

A New Lockset Algorithm and Its Applications

Tayfun Elmas Shaz Qadeer
Koc University Microsoft Research
telmas@ku.edu.tr qadeer@microsoft.com

October 2005

Technical Report
MSR-TR-2005-118

In this study we present a new dynamic lockset algorithm that detects race conditions from execution traces of concurrent programs. The algorithm checks if two accesses to a variable are ordered by a happens-before relation. We can handle interesting cases including object initialization, thread-locality, and dynamically changing locksets over time. Our algorithm is different from traditional algorithms for maintaining the happens-before relation that is based on clock vectors. Instead, our algorithm is purely based on computing and reasoning about locksets only. We elaborate one application of the algorithm to improve transaction-based partial order reduction in model checking. Our results show the effect of using race detection on reducing the state space.

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

1 Introduction

However, the standard lockset algorithm is unable to handle a number of interesting scenarios in which there is no race. Among them are (1) object initialization scenarios in which an object remains thread-local during its initialization, (2) thread-locality, in which an object becomes local to different threads at different times during the execution. The most interesting scenario that drove us to derive our lockset algorithm was (3) dynamically changing locksets. For programs that use complex protection mechanisms for their data, some variables can be protected by different locks according to the program state.

2 Definitions

Domains

pc	\in	PC	
$addr$	\in	$Addr$	
t, u	\in	Tid	
v	\in	$Value$	$= PC \cup Addr \cup Tid \cup Integer \cup \{\perp\}$
f	\in	$Field$	
q	\in	$Variable$	$= Addr \times Field$
x, y, z	\in	$LocalVar$	
α, α_1	\in	$Action$	$= x = new \mid x = y.f \mid x.f = y$ $\mid x = op(y_1, \dots, y_m)$ $\mid acq(x) \mid rel(x) \mid x = fork \mid join(x)$
h	\in	$Heap$	$= Addr \rightarrow Field \rightarrow Value$
l	\in	$LocalStore$	$= LocalVar \rightarrow Value$
ℓ	\in	$LocalState$	$= PC \times LocalStore$
ls	\in	$LocalStates$	$= Tid \rightarrow LocalState$
$(\ell s, h)$	\in	$State$	$= Heap \times LocalStates$

A state of a program is a pair $(\ell s, h)$. $\ell s : Tid \rightarrow LocalState$ is a partial map such that $\ell s(t)$ is the local state of thread t . The local state $\ell s(t)$ of a thread 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. Formally, the heap h is a partial function mapping addresses to a function that maps fields to values. Given address a and field f , 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.

The set Tid is the set of thread identifiers, the set $Addr$ is the set of heap addresses, and the set $Integer$ is the set of integers. Each local variable or field of a cell may contain values from the set $Tid \cup Addr \cup Integer$.

The behavior of a concurrent program is specified by a control flow graph over a set PC of control locations. A labeling function $Label : PC \rightarrow LocalVar$ labels each location with a local variable. The set of control flow edges are specified by two functions $Then : PC \rightarrow Action \times (PC \cup \{end, wrong\})$ and $Else : PC \rightarrow Action \times (PC \cup \{end, wrong\})$. 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 by failing an assertion.

The action $x = new$ allocates a new object on the heap and stores its

address in the local variable x . The action $x = y.f$ reads into x the value contained in the f field of the object whose address is in y . If y 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 $x = op(y_1, \dots, y_n)$ models local computation such as addition, subtraction, comparison, etc.

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 writes the identifier of the executing thread into $x.owner$. The action $rel(x)$ releases the lock on the object whose address is contained in x . 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.

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 following rules where we do a case analysis on α .

Transition relation

(ALLOCATE)	$\alpha = (x = new) \quad h(addr) = \perp$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l[x := addr] \rangle], h[addr := \lambda_I])}$
(READHEAP)	$\alpha = (x = y.f) \quad h(l(y)) \neq \perp$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l[x := h(l(y), f)] \rangle], h)}$
(READHEAP FAIL)	$\alpha = (x = y.f) \quad h(l(y)) = \perp$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$
(WRITEHEAP)	$\alpha = (x.f = y) \quad h(l(x)) \neq \perp$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle], h[l(l(x), f) := l(y)])}$
(WRITEHEAP FAIL)	$\alpha = (x.f = y, pc') \quad h(l(x)) = \perp$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$
(OPERATION)	$\alpha = (x = op(y_1, \dots, y_m))$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l[x := op(l(y_1), \dots, l(y_m))] \rangle], h)}$
(ACQUIRE)	$\alpha = acq(x) \quad h(l(x), owner) = 0$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle pc', l \rangle], h[l(l(x), owner) := t])}$
(ACQUIRE FAIL)	$\alpha = acq(x) \quad h(l(x)) = \perp$
	$\frac{}{(\ell s, h) \xrightarrow{\alpha}_t (\ell s[t := \langle wrong, l \rangle], h)}$

$$\begin{array}{l}
\text{(RELEASE)} \\
\frac{\alpha = \text{rel}(x) \quad h(l(x), \text{owner}) = t}{(\ell s, h) \xrightarrow{\alpha}_{t_1} (\ell s[t := \langle pc', l \rangle], h[l(x), \text{owner}] := 0)} \\
\text{(RELEASE FAIL)} \\
\frac{\alpha = \text{rel}(x) \quad (h(l(x)) = \perp \vee h(l(x), \text{owner}) \neq t)}{(\ell s, h) \xrightarrow{\alpha}_{t_1} (\ell s[t := \langle \text{wrong}, l \rangle], h)} \\
\text{(FORK)} \\
\frac{\alpha = (x = \text{fork}) \quad \ell s(u) = \perp}{(\ell s, h) \xrightarrow{\alpha}_{t_1} (\ell s[t := \langle pc', l \rangle][u := \langle pc_l, l \rangle], h)} \\
\text{(JOIN)} \\
\frac{\alpha = \text{join}(x) \quad \ell s(l(x)) = \langle \text{end}, l' \rangle}{(\ell s, h) \xrightarrow{\alpha}_{t_1} (\ell s[t := \langle pc', l \rangle], h)}
\end{array}$$

An execution σ of the program is a finite sequence $(\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$ such that $(\ell s_1, h_1) = (\ell s_l, h_l)$ and $(\ell s_k, h_k) \xrightarrow{\alpha_k}_{t_k} (\ell s_{k+1}, h_{k+1})$ for all $1 \leq k \leq n$.

3 Lockset Algorithm

In this section, we describe an algorithm to detect if two accesses to a variable are ordered by the happens-before relation. Our algorithm is different from traditional algorithms for maintaining the happens-before relation based on clock vectors. Instead, our algorithm only maintains locksets.

The algorithm introduced in this section accepts the locking discipline in which any accesses, whether reads or writes, to a variable are mutually-exclusive: We will call this discipline **MutExAccessDiscipline**. A thread that obeys **MutExAccessDiscipline** must acquire the locksets for the variable before reading or writing to the variable. The locking discipline forces to programmer to obey a concurrent programming guideline and helps him to avoid races while developing the program. One of our main conclusions is that we can prove that the program obeys a particular locking discipline, we can show that it is race-free.

The other locking disciplines and extensions to the lockset algorithm due to these disciplines are given in the subsequent sections.

Let $\sigma = (\ell s_1, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell s_2, h_2) \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_n}_{t_n} (\ell s_{n+1}, h_{n+1})$ be an execution of the program. Let s_i denote the i^{th} state $(\ell s_i, h_i)$ along the execution.

Consider an action α_k ($1 \leq k \leq n$) in the execution σ . If $\alpha_k = (x = y.f)$, then thread t_k reads the heap variable $(\ell s_k(t_k)(y), f)$. If $\alpha_k = (x.f = y)$, then thread t_k writes the heap variable $(\ell s_k(t_k)(x), f)$. The thread t_k accesses the variable (a, f) if it either reads or writes (a, f) .

DEFINITION 1. *The execution σ is race-free if there is no state s_i ($1 \leq i \leq n+1$) such that (1) there are two actions α_k and α_m from different threads that are enabled at s_i , (2) $\alpha_k = (x.f = y)$ and (3) $\alpha_m = (x.f = z)$ or $\alpha_m = (z = x.f)$.*

DEFINITION 2. *A program is race-free if all executions of the program are race-free.*

We will reason about race-freedom of a execution using a happens-before relation based only on operations *acq* and *rel* on locks, and *fork* and *join* on threads.

DEFINITION 3. *The happens-before relation \xrightarrow{hb} for σ is the smallest transitively-closed relation on the set $\{1, 2, \dots, n\}$ such that for any k and l , $k \xrightarrow{hb} l$ if $1 \leq k < l \leq n$ and one of the following holds:*

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

LEMMA 1. *An execution σ obeys **MutExAccessDiscipline** iff whenever α_k and α_l , along σ , access (a, f) and $k < l$ then $k \xrightarrow{hb} l$.*

LEMMA 2. *A program obeys **MutExAccessDiscipline** if all executions of the program obeys **MutExAccessDiscipline**.*

THEOREM 1. *A program is race-free if it obeys **MutExAccessDiscipline**.*

The lockset algorithm, given an execution σ as input, shows that there is a happens-before relation \xrightarrow{hb} between any two accesses to any variable along σ . Then using the lemmas above we conclude that σ is race-free.

For the basic usage of the algorithm for dynamic race detection at runtime, we do not give any arguments about race-freedom of the program itself. One must apply model checking on the program to show that all executions of the program is race-free by applying the dynamic race detection and concluding that the all possible executions of the program is race-free. We will give a model checking algorithm that makes use of the lockset algorithm to (1) detect races and (2) reduce redundant execution paths for transaction-based partial-order reduction.

In this section, we describe an algorithm to detect if two accesses to a variable are ordered by the happens-before relation. Our algorithm is different from traditional algorithms for maintaining the happens-before relation based on clock vectors. Instead, our algorithm only maintains locksets.

The algorithm introduced in this section accepts the locking discipline in which any accesses, whether reads or writes, to a variable are mutually-exclusive: A thread must acquire the locksets for the variable before reading or writing to the variable. The extensions to the algorithm for other locking disciplines are given in the subsequent sections.

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

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

Consider an action α_k in the execution σ . If $\alpha_k = (x = y.f)$, then

thread t_k reads the heap variable $(\ell_{s_k}(t_k)(y), f)$. If $\alpha_k = (x.f = y)$, then thread t_k writes the heap variable $(\ell_{s_k}(t_k)(x), f)$. The thread t_k accesses the variable (a, f) if it either reads or writes (a, f) . The execution σ is *race-free* for (a, f) if whenever α_k and α_l access (a, f) and $k < l$ then $k \xrightarrow{hb} l$.

We define two maps, LH and LS , and the algorithm for initializing them and updating them as the program executes. LH is a partial function from Tid to $Powerset(Addr \cup Tid)$. Additionally, if $LH(t)$ is defined then $LH(t) \subseteq Addr \cup \{t\}$. Initially, LH is defined only at t_1 and $LH(t_1) = \{t_1\}$. LS is a partial function from $Variable$ to $Powerset(Addr \cup Tid)$. Initially, LS is not defined for any variable.

Suppose thread t executes an action α to go from state (ℓ, h) to (ℓ', h') . Let $\ell_s(t) = \langle pc, l \rangle$ and $\ell_{s'}(t) = \langle pc', l' \rangle$. We update the partial maps LH and LS by doing a case analysis on α as follows:

1. $x = \text{new}$: For all $f \in Field$, initialize $LS(l'(x), f)$ to $Addr \cup Tid$.
2. $x = y.f$: Let $q = (l(y), f)$.
If $(LS(q) \cap LH(t) \neq \emptyset)$
Then
 $LS(q) = LH(t)$
Else
 $LS(q) = \emptyset$
3. $x.f = y$: Let $q = (l(x), f)$.
If $(LS(q) \cap LH(t) \neq \emptyset)$
Then
 $LS(q) = LH(t)$
Else
 $LS(q) = \emptyset$
4. $acq(x)$: $LH(t) = \{l(x)\} \cup LH(t)$. For all $q \in Variable$ such that $LS(q) \neq \perp$:
If $(LH(t) \cap LS(q) \neq \emptyset)$
Then
 $LS(q) = LH(t) \cup LS(q)$
5. $rel(x)$: $LH(t) = LH(t) \setminus \{l(x)\}$.
6. $x = \text{fork}$: For all $q \in Variable$ such that $LS(q) \neq \perp$:
If $(LH(t) \cap LS(q) \neq \emptyset)$
Then
 $LS(q) = \{l'(x)\} \cup LS(q)$
Initialize $LH(l'(x)) = \{l'(x)\}$.
7. $join(x)$: For all $q \in Variable$ such that $LS(q) \neq \perp$:
If $(LH(l(x)) \cap LS(q) \neq \emptyset)$
Then
 $LS(q) = \{t\} \cup LS(q)$
8. $x = op(y_1, \dots, y_m)$: No update on locksets.

4 Correctness Proof of The Algorithm

LEMMA 3. Let $\sigma = (\ell_{s_1}, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell_{s_2}, h_2) \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell_n, h_n)$ be an execution and (a, f) a heap variable such that $h_n(a, f) \neq \perp$. If applying the algorithm above to σ ends with $LS(a, f) \neq \emptyset$, there is a happens before relation \xrightarrow{hb} between any two accesses α_i and α_j ($1 \leq i < j \leq n-1$) to (a, f) .

PROOF. Let q be a variable such that $q \mapsto a.f$. Let s_i denote the state (ℓ_{s_i}, h_i) and $LS_i(q)$ and $LH_i(t)$ denote locksets for q and t at state s_i for any $1 \leq i \leq n$.

The proof will be by induction on the accesses to q that for any prefix $\sigma' = s_1 \xrightarrow{\alpha_1}_{t_1} s_2 \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_{m-1}}_{t_{m-1}} s_m$ ($m \leq n$) of σ , if applying the algorithm to σ' ends with $LS_m(a, f) \neq \emptyset$, there is a happens before relation \xrightarrow{hb} between any two accesses α_i and α_j ($1 \leq i < j \leq m-1$) to (a, f) .

Base Case: Let α_l be the first access to q by thread t . $LS(q)$ is initialized to $Addr \cup Tid$ at creation point of $l(x)$ so that $LS_l(q) = Addr \cup Tid$. It is the case that $\{t\} \in Addr \cup Tid \cap LH_l(t)$. After α_l is completed $LH(t) \cap LS_l(q) = LH(t) \cap Addr \cup Tid \neq \emptyset$ and then according to update rules for LS , $LS_{l+1}(q) = LH_{l+1}(t) \neq \emptyset$. Since α_l is the first access to q , for $m = l+1$ the happens before relation is valid for the only action α_l .

Induction Step: Let α_i , α_k and α_j ($1 \leq i < k < j \leq n$) be three accesses to q . Since \xrightarrow{hb} is transitive-closure, $((\alpha_i \xrightarrow{hb} \alpha_k) \wedge (\alpha_k \xrightarrow{hb} \alpha_j)) \implies (\alpha_i \xrightarrow{hb} \alpha_j)$. Then we omit any such α_k ($i < k < j$) that access q and show that $i \xrightarrow{hb} j$ for α_i and α_j when there is no such α_k ($i < k < j$).

Let α_i and α_j ($1 \leq i < j \leq m$) be two accesses to q . Assume that $LS_{i+1}(q) \neq \emptyset$ and there is some happens-before relation such that $I \xrightarrow{hb} \dots \xrightarrow{hb} i$. Now we will prove that if $LS_{j+1}(q) \neq \emptyset$ it also holds that $I \xrightarrow{hb} \dots \xrightarrow{hb} i \xrightarrow{hb} j$. For $LS_{j+1}(q) \neq \emptyset$ to be true, it must hold that $LS_j(q) \cap LH_j(t_j) \neq \emptyset$. Let $LH_j^\cap = LS_j(q) \cap LH_j(t_j)$.

If $t_i = t_j$ then it follows that $i \xrightarrow{hb} j$ from the definition of \xrightarrow{hb} . Note that in this case $LH_j^\cap \neq \emptyset$ thus because $LH_i(t_i) = LH_j(t_j)$ and we assumed that $LS_j(q) \cap LH_j(t_j) \neq \emptyset$.

Therefore the proof continues with the fact $t_i \neq t_j$. There are two cases for contents of LH_j^\cap :

Case 1. ℓ is allocated in σ and $\ell \in LH_j^\cap$. Then there exists $\alpha_k = rel(x)$ and $\alpha_l = acq(y)$ ($i < k < l < j$) such that $\ell_{s_k}(t_k)(x) = \ell_{s_l}(t_l)(y) = \ell$, $t_k = t_i$ and $t_l = t_j$. In this case $i \xrightarrow{hb} k \xrightarrow{hb} l \xrightarrow{hb} j$.

Proof. Recall that $t_i \neq t_j$. Because acquisition of lock ℓ is mutually-exclusive, there must be some $rel(x)$ by t_i (at state s_k) and $acq(y)$ by t_j (at state s_l) on ℓ where $\ell_{s_k}(t_k)(x) = \ell_{s_l}(t_l)(y) = \ell$. Then because $t_k = t_i$ $i \xrightarrow{hb} k$ and because $t_l = t_j$ $l \xrightarrow{hb} j$. Since both $rel(x)$ and $acq(y)$ on ℓ $k \xrightarrow{hb} l$. Since \xrightarrow{hb} is transitive-closure it follows that $i \xrightarrow{hb} k \xrightarrow{hb} l \xrightarrow{hb} j$.

Case 2. $t_j \in LH_j^\cap$. Then there exists $\alpha_k = (x = fork(x))$ ($i < k < j$) such that $\ell_{s_{k+1}}(t_k)(x) = t_j$. In this case $i \xrightarrow{hb} k \xrightarrow{hb} j$.

Proof. First, if $t_k = t_i$, then it follows that $i \xrightarrow{hb} k$ and $k \xrightarrow{hb} j$, that yields $i \xrightarrow{hb} k \xrightarrow{hb} j$ from the transitive-closure relation. Now assume that $t_k \neq t_i$.

Since there is no $t \in LH_j(t_j)$ such that $t \neq t_j$, there must be some action $\alpha_k = (x = fork(x))$ ($\ell_{s_{k+1}}(t_k)(x) = t_j$) that adds t to $LS_{k+1}(q)$. The update rule for $fork$ requires that $LS_k(q) \cap LH_k(t_k) \neq \emptyset$. Let $LH_k^\cap = LS_k(q) \cap LH_k(t_k)$. Case 1 or 2 applies recursively. Let Case 1 apply; then there is some lock $\ell \in LH_k^\cap$ and proof of Case 1 applies. Now let Case 2 apply recursively; it yields an ac-

tion $\alpha_i = (x = \text{fork}(x))$ ($\ell_{s_{l+1}}(t_l)(x) = t_k$) such that $t_l = t_i$. It gives $i \xrightarrow{hb} l \xrightarrow{hb} \dots \xrightarrow{hb} k \xrightarrow{hb} j$.

□

LEMMA 4. Let $\sigma = (\ell_{s_1}, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell_{s_2}, h_2) \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_{n-1}}_{t_{n-1}} (\ell_{s_n}, h_n)$ be an execution and (a, f) a heap variable such that $h_n(a, f) \neq \perp$. If there is a happens before relation \xrightarrow{hb} between any two accesses α_i and α_j ($1 \leq i < j \leq n-1$) to (a, f) , then applying the algorithm above to σ ends with $LS(a, f) \neq \emptyset$.

PROOF. Let q be a variable such that $q \mapsto a.f$. Let s_i denote the state (ℓ_{s_i}, h_i) and $LS_i(q)$ and $LH_i(t)$ denote locksets for q and t at state s_i for any $1 \leq i \leq n$.

The proof will be by induction on the accesses to q that for any prefix $\sigma' = s_1 \xrightarrow{\alpha_1}_{t_1} s_2 \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_{m-1}}_{t_{m-1}} s_m$ ($m \leq n$) of σ , if there is a happens before relation \xrightarrow{hb} between any two accesses α_i and α_j ($1 \leq i < j \leq m-1$) to (a, f) , then applying the algorithm above to σ' ends with $LS_m(a, f) \neq \emptyset$.

Base Case: Let α_I be the first access to q by thread t . $LS(q)$ is initialized to $Addr \cup Tid$ at creation point of $l(x)$ so that $LS_I(q) = Addr \cup Tid$. It is the case that $\{t\} \in Addr \cup Tid \cap LH_I(t)$. After α_I is completed $LH(t) \cap LS_I(q) = LH(t) \cap Addr \cup Tid \neq \emptyset$ and then according to update rules for LS , $LS_{I+1}(q) = LH_{I+1}(t) \neq \emptyset$. Since α_I is the first access to q , for $m = I + 1$ the happens before relation is valid for the only action α_I .

Induction Step: Let α_i , α_k and α_j ($1 \leq i < k < j \leq n$) be three accesses to q . Since \xrightarrow{hb} is transitive-closure, $((\alpha_i \xrightarrow{hb} \alpha_k) \wedge (\alpha_k \xrightarrow{hb} \alpha_j)) \implies (\alpha_i \xrightarrow{hb} \alpha_j)$. Then we omit any such α_k ($i < k < j$) that access q and show that $i \xrightarrow{hb} j$ for α_i and α_j when there is no such α_k ($i < k < j$).

Let α_i and α_j ($1 \leq i < j \leq m$) be two accesses to q . Assume that $LS_{i+1}(q) \neq \emptyset$ and there is some happens-before relation such that $I \xrightarrow{hb} \dots \xrightarrow{hb} i$. Now we will prove that if $i \xrightarrow{hb} j$ then $LS_{j+1}(q) \neq \emptyset$. Because $LS_{j+1}(q) \neq \emptyset$, it must hold that $LS_j(q) \cap LH_j(t_j) \neq \emptyset$. Let $LH_j^\cap = LS_j(q) \cap LH_j(t_j)$. We will split the proof for different cases that yield \xrightarrow{hb} :

Case 1. $t_j = t_i$, and thus it holds that $i \xrightarrow{hb} j$ from the definition of \xrightarrow{hb} . Additionally there is no action α_k ($i < k < j$) such that $t_k = t_j$.

Proof. Because $LS_j(q) \cap LH_j(t_j) \neq \emptyset$ and $LH_i(t_i) = LH_j(t_j)$, it follows that $LH_j^\cap \neq \emptyset$ which causes the assignment that makes $LS_{j+1}(q) = LH_j(t_j) \neq \emptyset$ true.

Case 2. There are some $\alpha_k = \text{rel}(x)$ and $\alpha_l = \text{acq}(y)$ ($i < k < l < j$) on ℓ where $t_k = t_i$, $t_l = t_j$, and $\ell_{s_k}(t_k)(x) = \ell_{s_l}(t_l)(y) = \ell$.

Proof. Because $LS_{j+1}(q) = LH_j(t_j)$ it holds that $\ell \in LS_{i+1}(q)$. Because $LS(q)$ remains the same or increases until s_l , $\ell \in LS_l(q)$. Then according to the update rule for acq , because $LH_l^\cap \neq \emptyset$ that yields $\ell \in LS_{l+1}(q)$ and $t_l \in LS_{l+1}(q)$.

Case 3. There are some $\alpha_k = (x = \text{fork}(x))$ ($i < k < j$) where $t_k = t_i$ and $\ell_{s_{k+1}}(t_k)(x) = t_j$.

Proof. Because $LS_{j+1}(q) = LH_j(t_j)$ it holds that $t_i \in LS_{i+1}(q)$. Because $LS(q)$ remains the same or increases until s_k , $t_i \in LS_k(q)$. Then according to the update rule for fork , because $LH_k^\cap \neq \emptyset$ that yields $t_j \in LS_{k+1}(q)$.

Case 4. There are some $\alpha_k = \text{join}(x)$ ($i < k < j$) where $t_k = t_j$ and $\ell_{s_k}(t_k)(x) = t_i$.

Proof. Because $LS_{j+1}(q) = LH_j(t_j)$ it holds that $t_i \in LS_{i+1}(q)$. Because $LS(q)$ remains the same or increases until s_k , $t_i \in LS_k(q)$. Then according to the update rule for join , because $LH_k^\cap \neq \emptyset$ that yields $t_j \in LS_{k+1}(q)$.

□

THEOREM 2. Let $\sigma = (\ell_{s_1}, h_1) \xrightarrow{\alpha_1}_{t_1} (\ell_{s_2}, h_2) \xrightarrow{\alpha_2}_{t_2} \dots \xrightarrow{\alpha_m}_{t_m} (\ell_{s_{n+1}}, h_{n+1})$ be an execution and (a, f) a heap variable such that $h_{n+1}(a, f) \neq \perp$. Then σ is race-free for (a, f) iff applying the algorithm does not report any race condition, meaning that $LS(a, f)$ never gets empty during σ and σ ends with $LS(a, f) \neq \emptyset$.

PROOF. From Lemma 1 and Lemma 2, it follows that (1) if σ is race-free due to the happens-before relation by acq , rel , fork and join operations, $LS(a, f)$ never gets empty for a variable (a, f) and (2) if the algorithm does not trigger any race condition, then there is a happens-before relation by the same set of operations. □

5 Stateless model checking

```
record Node {
  State state;
  (Variable  $\rightarrow$  Powerset(Tid  $\cup$  Addr)) LS;
  (Variable  $\rightarrow$  Node) la;
  (Tid  $\cup$  {0}) tid;
  Powerset(Tid) done;
}
```

```
Search() {
  Node curr = new Node;
  curr.state = ( $\ell_I, h_I$ );
  curr.LS =  $\lambda q \in$  Variable. Addr  $\cup$  Tid;
  curr.la =  $\lambda q \in$  Variable. null;
  curr.tid = 0;
  curr.done =  $\emptyset$ ;
```

```
Stack(Node) stack = new Stack(Node);
stack.Push(curr);
```

```
while ( $\neg$ stack.IsEmpty()) {
  Tid t;
  curr = stack.Peek();
  if (curr.tid = 0  $\wedge$  done  $\subset$  enabled(curr.state))
    t = choose(enabled(curr.state) \ done);
  elsif (curr.tid  $\neq$  0  $\wedge$  curr.tid  $\notin$  done)
    t = curr.tid;
  else {
    stack.Pop();
    continue;
  }
  curr.done = curr.done  $\cup$  {t};
  stack.Push(Successor(curr, t));
}
```

```

Node Successor(Node curr, Tid t) {
  Heap h, h';
  LocalStates ls, ls';
  Action  $\alpha$ ;
  (h, ls) = curr.state;
  let (h, ls)  $\xrightarrow{\alpha}_t$  (h', ls');
  Node next = new Node;
  next.state = (h', ls');

  switch ( $\alpha$ ) {
    case x = new :
    case x = op(y1, ..., ym) :
      next.LS = curr.LS;
      next.la = curr.la;

    case y = x.f :
    case x.f = y :
      Variable q = (ls(t)(x), f);
      if (curr.LS(q)  $\cap$  LH(curr.state, t) =  $\emptyset$ )
        curr.tid = curr.la(q).tid = 0;
      next.LS = curr.LS[q := LH(curr.state, t)];
      next.la = curr.la[q := next];

    case acq(x) :
      curr.tid = 0;
      next.LS =  $\lambda q \in \text{Variable.}$ 
        (curr.LS(q)  $\cap$  LH(curr.state, t)  $\neq \emptyset$ )
        ? curr.LS(q)  $\cup$  LH(curr.state, t)
        : curr.LS(q);
      next.la = curr.la;

    case rel(x) :
      next.LS = curr.LS;
      next.la = curr.la;

    case x = fork :
      next.LS =  $\lambda q \in \text{Variable.}$ 
        (curr.LS(q)  $\cap$  LH(curr.state, t)  $\neq \emptyset$ )
        ? curr.LS(q)  $\cup$  {ls'(t)(x)}
        : curr.LS(q);
      next.la = curr.la;

    case join(x) :
      curr.tid = 0;
      Tid t' = ls(t)(x);
      next.LS =  $\lambda q \in \text{Variable.}$ 
        (curr.LS(q)  $\cap$  LH(curr.state, t')  $\neq \emptyset$ )
        ? curr.LS(q)  $\cup$  {t}
        : curr.LS(q);
      next.la = curr.la;
  }

  next.tid = (t  $\in$  enabled(next.s)) ? t : 0;
  next.done = 0;
  return next;
}

```

6 Extending The Lockset Algorithm for Concurrent Reads

The lockset algorithm introduced in Section 3 accepts the locking discipline in which accesses a variable are mutually-

exclusive. However, a locking discipline in which reads are concurrent increases performance while still guaranteeing race-freedom. This section extends the lockset algorithm for concurrent-reads/mutually-exclusive-writes.

To extend the scheme given in Section 3, we divide LS into two separate maps LSR and LSW . LSW is a partial function from Variable to $\text{PowerSet}(\text{Addr} \cup \text{Tid})$ and LSR is a partial function from $\text{Variable} \times \text{Tid}$ to $\text{PowerSet}(\text{Addr} \cup \text{Tid})$. Initially, both are not defined for any variable and thread.

Suppose that thread t executes an action α to go from state (ls, h) to (ls', h') . Let $ls(t) = \langle pc, l \rangle$ and $ls'(t) = \langle pc', l' \rangle$. We update the partial maps LH , LSR and LSW by doing a case analysis on α as follows:

1. $x = \text{new}$: Let $q = (l'(x), f)$. For all $f \in \text{Field}$, initialize both $LSR(q, t)$ and $LSW(q)$ to $\text{Addr} \cup \text{Tid}$.
2. $x = y.f$: Let $q = (l(y), f)$.
If $((LSW(q) \cap LH(t) \cap LSR(q, t) \neq \emptyset))$
Then
 $LSR(q, t) = LSW(q) \cap LH(t) \cap LSR(q, t)$
Else
 $LSW(q) = \emptyset$
 $\forall t'. LSR(q, t') = \emptyset$
3. $x.f = y$: Let $q = (l(y), f)$.
If $((LSW(q) \cap LH(t) \neq \emptyset)$ and $\forall t' \neq t. (LSW(q) \cap LH(t) \cap LSR(q, t') \neq \emptyset))$
Then
 $LSW(q) = LH(t)$
 $\forall t'. LSR(q, t') = LSW(q)$
Else
 $LSW(q) = \emptyset$
 $\forall t'. LSR(q, t') = \emptyset$.
4. $\text{acq}(x)$: $LH(t) = \{l(x)\} \cup LH(t)$. For all $q \in \text{Variable}$ such that $LS(q) \neq \perp$,
If $((LSW(q) \cap LH(t) \neq \emptyset)$ and $\forall t' \neq t. (LSW(q) \cap LH(t) \cap LSR(q, t') \neq \emptyset))$
Then
 $LSW(q) = LSW(q) \cup LH(t)$
 $\forall t'. LSR(q, t') = LSW(q)$
5. $\text{rel}(x)$: $LH(t) = LH(t) \setminus \{l(x)\}$.
6. $x = \text{fork}$: For all $q \in \text{Variable}$ such that $LS(q) \neq \perp$,
If $((LSW(q) \cap LH(t) \neq \emptyset)$ and $\forall t' \neq t. (LSW(q) \cap LH(t) \cap LSR(q, t') \neq \emptyset))$
Then
 $LSW(q) = LSW(q) \cup \{l'(x)\}$
 $\forall t'. LSR(q, t') = LSR(q, t') \cup \{l'(x)\}$
7. $\text{join}(x)$: For all $q \in \text{Variable}$ such that $LS(q) \neq \perp$,
If $((LSW(q) \cap LH(l(x)) \neq \emptyset)$ and $\forall t' \neq t. (LSW(q) \cap LH(l(x)) \cap LSR(q, t') \neq \emptyset))$
Then
 $LSW(q) = LSW(q) \cup \{t\}$
 $\forall t'. LSR(q, t') = LSR(q, t') \cup \{t\}$
8. $x = \text{op}(y_1, \dots, y_m)$: No update on locksets.