h_n, LS_n)$ *is an execution such that* $\ell s_n(t) = \langle wrong, l\rangle$ *for some* t *and* l. *Then, the algorithm explores a state* $(\ell s, h, LS)$ *such that* $\ell s(t) = \langle wrong, l\rangle$.

PROOF. We perform induction over the $\ll$ on executions with increasing lengths.

Base case: For the executions of length 0, the state $(\ell s_1, h_1, LS_1)$ itself is an erroneous state and the proof is trivial.

Inductive case 1: Suppose we know that the lemma holds for all executions of length up to n. Consider an erroneous execution of length $n+1$ from $(\ell s_1, h_1, LS_1)$ where $(\ell s, h, LS)$ is the first state ever popped.

Let $z \in stack$ be the node containing the state $(\ell s, h, LS)$. If $enabled(z_n.state) = \emptyset$, then we are done immediately. Otherwise, since z is the first node to be popped from the stack, all transitions explored from z must hit on the stack. Therefore $z.tid = 0$ after each hit on the stack, and in the end all the transitions from z are explored such that $z.done = enabled(z.state)$. Suppose that a transition $(\ell s, h, LS) \xrightarrow{\alpha}_t (\ell s', h', LS')$ hits on the stack. If $(\ell s', h', LS')$ is an error state, we are done. Otherwise, we have an erroneous execution of length $\leq n$ from a state $(\ell s', h', LS')$ on the stack and we can apply the inductive hypothesis.

Inductive case 2: Suppose, we know that the lemma holds for all executions of length up to n and for all executions of length $n+1$ from states popped at time x or less. Now consider a state $(\ell s, h, LS)$ that is being popped at time $x+1$. Let z be the node such that $z.state = (\ell s, h, LS)$. There are two cases:

1. All the transitions from z are explored such that $z.done = enabled(z.state)$. Therefore, the first transition $(\ell s, h, LS) \xrightarrow{\alpha}_t (\ell s', h', LS')$, say of the erroneous extension is explored. If $(\ell s', h', LS')$ hits on the stack, then we are done because we have an erroneous execution from $(\ell s', h', LS')$ on the stack of length $\leq n$ and we can apply the inductive hypothesis. The same reasoning applies even if the transition hits in the hashtable but not on the stack.

2. Suppose not all the transitions from z are explored. In this case, we have $(\exists u \in enabled(z.state).\ u \notin z.done)$ and $z.tid = t$ for some $t \in Tid$. From Lemma 3, we can conclude that the transition $(\ell s, h, LS) \xrightarrow{\alpha}_t (\ell s', h', LS')$ of thread t explored from z is a mover. $(\ell s, h, LS)$ does not hit on the stack because otherwise line 45 in *Search* would set $z.tid = 0$ and the case above would apply. Therefore, $(\ell s, h, LS) \prec (\ell s', h', LS')$. Since $(\ell s, h, LS) \xrightarrow{\alpha}_t (\ell s', h', LS')$ is a mover, there is an erroneous execution of length $\leq n$ from $(\ell s', h', LS')$ (according to Lemma 2). We make an appeal to the inductive hypothesis on $(\ell s', h', LS')$.

□

THEOREM **2** **(Soundness).** *Let* $\sigma = (\ell s_1, h_1, LS_1) \xrightarrow{\alpha_1}_{t_1} \ldots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1}, LS_{n+1})$ *be an execution of the program where* $\ell s_{n+1}(t) = \langle wrong, l\rangle$ *for some* $t \in Tid$ *and* $l \in LocalStore$. *Then the algorithm explores a state* $(\ell s, h, LS)$ *such that* $\ell s(t) = \langle wrong, l\rangle$.

PROOF. The proof is by a straightforward application of Lemma 3 on the initial state $(\ell s_I, h_I, LS_I)$. When $(\ell s_1, h_1, LS_1)$ is popped from the stack and the exploration terminates, all states $(\ell s, h, LS)$ such that $\exists t \in Tid, l \in LocalStore.\ \ell s_{n+1}(t) = \langle wrong, l\rangle$ have been visited. □