# Systematic Concurrency Testing with Maximal Causality

Qingzhou Luo
University of Illinois at Urbana-Champaign
qluo2@illionis.edu

Jeff Huang
Texas A&M University
jeff@cse.tamu.edu

Grigore Rosu
University of Illinois at Urbana-Champaign
grosu@illionis.edu

## Abstract

We propose the first systematic concurrent program testing approach that is able to cover the entire scheduling space with a *provably minimal* number of test runs. Each run corresponds to a distinct *maximal causal model* extracted from a given execution trace, which captures the largest possible set of causally equivalent legal executions. The maximal causal models can be represented using first-order logic constraints, and testing all the executions comprised by a maximal causal model reduces to offline constraint solving. Based on the same constraint model, we also develop a schedule generation algorithm that iteratively generates new casually different schedules. The core idea is to systematically force previous read operations to read different values, thus enumerating all the causal models. We have implemented our approach in an explicit stateless model checker, and our evaluation showed that our technique is able to 1) find concurrency bugs faster; 2) finish state space exploration with much fewer schedules than previous techniques.

## 1. Introduction

Concurrent programs are becoming more and more popular with the advent of multi-processors era. Well designed concurrent programs can greatly increase the overall throughput of the system. However, it is also very hard to verify the correctness of concurrent programs because their inherent nondeterminism. Bugs in concurrent programs could be caused by interaction between multiple threads; such bugs are hard to be found and reproduced because (1) the number of all the possible thread interleavings is huge and (2) thread scheduling is usually non-deterministic.

Therefore, a fundamental challenge in testing concurrent programs is how to effectively cover the astronomical thread interleaving or scheduling space to either find out the buggy thread interleaving or prove the program is correct. In theory, a bug may be hidden anywhere in the state space and finding it is as hard as finding a needle in a haystack. Worse, the diversity of the exercised interleavings tends to be highly correlated with the execution environments [18, 24]. Naively executing the program on the same platform repeatedly (such as stress testing) results in redundant exploration of similar interleavings, keeping the buggy interleaving space still uncovered.

Systematic testing approaches [6, 14, 17, 18, 22, 24] offer a more promising solution for testing concurrent programs. It avoids testing repeated interleavings by actively controlling the thread scheduler to systematically explore all legal but distinct interleavings. If a buggy interleaving is hit during the exploration, then that interleaving can be used to reproduce the bug. If no buggy interleaving is found after the exploration finishes, then the concurrent program is proven to be correct. Systematic testing is more effective than blindly repeated executing the concurrent program because it guarantees that each test execution covers a different interleaving. However, the core challenge still remains: to cover the astronomical scheduling space, the same astronomical number of test executions must be done.

Researchers have proposed various methods to reduce the exploration space for systematic testing approaches. For example, context bounding techniques [14, 17] limit the number of preemptions each explored interleaving could have and coverage-driven techniques [22, 24] and priority-based techniques [6, 11, 18] prioritize schedules during exploration. Those techniques have been proven to be effective for finding certain bugs in concurrent programs. However they only try to select or prioritize schedules in exploration space, so those techniques cannot prove the program is correct and may also miss bugs.

Several researchers have also proposed partial-order reduction techniques [7, 8] to reduce the cost of state space exploration. The idea is to only explore schedules that lead to different program states. For example, dynamic partial order reduction (DPOR) techniques [7] prune state space by looking at all the currently active transitions and only explore one of them if they do not interfere with each other. This technique could indeed greatly reduce state space exploration cost. However, it is based on *happens-before* casaulity and thus it does not prune the possible maximal number of interleavings. In other words, many interleavings explored by DPOR could possibly belong to the same maximal causal model [19].

We propose a new approach that systematically explores the interleaving space with significantly less number of test

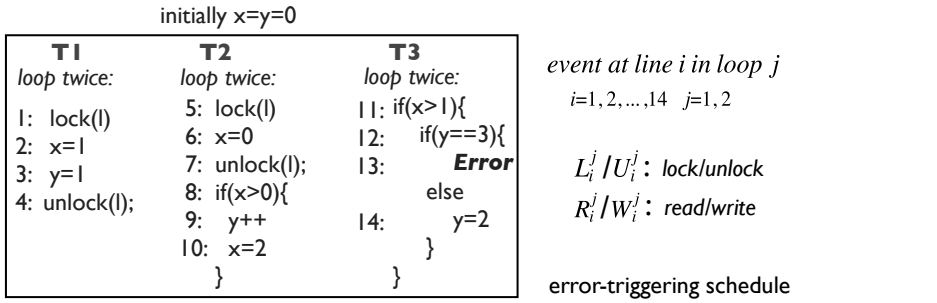


Figure 2: Example

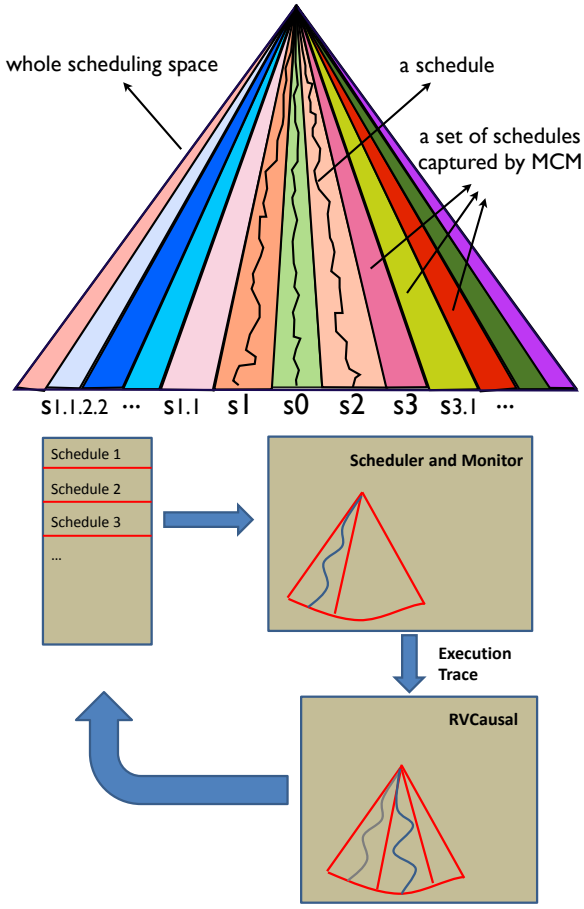


Figure 1: Overview

executions. Our key insight is to look at the state space exploration problem from the perspective of the *causal model* (instead of interleavings), which characterizes a set of legal interleavings that are causally equivalent and can be derived from each other. These causal sets have the important property that if any single interleaving is tested then there is no need to test any other interleaving in the same causal set. Moreover, the checking of each causal set can be done offline and in parallel, so our technique is particularly suitable when online testing is more expensive than offline checking.

Generally speaking, the classical happens-before relationship [13] (HB) yields such a causal model. HB characterizes the set of interleavings in which the order between operations can be altered if they have no HB relation. However, HB is rather strict, in that its power of characterizing causality is quite limited. Instead, our approach builds upon the *maximal causal model* [19] (MCM) technique, which yields the largest possible causal equivalence classes. In other words, from any single execution trace, MCM is able to derive the largest causal set of legal interleavings w.r.t. that trace. Underpinned by this property, our approach minimizes the number of executions that are needed to run in order to cover the entire scheduling space w.r.t. a given input.

Figure 1 shows an overview of our approach. Given a certain input, starting with any schedule, our approach systematically covers the entire interleaving space w.r.t. the input by iteratively generating and executing new schedules. In each iteration, our core component, RV-CAUSAL, takes the trace (an ordered sequence of events) emitted from executing a schedule on the program with our scheduler and monitor, computes a causal set of interleavings corresponding to the schedule according to MCM, and generates new schedules that are not in this causal set. Each causal set is distinct and accounts for a different subspace of the whole scheduling space. To enable checking runtime properties (i.e., safety and liveness properties) over this causal set offline, we encode MCM as a formula of first order logical constraints over a set of order variables (denoting the possible order of each event in the execution), such that any solution to the formula corresponds to a legal interleaving represented by the value of order variables. By encoding the runtime properties as additional constraints and solving a conjunction of the formula and the property constraints with an SMT solver, we can determine whether a property holds or not for all the interleavings in the causal set.

A main technical challenge here is how to systematically generate new schedules such that: 1) no two subspaces (corresponding to two different traces) overlap, and 2) all the subspaces together cover the entire scheduling space. Our approach works by pivoting around the value of reads in the trace. Specifically, we ensure that each generated new schedule has at least one new event: a read event that reads a new data (i.e., a different value from that in other schedules). All such new events are considered and their corresponding schedules are generated, as long as the schedule is legal (permitted by MCM). In this way, we guarantee that no two schedules are redundant in terms of MCM, i.e., the corresponding trace of each schedule contains at least one distinct event compared to others. Moreover, because the generated schedules consider all possible legal combinations of read values, it guarantees that our approach would cover all state space eventually.

We make following contributions in this paper:

- A new systematic concurrency testing approach that leverages the maximal causal model to minimize the number of executions needed to execute, and shifts the runtime computation cost to offline inference and property checking through constraint solving.
- A schedule generation technique that systematically generates new schedules that cover distinct thread interleavings until the whole scheduling space is covered.
- A set of evaluation shows that our technique is able to find concurrency bugs and explore the entire state space more efficient and effective than existing techniques.

## 2. Motivating Example

We illustrate our approach using the example in Figure 2. Three threads `T1`, `T2` and `T3` are started concurrently, each one has an outer loop with two iterations. `x` and `y` are shared variables among those threads. An error will be triggered in `T3` if (1) `y == 3` and (2) `x > 1` are both satisfied, as shown at line 12. For (1) to be true, line 9 must be executed after line 14; for (2) to be true, line 2 must be executed between line 7 and line 8. The exact buggy schedule is described in Figure 2 with four different types of events: $R$ (Read), $W$ (Write), $L$ (Lock) and $U$ (Unlock), and each event is annotated with line number and loop iteration number. For example, $R_8^2$ means the read event at line 8 in the second loop iteration.

This bug is hard to find by existing state space exploration tools. The reason is that there is a large number of schedules in this program and the bug is hidden deeply with complex thread interleavings. We ran this program with a stateless exploration tool similar to CHESS [17] and used the same iterative context bounding exploration strategy, which is usually the most efficient for finding concurrency bugs. It took 58478 schedules until it hit the bug.

Our approach differs from existing techniques by covering a *set* of schedules from one *single* execution. With the maximal causal model [19] as the foundation, our approach is able to analyze an exponential and provably *maximal* number of schedules derived from one single execution trace. From a high level view, our approach works in an iterative manner. In each iteration, we work on one schedule and generate more new schedules, which consists of three functional steps:

***Record trace from one execution:*** We record in this step the necessary information for constructing the maximal causal model. All the reads and writes to shared data will be recorded as well as their value operands. Note that the trace collected here is not required to hit the bug. For example, in the second iteration of `T2`, at line 9 we may read the value 1 for `y`. This will not trigger the error in line 13.

***Generate causally different schedules:*** We use the trace collected in the previous step to construct a maximal causal model. Each maximal causal model contains a set of schedules and our approach will analyze those schedules offline. However, this is still not enough to cover the entire state space. A key novelty of our approach is to systematically generating causally different schedules by forcing reads in the program to *read different values*. For example, suppose we have a trace $R_9^2$ reads value 1 and $W_{14}^1$ writes value 2, and $R_9^2$ happens before $W_{14}^1$. Our approach will try to force $R_9^2$ to read a different value, 2 in this case, written by $W_{14}^1$. All the corresponding constraints will be generated and solved by an SMT solver. If they are satisfiable, such a schedule will be generated.

Note that there may be multiple read operations in the program, so our approach will generate multiple schedules from one input trace. In each generated schedule there will be *at least one* read operation that reads a different value from the original trace, thus guaranteeing that each generated schedule falls into a different causal model.

Figure 3 illustrates the schedule generation for our example. `S0` is the schedule in the initial trace. In the first iteration, we generate four new schedules (`S1`, `S2`, `S3`, `S4`), which enforce the four reads ($R_8^1$, $R_8^2$ $R_{11}^1$, and $R_{11}^2$), respectively, to read value 1. In the second iteration, we continue to work on the traces corresponding to $\texttt{S}_i$ ($i$=1,2,3,4 in parallel), and generate `S1.1`, `S1.2`,...,`S2.1`, `S2.2`, etc. All schedules form a hierarchy, with each child schedule enforcing a different read value. Again, each new schedule may produce a trace containing new read events and/or write values, which can generate new children schedules. For example, our approach will eventually generate the schedule `S1.1.2.2`, which enforces $R_{12}^2$ to read 3 and triggers the error.

***Re-execute program following generated schedules:*** our generated schedules contain the execution order of threads so they can be used as input for our scheduler to re-execute the program. All the generated schedules will be place into a priority queue. After executing a new schedule in the queue, more causally different schedules may be

generated and put in the queue. Our approach will terminate when the queue becomes empty, meaning that there is no more causally different schedule. This is the indication that the entire state space is covered.

# 3. Approach

## 3.1 Maximal Causal Model

Our approach builds upon the *maximal causal model* (MCM) foundation, first presented in [19] (for sequential consistency). We briefly review it below.

Multithreaded programs $\mathcal{P}$ are abstracted as the prefix-closed sets of finite traces that they can produce when completely or partially executed, called $\mathcal{P}$*-feasible* traces. A *trace* is abstracted as a sequence of events. *Events* are operations performed by threads on concurrent objects, abstracted as tuples of *attribute-value* pairs. For example, $(\mathit{thread} = t_1, \mathit{op} = \mathit{read}, \mathit{target} = x, \mathit{data} = 1)$ is a read event by thread $t_1$ to memory location $x$ with value 1. We consider the following common event types:

- $\mathit{begin}(t)$/$\mathit{end}(t)$: the first/last event of thread $t$;
- $\mathit{read}(t, x, v)$/$\mathit{write}(t, x, v)$: read/write a value $v$ on a variable $x$;
- $\mathit{lock}(t, l)$/$\mathit{unlock}(t, l)$: acquire/release a lock $l$;
- $\mathit{fork}(t, t')$: fork a new thread $t'$;
- $\mathit{join}(t, t')$: block until thread $t'$ terminates;

The sets of $\mathcal{P}$-feasible traces must obey some basic consistency axioms. We proposed two axioms: *prefix closedness* and *local determinism*. The former says that the prefixes of a $\mathcal{P}$-feasible trace are also $\mathcal{P}$-feasible. The latter says that each thread has a deterministic behavior, that is, only the previous events of a thread (and not other events of other threads) determine the next event of the thread, although if that event is a read then it is allowed to get its value from the latest write. These two axioms allow us to associate a causal model $\mathit{feasible}(\tau)$ to any consistent trace $\tau$, which comprises precisely the traces that can be generated by any program that can generate $\tau$. As shown in [19], $\mathit{feasible}(\tau)$ is both *sound* and *maximal*: any program which can generate $\tau$ can also generate all traces in $\mathit{feasible}(\tau)$, and for any trace $\tau'$ not in $\mathit{feasible}(\tau)$ there exists a program generating $\tau$ which cannot generate $\tau'$. Comparatively, conventional happens-before causal models consisting of all the legal interleavings of $\tau$ and their prefixes are *not* maximal [19].

In our approach, we realize MCM using constraints and represent $\mathit{feasible}(\tau)$ by a formula $\Phi$ of first order logic clauses over a set of order variables, each of which corresponds to an event in $\tau$. Any solution to $\Phi$ denotes a legal schedule that can produce a corresponding trace in $\mathit{feasible}(\tau)$. We next describe our constraint modeling.

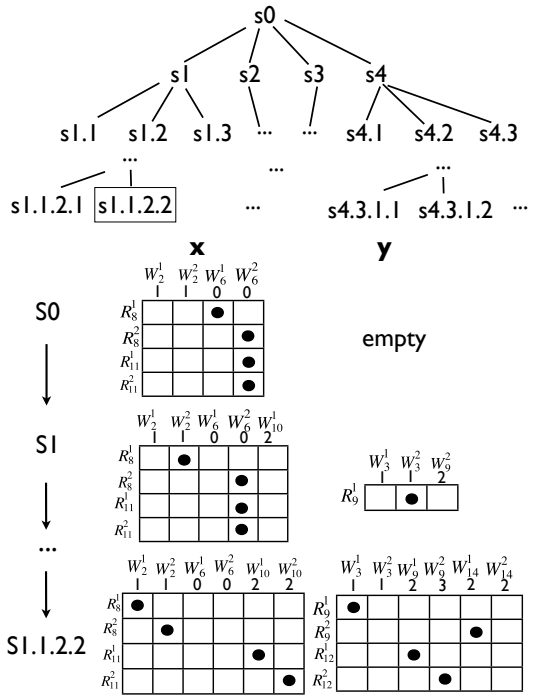


Figure 3: Technical overview of schedule generation

## 3.2 Constraint Modeling

From a high level view, $\Phi$ contains only variables of the form $O_e$ corresponding to events $e$, which denote the order of the events in a trace in $\mathit{feasible}(\tau)$. $\Phi$ is constructed by a conjunction of three sub-formulas: $\Phi = \Phi_{mhb} \wedge \Phi_{lock} \wedge \Phi_{rw}$.

***Must happen-before constraints*** **($\Phi_{mhb}$)** The must happen-before (MHB) constraints requires that (1) the total orders of the events in each thread are always the same; (2) a *begin* event can happen only as a first event in a thread and only after the thread is forked by another thread; (3) an *end* event can happen only as the last event in a thread, and a *join* event can happen only after the *end* event of the joined thread. MHB yield an obvious partial order $\prec$ on the events of $\tau$ which must be respected by any trace in $\mathit{feasible}(\tau)$. We can specify $\prec$ easily as constraints over the $O$ variables: we start with $\Phi_{mhb} \equiv \mathit{true}$ and conjunct it with a constraint $O_{e_1} < O_{e_2}$ whenever $e_1$ and $e_2$ are events by the same thread and $e_1$ occurs before $e_2$, or when $e_1$ is an event of the form $\mathit{fork}(t, t')$ and $e_2$ of the form $\mathit{begin}(t')$, etc.

***Locking Constraints*** **($\Phi_{lock}$)** Lock mutual exclusion semantics requires that two sequences of events protected by the same lock do not interleave. $\Phi_{lock}$ captures the ordering constraints over the lock *lock* and *unlock* events. For each lock $l$, we extract the set $S_l$ of all the corresponding pairs $(a, r)$ of *lock*/*unlock* events on $l$, following the program order locking semantics: the *unlock* is paired with the most recent *lock* on the same lock by the same thread. Then we

conjunct $\Phi_{lock}$ with the formula

$$\bigwedge_{(a,r),(a',r')\in S_l} (O_r < O_{a'} \vee O_{r'} < O_a)$$

***Read-write constraints*** **($\Phi_{rw}$)** The read-write constraints ensure that every event in the trace is feasible. For an event to be feasible, all the events that must happen-before it should also be feasible. Moreover, any read event that must happen-before it should read the *same value* as that in the original trace. Consider a read event $r$, say $read(t, x, v)$, we let $W^r$ be the set of $write(\_, x, \_)$ events in $\tau$ (here '-' denotes any value), and $W^r_v$ the set of $write(\_, x, v)$ events in $\tau$, then we have the formula defining its feasibility as following:

$$\Phi_{rw}(r) = \bigvee_{w\in W^r_v} (\Phi_{rw}(w) \wedge O_w < O_r \bigwedge_{w\neq w'\in W^r} (O_{w'} < O_w \vee O_r < O_{w'}))$$

The above states that the read event $r = read(t, x, v)$ may read the value $v$ on $x$ written by any *write* event $w = write(\_, x, v)$ in $W^r_v$ (the top disjunction), subject to the condition that the order of $w$ is smaller than that of $r$ and there is no interfering $write(\_, x, \_)$ in between. Moreover, $w$ itself must be concretely feasible, which is ensured by $\Phi_{rw}(w)$. Similarly, $\Phi_{rw}(w)$ is defined by requiring all the reads that must happen-before it are feasible. $\Phi_{rw}$ is a conjunction of $\Phi_{rw}(r)$ for all reads in the considered trace.

### 3.3 Schedule Generation

The goal of schedule generation is to generate schedules that produce traces not in $feasible(\tau)$. Intuitively, this problem is the opposite of constraint modeling in Section 3.2 which encodes $feasible(\tau)$. Hence, we can directly leverage the constructed formulae in $\Phi$ and negate those that can be negated. Clearly, the only type of such constraints is $\Phi_{rw}$, in which the mapping from read to write may be changed: rather than enforcing a read to read the same value as that in $\tau$, we can instead enforce it to read a different value. The new formula $\Phi'$ then encodes a feasible schedule that can produce a different trace (with at least one new event: the read event with a different value). When being re-executed, this new event might change the control flow of the thread, producing more new events. A caveat of this process is that when enforcing a read to read a different value, we must make sure all the reads that must happen before it are matched with writes that write the same values as that in $\tau$. Otherwise, this read event may not be feasible.

Therefore, our algorithm enumerates each read event in $\tau$ on the set of all values by the writes on the same variable. For each value that is different from what it reads in $\tau$, we construct $\Phi'$ that constrains the read to read the value. We then invoke a constraint solver (such as Z3) to solve $\Phi'$. If the solver returns a solution, the solution represents a new schedule which is feasible and in which the read will read that new value. Note that each read only concerns about the distinct values but not distinct writes. If there are multiple writes writing the same value, it suffices to generate only one new schedule for all of them. This is another salient advantage of our approach: it avoids generating redundant schedules that have the same effect on the program state.

An important property of our algorithm is that it would eventually cover the entire scheduling space.

PROOF. (Sketch) For each read, for each new value it can read, our approach generates a new schedule. Suppose there exists a schedule $s$ not covered, then it must be the case that $s$ contains a new event. There could only be two possibilities for this new event: (1) it is a previously observed event, but reads a new value; (2) it is a previously unseen event. The case (1) is actually impossible, because our algorithm guarantees generating a new schedule for each such read. For (2), it must be the case that the event depends on a branch, the condition of which none of our generated schedules satisfies. However, this means that the branch condition depends on at least one previous read reading a new value, which contradicts to the fact that we already generated one schedule for each read with a different value. □

## 4. Implementation

### 4.1 Overflow

Our implementation is on top of ReEx [11], a stateless state space exploration tool. ReEx is a Java framework used to re-execute multithreaded Java programs based on different exploration strategies. It already has a set of exploration strategies, such as iterative context bounding exploration strategy (Chess) and depth first exploration strategy. We implement our technique as another strategy in ReEx. We use ASM to instrument Java bytecode, such that after each execution all the necessary information is stored in a trace object. In our implemented exploration strategy that trace object serves as input to build constraint model. We solve all the constraints (using Z3) to generate new schedules such that read operation will read a different value. All the new generated schedules will be put in a queue and our exploration strategy will pick the next schedule in the queue to re-execute the program and generate new trace objects. After the queue becomes empty, no new schedule will be generated and the exploration will finish.

### 4.2 Generation of Read Write Matching Pairs

The main part of our implementation is to generate all the possible read write matching pairs, such that the only one read will read a different value, while all other preceding reads read the same values. The algorithm is described in Figure 4.

The basic idea of the algorithm is to find all the values a specific read could possible read of, and then recursively generate matching pairs for all those values. Line 5, `getDependentNodes` returns all the nodes that *should happen before* the input node. Those nodes need to appear if the input node will appear in the new schedule. Figure 5 de-

```
1  // Global variables
2  boolean foundSchedule = false;
3
4  // Get dependent nodes
5  Set⟨AbstractNode⟩ getDependentNodes(Trace trace, AbstractNode node) {
6    // return all the nodes that should happen before the current node
7  }
8
9  // Construct read write pairs
10 void constructAllReadWritePairs(Trace trace, ReadNode targetReadNode) {
11   Set⟨AbstractNode⟩ dependentNodes = getDependentNodes(trace, targetReadNode);
12   Set⟨String⟩ allValues = trace.getAllWriteValuesOnAddr(targetReadNode);
13   for (value ∈ allValues) {
14     if (value == targetReadNode.value) {
15       // match targetReadNode with a different value than the previous trace
16       continue;
17     }
18     foundSchedule = false;
19     constructReadWritePairs(trace, getReadNodes(dependentNodes), dependentNodes,
20       targetReadNode, value, new HashMap⟨ReadNode, AbstractNode⟩());
21   }
22 }
23
24 void constructReadWritePairs(Trace trace, List⟨AbstractNode⟩ readNodes,
25   Set⟨AbstractNode⟩ dependentNodes, ReadNode targetReadNode, String value,
26   Map⟨ReadNode, AbstractNode⟩ readWriteMapping) {
27   if (foundSchedule) {
28     return;
29   }
30   if (readNodes == ∅) {
31     if (readWriteMapping != ∅) {
32       if (canGenerateSchedule(readWriteMapping, targetReadNode, value)) {
33         generateSchedule(readWriteMapping, targetReadNode, value);
34         foundSchedule = true;
35       }
36     }
37     return;
38   }
39
40   ReadNode currentReadNode = removeFirst(readNodes);
41
42   // Already matched or impossible to match currentReadNode
43   while (currentReadNode ∈ readWriteMapping.keySet() || (currentReadNode != targetReadNode
44     && trace.getWriteNodesWithSameValue(currentReadNode) == ∅) {
45     if (readNodes == ∅) {
46       currentReadNode = removeFirst(readNodes);
47     } else {
48       if (readWriteMapping != ∅) {
49         if (canGenerateSchedule(readWriteMapping, targetReadNode, value)) {
50           generateSchedule(readWriteMapping, targetReadNode, value);
51           foundSchedule = true;
52         }
53       }
54       return;
55     }
56   }
57
58   Set⟨WriteNode⟩ writeNodes = trace.getWriteNodesWithSameValue(currentReadNode);
59   for (writeNode ∈ writeNodes) {
60     // Optimizations to prune impossible cases
61     readWriteMapping.put(currentReadNode, writeNode);
62     readNodes.addAll(getReadNodes(getDepedentNodes(trace, writeNode)));
63     dependentNodes.addAll(getDepedentNodes(trace, writeNode));
64     constructReadWritePairs(trace, readNodes, dependentNodes, targetReadNode, value,
               readWriteMapping);
65   }
66
67   // Handle matching with initial values
68 }
```

Figure 4: Generate Read Write Matching Pairs

scribes how we compute dependency nodes. The first case is that two nodes are in the same thread following the program order, then the later node must happen after the early node. In the second case the first thread starts the second thread, then the start node should happen before the very first node from the second thread. In the third case one thread joins on another thread, which means the last node of the joined thread should happen before the join node. In the last case we handle the semantics of `wait-notify` by modeling `wait` operation as `wait` followed by `unlock` and `lock`. Therefore, the wait node should happen before notify node, and notify node should happen before the next node (lock node) following wait node in that thread. By taking into consideration of all the possible *should happen before* relationships in the program, we transitively compute all the dependency nodes for a given node and use that result in the main algorithm.

On line 10, `constructAllReadWritePairs` gets all the values that were written to the same address in previous trace, and calls `constructReadWritePairs` to get the

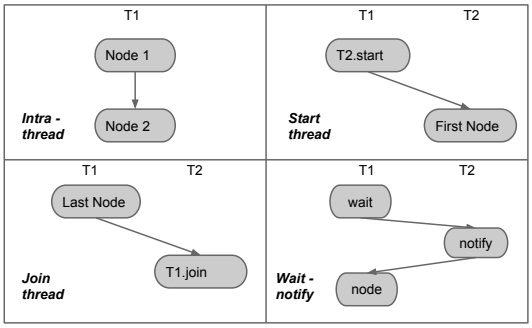


Figure 5: Compute dependency nodes

mappings between a read to a specific write node. On line 24, `constructReadWritePairs` takes as input a list of read nodes to be matched, and recursively generates and stores the result in a map. This algorithm will terminate on line 37 and 54 when the list of read nodes is empty. Note that for each read node paired with each possible value it could read, we only need to generate one valid schedule. So we use the global variable `foundSchedule` to terminate the execution earlier in that case for optimization purpose. When the list of read nodes is not empty, the algorithm will get the first read node and pair it with a write node with the same value, put this information in the result map, and recursively execute the algorithm.

***Handling matching with initial values:***

One problem we encountered during implementation is that for uninitialized variables, JVM will use default values for their data types. Those values are not written by any write nodes, therefore they are not captured in the trace. We handle those nodes in a specific way by using a map to store its initial values. When accessing those variables, we will not only look for existing write nodes in the trace, but also search for the map to find its initial values. For the simplicity purpose, the details of this part is not included in the presented algorithm here.

***Optimization:***

If we naively pair all the read nodes with possible write nodes in the trace, we may end up with too many possible pairings. Many of those pairings are impossible because of program order constraints. However, we also do not want to check and prune too many impossible pairings in our algorithm, because SMT solver is used to check validity and generate valid schedule. In our implementation, we have done some simple Optimization to quickly check and prune some cases. For example, if there is a cycle between two read write pairs from two different threads, then it is impossible because we are using sequential consistent memory model. Also, if another write nodes that writes a different value to the same address and it should happen before the paired

Table 1: Subject Faults and Programs Statistics

| | Source | Error |
|---|---|---|
| Airline | [20] | Assertion Violation |
| Account | [20] | Assertion Violation |
| Allocation | [20] | Assertion Violation |
| BubbleSort | [20] | Assertion Violation |
| Lang | [1] | Assertion Violation |
| Pool | [4] | Assertion Violation |
| Log4J1 | [3] | NullPointer Exception |
| Log4J2 | [2] | NullPointer Exception |
| Logger | [12] | NullPointer Exception |

read node and after the paired write node, then it is also impossible for the read node to read the value from the paired write node. The details of those optimization are also omitted on line 60. In practice, those simple optimization give us good performance improvement without losing the benefits of using SMT solver.

## 4.3 Evaluation

### 4.3.1 Methodology and Subjects

In order to evaluate our technique, we have performed two sets of experiments. First we want to see how RV-CAUSAL would help detecting concurrency bugs. Second, we want to see how RV-CAUSAL would reduce the cost of state-space exploration.

Our subject programs are described in table 1. We collected several concurrency programs from SIR [20] and also from several large open source systems. Each program has a concurrency bug caused by thread scheduling, associated with a multithreaded unit test which will expose the bug. However, those unit tests will only fail under certain thread schedules. We clarify those bugs into two categories. The first one is `Assertion Violation`, caused by test oracles being violated at the end of the test execution; the second one is `NullPointer Exception`, caused by dereferencing null pointer in test execution.

RV-CAUSAL explores only one schedule in each causal model, and it is useful to find those concurrency bugs because `Assertion Violation` and `NullPointer Exception` are both caused by a read operation matched with a "wrong" operation in one execution. Therefore, it suffices to explore only one schedule in each causal model to reveal those bugs. We think most concurrency bugs fall in those categories except `Deadlock`. We leave that to future work for RV-CAUSAL.

The main objective of our evaluation is to first see how many thread schedules it would take for RV-CAUSAL to hit those bugs, compared with other existing techniques and tools. We then fix those bugs and see how many schedules RV-CAUSAL would take to finish exploring the entire state space. This shows how much state space reduction we can gain by using the maximal causal model.

Table 2: Num of Schedules to Find Bugs

| | DEPTH FIRST SEARCH | CHESS | RV-CAUSAL |
|---|---|---|---|
| Airline | 7 | 42 | 3 |
| Account | 196 | 14 | 4 |
| Allocation | TIMEOUT | 15 | 2 |
| BubbleSort | TIMEOUT | 166 | 9 |
| Lang | TIMEOUT | 18 | 20 |
| Pool | TIMEOUT | 128 | 164 |
| Log4J1 | 123 | 8 | 3 |
| Log4J2 | 20 | 29 | 5 |
| Logger | 20 | 9 | 3 |

During our experiments, we compare RV-CAUSAL with DepthFirst Strategy (DFS) and Iterative Context Bounding Strategy [15] (Chess) implemented in ReEx [11]. We choose number of schedules as the metric in our experiments because it was proven to be effective for evaluating state-space exploration techniques in previous work [10].

### 4.3.2 State-Space Exploration Results

**Find concurrency bugs**

Table 2 summarizes our results of using RV-CAUSAL to find concurrency bugs, compared with DFS and Chess. We set the time limit to be 15 minutes for each subject program, if the exploration does not terminate within that time limit we mark it as `TIMEOUT`.

Overall RV-CAUSAL takes significantly fewer schedules to find bugs than using DFS or Chess exploration strategies. DFS is the basic exploration strategy, which exhaustively enumerates all possible thread schedules one by one until it hits the bugs. Because of the potential large number of possible thread interleavings, DFS cannot finish in time for 5 out of 9 subject programs. Chess is using iterative context bounding approach [15] with bound 2. It only explores schedules with preemptions less or equal to 2, so it does not guarantee to find the bug. However in practice it works very well, as it finds all the bugs in the 9 subject programs. RV-CAUSAL takes even fewer schedules to find those bugs in 7 out of 9 subject programs. That is because many schedules DFS and Chess explores fall into the same causal model, so that RV-CAUSAL will only execute one of them. By forcing a read operation to read a different value in each newly generated schedule, RV-CAUSAL is more likely to lead the program into a new state (e.g., executing a new branch or writing a different value to a shared memory location), thus easier to hit concurrency bugs.

For example, consider the code snippet used in `Allocation` example in Figure 6. A few threads are accessing the shared `resultBuf` array concurrently in `run` method. Using DFS or Chess strategy, many context switch points will be created inside those `for` loops on line 2 and 7, even if threads are accessing different elements of the array. RV-CAUSAL will look for the actual dynamic memory locations in each trace and only generate new schedules which lead to different values written to `resultBuf` array,

```
1  void run() {
2      for (int i = 0; i < resultBuf.length; i++) {
3          resultBuf[i] =
4              vector.getFreeBlockAndMarkAsAllocated();
5      }
6
7      for (int i = 0; i < resultBuf.length; i++) {
8          if (resultBuf[i] != -1) {
9              vector.markAsFreeBlock(resultBuf[i]);
10         }
11     }
12 }
13
14 public int getFreeBlockAndMarkAsAllocated() {
15     // bug fix: synchronized (this) {
16         int freeBlockIndex = getFreeBlockIndex();
17         if (freeBlockIndex != -1) {
18             markAsAllocatedBlock(freeBlockIndex);
19         }
20         return freeBlockIndex;
21     // }
22 }
```

Figure 6: Allocation example

therefore it will not create those unnecessary context switch points.

**Explore entire state space**

Table 3 summarizes our results of using RV-CAUSAL to explore the entire state space on the fixed subject programs. Since all the concurrency bugs are fixed in those subject programs, ReEx will finish exploration only when all the possible thread interleavings are enumerated. Because of the exponential number of thread schedules for multithreaded programs, the naive DFS approach would not be able to finish exploration for 8 out of 9 subject programs. Chess, as a contrast, is able to finish exploration for most subject programs. However, that is due to the fact that Chess only explores thread schedules with preemptions less than 3 among all the possible thread schedules. Therefore, using the Chess approach could possibly miss concurrency bugs (although it was proven to be effective in practice and in our experiments).

RV-CAUSAL takes significantly less schedules to finish exploration in most subject programs, except in POOL where all three approaches could not finish exploration within the time limit. The improvement also comes from the fact that RV-CAUSAL only explores one schedule from each causal model. Consider the example in Figure 6 again. The bug was fixed by locking `getFreeBlockAndMarkAsAllocated` method. However Chess and DFS will still explore many alternate interleavings in other methods, resulting in a much larger number of schedules to finish exploration.

Note in some of our subject programs, RV-CAUSAL takes very few schedules to finish exploration. In those programs, developers fixed the bugs by wrapping accesses to shared variables with common locks, or using thread local variables instead of shared variables. In those cases, the total number of causal models decreased significantly compared with those programs before applying their fixes.

Table 3: Num of Schedules to Finish Exploration

| | DEPTH FIRST SEARCH | CHESS | RV-CAUSAL |
|---|---|---|---|
| Airline | TIMEOUT | 8309 | 17 |
| Account | TIMEOUT | 819 | 5 |
| Allocation | TIMEOUT | 16311 | 22 |
| BubbleSort | TIMEOUT | 115827 | 103 |
| Lang | TIMEOUT | 9990 | 334 |
| Pool | TIMEOUT | TIMEOUT | TIMEOUT |
| Log4J1 | 329 | 329 | 3 |
| Log4J2 | TIMEOUT | TIMEOUT | 9 |
| Logger | 577 | 138 | 2 |

## 4.4 Discussion

### 4.4.1 Comparison with Dynamic Partial-Order Reduction

Dynamic Partial-Order Reduction (DPOR) is a well known technique for reducing the cost of state-space exploration [7]. The main idea behind DPOR is to look for conflicting and co-enabled transition when program executes. Two transitions are conflicting with each other if at least one of them is a write operation. Whenever two transactions that are accessing the same memory location and are both enabled, a backtrack point will be created to explore the alternative path.

In [7], the authors presented the following example:

$$T1 : x = 1; x = 2;$$
$$T2 : y = 1; x = 3;$$

T1 and T2 are two different threads executing concurrently. Suppose the first interleaving is $< T1 - T1 - T2 - T2 >$. A backtrack point will be created after executing the first instruction in T1, resulting in interleaving $< T1 - T2 - T2 - T1 >$. Similarly, a backtrack point will be created before executing the first instruction in T1, resulting in interleaving $< T2 - T2 - T1 - T1 >$.

The rationale behind DPOR is that if two transitions are conflicting with each other, then executing them in different orders will lead the program into different states. However, the main difference between DPOR and RV-CAUSAL is that DPOR only looks at all the currently enabled transitions. In other words, it does not take into account the "causal effects" of those transitions. Back into the above example, DPOR would not consider whether there are any read operation that will read those values that being written earlier. Even if there is such a read operation, DPOR would also not consider whether it will read the same value or not. If there is another write operation writes to the same location but with a different value, then all the above interleavings would not show any causally difference.

RV-CAUSAL, on the other hand, looks for interleavings that will result the program fall into another different causal model. Therefore, for a write operation to be considered as

a backtrack point, there must be a read operation that reads its value; moreover, it must read a different value than in the previous execution. For example, executing the above program in RV-CAUSAL would only exercise one interleaving, since there is no read operation following those write operations. RV-CAUSAL achieves this goal by modeling the entire program execution and find a viable solution for its causal model. Therefore, RV-CAUSAL is able to further reduce the state space for exploration compared to DPOR.

#### 4.4.2 Deadlock Bugs

Currently our technique tracks each read and write values in the trace and generates different traces such that at least one read will read a new value, due to the definition of the maximal causal model.

Consider a simple example with nested locks: thread T1 acquires lock L1 first and then lock L2, and then it releases L2 followed by L1; thread T2 acquires lock L2 first then lock L1, and then releases L1 followed by L2. After the program finishes executing the first trace (suppose the program does not deadlock in the first trace), RV-CAUSAL will finish exploration because it could not find any new causally different trace. However this program could potentially deadlock if a context switch happens after each thread acquires their first lock.

The reason for our technique to miss this deadlock bug is that when constructing constraints, we only generate *synchronization consistent* traces. That is, we only generate traces in which each lock operation will successfully get the lock. The same goes for wait/notify operations in the program. We currently cannot generate schedules that manifest deadlocks caused by missing notification.

To solve this problem, we need to model lock/unlock and wait/notify operations differently. We will need to model those operations as special kinds of read/write operations and match them with different values in each execution. By doing this our technique will be able to explore schedules that could lead to deadlocks.

## 5. Related Work

Many work have been proposed for state space exploration of multithreaded programs. There are typically two ways to explore the state space: *stateful* search and *stateless* search. Stateless search [9, 21] models the state of the program when it executes and use the modeled states to check for errors. For example, Java PathFinder [21] is an explicit state space exploration tool for checking Java programs. It uses state comparison to do backtracking in its search process. Stateless search [11, 16] does not model the state of the program. Instead, it re-executes the program at all the possible choice points to enumerate all the possible output of program execution. Our work here is built on top of a stateless state space exploration tool ReEx, however it is possible to extend our work for stateful state space exploration tools.

Since the entire state space for a multithreaded program is large, it is usually infeasible to explore the whole state space. Researchers have proposed different heuristics to find concurrency bug faster when doing exploration. Chess [16] is built on top of the fact that most concurrency bugs can be found within a small number of preemptions. It then proposes an iterative preemption bounding approach to first explore schedules with a smaller number of preemptions. Follow up work of preemption sealing [5] limits preemptions in a set of selected methods to further improve the efficiency of finding concurrency bugs. Wang et el. proposes another heuristic which uses *PSet* coverage information as a guideline when exploring state space [23]. Our earlier work [11] employs a set of heuristics to utilize the change information between program revisions to find concurrency bugs faster. Compared to existing heuristic based work, we do not sacrifice coverage for efficiency when exploring state space. Since our approach is based on maximal causality model, we are able to find bugs faster while being guaranteed to cover all the possible behaviors of multithreaded programs.

Dynamic partial-order reduction [7] explores the relationship between enabled transitions during each step of state space exploration. If switching the order of two co-enabled transitions does not result in new program state, then it is safer to pick one instead of trying both of them. Our approach tracks the value of each read and write instructions and also takes into account all the constraints when building maximal causality model, which makes our approach subsumes previous partial-order reduction work.

## 6. Conclusion

In this paper we have presented RV-CAUSAL, a novel approach to systematically testing concurrent programs. Our approach differs from existing techniques by building upon the maximal causal model foundation in its state space exploration. Our technique is efficient because Only one schedule is analyzed from each maximal causal space; our technique is also effective because the entire scheduling space can be covered incrementally. Our implementation and evaluation have shown that RV-CAUSAL is able to find concurrency bugs and explore the entire state space much faster than existing techniques.

## 7. Acknowledgement

This work was supported in part by the NSF grant CCF-1218605, the NSA grant H98230-10-C-0294, the DARPA HACMS program as SRI subcontract 19-000222 and the Rockwell Collins contract 4504813093.

## References

[1] Apache Software Foundation. LANG-481. `https://issues.apache.org/jira/browse/LANG-481`.

[2] Apache Software Foundation. LOG4J-44032. https://issues.apache.org/bugzilla/show_bug.cgi?id=44032.

[3] Apache Software Foundation. LOG4J-509. https://issues.apache.org/bugzilla/show_bug.cgi?id=509.

[4] Apache Software Foundation. POOL-120. https://issues.apache.org/jira/browse/POOL-120.

[5] T. Ball, S. Burckhardt, K. Coons, M. Musuvathi, and S. Qadeer. Preemption sealing for efficient concurrency testing. In *TACAS*, 2010.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, 2010.

[7] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, 2005.

[8] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science.

[9] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[10] V. Jagannath, M. Kirn, Y. Lin, and D. Marinov. Evaluating machine-independent metrics for state-space exploration. In *ICST*, 2012.

[11] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, 2011.

[12] JDK. LOGGER-4779253. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4779253.

[13] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 1978.

[14] M. Musuvathi and S. Qadeer. Chess: systematic stress testing of concurrent software. In *LOPSTR*, 2006.

[15] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[16] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, 2007.

[17] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, 2008.

[18] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, 2012.

[19] T. F. Serbanuta, F. Chen, and G. Rosu. Maximal causal models for sequentially consistent systems. In *RV*, 2012.

[20] University of Nebraska Lincoln. Software-artifact Infrastructure Repository. http://sir.unl.edu.

[21] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Springer ASE*, 2003.

[22] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *ICSE*, 2011.

[23] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *ICSE*, 2011.

[24] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *OOPSLA*, 2012.