Copyright
by
Yen-Jung Chang
2016

The Dissertation Committee for Yen-Jung Chang
certifies that this is the approved version of the following dissertation:

# Predicate Detection for Parallel Computations

Committee:

Vijay K. Garg, Supervisor

Craig M. Chase

Christine Julien

Sarfraz Khurshid

Keshav Pingali

Lingming Zhang

# Predicate Detection for Parallel Computations

**by**

**Yen-Jung Chang, B.S., M.S.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2016

In loving memory of my beautiful grandma, Wu-Hao Chang (1933-2014).

# Acknowledgments

I would like to express my deep appreciation and gratitude to my advisor, Dr. Vijay Garg, for the patient guidance and mentorship he provided to me. His guidance has made this a thoughtful and rewarding journey. I would like to thank Dr. Sarfraze Khurshid for the friendly words of encouragement during my difficult times. I would also like to thank my dissertation committee of Drs. Craig Chase, Christine Julien, Keshav Pingali, and Lingming Zhang for the support over the past two years. I would like to thank my fellow students Wei-Lun Hung, Himanshu Chauhan, John Bridgman, Bharath Balasubramanian, Yuqun Zhang, James Zheng, Rui Qiu, and Lisa Hua for their advices and friendship. I am very thankful to Yen-Yu Chang, a great roommate who takes care of me and cooks delicious meals for me. I would like to thank to Jim Wall for the friendship and all the fun he brings to me during my long journey in finishing this doctoral program. I am deeply thankful to my parents, Jung-Feng Chang and Su-Luwan Lee, for their unconditional love. Without them, this dissertation would never have been written. The last word of acknowledgment I have saved for my dear girlfriend Nai-Yu Chen, who has been with me all these years and has made them the best years of my life.

Yen-Jung Chang

*The University of Texas at Austin*
*May 2016*

# Predicate Detection for Parallel Computations

Yen-Jung Chang, Ph.D.
The University of Texas at Austin, 2016

Supervisor: Vijay K. Garg

One of the fundamental problems in runtime verification of parallel program is to check if a predicate could become true in any global state of the system. The problem is challenging because of the nondeterministic process or thread scheduling of the system. Predicate detection alleviates this problem by analyzing the computation of the program and predicting whether the predicate could become true by exercising an alternative process schedule. The technique was first introduced by Cooper et al. and Garg et al. for distributed debugging. Later, jPredictor applies this technique for concurrent debugging.

We improve the technique of predicate detection in three ways. The first part of this dissertation presents the first online-and-parallel predicate detector for general-purpose predicate detection, named ParaMount. ParaMount partitions the set of consistent global states and each subset can be enumerated

in parallel using existing sequential enumeration algorithms. Our experimental results show that ParaMount speeds up the existing sequential algorithms by a factor of 6 with 8 threads. Moreover, Paramount can run along with the execution of users' program and hence it is applicable even to non-terminating programs.

The second part develops a fast enumeration algorithm, named QuickLex, for consistent global states. In comparison with the original lexical algorithm (Lex), QuickLex uses an additional $O(n^2)$ space to reduce the time complexity from $O(n^2)$ to $O(n \cdot \Delta(\mathcal{P}))$, where $n$ is the number of processes or threads in the computation and $\Delta(\mathcal{P})$ is the maximal number of incoming edges of any event.

The third part introduces Loset — a new model for parallel computations with locking constraints. We show that the reachability problem in a loset is NP-complete. To tackle the NP-completeness, we present several useful properties. Specifically, if the final global state is reachable, then all lock-free feasible global states are reachable. In addition, we show that the reachability of a global state G can be determined using a sub-computation instead of the entire computation. Moreover, we introduce the strong feasibility of a global state, which is an upper approximation of reachability that can be calculated efficiently. Our experiments show that the property accurately models the reachability for all 11 benchmarks.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

One of the fundamental problems in debugging or runtime verification of a parallel program is to determine whether an user specified condition (*predicate*) could become true in any reachable global state of the program. This problem is challenging because each run of the program may reach a different set of global states due to nondeterministic thread scheduling even for the same user input.

A common approach for detecting the possibility of the predicate in the system is to execute the program repeatedly and incorporate a thread scheduler to ensure that each run of the program explores some previously unexplored global states [MQ07, VHB+03, LC06]. Since a global state could be reached repeatedly in different runs, this approach usually incorporates partial order reduction to reduce the number of repeated explorations [MQ07, VHB+03]. Nonetheless, re-executing the program could be time-consuming.

The technique of *predicate detection* alleviates the problem by analyzing a given *computation* (execution trace) of the program and predicting whether the predicate could become true in any of the global states that can be reached with a different thread schedule. The technique is predictive because it does

```
Thread t1                Thread t2
--------------           --------------
a1: read(x)              b1: revMsg(t1, &m)
a2: sendMsg(t2, m)       b2: write(x)
a3: write(x)
```

Figure 1.1: A parallel program in which threads use messages to synchronize with each other.

not actually re-execute the program in order to explore different global states due to the thread scheduling. Instead, it generates inferred reachable global states from the given computation. Then, it checks if the predicate can become true in any of the inferred global states. The technique is first introduced by Cooper et al. [CM91] and Garg et al. [GW91] for distributed debugging. Later, jPredictor [CSR08] applies this technique for concurrent programs.

## 1.1 Predicate Detection for Debugging

As an example of predicate detection, consider the condition $\Phi$: ($x$ *is written by two threads at the same time*) which corresponds to a bug in the parallel program shown in Figure 1.1. We would like to know if it is possible to reach a global state of the program such that $\Phi$ is true. As mentioned before, one of the popular debugging methods is to run the program and collect a total order of events. Suppose that the total order recorded is $a1, a2, a3, b1, b2$. In this total order, $\Phi$ does not become true. However, the predicate is indeed possible if the sequence of events starts with the prefix $a1, a2, b1$. Hence, the only way possibility of $\Phi$ would be detected is via multiple executions and

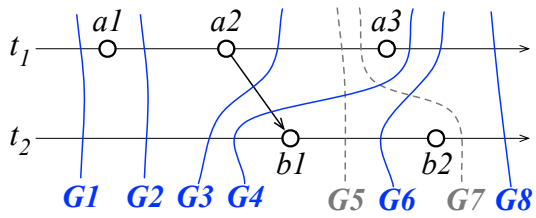



Figure 1.2: The captured logical order between events, which form a poset. $G1$ to $G8$ are consistent global states of the program.

hope that one of the executions runs a total order that makes the predicate true.

To alleviate the problem, the technique of predicate detection models the computation as a partially ordered set (poset) of events, in which the events are ordered by Lamport's happened-before (HB) relation [Lam78] (denoted by $\rightarrow$). In the poset model, a global state $G$ is consistent iff $\forall e, f : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G)$, where $e$ and $f$ are events of the computation. For each consistent global state, there exists at least one sequence of events to reach the global state from the initial global state [CM91]. Therefore, the possibility of $\Phi$ in the parallel program is predictively detected by checking if $\Phi$ could become true in any of the consistent global states of the poset.

Figure 1.2 shows the computation that is captured from the execution of the program in Figure 1.1. Each event of the computation corresponds to an operation of the program. The HB relation between events $a2$ and $b1$ is established by the message passing between the two threads. The horizontal lines are the consistent global states of the computation; in the graphical representation, a global state contains the events on its left, e.g., the global

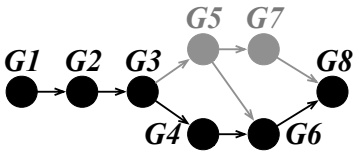



Figure 1.3: The relationship among the consistent global states of the computation.

state $G4$ contains the events $a1, a2$, and $a3$.

To reach the global state $G4$, the events are executed in this sequence: $a1, a2, a3$. Some global states may be reached by multiple sequences of events. For instance, $G6$ can be reached by the two sequences: 1) $a1, a2, a3, b1$, and 2) $a1, a2, b1, a3$. The relationship among the consistent global states of the computation is shown in Figure 1.3. We can see that if the event sequences 1) and 2) reach global states $G4$ and $G5$, respectively, then they can reach $G6$ by executing events *b1* and $a3$, respectively. So, we say that the global state $G6$ is reachable from both $G4$ and $G5$.

When a program executes an event, the program reaches the next global state. Thus, the observed execution of the program can be represented by a sequence of global states. Assume that $G1, G2, G3, G4, G6, G8$ is the observed execution. Then the objective of predicate detection is to generate the global states $G5$ and $G7$ and hence the two inferred executions, $G1, G2, G3, G5, G6, G8$ and $G1, G2, G3, G5, G7, G8$, can be predictively verified without re-executing the program.

A predicate is defined to determine if the user specified condition could become true in a global state. We use the condition of data races to explain

```
1. predicate(GlobalState G) {
2.   for (int i = 1; i <= n; ++i) {
3.     for (int j = i; j <= n; ++j) {
4.       if (G[i] and G[j] are concurrent and
                conflicting events)
5.           // a bad condition is found.
6.     }
7.   }
8. }
```

Figure 1.4: A predicate that looks for if there exists a pair of maximal events that are conflict in the global state $G$.

how a predicate is defined. A race condition occurs when conflicting operations (e.g., a pair of read-write or write-write operations) are *concurrently* executed on the same memory address by different threads. If two events of different threads have no causal dependency in a global state, they can be executed concurrently. For instance, the two maximal events, $a3$ and $b2$, in the global state $G8$ in Figure 1.2 do not have causal dependency. Hence, they can be executed concurrently. Figure 1.4 shows a predicate for detecting data races. The nested *for* loop at lines 2 and 3 gets all pairs of maximal events of the global state $G$; the symbols $G[i]$ and $G[j]$ at line 4 are the maximal events of thread $t_i$ and $t_j$, respectively. In Figure 1.2, since the events $a3$ and $b2$ are the write operations to the same memory address (i.e., variable $x$), a potential race condition is detected in the global state $G8$.

In summary, predicate detection allows us to predictively detect $\Phi$ in the system if it could become true in any of the consistent global states of

the given poset. The technique contains three major steps. First, the computation is captured from the execution of the program and modeled as poset of events. Second, an enumeration algorithm takes as input the computation and generates all consistent global states of that computation. Third, the consistent global states are checked if any one of them satisfies the condition of the predicate. Since the technique can be applied in both concurrent and distributed systems, the term *computations* refers to both concurrent and distributed computations from now on. In addition, the terms *thread* and *process* are interchangeable; unless specified otherwise.

This dissertation improves the technique of predicate detection in three ways. The first part of this dissertation presents an online-and-parallel predicate detector, named ParaMount [CG15a], for concurrent systems. The second part gives a fast enumeration algorithm, named QuickLex [CG15b], for consistent global states. The first and second parts mainly focus on the technique of predicate detection using the conventional poset model, in which the real-time order between critical sections are also represented by Lamport's happened-before relation. The third part of this dissertation presents a new model, called Loset (Locking Poset), for computations with locking constraints. In a loset, synchronization due to locks are not modeled using the happened-before relation; instead, the sets of events that are executed under one or more locks are modeled separately.

## 1.2 Online-and-Parallel Predicate Detection

One common approach for the debugging of concurrent program is to re-execute the program multiple times and check if any one of the executions induces a bug. For instance, CHESS [MQ07], Java PathFinder [VHB+03], and RichTest [LC06] execute the program repeatedly and incorporate a scheduler to ensure that each run of the program explores some new global states. CHESS and Java PathFinder schedule concurrent events into a totally ordered sequence and enable the corresponding threads to execute the events one at a time. The scheduler of RichTest directly changes the partial order among concurrent events. The approach retains the concurrency of events and hence they can be executed concurrently during the testing of the program.

Some debugging tools combine the technique of scheduler and the technique of predicate detection. These tools have two phases: prediction and replay. In the prediction phase, inferred reachable global states are generated from the computation and are checked to determine if the predicate holds [LTQZ06, HZ11]. In the replay phase, the program is re-executed incorporating a scheduler in order to determine whether the inferred global states, where the predicate holds, can actually be reached [PLZ09, SFM10, YNPP12]. However, their method assumes that the condition to be detected involves only two threads (e.g., data races, atomicity violations, etc.) and uses heuristic strategies to enumerate the set of inferred reachable global states.

jPredictor [CSR08] is the first general-purpose predicate detector for concurrent debugging; it ensures that every consistent global state is enumer-

ated at least once. However, jPredictor enumerates the set of consistent global states in an offline and sequential fashion.

## Contribution

In the first part of this dissertation, we present the first online-and-parallel algorithm, named ParaMount [CG15a], for consistent global states enumeration and predicate detection. ParaMount partitions the set of consistent global states of the given poset into multiple subsets. It ensures that every consistent global state belongs to exactly one subset. For each subset of consistent global states, ParaMount can use existing sequential enumeration algorithms as its subroutine without increasing the asymptotic work complexity. In this dissertation, we use the BFS algorithm [CM91] or the lexical algorithm [Gan10, Gar03, CG15b] for the subroutine.

From the experimental results, ParaMount is 6 to 11 times faster than the original sequential algorithms when using 8 threads. The reason that ParaMount sometimes shows superlinear speedup is that partitioning the set of consistent global states transforms the original problem into multiple subproblems that are much easier to solve. Moreover, partitioning also reduces the memory space consumed by intermediate data which eliminates the running time wasted by Java garbage collector.

ParaMount is also an *online* enumeration algorithm, which can incrementally enumerate the consistent global states during the construction of the poset. Because of this property, ParaMount can run along with the execu-

tion of parallel programs and is applicable even to non-terminating programs such as web-server applications. Note that the online and parallel property of ParaMount can be applied together. Thus, it is possible to use ParaMount to perform an *online-and-parallel* predicate detection.

We evaluate the online property of ParaMount by conducting an online-and-parallel predicate detection of data races in concurrent programs. We compare our predicate detector with another general-purpose predicate detector, RV runtime [MR10] (the successor of jPredictor), and an online data race detector, FastTrack [FF09], using several benchmarks, e.g., *sor, tsp*, and *hedc* [CSR08, FF09, vPG01]. On average, our detector is 10 to 50 times faster than RV runtime. On the benchmark *raytracer*, RV runtime runs out of memory whereas our detector uses only 25% of the system memory. The performance of our detector is also comparable to that of FastTrack for most benchmarks even though the enumeration algorithm of ParaMount is not designed specifically for detecting data races.

## 1.3 A Fast Enumeration Algorithm for Consistent Global States

For certain classes of predicates, the computation time of enumeration algorithm can be reduced to polynomial time because only a partial set of global states needs to be enumerated [GW94, HMRS96, TG97, CG98, OG07, SG02, LTQZ06, HZ11, FF09, PLZ09, SFM10]. In this dissertation, we focus on the techniques that do not have any assumption on the nature of the predicate,

Table 1.1: Time and space complexity of existing general-purpose enumeration algorithms.

| **Algorithms** | **Time per CGS** | **Space** |
|---|---|---|
| ***Cooper–Marzullo*** [CM91] | $O(n^3)$ | exp. in $n$ |
| ***Alagar–Venkatesan*** [AV01] | $O(n^3)$ | $O(\|E\|)$ |
| ***Steiner*** [Ste86] | $O(\|E\|)$ | not available |
| ***Squire*** [Squ95] | $O(log\|E\|)$ | not available |
| ***Pruesse–Ruskey*** [PR93] | $O(\|E\|)$ | exp. in $n$ |
| ***Jegou and Habib et al.*** [JMN95, HMNS01] | $O(\Delta(\mathcal{P}))$ | $O(\|E\|)$ |
| ***Lexical*** [Gan10, Gar03] | $O(n^2)$ | $O(n)$ |
| ***QuickLex*** [CG15b] | $O(n\cdot \Delta(\mathcal{P}))$ | $O(n^2)$ |

$n$: the number of processes in the computation $\mathcal{P}$.
$E$: the set of events in $\mathcal{P}$.
$\Delta(\mathcal{P})$: the maximal in-degree of any event in $\mathcal{P}$.

i.e., the detection is general-purpose. If no assumption is made regarding the predicate, then enumerating every consistent global state is necessary, which requires exponential time because the number of consistent global states, $i(\mathcal{P})$, grows exponentially in the number of processes in the computation $\mathcal{P}$.

The time complexity of a general-purpose enumeration algorithm can be calculated by multiplying $i(\mathcal{P})$ by *the time complexity per consistent global state*, which is the time to advance from one consistent global state to the other. For simplicity, we use the time complexity per consistent global state to represent the time complexity of a general-purpose enumeration algorithm.

Cooper and Marzullo [CM91] gave the first general-purpose enumeration algorithm based on a breadth first strategy (BFS) that requires $O(n^3)$ time and exponential space in $n$, where $n$ is the number of processes in the computation $\mathcal{P}$. Alagar and Venkatesan [AV01] presented the notion of global

interval which reduces the space complexity to $O(|E|)$, where $|E|$ is the number of events in $\mathcal{P}$. Steiner [Ste86] gave an algorithm that uses $O(|E|)$ time, and Squire [Squ95] further improved the computation time to $O(log|E|)$.

Pruesse and Ruskey [PR93] gave an algorithm that enumerates consistent global states in a combinatorial Gray code manner. The algorithm uses $O(|E|)$ time and can be reduced to $O(\Delta(\mathcal{P}))$, where $\Delta(\mathcal{P})$ is the maximal in-degree of any event; however, the space grows exponentially in $n$. Later, Jegou et al. [JMN95] and Habib et al. [HMNS01] improved the space complexity to $O(|E|)$.

Ganter [Gan10] presented an algorithm, which enumerates consistent global states in the lexical order, and Garg [Gar03] gave an implementation using vector clocks [Fid88, Mat88]. The lexical algorithm requires $O(n^2)$ time, but the algorithm requires only $O(n)$ space besides the input, i.e., the computation. The $O(n)$ space is only used for storing the vector clock that represents the current global state. Table 1.1 summarizes the time and space complexity of those general-purpose enumeration algorithms. Note that the space complexity of an enumeration algorithm only considers the memory space that stores the intermediate information during the enumeration.

## Constributions

In the second part of this dissertation, we present QuickLex — a fast algorithm for enumerating the set of consistent global states of a given poset in the lexical order. In comparison with the existing lexical algorithm

(Lex) [Gan10, Gar03], QuickLex reduces the time complexity from $O(n^2)$ to $O(n \cdot \Delta(\mathcal{P}))$. The time complexity can be reduced to $O(n)$ for the commonly used computations [CSR08, LC06, FF09, HMNS01, JMN95], in which most events send and receive at most one message.

We evaluate QuickLex using multiple benchmarks including four computations that are captured from the executions of benchmark programs. In our experiments, QuickLex is 7 times faster than the Lex algorithm [Gan10, Gar03] and 4–5 times faster than the Tree algorithm [HMNS01, JMN95]. We note here that QuickLex is faster than the Tree algorithm even though the asymptotic worst case time complexity for the Tree algorithm is lower. There are two reasons for this. First, the time complexity of QuickLex is calculated as the worst case, which is not a common computation in practice. Second, the Tree algorithm needs to store its temporary spanning tree in a linked-list, which induces large performance overhead during the enumeration; QuickLex only uses arrays.

As far as space complexity is concerned, QuickLex uses almost the same amount of memory as Lex, which shows that the extra space for dynamic programming in QuickLex is quite small. The Tree algorithm uses 2–10 times more memory than QuickLex.

```
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)
```

Figure 1.5: A program which has two threads that might open the file `f` at the same time.

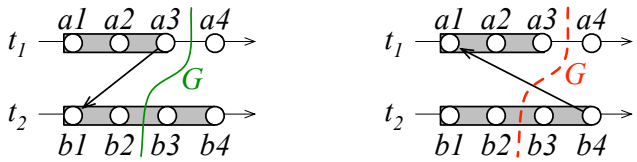



Figure 1.6: The global state $G$ contains the events {a1, a2, a3, b1, b2} and the predicate $\Phi$ is true only in $G$. (a) In this poset, $G$ is reachable and thus $\Phi$ can be correctly detected. (b) In this poset, $G$ is unreachable and thus $\Phi$ cannot be detected.

## 1.4 Predicate Detection for Computations with Locking Constraint

Since the poset model does not consider the constraints due to locks, one common modification to the model is to capture the real-time order of lock synchronizations as the causality of the program [FF09, LC06, CG15a, CSR08], i.e., the release of a lock happened before the subsequent acquisition of that lock. However, a lock of the program specifies the sets of events that cannot be concurrently executed instead of the causality between events. Hence, the mutual exclusion provided by locks has a different nature than the HB relation.

For the program in Figure 1.5, a possible computation modeled as a poset is Figure 1.6(a), which corresponds to the execution where the thread $t_1$ obtains the lock $l$ before $t_2$. Assume that we are interested in detecting the condition $\Phi$: *file $f$ is opened by two threads at the same time*, which corresponds to a bug in the program. As it can be seen, the predicate $\Phi$ can be detected in the global state $G$ in Figure 1.6(a). However, we still have not solved the problem of predicate detection for all thread schedules. Suppose that the thread $t_2$ obtains the lock before $t_1$ during the execution. Then, we put a happened-before order between $b4$ and $a1$ as shown in Figure 1.6(b). In this poset, it is not possible to reach the global state $G$, which is inconsistent, where $\Phi$ is true. Consequently, a poset based predicate detection algorithm will miss the global state reached under a different locking schedule.

An alternative approach for modeling the synchronizations due to locks is incorporating the notion of lockset [SBN+97], which provides the information for identifying the events that cannot be concurrently executed instead of the happened-before relation. However, most of the existing models that use this notion only consider the conditions that involve only two threads (e.g., data races and atomicity violations) [KIG05, KW10, SFM10, OC03]. 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.

Assume that we have a program (see Figure 1.7) which has three threads. Because of the conditional wait of $c2$ on $a2$, the system would ensure that the thread $t_1$ obtains the lock before $t_3$ and hence we get the computa-

```
Thread t1                Thread t2              Thread t3
-----------              -----------            -----------
a1: acquireLock(l)       b1: recMsg(t3,&m)      c1: acquireLock(l)
a2:   l.notify()         b2: f.openFile()       c2:   l.waitUntilNotified()
a3:   f.openFile()                              c3:   sendMsg(t2,m)
a4:   f.closeFile()                             c4: releaseLock(l)
a5: releaseLock(l)
```

Figure 1.7: A program which has three threads but the file `f` can only be opened by one thread at a time.

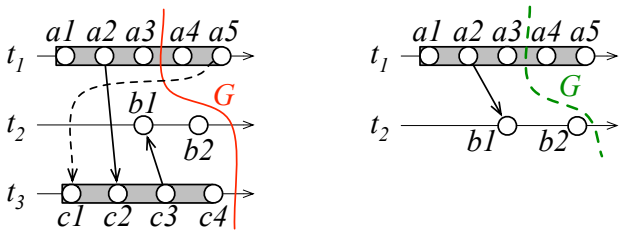



Figure 1.8: (a) The global state $G$, where $\Phi$ is true, is indeed unreachable because of the implicit order (the dashed arrow) between the two critical sections. (b) The local view that contains only two of the threads, where $G$ is mistakenly considered reachable.

tion as shown in Figure 1.8(a). Because of the conditional wait and the lock, the order $a5 \rightarrow c1$ is always implicitly induced during the execution of the program. Hence, the global state $G$, where $\Phi$ is true, is indeed unreachable. However, if we try to detect the condition $\Phi$ in a local view that contains only two of the threads (see Figure 1.8(b)), then the global state $G$ could be mistakenly considered reachable and result in a false-positive.

## Contributions

In this dissertation, we argue that the synchronization due to locks is fundamentally different from the potential causality. We present an alternative model that makes a distinction between the happened-before relation and the synchronization of locks. Our model, named Loset (Locking Poset), is a generalization of the poset model. It allows us to detect possibility of violation of invariants which would not be possible to detect using an arbitrary one of the posets of parallel computation. Just as a poset is equivalent to possibly an exponential number of total orders, a loset is equivalent to possibly an exponential number of posets. Therefore, detecting a predicate on a loset is equivalent to detecting if that predicate became true in any of the posets.

Given a loset $\mathcal{L}$ and a global state $G$, the reachability problem asks if $G$ is a reachable global state of $\mathcal{L}$. Note that this problem is trivial for a poset: $G$ is reachable iff $G$ is a consistent global state [CM91]. However, we show that the reachability problem for a loset is NP-complete. Our proof uses NP-completeness of the predicate control problem shown in [Tar00].

Since reachability is NP-complete, in this dissertation we 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. We show that for computations with two threads, the set of strongly feasible global states is identical to the set of reachable global states. We also give examples of computations in which a strongly feasible state is not reachable. However, for many practical applications, strongly feasible global

states provide exact approximation of reachability. We have implemented a predicate detector based on strongly feasible instead of reachability for debugging concurrent computations. The experimental results show that the strongly feasible property accurately models the reachable global states for all 11 benchmark programs with more than two threads.

We also 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 also show that the set of lock-free feasible global states forms a finite distributive lattice under the usual less than relation of global states. Furthermore, we show that the reachability of a global state $G$ can be determined using only a subset of events which is located between the greatest lock-free global state that precedes $G$ and $G$. Thus, lock-free feasible states act as "reset" points for reachability and can be used to drastically reduce the time for checking reachability, by checking reachability in a subcomputation rather than the entire computation.

We note here that reachability of a global state in a parallel computation has also been solved using SAT/SMT solver [WKGG09, WLGG10, HZ11]. These solvers take exponential amount of time in the worst case. Our focus in this dissertation is on techniques that take polynomial time. Moreover, our techniques are orthogonal to 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 its greatest preceding lock-free consistent global state. Moreover, we only

need to calculate the reachability with a SAT/SMT solver only if $G$ is strongly feasible.

## 1.5 Summary

- **ParaMount**: We present the first *online-and-parallel* predicate detector for detecting general-purpose predicates in concurrent systems. We develop a parallel enumeration algorithm, named ParaMount, which partitions the set of consistent global states into multiple subsets. ParaMount guarantees that every global state is enumerated exactly once; therefore, it is applicable for detecting general-purpose predicate. Moreover, the online property of ParaMount allows it to be run along with the execution of parallel programs and hence is applicable even to non-terminating programs. From the experimental results, ParaMount speedups the existing sequential algorithms up to 11 times with 8 threads. We also build a simple online-and-parallel predicate detector using ParaMount to detect data races in concurrent programs. We compare the detection results from ParaMount with those from another general-purpose predicate detector, RV runtime [MR10], and an online data race detector, FastTrack [FF09] on several benchmarks. Even though ParaMount is not specifically designed for detecting data races, the experiments show that it is still useful even for data races for most benchmarks.

- **QuickLex**: We develop a fast algorithm, named QuickLex, for enumerating the set of consistent global states of the given poset. In com-

parison with the original lexical algorithm, QuickLex has a preprocessing procedure and incorporates dynamic programming to reduce the time complexity from $O(n^2)$ to $O(n \cdot \Delta(\mathcal{P}))$. We also implement and compare QuickLex with several existing enumeration algorithms, i.e., BFS [CM91, Gar03], Lex [Gan10, Gar03], and Tree [JMN95, HMNS01]. In our experiments, the performance of these existing algorithms are enhanced with different techniques. From the experimental results, QuickLex is 7 times faster than Lex and 4–5 times faster than Tree. The experiments also show that QuickLex can achieve amortized constant time for a certain type of computations. QuickLex uses almost the same amount of memory as Lex while Tree requires 2–10 times more memory than QuickLex.

- **Loset Model**: We propose a new model called Loset (Locking Poset) for modeling parallel computations with locking constraints. We first show that determining the reachability of a global state in a loset is NP-complete. To cope with the NP-completeness, several useful properties of the loset model are given. Specifically, if a loset 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 lock-free feasible states act as "reset" points for reachability and can be used to drastically reduce the time for checking reachability. Moreover, we present a method to calculate the strong feasibility of a global state, which is an upper approximation of reachability, in polynomial time.

The calculation is based on the inferred causality due to the locking constraints and hence a reachable global state must be strongly feasible. Finally, we have implemented a predicate detector based on strongly feasible instead of reachability for debugging concurrent computations. Our experimental results show that the strongly feasible property accurately models the reachable global states for all 11 benchmark programs with more than two threads.

## 1.6 Overview

The reminder of this dissertation is organized as follows. Chapter 2 gives the definition of the conventional poset model, which is based on Lamport's happened-before relation, and introduces the problem of consistent global states enumeration. Next, we have the three main chapters of this dissertation: Chapter 3 investigates the parallelism of predicate detection problem. Chapter 4 presents QuickLex – a fast enumeration algorithm. Chapter 5 presents the new model, called Loset, for considering the locking constraint in parallel computations. Moreover, Chapter 6 studies the reachability of global states in the loset model. Finally, Chapter 7 concludes the dissertation and Chapter 8 discusses future directions.

# Chapter 2

# The Computation of Poset Model

In this chapter, we give the definition of poset model, mechanism to track the relation, and the notion of consistent global states.

## 2.1 Poset Model

The computation (the execution trace) of a parallel program is commonly modeled as a partially ordered set (poset) of events [Lam78]. The advantage of using the poset model is that the results of predicate detection do not have false positives [FF09], i.e., if the detector finds a consistent global state that satisfies the predicate, then there exists a sequence of events to reach the global state. Hence, it has been widely used for the debugging of parallel programs [FF09, LC06, CSR08, GW94, CG98, TG97, OG07]. Note that the model assumes that process and thread scheduling is the only source of nondeterminism in the program.

A poset $\mathcal{P} = (E, \rightarrow)$ contains a set $E$ of events together with Lamport's happened-before (HB) relation [Lam78] (denoted by $\rightarrow$). Each event in the

This chapter is previously published in [CG15b].

set $E$ of events corresponds to an operation of the program, e.g., a read or a write operation, the acquisition of a lock, the receiving of a message, etc.) Moreover, the events in $E$ are partitioned into $n$ sequences $E_1, E_2, \cdots, E_n$ of events such that the events belong to the same sequence, say $E_i$, are totally ordered, i.e., for all distinct $e, f \in E_i : (e \rightarrow f) \vee (f \rightarrow e)$. The sequence $E_i$ represents the process $p_i$ in the computation.

## 2.2 Causality and the Happened-Before Relation

Give a set $E$ of events, the happened-before relation $\rightarrow$ is the smallest binary relation such that:

1. **Process Order:** If $e$ occurs before $f$ on the same process, then $e \rightarrow f$.
2. **Events Synchronization:** If $e$ sends a message and $f$ receives the message, then $e \rightarrow f$.
3. **Transitivity:** If $e \rightarrow g$ and $g \rightarrow f$, then $e \rightarrow f$.

For convenience, we define the process order relation (denoted $\prec$) such that $e \prec f$ means $e \rightarrow f$ in some $E_i$. In concurrent systems, the happened-before relation is also established by the following rules [LC06, FF09]:

4. **Lock Atomicity**: If event $e$ corresponds to a thread releasing a lock and $f$ corresponds to subsequent acquisition of that lock (including implicit locks and monitors), then $e \rightarrow f$.

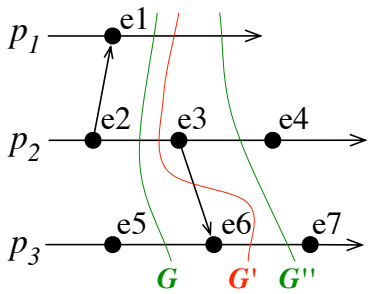



Figure 2.1: A poset $P$ of events corresponding to an execution of the program. The global states $G$ and $G''$ are consistent global states and $G'$ is not.

5. **Fork-Join**: If the parent thread forks a new thread on event $e$ and the child thread is created on event $f$, the $e \rightarrow f$. Similarly, if a child thread terminates on event $e$ and the parent thread joins the child thread on event $f$, then $e \rightarrow f$.

6. **Wait-Notification**: If a thread waits on a monitor on the event $e$ and a thread sends the notifications of that monitor on $f$, then $e \rightarrow f$. In addition, if a thread receives the notification on the event $f$ and the notification is sent from the $e$, then $e \rightarrow f$.

If events $e$ and $f$ have no happened-before relation, then they are *concurrent* (denoted by $e \parallel f$).

Figure 2.1 shows a graphical representation of a poset. The computation contains three processes $p_1$, $p_2$, and $p_3$. The horizontal arrows represent the total order of the events that occur on the same process and the arrows

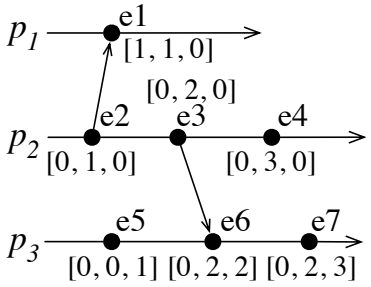



Figure 2.2: The vector clocks of the events.

**Algorithm 1** Calculate vector clock for event $e$

**Input:** Event $e$ occurs on process $p_i$.
**Output:** The vector clock for event $e$.

```
1: function calVectorClock(e)
2:     e.vc = d.vc                    ▷ d is the direct predecessor of e.
3:     e.vc[i] = d.vc[i] + 1
4:     for any event (f → e) ∧ (f ↛ d)) do
5:         for j from 1 to n do
6:             e.vc[j] = max(e.vc[j], f.vc[j])
7:         end for
8:     end for
9: end function
```

across different processes represent the direct happened-before relation between events on different processes.

During the execution of the program, the happened-before relation between events is captured using vector clocks [Fid88, Mat88]. A vector clock, $vc$, is an array of integers. For an event $e$, which occurs on process $p_i$, the integer $e.vc[i]$ is the index of $e$ among the events that occur on $p_i$. For $j \neq i$, $e.vc[j]$ is the largest index of event $f$ among the events that occur on process

$p_j$ such that $f \rightarrow e$. Figure 2.2 shows the vector clocks of the computation in Figure 2.1. The vector clock of the event $e7$ is $[0, 2, 3]$, which means the index of the current event $e7$ is 3. Moreover, the event $e3$, which has index 2 in $p_2$, happened before the event $e7$.

Algorithm 1 shows the procedure for calculating the vector clock $e.vc$ of any event $e$, which occurs on process $p_i$, during the execution of the program. For instance, the vector clock of event $e6$ in Figure 2.2 is set to $[0, 0, 2]$ because of its predecessor $e5$. Then, its vector clock is set to $[0, 2, 2]$ because $e3 \rightarrow e6$.

## 2.3 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 Figure 2.1, $\{e1, e2, e3\}$ is a global state, but $\{e1, e3\}$ is not a global state because it contains $e3$ but not $e2$ even though $e2 \prec e3$. In the graphical representation of computation, a global state is drawn as a curved vertical line and contains all the events on its left. For instance, the global state $G$ in Figure 2.1 contains the events: $e1, e2$, and $e5$.

In this dissertation, a global state can equivalently be identified by the maximal events of each process, called *frontier*. These maximal events are simply represented by an array of integers, in which the $i$-th integer indicates the index of the maximal event among the events that occur on process $p_i$. If the index is zero then no event on the corresponding process is included in

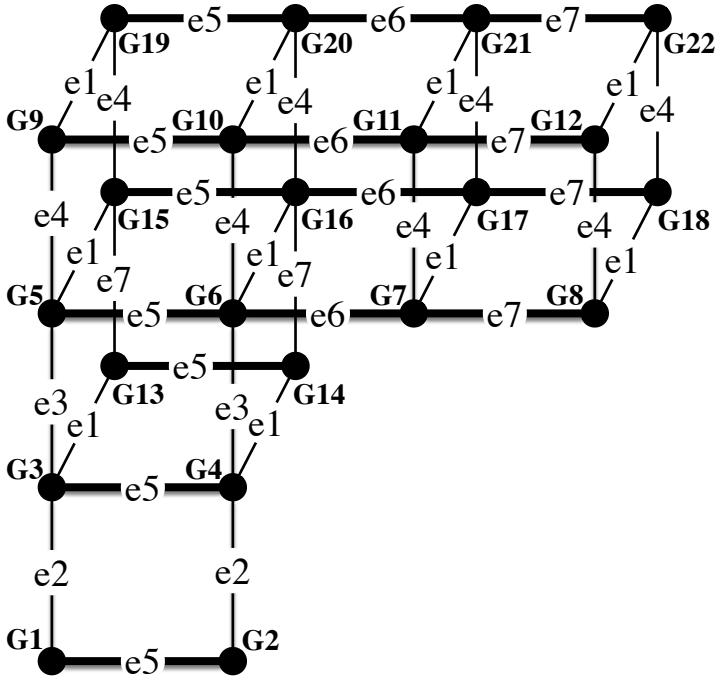



Figure 2.3: A distributive lattice formed by the set of consistent global states of the poset shown in Figure 2.1.

the global state. For instance, $G''$ in Figure 2.1 is represented by $[1, 2, 2]$. The symbol $G[i]$ denotes the maximal event of process $p_i$ in $G$, e.g., $G''[2]$ refers to event $e3$.

## 2.4 Consistent Global States

A *consistent global state* $G$ is a subset of $E$, such that if $G$ includes any event $f$, then it also includes all events that happened before $f$ [CL85].

Formally, $G \subseteq E$ is a consistent global state if

$$\forall e, f \in E : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G).$$

In Figure 2.1, the global states $G$ and $G''$ are consistent and $G'$ is not, because $e3 \rightarrow e6$ but $e3 \notin G'$.

Given a computation, let $\leq$ be a relation on global states which is defined as follows:

$$G \leq H \stackrel{\text{def}}{\equiv} \forall i : 1 \leq i \leq n : G[i] \leq H[i].$$

It is well known that the set of consistent global states of the computation forms a lattice [DP90]. Specifically, let $M$ be the meet of $G$ and $H$ (i.e., $M = G \sqcap H$) and $J$ be the join of $G$ and $H$ (i.e., $J = G \sqcup H$). If $G$ and $H$ are consistent global states, then $M$ and $J$ are also consistent global states. Moreover, the lattice of consistent global states is distributive, i.e., $G \sqcap (F \sqcup H) = (G \sqcap F) \sqcup (G \sqcap H)$ and $G \sqcup (F \sqcap H) = (G \sqcup F) \sqcap (G \sqcup H)$ hold.

Figure 2.3 shows the lattice that is formed by the consistent global states of the computation in Figure 2.1. Each node of the lattice corresponds to a consistent global state and the edge label denotes the event that takes the system from one consistent global state to the other. The objective of a general-purpose enumeration algorithm is to enumerate every consistent global state in the lattice at least once.

# Chapter 3

# Online-and-Parallel Enumeration of Consistent Global States

In this chapter, we describe our online-and-parallel predicate detector for detecting general-purpose predicates in concurrent systems.

jPredictor [CSR08] is the first general-purpose predicate detector for concurrent debugging. It does not have assumptions on the property of the predicate and ensures that every consistent global state is enumerated at least once. In jPredictor, the consistent global states of the given poset are enumerated in an offline fashion using the BFS algorithm [CM91]. There are two reasons that jPredicator performs offline detections. First, jPredictor constructs the computation in a backward fashion. Second, the BFS algorithm [CM91] cannot enumerate the set of consistent global states incrementally. Consequently, jPredictor has to wait until the program terminates.

In fact, most of the existing enumeration algorithms [CM91, AV01, Ste86, Squ95, PR93, HMNS01, Gan10, Gar03], whose details are discussed in section 1.3, are single threaded and can only be used in an offline predicate

This chapter is previously published in [CG15a].

detection for terminating programs. Although Jegou et al. [JMN95] had studied the algorithm for online enumeration, their algorithm cannot enumerate the set of consistent global states in parallel. The reason is that their algorithm incrementally builds the subset of consistent global states using the information from previously built subsets.

We have developed the first online-and-parallel algorithm, named Para -Mount [CG15a]. ParaMount divides the set of consistent global states of a poset into multiple intervals (subsets) and provides the following two properties: 1) every consistent global state is contained in one of the intervals and 2) all intervals are disjoint. For each interval, ParaMount can use existing sequential enumeration algorithms as its subroutine without increasing the asymptotic work complexity.

In our experiments, we use the breadth-first-strategy (BFS) [CM91] algorithm or the lexical enumeration algorithm [Gan10, Gar03] for the subroutine. From the experimental results, ParaMount can speed up these sequential algorithms from 6 to 11 times with 8 threads. The reason for the speed up of more than 8 is that partitioning the set of global states helps the sequential algorithm (i.e., BFS) reduce the memory space (which is consumed by intermediate data) and hence the running time that is wasted by Java garbage collector.

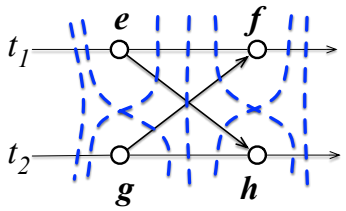



Figure 3.1: A poset of events. The consistent global states of the poset are shown by the dashed lines.

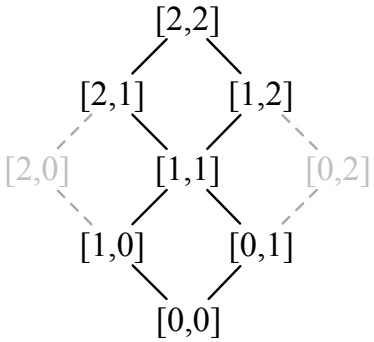



Figure 3.2: The relationship among the consistent global states of the poset in Figure 3.1. The grayed out global states are inconsistent.

## 3.1 Partitioning the Set of Consistent Global States

In the rest of this chapter, we use the computation in Figure 3.1 to show how ParaMount works. In Figure 3.1, all consistent global states of the poset are shown by the dashed lines. In addition, the relationship among those consistent global states is illustrated in Figure 3.2, in which the grayed out global states are inconsistent. *The objective of ParaMount is to enumerate the set of consistent global states in parallel.* The symbol $G(e)$ denotes a global state containing the event $e$ in its frontier. Given a set of global states such that

**Algorithm 2** `ParaMountWorker`$(P)$

**Input:** A poset of events $P$.

1: **while** true **do**
2: Event e $\leftarrow P$.getNextEventInTotalOrder$\rightarrow_p()$
3: **if** e is null **then** break;
4: $G_{min}(e) = e.vc$ ▷ Get least global state from $e$'s vector clock.
5: $G_{bnd}(e) \leftarrow P$.getBoundaryGlobalState()
6: `BoundedEnumeration`$(P, G_{min}(e), G_{bnd}(e), e)$ ▷ Enumerate $\forall G : G_{min}(e) \leq G \leq G_{bnd}(e)$ using Algorithm 3.
7: **end while**

each of them contains $e$ in its frontier, let $G_{min}(e)$ denote the *least global state*, and $G_{max}(e)$ denote the *greatest global state*. In Figure 3.1, $G_{min}(e) = [1, 0]$ and $G_{max}(e) = [1, 2]$.

Algorithm 2 shows the worker procedure of ParaMount; each worker is executed by a thread during the enumeration. Before starting the workers, ParaMount determines a total order $\rightarrow_p$ among the events in the poset. Since the poset of events forms a directed acyclic graph (DAG), the order $\rightarrow_p$ can be calculated using any topological sort algorithm for DAG [CSRL01]. Because of the topological sort, this property holds:

**Property 1.** *For all events $e$ and $f$, $e \rightarrow f \Rightarrow e \rightarrow_p f$.*

In addition, two concurrent events $e$ and $f$ can be sorted in either $e \rightarrow_p f$ or $f \rightarrow_p e$. In other words, the total order $\rightarrow_p$ among the events is equivalent to the execution order of the events when the program is run on one single thread. For the poset in Figure 3.1, the four possible total orders among the events are:

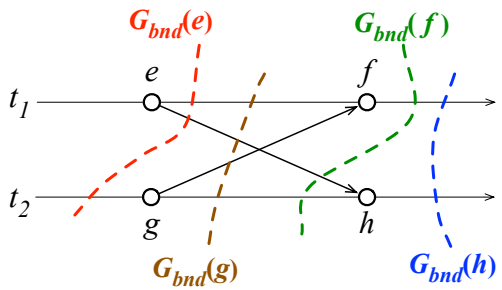



Figure 3.3: The boundary global states of the events in the poset. Assume that the total order among the events is $e \rightarrow_p g \rightarrow_p f \rightarrow_p h$.

1. $e \rightarrow_p g \rightarrow_p f \rightarrow_p h$,
2. $e \rightarrow_p g \rightarrow_p h \rightarrow_p f$,
3. $g \rightarrow_p e \rightarrow_p h \rightarrow_p f$,
4. $g \rightarrow_p e \rightarrow_p f \rightarrow_p h$.

Any one of the total orders can be used by ParaMount to partition the set of global states, which is performed from lines 2 to 5 at Algorithm 2.

The boundary of an interval of global states is defined by two global states $G_{min}$ and $G_{bnd}$, which are determined with respect to the events in the poset. Specifically, ParaMount computes $G_{min}(e)$ and $G_{bnd}(e)$ for each event $e$. Then, any global state $G$ such that $G_{min}(e) \leq G \leq G_{bnd}(e)$ is contained in that interval. Here, $G_{bnd}(e)$ is defined as follows:

**Definition 1.** $G_{bnd}(e) = \{f \in E \mid (f = e) \vee (f \rightarrow_p e)\}$.

The examples of $G_{bnd}$ are shown in Figure 3.3, in which we assume that the total order among the events is $e \rightarrow_p g \rightarrow_p f \rightarrow_p h$. For event

$f$, $G_{bnd}(f)$ includes all events that are totally ordered before $f$. Hence, we get $G_{bnd}(f) = [2, 1]$. For the same total order, we also get $G_{bnd}(e) = [1, 0]$, $G_{bnd}(g) = [1, 1]$, and $G_{bnd}(h) = [2, 2]$. We next show that $G_{bnd}(e)$ is consistent:

**Theorem 1.** *$G_{bnd}(e)$ is a consistent global state for all event $e$.*

*Proof.* To show that $G_{bnd}(e)$ is consistent, we show that for any event $f \in G_{bnd}(e)$ if there exists any event $g$ such that $g \rightarrow f$, then $g \in G_{bnd}(e)$.

From Property 1, $g \rightarrow f$ implies $g \rightarrow_p f$. Since $f \in G_{bnd}(e)$, we get $(f = e) \vee (f \rightarrow_p e)$ from Definition 1. If $(f = e)$, we get $g \rightarrow_p e$. And if $(f \rightarrow_p e)$, we get $g \rightarrow_p e$ because of the transitivity of $\rightarrow_p$. In both cases, $g \in G_{bnd}(e)$. □

An enumeration interval $I(e)$ of global states corresponding to any event $e$ is formally defined as follows:

**Definition 2.** $I(e) = \{G \mid G_{min}(e) \leq G \leq G_{bnd}(e)\}$

Figure 3.4 shows the intervals of global states that are calculated for the events in the poset. In Figure 3.4(a), the global state $[0, 0]$ is a special case and is always enumerated by the first event in the total order $\rightarrow_p$, i.e., the event $e$.

At line 2 of Algorithm 2, ParaMount gets the next event in the total order $\rightarrow_p$. If there are no more events, then all intervals are processed. At line 4, $G_{min}(e)$ is simply obtained from the vector clock $e.vc$. At line 5, $G_{bnd}(e)$ is calculated according to the total order among the events in the computation.

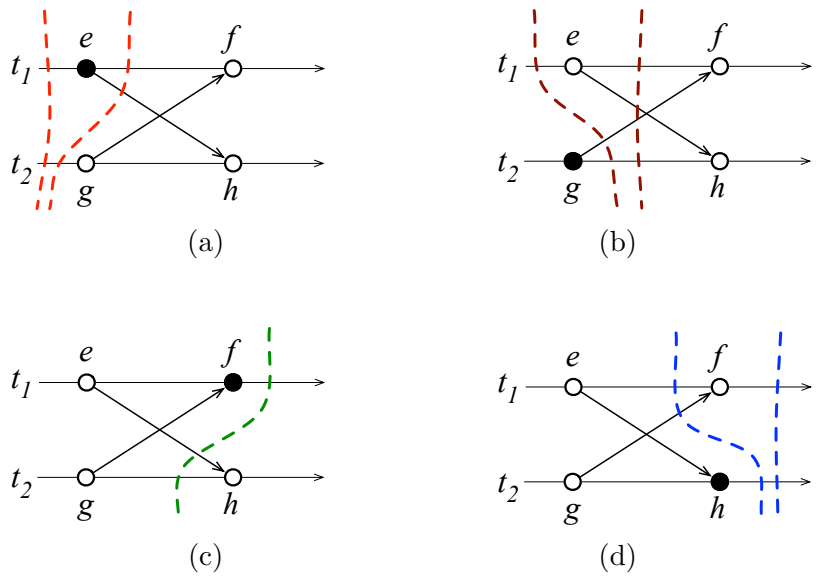



Figure 3.4: Assume that the total order among the events is $e \rightarrow_p g \rightarrow_p f \rightarrow_p h$, then we get (a) the interval $I(e)$, (b) the interval $I(g)$, (c) the interval $I(f)$, and (d) the interval $I(h)$ of global states. The global state $[0, 0]$ is a special case and always belongs to the interval of the first event in the total order $\rightarrow_p$, which is the event $e$.

At line 6 of Algorithm 2, ParaMount enumerates the interval of global state for the corresponding event.

## 3.2 Bounded Enumeration Algorithm

ParaMount can use existing sequential algorithm as its subroutine to enumerate the intervals of global states. However, the sequential algorithm needs to be modified (or bounded) to provide the two properties. First, it takes as input the boundary of an interval of global states and enumerates only the global states in the interval. Second, it enumerates each global state

in the given interval exactly once.

We use the lexical enumeration algorithm in [Gan10, Gar03] as an example to show the modification for the subroutine. Algorithm 3 shows the bounded lexical algorithm. The first modification is located at line 1: the least global state $G_{min}(e)$ of event $e$ is used as the initial global state. The second modification is located at lines 2, 4, and 6: the boundary global state $G_{bnd}(e)$ is used to limit the global states that are enumerated by the algorithm. At line 3, the user specified condition is checked whether it can become true in current global state $G$. Lines 5 to 14 is a simplified implementation of the original lexical enumeration algorithm in [Gan10, Gar03].

**Lemma 1.** *Given an event $e$ in the poset $\mathcal{P}$, Algorithm 3 enumerates every consistent global state $G$ in the interval $I(e)$ exactly once.*

*Proof.* Suppose there exists a poset $Q$, which has an initial global state $G_{init}$ and a final global state $G_{final}$. Lines 5-15 give the least consistent global state in lexical order as shown in [Gar03]. Specifically, the *while* loop at line 2 enumerates every global state $G$ of $Q$ such that $G_{init} \leq G \leq G_{final}$. By the definition of $G_{min}(e)$, $G_{min}(e) \leq G_{bnd}(e)$. Algorithm 3 uses the property by assigning $G_{min}(e)$ to $G_{init}$ and $G_{bnd}(e)$ to $G_{final}$. Hence, Algorithm 3 enumerates every consistent global state $G$ of $P$ exactly once such that $G_{min}(e) \leq G \leq G_{bnd}(e)$. □

In the evaluation section, the similar modification is applied into the BFS algorithm [CM91] for comparison.

**Algorithm 3** `BoundedEnumeration`$(\mathcal{P}, G_{min}(e), G_{bnd}(e), e)$

**Input:** A poset $\mathcal{P}$, the new event $e$, and the least $G_{min}(e)$ and boundary $G_{bnd}(e)$ global state of $e$.

1: $G \leftarrow G_{min}(e)$ ▷ $G$: the current global state.
2: **while** $G \leq G_{bnd}(e)$ **do**
3: $predicate(P, G, e)$ ▷ Check the predicate upon $G$.
4: **if** $G = G_{bnd}(e)$ **then** break; ▷ Reached the boundary of $I(e)$.
5: **end if**

6: $k \leftarrow n$
7: **for** $k \leftarrow n$ to $1 : G[k] \leq G_{bnd}(e)[k]$ **do** ▷ Select a new event $e_k$ to add into $G$.
8: Event $e_k$ = the next event on thread $t_k$.
9: **if** $e_k$ is enabled **then** break;
10: **end if**
11: **end for**

12: $G[k] \leftarrow G[k] + 1$ ▷ Add event $e_k$ into $G$.
13: **for** $i \leftarrow (k+1)$ to $n$ **do** $G[i] \leftarrow G_{min}(e)[i]$ ▷ Reset events due to lexical order.
14: **end for**
15: **for** $i \leftarrow (k+1)$ to $n$ **do**
16: **for** $j \leftarrow 1$ to $k$ **do**
17: Event $e_j$ = the current maximal event on $t_j$.
18: $G[i] \leftarrow \max(G[i], e_i.vc[i])$
19: **end for**
20: **end for**
21: **end while**

## 3.3 Correctness of ParaMount

Now we show that every global state is contained in one of the intervals of global states (Lemma 2) and all intervals are disjoint (Lemma 3).

**Lemma 2.** *In Algorithm 2, for every consistent global state $G$ of the poset $\mathcal{P}$, there exists an event $e$ such that $G \in I(e)$.*

*Proof.* We show that for any consistent global state $G$ in the poset $\mathcal{P}$, there exists an event $e$ such that $G_{min}(e) \leq G \leq G_{bnd}(e)$. Let $e$ be the last event (with respect to the total order $\rightarrow_p$) in $G$. From the definition of $G_{min}(e)$, we get $G_{min}(e) \leq G$. Since $e$ is the last event in $G$, for any event $f$ in $G$, either $(f \rightarrow_p e)$ or $(f = e)$. Then from the definition of $G_{bnd}(e)$, we get $G \leq G_{bnd}(e)$. □

**Lemma 3.** *In Algorithm 2, for every consistent global state $G$ of the poset $\mathcal{P}$, there exists at most one $e$ such that $G \in I(e)$.*

*Proof.* Suppose event $e$ is the last event (with respect to the total order $\rightarrow_p$) in $G$. We now show that there does not exist any event $f \neq e$ such that $G_{min}(f) \leq G \leq G_{bnd}(f)$. The proof is by contradiction.

Suppose that $G_{min}(f) \leq G \leq G_{bnd}(f)$. Since $f \in G_{min}(f)$, we get $f \in G$. Because $e$ is the last event in $G$, we get $(f \rightarrow_p e)$, which implies $(e \neq f) \wedge (e \not\rightarrow_p f)$. From the definition of $G_{bnd}$ (Definition 1), we get $G \not\leq G_{bnd}(f)$, which is a contradiction. □

**Theorem 2.** *Algorithm 2 enumerates every consistent global state of the poset $\mathcal{P}$ exactly once when it uses Algorithm 3 as a subroutine.*

*Proof.* Follows from Lemma 1, Lemma 2, and Lemma 3. □

## 3.4 Work and Space Complexity of ParaMount

Now we analyze the work complexity of ParaMount when using Algorithm 3 as its subroutine. Suppose that the poset $\mathcal{P}$ consists of $n$ threads, $|E|$ events, $|H|$ pairs of happened-before relation, and $i(\mathcal{P})$ global states. The work complexity of the topological sort is $O(|E| + |H|)$. For each worker, the work complexity is $O(n)$ because it has to store $G_{min}$ and $G_{bnd}$ at lines 4 and 5. Algorithm 3 takes $O(n^2)$ work for each global state because of the nested *for* loop at lines 11 and 12. Due to Theorem 2, Algorithm 3 cumulatively enumerates exactly $i(\mathcal{P})$ global states. As a result, the combination of ParaMount and Algorithm 3 takes $O(n^2 \cdot i(\mathcal{P}))$ work, which is as the same as that of the sequential lexical algorithm. In this sense, ParaMount is work optimal.

As for space complexity, ParaMount uses $O(n)$ space for storing $G_{min}$ and $G_{bnd}$. Hence, the total space complexity of ParaMount is $O(n \cdot |E|)$. Note that the existing general-purpose predicate detector, RV runtime [MR10], uses the BFS algorithm [CM91], which consumes memory space exponential in the number of threads in the poset. Thus, it could run out of memory even for a moderately sized benchmark.

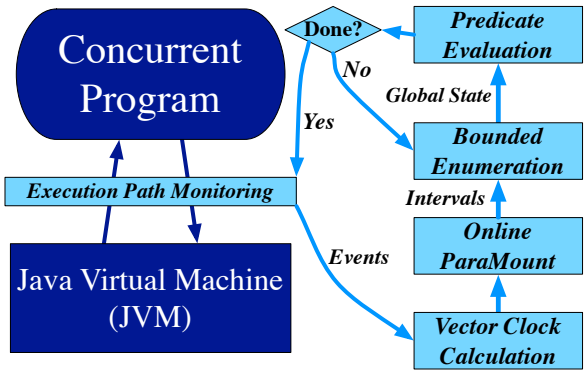



Figure 3.5: The framework of our online-and-parallel predicate detector.

## 3.5 Implementation of Online Predicate Detector

To evaluate the online property of ParaMount, we use it to build an online-and-parallel predicate detector. Figure 3.5 shows the framework of our predicate detector, which is described next.

### 3.5.1 Construction of Poset $\mathcal{P}$

In the first part, the detector captures the events, which are relevant to the condition to be detected, and their causal dependencies from the observed execution path of the program. When the program is loaded into JVM in the first time, the detector uses bytecode injection technique [ASM] to inject monitoring instructions into the program during runtime. The injected bytecode are stored in memory, so the original Java program and Java bytecode are unmodified.

When the program starts, the operations of the program are captured as events. Then, the events along with their causal dependencies are converted

**Algorithm 4** `calculateVectorClock`$(t, l)$

**Input:** The thread $t$, whose pid is $i$, that acquires the lock $l$
**Output:** The vector clock for the new event $e$ of the acquisition.

1: $t.vc[i] \leftarrow t.vc[i] + 1$
2: **for** $k \leftarrow 1$ to $n$ **do**
3: $\quad t.vc[k] = \max(t.vc[k], l.vc[k])$
4: **end for**
5: $l.vc \leftarrow t.vc$
6: $e.vc \leftarrow t.vc$

**Algorithm 5** `calculateVectorClock`$(t_i, t_j)$

**Input:** Thread $t_i$ executes an event $e$ that happened-before $f$, which occurs on thread $t_j$.
**Output:** The vector clocks for events $e$ and $f$.

1: $t_i.vc[i] \leftarrow t_i.vc[i] + 1$
2: $t_j.vc[j] \leftarrow t_j.vc[j] + 1$
3: **for** $k \leftarrow 1$ to $n$ **do**
4: $\quad t_j.vc[k] = \max(t_i.vc[k], t_j.vc[k])$
5: **end for**
6: $e.vc \leftarrow t_i.vc$
7: $f.vc \leftarrow t_j.vc$

into the poset $\mathcal{P}$ which is defined in Chapter 2. During bytecode injection, every thread and lock object is automatically attached with a vector clock. When a lock-atomicity event is inserted into $\mathcal{P}$, the vector clocks of thread and lock are updated using Algorithm 4. For example, let event $e$ correspond to the operation of a thread $t$ acquiring a lock $l$, then the two vector clocks, $t.vc$ and $l.vc$ are updated using Algorithm 4. The returned vector clock is copied to the event's vector clock $e.vc$. If the new events are related to fork-join or wait-notification operations, the vector clock of the involved threads and events are

**Algorithm 6** `OnlineParaMountWorker`($\mathcal{P}$, $e$)

**Input:** The new event $e$ to be inserted into the poset $\mathcal{P}$.

1: **atomic {**
2: Insert $e$ into the data structure of $\mathcal{P}$.
3: $G_{min}(e) = e.vc$
4: $G_{bnd}(e) \leftarrow \mathcal{P}$.snapshotOfMaximalEventsOfThreads()
5: **}**
6: `BoundedEnumeration`$(\mathcal{P}, G_{min}(e), G_{bnd}(e), e)$ ▷ Enumerate the interval $I(e)$ of global states using Algorithm 3.

updated using Algorithm 5. If the inserted event is a process-ordered event, the vector clock of the thread is simply incremented and copied to the event.

### 3.5.2 Online Consistent Global States Enumeration

In the second part, the online detector uses ParaMount to enumerate global states along with the execution of the concurrent program. When an event $e$ is captured, a callback function is triggered to insert $e$ into $\mathcal{P}$ and to enumerate $I(e)$. Since multiple events may occur concurrently, the intervals of global states are enumerated in parallel. By default, the bounded lexical algorithm is used as the subroutine of ParaMount.

Algorithm 6 shows the worker of ParaMount which is modified for the online predicate detection. In comparison with Algorithm 2, there are two differences. First, the worker in Algorithm 6 is instantiated for each interval of global states. Second, the poset $\mathcal{P}$ is not a complete poset because of the online detection. Hence, the events in $\mathcal{P}$ cannot be topologically sorted at this

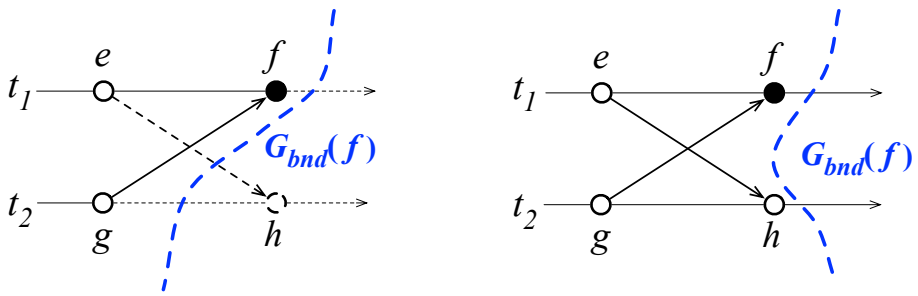



Figure 3.6: (a) One possible observed execution path when event $e$ is inserted into P. Suppose the insertion order is $e \rightarrow_p g \rightarrow_p f \rightarrow_p h$. (b) Another possible observed path, in where the insertion order is $e \rightarrow_p g \rightarrow_p h \rightarrow_p f$.

point. Therefore, we use the order of insertion into the data structure of $\mathcal{P}$ at line 2 as the total order $\rightarrow_p$. Specifically, the atomic block from line 1 to 5 ensures that the events are inserted sequentially. Furthermore, the injected callback function ensures that a thread cannot execute the next event until it has successfully inserted the current event into $\mathcal{P}$. Thus, Property 1 is achieved by the insertion order of the events.

At line 3, $G_{min}(e)$ is obtained from the vector clock of $e$, which is calculated using Algorithm 4. At line 4, $G_{bnd}(e)$ is determined by taking a snapshot of the maximal events of threads. Figure 3.6 shows an example of computing $G_{bnd}(f)$. In Figure 3.6(a), if the insertion order is $e \rightarrow_p g \rightarrow_p f \rightarrow_p h$, then the detector will not see $h$ when taking the snapshot for event $f$. Hence, our detector gets $G_{bnd}(f) = [2, 1]$. Figure 3.6(b) shows another example. If events $e$, $g$, and $h$ are inserted before event $f$, then it gets $G_{bnd}(f) = [2, 2]$. It is easy to see that the snapshot of maximal events satisfies the definition of

$G_{bnd}$ in Definition 1.

Since ParaMount allows multiple intervals of global states to be enumerated in parallel, we need to show that the combination of Algorithm 3 and Algorithm 6 can be executed concurrently.

**Theorem 3.** *Algorithm 3 and Algorithm 6 can be executed concurrently without violating correctness.*

*Proof.* The freedom from deadlock is obvious since the atomic block of Algorithm 6 can be implemented using one mutex with no wait inside the atomic block.

We now show that the execution of Algorithm 6 does not affect the concurrent executions of Algorithm 3. Suppose that Algorithm 3 is enumerating the global states corresponding to event $e$ and Algorithm 6 is concurrently inserting event $f$. Since Algorithm 3 stops at $G_{bnd}(e)$, it does not require the information on $f$. Moreover, the only modification of the poset $\mathcal{P}$ happens in the atomic block of Algorithm 6. Hence, there is no interference between the two algorithms. □

### 3.5.3 Predicate Evaluation

We use the predicate for detecting data races as an example, because the condition is easy to understand and requires little knowledge about the concurrent programs. A data race occurs when a pair of conflicting operations

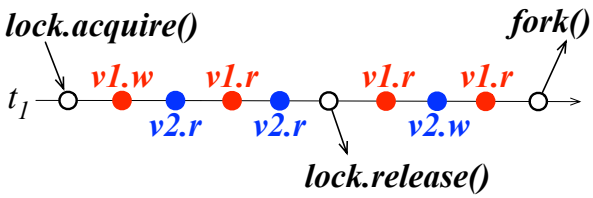
**Algorithm 7** `predicate`$(\mathcal{P}, G, e)$

**Input:** A global state $G$ and the new event $e$.
**Output:** the global state that contains data races.

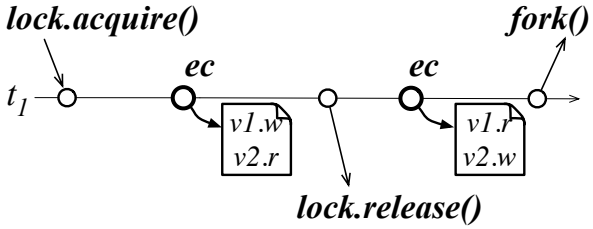
```
 1: if e.op = W then                                   ▷ e is a write event.
 2:    for i ← 1 to n : e' ← G[i] do
 3:       if (e'.op = W ∨ R) ∧ sameMemoryAddress(e, e') then
 4:          // a data race detected.
 5:       end if
 6:    end for
 7: else if e.op = R then                              ▷ e is a read event.
 8:    for i ← 1 to n : e' ← G[i] do
 9:       if (e'.op = W) ∧ sameMemoryAddress(e, e') then
10:          // a data race detected.
11:       end if
12:    end for
13: end if
```

(e.g., read-write or write-write operations) is executed concurrently by different threads on the same memory address.

Algorithm 7 detects data races when the current event $e$ is a write or a read event (line 1 and line 5). Assume that $e$ is a write event. Then the *for*-loop at line 2 gets the maximal event $e'$ of other threads. From the construction rules in Part I, two process-ordered events of different threads would not have direct HB relation. Therefore, any two process-ordered events in the frontier of global state can be executed concurrently. Subsequently, if events $e$ and $e'$ at line 3 are conflicting operations on the same memory address, then a data race has detected.




(a)




(b)

Figure 3.7: (a) The original process ordered events. (b) Only the first write or read event of a variable in a sequence of process ordered events is captured. Moreover, the events are merged into an event collection, *ec*.

### 3.5.4 Other Implementation Details

Our detector captures only the process-ordered events that are relevant to the predicate, which are the read and write operations of variables. Moreover, multiple consecutive process-ordered events, which are executed by the same thread, are merged into one *event collection*. Two process-ordered events are considered consecutive if there is no fork-join or lock atomicity event between them. The event collection only stores the first write operation of each variable. If there is no write operation for that variable, then its first read

**Algorithm 8** `predicateOnEventCollection`$(\mathcal{P}, G, e)$

**Input:** A global state $G$ and the new event $e$.
**Output:** the global state that contains data races.

```
 1: if e.op = W then                                   ▷ e is a write event.
 2:    for i ← 1 to n : ec ← G[i] do
 3:       for all e' ∈ ec do
 4:          if (e'.op = W ∨ R) ∧ sameMemoryAddress(e, e') then
 5:             // a data race detected.
 6:          end if
 7:       end for
 8:    end for
 9: else if e.op = R then                              ▷ e is a read event.
10:    for i ← 1 to n : ec ← G[i] do
11:       for all e' ∈ ec do
12:          if (e'.op = W) ∧ sameMemoryAddress(e, e') then
13:             // a data race detected.
14:          end if
15:       end for
16:    end for
17: end if
```

operation is stored. In addition, the events in the same event collection share the same vector clock.

Figure 3.7 shows an example. At the left side of Figure 3.7(a), thread $t_1$ performs a write and then a read operation on variable $v1$. In addition, it performs two read operations on variable $v2$. Then, our detector only inserts the first write event for $v1$ and the first read event for $v2$ into $\mathcal{P}$, as shown at the left side of Figure 3.7(b). The events in the event collection $ec$ will share the same vector clock and $ec$ is used as an event instance during the enumeration of global states. Algorithm 8 shows the modified predicate for

Table 3.1: The benchmarks for evaluating ParaMount.

| Benchmark | n | #events | #CGS |
|---|---|---|---|
| ***d-300*** | 10 | 300 | 42million |
| ***d-500*** | 10 | 500 | 237million |
| ***d-10K*** | 10 | 10,000 | 4,962million |
| ***bank*** | 8 | 96 | 815million |
| ***tsp*** | 8 | 10,528 | 13million |
| ***hedc*** | 12 | 216 | 4,486million |
| ***elevator*** | 12 | 38,528 | 27,643million |

the implementation. The loops at lines 2 and 7 retrieve the event collection on each thread, then the inner loops at lines 4 and 9 check whether the event collection contain any event that conflicts with the current event.

## 3.6 Evaluation

### 3.6.1 Experimental Results of ParaMount

Table 3.1 shows the benchmarks that are used in the experiment. The benchmarks with the prefix *"d-"* are randomly generated posets for modeling distributed computations. The benchmarks *bank*, *tsp*, *hedc*, and *elevator* are the posets that are generated from real-world concurrent programs. The benchmark *banking* is a toy program for demonstrating typical error patterns in concurrent programs [FNU03]; *tsp* is a parallel solver for the traveling salesman problem; *hedc* is a crawler for searching Internet archives; and *elevator* is a discrete event simulator for an elevator system. The benchmark programs *tsp*, *hedc*, and *elevator* are also used in [CSR08, FF09, vPG01]. Every program is run once and its execution trace is converted to a poset of events using

Table 3.2: The running time (seconds) of BFS algorithm and ParaMount.

| Benchmark | BFS | BPara(1) | BPara(2) | BPara(4) | BPara(8) |
|---|---|---|---|---|---|
| ***d-300*** | 47.0 | 35.9 | 19.4 | 10.6 | 6.9 |
| ***d-500*** | 380.8 | 195.4 | 100.5 | 54.5 | 33.6 |
| ***d-10K*** | 8,599.1 | 4,089.0 | 2,190.5 | 1,150.4 | 757.7 |
| ***bank*** | o.o.m. | 635.3 | 521.4 | 372.4 | 302.5 |
| ***tsp*** | 8.6 | 7.1 | 3.7 | 1.9 | 1.1 |
| ***hedc*** | o.o.m. | 10,850.7 | 10,182.2 | 8,032.5 | 4,646.9 |
| ***elevator*** | o.o.m. | 28,655.3 | 13,903.2 | 6,985.4 | 3,696.2 |

o.o.m.: Out of memeory.

the rules that had been discussed in the implementation section. Then the enumeration algorithm takes as input the poset and outputs the set of global states of that poset. The column "$n$" shows the number of threads or processes in the poset.

Now we evaluate the performance of ParaMount, whose subroutine uses bounded BFS algorithm (which is modified from the BFS algorithm [CM91] and denoted by B-Para) or bounded lexical algorithm (which is modified from the lexical algorithm in [Gan10, Gar03] and denoted by L-Para). Note that the BFS algorithm in [CM91] may enumerate the same global state multiple times. In this experiment, we have enhanced it with the technique mentioned in [Gar03], so the BFS algorithm and the subroutine of B-Para enumerates every global state exactly once.

Table 3.2 and Table 3.3 show the running times of the compared algorithms. The number of threads that are used by B-Para and L-Para are shown in the parentheses. All the experiments are conducted on a Linux ma-

Table 3.3: The running time (seconds) of the lexical algorithm and ParaMount.

| Benchmark | Lexical | LPara(1) | LPara(2) | LPara(4) | LPara(8) |
|---|---|---|---|---|---|
| ***d-300*** | 3.4 | 3.5 | 1.5 | 0.8 | 0.5 |
| ***d-500*** | 17.8 | 15.3 | 7.6 | 3.9 | 2.1 |
| ***d-10K*** | 406.8 | 327.3 | 163.4 | 105.0 | 43.1 |
| ***bank*** | 50.8 | 40.3 | 20.5 | 11.0 | 5.8 |
| ***tsp*** | 1.6 | 1.5 | 0.8 | 0.4 | 2.3 |
| ***hedc*** | 487.4 | 406.5 | 203.3 | 110.8 | 72.1 |
| ***elevator*** | 4,233.8 | 3,491.6 | 1,742.7 | 870.2 | 435.7 |

chine with an Intel Core i7 1.6 GHz CPU and the heap size of Java virtual machine is limited to 2GB. The running time is wall-clock time measured in seconds.

From Table 3.1, BFS algorithm has the worst performance because of its expensive time complexity. Moreover, it failed to finish almost half of the benchmarks because it ran out of the available memory (o.o.m.). The reason is that BFS algorithm has to store intermediate global states for future enumerations and the number of the intermediate global states might grow exponentially in the number of threads in the worst case. In B-Para, the benchmarks *bank*, *hedc*, and *elevator* are able to finish because the set of global states are partitioned into multiple small subsets; each of which induces much fewer number of intermediate global states and hence the consumed memory can be less than 2GB.

Partitioning the set of global states helps the performance of the original enumeration algorithm. Figure 3.8 show the speedup rate of B-Para with

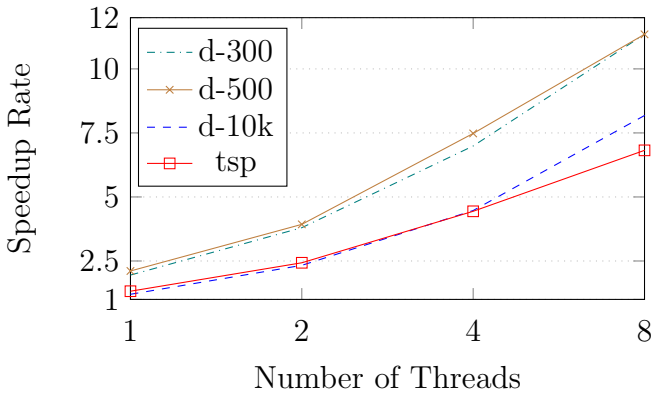



Figure 3.8: Speedup rate of B-Para with respect to the sequential BFS algorithm.

respect to the running time of BFS algorithm. The speedup rate on benchmarks *bank, hedc*, and *elevator* are not shown because BFS algorithm cannot finish the enumeration. When B-Para uses one single thread, its performance can be even faster than the original BFS algorithm. The reason is that BFS algorithm continuously triggers Java garbage collector to release the memory, which is used for storing the intermediate global states. In B-Para, the number of intermediate global states is reduced and hence the running time spent by Java garbage collector is significantly reduced. Moreover, B-Para can be up to 11 times faster than BFS algorithm when using 8 threads.

Figure 3.9 show the speedup rate of L-Para with respect to the sequential lexical algorithm. We show 4 of the benchmarks because the other benchmarks have the similar trend. For lexical algorithm, partitioning the set of global states still helps the performance for most benchmarks. When using one single thread, L-Para can reduce 20% of the running time in average.

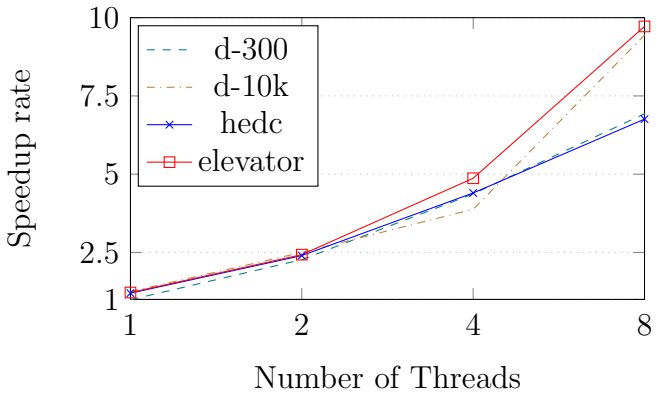



Figure 3.9: Speedup rate of L-Para with respect to the sequential lexical algorithm.

When using 8 threads, L-Para can be 6 to 10 times faster than the original lexical algorithm. The reason that ParaMount sometimes shows superlinear speedup is that partitioning the set of consistent global states transforms the original problem into multiple sub-problems that are much easier to solve.

Figure 3.10 shows the memory usage of lexical algorithm and L-Para. Since lexical algorithm is stateless, the memory is mainly used to store the poset, which is the input itself. Although ParaMount requires additional space to store $G_{min}(e)$ and $G_{bnd}(e)$ for each event $e$, the consumed memory is quite small. For most of the benchmarks, the memory usage of ParaMount is identical to that of the original enumeration algorithm.

### 3.6.2 Experimental Results of Online Predicate Detection

To evaluate the online property of ParaMount, we use it to implement an online-and-parallel predicate detector and then use the detector to detect

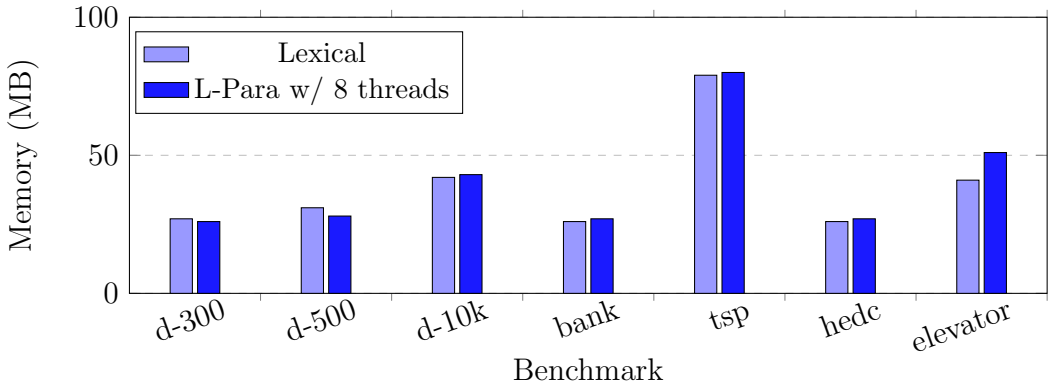



Figure 3.10: Memory usage of the lexical algorithm and L-Para.

data races in concurrent programs. In this experiment, the bounded lexical algorithm is used as the subroutine of ParaMount.

Table 3.4 shows the benchmarks that are used in the experiment. "LoC" shows the lines of code. "Thread" shows the number of threads that are used to drive each benchmark and ParaMount; after a thread executes an event, the thread is immediately used to enumerate the interval of global states. Thus, no additional threads are spawn for ParaMount. "#Var" shows the number of variables of the benchmark.

Besides the concurrent benchmarks that are used in previous experiment, we also use the following benchmarks. Benchmarks *set (faulty)* and *set (correct)* are incorrect and correct implementations of the concurrent set [HS08]; *arraylist1* is a non-thread-safe container and *arraylist2* is a thread-safe container from Java library; *sor* is a scientific computation application; and *raytracer* is a benchmark for measuring the performance of a 3D raytracer.

Table 3.4: The information of the benchmarks for data race detection.

| Benchmark | LoC | Thread | # Var |
|---|---|---|---|
| ***banking*** | 139 | 4 | 7 |
| ***set (faulty)*** | 223 | 4 | 10 |
| ***set (correct)*** | 260 | 4 | 10 |
| ***arraylist1*** | 1,474 | 4 | 6 |
| ***arraylist2*** | 1,377 | 4 | 16 |
| ***sor*** | 255 | 4 | 20 |
| ***elevator*** | 547 | 4 | 23 |
| ***tsp*** | 702 | 4 | 36 |
| ***raytracer*** | 1,885 | 4 | 77 |
| ***hedc*** | 25,027 | 8 | 345 |

The benchmarks *sor* and *raytracer* are also used in [CSR08, FF09, vPG01].

We compare our online-and-parallel predicate detector (denoted as ParaMount) with another general predicate detector, RV runtime [MR10], and an online race detector, FastTrack [FF09]. We chose RV runtime because it is the successor of jPredictor [CSR08] and it uses the notion of predicate detection. The enumeration algorithm that is used in RV runtime is the BFS algorithm [CM91]. We chose FastTrack because it is the fastest online race detector that uses the technique of vector clocks, even though its algorithm detects only data races. The input of each detector is a concurrent program and the output is a list of variables with data races.

The experimental results are shown in Table 3.5, in which the column "Base" shows the original execution time of the benchmarks. Each running time of ParaMount, RV runtime, and FastTrack includes the time to inject bytecode for monitoring, to execute the benchmark program, and to perform

Table 3.5: The result of data race detection.

| Benchmark | Running Time (ms) | | | | # Detection | | |
|---|---|---|---|---|---|---|---|
| | Base | Para-Mount | RV runtime | Fast-Track | Para-Mount | RV runtime | Fast-Track |
| ***banking*** | 3 | 72 | 32,000 | 40 | 1 | 1 | 1 |
| ***set (faulty)*** | 61 | 152 | 37,000 | 428 | 1 | 3 | 1 |
| ***set (correct)*** | 94 | 110 | 39,000 | 468 | 0 | 3 | 1 |
| ***arraylist1*** | 3 | 7 | exception | 29 | 3 | 4[a] | 3 |
| ***arraylist2*** | 4 | 5 | exception | 4 | 0 | – | 0 |
| ***sor*** | 19 | 81 | 41,000 | 179 | 0 | 0 | 0 |
| ***elevator*** | 16,000 | 16,000 | 83,000 | 16,000 | 0 | 0 | 0 |
| ***tsp*** | 7 | 114 | exception | 146 | 1 | – | 1 |
| ***raytracer*** | 32 | 1240 | o.o.m. | 998 | 1 | 0[b] | 1 |
| ***hedc*** | 241 | 940 | exception | 1,140 | 4 | – | 4 |

[a] Acquired before the exception is thrown.

[b] The field with data races is not shown in the candidate list of RV runtime.

predicate detection. In RV runtime, bytecode injection and predicate detection are performed in offline; and in both ParaMount and FastTrack, they are performed in online. The running time is wall-time measure in milliseconds. The benchmark *elevator* contains several `sleep()` function calls, which dominate the overall running time, so its running time is almost the same on different detectors; except the one on RV runtime. The numbers of the variables that have data races are also shown in the table.

On average, RV runtime takes 15 seconds to inject the monitor instruments into the benchmark programs. Without considering the running time of bytecode injection, RV runtime still requires 15 seconds or more to finish predicate detection for most of the benchmarks while our predicate detector is able to finish within one second. In the benchmark *raytracer*, RV runtime ran

out of the available memory because its BFS enumeration algorithm requires exponential memory space. Furthermore, RV runtime reported a false alarm on the benchmark – *arraylist1*. The reported variable is located in the test driver and its data race is benign; however, both our predicate detector and FastTrack can correctly rule out the variable. In *set (faulty)* and *set (correct)*, RV runtime reported several benign races. Moreover, it failed to detect the data race in *raytracer*. Currently, the results of RV runtime are not completely collected because the tool throws exceptions on some benchmarks.

When compared with FastTrack, the experiments show that ParaMount is as fast as FastTrack for most benchmarks even though its enumeration strategy is not specifically designed for detecting data races. In *set (faulty)* and *set (correct)*, the concurrent set uses a single linked list to store the data; the linked list is synchronized using a fine-grained hand-over-hand lock-mechanism [HS08]. Whenever a new data is added to the set, a node object of the linked list is created. In *set (faulty)*, the variable `next` of a node has data races because the variable will be illegally accessed when a thread is adding a new entry and another thread is removing an existing entry.

In *set (correct)*, the access of the variable `next` is always protected by a lock. However, the variable `next` is initialized without the protection of locks; consequently, FastTrack reports the variable even if it is well protected in subsequent accesses. In our implementation, we do not consider initialization events to ever cause the data race since no other thread can have reference to uninstantiated object or variable. In this manner we avoid reporting benign

Table 3.6: Comparisons of the predicate detectors.

| Detector | Type | Poset Construction | Global States Enumeration | Predicate Assumption |
|---|---|---|---|---|
| ***ParaMount*** | Online | 1-pass | Parallel | No assumption |
| ***RV runtime*** | Offline | 2-passes optimization | Sequential | No assumption |
| ***FastTrack*** | Online | 1-pass | No enumeration involved | Data races |

races due to initialization. The source code and the proof of the correctness of the benchmark *set (correct)* are available in [HS08].

Table 3.6 lists the properties of the detectors that are used in this experiment. RV runtime is an offline predicate detector and hence it can construct the poset of events in 2-passes. It first logs the event on the observed execution path and then uses a pre-processor to optimize the poset of events with respect to the property of the predicate. The construction method [FF09, LC06] used by FastTrack and ParaMount is 1-pass and hence is difficult to optimize; however, it can be used in an online fashion.

For enumeration of global states, RV runtime uses the BFS algorithm [CM91] to perform offline enumeration. The enumeration algorithm is general-purpose, which makes no assumptions on the nature of the predicate and guarantees that every global state is enumerated at least once. Unfortunately, the algorithm may enumerate the same global state multiple times. ParaMount is also general-purpose but it ensures that every global state is enumerated exactly once. FastTrack does not have any algorithm for global states enumer-

**Algorithm 9** `predicate2`$(\mathcal{P}, G, e_i)$

**Input:** A poset $\mathcal{P}$, a global state $G$, and the new event $e_i$.

```
 1: count ← 0
 2: if e_i is an event of transfer then
 3:    for i ≠ j : j ← 1 to n do
 4:       if G[j] is an event of transfer then
 5:          count ← count + 1
 6:       end if
 7:    end for
 8:    if count > capacity then
 9:       // the system may be overwhelmed.
10:    end if
11: end if
```

ation, because its detection method is particularly designed for data races.

## 3.7 Other Predicate Examples

Algorithm 7 detects the condition of the form $(e1 = a \wedge e2 = b)$, where $e1$ and $e2$ are events on two threads, and $a$ and $b$ are the conditions for the two events. Now we show a predicate of the form $(e1 + ... + en = a)$ in Algorithm 9. In the benchmark *banking*, the function `transfer` is invoked to transfer an amount of money between accounts. Suppose that the computational capacity of the user's system cannot handle more than a number, say *capacity*, of transfers at the same time. Therefore, a programmer has developed an algorithm using Java monitors or locks. To check if the algorithm works correctly, the programmer can insert an event before each invocation of `transfer` and sum up the number of invocations. Algorithm 9 defines the predicate for detecting

**Algorithm 10** `predicate3`$(\mathcal{P}, G, e_i)$

```
 1: count ← 1
 2: if e_i is an elevator event then
 3:    for i ≠ j : j ← 1 to n do
 4:       if G[j] is an elevator event ∧ G[j].floor = e_i.floor ∧ G[j].dir = e_i.dir
    then
 5:          count ← count + 1
 6:       end if
 7:    end for
 8:    if count > 2 then
 9:       // the predicate is detected.
10:    end if
11: end if
```

the condition.

As another example condition in the benchmark *elevator*, which simulates the elevator system of a building and each elevator is a thread. Suppose that a programmer has developed an algorithm to synchronize the elevators so that no more than three elevators, which are heading the same direction, stop at the same floor concurrently. To detect the predicate, the program can declare two more variables *dir* and *floor* in the event to log an elevator's direction and floor. When the elevator stops at any floor, the event is inserted into $\mathcal{P}$. Algorithm 10 defines the predicate, in which the *if* condition at line 4 checks for the condition.

# Chapter 4

# A Fast Enumeration Algorithm for Consistent Global States

In this chapter, we present QuickLex – a fast enumeration algorithm for consistent global state – and compare it with the BFS algorithm [CM91], the Tree algorithm [JMN95, HMNS01], and the original lexical algorithm [Gan10, Gar03]. The example computation and its corresponding lattice of consistent global states that are used in this chapter are shown in Figure 4.1 and Figure 4.2, respectively.

The first general-purpose enumeration algorithm for predicate detection is introduced by Cooper and Marzullo [CM91]. The algorithm uses a breadth first strategy (BFS) to enumeration consistent global states and requires $O(n^3)$ time and exponential space in $n$, where $n$ is the number of processes in the computation $\mathcal{P}$. The BFS algorithm enumerates the lattice of consistent global states one level at a time; the consistent global states at the same level of lattice consists the same number of events. When the algorithm enumerates the consistent global states in one level, it needs to store the consistent global

This chapter is previously published in [CG15b].

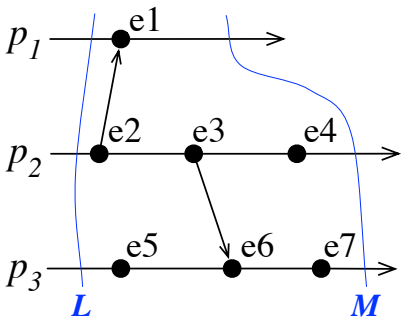



Figure 4.1: A poset $P$ of events corresponding to an execution of the program.

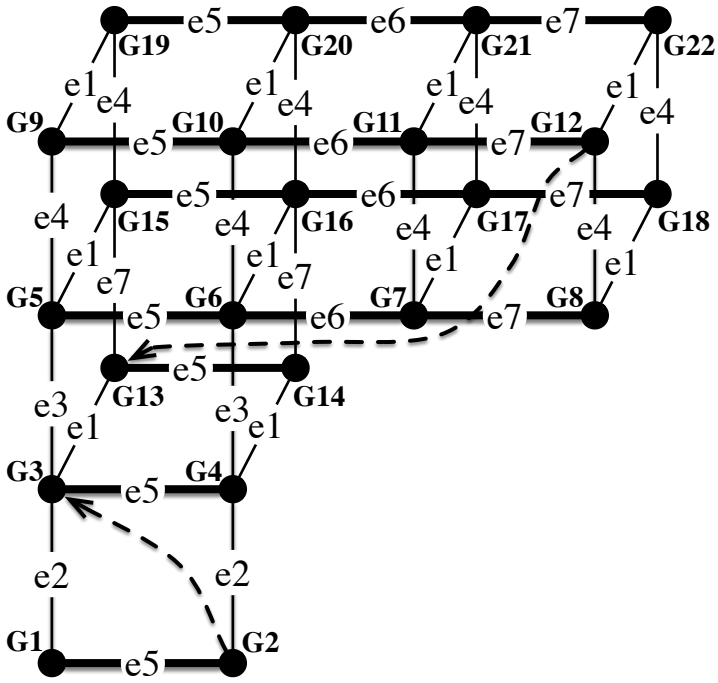



Figure 4.2: A lattice formed by the set of consistent global states of the poset shown in Figure 4.1.

states in the next level in a queue for future enumeration. In the worst case, the number of consistent global states at one level can be exponential in $n$.

Jegou et al. [JMN95] and Habib et al. [HMNS01] propose the Tree algorithm, which improves the space complexity of the first algorithm that enumerates consistent global states in a combinatorial Gray code manner [PR93]. The Tree algorithm uses $O(\Delta(\mathcal{P}))$ time, where $\Delta(\mathcal{P})$ is the maximal in-degree of any event in the computation. The algorithm first finds a backward spanning tree in the lattice of consistent global states, where the root is the global state that contains all events of the computation, e.g., the global state $G22$ shown in Figure 4.2; $G22$ is the global state [1, 3, 3] in Figure 4.1. Then, it traverses the spanning three in a depth-first manner. The Tree algorithm [JMN95,HMNS01] requires a stack to store the intermediate information regarding its spanning tree. The size of the stack is equal to the total number of events in the computation in the worst case. Hence, its space complexity is $O(|E|)$.

The lexical algorithm [Gan10, Gar03] explores the lattice of consistent global states using a pre-defined total order, called lexical order, among the consistent global states. The order $<_x$ is defined as follows:

$$G <_x G' \equiv \exists k : (\forall i : 1 \leq i < k : G[i] = G'[i]) \wedge (G[k] < G'[k]),$$

where $G$ and $G'$ are two arbitrary consistent global states in the lattice. In Figure 4.2, the lexical order of the two global states $G2 = [0, 0, 1]$ and $G3 = [0, 1, 0]$ is $G2 <_x G3$, where $k = 2$. The number of each global state in Figure 4.2 is its lexical order among the consistent global states in the lattice.

The lexical algorithm requires $O(n)$ space besides the input, i.e., the computation, because it avoids storing any intermediate consistent global state by exploiting the fact that the graph is a distributive lattice generated from the poset of the computation. The $O(n)$ space is only used for storing the vector clock that represents the current global state. Table 4.1 summarizes the time and space complexity of the above-mentioned enumeration algorithms.

In this chapter, we present QuickLex which reduces the time complexity of the original lexical algorithm [Gan10, Gar03] from $O(n^2)$ to $O(n \cdot \Delta(\mathcal{P}))$. The time complexity of QuickLex can be reduced to $O(n)$ for the commonly used model of computations [CSR08, LC06, FF09, HMNS01, JMN95], in which most events send and receive at most one message. Both QuickLex and Lex algorithms enumerate consistent global states in the same order. However, they are fundamentally different in computing the next consistent global state in the lexical order. The Lex algorithm simply uses the current consistent global state and vector clocks to determine the next consistent global state. Thus, it has to repeatedly calculate the information that is reusable. QuickLex reduces the computational cost using two approaches. First, it preprocesses the computation and pre-calculates the statically reusable information. Second, it incorporates dynamic programming to reuse the dynamic information during the enumeration. Although QuickLex uses $O(n^2)$ space for dynamic programming; however, the additional space is insignificant from our experimental results.

Table 4.1: Time and space complexity of the related enumeration algorithms.

| **Algorithms** | **Time per CGS** | **Space** |
|---|---|---|
| ***BFS*** [CM91] | $O(n^3)$ | exp. in $n$ |
| ***Tree*** [JMN95, HMNS01] | $O(\Delta(\mathcal{P}))$ | $O(\|E\|)$ |
| ***Lex*** [Gan10, Gar03] | $O(n^2)$ | $O(n)$ |
| ***QuickLex*** [CG15b] | $O(n \cdot \Delta(\mathcal{P}))$ | $O(n^2)$ |

$n$: the number of processes in the computation $\mathcal{P}$.
$E$: the set of events in $\mathcal{P}$.
$\Delta(\mathcal{P})$: the maximal in-degree of any event in $\mathcal{P}$.

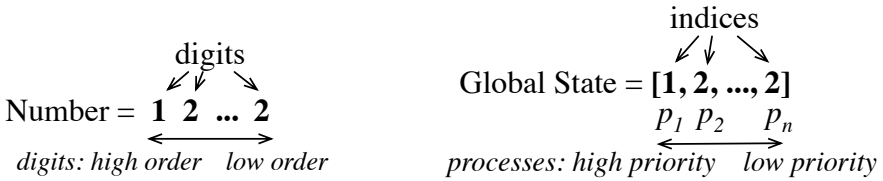



Figure 4.3: (a) A number consists of multiple digits. The digits at the left are high order digit and those at the right are low order digits. (b) A global state that is represented by an array of indexes. The array can be considered as a number and each index as a digit of that number. The processes whose indexes are located at the left are high priority processes and the processes whose indexes at the right are low priority processes.

## 4.1 Overview of QuickLex

In QuickLex, the array of indices of a global state is considered as a number and each index is a single digit of that number. Figure 4.3 shows the mapping between an array of indices and a number of digits. In a global state, the processes at the left are high priority processes and those at the right are low priority processes.

To advance from one global state to the other (which is also referred

**Algorithm 11** QUICKLEX($P$)

**Input:** A computation $P$ with $L$ as the least global state and $M$ as the greatest global state.

```
1: G := L                                   ▷ Use L as the initial global state.
2: for every event e in P do LOCATEREMOTEEVENTS(e)
3: INITIALSTACKS()
4: while true do
5:    enumerate(G)                          ▷ Evaluate the predicate on G.
6:    k := PROPAGATE(G, M)                  ▷ Find p_k to propagate.
7:    if k < 1 then break                   ▷ true: no process to propagate.
8:    G[k] := G[k] + 1                      ▷ Add the new event e_k into G.
9:    RESET(G, k)   ▷ Reset the maximal events of lower priority processes,
   i.e., p_{k+1} to p_n.
10: end while
```

as one *iteration*) in the lexical order, we use the notion of *carrying over* from arithmetic addition, in which we continuously add one to the low-order digit of a number and propagate the carry to a higher order digit that has not reached its limit. Then, all lower order digits are reset to their least value.

Similarly, QuickLex contains two main parts. The first part adds the next event of the least priority process $p_n$ into the current global state. If the next event of $p_n$ is not available (e.g., if the limit of the digit is reached), the carry is propagated to a higher priority process, say $p_k$. The second part resets the maximal events of lower priority processes, i.e., $p_{(k+1)}$ to $p_n$.

Algorithm 11 shows the pseudo code of QuickLex, which takes as input a computation $\mathcal{P}$. The least global state $L$ and the greatest global state $M$ of $\mathcal{P}$ are acquired from the computation itself and no additional calculation is needed. Take the computation in Figure 4.1 for example, where

$L = [0, 0, 0]$ and $M = [1, 3, 3]$. QuickLex enumerates every global state $G$ such that $L \leq_x G \leq_x M$. The function LOCATEREMOTEEVENTS at line 2 pre-calculates the reusable information for the PROPAGATE procedure. The function INITIALIZESTACK at line 3 initializes the memory space for dynamic programming, which speeds up the RESET procedure.

***Part 1 (lines 6-8):*** Informally, an event is *enabled* if it can be added into the current global state $G$ without violating the consistency of $G$. There might be multiple enabled events with respect to $G$. Since we enumerate global states in the lexical order, the PROPAGATE procedure locates the enabled event that occurs on the process that has the least priority, say $p_k$. If $k$ is 0, then the next global state has exceeded the maximal global state $M$ and hence the enumeration is terminated; otherwise, the enabled event is added into $G$.

Once $k$ is decided by the PROPAGATE procedure, the processes in the computation are divided into two sets: $P_h$ and $P_l$. The set $P_h$ of processes contains the processes whose priorities are higher or equal to process $p_k$, and $P_l$ contains those whose priorities are lower than $p_k$. In Figure 4.1, for example, if $k = 2$, then the set $P_h = \{p_1, p_2\}$ and the set $P_l = \{p_3\}$. From now on, the symbols $p_h$ and $p_l$ denote an arbitrary process in $P_h$ and $P_l$, respectively. In addition, the condition $h \leq k < l$ always holds.

***Part 2 (line 9):*** After part 1, the maximal events for $P_h$ are decided and fixed. Thus, we need to ensure that all the events of $P_l$ that happened before the events of $P_h$ are included in the next global state. We define the *maximum dependency event* of any process $p_l$ as the event, which has the

largest index among the events that occur on $p_l$, that has to be included in $G$ due to the consistency of the HB relation. The procedure RESET finds the maximum dependency event for every $p_l$.

The details of the first and second part of QuickLex are described next.

## 4.2 Part 1: Procedure PROPAGATE and Enabled Events

The procedure PROPAGATE determines the next enabled event to be included in the global state $G$ for lexical enumeration. We use the computation in Figure 4.1 and the lattice in Figure 4.2 to show how the procedure PROPAGATE works during an iteration of QuickLex. Assume that the current global state is $G2 = [0, 0, 1]$ and thus the next global state to be enumerated is $G3 = [0, 1, 0]$. The advancement from $G2$ to $G3$ is shown as a dashed arrow in Figure 4.2. First, event $e6$ is considered as the next event to be added into $G2$. However, $e6$ cannot be included in $G2$ because $e3 \rightarrow e6$ and $e3 \notin G2$, i.e., $e6$ is not enabled. Thus, the carry is propagated to $p_2$. Since event $e2$ is enabled, it is added to $G2$. Now, we have reached an intermediate global state $[0, 1, 1]$. In the second part of QuickLex, the maximal event $G[3]$ of $p_3$ will be reset to 0 and hence $G3 = [0, 1, 0]$ is reached.

**Definition 3.** *An event $e$ is enabled in a global state $G$ iff all events that happened before $e$ are included in $G$.*

Assuming that event $e$ occurs on process $p_i$, this condition can be de-

termined using the property of vector clocks [Fid88, Mat88]:

$$(e.vc[i] = G[i] + 1) \wedge (\forall j \neq i : e.vc[j] \leq G[j]).$$

Unfortunately, it takes $O(n)$ time to compare the vector clocks in the latter part of the condition. QuickLex reduces the time complexity by pre-calculating the *remote events* for each event and an event is enabled if all of its remote events have been included in the current consistent global state $G$.

Informally, if an event $r$ sends a message to an event $e$, $r$ is the remote event of $e$. Formally, an event $r$ is a *remote* event of event $e$ if 1) $r \rightarrow e$, 2) $r$ and $e$ occur on different processes, and 3) there does not exist any event $f$ such that $r \rightarrow f \rightarrow e$. If an event does not have any remote event, it is a local event. In Figure 4.1, for example, event $e6$'s remote event is event $e3$, and event $e6$'s remote event is event $e3$. Events $e2$, $e3$, $e4$, $e5$, and $e7$ are local events. Similarly, event $d$ is the *predecessor* of $e$ if 1) $d \rightarrow e$, 2) $d$ and $e$ occur on the same process, and 3) there does not exist any event $f$ such that $d \rightarrow f \rightarrow e$. In Figure 4.1, event $e6$'s predecessor is $e5$.

QuickLex uses the following theorem to reduce the time complexity to $O(\Delta(\mathcal{P}))$, where $\Delta(\mathcal{P})$ is the maximal number of remote events for any event:

**Theorem 4.** *Let $R(e)$ be the set of remote events of event $e$, which occurs on process $p_i$, and event $d$ be the predecessor of $e$, then $e$ is enabled iff $d \in G$ and $\forall r \in R(e) : r \in G$.*

*Proof.* ($\Rightarrow$): From Definition 3.

($\Leftarrow$): The proof is shown by the information of vector clocks. Assume that the predecessor $d$ of $e$ is included in $G$, we get $(e.vc[i] = G[i] + 1)$.

Since $d$ is included in $G$, we also get $\forall j : 1 \leq j \leq n : d.vc[j] \leq G[j]$ due to the property of vector clocks. Assume that all remote events of $e$ are also included in $G$, we get $\forall r \in R(e) : (\forall j : 1 \leq j \leq n : r.vc[j] \leq G[j])$. From the property of vector clocks, we get $(\forall j \neq i : e.vc[j] \leq G[j])$. As a result, $e$ is enabled when its predecessor and all remote events are included in $G$. $\square$

Theorem 4 reduces the computational cost of the procedure that determines whether event $e$ is enabled by ignoring the events that transitively happened before $e$. For example, if event $e$ is a local event, which does not have any remote event, then $e$ is enabled when its predecessor is included in $G$. In a computation $\mathcal{P}$, $\Delta(\mathcal{P})$ is at most $(n-1)$ because there are at most $(n-1)$ events that occur on different processes and send messages to $e$. If any event in $\mathcal{P}$ can have at most one remote event [CSR08, LC06, FF09, HMNS01, JMN95], then $\Delta(\mathcal{P})$ is $O(1)$. Note that the Tree algorithm [HMNS01, JMN95] also uses this assumption to reduce its time complexity.

Algorithm 12 shows a procedure which uses the property of vector clocks to locate the set $R(e)$ of remote events for any event $e$. The function has two steps. In the first step (lines 2-5), the vector clock of $e$ and that of $e$'s predecessor are compared. If the $i$-th value (except the one for $e$ itself) of $e$'s vector clock is updated, then a new HB relation is established between $e$ and *event(i, e.vc[i])*, which is the event, whose index is $e.vc[i]$, that occurs on

**Algorithm 12** Locate the set $R(e)$ of remote events for event $e$

**Input:** An event $e$ of the computation $P$.
**Output:** The set $R(e)$ of remote events for the event $e$.

```
 1: function LOCATEREMOTEEVENTS(e)
 2:     Let d be e's predecessor.

        ▷ The following loop finds the new HB relation on event e.
 3:     for i from 1 to n except e.pid do      ▷ e.pid is the id of the process on
    which e occurs.
 4:         if d.vc[i] ≠ e.vc[i] then Add event(i, e.vc[i]) into RCandidate.
 5:     end for

        ▷ The following loop finds the direct HB relation on event e.
 6:     for every r ∈ RCandidate do
 7:         Let r' be any other event in RCandidate.
 8:         if r.vc[r.pid] is larger than all r'.vc[r.pid] then Add r to R(e).
 9:     end for
10: end function
```

process $p_i$. However, we are interested in only direct HB relation because of Theorem 4. Thus, the second step (lines 6-9) uses another property of vector clocks: if event $r$ has not happened-before event $r'$, then the vector clock of $r'$ does not contain $r$'s latest clock value, i.e., $r.vc[r.pid]$, where $pid$ is the id of the process on which $r$ occurs. Note that Algorithm 12 is executed only once at the beginning of QuickLex and the calculated $R(e)$ for event $e$ is reused during the enumeration.

Algorithm 13 shows the procedure PROPAGATE. The procedure decides which process to propagate starting from the least to the highest priority processes in order to follow the lexical order. Moreover, the event that occurs

**Algorithm 13** Procedure PROPAGATE and Function ISENABLED

**Input:** The maximal global state $M$.
**Output:** The process $p_k$ to propagate.

```
1: procedure PROPAGATE(G, M)
2:    for k from n to 1 do                          ▷ From p_n to p_1.
3:       if G[k] + 1 ≤ M[k] then                    ▷ G + e_k ≤_x M
4:          e_k := the next event on process p_k.
5:          if ISENABLED(G, e_k) then return k
6:       end if
7:    end for
8:    return 0                                      ▷ No process to propagate.
9: end procedure
```

**Input:** The next event $e_k$ on process $p_k$.
**Output:** Returns *true* if $e_k$ is enabled w.r.t. $G$.

```
10: function ISENABLED(G, e_k)
11:    if e_k is a local event then return true
12:    if ∀r ∈ R(e_k) s.t. r.vc[r.pid] > G[r.pid] then return true ▷ r.pid is the
       id of the process on which r occurs.
13:    return false
14: end function
```

after the currently maximal event of process $p_k$ is chosen. Thus, the predecessor of $e_k$ is always included in $G$. The function isEnabled checks if either one of the following two conditions holds to determine whether $e_k$ is enabled: 1) $e_k$ is a local event or 2) all remote events of $e_k$ are included in $G$. If any event in the computation has at most one remote event, then isEnabled takes constant time. If $e_k$ is enabled, then PROPAGATE has found the process $p_k$ and it returns $k$. If the process $p_k$ does not exist, which implies that $M$ is reached, then PROPAGATE returns 0.

## 4.3 Part 2: Procedure RESET and Maximum Dependency Events

The maximal events of $P_l$ are not always reset to index 0. Assume that we are advancing from $G12 = [0, 3, 3]$ to $G13 = [1, 1, 0]$ in Figure 4.2. After PROPAGATE decides $k = 1$, we reach the intermediate global state $[1, 3, 3]$. However, we cannot simply reset the global state to $[1, 0, 0]$ because it is not consistent; it includes $e1$ but does not include $e2$ even though $e2 \rightarrow e1$ (see Figure 4.1). So, the procedure RESET has to find the *maximum dependency events* of $p_2$ and $p_3$ that would satisfy the consistency of the global state.

From now on, the symbol $G_m[l]$ denotes the maximum dependency event of $p_l$, which becomes the maximal event $G[l]$ of $p_l$ after RESET. When $e_k$ is decided, the maximal events of $P_h$ are also decided. The maximum dependency event $G_m[l]$ for every $p_l$ can be calculated using the property of

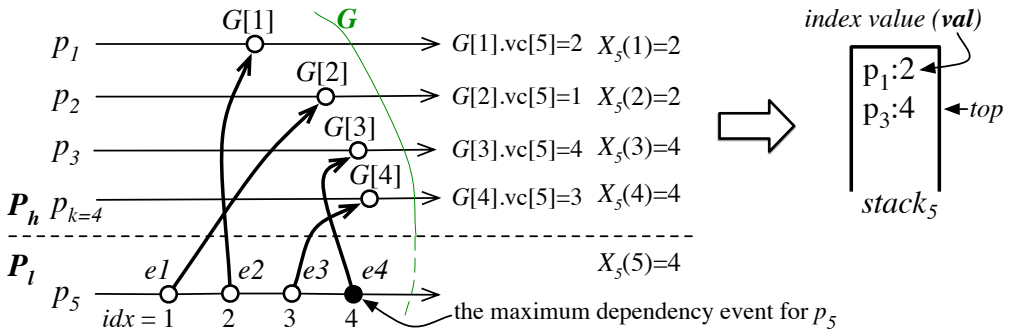



Figure 4.4: The symbol $X_l(i)$ denotes the function $max_{1\leq j\leq i}G[j].vc[l]$. The upside-down $stack_5$ on the right is the actual $stack_l$ that is used by QuickLex.

vector clocks:

$$G_m[l] = \max_{1\leq j\leq n}(G[j].vc[l])$$

For simplicity, we use the symbol $X_l(i)$ to denote the expression $\max_{1\leq j\leq i}(G[j].vc[l])$. We next use the computation in Figure 4.4 to explain how the maximum dependency event $G_m[l]$ of a process $p_l$ is identified by $X_l(n)$.

In Figure 4.4, the events $e1$, $e2$, $e3$, and $e4$ are four events that occur on process $p_5$. Assume that their indices are 1, 2, 3, and 4, respectively. Suppose that $k = 4$. Thus, $G[4]$ is the new event $e_k$. The fifth indices of the vector clocks of the maximal events of $p_1$, $p_2$, $p_3$, and $p_4$ are shown in the figure (i.e., $G[1].vc[5]$, $G[2].vc[5]$, $G[3].vc[5]$, and $G[4].vc[5]$). The bold arrows between events are the HB relations that are obtained from these indices. Since $G[3].vc[5]$ has the largest index, i.e., 4, it follows that $e4$ is the maximum dependency event of $p_5$. In other words, $G_m[5] = X_5(4) = 4$.

In fact, $G_m[l]$ can be identified by $X_l(k)$ instead of $X_l(n)$:

**Theorem 5.** *For a global state $G$, $k$, and any process $p_l$, $X_l(i) = X_l(k)$ for all $i > k$.*

*Proof.* Assume that the condition is not true, i.e., $\exists i : i > k : X_l(i) > X_l(k)$. The condition implies that $G_m[l] \rightarrow e_i$, which is an event that occurs on process $p_i$. Because $i > k$, we get $p_i \in P_l$ and thus $e_i \rightarrow G_m[i]$; so $e_i$ is included in $G$. Moreover, since $G_m[i]$ is a maximal dependency event, there exists an event $e_h$ such that $G_m[i] \rightarrow e_h$, where $e_h$ occurs on a process $p_h$, where $h \leq k$.

Due to the transitivity of HB relation, we get $G_m[l] \rightarrow e_i \rightarrow G_m[i] \rightarrow e_h$ and hence $X_l(h)$ also contains the largest value of $X_l(i)$. Since $h \leq k < i$, we get $X_l(h) = X_l(k) = X_l(i)$, which contradicts the assumption. □

According to Theorem 5, $X_l(k)$ has the largest clock value among $X_l(i)$ for all $i$. Consequently, $G_m[l]$ can be identified by $X_l(k)$. Now we show how to calculate the value of $X_l(k)$ in amortized constant time for each iteration using dynamic programming. It is easy to see that the value of $X_l(i)$ satisfies the following recursive equation:

$$X_l(i) = \begin{cases} G[1].vc[l], & \text{if } i = 1 \\ max\big(X_l(i-1), G[i].vc[l]\big), & \text{otherwise} \end{cases} \tag{4.1}$$

We use an auxiliary integer array $X_l$ for each process $p_l$, in which each value $X_l[i]$ stores the value of $X_l(i)$. Note that $X_l(i)$ is the value of $\max_{1 \leq j \leq i}(G[j].vc[l])$ and $X_l[i]$ is a calculated result. The array $X_l$ has to satisfy the invariant:

$$\forall i : 1 \leq i \leq n : X_l[i] = X_l(i)$$

**Algorithm 14** Incremental update of array $X_l$

**Input:** The process id of $p_l$, the decided $k$, and $\forall i : 1 \leq i \leq n :$ $X_l[i] = X_l(i)$ w.r.t. global state $F$.

**Output:** $\forall i : 1 \leq i \leq n : X_l[i] = X_l(i)$ w.r.t. global state $G$.

```
1: function updateArrayX(l, k)
2:     X_l[k] := max(X_l[k − 1], G[k].vc[l]))
3:     for i from (k + 1) to n do X_l[i] := X_l[k]
4: end function
```

For any global state $G$ and a given $k$, we can calculate the array $X_l$ for each process $p_l$ with respect to $G$. Assume that $F$ is the previous global state of $G$ in the lexical order. Instead of calculating the array $X_l$ for $G$ from scratch, we incrementally construct $X_l$ from that of $F$. The incremental update procedure is shown in the function updateArrayX in Algorithm 14.

**Theorem 6.** *Function* updateArrayX *maintains the invariant of $X_l$ after the incremental update.*

*Proof.* We consider the three intervals of the values in $X_l$:

(a) $i < k$: Since the maximal events of $P_h$ are not changed, the values of $X_l(i)$ for $i < k$ remain the same. Thus, updateArrayX does not need to update $X_l[i]$ for $i < k$.

(b) $i = k$: $X_l[i]$ is updated at line 2 using equation (4.1), where the value of $X_l(i-1)$ is obtained from $X_l[i-1]$.

(c) $i > k$: $X_l[i]$ is updated at line 3 using Theorem 5. □

**Algorithm 15** Initialize stacks for every process

```
1: function initializeStacks()
2:    for i from 1 to n do                    ▷ For every process p_i in P.
3:       push [p_1 : G[1].vc[i]] into stack_i
4:       for j from 1 to (i − 1) do           ▷ k < i is always true.
5:          if top.val < G[j].vc[i] then
6:                push [p_j : G[j].vc[i]] into stack_i
7:       end for
8:    end for
9: end function
```

### 4.3.1 Calculating Maximum Dependency Event in Amortized Constant Time

Since the results of $X_l$ are non-decreasing, we only need to store the values that are larger than their previous one and the process ids of the events that provide the values. For instance, $stack_5$ in Figure 4.4 is the actual stack (which is shown upside down) for storing the results of $X_5$. In $stack_5$, the top entry $[p_3 : 4]$ means $X_5[3] = X_5[4] = \cdots = X_5[n] = 4$ and the bottom entry $[p_1 : 2]$ means $X_5[1] = X_5[2] = 2$.

Algorithm 15 constructs the $stack_i$ of each process $p_i$ for the initial global state of a computation, which is $[0, 0, ..., 0]$. Although $k$ does not exist in the initial global state, we know that $k < i$ for each process $p_i$ because of the definition of $P_l$. Therefore, it is safe to assume that $k = (i - 1)$ when constructing $stack_i$. It is easy to see that the construction of $stack_i$ is equivalent to the construction of the array $X_i$. Moreover, the function updateArrayX in Algorithm 14 can be converted to the function updateStack in Algorithm

**Algorithm 16** Function UPDATESTACK and Procedure RESET

**Input:** The process id of $p_l$ and the decided $k$.
**Output:** The top value of $stack_l$ is $G[l]$.

```
 1: function UPDATESTACK(l, k)
 2:     pop stack_l until top.pid ≤ k.
 3:     if top.val < G[k].vc[l] then
 4:         if top.pid = k then top.val := G[k].vc[l]
 5:         else push [p_k : G[k].vc[l]] into stack_l
 6:     end if
 7: end function
```

**Input:** The decided $k$.
**Output:** The maximum dependency events of $P_l$ are found.

```
 8: procedure RESET(G, k)
 9:     for l from (k + 1) to n do
10:         UPDATESTACK(l, k)
11:         G[l] := top.val                    ▷ Set G[l] to G_m[l].
12:     end for
13: end procedure
```

16. Line 2 of UPDATEARRAYX is equivalent to lines 2-6 of UPDATESTACK and line 3 of UPDATEARRAYX is achieved by the property of $stack_l$.

**Theorem 7.** *$G_m[l]$ can be identified using $stack_l$ in an amortized constant time per global state.*

*Proof.* At line 2 of Algorithm 16, if $stack_l$ pops $m$ entries, then there exist $m$ iterations that cumulatively pushed $m$ entries into $stack_l$. Therefore, the cost of the pop operations can be evenly charged to the $m$ iterations and be reduced to amortized constant time. The operations at lines 4 and 5 take constant time. As a result, the time complexity for updating a stack is amortized constant time per global state. □

Finally, lines 8-13 of Algorithm 16 shows the procedure RESET, which updates $stack_l$ for every $p_l$. The maximum dependency event of $p_l$ is identified from the top entry of $stack_l$.

## 4.4 Correctness and Worst Case Time Complexity of QuickLex

We first show the correctness of QuickLex algorithm:

**Theorem 8.** *QuickLex enumerates the lattice of global states of a computation in the lexical order such that every global state is enumerated exactly once.*

*Proof.* Assume that $F$ is the previously enumerated global state and $G$ is the current global state to be enumerated.

**Lexical Order:** Since PROPAGATE adds a new event $e_k$ to $F$, we get $\exists k : (\exists i : 1 \leq i < k : F[i] = G[i]) \wedge (F[k] < G[k])$ and hence $F <_x G$.

**Exactly Once:** Since $F <_x G$, every global state is enumerated at most once. We next show that every global state is enumerated at least once. Since $F <_x G$, we get $\forall i : 1 \leq i < k : F[i] = G[i]$ and $G[k] = F[k]+1$. Assume that $F'$ is a consistent global state such that $F <_x F' <_x G$. We consider the following cases:

(a) $F'[k] < F[k]$: This case implies that $F' <_x F$, which contradicts the assumption $F <_x F'$.

(b) $F'[k] = F[k]$: Since $F <_x F'$, this case implies that there exists a process $p_{k'}$ such that $k' > k$ and $p_{k'}$ has an enabled event w.r.t. $F$. However, PROPAGATE locates the enabled event from $p_n$ to $p_1$ and hence $k' \leq k$. A contradiction.

(c) $F'[k] = F[k]+1 = G[k]$: After RESET, any $p_l$ cannot have a maximal event that is smaller than its maximum dependency event $G_m[l]$ due to the consistency of the HB relation. Thus, we get $\nexists l : F'[l] < G[l] = G_m[l]$. So, $F'$ does not exist.

(d) $F'[k] > F[k] + 1 = G[k]$: This case implies that $G <_x F'$, which contradicts the assumption $F' <_x G$. □

We next calculate the time complexity of QuickLex algorithm:

**Theorem 9.** *The worst case time complexity of QuickLex is $O(n \cdot \Delta(\mathcal{P}))$ per global state.*

Table 4.2: The information of benchmarks and runtimes (sec.) of the compared algorithms.

| Benchmark | $n$ | **#events** | **#consistent global states** | **BFS** | **Tree** | **Lex** | **QuickLex** |
|---|---|---|---|---|---|---|---|
| ***d-300*** | 10 | 300 | 42,695,907 | 58.43 | 3.80 | 3.41 | 0.76 |
| ***d-500*** | 10 | 500 | 237,475,992 | 375.06 | 19.40 | 18.67 | 3.78 |
| ***d-10K*** | 10 | 10,000 | 4,962,876,973 | 8,211.87 | 393.74 | 448.28 | 86.38 |
| ***bank*** | 8 | 96 | 815,730,721 | o.o.m. | 56.67 | 64.37 | 9.69 |
| ***tsp*** | 8 | 105,282 | 13,474,170 | 9.85 | 1.63 | 2.37 | 0.37 |
| ***hedc*** | 12 | 216 | 4,486,599,595 | o.o.m. | 322.04 | 488.22 | 78.34 |
| ***elevator*** | 12 | 38,528 | 27,643,588,608 | o.o.m. | 2,248.39 | 4,677.12 | 660.40 |
| ***w-4*** | 4 | 480 | 9,381,251 | 2.51 | 0.88 | 0.38 | 0.16 |
| ***w-8*** | 8 | 480 | 7,392,009,768 | o.o.m. | 609.74 | 454.28 | 128.03 |
| ***w-12*** | 12 | 480 | 206,379,406,870 | o.o.m. | 19,225.98 | 21,303.66 | 3,996.17 |
| ***w-16*** | 16 | 480 | 991,493,848,554 | o.o.m. | 111,452.52 | 179,844.62 | 23,263.05 |

o.o.m.: Out of memeory.

*Proof.* There are two main procedures during each iteration of QuickLex: PROPAGATE and RESET. We first analyze the worst time complexity of PROPAGATE. Each invocation the function ISENABLED takes $O(\Delta(\mathcal{P}))$ time and the *for* loop of PROPAGATE is executed at most $n$ iterations. So, the worst time complexity of PROPAGATE is $O(n\cdot\Delta(\mathcal{P}))$ time.

We now analyze the worst case time complexity of RESET. Each invocation of the function UPDATESTACK takes amortized $O(1)$ time and the *for* loop of RESET is executed at most $n$ iterations. So, the worst case time complexity of RESET is amortized $O(n)$ time. As a result, the worst time complexity of each iteration of QuickLex is $O(n\cdot\Delta(\mathcal{P}))$. □

## 4.5 Evaluation

Table 4.2 shows the information of the benchmarks that are used in the experiments. The benchmarks contain three different sets of computations. The benchmarks that start with the prefix "*d-*" are randomly generated posets of events for modeling distributed computations. The benchmarks *bank*, *tsp*, *hedc*, and *elevator* are the computations that are captured from the executions of real-world concurrent applications. We establish the HB relation in these concurrent computations using the following rules that are defined in chapter 2.

The benchmark *banking* is a toy program for demonstrating typical error patterns in concurrent programs [FNU03]; *tsp* is a parallel solver for the traveling salesman problem; *hedc* is a crawler for searching Internet archives; and *elevator* is a discrete event simulator for an elevator system. The benchmarks *tsp*, *hedc*, and *elevator* are the benchmark programs that are used in [CSR08, FF09, vPG01].

Finally, the benchmarks that start with the prefix "*w-*" have the same number of events, i.e., 480 events, but different number of processes in the computation. The set of benchmarks is used to show how different $n$ influences the performance of enumeration algorithms, and therefore we keep the number of events constant.

### 4.5.1 Improvements to the Related Enumeration Algorithms

Besides QuickLex, we implemented the breadth-first strategy (BFS) algorithm [CM91, Gar03], the ideal tree traversal algorithm (Tree) [JMN95,

HMNS01], and the original lexical algorithm (Lex) [Gan10, Gar03]. In BFS algorithm [CM91], a global state might be enumerated more than once, so we use the strategy in [Gar03] to ensure that every global state is enumerated exactly once. In our experiments, we use the improved BFS algorithm.

For Lex [Gar03], we improve the nested *for* loops of function LEASTGLOBALSTATE(). Each of the *for* loop goes through process $p_1$ to process $p_n$, which takes $O(n^2)$ time. However, looping through all processes is not necessary. We modify the first loop, which only loops from $p_1$ to $p_k$, and the second loop, which only loops from $p_{k+1}$ to $p_n$. In other words, the Lex algorithm incorporates Theorem 5 but not Theorem 6. Although the time complexity remains the same, the practical runtime is improved significantly. In our experiments, we use the improved Lex algorithm.

The Tree algorithm [HMNS01, JMN95] finds a backward spanning tree in the lattice of global states, where the root is the global state that contains all events, e.g., the state $G22$ that is shown in Figure 4.2. Then it traverses the spanning three in a depth-first manner. The performance of Tree mainly dependents on the data structure *SList* [JMN95], which is a customized linked list that continuously adds and removes the nodes of the spanning tree. So, we use the following implementation techniques to improve its performance:

- First, we calculate the least number of nodes that is required by *SList* during the enumeration. Then, we pre-allocate all the nodes in an object pool, which is implemented using an array, and reuse the nodes through

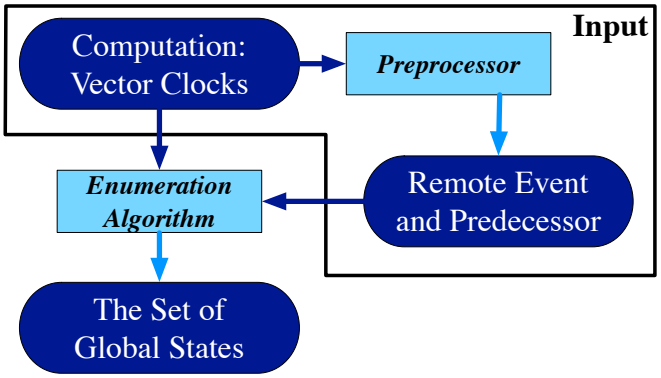



Figure 4.5: The setup of the experiment.

the enumeration procedure.

- Second, each node of *SList* has a counter that has to be updated and there are $\Delta(\mathcal{P})$ nodes that need to be updated in each iteration. We replace the counter with a timestamp, which achieves the same functionality but only needs to be set once and requires no further updates. Hence, the cost of the update is reduced from $O(\Delta(\mathcal{P}))$ time to constant time.

From our empirical observations, the two implementation enhancements have reduced approximately 50% of the original running time and 90% of the original memory usage. In our experiments, we use the improved Tree algorithm.

### 4.5.2 Experimental Results

Figure 4.5 shows the setup of the experiment. The input of the Tree, BFS and QuickLex algorithms is the vector clocks of the events in the computation. For the Tree and QuickLex algorithm, the information about the remote and the predecessor event for every event is extracted from vector

clocks [Fid88, Mat88] in a preprocessing stage, which is performed once for each benchmark. The time complexity of the extraction is far smaller than $O(i(\mathcal{P}))$, so the runtime can be omitted in comparison with that of enumeration algorithms. Afterwards, the enumeration algorithm outputs the set of consistent global states of the computation.

The input of the compared algorithms is the vector clocks of the events in the computation and the output is the set of global states of the computation. Table 4.2 also shows the experimental results. All the experiments are conducted on a Linux machine with an Intel Xeon 2.67GHz CPU and the heap size of Java virtual machine is limited to 2GB. The runtime is measured in seconds. As it can be seen, BFS algorithm has the worst performance because of its high time complexity. Moreover, it failed to finish on more than half of the benchmarks because it ran out of the available 2GB memory. The reason is that it has to store intermediate global states for future iterations and the number of intermediate global states might grow exponentially in $n$ in the worst case.

We first compare the runtimes of Tree, Lex, and QuickLex in the first and second set of benchmarks. Figure 4.6 shows the normalized runtimes of each algorithm with respect to the runtime of Tree. We normalized the runtimes to those of Tree because it has an amortized time complexity of $O(1)$ per global state and the smallest theoretical time complexity among the existing enumeration algorithms. From Figure 4.6, QuickLex is approximately 7 times faster than Lex and consistently 4–5 times faster than Tree. One reason

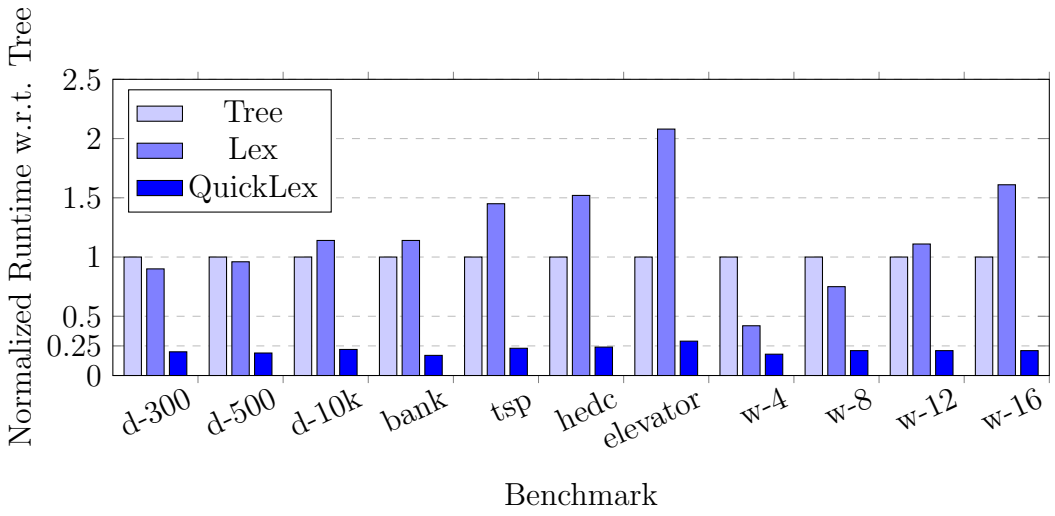



Figure 4.6: Normalized runtime of each algorithm w.r.t. the runtime of Tree algorithm.

that Tree is not as fast as QuickLex is that its intermediate information has to be stored in a linked list and therefore the cost of accessing the information is high.

We now compare the runtimes of Tree, Lex, and QuickLex in the third set of benchmarks; the benchmarks that start with the prefix "*w*-". From Figure 4.6, we can see that the normalized runtimes of Lex increase as the number of processes increases. On the other hand, the normalized runtimes of QuickLex are consistently 4 times faster than those of Tree, which shows that the time complexity of QuickLex can achieve amortized $O(1)$ per global state in practice.

We now explain how QuickLex achieves amortized $O(1)$ time per global state in practice. Suppose that any event in the computation can have at most

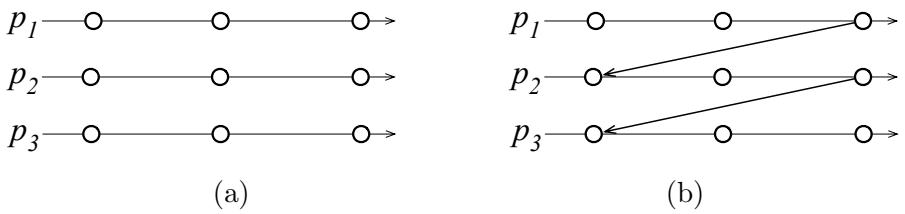



Figure 4.7: (a) The best case for QuickLex. (b) The worst case for QuickLex.

one remote event, then the worst time complexity of PROPAGATE is $O(n)$ per global state. Recall that each call of PROPAGATE runs through $(n - k + 1)$ processes before returning $k$. If there exist more than $(n-k+1)$ global states between current and most recent PROPAGATE call that returns the same $k$, then the cost of current PROPAGATE call can be charged to the iterations between these two PROPAGATE calls, which cumulatively enumerated $(n-k+1)$ global states. Thus, the current PROPAGATE call is amortized to $O(1)$.

Figure 4.7(a) illustrates the explanation. Assume that the cost of a PROPAGATE call is $c$ if the *while* loop of PROPAGATE executes $c$ iterations. For instance, the cost of a PROPAGATE call that returns $k = 2$ is 2. However, QuickLex has enumerated 4 global states, e.g., $[0, 0, 0]$, $[0, 0, 1]$, $[0, 0, 2]$, and $[0, 0, 3]$, between any two PROPAGATE calls that return $k = 2$. Consequently, the additional cost of the current PROPAGATE call, which returns $k = 2$, can be evenly charged to 5 global states, including the current one. Similarly, there are 17 global states for any PROPAGATE call that returns $k = 1$ to share the additional cost. As a result, the time complexity of any PROPAGATE call can be amortized to $O(1)$ time per global state. The same reason holds for the

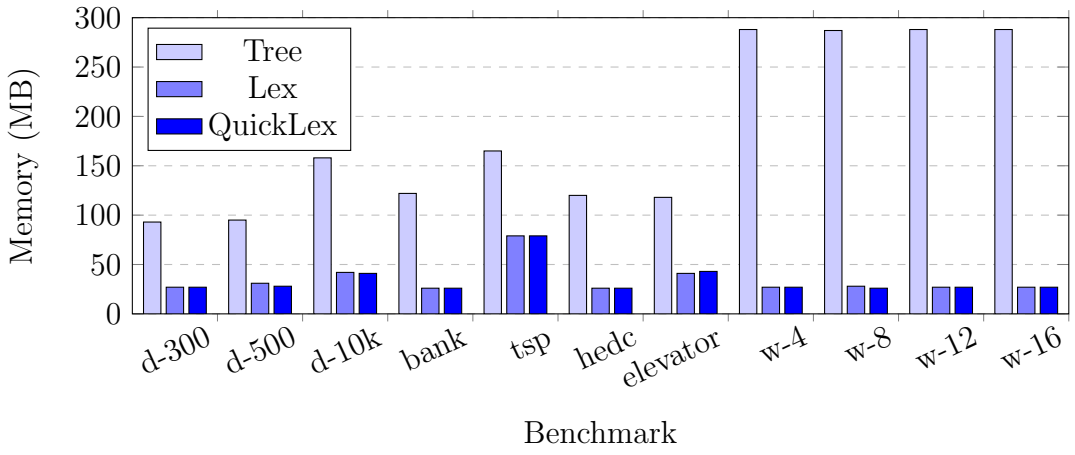



Figure 4.8: Memory usage of Tree, Lex, and QuickLex algorithm.

time complexity of RESET.

Figure 4.7(b) shows the worst case for QuickLex, in which only one global state exists between PROPAGATE calls. Therefore, the cost cannot be amortized and hence PROPAGATE takes $O(n)$ time. The events in this computation are totally ordered, which is not a common computation.

Figure 4.8 shows the memory usage of the compared enumeration algorithms. Since Lex is stateless, its memory is mainly used for storing the input, i.e., the computation. From Figure 3.10, QuickLex uses almost the same amount of memory even though QuickLex requires additional $O(n^2)$ space to store the stacks for dynamic programming. The $O(n^2)$ space is quite small because the space only stores integers. Tree, however, consumes much more memory space than Lex and QuickLex because it needs to store the information regarding its backward spanning tree, whose size is linear to $O(|E|)$. Note

Table 4.3: The performance of ParaMount with different enumeration algorithms.

| Benchmark | Information | | | Runtime (ms) | | # Detection |
|---|---|---|---|---|---|---|
| | LoC | Thread | #Var | Lex | QuickLex | |
| ***banking*** | 139 | 4 | 7 | 72 | 20 | 1 |
| ***set (faulty)*** | 223 | 4 | 10 | 152 | 69 | 1 |
| ***set (correct)*** | 260 | 4 | 10 | 110 | 51 | 0 |
| ***arraylist1*** | 1,474 | 4 | 6 | 19 | 19 | 3 |
| ***arraylist2*** | 1,377 | 4 | 16 | 22 | 15 | 0 |
| ***sor*** | 255 | 4 | 20 | 81 | 25 | 0 |
| ***tsp*** | 702 | 4 | 36 | 114 | 42 | 1 |
| ***raytracer*** | 1,885 | 4 | 77 | 1240 | 236 | 1 |
| ***hedc*** | 25,027 | 8 | 345 | 940 | 335 | 4 |

that $|E|$ is much larger than $n^2$ in practice.

## 4.6 Applications of QuickLex

### 4.6.1 Predicate Detection in Concurrent Systems

In this section, we compare the performance of Lex and QuickLex in real-world applications. In Chapter 3, we implemented a predicate detector, named ParaMount, for concurrent programs. The detector captures the execution trace of users' program using Java bytecode injection. The captured execution trace is converted to a concurrent computation using the methods discussed in [FF09, LC06]. In short, the detector captures 1) the read and write operations of all variables, 2) the causal dependency of fork-and-join operations of thread, and 3) the causal dependency of the acquisition-and-release operations of locks (including implicit locks and monitors) in users' program. The causal dependency is represented by HB relation in the computation.

ParaMount uses a sequential enumeration algorithm (e.g., Lex or Quick-Lex) as the subroutine to enumerate the set of global states in an online-and-parallel fashion. During the enumeration, each global state is checked for the predicate corresponding to data races. A data race occurs when conflicting operations (i.e., a pair of read-write or write-write operations) are concurrently executed on the same memory address by different threads. In summary, the detector takes as input a program and outputs the variables that have data races.

Table 4.3 shows the result of the detection. "LoC" shows the lines of code of the benchmark program. "Thread" shows the number of threads that are used to drive each benchmark. "#Var" shows the number of variables of the benchmark. Every variable is checked if it is accessed by different threads without the protection of any lock. Besides the four real-world applications that are used in Section 4.5, we also use the following applications. The benchmarks *set (faulty)* and *set (correct)* are incorrect and correct implementations of the concurrent set [HS08]; *arraylist1* is a non-thread-safe container and *arraylist2* is a thread-safe container from Java library; *sor* is a scientific computation application; and *raytracer* is a benchmark for measuring the performance of a 3D raytracer. The benchmarks *sor*, *tsp*, *raytracer*, and *hedc* are also used in [CSR08, FF09, vPG01].

The running time of ParaMount includes the time to inject bytecode for monitoring, to execute the benchmark program, to capture the executed events, to enumerate global states, and to evaluate the predicate of data races.

The column "Lex" shows the original execution time of ParaMount using the Lex as its subroutine and column "QuickLex" shows the improved execution time. On average, QuickLex improves the execution time of ParaMount by a factor of 3. "#Detection" shows the number of variables that have data races; all the detected variables are also detected by [CSR08,FF09], so the results do not have false positives.

### 4.6.2 Other Applications of QuickLex

In [Gar06, Gar15], it has been shown that many families of combinatorial objects can be mapped to the lattice of global states of appropriate posets. Thus, lexical traversal that is discussed in this dissertation can also be used to efficiently enumerate all subsets of $[n]$, all subsets of $[n]$ of size $m$, all permutations, all permutations with a given inversion number, all integer partitions less than a given partition, all integer partitions of a given number, and all $n$-tuples of a product space.

# Chapter 5

# A Model for Computations with Locking Constraints

In this chapter, we introduce a new model called Loset (Locking Poset) for modeling parallel computations with locking constraints.

The poset model [Lam78] originally does not consider the constraints due to locks, one common modification to the model is to capture the real-time order of lock synchronizations as the causality of the program [FF09, LC06, CG15a,CSR08]. With the modification, the results of predicate detection using the poset model do not have false positives [FF09] (assuming that process and thread scheduling is the only source of nondeterminism in the program). However, the detection may miss the predicate if it does not become true in the locking schedule that is captured by the current poset.

In this chapter, we argue that the synchronization due to locks is fundamentally different from the potential causality. We present an alternative model that makes a distinction between the happened-before relation and the synchronization of locks. Informally, a Loset is a Poset augmented with the notion of locks and locking intervals. In a loset, synchronization due to locks are not modeled using the happened-before relation. Instead, the intervals of

events that are executed under one or more locks are modeled separately. If two locking intervals $I1$ and $I2$ are executed under the same lock, then it is understood that events in $I1$ and $I2$ cannot be interleaved but they 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.

In the following section, we give the formal definition of the loset model.

## 5.1 Loset Model of a Computation

A Loset (Locking Poset) of events represents the computation that is captured from the execution of parallel programs. Formally, a Loset is defined as follows.

**Definition 4** (Loset)**.** *A loset $\mathcal{L}$ is a six-tuple $\mathcal{L} = (E, \rightarrow, n, L, pid, \mathcal{I})$ 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,*
- *$pid$: is a partition of $E$ to $E_1, E_2, \cdots, E_n$ such that each of $E_i$ is totally ordered, i.e. for all distinct $e, f \in E_i : (e \rightarrow f) \vee (f \rightarrow e)$. For convenience, we define the process order relation (denoted by $\prec$) such that $e \prec f$ means $e \rightarrow f$ in some $E_i$.*

- $\mathcal{I}$*: is a set of locking intervals, where each locking interval* $I$ *is a four-tuple* $I = (t, l, acq, rel)$ *where* $t \in \{1..n\}, l \in \{1..L\}$, $acq, rel \in E_t$, *and* $acq \prec rel$.

The locking interval $I = (t, l, acq, rel)$ denotes that thread $I.t$ acquired the lock $I.l$ at event $I.acq$ and released it at event $I.rel$. The relation $\rightarrow$ 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 usual Lamport's happened-before (HB) relation [Lam78]. In concurrent systems, we may have additional order constraints due to the *Fork-Join* events of threads and the *Wait-Notification* events of conditional synchronizations [FF09, LC06, CG15a, CSR08]. In the rest of this chapter, we assume that the HB relation between events is traced using vector clocks [Fid88, Mat88].

Note that the objective of the HB relation is to capture the causality of events but not the real-time locking order between the acquisition and release events of locks. Formally,

**Definition 5** (Valid Poset of a Loset)**.** *A poset* $P = (E, \rightarrow_P)$ *is a* valid poset *of a loset* $\mathcal{L} = (E, \rightarrow, n, L, pid, \mathcal{I})$ *if* $(\rightarrow \subseteq \rightarrow_P)$ *and for all* $I, J \in \mathcal{I}$ *such that* $I.l = J.l$, *we have* $(I.rel \rightarrow_P J.acq) \vee (J.rel \rightarrow_P I.acq)$.

Informally, the intervals for the same lock in loset $\mathcal{L}$ are totally ordered in the poset $\mathcal{P}$. For instance, the loset in Figure 5.1(c) is equivalent to the two valid posets in Figure 5.1(a) and Figure 5.1(b). In Figure 5.1(d), suppose that

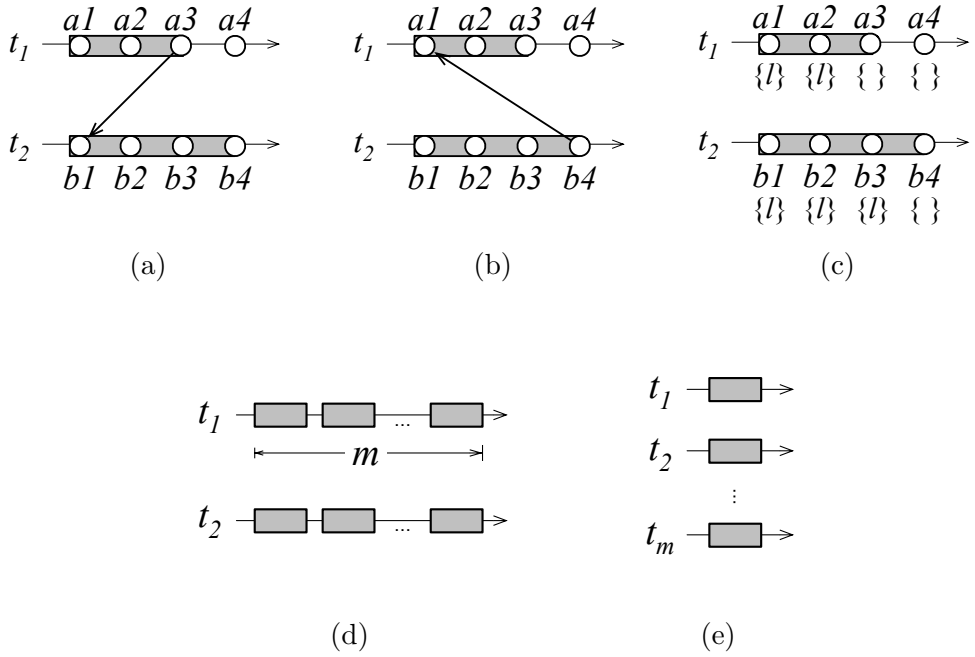



Figure 5.1: (a) and (b) Two posets that are captured from different executions of the same program. (c) The loset that is equivalent to the two posets in (a) and (b). (d) A loset that is equivalent to $C_m^{2m}$ posets. (e) A loset that is equivalent to $m!$ posets.

each thread contains $m$ locking intervals for the same lock, then the loset is equivalent to $C_m^{2m}$ valid posets because $m$ intervals of $t_1$ can be interleaved with $m$ intervals of $t_2$ in any order. Similarly, the loset in Figure 5.1(e) is equivalent to $m!$ valid posets. Figure 6.3 shows a more complex loset.

### 5.1.1 Global States

We next define the notion of a global state of a loset.

**Definition 6** (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. 5.1(c), $\{a1, a2, b1\}$ is a global state, but $\{a2, b1\}$ is not a global state because it contains $a2$ 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 $G \preceq H$ between the two global states means $G[i] \preceq H[i]$ holds for any thread $i$.

**Definition 7** (Consistent Global States)**.** *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. Next, we ensure that the global state also respects the locking

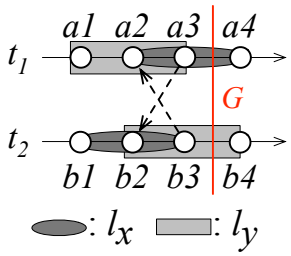



Figure 5.2: The global state $G$ is feasible but not reachable.

constraints. We define the set $\text{EL}(e)$ of **effective locks** for any event $e$, which are the locks being held by the thread that has executed $e$:

**Definition 8** (Effective Locks)**.** $\text{EL}(e) = \{I.l \mid I.acq \preceq e \prec I.rel\}$.

In Figure 5.1(c), 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.

**Definition 9** (Compatible Global States)**.** *A global state $G$ is (lock) compatible iff for any $i \neq j, G[i]$ and $G[j]$ are pairwise (lock) compatible, i.e.,* $\text{EL}(G[i]) \cap \text{EL}(G[j]) = \varnothing$.

Finally, the feasibility of a global state is defined as follows:

**Definition 10** (Feasibility)**.** *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. Therefore, only feasible global states

are reachable from the initial global state. However, not all feasible global states are reachable. In Figure 5.2, for example, the global state $G$ is feasible but not reachable. In $G$, the thread $t_1$ holds the lock $l_x$. Therefore, $t_2$ has to release $l_x$ before $t_1$ acquires $l_x$ and thus we get the inferred locking order $b3 \rightarrow a2$. Similarly, the thread $t_1$ has to release $l_y$ before $t_2$ acquires $l_y$ and thus we get the inferred locking order $a3 \rightarrow b2$. This results in a cycle in the $\rightarrow$ relation: $a3 \rightarrow b2 \rightarrow b3 \rightarrow a2 \rightarrow a3$. Hence, $G$ is unreachable.

### 5.1.2 Reachable Global States and Runs

We first introduce a sequence of events called 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 any global state $G$. The symbol $\mathcal{R}^i$ denotes the prefix of $\mathcal{R}$ that contains $i$ events. Since only feasible states are reachable in a loset, we require that a *run* go through only feasible global states. Formally, a *run* $\mathcal{R}$ is defined as follows:

**Definition 11** (Run)**.** *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 $0 \leq i \leq |\mathcal{R}|$.*

The *reachability* of a global state $G$ (from the initial global state $\phi$) is defined as follows:

**Definition 12** (Reachability)**.** *A global state $G$ is reachable from $\phi$ iff there exists a run $\mathcal{R}$ such that $\delta(\phi, \mathcal{R}) = G$.*

The reachability problem is defined as:

**Definition 13** (Reachability Problem)**.** *Given a loset $\mathcal{L}$ and a global state $G$, is $G$ a reachable global state of $\mathcal{L}$?*

**Theorem 10.** *The reachability problem of any global state $G$ in a loset $\mathcal{L}$ is NP-complete.*

*Proof.* The reachability problem is in NP because given a global state $G$ 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 [Tar00], 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 $\Phi$ remains true after the control sequence is added to the computation $\mathcal{P} = (E, \rightarrow)$. In other words, the control sequence adds additional $\rightarrow$ relations to $\mathcal{P}$ such that the critical sections for the same lock are totally ordered. The new computation, say, $\mathcal{Q}$, cannot contain any cycle of the $\rightarrow$ relation. In addition, every consistent global state $G$ of $\mathcal{P}$ such that $\Phi$ 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." Consequently, 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 [Tar00] 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 reachability problem of a loset. □

## 5.2 Valid Losets

Since we use the loset model for analyzing parallel computations, we are interested only in those losets that capture a possible execution from a real-world application.

**Definition 14** (Valid Loset)**.** *A loset is valid iff its final global state $E$ is reachable from the initial global state $\phi$.*

It is easy to see that if a loset contains a cycle of the $\rightarrow$ relation, then its

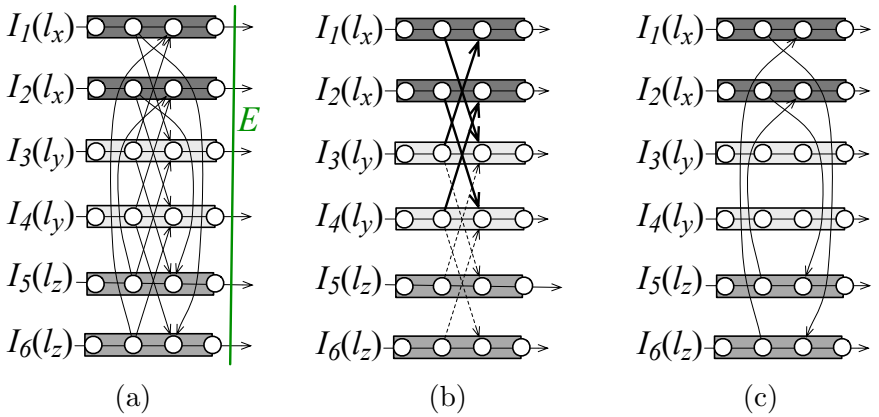



Figure 5.3: (a) A loset whose final global state is unreachable. (b)(c) The $\rightarrow$ relations in (a) is partitioned into two groups.

final global state is unreachable. We now show that it is possible to construct a loset that does not contain any cycle of the $\rightarrow$ relation and its final global state is still unreachable.

The example of a loset that is not valid is shown in Figure 5.3(a). The computation 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 explain why the final global state is unreachable.

Figure 5.3(b) shows the central part of the $\rightarrow$ relation in Figure 5.3(a). In Figure 5.3(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. 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 $I_4 \rightarrow I_3$. Therefore, the solid arrows in Figure 5.3(b) would induce one of the two sets of locking orders.

$$(I_1 \rightarrow I_2 \wedge I_3 \rightarrow I_4) \vee (I_2 \rightarrow I_1 \wedge I_4 \rightarrow I_3). \tag{5.1}$$

Due to the dashed arrows, our two sets of locking orders become:

$$(I_1 \rightarrow I_2 \wedge I_3 \rightarrow I_4 \wedge I_5 \rightarrow I_6) \vee (I_2 \rightarrow I_1 \wedge I_4 \rightarrow I_3 \wedge I_6 \rightarrow I_5). \tag{5.2}$$

Similar to Figure 5.3(b), the $\rightarrow$ relation in Figure 5.3(c) induces one of the two sets of locking orders depending upon whether $I_1[1]$ is executed before or after $I_2[1]$:

$$(I_1 \rightarrow I_2 \wedge I_6 \rightarrow I_5) \vee (I_2 \rightarrow I_1 \wedge I_5 \rightarrow I_6). \tag{5.3}$$

Figure 5.3(a) merges the relations $\rightarrow$ in Figure 5.3(b) and Figure 5.3(c). Initially, the computation does not have any cycle because every pair of the $\rightarrow$ 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 (5.2), and $(I_1 \rightarrow I_2) \wedge (I_6 \rightarrow I_5)$ from (5.3). 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 Figure 5.3(a) is unreachable, this computation cannot correspond to an actual execution of a program.

# Chapter 6

# Reachability of Global States in a Loset

In this chapter, we present two useful classes of global states — lock-free feasible global states and strongly feasible global states in the loset model. A lock-free feasible global state is always reachable and a reachable global state is always strongly feasible. Thus, these two sets of global states provide a lower and an upper bound on the set of reachable global states (see Figure 6.1). 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.

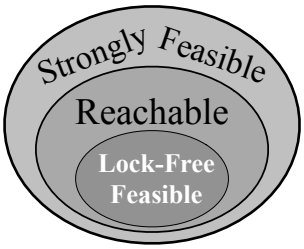



Figure 6.1: The set of lock-free feasible global states and the set of strongly feasible global states are a lower and an upper approximation of reachability, respectively, in a valid loset.

## 6.1 Lock-Free Feasible Global States

We first show that given a reachable global state $G$ of any loset (not just valid losets), the reachability of any lock-free feasible global state $F \preceq G$ is implied:

**Theorem 11.** *Given a reachable global state $G$ of a loset and a lock-free feasible global state $F \preceq G$, 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. $\forall \boldsymbol{i : 0 \leq i \leq |\mathcal{S}| : \delta(\phi, \mathcal{S}^i)}$ is consistent**:
It is sufficient to show that $\mathcal{S}$ is a linear extension of $\mathcal{L}$, i.e., the partial order $\rightarrow$ is preserved by the total order $\prec_{\mathcal{S}}$. 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 relation $\rightarrow$ 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. $\forall i : 0 \leq i \leq |\mathcal{S}_1| : \delta(\phi, \mathcal{S}_1^i)$ is compatible**:
Let the global state $V = \delta(\phi, \mathcal{S}_1^i)$. We show that

$$\forall s \neq t : \text{EL}(V[s]) \cap \text{EL}(V[t]) = \varnothing. \tag{6.1}$$

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:

$$\forall s \neq t : \text{EL}(W[s]) \cap \text{EL}(W[t]) = \varnothing. \tag{6.2}$$

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 (6.2), (6.1) 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

$$\forall s \neq t : \text{EL}(V[s]) \cap \text{EL}(V[t]) = \varnothing. \tag{6.3}$$

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:

$$\forall s \neq t : \textsc{el}(W[s]) \cap \textsc{el}(W[t]) = \varnothing. \tag{6.4}$$

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 $\textsc{el}(W[t]) \supseteq \textsc{el}(V[t]) = \varnothing$.

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

From the two cases, $\textsc{el}(W[t]) \supseteq \textsc{el}(V[t])$ holds for any thread $t$. Then, from (6.4), (6.3) 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 $\mathcal{S}_2$. □

A simple consequence of Theorem 11 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 any valid loset $\mathcal{L}$ are reachable.*

*Proof.* From the definition of valid loset, the final global state of the loset $\mathcal{L}$ is reachable. Then from Theorem 11, given any lock-free feasible global state $G$ of $\mathcal{L}$, we can obtain a run that reaches $G$ by reordering the run that reaches the final global state. Consequently, every lock-free feasible global state of $\mathcal{L}$ is reachable. □

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

**Theorem 12.** *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 $M = (G \sqcap H)$ and join $J = (G \sqcup H)$ 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 11, $M$ is reachable because $M \preceq G$. Now we show that their join $J$ is reachable.

Because $G$ is reachable, there exists a run $\mathcal{R}_G$. Then, from Theorem 11, 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. $\forall i : 0 \leq i \leq |\mathcal{S}_J| : \delta(\phi, \mathcal{S}_J^i)$ is consistent**:
Similar to the claim 1 of Theorem 11, 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 $\mathcal{R}_{MH}^i$ starting from $G$ results in a compatible global state:

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

$$\forall s \neq t : \text{EL}(V[s]) \cap \text{EL}(V[t]) = \varnothing. \tag{6.5}$$

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

$$\forall s \neq t : \text{EL}(W[s]) \cap \text{EL}(W[t]) = \varnothing. \tag{6.6}$$

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 $\text{EL}(V[t]) = \text{EL}(W[t])$ for any thread $t$. Then, from (6.6), (6.5) 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. □

Theorem 12 has two important implications. First, since the set of lock-free feasible global states forms a distributive lattice, we can concisely represent all lock-free feasible global states using the set of *join-irreducible* elements of the distributive lattice [DP90] and use slicing to study various sublattices [MSG07,Gar15]. 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 state $G$, we first find the greatest lock-free global state $F$ that precedes $G$. On account of Theorem 12, $F$ is well-defined whenever there exists any lock-free feasible global state $F \preceq \text{G}$. Given $F$, the following theorem shows that the search for the reachability of $G$ in a valid loset can be restricted to the events in $G \backslash F$.

**Theorem 13.** *Given a global state $G$ of a valid loset and the greatest lock-free feasible global state $F$ such that $F \preceq G$, the reachability of $G$ can be determined by the events $G \backslash F$.*

*Proof.* From Theorem 11, $F$ is reachable and the run that reaches the final global state $E$ 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 11 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 on only the events $G \backslash F$. □

Theorem 13 has one useful implication: the reachability of a global state $G$ can be determined using only a subset of events which is located between $G$ and the greatest lock-free global state that precedes $G$. Thus, lock-free feasible global states act as "reset" points for reachability and can be used to drastically reduce the time for checking reachability, by checking reachability in a subcomputation rather than the entire computation.

Besides lock-free feasible global states, the condition for a global state to be a reset point of reachability can be weakened. For instance, if a global state $G$ is feasible and any lock held by $G$ is never released after $G$, then $G$ is also a reset state. The reason is that the locking intervals that correspond to the locks that are held by $G$ and never released afterwards can be removed from the loset after $G$. Consequently, $G$ is reduced to a lock-free feasible global state. Similarly, if $G$ is feasible and any lock held by $G$ is never acquired by any different thread after $G$, then $G$ can also be reduced to a lock-free feasible global state and become a reset state. In this dissertation, we use only lock-free feasible global states as the reset points of reachability for simplicity.

## 6.2 Strongly Feasible Global States

So far we have discussed lock-free feasible global states of a valid loset which are guaranteed to be reachable. The set of global states is a lower-approximations of reachability. In this section, we give an upper-approximation of reachability. We define *strongly feasible global state* such that every reachable global state is strongly feasible. Also, just as feasibility and lock-freedom can be evaluated in polynomial time, strong feasibility can also be evaluated in polynomial time.

### 6.2.1 Locking Order

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

The relation $\mapsto$ 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 15** (The 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 $(I(l).acq \preceq e) \wedge (e \rightarrow f) \wedge (f \preceq J(l).rel)$.*

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:

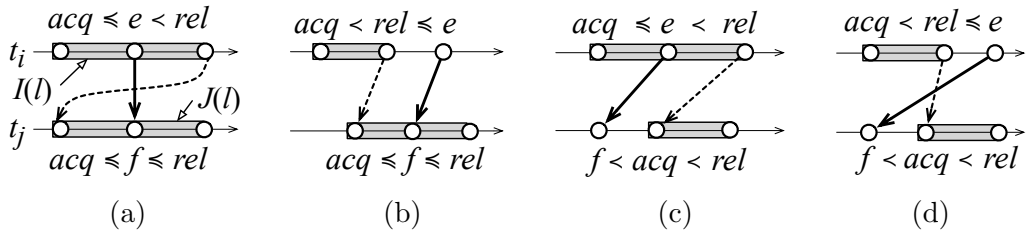



Figure 6.2: All possible cases of $I(l) \mapsto J(l)$ across different threads and the locking order $I(l).rel \rightarrow J(l).acq$.

**Definition 16** (Locking Order)**.** $\rightarrow_L \stackrel{\text{def}}{\equiv} \{(e, f) \mid \exists I(l), J(l) : (e = I(l).rel) \wedge (f = J(l).acq) \wedge (I(l) \mapsto J(l)).\}$

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. Figure 6.2 shows all possible cases of $I(l) \mapsto J(l)$ and the corresponding locking order. 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 dissertation, we assume for simplicity that the initial global state of the loset are lock-free. 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. Similarly, an interval $J(l)$ that is part of the final global state would be ordered after all other intervals with the same lock.

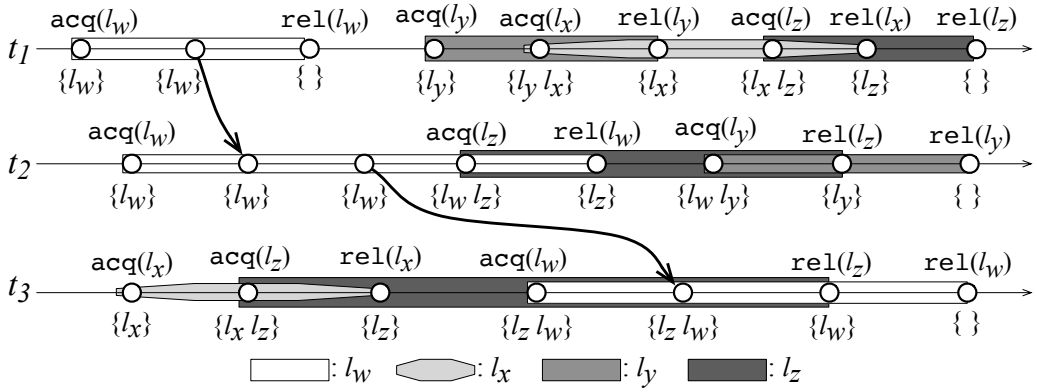



Figure 6.3: An initial loset $\mathcal{L}$, which contains only the HB relation.

### 6.2.2 Normalization of Loset

Since the combination of HB orders and locking constraints may introduce additional order constraints $\rightarrow_L$ during execution, it is easier to determine the reachability of a global state in 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 17** (Normal Loset)**.** *A loset $\mathcal{L} = (E, \rightarrow, n, L, pid, \mathcal{I})$ is normal if $\forall e, f \in E : e \rightarrow_L f \Rightarrow e \rightarrow f$.*

Figure 6.3 shows a loset $\mathcal{L}$, which contains only the HB relation. 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 relations 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. Figure 6.4 shows the corresponding normal loset $\mathcal{L}'$, which has locking orders added to $\mathcal{L}$. The dashed arrows in Figure 6.4 are used to

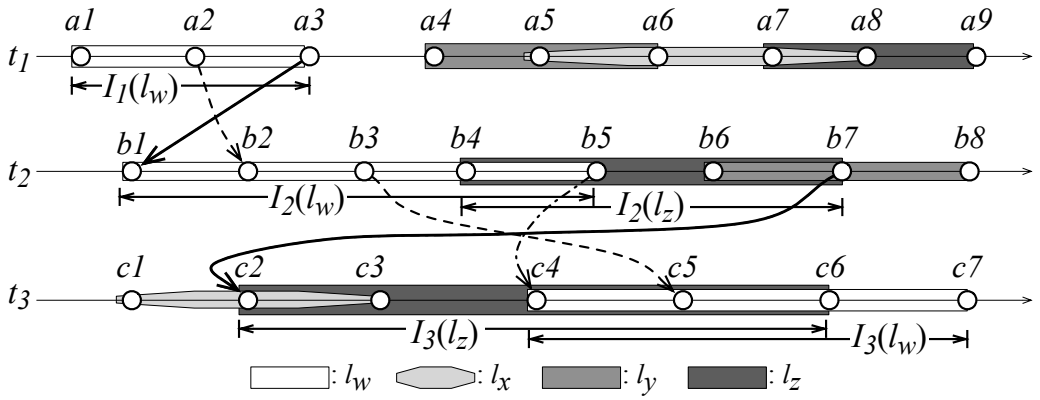



Figure 6.4: A normalized loset $\mathcal{L}'$, where the locking orders (the solid arrows) are added to the original loset $\mathcal{L}$.

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 relation $a3 \rightarrow b1$ is added. 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 Figure 6.4, 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 17 shows a procedure to normalize a loset $\mathcal{L}$. The algorithm takes as input the direct and transitive HB relations in the computation (i.e., $a2 \rightarrow b2$, $b3 \rightarrow c5$, and $a2 \rightarrow c5$ in Figure 6.3) and iteratively adds the locking orders to the computation by locating the cases of the $\mapsto$ relation in

**Algorithm 17** NormalizeLoset($\mathcal{L}$, $\mathcal{H}$)

**Input:** A loset $\mathcal{L}$ and a set $\mathcal{H}$ of seed relations, which initially contains all HB relations in $\mathcal{L}$.

**Output:** Returns false if a cycle in the $\rightarrow$ relation is detected; otherwise, the loset $\mathcal{L}$ is normalized.

1: **for** each seed order $e_i \rightarrow e_j$ in $\mathcal{H}$ **do** ▷ $\mathcal{H}$ initially contains all direct and transitive $\rightarrow$ relations.
2: **for** each $l \in EL(e_i) \cup EL(e_j)$ **do** ▷ Exclude the case of Figure 6.2(d).
3: Let $I(l)$ be the most recent locking interval for $l$ s.t. $I(l).acq \preceq e_i$.
4: Let $J(l)$ be the first locking interval for $l$ s.t. $e_j \preceq J(l).rel$.
5: **if** either $I(l)$ or $J(l)$ does not exist **then** continue ▷ None of the cases, Figure 6.2(a), 6.2(b), or 6.2(c), holds.
6: **if** the relation $I(l) \rightarrow J(l)$ completes a cycle **then return** false
7: **else**
8: Add $I(l) \rightarrow J(l)$ to the loset and to the set $\mathcal{H}$ ▷ $I(l) \rightarrow J(l)$ means $I(l).rel \rightarrow J(l).acq$.
9: Append new transitive relations due to $I(l) \rightarrow J(l)$ to $\mathcal{H}$
10: **end if**
11: **end for**
12: **end for**
13: **return** true

Figure 6.2(a), 6.2(b), and 6.2(c). The case of Figure 6.2(d) is ruled out in Algorithm 17 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 show that the normalized loset contains the same set of runs, which reach the final global state, as the original loset. We first define the runs $Runs(\mathcal{L})$ of a global state $G$ in the loset $\mathcal{L}$:

**Definition 18** (Runs of a Loset)**.** *Given any loset $\mathcal{L}$, the set $Runs(\mathcal{L}) = \{\mathcal{R} \mid \mathcal{R}$ is a run that reaches the final global state $E$ of $\mathcal{L}$ from the initial global state $\phi\}$.*

**Theorem 14.** *Given a loset $\mathcal{L}$ and the corresponding normal loset $\mathcal{L}'$, then $Runs(\mathcal{L}) = Runs(\mathcal{L}')$.*

*Proof.* We show that $Runs(\mathcal{L}') \subseteq Runs(\mathcal{L})$ and $Runs(\mathcal{L}) \subseteq Runs(\mathcal{L}')$. Since $\mathcal{L}'$ contains more constraints of the $\rightarrow$ relation, we get $Runs(\mathcal{L}') \subseteq Runs(\mathcal{L})$.

On the other hand, we show that any run $\mathcal{R}$ in $Runs(\mathcal{L})$ is also a run of $Runs(\mathcal{L}')$. Since $\mathcal{R}$ cannot violate any locking order constraint and therefore only goes through feasible states, it is sufficient to show that the normalization of $\mathcal{L}$ does not remove any feasible global that is contained in any $\mathcal{R}$ of $Runs(\mathcal{L})$. Figure 6.5 shows all three cases in which the feasible global state $G$ of $\mathcal{L}$ is removed during the normalization. Note that Figure 6.5 shows the cases in which $\mapsto$ relation across two threads, but the relation can be extended to the

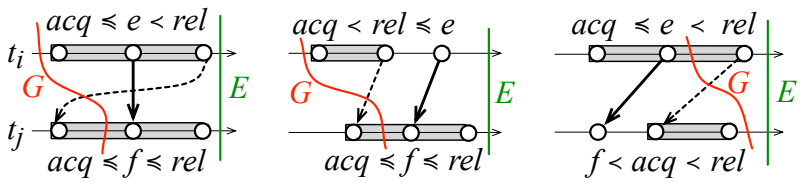



Figure 6.5: All possible cases of the removed feasible global state $G$ during the normalization of a loset $\mathcal{L}$, i.e., $G$ is feasible in $\mathcal{L}$ but not feasible in $\mathcal{L}'$. The dashed arrows only appear in $\mathcal{L}'$.

cases with more than 2 threads. As it can be seen, the removed feasible global state $G$ implies that thread $t_j$ has to acquire the lock before $t_i$. However, the relation $e \to f$ implies that $t_i$ has to acquire the lock before $t_j$, which is a contradiction. Consequently, $G$ is either leading to a deadlock (Figure 6.5(a) and 6.5(b)) or unreachable (Figure 6.5(c)). Hence, the normalization of a loset only removes feasible global states that cannot be contained in any run $\mathcal{R}$ of $Runs(\mathcal{L})$. Therefore, we get $Runs(\mathcal{L}) \subseteq Runs(\mathcal{L}')$. □

We now discuss the complexity of the normalization procedure.

**Theorem 15.** *The time complexity of Algorithm 17 is* $O(n|E|^3L)$.

*Proof.* Line 1 executes at most $O(|E|^2)$ times because there are at most $O(|E|^2)$ pairs of the $\to$ 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 $I(l) \to J(l)$ at line 8. □

### 6.2.3 Strong Feasibility of Global States

The main idea behind strong feasibility is as follows. If a lock $l$ is held by a thread $t$ in the global state $G$, then the release of $l$ that occurred on the other threads prior to $G$ should have happened before the acquisition of $l$ that occurred on $t$. We refer to this order as the *dynamic locking order*:

**Definition 19** (Dynamic Locking Order)**.** *Let $J(l)$ be the locking interval that contains $G[j]$. Let $I(l)$ be the most recent interval, if any, such that $I(l).rel \preceq G[i]$. Then, $I(l).rel \rightarrow_L J(l).acq$.*

Similar to the normalization of a loset $\mathcal{L}$, the dynamic locking orders due to $G$ can be added to $\mathcal{L}$ and then be normalized. We now define *the strong feasibility* of a global state as follows:

**Definition 20** (Strong Feasibility)**.** *A feasible global state $G$ is strongly feasible iff the normalization due to the dynamic locking orders of $G$ does not induce any cycle in the relation $\rightarrow$.*

We use the feasible global state $G = [8, 7, 7]$ in Figure 6.6 to show the calculation of strong feasibility:

**Step 1**: From Theorem 13, 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 Figure 6.6 are excluded.

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

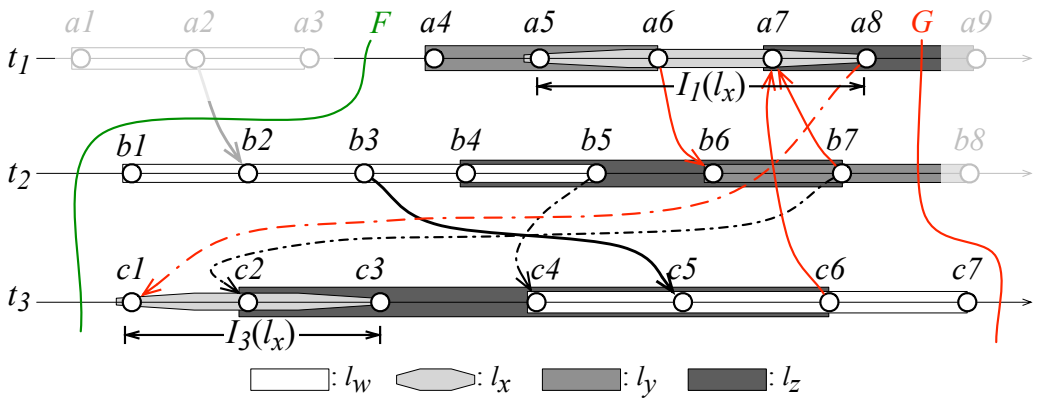



Figure 6.6: The feasible global state $G$ is unreachable because the dynamic locking order completes a cycle in the relation $\rightarrow$.

**Step 3**: The HB relations of the bounded 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 $b7 \rightarrow c2$. Moreover, 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 relation $\rightarrow$ is induced: $a8 \rightarrow c1 \rightarrow c6 \rightarrow a7 \rightarrow a8$. Hence, $G$ is not strongly feasible.

**Theorem 16.** *The time complexity for calculating the strong feasibility of a global state is* $O(n|E|^3L)$.

*Proof.* In step 1, the bound $F$ can be identified using the detection algorithm of conjunctive predicate [GW91] in a backward fashion starting from $G$. The predicate to detect is "*all threads hold no locks*". In addition, the algorithm 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 $O(n^2L)$ time. In step 3, the dynamic locking orders and the HB relations in the bounded loset are used as the set $\mathcal{H}$ for Algorithm 17, which takes at most $O(n|E|^3L)$ time. □

## 6.3 Reachability of Strongly Feasible Global States

The set of strongly feasible global states is a superset of reachable global states because of the following two reasons. First, a reachable global state $G$ is strongly feasible because the normalization during the calculation of strong feasibility does not remove any run that reaches $G$; from Theorem 13, we can replace $\mathcal{L}$ and $E$ of Theorem 14 with the bounded loset and $G$ during the calculation of strong feasibility, respectively. Second, strong feasibility does not imply reachability; in section 6.3.1, we show an example loset where a strongly feasible global state is not reachable. However, strong feasibility is still useful in practice. In section 6.3.2 we show that reachability and strong feasibility are equivalent for any loset with two threads. Moreover, in section 6.3.3 we present experiments to show that the gap between the strong feasibility and the reachability seldom exists in practice.

### 6.3.1 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

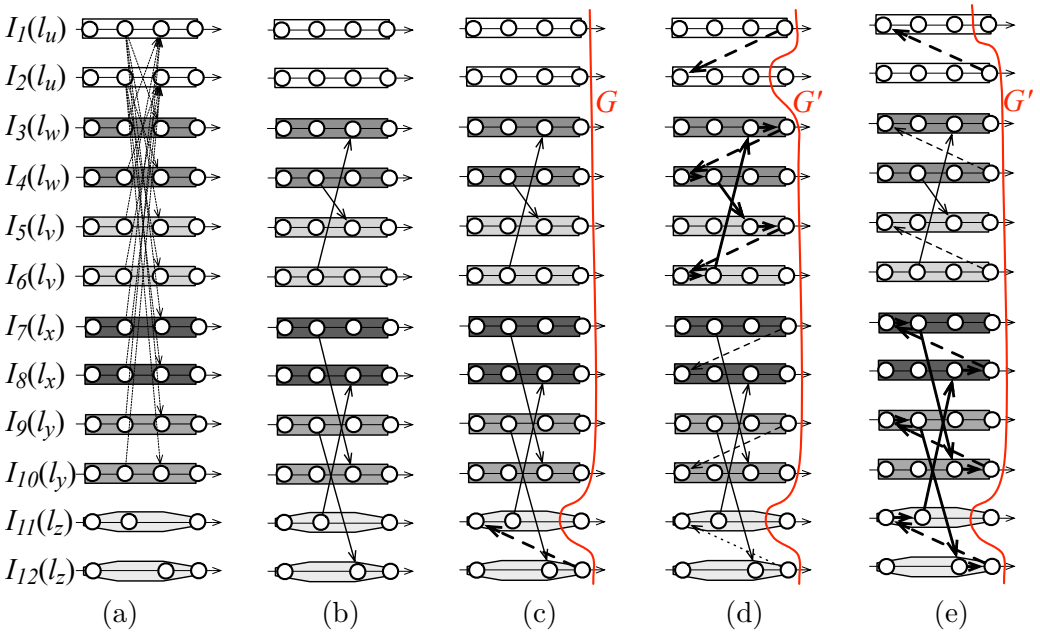



Figure 6.7: A computation whose final global state is reachable. In addition, $G$ is strongly feasible but unreachable. The dynamic locking orders are drawn in dashed arrows.

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.

The example computation is shown in Figure 6.7(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 in Figure 5.3(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 dynamic 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 removal of the last event of the intervals $I_1$, $I_3$, $I_5$, $I_7$, and $I_9$, induces the set $B$ of dynamic locking orders: $(I_2 \rightarrow I_1) \wedge (I_4 \rightarrow I_3) \wedge (I_6 \rightarrow I_5) \wedge$

$(I_8 \rightarrow I_7) \wedge (I_{10} \rightarrow I_9)$.

Figure 6.7(b) shows the remaining $\rightarrow$ relation in the computation, i.e., the combination of Figure 6.7(a) and 6.7(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 Figure 6.7(a) are omitted in the other figures of Figure 6.7. The final global state can be reached by the run that preserves the partial order: (1) $I_{11} \rightarrow I_{12}$, and (2) $(I_1 \rightarrow I_2) \wedge (I_3 \rightarrow I_4) \wedge (I_5 \rightarrow I_6) \wedge (I_7 \rightarrow I_8) \wedge (I_9 \rightarrow I_{10})$.

Figure 6.7(c) shows the strong feasible global state $G$, where the dynamic 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 dynamic locking orders and the ones that induce the set $B$ of dynamic 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 dynamic locking orders is induced, which is shown in Figure 6.7(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 dynamic locking orders is induced by the removal of $G[1]$

(i.e., $I_1[4]$) as shown in Figure 6.7(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.

Note that since our loset model allows the locking intervals to be overlapped, the number of threads could be reduced to 9 threads by overlapping the threads $t_4$ with $t_5$, $t_8$ with $t_9$, and $t_{10}$ with $t_{11}$.

### 6.3.2 Strong Feasibility Equals to Reachability in Losets with Two Threads

This section shows that the reachability and strong feasibility are equivalent for any loset with two threads:

**Theorem 17.** *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. From induction hypothesis, we can then 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

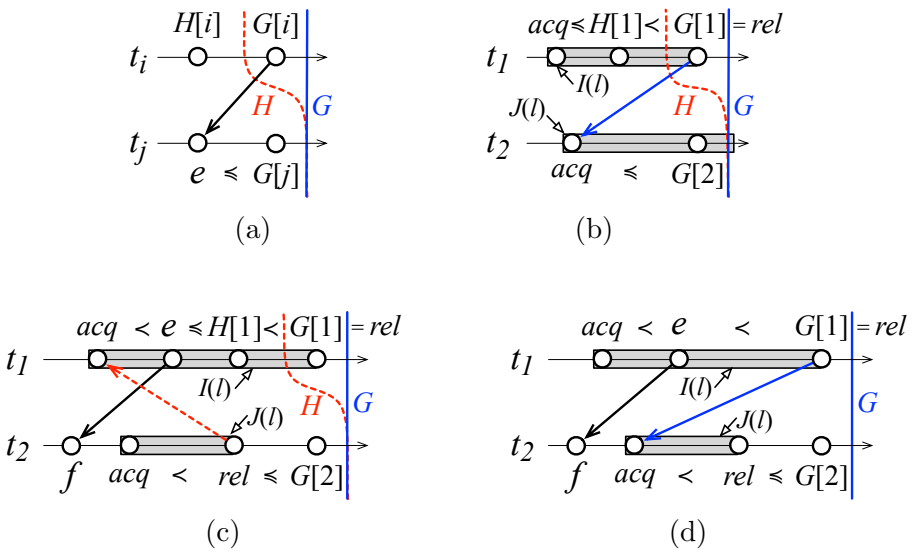



Figure 6.8: (a) CASE 1: $H = G - G[1]$ is inconsistent. (b) CASE 2: $H$ is incompatible. (c) CASE 3: $H$ induces a cycle in the $\rightarrow$ relation and either $(f \preceq acq)$ or $(acq \preceq f)$ holds. (d) CASE 3: The cycle in (c) implies $G[1] \rightarrow G[2]$.

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 Figure 6.8(a).)

CASE 2. *$H$ is not compatible*: An example loset is shown in Figure 6.8(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 (i.e., $\text{EL}(G[1]) \cap \text{EL}(G[2]) = \varnothing$), we get $G[1] = I(l).rel$. Consequently, the dynamic

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*: Figure 6.8(c) shows an example loset. Since $G$ is strongly feasible, the cycle must be completed by a dynamic locking order that is induced by $H$. Suppose that the dynamic order is induced because of the lock $l$, then the following conditions hold:

1. Since the dynamic 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 dynamic 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 dynamic locking order due to $H$ must point toward the events that occur on $t_i$. Hence, the relation $e \rightarrow f$ is either an existing HB relation of the computation or a dynamic 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 Figure 6.8(d)) and hence the dynamic 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. $\square$

### 6.3.3 Enumeration of Reachable Global States Using Strong Feasibility

In this section, we present experiments to show that the gap between the strong feasibility and the reachability seldom exists in practice. Specifically, 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.

There are two approaches in literature to enumerate reachable global states of a computation $\mathcal{L}$. The first approach uses breadth (BFS) or depth (DFS) first strategy to add one event to the current global state $G$ at a time [CM91, Gar03]. 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 $\mathcal{L}$.

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 [PR93, JMN95, HMNS01, Gan10, Gar03, CG15b]. How-

ever, an edge may pass through unreachable global states because the set of consistent global states is a superset of reachable global states a loset. Therefore, this approach needs to incorporate an additional function to prune the consistent but unreachable global states. In this dissertation, we use QuickLex [CG15b] to enumerate the consistent global states and use the strong feasibility to prune the unreachable global states.

#### 6.3.3.1 Enumerating the Reachable Global States in a Loset Using QuickLex

Since QuickLex only guarantees that the global state $G$ to be enumerated is consistent, so $G$ has to be checked if it is strongly feasible. QuickLex checks the strong feasibility of $G$ in two steps: First, it checks if $G$ is feasible. Then, it calculates the strong feasibility of $G$.

Algorithm 18 shows the modified lexical algorithm, where the original lines in QuickLex are underlined. To speed up the calculation of the feasibility and the strong feasibility of a global state, QuickLex has the following modifications. First, the strong feasibility of the current global state $G$ is implied if $G = F + e$, where $e$ is an event such that $G$ is feasible and $F$ is a reachable or strongly feasible global state. In Algorithm 18, we only use the global state from the previous iteration as the global state $F$ for performance and memory space concerns. The global variable `sum`, which is updated whenever an event is added to or removed from $G$, indicates the difference of events between the current $G$ and the global states from the previous iteration. The vari-

**Algorithm 18** EnumerateStrongFeasibleGlobalStates($\mathcal{L}$)

**Input:** A normal loset $\mathcal{L}$ that contains $n$ threads.

1: `sum` := 0 ▷ A global variable which indicates the number of events that have been added in and removed from $G$.
2: `compatibleUntilT` := $(n+1)$ ▷ A global variable which means that the maximal events from $G[1]$ to $G$[`compatibleUntilT`-1] are pairwise compatible.
3: $prevGisStrFeasible$ := true
4: **while** true **do**

▷ Lines 5–10 reduces the number of the calculation of strong feasibility of global states.

5: **if** `compatibleUntilT` $= (n+1)$ and ($(prevGisStrFeasible$ and `sum` $\leq 1$) or isStronglyFeasible($G$)) **then**
6: $prevGisStrFeasible := true$
7: Evaluate the predicate on $G$.
8: **else**
9: $prevGisStrFeasible := false$
10: **end if**

11: $k$ := Propagate() ▷ Include events of $E_k$ into $G$.
12: **if** $k = 0$ **then** break ▷ No more events can be included.
13: Reset($k$) ▷ Reset low order part and find the next forbidden maximal-event.
14: **end while**

**Algorithm 19** GetForbiddenMaximalEvent($G$)

**Input:** A global state $G$.
**Output:** The maximal forbidden event of $G$.

```
1: for i from 1 to n do
2:    for j from 1 to (i − 1) do
3:       if G[i] is incompatible with G[j] then return G[i]
4:    end for
5: end for
6: return null                                  ▷ G is compatible.
```

able $prevGisStrFeasible$ indicates whether the global state from the previous iteration is strongly feasible.

Second, QuickLex provides an useful property: the high-order parts of two consecutive global states are identical. This property allows us to skip multiple incompatible global states using the *forbidden maximal-event*, which is defined as follows:

**Definition 21** (Forbidden Maximal Event)**.** *For $i$ from 1 to $n$, the maximal event $G[i]$ is a forbidden maximal-event iff it is the first $G[i]$ that is not pairwise compatible with $G[j]$, where $j$ ranges from 1 to $(i-1)$.*

The notion of forbidden maximal event is that if $G[i]$ is a forbidden maximal-event, then the global state $G$ remains incompatible unless more events of $E_i$ are included. Algorithm 19 shows the procedure for locating the forbidden maximal event for any given global state $G$. Note that our implementation does not actually use Algorithm 19; instead, we use the global variable `compatibleUntilT` to indicate the forbidden maximal-event of $G$ and the vari-

**Algorithm 20** PROPAGATE()

**Output:** Returns $k = 0$ if no more events can be included; otherwise, includes new events from $E_k$ while the maximal events, $G[1]$ to $G[k]$, are pairwise compatible.

1: $n' := min(n, \texttt{compatibleUntilT})$
2: **while** $k$ from $n'$ to 1 **do**
3: $\texttt{sum} := 0$ ▷ Reset `sum`.
4: $orgGk := G[k]$ ▷ The current maximal event of $E_k$.
5: **while** $G + e_k \preceq E$ **do** ▷ $e_k$ is the successor event of $G[k]$ and $E$ is the final global state of $\mathcal{L}$.
6: **if** $e_k$ does not consistent with $G[1]$ to $G[k-1]$ **then** break
7: $G := G + e_k$ ▷ Include one more event from $E_k$
8: $\texttt{sum} := \texttt{sum} + 1$
9: **if** $e_k$ is compatible with $G[1]$ to $G[k-1]$ **then return** $k$
10: **end while**
11: $G[k] := orgGk$ ▷ Restore $G[k]$ and proceed to $E_{k-1}$.
12: **end while**
13: **return** 0

able is updated incrementally. Therefore, $G$ is compatible when `compatibleUntilT` equals $(n + 1)$.

Algorithm 20 shows the details of the function PROPAGATE, which is modified to ensure that the maximal events at the high-order part of the global state, i.e., $G[1]$ to $G[k]$, are pairwise compatible with each other. The original function PROPAGATE includes one event of $E_k$ at a time, the modified PROPAGATE may include multiple events of $E_k$ in order to resolve the incompatibility of locks. In Algorithm 20, since $E_{\texttt{compatibleUntilT}}$ contains the forbidden maximal-event, PROPAGATE initially assigns $E_{\texttt{compatibleUntilT}}$ to $E_k$ and includes more events until the incompatibility is resolved. However, if the

**Algorithm 21** RESET()

**Input:** The decided $k$.
**Output:** The maximal event of every $G[l]$ is set to its least value.

```
 1: procedure RESET(k)
 2:    compatibleUntilT := (n + 1)
 3:    for l from (k + 1) to n do
 4:       leastIdx := compute the least index (value) for G[l]
 5:       G[l] := leastIdx
 6:       sum := sum + ABS(G[l] − leastIdx)
 7:       if (l < compatibleUntilT) and G[l] is not pairwise compatible with
    any of G[1] to G[l − 1] then
 8:          compatibleUntilT := l
 9:       end if
10:    end for
11: end procedure
```

event to be included violates the consistency of $G$, then all succeeding events also violate the consistency [Gar03, Gan10]; hence, PROPAGATE proceeds to the next sequence $E_{k-1}$ of events and try to resolve the incompatibility by including more events from $E_{k-1}$. Algorithm 21 shows the details of the function RESET, which finds the least value for all $G[l]$, where $l > k$, and the next forbidden maximal-event.

Finally, since the high order maximal events, i.e., $G[1]$ to $G[k-1]$, remains the same between consecutive global states, the compatibility of $G$ in the line 9 of PROPAGATE and the line 7 of RESET can be calculated incrementally. We define the array $\mathbb{EL}$ that stores the sets of effective locks such that $\mathbb{EL}[i] = \bigcup_{j=1}^{i} \text{EL}(G[j])$. When checking if $e_k$ is compatible with the high order maximal events, i.e., $G[1]$ to $G[k-1]$, we check if $\text{EL}(e_k) \cap \mathbb{EL}[k-1] = \varnothing$.

Table 6.1: The information of benchmarks and runtimes (sec.) of each enumeration approach.

| Benchmark | n | #events | #GS | BFS | DFS | Lex1 | Lex2 |
|---|---|---|---|---|---|---|---|
| ***bank*** | 7 | 91 | 664,325 | 0.99 | 3.20 | 0.20 | 0.09 |
| ***set (faulty)*** | 6 | 114 | 947,951 | 1.36 | 5.25 | 4.55 | 1.16 |
| ***set (correct)*** | 6 | 140 | 2,762,420 | 3.55 | 28.70 | 16.84 | 3.16 |
| ***arraylist1*** | 12 | 56 | 354,293 | 0.57 | 1.06 | 0.11 | 0.07 |
| ***arraylist2*** | 7 | 103 | 3,045,808 | 4.48 | 30.28 | 2.42 | 0.22 |
| ***sor*** | 14 | 66 | 3,188,645 | 9.16 | 32.29 | 0.29 | 0.22 |
| ***tsp*** | 8 | 76 | 1,235,981 | 1.99 | 11.26 | 0.80 | 0.17 |
| ***raytracer*** | 9 | 121 | 4,882,833 | 10.36 | 42.57 | 1.67 | 0.54 |
| ***hedc*** | 7 | 92 | 458,334 | 0.64 | 1.50 | 0.66 | 0.38 |
| ***bank*** | 9 | 121 | 53,808,433 | 350.27 | o.o.m. | 21.76 | 4.47 |
| ***set (faulty)*** | 7 | 147 | 15,040,942 | 40.21 | o.o.m. | 183.68 | 23.02 |
| ***set (correct)*** | 7 | 189 | 78,130,591 | 452.43 | o.o.m. | 1476.82 | 160.38 |
| ***arraylist1*** | 16 | 76 | 28,697,813 | 175.80 | o.o.m. | 3.22 | 1.66 |
| ***arraylist2*** | 8 | 118 | 25,740,144 | 104.81 | o.o.m. | 43.62 | 1.75 |
| ***sor*** | 16 | 76 | 28,697,813 | 174.48 | o.o.m. | 2.47 | 1.64 |
| ***tsp*** | 10 | 90 | 25,000,001 | 115.77 | o.o.m. | 807.08 | 52.33 |
| ***raytracer*** | 10 | 132 | 24,414,083 | 98.15 | o.o.m. | 10.30 | 2.83 |
| ***hedc*** | 9 | 121 | 24,522,560 | 108.37 | o.o.m. | 90.29 | 7.30 |

o.o.m.: Out of memeory.

Afterwards, $\mathbb{EL}[k] = \textsc{el}(e_k) \cup \mathbb{EL}[k-1]$.

#### 6.3.3.2 Experimental Results

Table 6.1 shows the information of the benchmarks that are used in the experiment. The benchmark *banking* is an toy program, which was used to demonstrates typical error patterns in concurrent programs [FNU03]; *set (faulty)* and *set (correct)* are incorrect and correct implementations of the concurrent set [HS08]; *arraylist1* is a non-thread-safe container and *arraylist2*

is a thread-safe container from Java library; *sor* is a scientific computation application; *tsp* is a parallel solver for the traveling salesman problem; *raytracer* is a benchmark for measuring the performance of a 3D raytracer; and *hedc* is a crawler for searching Internet archives. The benchmarks *sor*, *tsp*, *raytracer*, and *hedc* are the benchmark programs that are used in [CSR08, FF09, vPG01]. 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 6.1 contains two sets of results. The set at the upper part 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 have one more thread. On the other hand, the set at the lower part of the table shows the largest computations that the BFS algorithm can handle. The BFS and DFS algorithms generate the reachable global states and the lexical algorithms generate strongly feasible global states. However, all the compared algorithms generate the same set of global states.

The runtimes of our enumeration approach are shown in Lex1 and Lex2, where Lex1 checks the strong feasibility on every consistent global state and Lex2 only checks the strong feasibility when the previous global state is not strongly feasible. As it can be seen, Lex2 reduces 60% of runtime in average.

Moreover, Lex2 reduces 84% and 61% of runtime in comparison with BFS and DFS algorithms, respectively.

## 6.4 Viable Global States

In this section, we present *viable* global states such that a viable global state is always reachable and viability can be determined in polynomial time. Moreover, every lock-free feasible state is always viable although the converse is false. Both viable and lock-free feasible global states are a lower approximation of reachability. However, the set of viable global states is larger than or equal to the set of lock-free feasible global states.

**Definition 22.** *In a valid loset $\mathcal{L}''$ that is normalized with the HB relation, a global state $G$ is* viable *if (1) it is feasible, and (2) if $I(l)$ is any interval on thread $i$ such that $I.l \in \text{EL}(G[i])$ and $J(l)$ is any interval on thread $j$ such that $J(l).rel \preceq G[j]$, then $J(l) \mapsto I(l)$.*

Note that the loset $\mathcal{L}''$ is normalized before the dynamic locking orders due to $G$ are identified, i.e., only the HB relation of the loset are initially added to the set $\mathcal{H}$ of seed relation. Since a lock-free global state is feasible and cannot have any interval $I(l)$ such that $I(l).acq \preceq G[i] \prec I(l).rel$, it is trivially viable. We now show

**Theorem 18.** *Given a loset $\mathcal{L}''$ that is normalized with the HB relation, a reachable global state $G$ of $\mathcal{L}''$, and a viable global state $V \preceq G$, $V$ is reachable.*

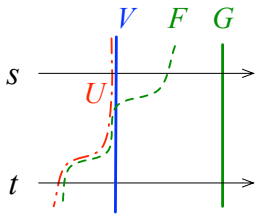



Figure 6.9: $G$ is a reachable global state of the normal loset; $V$ is a viable global state; $F$ follows $\mathcal{R}$ reaching $G$ from $\phi$; and $U$ follows $\mathcal{S}$ reaching $V$ from $\phi$.

*Proof.* Let $\mathcal{R}$ be the run that takes the loset to $G$, i.e., $\delta(\phi, \mathcal{R}) = G$, and the sequence $\mathcal{S} = \mathcal{R} \uparrow V$ of events be the projection of $\mathcal{R}$ that contains exactly the events in $V$. If $\mathcal{S}$ is a run, then $V$ is reachable. Since $\mathcal{S}$ is a projection of $R$, it also preserves the $\rightarrow$ relation and hence $\forall i : 0 \leq i \leq |\mathcal{S}| : \delta(\phi, \mathcal{S}^i)$ is consistent. We now show that $\forall i : 0 \leq i \leq |\mathcal{S}| : \delta(\phi, \mathcal{S}^i)$ is compatible:

Let the global state $U = \delta(\phi, \mathcal{S}^i)$. We need to show that

$$\forall s \neq t : \text{EL}(U[s]) \cap \text{EL}(U[t]) = \varnothing. \tag{6.7}$$

Let $\mathcal{R}^j$ be the shortest prefix of $\mathcal{R}$ such that $\mathcal{R}^j \uparrow V = \mathcal{S}^i$ and let $F = \delta(\phi, \mathcal{R}^j)$. Since $\mathcal{R}$ is a run, we get

$$\forall s \neq t : \text{EL}(F[s]) \cap \text{EL}(F[t]) = \varnothing. \tag{6.8}$$

The global states $G$, $V$, $U$, and $F$ are shown in Figure 6.9. We now consider the following three cases for any $s \neq t$ in the global state $U$:

CASE 1. $(U[s] = F[s]) \wedge (U[t] = F[t])$: In this case, we get $(\text{EL}(U[s]) = \text{EL}(F[s])) \wedge (\text{EL}(U[t]) = \text{EL}(F[t]))$. Then from (6.8), $\text{EL}(U[s]) \cap \text{EL}(U[t]) = \varnothing$ holds.

CASE 2. $(U[s] \prec F[s]) \wedge (U[t] \prec F[t])$: Because $\mathcal{S}^i = \mathcal{R}^j \uparrow V$, $(U[s] \prec F[s])$ holds only if $\mathcal{R}^j$ contains the events in $G \backslash V$ w.r.t. $E_s$, which implies that $\mathcal{S}^i$ contains all the events in $V$ w.r.t. $E_s$. Thus, we get $(U[s] = V[s] \prec F[s])$. Similarly, $(U[t] = V[t] \prec F[t])$. Since $V$ is feasible, $\text{EL}(U[s]) \cap \text{EL}(U[t]) = \varnothing$ holds.

CASE 3. $(U[s] \prec F[s]) \wedge (U[t] = F[t])$: Because $\mathcal{S}^i = \mathcal{R}^j \uparrow V$, we get $(U[s] = V[s] \prec F[s])$ and $(U[t] = F[t] \preceq V[t])$. If $(U[t] = F[t] = V[t])$, we get $\text{EL}(U[s]) \cap \text{EL}(U[t]) = \varnothing$ because $V$ is feasible.

We now consider $(U[t] = F[t] < V[t])$. Assume that there exists a lock $l \in (\text{EL}(U[s]) \cap \text{EL}(U[t]))$. Let $S$ and $T$ be the locking intervals that contain $U[s]$ and $U[t]$, respectively. Since $U[t] < V[t]$ and $V$ is feasible, we get $U[t] \prec T.rel \preceq V[t]$. Then, because $V$ is viable and $\mathcal{L}''$ is normalized, we get $T \mapsto S$ and hence $T.rel \rightarrow S.acq$. Since $U[t] \prec T.rel$ and $S.acq \preceq U[s]$, $T.rel \rightarrow S.acq$ violates the consistency of $U$ and hence a contradiction. Consequently, $\text{EL}(U[s]) \cap \text{EL}(U[t]) = \varnothing$ holds.

From cases 1, 2, and 3, (6.7) holds. Hence, $\mathcal{S}$ is a run. □

Note that a viable global state $V$ cannot be used as a reset point of reachability. Assume that the global state $G$ in the normal loset shown in Figure 6.10 is reachable. The viable global state $V$ is reachable because of Theorem 18, but $G$ cannot be reached from $V$ because the thread $t$ of $V$ is not able to acquire the lock that is currently held by the thread $s$.

## 6.5 Relationship Among Various Classes of Global States

The relationship among different sets of global states in a valid loset, whose final global state is reachable, is shown in Figure 6.11. The set of strongly feasible global states is a superset of reachable global states because of Theorem 10. Moreover, if a global state $G$ is not strongly feasible, then the dynamic locking order due to $G$ induces a cycle in the relation $\rightarrow$, and therefore, $G$ cannot be reachable. Corollary 1 and Theorem 18 show that all lock-free feasible global states and viable global states are reachable, respectively. Hence, they are subsets of reachable global states.

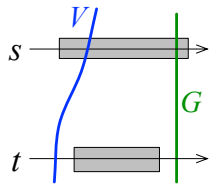



Figure 6.10: $G$ and $V$ is a reachable and viable global state of the normal loset, respectively.

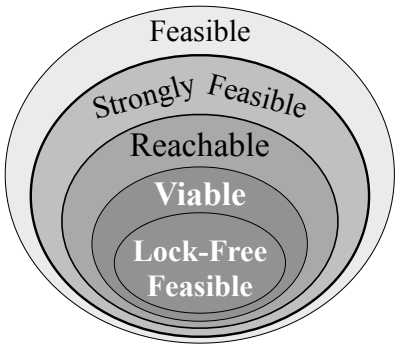



Figure 6.11: The relationship among various classes of global states in a valid loset.

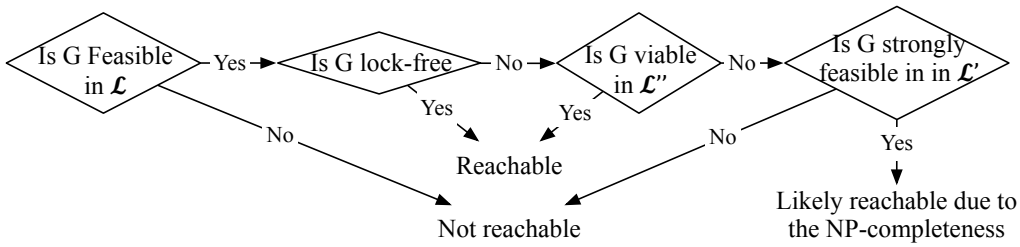



Figure 6.12: The decision flow for determining the reachability of a global state in a loset $\mathcal{L}$.

Figure 6.12 shows a flow for determining the reachability of a global state $G$ in a loset $\mathcal{L}$. The loset $\mathcal{L}''$ is the loset $\mathcal{L}$ that is normalized to the HB relation, i.e., the dynamic locking orders are initially excluded from the set $\mathcal{H}$. The loset $\mathcal{L}'$ is the sub-loset of $\mathcal{L}$ that is normalized to the HB relation along with the dynamic locking orders due to the given $G$. From Figure 6.12, given a feasible global state $G$, it is easy to answer reachability if either it is lock-free feasible, viable, or not strongly feasible. If none of these cases holds, then the precise reachability cannot be determined efficiently because the NP-completeness of reachability problem. However, from Theorem 13, the calculation of reachability of $G$ can be bounded in the sub-loset $(G \backslash F)$, where $F$ is the greatest lock-free feasible global state that precedes $G$, rather than the entire computation.

# Chapter 7

# Conclusions

In this dissertation, we study the technique of predicate detection for general-purpose predicates. The problem of predicate detection is to predictively detect if the predicate could become true in any reachable global state of the given computation, i.e., the execution trace of the program. This predictive technique assumes that process or thread scheduling is the only source of the non-determinism of the program. Moreover, our work focus on general-purpose predicate detection which does not make assumptions on the nature of the predicate. Hence, our technique of predicate detection ensures that every reachable global state of the computation is enumerated exactly once.

The first part of this dissertations introduces the first online-and-parallel predicate detector, named ParaMount [CG15a], for detecting general-purpose predicates. ParaMount provides a strategy to partition the set of consistent global states. In addition, ParaMount can run along with the execution of user's program and hence is applicable even to non-terminating programs such as web-server applications. In ParaMount, each subset of consistent global states can be enumerated individually using existing sequential enumeration algorithms, e.g., the BFS algorithm [CM91] or the lexical algo-

rithm [Gan10, Gar03, CG15b].

Our online predicate detector, ParaMount, uses the method [FF09, LC06] for capturing computation from the execution of the program. Although the method can be used in an online setting, it does not consider the commuting of mutex. Therefore, the detector may miss the predicate under a different locking schedule. The problem can be solved by incorporating the technique of RichTest [LC06], which uses a thread scheduler to control the threads and changes the acquisition order of locks. The technique ensures that every re-execution of the program produces a new poset of events. Therefore, RichTest and our online predicate detector are complementary tools.

The second part presents a fast sequential enumeration algorithm, named QuickLex, for consistent global states. In comparison with the original lexical algorithm, QuickLex incorporates a preprocessing procedure and dynamic programming to reduce the time complexity from $O(n^2)$ to $O(n \cdot \Delta(\mathcal{P}))$, where $\Delta(\mathcal{P})$ is the maximal in-degree of any event in the computation. Although QuickLex uses $O(n^2)$ space for dynamic programming, the additional space is insignificant from our experimental results.

The third part of this dissertation proposes a new model, named Loset, for modeling the computation with locking constraints. We have shown that the reachability problem is NP-complete in the loset model. To cope with the NP-completeness, we introduce the set of lock-free feasible global states and the set of strongly feasible global states, which are a lower and upper approximation of reachability, respectively, that can be calculated in polyno-

mial time. We also show that reachability equals to strong feasibility of global states in computations that contain at most two threads. Moreover, our experiments show that the gap between reachability and strong feasibility of global states seldom exists in practice. We also show that the set of viable global states could reduce the gap between the lower and upper approximation of the reachability of global states.

Altough the reachability of a global state in a parallel computation has also been solved using SAT/SMT solver [WKGG09, WLGG10, HZ11], these solvers take exponential amount of time in the worst case. Our techniques are orthogonal to techniques using SAT/SMT solvers and take only polynomial time. Specifically, given 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 its greatest preceding lock-free consistent global state. Moreover, we only need to calculate the reachability with a SAT/SMT solver only if $G$ is strongly feasible. Therefore, we could restrict the input of the SAT/SMT solver in a subcomputation rather than the entire computation.

# Chapter 8

# Future Work

This chapter describes the future work for ParaMount, QuickLex, and Loset model.

## 8.1 Future Work of ParaMount

In the context of distributed systems, the techniques of predicate detection for different kinds of condition have been extensively studied. These predicates can be roughly categorized into conjunctive predicates [GW94,HMRS96], linear and semi-linear predicates [CG98], relational predicates [TG97], restricted temporal logics [OG07,SG02], etc. Those techniques studies the properties of the predicates and the time complexity of the detection could be reduced to polynomial time because only a partial set of consistent global states is needed to be enumerated. Therefore, one possible future direction of ParaMount is developing online partitioning algorithms for different categories of predicates.

Another future direction for ParaMount is to optimize the construction the poset during runtime verification. One possible solution is incorporating

the technique of computation slicing (or simply *slicing*), which is commonly used technique in distributed debugging. Briefly, slicing is a technique that efficiently find all global states that satisfy the given predicate without enumerating all global states explicitly. Suppose that we wants to detect if the predicate $(x_1 + x_2 + x_3 < 10) \wedge (x_1 = 3) \wedge (x_2 = 3)$ can become true in one of the global states. We can compute a slice of the poset in which this part of the predicate, $(x_1 = 3) \wedge (x_2 = 3)$, is always true for all consistent global states. Then, we enumerate the lattice of consistent global states of the sliced poset, which is much smaller than the original lattice. Chauhan et al. [CGNM13] has proposed an online slicing technique for distributed computations. Therefore, one future direction of ParaMount is to apply the online slicing technique and reduce the size of the constructed poset.

## 8.2 Future Work of QuickLex

In QuickLex, the bottleneck of its time complexity is the PROPAGATE function, which takes $O(n \cdot \Delta(\mathcal{P}))$ time for finding the enabled event for the current iteration. Although the time complexity could be reduced to $O(n)$ for the commonly used computations [CSR08, LC06, FF09, HMNS01, JMN95], the time complexity might be $O(n^2)$ for general posets of computations, where an event may have $(n-1)$ incoming HB relations. One future direction for QuickLex is to investigate the possibility of reducing the time complexity of PROPAGATE function from $O(n \cdot \Delta(\mathcal{P}))$ to amortized constant time.

## 8.3 Future Work of Loset

One future work of Loset model is to study the problem of online-and-parallel predicate detection using the loset model. Without considering the compatibility of global states, the consistent global states of a loset also forms a distributive lattice, which means that we could use ParaMount to partition the lattice into multiple subsets. After that, we could use the strong feasibility to prune the consistent global states that are unreachable. However, Theorem 13 shows that the calculation of strong feasibility can be bounded between two lock-free feasible global states. So, the calculation may need the events which happen after the current global state. Besides parallelism, the online property is limited by the normalization procedure of the loset mode, which may also need events that happen after the current global state of the system. Therefore, two possible future work are to develop a partition method to ensure all required events for the calculation are included; and to develop an online normalization procedure for losets.

Another future work is to investigate the equivalence between the reachability and the strongly feasibility. Specifically, we are interested in the upper bound of the number of threads with which the reachability is still equivalent to the strong feasibility of a global state. In addition, we are also interested in finding the sets of global states that can further reduce the gap while the approximate reachability of a global state can still be calculated in polynomial time.

# Bibliography

[ASM] ASM. A java bytecode engineering library.

[AV01] Sridhar Alagar and Subbarayan Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Transactions on Software Engineering*, 27:412–417, 2001.

[CG98] C. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[CG15a] Yen-Jung Chang and Vijay 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.

[CG15b] Yen-Jung Chang and Vijay K. Garg. Quicklex: A fast algorithm for consistent global states enumeration of distributed computations. In *International Conference On Principles of Distributed Systems*, 2015.

[CGNM13] Himanshu Chauhan, Vijay K. Garg, Aravind Natarajan, and Neeraj Mittal. A distributed abstraction algorithm for online predicate

detection. In *Symposium on Reliable Distributed Systems*, pages 101–110, 2013.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

[CSR08] Feng Chen, Traian Florin Serbanuta, and Grigore Roşu. jPredictor: a predictive runtime analysis tool for java. In *Proceedings of the International Conference on Software Engineering*, pages 221–230, 2008.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[DP90] B. A. Davey and H. A. Priestley. Introduction to lattices and order. In *Cambridge University Press*, Cambridge, UK, 1990.

[FF09] Cormac Flanagan and Stephen 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.

[Fid88] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the Australian Computer Science Conference*, pages 56–66, 1988.

[FNU03] 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.

[Gan10] Bernhard Ganter. Two basic algorithms in concept analysis. In *Proceedings of the International Conference on Formal Concept Analysis*, pages 312–340, 2010.

[Gar03] Vijay 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.

[Gar06] Vijay K. Garg. Algorithmic combinatorics based on slicing posets. *Theor. Comput. Sci.*, 359(1-3):200–213, 2006.

[Gar15] Vijay K. Garg. *Introduction to Lattice Theory with Computer Science Applications*. John Wiley & Sons, Inc., 2015.

[GW91] Vijay K. Garg and B. Waldecker. Detection of unstable predicates. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, 1991.

[GW94] Vijay K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, 1994.

[HMNS01] Michel Habib, Raoul Medina, Lhouari Nourine, and George Steiner. Efficient algorithms on distributive lattices. *Discrete Appl. Math.*, 110(2-3):169–187, 2001.

[HMRS96] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunctions of local predicates in asynchronous computations. In *Proceedings of the IEEE Symposium on Parallel and Distributed Processing*, pages 588–594, 1996.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[HZ11] Jeff Huang and Charles Zhang. Persuasive prediction of concurrency access anomalies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 144–154, 2011.

[JMN95] Roland Jegou, Raoul Medina, and Lhouari 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.

[KIG05] Vineet Kahlon, Franjo Ivancic, and Aarti Gupta. Reasoning about threads communicating via locks. In *Proceedings of Inter-*

*national Conference on Computer Aided Verification*, pages 505–518, 2005.

[KW10] Vineet Kahlon and Chao 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.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[LC06] Y. Lei and R.H. Carver. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering*, 32(6):382–403, 2006.

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[Mat88] Friedemann 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.

[MQ07] Madanlal Musuvathi and Shaz 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.

[MR10] Patrick Meredith and Grigore Roşu. Runtime Verification with the RV system. In *the International Conference on Runtime Verification*, volume 6418, pages 136–152, 2010.

[MSG07] Neeraj Mittal, Alper Sen, and Vijay K. Garg. Solving computation slicing using predicate detection. *IEEE Transactions of Parallel Distributed Systems*, 18(12):1700–1713, 2007.

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[OG07] Vinit A. Ogale and Vijay K. Garg. Detecting temporal logic predicates on distributed computations. In *Proceedings of International Symposium in Distributed Computing*, pages 420–434, 2007.

[PLZ09] Soyeon Park, Shan Lu, and Yuanyuan Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the International Conference on Architectural support for programming languages and operating systems*, pages 25–36, 2009.

[PR93] Gara Pruesse and Frank Ruskey. Gray codes from antimatroids. *Order 10*, pages 239–252, 1993.

[SBN+97] 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.

[SFM10] Francesco Sorrentino, Azadeh 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.

[SG02] A. Sen and V. K. Garg. Detecting temporal logic predicates on the happened-before model. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002.

[Squ95] Matthew B. Squire. Enumerating the ideals of a poset. In *PhD Dissertation, Department of Computer Science, North Carolina State University*, 1995.

[Ste86] George Steiner. An algorithm to generate the ideals of a partial order. *Oper. Res. Lett.*, 5(6):317–320, 1986.

[Tar00] Ashis 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.

[TG97] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, 1997.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[vPG01] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 70–82, 2001.

[WKGG09] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. *Formal Methods*, 29:256–272, 2009.

[WLGG10] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti 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.

[YNPP12] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 485–502, 2012.

# Vita

Yen-Jung Chang was born in Taipei, Taiwan on 14 July 1983, the son of Dr. Jung-Feng Chang and Su-Luwan Lee. He received the Bachelor of Science degree in Computer Science from the National Chiao Tung University at Hsinchu in May, 2002 and the Master of Science degree in Computer Science from the National Tsing Hua University at Hsinchu in July, 2008. He has been employed at Synopsys, Inc.; Intel Corporation; VMware, Inc.; and LinkedIn in the summers of 2007, 2012, 2013, 2014, respectively. He was also employed at TASS Consultant Group from 2008 to 2010. Thereafter, he started pursuing a Ph.D. degree in the University of Texas at Austin. He won the 1st place in the ISPD Global Routing Contest in 2008 and was awarded Best Thesis Award by Taiwan IC Design Community in 2008, SpringSoft Scholarship Award by SpringSoft, Inc. in 2008 and 2010, National Science Foundation Student Award by NSF in 2014, and Teaching Award by the University of Texas at Austin in 2015.

Permanent address: c/o Mrs. Su-Luwan Lee
1F., No.6, Ln. 65, Hougang 1st Rd.,
Xinzhuang Dist., New Taipei City 24259, Taiwan

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.