Iterative Cycle Detection via Delaying Explorers MSR-TR-2015-28

Ankush Desai
UC Berkeley

Shaz Qadeer
Microsoft

Sriram Rajamani
Microsoft

Sanjit Seshia
UC Berkeley

Abstract

Liveness specifications on finite-state concurrent programs are checked using algorithms to detect reachable cycles in the state-transition graph of the program. We present new algorithms for cycle detection based on the idea of prioritized search via a delaying explorer. We present thorough evaluation of our algorithms on a variety of reactive asynchronous programs, including device drivers, distributed protocols, and other benchmarks culled from the research literature.

1 Introduction

Asynchronous software systems are ubiquitous; examples include cyber-physical systems, device drivers and other operating system extensions, workflows in web services, protocols in distributed systems, and client-server applications running natively or in the browser. Asynchronous programming is difficult due to the classical challenges of concurrency and reactivity. Concurrency introduces the problem of nondeterministic scheduling of simultaneously-executing activities; reactivity introduces the problem of frequent nondeterministic interaction with the environment. Both phenomena subvert the normal control flow of sequential computation and cause significant difficulty in design, implementation, testing, and debugging of asynchronous programs.

This paper presents the design and implementation of a framework for systematic testing of asynchronous programs. Systematic testing, popularized by the Chess [22] system, is a method of testing concurrent programs in which all sources of nondeterminism in a program are brought under the control of a *deterministic scheduler*. The deterministic scheduler gives the ability to reproduce executions; the executions of the program can then be enumerated and executed using this deterministic scheduler. It has been difficult to apply systematic testing to "live" concurrent programs because of the significant engineering challenge of controlling all sources of nondeterminism. However, emerging domain-specific languages for asynchronous programming, such as P [9], provide formal operational semantics and linguistic capability to compactly model a nondeterministic environment. A language with such features enable the development of simulators that are capable of faithfully executing the nondeterministic operational semantics of a program; therefore, we believe a systematic testing framework is ideally suited and tremendously useful for P and other languages like it.

Our testing framework is based on a class of deterministic schedulers called *delaying schedulers*. A delaying scheduler augments a deterministic scheduler with a *delay* operation; the delay operation enables the deterministic scheduler to enumerate the scheduling choices in the program in a priority order based on domain-specific heuristics. A prioritized search can be performed by writing a custom delaying scheduler and then performing iterative deepening [24] with respect to the number of delay operations. We demonstrate that combining iterative deepening with delaying schedulers is highly effective in finding errors quickly.

The primary application of our test framework is systematic testing of reactive asynchronous programs. It is well-known in the model checking literature [23] that the state space of a typical reactive program has the property that many paths lead to the same state. Consequently, the ability to cache visited states provides an important optimization to avoid redundant execution of interleavings; we would like to take advantage of this optimization. However, incorporating state caching in our testing framework is nontrivial because of the presence of the external scheduler. Since a delaying scheduler can be an arbitrary program with a huge state space of its own, the naive strategy of composing the program under test with the scheduler explodes the state space and negates the benefits of state caching. We need to devise algorithms that cache only the system state without losing coverage. To find errors fast, our testing strategy is based on iterative deepening with respect to the number of delay operations. We also need algorithms that reuse work from previous iterations as we deepen the search frontier. We have designed model checking algorithms, for both safety and liveness properties, that satisfy our aforementioned goals. We believe that our algorithm for checking liveness specifications is the first algorithm that works with iterative deepening.

Our systematic testing framework is being used extensively for analysis of realistic asynchronous systems, from device drivers to distributed protocols to workflows in web services. The conduit for all these applications is the P domain-specific language for asynchronous event-driven systems. Our experience with these applications has guided the design of the novel abstraction of delaying schedulers.

We summarize the contributions of this paper as follows:

1. We present algorithms for incorporating model checking techniques such as state caching and fair cycle detection into our testing framework. By caching states, we avoid testing redundant interleavings and enabled efficient checking of liveness specifications.

2. We demonstrate the usefulness of our framework in solving a variety of testing problems and present experimental evaluation on a number of realistic asynchronous programs.

2 Delaying schedulers

In this section, we formalize the abstractions used by our test framework for modeling concurrent programs and schedulers. The concepts defined in this section will be used to illustrate the capabilities of our system in Section 3 and to describe the algorithms in Section 4.

We model concurrent programs as transition systems. Let Pid be an uninterpreted set of process identifiers. A program $\mathcal{P}$ is a tuple $(S, T, Procs, s_0)$ such that

1. S is the set of states of $\mathcal{P}$.

2. $T \subset Pid \times S \to S$ is the transition function of $\mathcal{P}$. Given $pid \in Pid$ and $s \in S$, $T(pid, s)$ is the next state of the system obtained by executing the process pid from state s. If $T(pid, s) = s'$, we say that (s, s') is a transition of $\mathcal{P}$. By formalizing T as a function rather than a relation, we have explicitly chosen not to model internal nondeterminism inside each process; we note that the implementation of our testing framework does support internal nondeterminism though.

3. $Procs \in S \to Set\langle Pid\rangle$ is the finite set of alive processes in each state of $\mathcal{P}$. For any $s \in S$, we use $\#Procs(s)$ to denote the cardinality of $Procs(s)$. Thus, $\#Procs \in S \to \mathbb{N}$.

4. s_0 is the initial state of $\mathcal{P}$.

A sequence of states $s_0, s_1, s_2, \ldots, s_n$ is an execution of $\mathcal{P}$ if there is a sequence of process identifiers $pid_0, pid_1, \ldots, pid_{n-1}$ such that $T(pid_i, s_i) = s_{i+1}$ for all $i \in [0, n)$. A pair (s, s') is a *reachable transition* of $\mathcal{P}$ if there is an execution $s_0, s_1, s_2, \ldots, s_n$ such that $s = s_{n-1}$, and $s' = s_n$. A pair (s, s') is a *repeatedly reachable transition* of $\mathcal{P}$ if there is an execution $s_0, s_1, \ldots, s_n$ such that $s = s_{n-1}$, $s' = s_n$, and $s_n = s_i$ for some $i \in [0, n)$.

We also model deterministic schedulers as transition systems, with their own internal state. A deterministic scheduler $\mathcal{D}$ is a tuple $(D, Next, Step, d_0)$ such that

1. D is the set of states of $\mathcal{D}$. The state of the scheduler typically includes a data structure (such as

a stack) which maintains an ordering among the process ids of the program.

2. $\mathit{Next} \in D \rightarrow \mathit{Pid}$ is a total function. Given a program state s and a scheduler state d, $\mathit{Next}(d)$ is the id of the process that is prescribed by the scheduler to be executed next.

3. $\mathit{Step} \in S \times D \times S \rightarrow D$ is a total function. Suppose we have a program state s and a scheduler state d, and we execute the process with id $\mathit{Next}(d)$ at s, and obtain the resulting program state s'. Then $\mathit{Step}(s, d, s')$ yields the scheduler state corresponding to the program state s'. The *Step* function enables building schedulers which change their state in response to specific events that occur during execution of the program, such as sending or receiving of messages, changes in states of queues or other data structures etc.

4. d_0 is the initial state of $\mathcal{D}$.

We denote by $\mathcal{P} \| \mathcal{D}$ the composition of program $\mathcal{P}$ and deterministic scheduler $\mathcal{D}$. The sequence $(s_0, d_0), (s_1, d_1), (s_2, d_2), \ldots$ is the *unique* execution of of $\mathcal{P} \| \mathcal{D}$ if $T(\mathit{Next}(d_i), s_i) = s_{i+1}$ and $\mathit{Step}(s_i, d_i, s_{i+1}) = d_{i+1}$ for all $i \geq 0$.

A deterministic scheduler, although useful for reproducible execution, does not enable by itself thorough testing of an asynchronous program. To enable systematic testing, we leverage the notion of delaying scheduler [12], a deterministic scheduler with a *delay* operation. We model this operation as a total function $\mathit{Delay} \in D \rightarrow D$. Given a scheduler state d, the application $\mathit{Delay}(d)$ yields a new scheduler state such that $\mathit{Next}(\mathit{Delay}(d))$ indicates the "next" process that can be scheduled at state s. Given a program $\mathcal{P}$ and a delaying scheduler $\mathcal{D}$, a sequence $(s_0, d_0), (s_1, d_1), (s_2, d_2), \ldots$ is an execution of $\mathcal{P} \| \mathcal{D}$ if for all $i \geq 0$, either $s_i = s_{i+1}$ and $\mathit{Delay}(d_i) = d_{i+1}$ or $T(\mathit{Next}(d_i), s_i) = s_{i+1}$ and $\mathit{Step}(s_i, d_i, s_{i+1}) = d_{i+1}$. A pair (s, d) is a reachable state of $\mathcal{P} \| \mathcal{D}$ if it occurs on an execution.

Delay-bounded search with parameter n enumerates those executions in which the number of delay operations is bounded by n. For executions of length I, there can be at most I^n executions with no more than n delays. Thus, for small values of n, it is feasible to enumerate all executions even for large values of I. Our results in Section 5 show that if a bug exists, it is possible to write a domain-specific delaying scheduler for which the bug will manifest for small values of the delay bound. This property makes delaying schedulers useful for systematic testing.

In order to ensure that all scheduling choices are covered, the delaying scheduler must ensure that all processes are indeed generated by successive applications of *Delay*. To formalize this requirement, we define $\mathit{Next}^k(d)$ inductively as follows:

$$\begin{array}{rcl} \mathit{Next}^0(d) & = & \{\mathit{Next}(d)\} \\ \mathit{Next}^{k+1}(d) & = & \mathit{Next}^k(d) \cup \{\mathit{Next}(\mathit{Delay}^{k+1}(d))\} \end{array}$$

where $\mathit{Delay}^k(d)$ is k iterative applications of *Delay* to d. A delaying scheduler must satisfy the following property for every reachable state (s, d) of $\mathcal{P} \| \mathcal{D}$:

$$\mathit{Procs}(s) = \mathit{Next}^{\#\mathit{Procs}(s)}(d).$$

3 Exploiting schedulers for systematic testing

In this section, we give an overview of how our systematic testing framework exploits the abstraction of sealable delaying schedulers defined in the last section. We begin by informally describing three delaying schedulers we have implemented. Later, we describe how we add sealing to these schedulers to encode scheduling assumptions.

Random scheduler. The random delaying scheduler serves as a baseline for comparison when evaluating other schedulers with more sophisticated strategies. This scheduler picks a random order of the processes for each new visited state and explores transitions in that order. The delay operation simply skips over the next process to be scheduled. The random delaying scheduler is particularly useful in those scenarios where the user has no strong intuition about the schedules that can find bugs faster.

Round-robin scheduler. The round-robin delaying scheduler cycles through the tasks in task creation order. It schedules the next task on a delay or when the current task is completed. It can be used for finding bugs that manifest through a small number of interleavings between processes, and when the bugs manifest regardless of the order in which the processes are interleaved.

Run-to-completion scheduler. The run-to-completion delaying scheduler explores schedules that follow causal sequence of events. When a delay is applied, the schedule departs from the causal order. Even for small values of delay bound, this scheduler is able to explore long paths in the state space since it just follows event generation order as new events get generated. In our experience, this scheduler is able to find bugs which manifest only after certain events have occurred, even at low delay bound values.

Each of these schedulers can be used to systematically test programs by iteratively increasing the number of allowed delay operations along any execution. We have used these schedulers to test many asynchronous programs. For a given buggy program, different schedulers can be compared by measuring the number of states generated before reaching the error. Our results, presented in Table 2 and described in detail in Section 5, show a significant variation in the winning scdeduler across different programs. This result indicates the benefit of a general framework that enables testers to write a variety of delaying schedulers.

4 Incorporating model checking techniques

In this section, we show how to incorporate state caching techniques into our systematic testing framework. The introduction of state caching solves two different problems. First, it reduce potentially redundant enumeration of executions. Second, it enables efficient checking of liveness specifications. Liveness checking, although possible without state caching [21], requires heuristic detection of nonterminating executions which becomes very expensive in the presence of cycles.

There are two main challenges our algorithms attempt to solve. First, our test framework executes the composite system comprising the program and the sealable delaying scheduler being used for testing it. Since a sealable delaying scheduler can be an arbitrary program with a huge state space of its own, the naive strategy of composing the program under test with the scheduler explodes the state space and negates the benefits of state caching. Our algorithms must avoid caching the scheduler state without losing coverage. Second, our test framework is based on iterative deepening with respect to the number of delay operations. Our algorithms must handle the requirement of iterative deepening efficiently.

4.1 Safety

We model a safety property as a relation $E \subseteq S \times S$, that marks some of the state transitions of the program as error transitions. We are interested in checking if there is a reachable transition (s, s') such that $E(s, s')$. Our algorithm for solving this problem is given in Figure 1. The algorithm uses three global variables: (1) an integer *bound* (which is iteratively incremented); (2) a dictionary H to store (hashes of) visited states; and (3) a dictionary *Frontier* to store for each state s in the frontier (from which any further exploration exceeds the current delay bound) the scheduler state (d) with which s was discovered, the number of delays expended to enable the next transition (n), and the index of the next transition to be executed at the state (i).

Control starts at the procedure *IterativeSearch*, which increments the value of *bound* by δ during each iteration of the while loop, and invokes *BoundedDFS* for each state from the *Frontier* with the current values of delay and process id. We use the names s, s_o, and s' to denote program states and the names d and d_o to denote scheduler states.

BoundedDFS is invoked with a program state s, scheduler state d, current delay value n (which is the sum of the delays in the path that reaches (s, d) from the initial state), and i is the index of the next transition from s to be explored. *BoundedDFS* iterates through the transitions from s starting from index i, by applying the *Delay* operation in the outer while loop. For each application of the *Delay*, the delay value n and the count i are incremented. If the de-

```
var $bound : \mathbb{Z}$;
var $H : Dictionary\langle S \rangle$;
var $Frontier : Dictionary\langle S, (D \times \mathbb{Z} \times \mathbb{Z})\rangle$;

$BoundedDFS(s : S, d : D, n : \mathbb{Z}, i : \mathbb{Z})$ {
    var $s' : S$;
    while $(i < \#Procs(s))$ {
        if $(n > bound)$ {
            $Frontier(s) := (d, n, i)$; break;
        }
        $s' := T(Next(d), s)$;
        if $(E(s, s'))$ exit$(Yes)$;
        if $(s' \in H')$ continue;
        $H.Add(s')$;
        $BoundedDFS(s', Step(s, d, s'), n, 0)$;
        $d := Delay(s, d)$;
        $n := n + 1$; $i := i + 1$;
    }
}

$IterativeSearch()$ {
    var $Frontier' : Dictionary\langle S, (D \times \mathbb{Z} \times \mathbb{Z})\rangle$;
    $bound := \delta$;
    $H.Add(s_0)$;
    $Frontier(s_0) := (d_0, 0, 0)$;
    while $(Frontier \neq \emptyset)$ {
        $Frontier' := Frontier$; $Frontier := \emptyset$;
        foreach $((s, d, n, i) \in Frontier')$
            $BoundedDFS(s, d, n, i)$;
        $bound := bound + \delta$;
    }
    exit$(No)$;
}
```

Figure 1: Iterative safety checking

lay value n exceeds the *bound* value, then the current state (d, n, i) is stored in the frontier with s as the key. If the delay value $n \leq bound$, then we explore the successor s' of s which is obtained by executing the process with id $Next(d)$. If $E(s, s')$ holds, then we have found a violation of the safety property. Otherwise, we check if s' is already in the dictionary H; if not, we add it and explore the successors of s' by calling *BoundedDFS* recursively.

Since we do not store scheduler state in H, we greatly reduce the total number of visited states of the composite system (which consists of program state and scheduler state) *without* missing any reachable state of the system. Our empirical results (Table 1 in Section 5) quantify the efficiency improvement we obtain using this optimization.

To state the correctness of our algorithm formally, we need a couple of definitions. Let $\mathcal{P}$ be a program. An execution (finite or infinite) of $\mathcal{P}$ is *non-repeating* if all states in it are distinct. The program $\mathcal{P}$ is *reactive* if it has no non-repeating infinite executions. Note that it is not necessary for a reactive program to have a finite number of reachable states.

Theorem 1 *Consider a reactive program $\mathcal{P}$ and a delaying scheduler $\mathcal{D}$ satisfying the assumption stated at the end of Section 2. Let E be a safety property. The algorithm in Figure 1 returns Yes iff there is a reachable transition (s, s') of $\mathcal{P}$ such that $E(s, s')$. Furthermore, the algorithm terminates if $\mathcal{P}$ has finite number of reachable states.*

Proof sketch: There are three parts to the proof. First, we argue that every execution of $\mathcal{P}$ is possible in the composed system $\mathcal{P}\|\mathcal{D}$; this argument requires the assumption at the end of Section 2. Second, we have to argue that every iteration of the bounded search terminates. This argument is subtle if the reactive program $\mathcal{P}$ is not finite-state and is as follows. Because of state caching, the states in the reachable sub-graphs generated in each iteration are all distinct from each other. For a reactive program, we can show by contradiction that each sub-graph is finite. Suppose the reachable sub-graph is infinite. Since the delay bound used for generating this sub-graph is finite, this sub-graph must be finitely branching. Therefore, by Konig's lemma, there must be an infinite path in

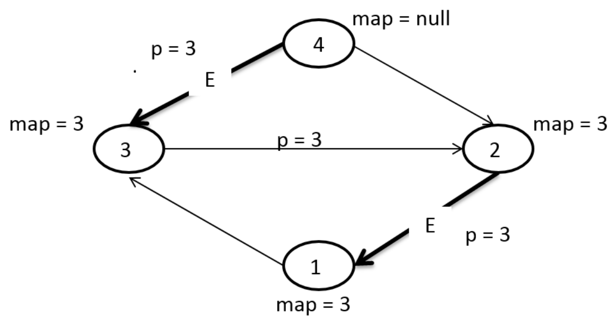

Figure 2: Calculate MAP

this graph. Since the program is reactive, some state must repeat along this path which creates a contradiction. Finally, since the number of choices to be explored at each state is bounded by the finite number of processes, we are guaranteed that every choice would eventually be taken as we keep increasing the bound. The above argument also shows that if the number of reachable states is finite, then the algorithm will terminate.

4.2 Liveness

Checking that a liveness property is satisfied by the program reduces to detecting a reachable cycle in the composition of the property automaton with the transition system of the program. In this paper, we focus only on this final algorithmic problem of detecting a reachable cycle, ignoring the details of how the property is actually specified by the user. Similar to safety, we model a liveness property as a relation $E \subseteq S \times S$ marking those state transitions of the program whose repeated visitation indicates a liveness error. We would like to check if there is a repeatedly reachable transition (s, s') such that $E(s, s')$. In the context of liveness checking, we denote a transition $(pid, s, s') \in T$ such that $E(s, s')$ as an *accepting transition* and a cycle containing an accepting transition is called an *accepting cycle*.

The most commonly used algorithm for detecting accepting cycles is the nested depth-first search (NDFS) algorithm [16]. NDFS works by computing reachable states using a standard depth-first search

```
var $bound : \mathbb{Z}$;
var $H : Dictionary\langle S, (D \times \mathbb{Z})\rangle$;
var $Frontier : Set\langle S\rangle$;
var $map : Dictionary\langle S, S\rangle$;

$IterativeCycleDetection()$ {
    var $Frontier' : Set\langle S\rangle$;
    var $map' : Dictionary\langle S, S\rangle$;
    $bound := 0$; $map(s_0) := null$;
    $H(s_0) := (d_0, 0)$; $Frontier := \{s_0\}$;
    while ($Frontier \neq \emptyset$) {
        $Frontier' := Frontier$; $Frontier := \emptyset$;
        foreach ($s \in Frontier'$)
            $CalculateMap(s)$;
        $map' := map$;
        while ($true$) {
            var $roots : Set\langle S\rangle$;
            var $root : S$;
            $roots := Range(map) \setminus \{null\}$;
            if ($roots = \emptyset$) break;
            choose ($root \in roots$);
            $map(root) := null$;
            $RefineMap(root, root, roots \setminus \{root\})$;
        }
        $map := map'$;
        $bound := bound + \delta$;
    }
    exit($No$);
}
```

Figure 3: Iterative cycle detection

```
$CalculateMap(s : S)$ {
    var $d : D$;
    var $n, i : \mathbb{Z}$;
    var $p, s' : S$;
    $(d, n) := H(s)$; $i := 0$;
    while $(i < \#Procs(s))$ {
        if $(n > bound)$ {
            $Frontier.Add(s)$; break;
        }
        $s' := T(Next(d), s)$;
        if $(E(s, s') \wedge map(s) = s')$ exit$(Yes)$;
        if $(s' \notin Domain(H))$
            $H(s') := (Step(s, d, s'), n)$;
        if $(E(s, s') \wedge map(s) < s')$
            $p := s'$;
        else
            $p := map(s)$;
        if $(s' \notin Domain(map) \vee map(s') < p)$ {
            $map(s') := p$;
            $CalculateMap(s')$;
        }
        $d := Delay(s, d)$;
        $n := n + 1$; $i := i + 1$;
    }
}
```

Figure 4: Calculate maximum accepting predecessor

```
$RefineMap(s : S, root : S, roots : Set\langle S \rangle)$ {
    var $d : D$;
    var $n, i : \mathbb{Z}$;
    var $p, s' : S$;
    $(d, n) := H(s)$; $i := 0$;
    while $(i < \#Procs(s) \wedge n \leq bound)$ {
        $s' := T(Next(d), s)$;
        if $(map(s') \notin roots)$ {
            if $(E(s, s') \wedge map(s) = s')$ exit$(Yes)$;
            if $(E(s, s') \wedge map(s) < s')$
                $p := s'$;
            else
                $p := map(s)$;
            if $(map(s') = root \vee map(s') < p)$ {
                $map(s') := p$;
                $RefineMap(s', root, roots)$;
            }
        }
        $d := Delay(s, d)$;
        $n := n + 1$; $i := i + 1$;
    }
}
```

Figure 5: Refine maximum accepting predecessor

and then invokes a nested depth-first search whenever an accepting transition is popped from the stack; this nested search looks for a path to target state of the accepting transition just popped. These two searches can be combined into a single stack and a state table using the "magic"-bit trick [16].

We have used a straightforward extension of the safety algorithm in Section 4.1 to implement an iterative version of NDFS. The correctness of the NDFS algorithm depends fundamentally on the constraint that the nested search must be invoked in the order states are popped from the stack. Therefore, our implementation of the iterative NDFS algorithm requires the sledgehammer of restarting the search from the initial state with an empty state table. While this approach is correct, it is inefficient since it causes each partial search to be done repeatedly (we present empirical results in Section 5).

Figures 3, 4, and 5 together provide pseudo-code describing a new algorithm that exploits a method of finding accepting cycles using maximal accepting predecessors [4]. The advantage of this method is that it does not require search to be performed in any fixed order. We exploit this feature to create an iterative version of the algorithm. To the best of our knowledge, this is the first model checking algorithm for liveness, which allows incremental exploration of the state space while effectively reusing work from previous iterations.

4.2.1 Maximum accepting predecessor algorithm

In this section, we review the method for detecting accepting cycles using maximal accepting predecessors. Let us assume that there is a total order $<$ on the set of states in the program. Let *null* be a distinguished state that is different from all reachable states and is the least according to $<$. A state s is an *accepting predecessor* of a state s' if for some $n > 0$, there are process identifiers $pid_0, \ldots, pid_{n-1}$ and states $s_1, \ldots, s_n$ such that $T(pid_i, s_i, s_{i+1})$ holds for all $i \in [0, n)$, $E(s_{i-1}, s_i)$ and $s = s_i$ for some $i \in (0, n]$, and $s' = s_n$. It is possible for a state to not have any accepting predecessors. We define the *maximum accepting predecessor* (MAP) of a state to be *null* if it does not have any accepting predecessors and otherwise the unique state that is maximum among its accepting predecessors.

The MAP of all reachable states can be computed by a simple adaptation of the standard reachability algorithm. This algorithm is captured by the *CalculateMap* procedure in Figure 4. Our algorithm uses four global variables—*bound*, H, *Frontier*, and *map*; only the *map* variable is important for MAP calculation, the other becoming relevant only in the context of iterative deepening. In Figure 4, let us ignore lines 7-9; we will cover them later in Section 4.2.2 when we discuss our iterative cycle detection method. The table *map* contains a mapping from each visited state to the current underapproximation of its MAP. If a state is not currently present in the table, we can think of *null* as its approximate MAP. This approximation is propagated from a state along all non-accepting outgoing transitions. On an outgoing accepting transition, if the target state is greater than the current MAP of the source state, then the target state is propagated instead. This propagation happens in lines 14–17 with the propagated value stored in the local variable p. Finally, in lines 18–21, the propagated value is used to revise the current MAP estimate of the target and schedule it for re-exploration if its estimate increased.

Clearly, this algorithm maintains the invariant that if $map(s) \neq null$, then there is a path from $map(s)$ to s. Therefore, during the above calculation of the maximum accepting predecessor, if it ever happens that an accepting transition from s to s' is explored and $map(s) = s'$, then the algorithm has discovered a reachable cycle containing an accepting transition and the execution terminates (line 11). However, it is possible that the MAP calculation finishes without discovering an accepting cycle. When this happens, it does not mean that such a cycle does not exist. For example, consider the state-transition graph in Figure 2. The number labeling the state represents its location in the total order $<$; the initial state is numbered 4; the accepting transitions are labeled E; the propagated values along a transition are denoted by p. In this graph the cycle comprising the nodes 3, 2, and 1 is accepting and reachable from the initial state 4; however, the MAP calculate phase will terminate without reporting a cycle.

The definition of accepting predecessors implies that all states in a strongly-connected component must have the same MAP value. Thus, the MAP calculation creates a partition of the set of strongly-connected components; we call the MAP value of the partition its root. The graph in Figure 2 contains two partitions $\{4\}$ and $\{1, 2, 3\}$, each containing a single strongly-connected component; the roots of these partitions are *null* and 3, respectively. The strongly-connected component $\{1, 2, 3\}$ has an accepting cycle. But the calculate MAP phase was unable to find this cycle because this component is reachable from the accepting transition $(4, 3)$ which ends up giving a MAP value of 3 to all states in the component. Since $map(2) \neq 1$ when the accepting edge from 2 to 1 is explored, the cycle is not discovered. If the MAP values were to be re-calculated from scratch, starting from the root of the partition and confining the search to the partition, the cycle would be discovered because the accepting edge $(4, 3)$ is outside the partition. This observation suggests a refinement of the algorithm to find an accepting cycle by iteratively recalculating the MAP values for a partition. Each such calculation either finds an accepting cycle or decomposes the partition into smaller sub-partitions. If no further sub-partitions are created, then there is no accepting cycle in the graph. The loop in Figure 3 from lines 11–19 does exactly this iterative refinement of MAP values. Instead of calling *CalculateMap*, the loop calls *RefineMap*; the differences between *CalculateMap* and *RefineMap* are relevant only to account for iterative deepening.

4.2.2 Iterative maximum accepting predecessor algorithm

In this section, we describe how to build iterative deepening into the algorithm described in Section 4.2.1. At the top-level, just like the iterative safety algorithm described in Section 4.1, iterative cycle detection also creates a increasing sequence of sub-graphs of the state-transition graph of the program. The bookkeeping for creating this increase sequence is based on global variables *bound*, H, and *Frontier*, similar to the safety case. In each iteration, instead of a reachability algorithm, we run the algorithm described in Section 4.2.1 to detect accepting cycles assuming the current sub-graph is the entire graph. If a cycle is detected, the algorithm terminates with *Yes*, otherwise the sub-graph expansion continues from the states in *Frontier*. This iterative process is captured in the pseudo-code of Figure 3.

There are important differences in the bookkeeping of the liveness algorithm compared to the safety case though. The main reason for these differences is that while the safety algorithm is purely reachability, the liveness algorithm is doing MAP calculation which may require exploring a state several times in order to propagate increasing MAP values. Since the sub-graph reachable from a state in a particular iteration depends on the scheduler state and delay value at the time of the first visit, both these values must be stored for all visited states in the H dictionary. The reader might recall that in the safety case, it sufficed to store these values only for the states in the frontier. Furthermore, since each state must be re-explored completely, the transition index i is not required to be stored in either H or *Frontier*; the exploration of outgoing transitions from a state always starts at index 0.

The following theorem characterizes the soundness of our algorithm.

Theorem 2 *Consider a reactive program $\mathcal{P}$ and a delaying scheduler $\mathcal{D}$ satisfying the assumption stated at the end of Section 2. Let E be a liveness property. The algorithm in Figure 1 returns Yes iff there is a repeatedly reachable transition (s, s') of $\mathcal{P}$ such that $E(s, s')$. Furthermore, the algorithm terminates if $\mathcal{P}$ has finite number of reachable states.*

The proof for the liveness algorithm is very similar to the proof sketch for the safety algorithm presented in Section 4.1.

4.3 Related work

Directed model checking [10] algorithms perform *local* prioritizion of state transitions during exploration; each transition is given a cost and transitions out of a visited state are taken in increasing order of the cost. A delaying scheduler also provides mechanism to order transitions coming out of a state; hence it can be used in combination with directed model checking. However, the main motivation of using delaying schedulers in our test framework is to be able to perform *global* prioritization over the set of all states; this prioritization is defined by the delay bound required to reach a state and the goal of iterative deepening is to explore states reachable with a smaller bound before states reachable with a larger bound.

Algorithms for detecting accepting cycles can be broadly classified into two families: those based on nested depth-first search (NDFS) [7] and those based on strongly connected components (SCC) [8]. In general, explicit-state model checkers have implemented NDFS-based algorithms, and symbolic model checkers have implemented SCC-based algorithms. To the best of our knowledge, we are the first to propose a cycle detection algorithm that works with iterative deepening.

	Tree Identification			Lann Protocol			ORSet		
p	Store (s)	Store (s,d)	overhead	Store (s)	Store (s,d)	overhead	Store (s)	Store (s,d)	overhead
2	2.50E+02	7.45E+02	2.98	1.12E+03	1.89E+03	1.68	2.31E+04	5.67E+04	2.46
3	3.35E+03	1.15E+04	3.45	2.45E+05	6.04E+05	2.47	5.57E+05	1.23E+06	2.21
4	7.19E+04	3.01E+05	4.19	5.57E+06	2.08E+07	3.74	4.45E+07	1.58E+08	3.55
5	4.96E+05	2.57E+06	5.18	9.05E+07	5.02E+08	5.55	2.60E+09	*	*

Table 1: Overhead of storing composite state (s,d) as compared to only program state (s)

5 Evaluation

In this section, we present an empirical evaluation of our sealable delaying scheduler framework and the model checking algorithms. All the experiments we report were performed on an Intel Xeon E5620, 2.39GHz (8 cores) with 16 GB of memory running 64-bit Windows server OS.

In order to study the relative efficacy of different delaying schedulers, we designed a benchmark suite consisting of various kinds of models. We used the protocol programming language P [9] to write all our benchmarks, and used the P compiler to generate Zing models for our experiments. Our benchmarks include: LCR [20] algorithm for leader election in synchronous ring; SyncBFS [20] algorithm for breadth-first search in synchronous network; CTS, a clocked transition system; TSP, a distributed time-synchronization protocol; OSR and USB, two Windows device drivers; ORSet and CSscale, two distributed state-replication protocols that guarantee eventual consistency; Lann, a leader-election protocol; a tree-identification protocol; elevator door, elevator planning, and truck lift controllers; German and MSMIE cache-coherence protocols. Our benchmarks have significant complexity and their sizes range from 250 lines (for Lann) to 2200 lines (for TSP) of P code. We remind the reader that P is a domain-specific language for writing asychronous protocols; the same functionality if implemented directly in C would be much larger. For lack of space, we are eliding the references to these benchmarks; we can provide details in the final version.

5.1 Importance of keeping scheduler state separate

Table 1 shows that the overhead of storing the state of the delaying scheduler alongside the state of the program is significant. This evaluation was performed using the run-to-completion delaying scheduler on three distributed protocols, each of them parameterized by the number of participating processes. Each row in the table presents the results for a different number of processes. The overhead of storing composite state appears to to be proportional to the number of processes p in the system, since the scheduler can be in p different states for a given program state. A nice property of delay-bounded scheduling is that its complexity is independent of the number of processes in the systems depending purely on number of scheduler invocations [12]. By developing algorithms that do not require storing the scheduler state together with the program state, our safety checker continues to enjoy this property.

5.2 Importance of a scheduling framework

Table 2 show our results evaluating three delaying schedulers. Each row in the table corresponds to a benchmark program. All benchmarks have bugs, and we stopped the exploration when the search finds the bug and recorded the number of states explored before the checker discovered the bug. The execution time is proportional to the number of states explored before discovering the bug; we omit reporting the actual time. There are two top-level columns labeled *Iterative Depth bounding* and *Iterative Delay Bounding*; the latter has sub-columns underneath for different delaying schedulers; we used three schedulers —*random*, *run-to-completion*, and *round-*

Models	Iterative Depth Bounding		Iterative Delay Bounding					
	Random DFS		Random Scheduler		Run-to-completion Scheduler		Round-robin Scheduler	
	Depth	States Explored	Delay	States Explored	Delay	States Explored	Delay	States Explored
Asynchronous Windows Device Drivers								
OSR	35	806585	8	15896	**6**	**7885**	12	71957
USB	40	1616157	2	1487	**2**	**687**	2	3380
Distributed State-Replication Protocol								
ORSet	*	*	3	8578	4	179930	**2**	**2231**
CScale	*	*	5	45021	6	894355	**4**	**15634**
Distributed Leader Election Protocol								
Lann Protocol	*	*	15	66785	**2**	**9075**	8	3680401
Tree Identification	*	*	26	259115	**6**	**27804**	*	*
Real-World Systems								
Elevator Door	25	13808	**4**	**1771**	4	2277	6	13848
Elevator Planning	35	18609583	**9**	**26532**	5	32907	16	24349202
Distributed Truck Lift	30	950005	**8**	**1656**	3	64249	4	13180
Coherence Protocol								
German	25	595723	**6**	**3011**	2	3613	*	*
MSMIE	35	2230542	8	44123	**6**	**5156**	9	223411
Time Synchronization Protocol								
TSP (bug 1)	*	*	**11**	**9.4E+04**	19	1.6E+07	*	*
TSP (bug 2)	*	*	22	2.9E+07	**16**	**5.1E+05**	31	3.1E+09

$* \rightarrow$ *Bug could not be found, search ran out of memory and explored more than* 10^{10} *states*

Table 2: Comparison between iterative depth bounding and iterative delay bounding using different delaying schedulers

robin— whose descriptions appear in Section 3. This table demonstrates two main points. First, iterative delay bounding works dramatically better than iterative depth bounding, which provides compelling justification of the use of delaying schedulers in our test framework. Second, there is *no* single delaying scheduler that performs best for all the models. The numbers for the winning scheduler in each row are in **bold**; a quick scan of the rows clearly indicates that bold entries appear in many different columns. Therefore, it is crucial to have a test framework that allows the use of different delaying schedulers depending on the domain.

Benchmarks such as OSR device driver, Windows USB driver, German and MSMIE protocol have bugs deep in the search space. That is, the bugs manifest only after a certain number of events/messages are exchanged among the processes. For such models, run-to-completion scheduler performs best, exploring the least number of states before finding the bug.

Benchmarks such as ORSet and CScale have bugs that occur when certain operations in the model interleave. For example, in the case of ORSet, the bug occurs when Add and Read Operations on the set are interleaved. The results show that in such cases a simple scheduler like round-robin scheduler performs best. Hence, for models that have bugs manifesting when some processes interleave, a simple scheduler like round-robin may work best.

We uncovered subtle errors in modeling assumption in the time synchronization protocol (TSP). Iterative depth bounding failed to find these bugs, also round-robin delaying scheduler failed to find the first bug. We found that the error occurs when a particular message is delayed for too long after reaching stable state in the protocol; this is a deep bug and run-to-completion scheduler performed well. Since we are doing prioritized search with iterative deepening on delays, performing the exploration till a sufficiently small bound gives enough confidence about the correctness of the protocol as compared to naive uniform search.

For models such as elevator door, elevator planning, and distributed lift, it is interesting to note that the random delaying scheduler performed best. By studying the kind of bugs and buggy schedules occurring in these models one may be able to design a new scheduler which gives higher priority to schedules that are similar to the buggy schedules. One can then use this scheduler for models that have similar bugs.

5.3 Importance of iterative deepening in liveness checking

All the benchmarks in Table 3 have liveness bugs, and we stopped the exploration when the search finds the bug and recorded the number of states explored and time taken by the checker to discover the bug. We implemented three algorithms for evaluation of our liveness checking algorithm —NDFS, iterative NDFS and iterative MAP. The results for both iterative NDFS and iterative MAP were obtained using the random delaying scheduler. As explained in Section 4.2, iterative NDFS runs NDFS with a delay bound; whenever the bound is incremented, the search restarts from the initial state.

We first compared NDFS with iterative NDFS to evaluate the efficacy of iterative deepening (with delaying schedulers) for detecting liveness errors, a key contribution of this paper. Table 3 reveals that iterative NDFS explores many fewer states before finding the cycle as compared to NDFS. This happens because instead of going deep, iterative NDFS tries to find cycle within the iterative bound, thereby finding cycles in the search space as soon as they are reachable. For all the models in Table 3, iterative NDFS was able to detect cycle faster than NDFS. For models like elevator planning and tree identification protocol, NDFS failed to find the cycle and ran out of memory whereas iterative NDFS found the cycle.

Next we compared the efficacy of iterative MAP in avoiding re-exploration compared to iterative NDFS. Table 3 shows that iterative MAP explores almost the same number of states as iterative NDFS before detecting a cycle. This happens because both algorithms have to explore all the states reachable till the iterative bound after which at least one cycle is reachable. The small difference is due to the search they perform in the iteration where the cycle is reachable. However, iterative MAP algorithm finds the cycle up to three times times faster than iterative NDFS because it avoids the overhead of complete re-exploration after each iteration.

It was easy to parallelize MAP algorithm using the existing framework for parallel search in Zing. Table 3 also presents results for parallel exploration of iterative MAP algorithm. We observed good scaling up to four threads and obtained improvements up to three times in time taken.

Finally, we also compared the time taken in the calculation of maximal accepting predecessors (*CalculateMap*) versus the time taken in refining this calculation (*RefineMap*). For all problems, the time taken for the two phases is roughly the same. This indicates to us that there are opportunities for optimizing both the phases of the algorithm. One possible optimization is to discard partitions without cycles early in the `RefineMap` phase. For a partition to have cycle, it is necessary that there is at least one node with in-degree greater than 1, and at least one of the incoming edges for that node should be a back edge. We can propagate this information back to the root node of partition and use it to eliminate partitions without cycles early.

6 Related Work

Work on testing of concurrent programs can be classified into three categories: (1) model checking, (2) deterministic testing, and (3) nondeterministic testing. Model checking tools [2, 1, 28] verify properties of concurrent models and view systematic testing as state-space exploration problem; related work about optimizations like state-caching and heuristic based search was described in Section 4.3. Deterministic testing [6, 18] is similar to simulating the program based on a given input and a fixed schedule. The testing coverage is obtained by repeatedly changing the (input, schedule) pair to guide program across different execution paths. Tools like VERISOFT [15] and CHESS [22] combine ideas from model checking and deterministic testing to systematically explore executions of a program on a given input. But they fail to scale in the presence of uncontrollable nondeterminism in the environment. Our approach belongs to this class of techniques; we exploit a domain-specific

Models	Number Of States Explored			Time Taken (HH:MM:SS)				MAP Time per Phase	
	NDFS	Iterative NDFS	Iterative MAP	NDFS	Iterative NDFS	Iterative MAP	Parallel-Iterative Map (4 threads)	Calc MAP	Refine MAP
Elevator Door	188067	34562	38675	2:13:34	1:24:23	0:28:55	0:12:05	0:10:11	0:18:44
Lann Protocol	3455431	443421	361342	28:01:56	7:34:13	4:43:22	2:10:35	2:21:21	2:22:01
Elevator Planning	*	1083223	887634	*	30:33:12	9:04:21	3:55:50	3:38:12	5:26:09
Tree Identification	*	4582233	3876253	*	61:23:34	20:10:34	6:23:44	9:03:18	11:07:16

Table 3: Results for liveness checking (with random delaying scheduler)

language like P [9], using which tester can precisely model the environment nondeteminism and also constrain it via sealing or delay bounding.

Nondeterministic testing involves repeatedly running the program on input test-cases by randomly adding delays during each run. Tools like Cuzz [5], ConTest [13], and RaceFuzzer [25] use fuzzing techniques for performing nondeterministic testing of concurrent programs. These techniques maximize concurrency testing coverage by randomizing the schedules in a systematic way and insert delays using domain specific heuristics. Though these techniques scale as compared to deterministic testing, even in the presence of uncontrolled nondeterminism in the environment, they fail to provide coverage guarantees. To mitigate the exponential path-enumeration problem, tools like Contessa [19] use symbolic analysis and partial-order reduction technique to improve test coverage. Penelope [27] exploits the heuristic that most of the concurrency bugs occur because of atomicity violations and hence prioritize schedules that exhibit these patterns. Our approach is to provide generic framework to plugin any deterministic delaying scheduler for prioritizing schedules, and hence any of the above strategies can be used for improving coverage guarantees.

Another recent tool, Concurrit [11], proposes a domain specific language for writing debugging scripts that help the tester specify thread schedules for reproducing concurrency bugs. The search is guided by the script without any prioritization. In contrast, our work is focused on finding rather than reproducing bugs. Instead of a debugging script, a tester writes a domain-specific scheduler with appropriate uses of sealing; iterative deepening with delays automatically prioritizes the search with respect to the given scheduler.

References

[1] The model checker SPIN. *IEEE Trans. on Software Engineering*, 1997.

[2] T. Andrews, S. Qadeer, S. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Proceedings of CAV*. 2004.

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

[4] L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting predecessors are better than back edges in distributed LTL model-checking. In *FMCAD*. 2004.

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

[6] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *International Joural on Software, IEEE*, 1991.

[7] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *CAV*. 1991.

[8] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *Formal Methods*. 1999.

[9] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of PLDI*, 2013.

[10] S. Edelkamp, V. Schuppan, D. Bo, A. Wijs, A. Fehnker, and H. Aljazzar. Survey on directed model checking. In *Model Checking and Artificial Intelligence*. 2009.

[11] T. Elmas, J. Burnim, G. Necula, and K. Sen. CONCURRIT: A domain specific language for reproducing concurrency bugs. In *Proceedings of PLDI*, 2013.

[12] M. Emmi, S. Qadeer, and Z. Rakamarić. Delay-bounded scheduling. In *Proceedings of POPL*, 2011.

[13] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proceedings of IPDPS*, 2003.

[14] J. Fisher, T. Henzinger, M. Mateescu, and N. Piterman. Bounded asynchrony: Concurrency for modeling cell-cell interactions. In *FMSB*. 2008.

[15] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of POPL*, pages 174–186, 1997.

[16] P. Godefroid and G. J. Holzmann. On the verification of temporal properties. In *Proceedings of PSTV*, 1993.

[17] T. L. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, 2003.

[18] G.-H. Hwang, K. chung Tai, and T. lu Huang. Reachability testing: An approach to testing concurrent software. *International Journal of Software Engineering and Knowledge Engineering*, 1995.

[19] S. Kundu, M. Ganai, and C. Wang. Contessa: Concurrency testing augmented with symbolic analysis. In *Proceedings of CAV*. 2010.

[20] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[21] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *Proceedings of PLDI*, 2008.

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

[23] R. Pelánek, T. Hanžl, I. Černá, and L. Brim. Enhancing random walk state space exploration. In *Proceedings of FMICS*, 2005.

[24] S. J. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[25] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of PLDI*, 2008.

[26] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[27] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of FSE*, 2010.

[28] W. Visser and P. C. Mehlitz. Model checking programs with Java Pathfinder. In *Proceedings of SPIN*, 2005.