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.
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

<table>
<thead>
<tr>
<th>Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc ∈ PC</td>
</tr>
<tr>
<td>addr ∈ Addr</td>
</tr>
<tr>
<td>t, u ∈ Tid</td>
</tr>
<tr>
<td>v ∈ Value = PC ∪ Addr ∪ Tid ∪ Integer ∪ {⊥}</td>
</tr>
<tr>
<td>f ∈ Field</td>
</tr>
<tr>
<td>q ∈ Variable</td>
</tr>
<tr>
<td>x, y, z ∈ LocalVar</td>
</tr>
<tr>
<td>α, α₁</td>
</tr>
<tr>
<td>Action</td>
</tr>
<tr>
<td>h ∈ Heap</td>
</tr>
<tr>
<td>l ∈ LocalStore</td>
</tr>
<tr>
<td>t ∈ LocalState</td>
</tr>
<tr>
<td>(s, h) ∈ LocalStates</td>
</tr>
<tr>
<td>(b, h) ∈ State</td>
</tr>
</tbody>
</table>

A state of a program is a pair (s, h). bs : Tid → LocalState is a partial map such that bs(t) is the local state of thread t. The local state bs(t) of a thread t consists 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 ∪ Addr ∪ 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 → LocalVar labels each location with a local variable. The set of control flow edges are specified by two functions Then : PC → Action × (PC ∪ {end, wrong}) and Else : PC → Action × (PC ∪ {end, wrong}). Suppose Label(pc) = x, Then(pc) = (α₁, pc₁), and Else(pc) = (α₂, pc₂). 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 α₁ and goes to pc₁. If the value of x is zero, then it executes the action α₂ and goes to pc₂. 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(y₁, ..., yₙ) 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 α →₁ State × State, where t ∈ Tid is a thread identifier and α ∈ Action is an action. This relation gives the transitions of thread t. Program execution starts with a single thread with identifier t₀ ∈ Tid at control location p₀. The initial state of the program is (⟨t₀, h₀⟩, h₀) = (⟨p₀, l₀⟩, l₀) and undefined elsewhere, and the heap h₀ is not defined at any address. The initial local store l₀ of thread t₀ 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 α →₁.

Let (s, h) be a state such that (s(t) = ⟨p₀, l⟩) and Label(p₀) = z. Let (α, pc) = Then(p₀) if l(z) ≠ 0 and Else(p₀) otherwise. Then, the relation α →₁ is given by the following rules where we do a case analysis on α.

Transition relation

(ALLOCATE) α = (x = new) h(addr) = ⊥

(READHEAP) α = (x = y.f) h(l(y)) = ⊥

(READHEAP FAIL) α = (x = y.f) h(l(y)) = ⊥

(WRITEHEAP) α = (x = y.f) h(l(x)) = ⊥

(WRITEHEAP FAIL) α = (x = y.f) h(l(x)) = ⊥

(OPERATION) α = (x = op(y₁, ..., yₙ))

(ACQUIRE) α = acq(x) h(l(x), owner) = 0

(ACQUIRE FAIL) α = acq(x) h(l(x)) = ⊥
An execution $\sigma$ of the program is a finite sequence $(\ell_1, h_1) \xrightarrow{\alpha_1} \ell_2 \xrightarrow{\alpha_2} \ell_3 \xrightarrow{\alpha_3} \ldots \xrightarrow{\alpha_n} \ell_{n+1} \xrightarrow{\beta} h_{n+1}$ such that $(\ell_1, h_1) = (s_1, h_1)$ and $(\ell_{k+1}, h_{k+1})$ 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 $\text{MutExAccessDiscipline}$. A thread that obeys $\text{MutExAccessDiscipline}$ must acquire the locksets for the variable before reading or writing to the variable. The lockset 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 $\sigma = (\ell_1, h_1) \xrightarrow{\alpha_1} (\ell_2, h_2) \xrightarrow{\alpha_2} (\ell_3, h_3) \ldots \xrightarrow{\alpha_n} (\ell_{n+1}, h_{n+1})$ be an execution of the program. Let $s_i$ denote the $\ell_i^h$ state $(\ell_i, h_i)$ along the execution.

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

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

**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 $\sigma$ is the smallest transitively-closed relation on the set $\{1, 2, \ldots, n\}$ 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. $\alpha_k = rel(x, a)$, $\alpha_l = acq(y)$, and $\ell_k(t_k)(x) = \ell_l(t_l)(y)$.
3. $\alpha_k = (x = fork)$ and $t_l = \ell_k(t_k)(x)$.
4. $\alpha_l = join(x)$ and $t_k = \ell_l(t_l)(x)$.

**Lemma 1.** An execution $\sigma$ obeys $\text{MutExAccessDiscipline}$ iff whenever $\alpha_k$ and $\alpha_l$, along $\sigma$, access $(a, f)$ and $k < l$ then $k \xrightarrow{hb} l$.

**Lemma 2.** A program obeys $\text{MutExAccessDiscipline}$ if all executions of the program obeys $\text{MutExAccessDiscipline}$.

**Theorem 1.** A program is race-free if it obeys $\text{MutExAccessDiscipline}$.

The lockset algorithm, given an execution $\sigma$ as input, shows that there is a happens-before relation $\xrightarrow{hb}$ between any two accesses to any variable along $\sigma$. Then using the lemmas above we conclude that $\sigma$ 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 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: We will call this discipline $\text{MutExAccessDiscipline}$. A thread that obeys $\text{MutExAccessDiscipline}$ must acquire the locksets for the variable before reading or writing to the variable. The lockset 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 $\sigma = (\ell_1, h_1) \xrightarrow{\alpha_1} (\ell_2, h_2) \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} (\ell_{n+1}, h_{n+1})$ be an execution of the program. Let $s_i$ denote the $\ell_i^h$ state $(\ell_i, h_i)$ along the execution.

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

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

**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.
thread \( t \) reads the heap variable \((b_0(t_0))((y), f)\). If \( a_k = (x, f = y)\), then thread \( t \) writes the heap variable \((b_0(t_0))((x), f)\). The thread \( t \) accesses the variable \((a, f)\) if it either reads or writes \((a, f)\). The execution \( \sigma \) is race-free for \((a, f)\) if whenever \( \alpha_k \) and \( \alpha_j \) 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 \( t \) and \( LH(t) = \{ t \} \). \( LS \) is a partial function from \( Variables(Addr \cup Tid) \). Initially, \( LS \) is not defined for any variable.

Suppose thread \( t \) executes an action \( \alpha \) to go from state \((ts, h)\) to \((ts', h')\). Let \( ts(t) = (pc, l) \) and \( ts'(t) = (pc', l') \). We update the partial maps \( LH \) and \( LS \) by doing a case analysis on \( \alpha \) as follows:

1. \( x = new: \) 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 \( q \in Variables \) such that \( LS(q) \neq \emptyset \):
   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. \( x = fork: \) For all \( q \in Variables \) such that \( LS(q) \neq \emptyset \):
   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 \( q \in Variables \) such that \( LS(q) \neq \emptyset \):
   If \((LH(l(x)) \cap LS(q)) \neq \emptyset \) Then \( LS(q) = \{ t \} \cup LS(q) \).
8. \( x = op(y_1, . . . , y_m): \) No update on locksets.

4 Correctness Proof of The Algorithm

**Lemma 3.** Let \( \sigma = (b_0, h) \) \( (b_0, h) = \alpha_0 t_1 \alpha_1 t_2 \alpha_2 t_3 \ldots \alpha_{m-2} t_{m-1} \alpha_{m-1} t_m \) be an execution and \( (a, f) \) a heap variable such that \( h_n(a, f) \neq \emptyset \). If applying the algorithm above to \( \sigma \) ends with \( LS(a, f) \neq \emptyset \), there is a happens before relation \( \xrightarrow{hb} \) between any two accesses \( \alpha_k \) and \( \alpha_j \) \((1 \leq i < j \leq m - 1)\) to \((a, f)\).

**Proof.** Let \( q \) be a variable such that \( q \xrightarrow{a, f} \). Let \( s_i \) denote the state \((b_0, h_i)\) and \( LS(q) \) and \( LH(t) \) 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 \( \sigma' = t_1 \alpha_0 t_1 t_2 \alpha_2 t_3 \ldots \alpha_{m-1} t_{m-1} \) \((m \leq n)\) of \( \sigma \), if applying the algorithm to \( \sigma' \) ends with \( LS_m(a, f) \neq \emptyset \), there is a happens before relation \( \xrightarrow{hb} \) between any two accesses \( \alpha_k \) and \( \alpha_j \) \((1 \leq i < j \leq m - 1)\) to \((a, f)\).

**Base Case:** Let \( \alpha_0 \) 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 \( LS_0(q) = Addr \cup Tid \). It is the case that \((t) \in Addr \cup Tid \cap LH(t) \). After \( \alpha_0 \) is completed \( LH(t) \cap LS_1(q) \neq \emptyset \) and then according to update rules for \( LS \), \( LS_1(q) = LH_1(t) \neq \emptyset \). Since \( \alpha_0 \) is the first access to \( q \) for \( m = 1 \) the happens before relation is valid for the only action \( \alpha_0 \).

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

Let \( \alpha_i \) and \( \alpha_j \) \((1 \leq i < j \leq m)\) be two accesses to \( q \). Assume that \( LS_{i+j}(q) \neq \emptyset \) and there is some happens-before relation such that \( i \xrightarrow{hb} \ldots \xrightarrow{hb} \alpha_0 \). Now we will prove that if \( LS_{i+j}(q) \neq \emptyset \) it also holds that \( i \xrightarrow{hb} \ldots \xrightarrow{hb} \alpha_0 \). For \( LS_{i+j}(q) \neq \emptyset \) to be true, it must hold that \( LS_j(q) \cap LH_j(t_j) \neq \emptyset \). Let \( LH_j = LS_j(q) \cap LH_j(t_j) \).

If \( t_i = t_j \) then it follows that \( i \xrightarrow{hb} j \) from the definition of \( \xrightarrow{hb} \). Note that in this case \( LH_j \neq \emptyset \) because \( LH_j(t_i) = LH_j(t_j) \) 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 \( LH_j \):

**Case 1.** \( \ell \) is allocated in \( \sigma \) and \( \ell \in LH_j \). Then there exists \( \alpha_k = rel(\ell) \) and \( \alpha_j = acq(\ell) \) \((i < k < l \leq j)\) such that \( b_k(t_k(x)) = b_k(t_k(y)) = \ell, t_k = t_l \) 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 \( \ell \) is mutually-exclusive, there must be some \( rel(\ell) \) by \( t_i \) (at state \( s_j \)) and \( acq(\ell) \) by \( t_j \) (at state \( s_j \)) on \( \ell \) where \( b_k(t_k(x)) = b_k(t_k(y)) = \ell \). Then because \( t_k = t_j \) \( i \xrightarrow{hb} k \) and because \( t_j = t_l \xrightarrow{hb} j \). Since both \( rel(\ell) \) and \( acq(\ell) \) on \( \ell \) \( k \xrightarrow{hb} l \) is transitively-closure if \( i \xrightarrow{hb} k \xrightarrow{hb} l \xrightarrow{hb} j \).

**Case 2.** \( t_j \in LH_i \). Then there exists \( \alpha_k = (x = fork(x)) \) \((i < k < j)\) such that \( b_k(t_k(x)) = t_j \). In this case \( i \xrightarrow{hb} k \xrightarrow{hb} j \).

**Proof.** First, if \( t_k = t_j \), 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 \( t_k \neq t_j \).

Since there is no \( t \in LH(t_j) \) such that \( t \neq t_i \), there must be some action \( \alpha_k = (x = fork(x)) \) \( (b_k+1(t_k)(x) = t_j) \) that adds \( t \) to \( LS_{k+1}(q) \). The update rule for \( for_k \) requires that \( LS_k(q) \cap LH_k(t_k) \neq \emptyset \). Let \( LH_k = LS_k(q) \cap LH_k(t_k) \). Case 1 or 2 applies recursively. Let Case 1 apply; then there is some lock \( \ell \in LH_i \) and proof of Case 1 applies. Now let Case 2 apply recursively; it yields an ac-
tion \( \alpha_t = (x = fork(x)) \) \((\delta_{t+1}(t)(x) = t_x)\) such that \(t = t_i\). It gives 
\[ \{ \begin{array}{l}
0^{k} \mid \{ \begin{array}{l}
1 \mid \{ \begin{array}{l}
\delta_{t+1}(t)(x) = t_x
\end{array}\}
\end{array}\}
\end{array}\]
\]

**Lemma 4.** Let \( \sigma = (\delta_1,h_1) \ldots \delta_n,h_n\) be an execution and \((a,f)\) a heap variable such that \(h_n(a,f) \neq \perp\). If there is a happens before \( \overrightarrow{hb} \) between any two accesses \(\alpha_i\) and \(\alpha_j\) \((1 \leq i < j \leq n)\) to \((a,f)\), then applying the algorithm above to \(\sigma\) ends with \(LS(a,f) \neq \emptyset\).

**Proof.** Let \(q\) be a variable such that \(q \rightarrow a.f\). Let \(s_i\) denote the state \((\delta_i,h_i)\) 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 \(q\) for any prefix \(\sigma' = s_1 \ldots s_m\) \((m \leq n)\) of \(\sigma\), if there is a happens before \( \overrightarrow{hb} \) between any two accesses \(\alpha_i\) and \(\alpha_j\) \((1 \leq i < j \leq m)\) to \((a,f)\), then applying the algorithm above to \(\sigma'\) ends with \(LS_m(a,f) \neq \emptyset\).

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

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

Let \(\alpha_i\) and \(\alpha_j\) \((1 \leq i < j \leq m)\) be two accesses to \(q\). Assume that \(LS_{t+1}(q) \neq \emptyset\) and there is some happens-before relation such that \(i \overrightarrow{hb} \ldots \overrightarrow{hb} j\). Now we will prove that if \(i \overrightarrow{hb} j\) then \(LS_{t+1}(q) \neq \emptyset\). Because \(LS_{t+1}(q) \neq \emptyset\), it must hold that \(LS_{t}(q) \cap LH_{t+1}(t) \neq \emptyset\). Let \(LH_{t}^{j} = LS_{t}(q) \cap LH_{t}(t_{j})\). We will split the proof for different cases that yield \( \overrightarrow{hb}\):

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

**Proof.** Because \(LS_{t}(q) \cap LH_{t}(t_{j}) \neq \emptyset\) and \(LH_{t}(t_{i}) = LH_{t}(t_{j})\), it follows that \(LH_{t}^{j} \neq \emptyset\) which causes the assignment that makes \(LS_{t+1}(q) = LH_{t+1}(t_{j}) \neq \emptyset\).

**Case 2.** There are some \(\alpha_k = rel(x)\) and \(\alpha_t = acq(y)\) \((i < k < l < j)\) on \(\ell\) where \(t_k = t_i, t_l = t_j\), and \(\delta_{t_k}(t_k)(x) = \delta_{t_l}(t_l)(y) = \ell\).

**Proof.** Because \(LS_{t+1}(q) = LH_{t}(t_{j})\) it holds that \(\ell \in LS_{t+1}(q)\). Because \(LS_{t}(q)\) remains the same or increases until \(t_j, \ell \in LS_{t}(q)\). Then according to the update rule for \(acq\), because \(LH_{t}^{j} \neq \emptyset\) that yields \(\ell \in LS_{t+1}(q)\) and \(t_j \in LS_{t+1}(q)\).

**Case 3.** There are some \(\alpha_t = (x = fork(x)) \) \((i < k < j)\) where \(t_k = t_i\) and \(\delta_{t_k}(t_k)(x) = t_x\).

**Proof.** Because \(LS_{t+1}(q) = LH_{t}(t_{j})\) it holds that \(t_j \in LS_{t+1}(q)\). Because \(LS_{t}(q)\) remains the same or increases until \(t_j, \ell \in LS_{t}(q)\). Then according to the update rule for \(fork\), because \(LH_{t}^{j} \neq \emptyset\) that yields \(\ell \in LS_{t+1}(q)\) and \(t_j \in LS_{t+1}(q)\).

**Theorem 2.** Let \( \sigma = (\delta_1,h_1) \ldots \delta_n,h_n\) be an execution and \((a,f)\) a heap variable such that \(h_n(a,f) \neq \perp\). Then \(\sigma\) is race-free for \((a,f)\) if applying the algorithm does not report any race condition, meaning that \(LS(a,f)\) never gets empty during \(\sigma\) and \( \sigma\) ends with \(LS(a,f) \neq \emptyset\).

**Proof.** From Lemma 1 and Lemma 2, it follows that (1) if \(\sigma\) 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(Addr∪Tid)) LS; |
| (Variable \rightarrow Node) la; |
| (Tid∪\{0\}) tid; |
| Powerset(Addr∪Tid) done; |

**Search() {**

| Node curr = new Node; |
| curr.state = (δ, h); |
| curr.LS = λq \in Variable. Addr∪Tid; |
| curr.la = λq \in Variable. null; |
| curr.tid = 0; |
| curr.done = 0; |

**Stack(Node) stack = new Stack(Node);**

**while (!stackIsEmpty) {**

| Tid t; |
| curr = stack.Peek(); |
| if (curr.tid = 0 ∧ curr.la \in enabled(curr.state)) |
| t = choose(enabled(curr.state) \{done\); |
| else if (curr.tid = 0 ∧ curr.tid \in done) |
| t = curr.tid; |
| else |
| stack.Pop(); |
| continue; |

| curr.done = curr.done∪\{t\}; |
| stack.Push(Successor(curr.t)); |

**}**
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 Powerset(Adr ∪ Tid) and LSR is a partial function from Variable × Tid to Powerset(Adr ∪ Tid). Initially, both are not defined for any variable and thread.

Suppose that thread t executes an action α to go from state (s, h) to (s′, h′). Let ls(t) = (pc, l) and ls′(t) = (pc′, l′). We update the partial maps LH, LSR and LSW by doing a case analysis on α as follows:

1. x = new: Let q = (l′(x), f). For all f ∈ Field, initialize both LSR(q, t) and LSW(q) to Addr ∪ Tid.
2. y = f: Let q = (l(y), f).
   If ((LSW(q) ∩ LH(t) ∩ LSR(q, t) ≠ 0))
   Then LSR(q, t) = LSW(q) ∩ LH(t) ∩ LSR(q, t)
   Else LSW(q) = 0 ∀t′ LSR(q, t′) = 0
3. x = y: Let q = (l(y), f).
   If ((LSW(q) ∩ LH(t) ≠ 0) and ∀t′ ≠ t. (LSW(q) ∩ LH(t) ∩ LSR(q, t′) ≠ 0))
   Then LSW(q) = LH(t)
   Else LSW(q) = 0 ∀t′ LSR(q, t′) = LSW(q)
4. x = acq(x): LH(t) = {l′(x)} ∪ LH(t). For all q ∈ Variable such that LSR(q) ≠ ⊥,
   If ((LSW(q) ∩ LH(t) ≠ 0) and ∀t′ ≠ t. (LSW(q) ∩ LH(t) ∩ LSR(q, t′) ≠ 0))
   Then LSW(q) = LSW(q) ∪ LH(t)
   ∀t′ LSR(q, t′) = LSW(q)
5. x = rel(x): LH(t) = LH(t) \ {l′(x)}.
6. x = fork: For all q ∈ Variable such that LSR(q) ≠ ⊥,
   If ((LSW(q) ∩ LH(t) ≠ 0) and ∀t′ ≠ t. (LSW(q) ∩ LH(t) ∩ LSR(q, t′) ≠ 0))
   Then LSW(q) = LSW(q) ∪ {l′(x)}
   ∀t′ LSR(q, t′) = LSR(q, t′) ∪ {l′(x)}
7. x = join: For all q ∈ Variable such that LSR(q) ≠ ⊥,
   If ((LSW(q) ∩ LH(l(x)) ≠ 0) and ∀t′ ≠ t. (LSW(q) ∩ LH(l(x)) ∩ LSR(q, t′) ≠ 0))
   Then LSW(q) = LSW(q) ∪ {l′(x)}
   ∀t′ LSR(q, t′) = LSR(q, t′) ∪ {l′(x)}
8. x = op(y1, ..., yn): No update on locksets.

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-