Fair Stateless Model Checking

Madanlal Musuvathi Shaz Qadeer
Microsoft Research
{madanm,qadeer}@microsoft.com

Abstract

Stateless model checking is a useful state-space exploration technique for systematically testing complex real-world software. Existing stateless model checkers are limited to the verification of safety properties on terminating programs. However, realistic concurrent programs are nonterminating, a property that significantly reduces the efficacy of stateless model checking in testing them. Moreover, existing stateless model checkers are unable to verify that a nonterminating program satisfies the important liveness property of livelock-freedom, a property that requires the program to make continuous progress for any input.

To address these shortcomings, this paper argues for incorporating a fair scheduler in stateless exploration. The key contribution of this paper is an explicit scheduler that is (strongly) fair and at the same time sufficiently nondeterministic to guarantee full coverage of safety properties. We have implemented the fair scheduler in the CHESS model checker. We show through theoretical arguments and empirical evaluation that our algorithm satisfies two important properties: 1) it visits all states of a finite-state program achieving state coverage at a faster rate than existing techniques, and 2) it finds all livelocks in a finite-state program. Before this work, nonterminating programs had to be manually modified in order to apply CHESS to them. The addition of fairness has allowed CHESS to be effectively applied to real-world nonterminating programs without any modification. For example, we have successfully booted the Singularity operating system under the control of CHESS.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification — formal methods, validation; D.2.5 [*Software Engineering*]: Testing and Debugging — debugging aids, diagnostics, monitors, tracing

General Terms Algorithms, Reliability, Verification

Keywords Concurrency, fairness, liveness, model checking, multithreading, shared-memory programs, software testing

1. Introduction

Concurrent programs are difficult to get right. Subtle interactions among communicating threads in the program can result in unexpected behaviors. These behaviors typically result in bugs that occur late in the software development cycle or even after the software is released. Traditional methods of testing, such as various forms of stress and random testing, more often than not miss these bugs.

PLDI'08, June 7–13, 2008, Tucson, Arizona, USA.


```
Phil1:
while ( true ) {
  Acquire ( fork1 );
  if ( TryAcquire ( fork2 ) )
    break;
  Release ( fork1 );
}
// eat
Release ( fork1 );
Release ( fork2 );
```

```
Phil2:
while ( true ) {
  Acquire ( fork2 );
  if ( TryAcquire ( fork1 ) )
    break;
  Release ( fork2 );
}
// eat
Release ( fork2 );
Release ( fork1 );
```

Figure 1. Example of a nonterminating program.

Model checking [5, 24] is a promising method for detecting and debugging deep concurrency-related errors. A model checker systematically explores the state space of given system and verifies that each reachable state satisfies a given property. This paper is concerned with *stateless* model checking, a style of state-space search first proposed in Verisoft [8]. A stateless model checker explores the state space of the program without capturing the individual program states. The program is executed under the control of a special scheduler that controls all the nondeterminism in the program. This scheduler systematically enumerates all execution paths of the program obtained by the nondeterministic choices.

Stateless model checking is particularly suited for exploring the state space of large programs, because precisely capturing all the essential state of a large program can be a daunting task. Apart from the global variables, heap, thread stacks, and register contexts, the state of a running program can be stored in the operating system, the hardware, and in the worst case, in a different machine across a network. Even if all the program state can be captured, processing such large states can be very expensive [12, 21].

On the downside, stateless model checking is directly applicable only to *terminating* programs. Such programs terminate under all executions and equivalently, have acyclic state spaces. In our experience, most realistic concurrent programs have cyclic state spaces. This paper introduces the novel technique of *fair stateless model checking* for effectively searching the state spaces of nonterminating programs.

Nontermination and cyclic state spaces present a significant obstacle to existing stateless model checkers. To illustrate the problem, consider the nonterminating program in Figure 1. The program is a variation of the dining philosophers example with two threads `Phil1` and `Phil2` trying to acquire two resources `fork1` and `fork2`. `Phil1` acquires `fork1` and then attempts to acquire `fork2` without blocking. If this attempt fails, then it releases `fork1` and retries. `Phil2` tries to acquire the two resources in the reverse

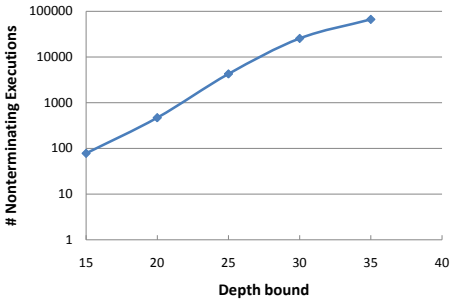

Figure 2. Number of nonterminating executions increases exponentially with the depth bound.

order. The retry loops in the two threads create cycles in the state space of the program.

A typical stateless model checker is ineffective in detecting errors in such programs for two fundamental reasons. First, to avoid divergence resulting from nonterminating executions, the model checker must be run with a depth bound. To get good coverage for safety verification, this bound must be large enough to allow exploring the deepest state in the state space. However, as the bound increases, the model checker spends exponentially more resources unrolling cycles in the state space than visiting new states. A useful measure of the wasteful work performed during the search is the number of non-terminating executions explored for a particular depth bound. Figure 2 shows that, for our example (Figure 1), as the depth bound increases the number of nonterminating executions explored increases exponentially. Second, nontermination introduces the possibility of livelocks, an entirely new class of errors characterized by the inability of the program to make progress. For example, the repeated execution of the transition sequence `Phil1: Acquire(fork1), Phil2: Acquire(fork2), Phil1: TryAcquire(fork2), Phil2: TryAcquire(fork1), Phil1: Release(fork1), Phil2: Release(fork2)` is a livelock. Depth-bounded stateless model checking does not have the ability to detect such errors.

Fair stateless model checking solves both the aforementioned problems by performing state-space search with respect to a *fair* and *demonic* scheduler. Our first key insight is that correct programs make continuous progress on fair schedules. A schedule is fair if every thread that is enabled infinitely often is scheduled infinitely often. Conversely, a schedule is unfair if a thread is starved of its chance to execute despite being enabled infinitely often. [1] For example, the schedule in which `Phil1` performs `Acquire(fork1)` and then `Phil2` repeatedly executes `Acquire(fork2), TryAcquire(fork1), Release(fork2)` is unfair. A cycle in the state space of a correct program corresponds to an unfair schedule in which an enabled thread is starved continuously so that the other threads participating in the cycle are unable to make progress. By performing state-space search with respect to a fair scheduler, the model checker is able to prune such cycles away. Note that a cycle in an incorrect program, such as the one in Figure 1, might correspond to a livelock. However, such an erroneous cycle must be fair, otherwise it would not be considered an error by the programmer. Our scheduler does not prune such cycles and will in fact generate an infinite execution in the limit. It is this ability to distinguish between fair and unfair executions that gives fair stateless model checking the ability to detect livelocks.

[1] In the literature, this notion of fairness is qualified as *strong* fairness. For brevity, we simply refer to this notion without the qualifier in this paper.

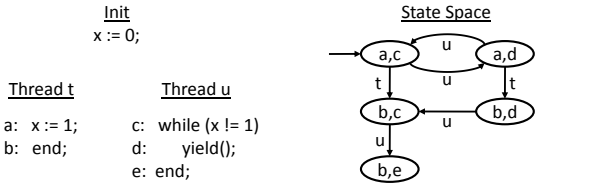

Figure 3. Example of a nonterminating program.

Obviously, a fair scheduler is restricted from making some scheduling decisions that are otherwise available to a scheduler with no fairness requirement. It is important to ensure that these restrictions do not reduce the coverage achieved during state-space exploration. Our second key insight enables us to do so. We observe that threads in correct programs indicate when they are unable to make progress by *yielding* the processor. A yield is usually indicated by the presence of a sleep operation or a timeout while waiting on a resource. To achieve fairness, the scheduler *only* penalizes yielding threads and prioritizes threads that are able to make progress. In particular, the scheduler is fully nondeterminstic in the absence of yield operations. Section 3 describes the fair scheduling algorithm and provides theoretical results to characterize its soundness. To support these theoretical results, Section 4 provides experimental results which indicate that our algorithm achieves complete state coverage on a variety of programs.

We have implemented our scheduler in the CHESS model checker. This algorithm extends the ability of CHESS to handle nonterminating programs. Prior to this implementation, any program given to the checker had to be manually modified to terminate under all schedules. This manual effort was a significant hurdle to the deployment of CHESS because in our experience, real programs are almost always nonterminating. By not requiring this manual effort, the fair scheduling algorithm has significantly improved the applicability of CHESS to real-world programs; we can now boot the Singularity operating system [13] under the control of CHESS. We present our evaluation, including several bugs found, in Section 4.

In summary, the main novel contributions of this paper are the following:

- We have introduced fair stateless model checking, a novel technique for systematic testing of nonterminating programs. Our method significantly enhances the applicability of stateless model checking to large programs with cyclic state spaces. In addition, it allows stateless model checkers to detect a new class of livelock errors.
- We have implemented our algorithm in the stateless model checker CHESS. The algorithm makes it much easier to apply CHESS to large programs and found three previously unknown errors in real-world programs of which two are livelocks.

2. Overview

In this section, we present an overview of our method for systematically testing nonterminating programs. We use the example program in Figure 3 to motivate the discussion. This program has two threads and a global variable `x` initially set to zero. The first thread `t` sets `x` to 1, while the second thread `u` spins in a loop waiting for the update of `x` by `t`. The state space of this program is shown at the

right of Figure 3. For this simple program, the state can be captured by the program counter of the two threads. For instance, the state `(a,c)` is the initial state where the two threads are about to execute the instructions at locations `a` and `c` respectively. The state space contains a cycle between `(a,c)` and `(a,d)`, resulting from the the spin loop of `u`. Obviously, this program does not terminate under the schedule that continuously runs `u` without giving a chance for `t` to execute.

Our method is applicable to programs that are expected to terminate under all fair schedules. That is, nontermination under a fair schedule is unexpected and is potentially an error. However, there is no requirement on these programs to terminate under unfair schedules. We call such programs *fair-terminating*. The program in Figure 3 is fair-terminating since its only infinite execution is not fair. This execution continuously starves thread `t` despite `t` being enabled infinitely often.

Our intuition for fair-terminating programs is based upon our observation of the test harnesses for real-world concurrent programs. In practice, concurrent programs are tested by combining them with a suitable test harness that makes them fair-terminating. A fair scheduler eventually gives a chance to every thread in the program to make progress ensuring that the program as a whole makes progress towards the end of the test. Such a test harness can be created even for systems such as cache-coherence protocols that are designed to "run forever"; the harness limits the number of cache requests from the external environment. In addition, the notion of fair termination coincides with the intuitive behavior programmers expect of their concurrent programs. For instance, one expects the program in Figure 3 to terminate when run on a real machine. This expectation is due to our implicit assumption that the underlying thread scheduler in the operating system is fair.

In this paper, we provide a solution to the following important problem:

> *Input:* A concurrent program Q and a safety property φ
> *Problem:* Determine if Q is fair-terminating and satisfies φ. If Q is not fair-terminating, produce a fair nonterminating execution of Q. If Q violates φ, produce a finite execution of Q violating φ.

All previous solutions proposed for this problem are *stateful*; they require capturing the state of the program Q. As discussed in the introduction, capturing the state of large program is error-prone and expensive. The main contribution of this paper is a practical *stateless* solution to this problem.

Our solution, called *fair stateless model checking*, uses a fair and demonic scheduler for systematically exploring the set of fair executions of the program Q. The scheduler maintains a partial order on the set of threads in each state. Intuitively, this partial order defines a scheduling priority over threads in each state — an enabled thread cannot be scheduled in a state if a higher priority thread, as determined by the partial order, is enabled in that state. The priority is updated appropriately during the execution of a program with the guarantee that any infinite execution generated by our scheduler is fair (Section 3). An execution obtained by unrolling an unfair cycle in the state space of a nonterminating program is pruned by our scheduler, thereby leading to a more efficient search.

While being fair, the scheduler must also be demonic and attempt to generate enough schedules to achieve full state coverage. For example, a scheduler that generates a single fair schedule is useless for finding bugs because it misses most behaviors of the program! Similarly, a round-robin scheduler does not consider many interleavings of the threads in the program. Ideally, it would be desirable for a fair scheduler to generate all possible fair executions of the program. But the set of all fair executions of a fair-terminating program, even for a fixed input, may be (enumerably) infinite. Therefore, it is impossible for any stateless model checker, including ours, to generate all fair executions in a bounded amount of time. However, for checking safety properties, it is only necessary to generate enough executions to cover all reachable states of the program.

To achieve full state-coverage, our scheduler depends on an important characteristic of correct programs. We observed that threads in correct programs indicate when they are unable to make progress by *yielding* the processor. Whenever a thread waits for a resource that is not available, it either blocks on the resource or yields the processor. A block or a yield tells the operating system scheduler to perform a context switch, hopefully to the thread holding the resource required by the waiting thread. If the waiting thread does not yield the processor and continues to spin idly, it will needlessly waste its time slice and slow down the program; such a behavior is consequently considered an error. Therefore, in addition to being fair-terminating, correct programs also satisfy the following *good samaritan* property: if a thread is scheduled infinitely often, then it yields infinitely often. The program in Figure 3 satisfies this property because there is a yield statement in the spin loop of thread `u`. Also, a thread that terminates after executing a finite number of steps obviously satisfies the good samaritan property.

Our scheduler introduces an edge in the priority order only when a thread yields and thereby indicates lack of progress. Thus, our scheduler ensures that in the absence of yield operations, the priority order remains empty and all threads have equal priority. At each scheduling point, the scheduler nondeterministically attempts all scheduling choices that respect the priority order. Our intuition is that programs are parsimonious in the use of the yield operations as their excessive use may unnecessarily decrease performance. Therefore, these operations are used sparingly, typically at the back edges of spin loops. Consequently, every reachable state is likely to be reachable via a yield-free execution along which our algorithm behaves like the standard nondeterministic scheduler used in model checkers. We provide theoretical results (in Section 3) characterizing the coverage of our algorithm. We also provide experimental evaluation (in Section 4) to show that our algorithm provides complete state coverage on a variety of realistic programs.

In summary, fair stateless model checking is a semi-algorithm which takes as input a program that is expected to satisfy the good samaritan property and be fair-terminating. There are four outcomes possible when this algorithm is applied to such a program.

1. The algorithm terminates with a safety violation.
2. The algorithm diverges and generates, in the limit, an infinite execution that violates the good samaritan property.
3. The algorithm diverges and generates, in the limit, an infinite fair execution.
4. The algorithm terminates without finding any errors.

In theory, the second and third outcomes manifest in an infinite execution being generated. In practice, it is not possible for a stateless model checker to identify or generate an infinite execution. Therefore, we ask the user to set a large bound on the execution depth. This bound can be orders of magnitude greater than the maximum number of steps the user expects the program to execute. The model checker stops if an execution exceeding the bound is reached and reports a warning to the user. This execution is then examined by the user to see if it actually indicates an error. In the rare case it is not, the user simply increases the bound and runs the model checker again. Using this mechanism, our algorithm is able to detect the livelock in the program of Figure 1.

3. Fair stateless model checking

In this section, we describe fair stateless model checking in detail. We fix a multithreaded program Q with a finite set Tid of threads. The program Q starts execution in its initial state s_0. At each step, one thread in Tid performs a transition and updates the state. In this presentation, we assume that the transition relation of each thread is deterministic and consequently thread scheduling is the only source of nondeterminism. However, our method is easily generalized to accommodate a nondeterministic but finitely-branching thread transition relation.

The program Q is equipped with state predicates $enabled(t)$ and $yield(t)$ for each thread $t \in Tid$. The predicate $enabled(t)$ is true in a state s iff thread t is enabled in s. The predicate $yield(t)$ is true in a state s iff thread t is enabled in s and executing thread t in s results in a yield.

An execution $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots$ is a finite or infinite sequence of states and transitions. Each such execution is equipped with a state predicate $sched(t)$ for each thread $t \in Tid$ such that for all $n \geq 0$, $sched(t)$ is true in s_n *if and only if* $t_n = t$. A finite execution $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots s_n$ is *terminating* if $enabled(t)$ is false at s_n for each $t \in Tid$. Such a state s_n is called a *deadlock*. A state s is *reachable* if it is the final state of an execution.

An infinite execution $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots$ is *fair* iff for all threads $t \in Tid$, if t is enabled infinitely often in σ then t is scheduled infinitely often in σ. This property is formalized as the following linear temporal logic [23] formula:

$$SF = \forall t \in Tid : \mathbf{GF}enabled(t) \Rightarrow \mathbf{GF}sched(t)$$

Every infinite execution σ of Q is expected to satisfy the following *good samaritan* property:

$$GS = \forall t \in Tid : \mathbf{GF}sched(t) \Rightarrow \mathbf{GF}(sched(t) \wedge yield(t))$$

Intuitively, this property states that for all threads t, if t is scheduled infinitely often, then in infinitely many of those t transitions thread t also yields. The fair stateless model checking algorithm, apart from detecting safety violations, also attempts to detect an infinite execution that either violates the good samaritan property or is fair.

We refer to the value of the predicates $enabled(t)$, $sched(t)$, and $yield(t)$ in state s as $s.enabled(t)$, $s.sched(t)$, and $s.yield(t)$ respectively. We also use $s.ES$ to refer to the set $\{x \in Tid \mid s.enabled(t)\}$ of threads enabled in state s. Given a relation $R \subseteq Tid \times Tid$ and a set $X \subseteq Tid$, we define

$$pre(R, X) = \{x \in Tid \mid \exists y \in Tid : (x, y) \in R \wedge y \in X\}.$$

We present the fair model checking algorithm (Algorithm 1) as a nondeterministic fair scheduler. Our algorithm makes explicit the available nondeterministic choices at each scheduling point. It is easy to augment this description with either a stack to perform depth-first search or a queue to perform breadth-first search. To focus on the essence of our algorithm, we have elided the (standard) search mechanism from the algorithm description.

The algorithm takes as input a multithreaded program Q together with its initial state $init$. It assumes that the program Q comes with a function $NextState$ that takes a state s and a thread t and returns the state that results from executing t in state s. The algorithm starts with the initial state and an empty execution. In each iteration of the loop (lines 7–30), the algorithm extends the current execution by one step. The algorithm terminates with a complete terminating execution, when the $return$ statement on line 9 is executed.

Each thread $t \in Tid$ partitions an execution σ into windows; a window of thread t lasts from a state immediately after a yielding transition by thread t to the state immediately after the next yielding transition by t. Our algorithm maintains for each state s several

```
1  init.P := {};
2  ∀u ∈ Tid : init.E(u) := {};
3  ∀u ∈ Tid : init.D(u) := Tid;
4  ∀u ∈ Tid : init.S(u) := Tid;
5  curr := init;
6  while true do
7      T := curr.ES \ pre(curr.P, curr.ES);
8      if T = {} then
9          return;
10     end
11     t := Choose(T);
12     next := NextState(curr, t);
13     next.P := curr.P \ (Tid × {t});
14     foreach u ∈ Tid do
15         next.E(u) := curr.E(u) ∩ next.ES;
16         if u = t then
17             next.D(u) :=
               curr.D(u) ∪ (curr.ES \ next.ES);
18         else
19             next.D(u) := curr.D(u);
20         end
21         next.S(u) := curr.S(u) ∪ {t};
22     end
23     if curr.yield(t) then
24         H := (next.E(t) ∪ next.D(t)) \ next.S(t);
25         next.P := next.P ∪ ({t} × H);
26         next.E(t) := next.ES;
27         next.D(t) := {};
28         next.S(t) := {};
29     end
30     curr := next;
31 end
```

Algorithm 1: Fair stateless model checking

auxiliary predicates that record information about a window of thread t.

1. $S(t)$ is the set of threads that have been scheduled since the last yield by thread t.
2. $E(t)$ is the set of threads that have been continuously enabled since the last yield by thread t.
3. $D(t)$ is the set of threads that have been disabled by some transition of thread t since the last yield by thread t.

In addition to these predicates, each state s also contains a relation $s.P$ which represents a priority ordering on threads. Specifically, if $(t, u) \in s.P$ then t will be scheduled in s only when $s.enabled(t)$ and $\neg s.enabled(u)$.

In each iteration, the algorithm first computes T (line 7), the set of schedulable threads that satisfy the priorities in $curr.P$. If T is empty, then the execution terminates. Otherwise, the algorithm selects a thread t *nondeterministically* from T (line 11) and schedules t to obtain the $next$ state. It is this nondeterminism inherent in the execution of the $Choose(T)$ on line 11 that a model checker must explore. As explained earlier, it is easy to add systematic depth-first or breadth-first search capability to our algorithm. Line 13 removes all edges with sink t from P to decrease the relative priority of t.

The loop at lines 14–22 updates the auxiliary predicates for each thread $u \in Tid$. The set E of continuously enabled threads is updated by taking the intersection of its current value with the set of enabled threads in $next$ (line 15). The set D of threads disabled by thread t is updated by taking the union of its current value with

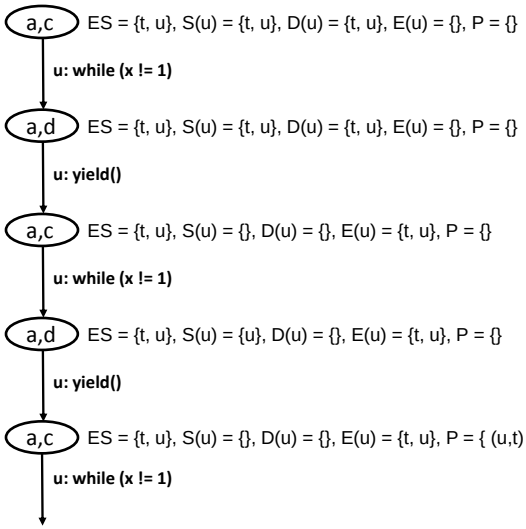

Figure 4. Emulation of Algorithm 1 on the spin loop in Figure 3. When thread u yields in the second transition from state (a,d), the P in the subsequent state ensures that u does not enter the spin loop the second time.

the set of threads disabled by the latest transition (line 17). The set of scheduled threads is updated on line 21.

Finally, if the transition just executed is a yielding transition, then the data structures are updated appropriately to mark the beginning of a new window of u (line 23–29). The set H computed on line 24 contains only those threads that were never scheduled in the current window of thread t and were either continuously enabled, or disabled by thread t at some point in the window. Line 25 reduces the priority of t with respect to the threads in H.

Figure 4 shows an emulation of Algorithm 1 for the program in Figure 3. For conciseness, Figure 4 only shows the emulation when the scheduler attempts to schedule the thread u continuously. We focus the emulation on the values of the relation P and the predicates $S(u)$, $D(u)$, and $E(u)$. The relation P is initialized to be empty. The predicates $S(u)$, $D(u)$ and $E(u)$ are initialized in such a way that their values remain unchanged until the first yield of thread u. These values also provide the additional guarantee that the update of P at the first yield of any thread is guaranteed to leave the value of P unchanged. This behavior ensures that the first window of any thread begins after its first yield at which point the predicates $S(u)$, $D(u)$ and $E(u)$ get initialized appropriately.

In the emulation, the scheduler executes thread u continuously. Starting from the initial state (a,c), the first window of u begins once the scheduler has scheduled u twice. At this point, u has gone through the spin loop once and the state is (a,c) again. In this state, $P = \{\}$, $S(u) = \{\}$, $D(u) = \{\}$, and $E(u) = ES = \{t, u\}$. When u is executed for one more step, u is added to $S(u)$ and the state becomes (a,d). In this state, $yield(u)$ is true as u will yield if executed from this state. However, the P relation is still empty allowing the scheduler to choose either of the two threads.

If the scheduler chooses to schedule u again, the thread completes the second iteration of the loop and the program enters the state (a,c). Algorithm 1 adds the edge (u, t) to P because the set H on line 24 evaluates to $\{t\}$. Thus, the algorithm giving the yielding thread u a lower priority than the pending thread t. This update to P makes the set of scheduler choices $T = \{t\}$. Thus, the scheduler is forced to schedule t, which enables u to exit its loop.

Generalizing this example, if the thread t was not enabled in the state (a,c), say if t was waiting on a lock currently held by u, the scheduler will continue to schedule u till it releases the lock. Further, if t was waiting on a lock held by some other thread v in the program, the fairness algorithm will guarantee that eventually v makes progress releasing the lock.

THEOREM 1. *Every infinite execution σ generated by Algorithm 1 satisfies the property $GS \Rightarrow SF$.*

PROOF. We do the proof by contradiction. Suppose Algorithm 1 generates an infinite execution $\sigma = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \ldots$ that satisfies GS but does not satisfy SF. Therefore, there is a thread u such that σ satisfies $\mathbf{GF}enabled(u) \wedge \mathbf{FG}\neg sched(u)$. That is, the execution σ eventually reaches a state s_i after which u is never scheduled but is enabled infinitely often. Let T be the set of threads that are scheduled infinitely often in σ. Since Tid is finite and σ is an infinite execution, the set T is nonempty. Since σ satisfies GS, every thread in T must yield infinitely often. There are two cases: thread u is enabled forever after s_i or thread u is both enabled infinitely often and disabled infinitely often after s_i.

Case 1: Suppose thread u is enabled forever after s_i. Consider an arbitrary $t \in T$. Since σ satisfies GS and t is scheduled infinitely often in σ, there exist j and k such that $i < j < k$, $s_j.sched(t)$, $s_j.yield(t)$, $s_k.sched(t)$ and $s_k.yield(t)$. Consider the iteration of the loop in lines 6–31 of Algorithm 1 in which the $curr = s_k$. Since u is forever enabled but never scheduled after s_i, we have that $u \in next.E(t)$ and $u \notin next.S(t)$ at line 24. Therefore, at line 25 $u \in H$ and the edge (t, u) is added to $next.P$. Since thread u is continuously enabled after s_i and therefore after s_k, the edge (t, u) precludes the scheduling of t after s_k (line 7). This is a contradiction since t is scheduled infinitely often.

Case 2: Suppose thread u is both enabled infinitely often and disabled infinitely often after s_i. Since T is finite, there must be some thread $t \in T$ that disables u infinitely often. Since σ satisfies GS and t is scheduled infinitely often in σ, t disables u at some point after i and between two states where $yield(t)$ holds. Formally, there exist j such that $i < j < k \leq l$, $s_j.sched(t)$, $s_j.yield(t)$, $s_k.sched(t)$, $s_k.enabled(u)$, $\neg s_{k+1}.enabled(u)$, $s_l.sched(t)$, and $s_l.yield(t)$. Consider the iteration of the loop in lines 6–31 of Algorithm 1 in which the $curr = s_l$. Since u is never scheduled after s_i, we have that $u \in next.D(t)$ and $u \notin next.S(t)$ at line 24. Therefore, at line 25 $u \in H$ and the edge (t, u) is added to $next.P$. Since thread u is never scheduled after s_i and therefore after s_k, the edge (t, u) is present in $s_n.P$ for all $n \geq k$. This edge precludes the scheduling of t after s_k (line 7) whenever thread u is also enabled. This is a contradiction since thread t disables thread u infinitely often. □

Theorem 1 yields the following termination guarantee about Algorithm 1.

THEOREM 2. *If no infinite execution of Q satisfies the property $GS \Rightarrow SF$, then Algorithm 1 terminates on Q.*

PROOF. We do the proof by contradiction. Suppose Algorithm 1 does not terminate on Q. Then the tree of executions explored by the algorithm is infinite. Since Tid is finite, this execution tree is finitely branching. By König's lemma, there must be an infinite execution in this tree. Since every execution generated by Algorithm 1 is an execution of Q, this execution must satisfy $GS \Rightarrow SF$ (by Theorem 1). Hence, we arrive at a contradiction. □

Now, we prove certain desirable properties of the fair model checking algorithm.

THEOREM 3. *At line 7 of Algorithm 1, the set T is empty if and only if the set $curr.ES$ is empty.*

PROOF. The proof relies on the fact that the P relation when viewed as edges in a graph with nodes from Tid contains no cycles. The loop invariant at line 6 requiring that $curr.P$ is an acyclic relation is sufficient to prove our theorem.

We first prove this loop invariant. Upon loop entry, we have $curr = init$. The relation $curr.P$ is empty and the invariant is trivially true. Consider an arbitrary iteration of the loop. We assume the loop invariant at the beginning of this iteration and prove it at the end. In each iteration of the loop, whenever outgoing edges from t are added to P at line 25, all incoming edges into t have already been removed earlier at line 13. Line 21 adds t to $next.S(u)$ for all $u \in Tid$ and therefore it is guaranteed that $t \in next.S(t)$ at line 24 and $t \notin H$ at line 25. Thus, even line 25 does not add any incoming edges into t and each iteration of the loop leaves P acyclic.

We now show that the loop invariant implies our theorem. Clearly, if $curr.ES$ is empty at line 7 then T is also empty. We prove the other direction by contradiction. Suppose T is empty but $curr.EnabledSet$ is nonempty. Therefore $curr.ES \subseteq pre(curr.P, curr.ES)$. Consider the projection of the relation $curr.P$ relation on to the set $curr.ES$. Since $curr.P$ is acyclic and $curr.ES$ is nonempty, this projection is a nonempty acyclic relation and therefore contains a maximal element. That is, there exists $t \in curr.ES$ such that $\forall u \in curr.ES : (t, u) \notin curr.P$. This contradicts our assumption $curr.ES \subseteq pre(curr.P, curr.ES)$. □

Theorem 3 guarantees that Algorithm 1 never reports a false deadlock. In practice, this means that the algorithm can always drive the program to a terminating state, without requiring the execution to be pruned at a partial execution. Such pruning, as can typically happen with depth-bounding techniques, avoids wasting scarce model checking resources.

While the theorems above hold even for infinite-state systems, the theorems stated below provide intuition for the efficacy of the fair model checking algorithm on finite-state systems. For the remainder of this section, we assume that Q is a finite-state program.

For finite-state systems, all infinite behaviors traverse cycles in the state space. A *cycle* τ is a transition sequence $x_0 \xrightarrow{t_0} x_1 \cdots x_n \xrightarrow{t_n} x_0$ such that the states $x_0, x_1, \ldots, x_n$ are all distinct. The cycle τ is reachable if the state x_0 is reachable. The cycle τ is *fair* if for every thread $t \in Tid$, either $\neg x_i.enabled(t)$ for all $i \in [0, n]$ or $t = t_i$ for some $i \in [0, n]$. The cycle τ is *unfair* if it is not fair. Note that an infinite execution that traverses an unfair cycle forever is not fair.

The following theorem shows that our algorithm unrolls an unfair cycle fully at most twice and thus significantly reduces wasteful search.

THEOREM 4. *Suppose every infinite execution of Q satisfies the property GS, $s_0 \xrightarrow{t} s_1 \cdots x_0$ is a finite execution of Q, and $x_0 \xrightarrow{t_0} x_1 \cdots x_n \xrightarrow{t_n} x_0$ is an unfair cycle in Q. Then, Algorithm 1 does not generate the execution $s_0 \xrightarrow{t} s_1 \cdots x_0 \xrightarrow{t_0} x_1 \cdots x_n \xrightarrow{t_n} x_0 \xrightarrow{t_0} x_1 \cdots x_n \xrightarrow{t_n} x_0 \xrightarrow{t_0} x_1 \cdots x_n$.*

PROOF. Since $x_0 \xrightarrow{t_0} x_1 \cdots x_n \xrightarrow{t_n} x_0$ is an unfair cycle, there is a thread u such that $u \neq t_i$ for all $i \in [0, n]$ and $x_i.enabled(u)$ for some $i \in [0, n]$. Let $N(i)$ denote $(i + 1) mod (n + 1)$. Since $x_i.enabled(u)$ for some $i \in [0, n]$, there are only two cases: either $x_i.enabled(u)$ for all $i \in [0, n]$ or there exists $i \in [0, n]$ such that $x_i.enabled(u)$ and $\neg x_{N(i)}.enabled(u)$.

Case 1: Suppose $x_i.enabled(u)$ for all $i \in [0, n]$. Since every infinite execution of Q satisfies GS, there exists $i \in [0, n]$ such that $x_i.yield(t_i)$ is true. Therefore the edge (t_i, u) is in the priority graph after the execution of t_i the second time the unfair cycle is executed. Since u is never scheduled, this edge is not removed, and consequently t_i cannot be scheduled at the next occurrence of x_i.

Case 2: Suppose there exists $i \in [0, n]$ such that $x_i.enabled(u)$ and $\neg x_{N(i)}.enabled(u)$. Since every infinite execution of Q satisfies GS, there exists $j \in [0, n]$ such that $t_i = t_j$ and $x_j.yield(t_j)$ is true. Therefore the edge (t_j, u) is added to the priority graph after the execution of t_j the second time the unfair cycle is executed. Since u is never scheduled, this edge is not removed, and consequently $t_i = t_j$ cannot be scheduled at the next occurrence of x_i. □

Now, we present two theorems that characterize the soundness of our algorithm on finite-state systems. Consider a finite transition sequence $\alpha = x_0 \xrightarrow{t_0} x_1 \cdots x_n$. The *yield count* of thread t in α, denoted by $\delta(\alpha, t)$, is the cardinality of the set $\{0 \leq i < n \mid t = t_i \wedge x_i.yield(t)\}$. The *yield count* of α, denoted by $\delta(\alpha)$ is the maximum of $\delta(\alpha, t)$ over all threads $t \in Tid$. The *yield count* of a reachable state s is the minimum of $\delta(\sigma)$ over all executions σ whose final state is s. The following theorem captures the soundness guarantee of our algorithm for safety properties.

THEOREM 5. *Algorithm 1 either generates an infinite execution or visits every reachable state of Q whose yield count is zero.*

PROOF. Suppose Algorithm 1 does not generate an infinite execution. Therefore, the tree of executions explored by the algorithm is finite (by König's lemma). This tree is guaranteed to contain all executions whose yield count is zero because along such an execution the priority graph remains empty throughout. Every reachable state of Q with yield count zero is the final state of an execution in which there are no yields. Therefore, the algorithm eventually visits all such reachable states. □

Every infinite execution generated by our algorithm reveals a liveness error (Theorem 1). Theorem 5 indicates that our algorithm is sound with respect to safety properties if the program Q does not have any liveness errors and all reachable states are reachable by yield-free executions. Theorem 6 below captures the soundness of our algorithm with respect to liveness properties as well.

THEOREM 6. *Suppose x_0 is a reachable state of Q whose yield count is zero and $\tau = x_0 \xrightarrow{t_0} x_1 \cdots x_n \xrightarrow{t_n} x_0$ is a fair cycle whose yield count is at most one. Then Algorithm 1 generates an infinite execution.*

PROOF. We do the proof by contradiction. Suppose Algorithm 1 does not generate an infinite execution. By Theorem 5, the state x_0 is eventually visited with an empty priority graph. If the yield count of τ is zero, then there are no yields in τ and τ can be executed repeatedly to generate an infinite execution. Suppose the yield count of τ is one. Since τ is fair, every thread that is enabled anywhere in the cycle is also scheduled in the cycle. Moreover, a thread t may yield at most once in τ. Therefore, the set $S(t)$ contains both $E(t)$ and $D(t)$ at the unique yield point of t, if any. Consequently, the yield of t does not add any edges to P. □

As discussed in Section 2, we expect that all reachable states are reachable by a yield-free execution due to the parsimonious use of the yield operation by real programs. If this is not the case, then our algorithm can be parameterized by a small constant $k > 0$ so as to only process every k-th yield of a thread. The soundness theorems (both for safety and liveness) for the parameterized algorithm are straightforward generalizations of the corresponding theorems stated above.

4. Evaluation

This section presents the empirical evaluation of the fair demonic scheduler described in Section 3. We have implemented our algo-

Programs	LOC	Threads	Synch Ops
Dining Philosophers	54	3	48
Work-Stealing Queue	1266	3	99
Promise	14044	3	26
APE	18947	4	247
Dryad Channels	16036	5	273
Dryad Fifo	18093	25	4892
Singularity kernel	174601	14	167924

Table 1. Characteristics of input programs to CHESS

rithm in the CHESS software model checker. CHESS is designed for systematic testing of shared-memory multithreaded programs. To use CHESS, the user provides a test case that exercises a concurrent scenario. CHESS executes this test repeatedly, while controlling the thread schedule such that every execution of the test takes a different interleaving. CHESS is stateless and avoids capturing any state, including the initial state of the program, during state-space search.

The implementation of the fair scheduler in CHESS maintains data structures to implement the auxiliary state used in Algorithm 1. An important issue is the inference of yielding transitions; our implementation treats every synchronization operation with a finite timeout and every explicit processor yield as yielding operations. Another important issue is the integration of fair-scheduling with the context-bounded search [22] strategy implemented in CHESS. In a concurrent execution, a *preemption* occurs when the scheduler forces a context switch despite the current running thread being enabled. Context-bounded search explores only those executions in which the number of preemptions is bounded by a small number provided by the user of CHESS. Fair scheduling is easily combined with context-bounding. The only subtle aspect of the combination is that fair scheduling can introduce a preemption when the currently running thread gets a lower priority than another enabled thread. For soundness of the context-bounded search, it is important to *not* count such preemptions.

4.1 Ability to handle large nonterminating programs

Prior to the implementation of the fair scheduler, CHESS, like other stateless model checkers, could only handle terminating programs. As depth bounding was unsatisfactory for our purposes, any input program with nonterminating behavior required manual modification. As an example of the effort required, consider the simple program in Figure 3. One can fix the nonterminating behavior by introducing a synchronization variable that `u` blocks on when waiting for an update to `x`. In addition, the behavior of `t` (and all other threads that access `x`) should be modified to appropriately signal the synchronization variable after an update to `x`, a non-local change to the program. Finally, one has to ensure that the introduced synchronization does not result in deadlocks due to adverse interactions with existing synchronizations in the program. In practice, such modifications typically require intimate knowledge of the program, and in our experience, are difficult and error-prone. Previously, it took us several weeks to prepare a realistic program as an input to CHESS.

With the fair scheduler in place, CHESS could readily handle nonterminating programs. Table 1 describes the programs CHESS is currently able to handle. Table 1 also provides the maximum number of threads created and synchronization operations performed per execution of these programs in CHESS. In particular, we are able to systematically test the *entire* boot and shutdown process of the Singularity operating system [13]. Also, we have run CHESS on *unmodified* versions of Dryad, a distributed execution engine for coarse-grained data-parallel applications [15], and APE (Asynchronous Processing Environment), a library in the Windows operating system that provides a set of data structures and functions for asynchronous multithreaded code.

Apart from these large programs, CHESS is also able to handle low-level synchronization libraries that typically employ nonblocking algorithms. Manually modifying them to be terminating is either impossible or requires algorithmic changes. We have applied CHESS to an implementation [20] of the work-stealing queue algorithm originally designed for the Cilk multithreaded programming system [7], and *Promise*, a library for data-parallel programs.

4.2 Coverage of safety properties

This section demonstrates that the fairness algorithm is effective for checking safety properties. First, we show that the algorithm achieves 100% state coverage for the first two programs in Table 1. Also, we show that fairness significantly improves the speed of the state space search, for various search strategies. Finally, we demonstrate the ability of the the fairness algorithm to find both existing and previously unknown safety violations in large programs.

All the experiments described in the paper were performed on an off-the-shelf computer with Intel Xeon 2.80GHz CPU with 2 processors and 3 GB of memory running Windows Vista Enterprise operating system.

4.2.1 State coverage

CHESS is a stateless model checker and thus does not have the capability to capture program states. To measure state coverage, we manually added facilities to extract states for two examples; the dining philosophers and the work-stealing queue. The state of these programs consists of the state of all global variables, the heap, and the stack of all threads in the program. While the bits comprising the state can be automatically extracted, we had to manually abstract the (infinite) state of the program into a reasonable, finite representation. Also, in order to avoid multiple representations of behaviorally equivalent heaps, we used a simple heap-canonicalization algorithm [14].

Table 2 shows the results from our coverage experiments for these two examples, each with two configurations. For each of the configurations, we used four search strategies — a context-bounded search with bounds (cb) from 1 to 3 and a depth-first search (dfs). For each strategy, we ran CHESS with and without the fairness algorithm. As termination is not guaranteed without fairness, the search proceeds only upto a depth bound (db) varying from 20 to 60. Once the depth-bound is reached, a random search [17] is performed until the end of the execution is reached. New states visited during the random search are included while measuring state coverage. To measure the total number of states reachable with a strategy, we also performed a stateful search of the state space and stored the state signatures in a hash table. We used this table to check if the subsequent runs cover all of the states.

In our experiments, we found that the fairness algorithm achieves 100% coverage on all but one of the cases. The fairness algorithm times out for a depth-first search strategy on the work-stealing queue with two stealers. The fourth column in Table 2 shows the number of states explored with fairness, and except for the one case above, this number is greater than or equal to the total number of states in the third column. The number of states explored with fairness is larger than the total number of states, as the fairness algorithm introduces additional preemption points. This essentially forces the fairness algorithm to visit states that are beyond the current context-bound.

For comparison, Table 2 also shows the number of states visited for runs without fairness with different depth bounds. For some runs, the search with small depth bounds terminate without visiting all the states. In other cases with larger depth-bounds, the search

Configuration	Search Strategy	Total States	With Fairness	Without fairness db=20	db=30	db=40	db=50	db=60
Dining Philosophers 2 philosophers	cb=1	27	27	27	27	27	27	27
	cb=2	28	28	28	28	28	28	28
	cb=3	29	29	29	29	29	29	29
	dfs	29	29	29	29	29	29	29
Dining Philosophers 3 philosophers	cb=1	102	102	102	102	102	102	102
	cb=2	144	144	143	144	144	144	144
	cb=3	167	171	167	169	169	169	169
	dfs	177	177	174	177	177	168*	139*
Work-Stealing Queue 1 stealer	cb=1	278	278	236	278	278	278	278
	cb=2	814	814	554	765	814	814	814
	cb=3	1287	1297	694	1133	1287	1287	1287
	dfs	1726	1726	871	1505	1726	1307*	683*
Work-Stealing Queue 2 stealers	cb=1	350	378	238	334	350	350	350
	cb=2	1838	2000	971	1630	1822	1838	1838
	cb=3	3271	3311	1805	2955	3269	3271	3271
	dfs	4826	1321*	1686*	1239*	460*	245*	245*

Table 2. Number of states visited for the context-bounded and depth-first strategies both with and without fairness. Search without fairness is not guaranteed to terminate and thus needs to be pruned at a depth bound. The state coverage achieved for runs that did not terminate within 5000 seconds is marked with a *.

times out. For the work-stealing queue with two stealers, all runs using the depth-bounded strategy time out.

4.2.2 Rate of state coverage

Fair stateless search can be more efficient as it does not unroll unfair cycles in the state space (Theorem 4). To quantify this, Figures 5 and 6 show the time taken to complete the search for two of the four configurations in Table 2. The results were similar for the other two (smaller) configurations. The figures show the time taken to complete the search for each of the search strategies with and without fairness. For the runs without fairness, we report the time for various depth bounds. Note, the y-axis on these figures is in log scale. The runs with fairness explores the state space exponentially faster than the runs without fairness, without sacrificing state coverage. In Figure 6, for cb=3 strategy, the run with a depth bound of 20 completes faster than the run with fairness but does not cover all states. Also, the depth-first strategy times out in all runs.

4.2.3 Ability to find errors

The experiments above show that fairness improves the efficacy of safety checking without sacrificing soundness for two programs. On larger programs, for which state extraction is not manually feasible, we demonstrate the efficacy of the fairness algorithm indirectly by demonstrating its ability to find safety errors in the programs.

A prior version of CHESS had found six bugs on versions of the work-stealing queue and the Dryad channels, modified to be terminating. We ran CHESS on the unmodified programs with a context-bound of 2 preemptions, both with and without fairness. Since these programs are nonterminating, we set the depth-bound to 250 for the search without fairness. This depth-bound is the minimum required to find these errors. Table 3 compares the performance of the search with fairness to the search without fairness. As the table shows the fairness algorithm finds the first five errors much faster both in terms of the number of executions explored prior to the buggy execution and the time taken for the search. The sixth error is not found by the search without fairness.

The seventh row in Table 3 shows a previously unknown bug that the fairness algorithm found in Dryad. The bug is caused by an incorrect fix of bug 3 by the developer of Dryad. This error is also not found by the search without fairness.

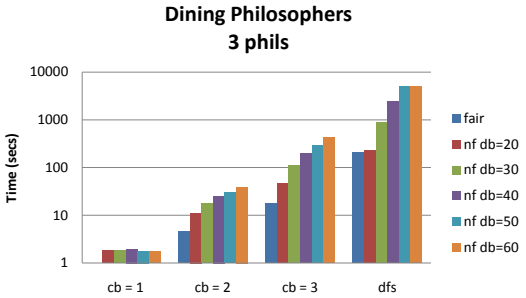

Figure 5. Dining philosophers (3).

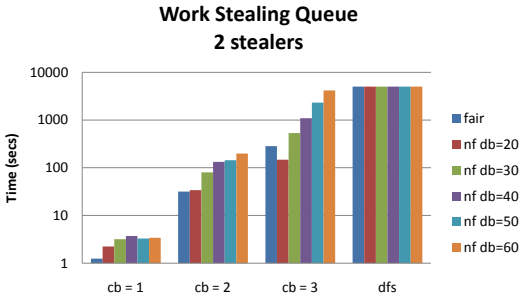

Figure 6. Work-stealing queue (2).

4.3 Liveness violations

The fairness algorithm diverges in two cases, when the program violates the good samaritan property (Section 2) or contains a livelock. Both these outcomes indicate correctness or performance errors in the program. We demonstrate one instance of each violation that CHESS found in existing programs.

Bugs	No. of executions		Time (secs)	
	With Fairness	Without Fairness	With Fairness	Without Fairness
WSQ bug 1	82	182	2	8
WSQ bug 2	112	432	8	24
WSQ bug 3	212	843	12	74
Dryad bug 1	310	11024	8	273
Dryad bug 2	4002	47030	114	1247
Dryad bug 3	25113	-	754	>7200
Dryad bug 4	21014	-	677	>7200

Table 3. Number of executions of the test and the time required to find errors with and without fairness in work-stealing queue (WSQ) and Dryad channels. The fourth Dryad error is a previously unknown error that CHESS found in the fix of the first three errors.

```
void Worker::Run(Object obj) {
    while (!stop) {
        while (!stop && task != null) {
           // perform task
           ...
           task = PopNextTask();
        }

        if (!stop) {
            task = group.Idle(this);
        }
    }
}

Task WorkerGroup::Idle(Worker currentWorker) {
    while (!stop) {
        ...
        // No work to be found
        // Yield to other threads.
        currentWorker.YieldExponential();
        ...
    }
    return null;
}
```

Figure 7. Violation of the good samaritan property. Under certain cases, the outer loop in `Worker::Run` results in the thread spinning idly till time-slice expiration.

4.3.1 Good samaritan property violation

We used CHESS to test the implementation of a library that provides efficient parallel execution of tasks. This library maintains a collection of worker threads, partitioned into a set of worker groups. CHESS detected a violation of the good samaritan property during the shutdown of the library. There is a field called `stop` in both the `Worker` and `WorkerGroup` classes. During the shutdown process, the `stop` field in a worker group and the `stop` field in each worker of that worker group is set to true, causing all the workers to eventually finish. However, there is a small window of time during which the `stop` field in the worker group is true but the `stop` field in one of the workers is false. In this situation, if the queue of tasks is empty, then the worker spins in a loop without yielding the processor until its time-slice expires. This behavior starves other threads, potentially including the one that is responsible for setting the `stop` field of the worker to true.

```
volatile int x;
//...
int x_temp = InterlockedRead(x);
if(common case 1) break;
if(common case 2) break;
...
// spin in the uncommon case
while(x_temp != 1){
   Sleep(1); //yield
   // BUG: should read x once again
}
```

Figure 8. The spinloop incorrectly waits on a temporary cache of the global variable, resulting in a livelock.

4.3.2 Livelock in Promise

We used CHESS to test the implementation of *promises*, a concurrency primitive for specifying data parallelism. The implementation is optimized for efficiency and selectively uses low-level hardware primitives for performance. CHESS detected a livelock in promises caused by simple programming error. While we are unable to provide the actual code snippet, Figure 8 shows pseudo-code exhibiting the same error.

For the sake of performance, programmers tend to make local copies of shared global variables. The livelock in Figure 8 occurs when the program erroneously waits for the local copy to change, without updating the copy with the value in the global variable. This bug was hard to detect as it only occurred in those rare thread interleavings in which the common cases shown in the pseudo-code were inapplicable.

5. Related work

The need for fairness when reasoning about concurrent programs is well known [19, 2, 6, 3, 10]. Of the different useful notions of fairness [6, 18], this paper deals with strong fairness (also known as *strong process fairness* [3] or *fairness* [19]).

Fairness has also been studied extensively in the context of model checking [5, 24, 25, 1, 12] of temporal logic specifications, all of which deal with stateful model checking. To our knowledge, this paper is the first to propose fairness as a means of improving the efficacy of stateless model checking for safety verification. Also, this paper is the first to extend the ability of stateless model checking to comprehensively detect livelocks.

Our fair scheduling algorithm is related to the explicit scheduler construction in Apt and Olderog [2, 6]. This scheduler is primarily used as a proof methodology for proving the termination of concurrent programs. In particular, the scheduler requires generating a *random* integer for the priority of a thread after every step. Such "unbounded nondeterminism" [10], while useful in generating *all* fair schedules of a program, cannot be effectively implemented in a model checker. In contrast, our algorithm requires a finite choice among a subset of enabled threads at each step of the execution.

Most operating system schedulers employ mechanisms to fairly share resources among competing threads and users. These algorithms typically manipulate priorities [16, 11] based on resource usage or use randomized schemes [27] to guarantee fairness. These algorithms are not designed to expose the nondeterministic choices of the scheduler and cannot be used in a model checker.

The model checker described in this paper belongs to the class that directly execute programs [8, 26, 21, 28] (as opposed to analyzing abstract program models). The technique of stateless model checking was proposed in Verisoft [8] and has been successful for systematically testing industrial concurrent systems [4]. Stateless

model checking typically relies on partial-order reduction techniques [9] to reduce the state space explored. Partial-order reduction can only determine the equivalence of two executions of the same length and thus is inherently incapable of detecting equivalence of executions that revisit the same state in a cycle. Partial-order reduction, however, can be used to significantly reduce the set of all fair schedules of fair-terminating programs, an interesting avenue of future research that we are currently pursuing.

Killian et al. [17] use a hybrid stateful and stateless technique to find liveness errors in network protocols. Their algorithm can only find those liveness violations that are characterized by the presence of "dead states" and will not find the livelock in the dining philosopher example (Section 1) or the violation of the good samaritan property, described in Section 4.3.1. Also, their algorithm provides no soundness guarantees of finding livelocks.

6. Conclusions and future work

This paper proposes the use of a fair scheduler in stateless model checking. Fairness both enables the detection of liveness errors and substantially improves the efficiency of safety verification in a stateless model checker. The incorporation of the fair scheduling algorithm in the CHESS model checker has significantly improved the applicability of CHESS to large nonterminating programs. The fairness-enhanced CHESS has found many liveness errors in several industry-scale programs.

Currently, CHESS checks two liveness properties: fair termination and the good-samaritan rule. We would like to extend CHESS to check an arbitrary liveness property. To motivate this work, we are currently identifying liveness properties that are useful for multithreaded software. We are also investigating extensions to the fair scheduler to handle unbounded thread creation.

Acknowledgements

We thank Tom Ball for help with the evaluation of the fair scheduler. We also thank Patrice Godefroid, Andreas Podelski, Mihalis Yannakakis, and the reviewers for valuable comments on a prior version of the paper.

References

[1] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, 1990.

[2] K.R. Apt and E.-R. Olderog. Proof rules and transformations dealing with fairness. *Science of Computer Programming*, 3:65–100, 1983.

[3] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *POPL 87: Principles of Programming Languages*, pages 189–198, 1987.

[4] Satish Chandra, Patrice Godefroid, and Christopher Palm. Software model checking in practice: an industrial case study. In *ICSE 02: International Conference on Software Engineering*, pages 431–441, 2002.

[5] E.M. Clarke and E.A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[6] Nissim Francez. Fairness. In *Texts and Monographs in Computer Science*. Springer-Verlag, 1986.

[7] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI 98: Programming Language Design and Implementation*, pages 212–223. ACM Press, 1998.

[8] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL 97: Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

[9] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. LNCS 1032. Springer-Verlag, 1996.

[10] Orna Grumberg, Nissim Francez, and Shmuel Katz. Fair termination of communicating processes. In *PODC 84: Principles of Distributed Computing*, pages 254–265. ACM Press, 1984.

[11] Joseph L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, 1993.

[12] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[13] Galen C. Hunt, Mark Aiken, Manuel Fhndrich, Chris Hawblitzeland Orion Hodson, James R. Larus, Steven Levi, Bjarne Steensgaard, David Tarditi, and Ted Wobber. Sealing OS processes to improve dependability and safety. In *Proceedings of the EuroSys Conference*, pages 341–354, 2007.

[14] Radu Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE 01: Automated Software Engineering*, pages 254–261, 2001.

[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the EuroSys Conference*, pages 59–72, 2007.

[16] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.

[17] Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI 07: Symposium on Networked Systems Design and Implementation*, pages 243–256, 2007.

[18] M. Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371–386, 1989.

[19] Daniel J. Lehmann, Amir Pnueli, and Jonathan Stavi. Impartiality, justice and fairness: The ethics of concurrent termination. In *ICALP 81: International Conference on Automata Languages and Programming*, pages 264–277, 1981.

[20] Daan Leijen. Futures: a concurrency library for C#. Technical Report MSR-TR-2006-162, Microsoft Research, 2006.

[21] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *OSDI 02: Operating Systems Design and Implementation*, pages 75–88, 2002.

[22] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI 07: Programming Language Design and Implementation*, pages 446–455, 2007.

[23] Amir Pnueli. The temporal logic of programs. In *FOCS 77: Foundations of Computer Science*, pages 46–57, 1977.

[24] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Fifth International Symposium on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1981.

[25] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS 86: Logic in Computer Science*, pages 322–331. IEEE Computer Society Press, 1986.

[26] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *ASE 00: Automated Software Engineering*, pages 3–12, 2000.

[27] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI 94: Operating Systems Design and Implementation*, pages 1–11, 1994.

[28] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.