Predicate Detection for Parallel Computations with Locking Constraints

Yen-Jung Chang and Vijay K. Garg

Parallel and Distributed Systems Laboratory
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712-1084, USA.
cyenjung@utexas.edu and garg@ece.utexas.edu

Abstract. The happened-before model (or the poset model) has been widely used for modeling the computations (execution traces) of parallel programs and detecting predicates (user-specified conditions). This model captures potential causality as well as locking constraints among the executed events of computations using Lamport's happened-before relation. The detection of a predicate in a computation is performed by checking if the predicate could become true in any reachable global state of the computation. In this paper, we argue that locking constraints are fundamentally different from potential causality. Hence, a poset is not an appropriate model for debugging purposes when the computations contain locking constraints. We present a model called Locking Poset, or a Loset, that generalizes the poset model for locking constraints. Just as a poset captures possibly an exponential number of total orders, a loset captures possibly an exponential number of posets. Therefore, detecting a predicate in a loset is equivalent to detecting the predicate in all corresponding posets. Since determining if a global state is reachable in a computation is a fundamental problem for detecting predicates, this paper first studies the reachability problem in the loset model. We show that the problem is NP-complete. Afterwards, we introduce a subset of reachable global states called lock-free feasible global states such that we can check whether a global state is lock-free feasible in polynomial time. Moreover, we show that lock-free feasible global states can act as "reset" points for reachability and be used to drastically reduce the time for determining the reachability of other global states. We also introduce strongly feasible global states that contain all reachable global states and show that the strong feasibility of a global state can be checked in polynomial time. We show that strong feasibility provides an effective approximation of reachability for many practical applications.

This is a regular paper.

The paper is eligible for the best student paper award. (Student: Yen-Jung Chang)

1 Introduction

One of the fundamental problems in debugging or runtime verification of a parallel program is to check if a *predicate* (user-specified condition) could become true in any global state that can be reached by the program. This problem is challenging because different runs of the program may reach different sets of global states due to the nondeterministic thread scheduling even for the same user input. In this paper, we propose a new model of parallel computation that captures the reachable global states on multiple thread schedules and thus enables efficient predicate detection.

As an example of predicate detection, suppose that the condition Φ: *file f is opened by two threads at the same time*, is a potential bug of the parallel program shown in Fig. 1. We would like to know if the program can possibly reach a global state where Φ is true, i.e., to detect *possibly* Φ. One popular debugging method is to run the program and collect a totally ordered sequence of events. Suppose that the sequence recorded is $\sigma = a1, a2, a3, a4, b1, b2, b3, b4$. In this total order, Φ does not become true. However, the predicate is indeed possible if the sequence of events starts with the prefix $(a1, a2, a3, b1, b2)$. Hence, the only way to detect possibly Φ is to perform multiple executions and hope that one of them produces a total order that makes the predicate true [24,30].

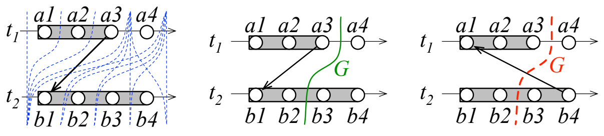
To alleviate this issue, the computation (the execution trace) of a parallel program is usually modeled as a partially ordered set (poset) of events, ordered by Lamport's happened-before relation (denoted by $\rightarrow$) [20]. In this poset, the events that are executed by a single thread are totally ordered and the events across threads are ordered based on their causality. Usually, the synchronization due to locks is also modeled with the happened-before relation. Specifically, the release of a lock is ordered before its subsequent acquisition [2, 4, 9, 21].

By modeling the computation as a poset, we are able to *predictively* detect the predicate if it becomes true in any *consistent global state* of the poset. In the poset model, a global state G is consistent iff for events $e, f{:}(e{\rightarrow}f) \wedge (f{\in}G){\Rightarrow}(e{\in}G)$. Moreover, consistent global states are considered reachable because for every consistent global state there always exists at least one sequence of events that leads the program to reach that global state [1]. Hence, detecting a predicate on one poset is equivalent to detecting the predicate on multiple sequences of events. In addition, if we do not know the nature of the predicate, then predicate detection is usually done by enumerating all consistent global states of the poset and checking if any one of them satisfies the predicate [2,4,5,17].

For the program in Fig. 1, the executions that produce σ and any total order with the prefix $(a1, a2, a3, b1, b2)$ are modeled as the same poset shown in Fig. 2(a), in which the dashed lines are consistent global states of the poset and each of which contains all the events on its left. Fig. 2(b) shows the only global state G where the predicate Φ becomes true. Since G is consistent, Φ would be successfully detected when G is enumerated. However, we still have not solved the problem of predicate detection for all thread schedules. Suppose that thread t_2 obtains the lock before t_1 during the execution. Then, we put a happened-before order from $b4$ to $a1$ instead of $a3$ to $b1$ as shown in Fig. 2(c). In this poset, G is inconsistent and will not be enumerated. Consequently, a purely poset based predicate detection algorithm will miss the predicate under a different locking schedule.

```
Thread t1                    Thread t2
-----------                  -----------
a1: acquireLock(l)           b1: acquireLock(l)
a2:  f.openFile()            b2:  f.openFile()
a3: releaseLock(l)           b3:  f.closeFile()
a4: f.closeFile()            b4: releaseLock(l)
```

Fig. 1. A program which has two threads that might open the file f at the same time.



(a) (b) (c)

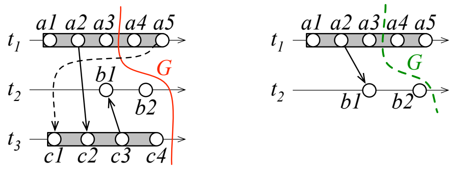
Fig. 2. (a) The dashed lines are consistent global states. (b) Φ only becomes true in the global state G and it can be correctly detected because G is consistent. (c) In this poset, G is inconsistent and thus Φ cannot be detected.

```
Thread t1
-----------
a1: acquireLock(l)
a2:  l.notify()
a3:  f.openFile()
a4:  f.closeFile()
a5: releaseLock(l)
```

```
Thread t2
-----------
b1: recMsg(t3,&m)
b2: f.openFile()

Thread t3
-----------
c1: acquireLock(l)
c2:  l.waitUntilNotified()
c3:  sendMsg(t2,m)
c4: releaseLock(l)
```

Fig. 3. A program which has three threads but the file f can only be opened by one thread at a time.



(a) (b)

Fig. 4. (a) The global state G, where Φ is true, is indeed unreachable because of the implicit order (the dashed arrow) between the two critical sections. (b) The local view that contains only two of the threads, where G is mistakenly considered reachable.

An alternative approach removes the happened-before (HB) relation due to lock synchronization and determines the reachability of a global state using the techniques of lockset and acquisition history instead of the HB consistency of the global state [18,19,25,27,28]. However, these techniques only consider predicates that involve two threads, i.e., data races and atomicity violations. If the computation contains more than two threads, the detection is performed on a local view that consists of only two threads at a time. Hence, they can induce false positives because of the lack of the global view. Consider the program in Fig. 3, which has three threads. Because of the conditional synchronization (e.g., Java's `notify()` and `wait()`) between events $a2$ and $c2$, thread t_1 will obtain the lock l before t_3; otherwise, t_3 will be forced to release the lock. Thus, we get a computation as shown in Fig. 4(a). Although the order $a5 \rightarrow c1$ is not explicitly captured in the computation, it is always implicitly induced during the execution of the program. Thus, the global state G, where Φ is true, is indeed unreachable. If we try to detect Φ in a local view that contains only two threads (see Fig. 4(b)), then G could be mistakenly considered reachable and result in a false-positive.

To deal with the co-existence of locks and the happened-before (HB) relation, one commonly used method is to convert mutual exclusion constraints and the HB relation to the constraints for SAT/SMT solvers [16, 32, 33]. When a global state that satisfies Φ is found, the solver is invoked in order to determine whether that global state is reachable in the computation. If it is reachable, then possibly Φ is detected. Although this method is applicable for detecting predicate that involve the global view of the system, these solvers may take exponential time in the worst case.

Since determining the reachability of a global state is a fundamental problem for the technique of predicate detection, our focus in this paper is on methods that take polynomial time for determining the reachability. We first introduce a model, named *Loset* (**Lo**cking po**set**), which is a generalization of the poset model. A Loset is a Poset augmented with the notion of locking intervals. In a loset, a lock synchronization is not modeled using the HB relation. Instead, the intervals of events that are executed under one or more locks are modeled separately. If two intervals $I1$ and $I2$ are executed under the same lock, then it is understood that events in $I1$ and $I2$ cannot be interleaved but can happen in either order. Since there can be an exponential number of different locking schedules, a loset in effect would model an exponential number of posets. Thus, a loset allows us to detect possibility of violation of invariants which would not be possible to detect using a single conventional poset. Moreover, our technique does not depend on the nature of the predicate. Thus, it can be used for detecting the predicate whose nature is unknown and requires the global view of the system.

Afterwards, we study the complexity of reachability problem in a loset: Given a loset $\mathcal{L}$ and a global state G, the reachability problem asks if there exists a sequence of events that leads the program to reach G in $\mathcal{L}$. The reachability problem is trivial for a poset: G is reachable iff G is consistent [1]. However, we show that the reachability problem for a loset is NP-complete. Our proof uses the NP-completeness of the predicate control problem shown in [29].

To cope with the NP-completeness, we introduce a subset of reachable global states called *lock-free feasible global states* such that we can efficiently check whether a global state is lock-free feasible in polynomial time. We show that the set of reachable lock-free feasible global states forms a finite distributive lattice under the usual less than relation $<$ of global states. With the property of distributive lattice, we show that the reachability of a global state G can be determined using only a subset $(F \backslash G)$ of events, where F is the greatest lock-free feasible global state such that $F \leq G$. Thus, lock-free feasible states act as "reset" points for reachability and can be used to drastically reduce the time for checking reachability, by limiting the calculation in a subcomputation rather than the entire computation.

We also introduce *strongly feasible global states* that contain all reachable global states such that checking whether a global state is strongly feasible for a loset can be done efficiently. For many practical applications, strongly feasible global states provide an effective approximation of reachability. In Appendix E, we show that for computations with two threads, the set of strongly feasible global states is identical to the set of reachable global states. In Appendix F, our experiments show that the gap between strong feasibility and reachability seldom exists in practice. We give a method to enumerate the strongly feasible global states of a loset. In comparison with two naive but accurate enumeration algorithms, which enumerate only reachable global states, our enumeration method shows that the strongly feasible property accurately models the reachable global states for all 11 benchmark programs while using only 15–40% of their runtime.

We note here that our techniques are orthogonal to the techniques using SAT/SMT solvers. Given a trace of a computation, instead of calculating the reachability of a global state G from the initial global state, we only need to compute if G is reachable from the greatest lock-free feasible global state that precedes G. Moreover, we only need to calculate the reachability with a SAT/SMT solver if G is strongly feasible.

The rest of the paper is organized as follows. Section 2 presents the loset model. Section 3 and 4 introduce the sets of lock-free feasible and strongly feasible global states. Section 5 discusses the reachability of various classes of global states in a loset. Finally, section 6 concludes this paper.

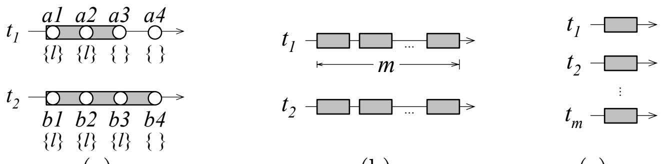
2 Loset Model of a Computation

A computation (i.e., an execution trace of a parallel program) is modeled as a *Loset* (**Lo**cking Po**set**) of events as defined next.

Definition 1 (Loset). *A loset $\mathcal{L}$ is a five-tuple $\mathcal{L} = (E, \rightarrow, n, L, \mathcal{I})$ where:*

- *E: is a set of events,*
- *$\rightarrow$: is an irreflexive transitive binary relation on E,*
- *n: is the number of threads,*
- *L: is the number of locks,*
- *$\mathcal{I}$: is a set of locking intervals.*

The $\rightarrow$ relation represents the potential causality between events, i.e., $e \rightarrow f$ means that the event e may directly or transitively cause the event f. For distributed systems, it corresponds to the Lamport's happened-before (HB) relation [20]. In concurrent systems, we may have additional order constraints due to the *fork-join* events of threads and the *wait-notification* events of conditional synchronization [2, 4, 9, 21]. In this paper, we maintain the $\rightarrow$ relation using vector clocks [8, 22]. The set E of events is partitioned into n sequences $E_1, E_2, \cdots, E_n$ such that each E_i represents a thread. For all distinct events $e, f \in E_i : (e \rightarrow f) \vee (f \rightarrow e)$. For convenience, we define the process order relation (denoted by $\prec$) such that $e \prec f$ means $e \rightarrow f$ in some E_i. A locking interval $I \in \mathcal{I}$ is a four-tuple $I = (t, l, acq, rel)$ such that $t \in \{1..n\}, l \in \{1..L\}$, $(acq, rel \in E_t)$, and $acq \prec rel$. An interval indicates that the thread $I.t$ acquired the lock $I.l$ at event $I.acq$ and released it at $I.rel$.



(a) (b) (c)

Fig. 5. (a) The loset that is equivalent to the two posets in Fig. 2(b) and 2(c). The gray boxes are the critical sections created by the same lock. (b) A loset that is equivalent to C_m^{2m} posets. (c) A loset that is equivalent to $m!$ posets.

Note that the objective of the $\rightarrow$ relation is to capture the causality of events but not the real-time locking order between the acquisition and release events of locks. Therefore, the locking intervals for the same lock are totally ordered in a poset but not in a loset. Formally,

Definition 2 (Valid Poset of a Loset). *A poset $P = (E, \rightarrow_P)$ is a* valid poset *of a loset $\mathcal{L} = (E, \rightarrow, n, L, \mathcal{I})$ if $(\rightarrow \subseteq \rightarrow_P)$ and $\forall I, J \in \mathcal{I}$ such that $I.l = J.l$, we have $(I.rel \rightarrow_P J.acq) \vee (J.rel \rightarrow_P I.acq)$.*

For instance, the two posets in Fig. 2(b) and Fig. 2(c) are the valid posets of the loset in Fig. 5(a). In Fig. 5(b), suppose that each thread contains m locking intervals for the same lock, then the loset is equivalent to C_m^{2m} valid posets because the m intervals of t_1 can be interleaved with those of t_2 in C_m^{2m} total orders. Similarly, the loset in Fig. 5(c) is equivalent to $m!$ valid posets. Fig. 7 shows a more complex loset. We now define global states and their reachability under the loset model.

2.1 Global States

A **global state** G is a subset of E such that $\forall e, f \in E : (f \in G) \wedge (e \prec f) \Rightarrow (e \in G)$. In Fig. 5(a), the set $\{a1, a2, b1\}$ is a global state, but $\{a2, b1\}$ is not a global state because it contains event $a2$ but not $a1$ even though $a1 \prec a2$. A global state G can equivalently be identified by the number of events of each E_i in G. For example, the global state $\{a1, a2, b1\}$ is represented by the array $[2, 1]$. The symbol $G[i]$ denotes the maximal (latest) event of E_i in the global state G. The order $G \leq H$ between the two global states means $G[i] \preceq H[i]$ holds for any thread i.

A global state G is **consistent** iff $\forall e, f \in E : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G)$. A consistent global state preserves the $\rightarrow$ relation of the loset. Note that the initial global state ($G = \phi$), and the final global state ($G = E$) are always consistent. We define the set $\textsc{el}(e)$ of **effective locks** for any event e, which are the locks being held by the thread that has executed e:

$$\textsc{el}(e) = \{I.l \mid I.acq \preceq e \prec I.rel\}.$$

In Fig. 5(a), the effective locks of the events in the computation are shown in curly brackets. We can now define the set of global states that respect the locking constraints. A global state G is (lock) **compatible** iff for any $i \neq j, G[i]$ and $G[j]$ are pairwise (lock) compatible, i.e., $\textsc{el}(G[i]) \cap \textsc{el}(G[j]) = \varnothing$. Finally, a global state is **feasible** iff it is consistent and compatible.

If a global state is not feasible then it violates either the consistency constraints or the locking constraints. Hence, only feasible global states are reachable from the initial global state. However, not all feasible global states are reachable. For example, the global state G in Fig. 4(a) is feasible but not reachable because of the implicit locking order induced by the conditional synchronization.

2.2 Reachable Global States and Runs

We first introduce a sequence of events called a run, $\mathcal{R}$, in which the total order between events is denoted by $\prec_{\mathcal{R}}$. The symbol $\delta(G, \mathcal{R})$ denotes the global state that is reached by executing the sequence $\mathcal{R}$ of events starting from the global state G. The symbol $\mathcal{R}^i$ denotes the prefix of $\mathcal{R}$ of length i. Since only feasible states are reachable, a *run* goes through only feasible global states.

Formally, a sequence $\mathcal{R}$ of events is a **run** starting from G iff the global state $\delta(G, \mathcal{R}^i)$ is feasible for any i such that $0 \leq i \leq |\mathcal{R}|$. A global state G is **reachable** from the initial global state ϕ iff there exists a run $\mathcal{R}$ such that $\delta(\phi, \mathcal{R}) = G$. The reachability problem is defined as:

Definition 3 (Loset Reachability Problem). *Given a loset $\mathcal{L}$ and a global state G, is G a reachable global state of $\mathcal{L}$?*

Theorem 1. *The loset reachability problem is NP-complete.*

Proof. (Outline) The details are shown in Appendix A. In [29], the predicate control problem asks if there exists a control sequence, which is a total order among the critical sections for the same lock, such that the predicate Φ remains true after the control sequence is added to the computation. It was shown that the predicate control problem is NP-complete. The model defined in [29] is a special case of our loset model, where locking intervals do not overlap. It can be shown that there exists a control sequence that reaches the global state G without violating mutual exclusion iff the global state G is reachable in the loset. Therefore, the predicate control problem is a special case of the reachability problem of a loset. □

In the following sections, we present two classes of global states — lock-free feasible global states and strongly feasible global states. A lock-free feasible global state is always reachable and a reachable global state is always strongly feasible. Thus, these two classes provide a lower and an upper bound on the set of reachable global states. Both of these classes can be checked efficiently (in polynomial time), whereas the reachability problem is NP-complete. Moreover, to check reachability of a global state G, it is sufficient to check its reachability from the greatest lock-free feasible global state that precedes G instead of checking it from the initial global state of the computation.

3 Lock-Free Feasible Global States

A *lock-free feasible global state* is a feasible global state that holds no lock. We show that given a reachable global state G of a loset, then any lock-free feasible global state $F \leq G$ is also reachable.

Theorem 2. *Given a reachable global state G of a loset and a lock-free feasible global state $F \leq G$, there exists a run that reaches both F and G.*

Proof. Since G is reachable, there exists a run $\mathcal{R}$ such that $\delta(\phi,\ \mathcal{R}) = G$. Let the sequence $\mathcal{S}_1$ of events be $\mathcal{R} \uparrow F$, which is the projection of $\mathcal{R}$ that contains only the events in F, and let $\mathcal{S}_2 = \mathcal{R} \uparrow (G \backslash F)$. Let $\mathcal{S} = \mathcal{S}_1 \oplus \mathcal{S}_2$ ($\mathcal{S}_1$ concatenated with $\mathcal{S}_2$). We show that the sequence $\mathcal{S}$ of events is also a run, i.e., $\delta(\phi, \mathcal{S}^i)$ is feasible for any $\mathcal{S}^i$, which implies $\delta(\phi, \mathcal{S}_1) = F$ and $\delta(F, \mathcal{S}_2) = G$.

Claim 1. $\boldsymbol{\forall i : 0 \leq i \leq |\mathcal{S}| : \delta(\phi, \mathcal{S}^i)}$ **is consistent**:
We show the partial order $\rightarrow$ of the computation is preserved by the total order $\prec_{\mathcal{S}}$ in $\mathcal{S}$. For any two events, e and f, in $\mathcal{S}$ such that $e \prec_{\mathcal{S}} f$, we have

Case 1. $(e, f \in \mathcal{S}_1) \vee (e, f \in \mathcal{S}_2)$: The $\rightarrow$ relation between e and f is preserved in $\prec_{\mathcal{R}}$ because $\mathcal{R}$ is a run. Since $\mathcal{S}_1$ and $\mathcal{S}_2$ are projections of $\mathcal{R}$, the $\rightarrow$ relation is preserved in $\prec_{\mathcal{S}_1}$ and $\prec_{\mathcal{S}_2}$.

Case 2. $e \in \mathcal{S}_1, f \in \mathcal{S}_2$: If $e \rightarrow f$, the $\rightarrow$ relation is preserved by the concatenation $\mathcal{S}_1 \oplus \mathcal{S}_2$. The case $f \rightarrow e$ is not possible because F is consistent and $e \in F$ but $f \notin F$.

Claim 2. $\boldsymbol{\forall i : 0 \leq i \leq |\mathcal{S}_1| : \delta(\phi, \mathcal{S}_1^i)}$ **is compatible**:
Let the global state $V = \delta(\phi,\ \mathcal{S}_1^i)$. We show that

$$\forall s \neq t : \textsc{el}(V[s]) \cap \textsc{el}(V[t]) = \varnothing. \tag{1}$$

Let $\mathcal{R}^j$ be the shortest prefix of $\mathcal{R}$ such that $\mathcal{R}^j \uparrow F = \mathcal{S}_1^i$ and let $W = \delta(\phi,\ \mathcal{R}^j)$. Then, the following condition holds because $\mathcal{R}$ is a run:

$$\forall s \neq t : \text{EL}(W[s]) \cap \text{EL}(W[t]) = \varnothing. \tag{2}$$

Since $\mathcal{S}_1^i$ contains the same or fewer events than $\mathcal{R}^j$, we get $V \subseteq W$, which implies $V[t] \preceq W[t]$ for any thread t. We now consider the following two cases:

Case 1. $V[t] \prec W[t]$: Because $\mathcal{S}_1^i = \mathcal{R}^j \uparrow F$, this case holds only if $\mathcal{R}^j$ contains the events in $G \backslash F$ w.r.t. E_t, which implies that $\mathcal{S}_1^i$ contains all the events in F w.r.t. E_t. Thus, we get $V[t] = F[t] \prec W[t]$. Since F is lock-free, we get $\text{EL}(V[t]) = \varnothing \subseteq \text{EL}(W[t])$.

Case 2. $V[t] = W[t]$: In this case, we get $\text{EL}(V[t]) = \text{EL}(W[t])$.

From cases 1 and 2, $\text{EL}(V[t]) \subseteq \text{EL}(W[t])$ holds for any thread t. Then, from (2), (1) holds.

Claim 3. $\forall i : 0 \leq i \leq |\mathcal{S}_2| : \delta(F, \mathcal{S}_2^i)$ is compatible:
Let the global state $V = \delta(F, \mathcal{S}_2^i)$. We show that

$$\forall s \neq t : \text{EL}(V[s]) \cap \text{EL}(V[t]) = \varnothing. \tag{3}$$

Let $\mathcal{R}^j$ be the shortest prefix of $\mathcal{R}$ such that $\mathcal{R}^j \uparrow (G \backslash F) = \mathcal{S}_2^i$ and $W = \delta(\phi, \mathcal{R}^j)$. Then, the following condition holds because $\mathcal{R}$ is a run:

$$\forall s \neq t : \text{EL}(W[s]) \cap \text{EL}(W[t]) = \varnothing. \tag{4}$$

Since V initially contains all the events in F and $\mathcal{S}_2^i$ contains the same events in $G \backslash F$ as $\mathcal{R}^j$, we get $W \subseteq V$, which implies that $W[t] \preceq V[t]$ holds for any thread t:

Case 1. $W[t] \prec V[t]$: Because $\mathcal{S}_2^i = \mathcal{R}^j \uparrow G \backslash F$, this case holds only if $\mathcal{R}^j$ contains only the events in F w.r.t. E_t, which implies that $\mathcal{S}_2^i$ does not contain any event of E_t. Thus, we get $W[t] \prec V[t] = F[t]$. Since F is lock-free, we get $\text{EL}(W[t]) \supseteq \text{EL}(V[t]) = \varnothing$.

Case 2. $W[t] = V[t]$: We get $\text{EL}(W[t]) = \text{EL}(V[t])$.

From the two cases, $\text{EL}(W[t]) \supseteq \text{EL}(V[t])$ holds for any thread t. Then, from (4), (3) holds.

From claims 1, 2, and 3, $\mathcal{S}$ is a run that reaches first F using the run $\mathcal{S}_1$ and then reaches G using the run $\mathcal{S}_2$. □

Since we use the loset model for analyzing the behavior of parallel programs, we are interested only in those losets that capture a possible execution from a real-world application, i.e., the reachability of the final global state of the computation is given by the execution of the program. Formally, a loset is **valid** iff its final global state E is reachable. An example of a loset, which is an artificial computation, that is not valid is shown in Appendix B. A simple consequence of Theorem 2 is that whenever $\mathcal{L}$ is a valid loset, then every lock-free feasible global state of $\mathcal{L}$ is reachable.

Corollary 1. *All lock-free feasible global states of a valid loset are reachable.*

Proof. The final global state of a valid loset is reachable. Therefore, from Theorem 2, we get that every lock-free feasible global state of that loset is reachable. □

The set of *reachable* lock-free feasible global states also satisfies the following nice property for all losets: (and not just valid losets).

Theorem 3. *The set of reachable lock-free feasible global states of a loset $\mathcal{L}$ forms a distributive lattice.*

Proof. (Outline) The details are shown in Appendix C. For any two reachable lock-free feasible global states, G and H, let $M = (G \cap H)$ be their meet and $J = (G \cup H)$ be their join. We first show that M and J are lock-free and feasible. Then, from Theorem 2, M is reachable because $M \leq G$. To show their join J is reachable, we construct a sequence $\mathcal{S}_J$ of events such that $\mathcal{S}_J = \mathcal{R}_G \oplus \mathcal{R}_{MH}$, where R_G is a run reaches G and $\mathcal{R}_{MH}$ reaches H from M. Then, we show that $\mathcal{S}_J$ is also a run. □

Theorem 3 has two important implications. First, since the set of reachable lock-free feasible global states forms a distributive lattice, we can concisely represent all lock-free feasible global states of a valid loset using the set of *join-irreducible* elements of the distributive lattice [6] and use slicing to study various sublattices, which reduces the time complexity of predicate detection to polynomial time for certain classes of predicates [12,23]. Secondly, as shown next, we can reduce the search space to determine reachability of a feasible global state that is not lock-free. Given a global

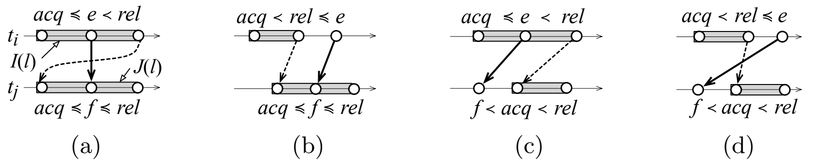

$acq \preccurlyeq e < rel$
t_i
$I(l)$
$J(l)$
t_j
$acq \preccurlyeq f \preccurlyeq rel$
(a)
$acq < rel \preccurlyeq e$
$acq \preccurlyeq f \preccurlyeq rel$
(b)
$acq \preccurlyeq e < rel$
$f < acq < rel$
(c)
$acq < rel \preccurlyeq e$
$f < acq < rel$
(d)

Fig. 6. All possible cases of $I(l) \mapsto J(l)$ and the locking order $I(l).rel \rightarrow_L J(l).acq$ (shown in dashed lines).

state G, we first find the greatest lock-free feasible global state $F \leq G$. On account of Theorem 3, F is well-defined whenever there exists any lock-free feasible global state that precedes G. Given G and F, the following theorem shows that the search for the reachability in a valid loset can be restricted to the events in $G \backslash F$.

Theorem 4. *Given a global state G of a valid loset and the greatest lock-free feasible global state F such that $F \leq G$, the reachability of G can be determined by the events $G \backslash F$.*

Proof. From Theorem 2, F is reachable because the final global state E is reachable. Moreover, the run that reaches E of $\mathcal{L}$ can be reordered so that it first reaches F and then E. We consider the following two cases: (1) If G is reachable, then from Theorem 2 there exists a run $\mathcal{R} = \mathcal{R}_1 \oplus \mathcal{R}_2$, where $\mathcal{R}_1$ is a run that reaches F and $\mathcal{R}_2$ is a run that reaches G from F. (2) If G is unreachable, then there exists no run from F to G because F is reachable and lock-free. Hence, the existence of the run $\mathcal{R}_2$ depends only on the events $G \backslash F$. □

4 Strongly Feasible Global States

In this section, we give an upper-approximation of reachability. We define the notion of *strong feasibility* based on the inferred causality due to the HB relation and locking constraints. Therefore, a reachable global state is always strongly feasible. Also, just as feasibility and lock-freedom can be evaluated in polynomial time, strong feasibility can be evaluated in polynomial time.

4.1 Locking Order

Even though real-time locking order is not modeled in a loset, some order between locks may be implied due to the HB orders between events and locking constraints (i.e., the events in different locking intervals of the same lock cannot be interleaved during the execution of the program). We next introduce the relation $\mapsto$ for capturing such implied ordering constraints.

The $\mapsto$ relation is defined between locking intervals of the same lock such that $I \mapsto J$ means the locking interval I has to start before J can finish:

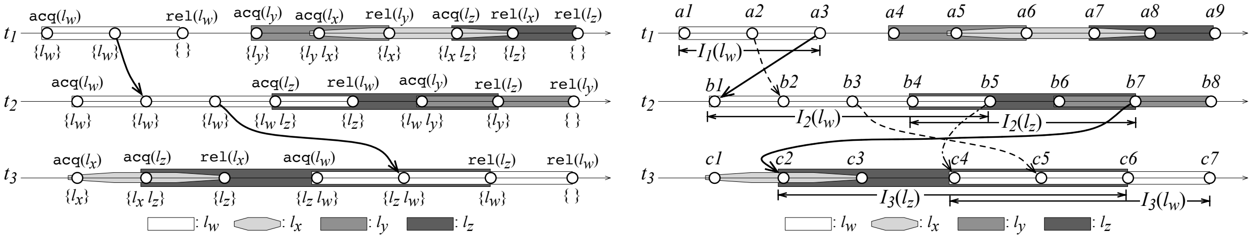
Definition 4 (Relation $\mapsto$)**.** *Let $I(l)$ and $J(l)$ be the locking intervals of the same lock l. $I(l) \mapsto J(l)$ iff there exist events e and f such that $(I(l).acq \preceq e) \wedge (e \rightarrow f) \wedge (f \preceq J(l).rel)$.*

Fig. 6 shows all possible cases of $I(l) \mapsto J(l)$. Because of the locking constraint from the lock l, the event $I(l).rel$ has to be executed before $J(l).acq$. Hence, we define the *locking order* $\rightarrow_L$ as follows:

Definition 5 (Locking Order $\rightarrow_L$)**.** *$(e \rightarrow_L f)$ iff there exists two locking intervals, $I(l)$ and $J(l)$, of the same lock l such that $(e = I(l).rel) \wedge (I(l) \mapsto J(l)) \wedge (f = J(l).acq)$.*

If $I(l)$ and $J(l)$ belong to the same thread, then the $\rightarrow_L$ relation is implied by their process order. Therefore, we only consider the $\rightarrow_L$ relation across different threads. Fig. 6 shows the corresponding locking order of all possible cases of $I(l) \mapsto J(l)$ in the dashed lines. For convenience, the locking order $I(l).rel \rightarrow_L J(l).acq$ is simplified as $I(l) \rightarrow J(l)$ from now on.

In this paper, we assume for simplicity that the initial global state does not hold any lock. If it is not lock-free, then any interval $I(l)$ that is part of the initial global state is ordered (by locking constraints) before all other intervals with the same lock l.



(a) (b)

Fig. 7. (a) An initial loset $\mathcal{L}$, which contains only the HB relation. (b) A normalized loset $\mathcal{L}'$, where the locking orders (the solid arrows) are added to the original loset $\mathcal{L}$.

Algorithm 1 NormalizeLoset($\mathcal{L}$, $\mathcal{H}$)

```
Input: A loset $\mathcal{L}$ that contains only HB orders, which are added to the set $\mathcal{H}$ of seed relations.
Output: Returns false if a cycle in the $\rightarrow$ relation is detected; otherwise, the loset $\mathcal{L}$ is normalized.
for each seed order $e_i \rightarrow e_j$ in $\mathcal{H}$ do          ▷ $\mathcal{H}$ initially contains all direct and transitive $\rightarrow$ relation.
  for each $l \in EL(e_i) \cup EL(e_j)$ do                             ▷ Exclude the case of Fig. 6(d).
    Let $I(l)$ be the most recent locking interval for $l$ s.t. $I(l).acq \preceq e_i$.
    Let $J(l)$ be the first locking interval for $l$ s.t. $e_j \preceq J(l).rel$.
    if either $I(l)$ or $J(l)$ does not exist then continue  ▷ None of the cases, Fig. 6(a), 6(b), or 6(c), holds.
    if the relation $I(l) \rightarrow J(l)$ completes a cycle then return false
    else
      Add $I(l) \rightarrow J(l)$ to the loset and to the set $\mathcal{H}$          ▷ $I(l) \rightarrow J(l)$ means $I(l).rel \rightarrow J(l).acq$.
      Append new transitive relations due to $I(l) \rightarrow J(l)$ to $\mathcal{H}$
    end if
return true
```

4.2 Normalization of Losets

Since the combination of happened-before orders and locking constraints may introduce additional order constraints $\rightarrow_L$ during execution, it is easier to analyze a loset that satisfies $\forall e, f : e \rightarrow_L f \Rightarrow e \rightarrow f$. Thus locking order leads us to the following definition:

Definition 6 (Normal Loset). *A loset* $\mathcal{L} = (E, \rightarrow, n, L, \mathcal{I})$ *is normal if* $\forall e, f \in E : e \rightarrow_L f \Rightarrow e \rightarrow f$.

Fig. 7(a) shows a loset $\mathcal{L}$, which contains only the HB relation and four locks l_w, l_x, l_y, and l_z. The events `acq(l)` and `rel(l)` correspond to the operations `acquireLock(l)` and `releaseLock(l)` of the program, respectively. The solid arrows are direct HB orders between events. The boxes of different gray-levels are the locking intervals with different locks. The effective locks of events are shown in the curly brackets. Fig. 7(b) shows the corresponding normal loset $\mathcal{L}'$, which has locking orders added to $\mathcal{L}$. The dashed arrows in Fig. 7(b) are used to explain the procedure of normalization as shown next.

At first, the HB relation $a2 \rightarrow b2$ induces the relation $I_1(l_w) \mapsto I_2(l_w)$ and hence the locking order $a3 \rightarrow b1$. Therefore, the locking order $a3 \rightarrow b1$ is added to $\mathcal{L}$. Similarly, the HB relation $b3 \rightarrow c5$ induces the relation $I_2(l_w) \mapsto I_3(l_w)$ and hence the locking order $b5 \rightarrow c4$. Afterwards, the relation $b5 \rightarrow c4$ induces $I_2(l_z) \mapsto I_3(l_z)$ and hence the locking order $b7 \rightarrow c2$. The procedure continues until no new locking order is induced. Note that the transitive HB relation $a2 \rightarrow c5$ is not shown in Fig. 7(b), which induces $I_1(l_w) \mapsto I_3(l_w)$ and hence the locking order $a3 \rightarrow c4$, because its corresponding locking order $a3 \rightarrow c4$ is transitively implied by other relations.

Algorithm 1 shows a procedure to normalize a loset $\mathcal{L}$. The algorithm takes as input the direct and transitive HB orders in the computation (i.e., $a2 \rightarrow b2$, $b3 \rightarrow c5$, and $a2 \rightarrow c5$ in Fig. 7(a)) and iteratively adds the locking orders to the computation by locating the cases of the $\mapsto$ relation in Fig. 6(a), 6(b), and 6(c). The case of Fig. 6(d) is ruled out in Algorithm 1 because the locking order is transitively implied by $I(l) \mapsto J(l)$ and does not induce any new $\rightarrow$ relation. At line 9, if the addition of $I(l) \rightarrow J(l)$ induces any transitive relation, say $e \rightarrow f$, then $e \rightarrow f$ is also appended to the set $\mathcal{H}$ for checking if any new $\mapsto$ relation is induced.

We now discuss the time complexity of the normalization procedure.

Theorem 5. *The time complexity of Algorithm 1 is $O(n|E|^3 L)$.*

Proof. Line 1 executes at most $O(|E|^2)$ times because there are at most $O(|E|^2)$ pairs of the $\rightarrow$ relation in the computation. Line 2 executes at most L times. The procedures at lines 3 and 4 can be done in constant time by using lookup tables. Finally, the time complexity for detecting the cycle at line 6 and for locating the transitive relations at line 9 is $O(n|E|)$ by recomputing vector clocks after the addition of the relation $I(l) \rightarrow J(l)$ at line 8. □

We now show that the normalized loset contains the same set of runs that reach the final global state as the original loset. We first define the runs $\mathit{Runs}(\mathcal{L})$ of a loset:

Definition 7 (Runs of a Loset)**.** *Given any loset $\mathcal{L}$, the set $\mathit{Runs}(\mathcal{L}) = \{\mathcal{R} \mid \mathcal{R}$ is a run that reaches the final global state E of $\mathcal{L}$ from the initial global state $\phi\}$.*

Theorem 6. *Let $\mathcal{L}$ be a loset and $\mathcal{L}'$ be the corresponding normal loset, then $\mathit{Runs}(\mathcal{L}) = \mathit{Runs}(\mathcal{L}')$.*

Proof. (Sketch) We show that $\mathit{Runs}(\mathcal{L}') \subseteq \mathit{Runs}(\mathcal{L})$ and $\mathit{Runs}(\mathcal{L}) \subseteq \mathit{Runs}(\mathcal{L}')$. Since $\mathcal{L}'$ contains more constraints of the $\rightarrow$ relation, we get $\mathit{Runs}(\mathcal{L}') \subseteq \mathit{Runs}(\mathcal{L})$. On the other hand, it is easily shown that any run $\mathcal{R}$ in $\mathit{Runs}(\mathcal{L})$ is also a run of $\mathit{Runs}(\mathcal{L}')$ because the run $\mathcal{R}$ in $\mathit{Runs}(\mathcal{L}, E)$ does not violate any locking order constraint and therefore only goes through feasible states of $\mathcal{L}'$. □

4.3 Strong Feasibility

If a lock l is held by a thread i in the global state G, then any other thread, say, j, that acquired the lock l prior to G should have released it before thread i subsequently acquires it. We refer this implicit order due to G as *dynamic locking order*. Formally,

Definition 8 (Dynamic Locking Order $\rightarrow_L$)**.** *$(e \rightarrow_L f)$ iff there exists two locking intervals, $I(l)$ and $J(l)$, of the same lock l such that $\big((e \in E_i) \wedge (e = I(l).rel \preceq G[i])\big) \wedge \big((f \in E_j) \wedge (f = J(l).acq \preceq G[j] \prec J(l).rel)\big)$.*

The dynamic locking orders due to G can be added to $\mathcal{H}$ and then be normalized in order to estimate the reachability of G. We now define *strong feasibility* of a global state as follows:

Definition 9 (Strong Feasibility)**.** *A feasible global state G is strongly feasible iff the normalization of the loset due to G does not induce any cycle in the $\rightarrow$ relation.*

We use the feasible global state $G = [8, 7, 7]$ in Fig. 8 to show the calculation of strong feasibility:

Step 1: From Theorem 4, this calculation can be bounded between G and the greatest lock-free feasible global state F that precedes G, i.e., the grayed out events in Fig. 8 are excluded.

Step 2: Since the lock l_z is currently held by the thread t_1, so we get the dynamic locking orders $c6 \rightarrow a7$ and $b7 \rightarrow a7$. Similarly, the lock l_y is currently held by the thread t_2, we get $a6 \rightarrow b6$.

Step 3: The HB orders of the sub-loset along with dynamic locking orders are added to the set $\mathcal{H}$ for normalization. From $b3 \rightarrow c5$, we get $b5 \rightarrow c4$ and then $b7 \rightarrow c2$. Then, the transitive relation $a6 \rightarrow c2$ establishes the relation $I_1(l_x) \mapsto I_3(l_x)$ and hence the locking order $a8 \rightarrow c1$. Consequently, a cycle in the $\rightarrow$ relation is induced: $a8 \rightarrow c1 \rightarrow c6 \rightarrow a7 \rightarrow a8$. Thus, G is not strongly feasible.

Theorem 7. *The time complexity for calculating the strong feasibility of a global state is $O(n|E|^3 L)$.*

Proof. In step 1, the lock-free feasible global state F can be identified using the detection algorithm for conjunctive predicate [13] starting from G in a backward fashion, which takes at most $O(|E|)$ time. In step 2, we can locate the dynamic locking orders due to G by pairwise processing the maximal events of G for each lock, which takes $O(n^2 L)$ time. In step 3, the normalization takes at most $O(n|E|^3 L)$ time using Algorithm 1. □

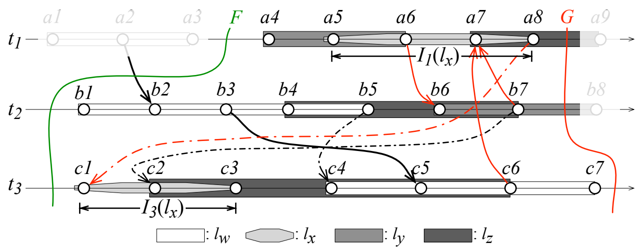


Fig. 8. The feasible global state G is unreachable because the locking order completes a cycle in the → relation.

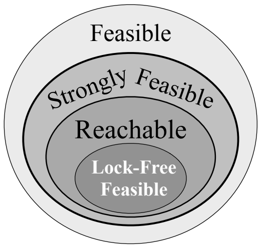


Fig. 9. The relationship among various classes of global states in a valid loset.

5 Relationship Among Various Classes of Global States

Fig. 9 shows the relationship among different sets of global states in a valid loset, whose final global state is reachable. Corollary 1 shows that all lock-free feasible global states are reachable and hence they are a subset of reachable global states. The set of strongly feasible global states is a superset of reachable global states: (1) Every reachable global state is strongly feasible because the normalization of a loset does not remove any run that reaches G, which can be shown by replacing E and $\mathcal{L}$ of Theorem 6 with G and the sub-loset from Theorem 4, respectively. Moreover, a reachable global state does not contain any cycle in the $\rightarrow$ relation. (2) A strongly feasible global state may be unreachable; an example is shown in Appendix D.

Strong feasibility is still useful in practice. In Appendix E we show that reachability equals to strong feasibility in any loset with two threads:

Theorem 8. *In a loset $\mathcal{L}$ with two threads, a global state is reachable iff it is strongly feasible.*

Moreover, in Appendix F we present experiments to show that the gap between strong feasibility and reachability seldom exists in practice. We enumerate the reachable global states, by enumerating the strongly feasible global states, of losets that are captured from the execution of benchmark programs. In comparison with two naive but accurate enumeration algorithms, which simulate the execution of the program using one thread in a BFS or DFS fashion and hence only reachable global states are enumerated, our enumeration approach is able to produce exactly the same set of global states while using only 15–40% of their runtime.

6 Conclusion

In this paper, we present Loset, a model for a computation that contains locking constraints. We first show that the reachability problem in a loset is NP-complete. Afterwards, we present several useful properties of the model. Specifically, if a loset $\mathcal{L}$ is valid, then all lock-free feasible global states are reachable. In addition, the set of reachable lock-free feasible global states forms a distributive lattice. We also show that the reachability of G can be determined using only the subset of events that is located between G and the greatest lock-free feasible global state F that precedes G. Therefore, the set of lock-free feasible global state acts as a lower approximation and “reset” point of reachability. We also present the property of strong feasibility, which is an upper approximation of reachability, and can be checked in polynomial time. The calculation is based on the inferred causality due to locking constraints and hence a reachable global state must be strongly feasible. Because of the lower and upper approximation of reachability, it is easy to answer the reachability of any given global state G in $\mathcal{L}$ if either G is lock-free feasible or not strongly feasible. If neither of these cases holds, then the reachability can be determined in the subcomputation $(G \backslash F)$ rather than the entire computation. Since our technique does not depend on the nature of predicates, it can be used for detecting the predicates whose nature are unknown and require the global view of the system.

References

1. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.
2. Y.-J. Chang and V. K. Garg. A parallel algorithm for global states enumeration in concurrent systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 140–149, 2015.
3. Y.-J. Chang and V. K. Garg. Quicklex: A fast algorithm for consistent global states enumeration of distributed computations. In *International Conference On Principles of Distributed Systems*, 2015.
4. F. Chen, T. F. Serbanuta, and G. Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.
5. R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.
6. B. A. Davey and H. A. Priestley. Introduction to lattices and order. In *Cambridge University Press*, Cambridge, UK, 1990.
7. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.
8. C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*, pages 56–66, 1988.
9. C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of ACM SIGPLAN the Conference on Programming Language Design and Implementation*, pages 121–133, 2009.
10. B. Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.
11. V. K. Garg. Enumerating global states of a distributed computation. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 134–139, 2003.
12. V. K. Garg. *Introduction to Lattice Theory with Computer Science Applications.* Wiley, 2015.
13. V. K. Garg and B. Waldecker. Detection of unstable predicates. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, 1991.
14. M. Habib, R. Medina, L. Nourine, and G. Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.
15. M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.
16. J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.
17. R. Jegou, R. Medina, and L. Nourine. Linear space algorithm for on-line detection of global predicates. In *Proceedings of the International Workshop on Structures in Concurrency Theory*, pages 175–189, 1995.
18. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proceedings of International Conference on Computer Aided Verification*, pages 505–518, 2005.
19. V. Kahlon and C. Wang. Universal causality graphs: A precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of International Conference on Computer Aided Verification*, pages 434–449, 2010.
20. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
21. Y. Lei and R. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.
22. F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 125–226, Chateau de Bonas, France, 1988.
23. N. Mittal and V. K. Garg. Techniques and applications of computation slicing. *Distributed Computing*, 17(3):251–277, 2005.
24. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of ACM SIGPLAN conference on Programming language design and implementation*, pages 446–455, 2007.
25. R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.
26. G. Pruesse and F. Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.
27. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 27–37, 1997.
28. F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 37–46, 2010.

29. A. Tarafdar. Software fault tolerance in distributed systems using controlled re-execution. In *PhD Dissertation, Department of Electrical and Computer Engineering, The University of Texas at Austin*, 2000.
30. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
31. C. von Praun and T. R. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.
32. C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. *Formal Methods*, 29:256–272, 2009.
33. C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-based symbolic analysis for atomicity violations. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342, 2010.

A Proof of Theorem 1

Theorem 1. *The loset reachability problem is NP-complete.*

Proof. The reachability problem is in NP because given a global state G of a loset $\mathcal{L}$ and a sequence $\mathcal{S}$ of events that contains exactly the same set of events as G, we can verify if $\mathcal{S}$ is a run of G by verifying that if $\mathcal{S}$ passes through only feasible global states, i.e., $\delta(\phi, \mathcal{S}^i)$ is feasible for any i such that $0 \leq i \leq |\mathcal{S}|$. The feasibility of global state can be checked in a polynomial time; specifically, it takes $O(n^2)$ and $O(n + L)$ time for checking the consistency and compatibility, respectively. Since $\mathcal{S}$ contains at most $|E|$ events, the verification takes at most $O(n^2|E|)$ time.

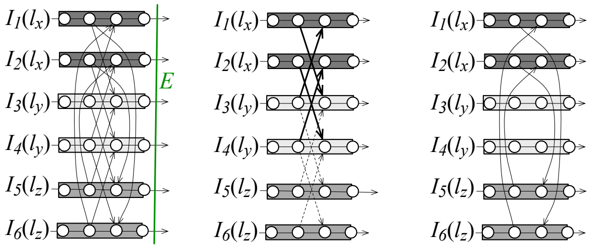
We now show that the reachability problem is NP-hard. In [29], the predicate control problem asks if there exists a control sequence, which is a total order among the critical sections for the same lock, such that the predicate Φ remains true after the control sequence is added to the computation $P = (E, \rightarrow)$. In other words, the control sequence adds additional $\rightarrow$ relations to P such that the critical sections for the same lock are totally ordered. The new computation, say, $\mathcal{Q}$, cannot contain any cycle in the $\rightarrow$ relation. In addition, every consistent global state G of P such that Φ is true remains consistent in $\mathcal{Q}$.

The NP-completeness of predicate control problem is proven by converting any 3-SAT instance into a computation, where the total orders between critical sections are the values for the corresponding variables. The predicate to detect is "every event in the set E of events of the computation is executed," i.e., the final global state E is reachable. Hence, the existence of the control sequence such that all events in E are executed is equivalent to the satisfiability of that 3-SAT instance.

The model defined in [29] is a special case of our loset model, where locking intervals do not overlap. Moreover, a control sequence does not violate the constraints of mutual exclusion and the happened-before consistency, so an execution that follows the control sequence only passes through feasible global states. Hence, the condition holds: there exists a control sequence that reaches the global state G iff there exists a run reaches G in the computation. As a result, the predicate control problem is a special case of the loset reachability problem. □

B An Example of A Loset That Is Not Valid

The example computation is shown in Fig. 10(a), which has three locks, l_x, l_y, and l_z; and six locking intervals, I_1 to I_6. The lock l_x is acquired by I_1 and I_2, l_y by I_3 and I_4, and l_z by I_5 and I_6. Moreover, each interval contains the sequence of events: the acquisition of the lock, a source of the $\rightarrow$ relation, a sink of the $\rightarrow$ relation, and the release of the lock. For simplicity, the symbol $I[i]$ denotes the event, whose index is i, that occurs in the locking interval I. We now use Fig. 10(b) and Fig. 10(c) to explain why the final global state E in Fig. 10(a) is unreachable.



(a) (b) (c)

Fig. 10. (a) A loset whose final global state is unreachable. (b)(c) The → relation in (a) is partitioned into two groups.

Fig. 10(b) shows the central part of the → relation in Fig. 10(a). In Fig. 10(b), if $I_1[1]$ is executed before $I_2[1]$, then the locking order $I_1 \rightarrow I_2$ (i.e., $I_1[4] \rightarrow I_2[1]$) is implicitly induced during the execution of the program. Then, from the chain of relations: $I_3[2] \rightarrow I_1[3] \rightarrow I_1[4] \rightarrow I_2[1] \rightarrow I_2[2] \rightarrow I_4[3]$, we get $I_3(l_y) \mapsto I_4(l_y)$ and hence the locking order $I_3 \rightarrow I_4$. On the other hand, if $I_2[1]$ is executed before $I_1[1]$, then we get $I_2 \rightarrow I_1$ and hence $I_4 \rightarrow I_3$. As a result, the solid arrows in Fig. 10(b) would induce one of the two sets of locking orders.

$$(I_1 \rightarrow I_2 \wedge I_3 \rightarrow I_4) \vee (I_2 \rightarrow I_1 \wedge I_4 \rightarrow I_3). \tag{5}$$

Moreover, because of the dashed arrows, our two sets of locking orders become:

$$(I_1 \rightarrow I_2 \wedge I_3 \rightarrow I_4 \wedge I_5 \rightarrow I_6) \vee (I_2 \rightarrow I_1 \wedge I_4 \rightarrow I_3 \wedge I_6 \rightarrow I_5). \tag{6}$$

Similar to Fig. 10(b), the → relation in Fig. 10(c) induces one of the two sets of locking orders depending upon whether $I_1[1]$ is executed before or after $I_2[1]$:

$$(I_1 \rightarrow I_2 \wedge I_6 \rightarrow I_5) \vee (I_2 \rightarrow I_1 \wedge I_5 \rightarrow I_6). \tag{7}$$

Fig. 10(a) merges the → relations of Fig. 10(b) and 10(c). Initially, the computation does not have any cycle because every pair of the → relation starts from the second event and ends at the third event of locking intervals. However, a cycle is formed whenever an event is executed. For instance, suppose that the event $I_1[1]$ is executed, then we get $(I_1 \rightarrow I_2) \wedge (I_3 \rightarrow I_4) \wedge (I_5 \rightarrow I_6)$ from (6), and $(I_1 \rightarrow I_2) \wedge (I_6 \rightarrow I_5)$ from (7). Thus, the cycle $I_6 \rightarrow I_5 \rightarrow I_6$ is formed. Consequently, the final global state E is unreachable.

Since the final global state of the computation in Fig. 10(a) is unreachable, this computation cannot correspond to an actual execution of a program.

C Proof of Theorem 3

Theorem 3. *The set of reachable lock-free feasible global states of a loset $\mathcal{L}$ forms a distributive lattice.*

Proof. We show that for any two reachable lock-free feasible global states, G and H, their meet $M = (G \cap H)$ and join $J = (G \cup H)$ are also reachable lock-free feasible global states. Since G and H are consistent global states, their meet and join are also consistent global states. Furthermore, the maximal events of G and H do not hold any lock, so the maximal events of M and J also do not hold any lock. As a result, M and J are lock-free feasible global states. Then, from Theorem 2, M is reachable because $M \leq G$. Now we show that their join J is reachable.

Because G is reachable, there exists a run $\mathcal{R}_G$. Then, from Theorem 2, the run $\mathcal{R}_G = \mathcal{R}_M \oplus \mathcal{R}_{MG}$, where $\mathcal{R}_M$ and $\mathcal{R}_{MG}$ are also runs such that $\delta(\phi, \mathcal{R}_M) = M$ and $\delta(M, \mathcal{R}_{MG}) = G$. Similarly, there exists a run $\mathcal{R}_H = \mathcal{R}_M \oplus \mathcal{R}_{MH}$ because H is reachable. We create a sequence $\mathcal{S}_J$ of events such that $\mathcal{S}_J = \mathcal{R}_G \oplus \mathcal{R}_{MH}$. Since $\mathcal{S}_J$ contains all the events in J, J is reachable if $\mathcal{S}_J$ is a run.

Claim 1. $\boldsymbol{\forall i : 0 \leq i \leq |\mathcal{S}_J| : \delta(\phi, \mathcal{S}_J^i)}$ **is consistent**:
Similar to the claim 1 of Theorem 2, we consider the two cases for any two events, e and f, in $\mathcal{S}_J$ such that $e \prec_{\mathcal{S}_J} f$:

Case 1. $(e, f \in \mathcal{R}_G) \vee (e, f \in \mathcal{R}_{MH})$: Since $\mathcal{R}_G$ and $\mathcal{R}_{MH}$ are runs, the $\rightarrow$ relation between e and f is preserved in $\prec_{\mathcal{R}_G}$ and $\prec_{\mathcal{R}_{MH}}$ and hence in $\prec_{\mathcal{S}_J}$.

Case 2. $e \in \mathcal{R}_G, f \in \mathcal{R}_{MH}$: If $e \rightarrow f$, the $\rightarrow$ relation is preserved by the concatenation $\mathcal{R}_G \oplus \mathcal{R}_{MH}$. The case $f \rightarrow e$ is not possible; otherwise, the consistency of G is violated.

Since $\mathcal{R}_G$ is a run, it is sufficient to show that the execution of $\mathcal{R}^i_{MH}$ starting from G results in a compatible global state:

Claim 2. $\boldsymbol{\forall i : 0 \leq i \leq |\mathcal{R}_{MH}| : \delta(G, \mathcal{R}^i_{MH})}$ **is compatible**:
Let $V = \delta(G, \mathcal{R}^i_{MH})$. We show that

$$\forall s \neq t : \textsc{el}(V[s]) \cap \textsc{el}(V[t]) = \varnothing. \qquad (8)$$

Let $W = \delta(M, \mathcal{R}^i_{MH})$, then the condition holds because R_{MH} is a run to reach H from M:

$$\forall s \neq t : \textsc{el}(W[s]) \cap \textsc{el}(W[t]) = \varnothing. \qquad (9)$$

Since both G in $\delta(G, \mathcal{R}^i_{MH})$ and M in $\delta(M, \mathcal{R}^i_{MH})$ are lock-free feasible global states, we get $\textsc{el}(V[t]) = \textsc{el}(W[t])$ for any thread t. Then, from (9), (8) holds.

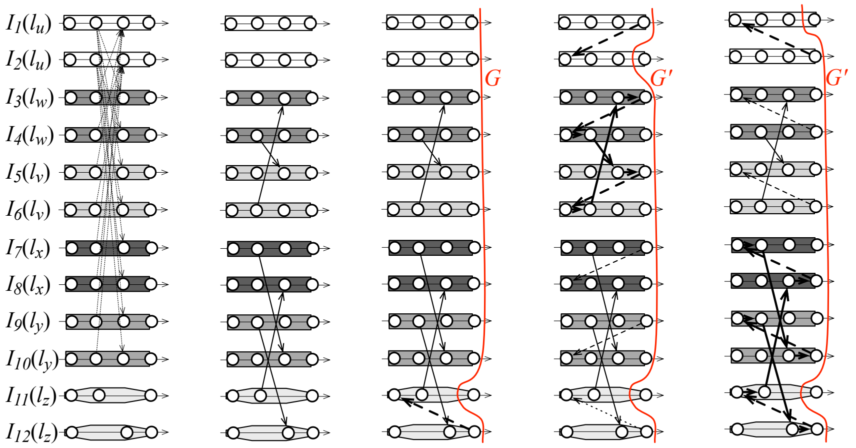
From claims 1 and 2, S_J is a run and hence J is reachable.

Finally, the lattice of lock-free feasible global states is distributive because it is a sub-lattice of the distributive lattice of consistent global states. □

D Strong Feasibility Does Not Imply Reachability

Since a reachable global state cannot contain any cycle in the $\rightarrow$ relation of a loset, a run can go through only strongly feasible global states. Hence, if none of the maximal events e of G can be removed from G such that $G - \{e\}$ is strongly feasible, then G is unreachable. In this section, we show a strongly feasible global state G such that removing any of its maximal events would result in a global state that is not strongly feasible, i.e., G is strongly feasible but not reachable. Moreover, the loset is valid, i.e., the final global state is reachable, so it is possible to capture this computation from the execution of a real-world application.

The example is shown in Fig. 11(a), which has six locks: l_u, l_w, l_v, l_x, l_y, and l_z. The lock l_u is a coordinator, which has the $\rightarrow$ relation that is similar to that of the computation shown in Fig. 10(b). In short, any removal of the last event of the intervals I_2, I_4, I_6, I_8, and I_{10}, induces the set A of locking orders: $(I_1 \rightarrow I_2) \wedge (I_3 \rightarrow I_4) \wedge (I_5 \rightarrow I_6) \wedge (I_7 \rightarrow I_8) \wedge (I_9 \rightarrow I_{10})$; and any



(a) (b) (c) (d) (e)

Fig. 11. A loset whose final global state is reachable. In addition, G is strongly feasible but unreachable. The locking orders are drawn in dashed arrows.

removal of the last event of the intervals I_1, I_3, I_5, I_7, and I_9, induces the set B of locking orders: $(I_2 \rightarrow I_1) \wedge (I_4 \rightarrow I_3) \wedge (I_6 \rightarrow I_5) \wedge (I_8 \rightarrow I_7) \wedge (I_{10} \rightarrow I_9)$.

Fig. 11(b) shows the remaining $\rightarrow$ relation in the computation, i.e., the combination of Fig. 11(a) and Fig. 11(b) is the complete computation. The computation does not contain any cycle in the $\rightarrow$ relation initially because every pair of the $\rightarrow$ relation starts from the second event and ends at the third event of locking intervals. For ease of reading, the arrows in Fig. 11(a) are omitted in the other figures of Fig. 11. The final global state can be reached by the run that preserves the partial order: (1) $I_{11} \rightarrow I_{12}$, and (2) $(I_1 \rightarrow I_2) \wedge (I_3 \rightarrow I_4) \wedge (I_5 \rightarrow I_6) \wedge (I_7 \rightarrow I_8) \wedge (I_9 \rightarrow I_{10})$.

Fig. 11(c) shows the strong feasible global state G, where the locking order $I_{12} \rightarrow I_{11}$ is induced because l_z is held by the thread t_{11}. In G, the removals of $G[11]$ and $G[12]$ would violate the consistency constraints and the locking constraints, respectively. Thus, we consider the removal of the maximal events on other threads, i.e., $G[1]$ to $G[10]$. Those maximal events can be divided into two groups: the ones that induce the set A of locking orders and the ones that induce the set B of locking orders.

Let the symbol $I[i]$ denote the event, whose index is i, that occurs in the locking interval I. We first consider the case where the set A of locking orders is induced, which is shown in Fig. 11(d). Without loss of generality, suppose that the set of orders is induced by the removal of $G[2]$ (i.e., $I_2[4]$). Then, the following cycle is induced: $I_3[4] \rightarrow I_4[1] \rightarrow I_4[2] \rightarrow I_5[3] \rightarrow I_5[4] \rightarrow I_6[1] \rightarrow I_6[2] \rightarrow I_3[3] \rightarrow I_3[4]$. On the other hand, suppose that the set B of locking orders is induced by the removal of $G[1]$ (i.e., $I_1[4]$) as shown in Fig. 11(e). Then, the following cycle is induced: $I_7[1] \rightarrow I_7[2] \rightarrow I_{10}[3] \rightarrow I_{10}[4] \rightarrow I_9[1] \rightarrow I_9[2] \rightarrow I_{12}[3] \rightarrow I_{12}[4] \rightarrow I_{11}[1] \rightarrow I_{11}[2] \rightarrow I_8[3] \rightarrow I_8[4] \rightarrow I_7[1]$. Therefore, the global state G is strongly feasible but unreachable.

E Proof of Theorem 8

Theorem 8. *In a loset $\mathcal{L}$ with two threads, a global state is reachable iff it is strongly feasible.*

Proof. It is sufficient to show that any strongly feasible global state G of a loset with two threads is always reachable. We show this by induction on the size of G. When $|G| = 0$, G is the initial global state and therefore reachable. Now consider any G such that $|G| > 0$. We will show that there exists a maximal event e in G such that $G - \{e\}$ is also strongly feasible. By the induction hypothesis, we can assume that $G - \{e\}$ is reachable and therefore G is reachable.

We now show that there does not exist a strongly feasible global state G such that removing any of its maximal event results in a global state that is not strongly feasible. Let $H = G - G[1]$ and $F = G - G[2]$. Without loss of generality, we show that if H is not strongly feasible, then $G[1] \rightarrow G[2]$. We consider the following three cases:

Case 1. *H is not consistent*: It is obvious that $G[1] \rightarrow G[2]$. (See Fig. 12(a).)

Case 2. *H is not compatible*: An example loset is shown in Fig. 12(b). If H is not compatible, then there exists one lock $l \in \text{EL}(H[1]) \cap \text{EL}(G[2])$. Let $I(l)$ and $J(l)$ be the two intervals for the lock l such that $I(l).acq \preceq H[1] \prec I(l).rel$ and $J(l).acq \preceq G[2] \prec J(l).rel$. Since G is compatible

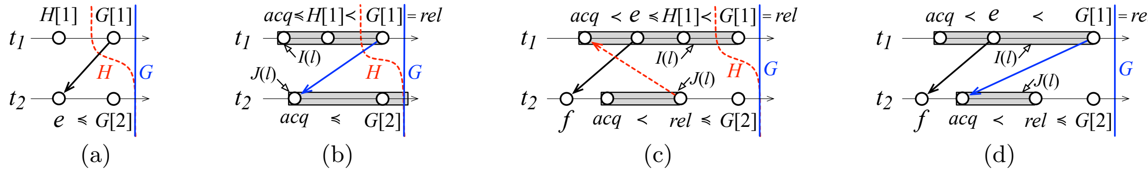


Fig. 12. (a) Case 1: $H = G - G[1]$ is inconsistent. (b) Case 2: H is incompatible. (c) Case 3: H induces a cycle in the $\rightarrow$ relation and either $(f \preceq acq)$ or $(acq \preceq f)$ holds. (d) Case 3: The cycle in (c) implies $G[1] \rightarrow G[2]$.

(i.e., $\textsc{el}(G[1]) \cap \textsc{el}(G[2]) = \varnothing$), we get $G[1] = I(l).rel$. Consequently, the locking order $I(l).rel \rightarrow_L J(l).acq$ is induced in G and hence $G[1] \rightarrow G[2]$.

Case 3. *H contains a cycle in the $\rightarrow$ relation*: Fig. 12(c) shows an example loset. Since G is strongly feasible, the cycle must be completed by a locking order that is induced by H. Suppose that the locking order is induced because of the lock l, then the following conditions hold:

1. Since the locking order only exits in H, there exists an interval $I(l)$ such that $H[1] \prec I(l).rel = G[1]$.
2. There exists an interval $J(l)$ such that $J(l).rel \preceq G[2]$. Thus, the locking order $J(l).rel \rightarrow_L I(l).acq$ can be induced in H but not G.

In order to complete the cycle, there exists a relation $e \rightarrow f$ in H such that $I(l).acq \prec e \preceq H[1]$ and $f \prec J(l).rel$. Since the computation has only two threads, any locking order due to H must point toward the events that occur on t_1. Hence, the relation $e \rightarrow f$ is either an existing HB relation of the computation or a locking order that is induced by $G[2]$. In either case, $e \rightarrow f$ also exists in G. Then, $e \rightarrow f$ would induce the relation $I(l) \mapsto J(l)$ in G (see Fig. 12(d)) and hence the locking order $G[1] \rightarrow_L J(l).acq$, which implies $G[1] \rightarrow G[2]$.

If both H and F are not strongly feasible, then we get $G[1] \rightarrow G[2]$ and $G[2] \rightarrow G[1]$. Therefore, G contains the cycle $G[1] \rightarrow G[2] \rightarrow G[1]$, which is a contradiction to the assumption that G is strongly feasible. □

F Enumeration of Reachable Global State in the Loset Model

There are two approaches in literature to enumerate reachable global states of a computation. The first approach uses breadth (BFS) or depth (DFS) first strategy to add one event to the current global state G at a time [5, 11]. The event to be added satisfies the feasibility of G. This approach simulates the execution the program using one thread and hence every enumerated global state is reachable. Because DFS and BFS algorithms might enumerate the same global state more than once, this approach has to store the enumerated global states. In the worst case, the memory space for storing might grow exponentially in the number of threads in the computation.

An alternative approach predefines or calculates a spanning tree among the lattice of consistent global states and enumerates the global states following the edges of the tree [3, 10, 11, 14, 17, 26]. However, an edge may pass through unreachable global states because the set of consistent global states is a superset of reachable global states in a loset. Therefore, this approach needs to incorporate an additional function to prune the consistent but unreachable global states. In this paper, we use QuickLex [3] to enumerate the consistent global states and use strong feasibility to prune the unreachable global states.

Table 1 shows the information of the benchmarks that are used in the experiment. The benchmark *banking* is a toy program, which was used to demonstrates typical error patterns in concurrent programs [7]; *arraylist1* is a non-thread-safe container and *arraylist2* is a thread-safe container from Java library; *set1* and *set2* are implementations of concurrent sets using different fine-grained locking strategies [15]; *sor* is a scientific computation application; *raytracer*, *moldyn*, and *montecarlo* are parallel programs from Java Grande benchmark suite; *hedc* is a crawler for searching Internet archives; and *tsp* is a parallel solver for the traveling salesman problem. The benchmarks *sor*, *raytracer*, *moldyn*, *montecarlo*, *hedc*, and *tsp* are the benchmark programs used in [4, 9, 31]. In addition, the columns of "n", "#events", and "#GS" show the number of threads, the number of events, and the number of enumerated global states of the computation, respectively.

All the experiments are conducted on a Linux machine with an Intel Xeon 2.67 GHz CPU and the heap size of Java virtual machine is limited to 2GB. The runtime is measured in seconds.

Table 1. The information of benchmarks and runtimes (sec.) of each enumeration approach.

Benchmark	n	#events	#GS	Runtimes			n	#events	#GS	Runtimes		
				BFS	DFS	Our Method				BFS	DFS	Our Method
bank	7	91	664,325	0.99	3.20	0.09	9	121	53,808,433	350.27	o.o.m.	4.47
arraylist1	12	56	354,293	0.57	1.06	0.07	16	76	28,697,813	175.80	o.o.m.	1.66
arraylist2	7	103	3,045,808	4.48	30.28	0.22	8	118	25,740,144	104.81	o.o.m.	1.75
set1	6	114	947,951	1.36	5.25	1.16	7	147	15,040,942	40.21	o.o.m.	23.02
set2	6	140	2,762,420	3.55	28.70	3.16	7	189	78,130,591	452.43	o.o.m.	160.38
sor	14	66	3,188,645	9.16	32.29	0.22	16	76	28,697,813	174.48	o.o.m.	1.64
raytracer	9	121	4,882,833	10.36	42.57	0.54	10	132	24,414,083	98.15	o.o.m.	2.83
moldyn	13	83	3,188,633	8.66	23.77	0.22	15	93	28,697,831	166.83	o.o.m.	2.08
montecarlo	12	78	354,315	1.53	1.06	0.05	16	98	28,697,835	227.51	o.o.m.	1.88
hedc	7	92	458,334	0.64	1.50	0.38	9	121	24,522,560	108.37	o.o.m.	7.30
tsp	8	76	1,235,981	1.99	11.26	0.17	10	90	25,000,001	115.77	o.o.m.	52.33

Table 1 contains two sets of results. The set at the left of the table shows the largest computations that the DFS algorithm can handle, i.e., the DFS algorithm would run out of memory when the computations contain one more thread. On the other hand, the set at the right of the table shows the largest computations that the BFS algorithm can handle. The BFS and DFS algorithms generate the reachable global states and our approach generates strongly feasible global states. However, all the compared algorithms generate the same set of global states. Meanwhile, our approach reduces 84% and 61% of runtime in comparison with BFS and DFS algorithms, respectively.