Scalable Multi-Core Model Checking Fairness Enhanced Systems

Yang Liu, Jun Sun and Jin Song Dong

School of Computing,
National University of Singapore
{liuyang,sunj,dongjs}@comp.nus.edu.sg

Abstract. Rapid development in hardware industry has brought the prevalence of multi-core systems with shared-memory, which enabled the speedup of various tasks by using parallel algorithms. The Linear Temporal Logic (LTL) model checking problem is one of the difficult problems to be parallelized or scaled up to multi-core. In this work, we propose an on-the-fly parallel model checking algorithm based on the Tarjan's strongly connected components (SCC) detection algorithm. The approach can be applied to general LTL model checking or with different fairness assumptions. Further, it is orthogonal to state space reduction techniques like partial order reduction. We enhance our PAT model checker with the technique and show its usability via the automated verification of several real-life systems. Experimental results show that our approach is scalable, especially when a system search space contains many SCCs.

1 Introduction

In recent years, the growth of computer CPU speed is slowly being replaced by the growth of number of CPUs (or CPU-cores) in the industry. To make full usage of the CPU cores naturally raises interest in applying parallelism in various problems. In this work, we focus on the parallelism of model checking fairness enhanced systems, which emits two challenges stated as follows.

Firstly, efficient parallel solution of many problems may result in dramatically different approaches from those to solve the same problems sequentially. Classical examples are list rankings, connected components, depth-first search in planar graphs etc. In the area of Linear Temporal Logic (LTL) model checking, the two best known enumerative sequential algorithms based on fair-cycle detection are the Nested Depth First Search (NDFS) algorithm [10, 18] (e.g., implemented in the model checker SPIN [17]) and SCC-based algorithms [32, 31] based on Tarjan's algorithm for strongly connected components (SCCs) detection [33]. However, both algorithms strongly rely on inherently sequential depth-first search of post-ordering of vertices (P-complete computation [28]). Hence it is difficult to adapt them to parallel architectures. Consequently, different techniques and algorithms are needed. Several existing parallel versions of LTL model checking algorithms are ineffective or hard to scale up. For example, SCC based parallel algorithms [8, 12, 9, 6, 3] gives quadratic or cubic order of the

search space. Multi-core SPIN [16] is only applicable to two cores for liveness properties. Note that unlike LTL model checking, deadlock-free or reachability analysis is a verification problem with efficient parallel solution. The reason is that the exploration of the state space can be partitioned using breadth-first search [16]. In this work, we will focus on the liveness properties.

Second, fairness, which is concerned with a fair resolution of non-determinism, is often important but expensive to be combined with model checking algorithms. Fairness is an abstraction of the fair scheduler in a multi-threaded programming environment or the relative speed of the processors in distributed systems. Without fairness, verification of liveness properties often produces unrealistic loops during which one process or event is infinitely ignored by the scheduler or one processor is infinitely faster than others. It is important to rule out those counterexamples and utilize the computational resource to identify the real bugs. However, systematically ruling out counterexamples due to lack of fairness is highly non-trivial. It requires flexible specification of fairness as well as efficient verification under fairness. Fairness and model checking with fairness have attracted much theoretical interests for decades [14, 24, 21]. Their practical implications in system/software design and verification have been discussed extensively. In our previous works [32, 31], we present a unified on-the-fly model checking algorithm which handles a variety of fairness including process-level weak/strong fairness, event-level weak/strong fairness, strong global fairness, etc. However, none of these works paid attention to parallel verification.

Contributions In this work, we propose an algorithm with the capacity of parallel verification of systems with various fairness constraints in the multi-core architecture with share-memory.

SCC-based LTL model checking algorithms conduct a depth first search starting from the root node, and check whether the SCC in the subtree is fair whenever a SCC is identified. Previous parallel algorithms focus on partition of the graph based on special properties of the nodes inside the SCCs, which requires multiple traverses of the whole search graph. These approaches are not practical for large systems, especially when there is a counterexample. Based on our previous work, we propose an on-the-fly parallel algorithm based on an improved version of Tarjan's algorithm. In our approach, a main thread performs the DFS searching of Tarjan's algorithm. Whenever a SCC is detected, a new worker thread is forked to process the found SCC. SCC processing contains both fair loop detection and fairness constraints satisfaction checking (if there is fairness assumption in the system), hence a fair amount of workload is divided to the worker threads to achieve load balancing. When a counterexample is identified in any worker thread, it will inform the main thread to stop the DFS and all other live worker threads. This makes our approach on-the-fly, i.e., without generating the entire search space. We have proved the correctness of our approach in the multi-core architecture with shared memory.

Effective reduction techniques are the keys to resolve the infamous "state explosion" problem in model checking, such as partial order reduction (to reduce the search space by exploring independence of system transitions), symmetric

reduction (to handle large or even unbounded number of similar processes). We show that all these reductions are compatible with our algorithm, if these reductions are applicable (see Section 4.3 for details).

Our engineering effort realizes this technique in our home-grown PAT model checker (available at http://pat.comp.nus.edu.sg). We show its usability via automated verification of several real-life systems. The experiments show that our technique offers a scalable verification support for multi-core model checking.

Section Organization The rest of the paper is structured as follows. Section 2 introduces our computational model, together with a family of different fairness notions. Section 3 presents a sequential fairness model checking algorithm based on SCC detection. Section 4 describes our proposed parallel algorithm in the shared-memory platform. Section 5 shows some experimental results to demonstrate the effectiveness of parallel algorithm. Section 6 discusses related work and Section 7 concludes.

2 Background

In this work, system models are described in the setting of Labeled Transition Systems (LTS). All the algorithms proposed in this paper are applicable to the models that can be interpreted as LTSs implicitly by defining a complete set of operational semantics. For example, PAT accepts modeling languages like Communicating Sequential Processes# (CSP#) [29], Web Service modeling language, real-time system modeling language. This section gives the LTS semantics and defines different fairness constraints based on it.

Let e be an event (in process algebra, e.g., CSP), which could be either an abstract event (e.g., a synchronization barrier if shared by multiple processes) or a data operation (e.g., a sequential program). Let Σ be the set of all events in the model.

Definition 1 (LTS). *A Labeled Transition System $\mathcal{L}$ is a 3-tuple $(S, init, \rightarrow)$ where S is a set of system configurations/states, $init \in S$ is the initial system configuration and $\rightarrow \subseteq S \times \Sigma \times S$ is a labeled transition relation.*

In this work, we focus on infinite system executions explained as follows. Finite behaviors are extended to infinite ones by appending infinite idling events at the rear. Given two states s and s' in S, we write $s \xrightarrow{e} s'$ to denote a transition from s to s' with event e. Given a LTS $\mathcal{L} = (S, init, \rightarrow)$, an execution $E = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ is an infinite sequence of alternating states and events, where $s_0 = init$ and for all $i \geq 0$ such that $s_i \xrightarrow{e_i} s_{i+1}$. Given a LTL property ϕ, $\mathcal{L}$ satisfies ϕ if and only if every execution of $\mathcal{L}$ satisfies ϕ.

Without fairness constraints, a system may behave freely as long as it starts with an initial state and conforms to the transition relation. A fairness constraint restricts the set of system behaviors to only those fair ones. Given a LTL property ϕ, verification under fairness means verifying whether all fair executions of the system satisfy ϕ. In the following, we briefly review a variety of different fairness

constraints. The following notions are used to define fairness. $enabledEvt(s)$ is the set of enabled events at state s, i.e., e is in $enabledEvt(s)$ if and only if there exist $s' \in S$ such that $s \xrightarrow{e} s'$. If the system is constituted by multiple processes running in parallel, we write $enabledPro(s)$ to be the set of enabled processes, which may make a move given the system state s. Given a transition $s \xrightarrow{e} s'$, we write $engagedPro(s, e, s')$ to be the set of participating processes, which have made some progress during the transition. Notice that if e is synchronized by multiple processes, the set contains all the participating processes. We write $engagedEvt(s, e, s')$ to denote $\{e\}$. In the following, we use $E = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ to denote an execution.

Weak fairness [24, 25] There are two different levels of weak fairness, i.e. event-level weak fairness (EWF) or process-level weak fairness (PWF). E satisfies event-level weak fairness, if and only if for every action e, if e eventually becomes enabled forever in E, then $e_i = e$ for infinitely many i, i.e., $\Diamond\Box$ *e is enabled* $\Rightarrow$ $\Box\Diamond$ *e is engaged*. Intuitively, event-level weak fairness states that if an event becomes enabled forever after some steps, then it must be engaged infinitely often. E satisfies process-level weak fairness, if and only if for every process p, if p eventually becomes enabled forever in E, then $p \in engagedProc(s_i, e_i, s_{i+1})$ for infinitely many i, which equals to $\Diamond\Box$ *p is enabled* $\Rightarrow$ $\Box\Diamond$ *p is engaged* in LTL. Intuitively, process-level weak fairness states that if a process becomes enabled forever after some steps, then it must be engaged infinitely often. From another point of view, process-level weak fairness guarantees that each process is only finitely faster than the others. Weak fairness is equivalent to justice conditions [25]. An alternative formulation of weak fairness is that every computation should contain infinitely many particular states (e.g. states where an event or a process is disabled or has just engaged).

Strong fairness [23, 11, 27] Strong fairness is particularly useful in the analysis of systems that use semaphores, synchronous communication, and other special coordination primitives. Likewise, there are two levels of strong fairness. E satisfies event-level strong fairness (ESF) if and only if, for every event e, if e is infinitely often enabled, $e = e_i$ for infinitely many i, which equals to $\Box\Diamond$ *e is enabled* $\Rightarrow$ $\Box\Diamond$ *e is engaged* in LTL. It states that if an event is infinitely often enabled, it must be infinitely often engaged. E satisfies process-level strong fairness (PSF) if and only if, for every process p, if p is infinitely often enabled, then $p \in engagedProc(s_i, e_i, s_{i+1})$ for infinitely many i, which equals to $\Box\Diamond$ *p is enabled* $\Rightarrow$ $\Box\Diamond$ *p is engaged* in LTL. Process-level strong fairness means that if a process is repeatedly enabled, it must eventually make some progress. Verification under (event-level/ process-level) strong fairness (or compassion condition) has been discussed previously [13, 15, 20, 26, 32, 31].

Strong global fairness [11] E satisfies strong global fairness (SGF) if and only if, for every s, e, s' such that $s \xrightarrow{e} s'$, if $s = s_i$ for infinite many i, $s_i = s$ and $e_i = e$ and $s_{i+1} = s'$ for infinitely many i. Intuitively, it states that if a *step* (from s to s' by engaging in event e) can be taken infinitely often, then it

must actually be taken infinitely often. Different from the previous notions of fairness, strong global fairness concerns about both events and states, instead of events only. It can be shown by a simple argument that strong global fairness is stronger than event-level strong fairness. Because it concerns about both events and states, it is 'event-level' and 'process-level'. Strong global fairness requires that an infinitely enabled event must be taken infinitely often in *all* contexts, whereas event-level strong fairness only requires the enabled event to be taken in *one* context. Many population protocols reply on strong global fairness, e.g., protocols presented in [1, 11].

3 Sequential Model Checking under Fairness

Given a LTS $\mathcal{L}$ and a LTL formula ϕ, model checking is about searching for an execution of $\mathcal{L}$ which fails ϕ. In automata-based model checking, the negation of ϕ is translated to an equivalent Büchi automaton $\mathcal{B}$, which is then composed with the LTS representing the system model. Model checking under fairness is to search for an infinite execution which is accepting to the Büchi automaton and at the same time satisfies the fairness constraints. Equivalently, it is to search a loop or a Strongly Connected Components (SCC) in the state graph such that the infinite execution traversing through every state/edges of the loop or SCC satisfies the fairness constraints.

SCC-based verification algorithms rely on the SCC detection, most of which are based on Tarjan's algorithm for identifying SCCs [33]. Figure 1 presents a sequential unified algorithm for automata-based model checking of LTL under fairness [31]. The algorithm works by searching on-the-fly for fair strongly connected subgraphs, which may constitute counterexamples. The basic idea is to identify one SCC at a time and then check whether it is fair or not. If it is, the search is over. Otherwise, the SCC may be partitioned into several smaller strongly connected subgraphs, which are then checked recursively one by one.

We briefly explain how the algorithm works. Interested readers should refer to [31] for details. Assumes that *States* is the set of states and *Transitions* is the set of transitions[1]. At the top level is a while-loop, which stops only if all states have been visited. At line 2, Tarjan's algorithm is used to identify a SCC [13]. If the found *scc* is fair, a counterexample is generated (at line 5) and the algorithm returns false. Without fairness assumptions, a SCC is fair if and only if it is accepting to the Büchi automaton (i.e. Büchi fair). The complexity of checking whether *scc* is fair or not under fairness assumption is linear in the size of *scc*. For instance, under weak fairness, we must first identify the set of processes/events that are always enabled and compare the set with the set of processes/events that make progress.

If *scc* is not fair, a procedure *prune* is used to prune *bad states* from *scc* (at line 8). Bad states are the reasons why *scc* is not fair. The intuition behind the pruning is that there may be a fair strongly connected subgraph in the remaining

[1] both of which may be constructed on-the-fly instead of known before-hand.

```
procedure mc(States, Transitions)
1.  while there are un-visited states
2.        let scc := tarjan(States, Transitions);
3.        mark states in scc as visited;
4.        if isFair(scc) = true then                          – *
5.              generate a counterexample;                    – *
6.              return false;                                 – *
7.        else                                                – *
8.              scc = prune(scc);                             – *
9.              if mc(scc, Transitions) = false then          – *
10.                   return false;                           – *
11.             endif                                         – *
12.       endif                                               – *
13. endwhile
14. return true;
```

Fig. 1. Algorithm for sequential model checking under fairness [31]

states after eliminating the bad states. By simply modifying *isFair* and *prune* method, the algorithm can be used to handle different fairness. For instance,the following defines the functions for event-level strong fairness [31].

$$isFair(scc) = true \text{ if and only if } \forall\, s : scc\ enabledEvt(s) \subseteq engagedEvt(scc)$$

where $engagedEvt(scc) = \{a \mid \exists\, s, s' : scc\ s \xrightarrow{a} s'\}$ is the set of events labeling a transition between two states in *scc*, i.e. the set of events that can be engaged if an execution visits only states in *scc*. Intuitively, *scc* satisfies event-level strong fairness if and only if all enabled events are engaged in the SCC.

$$prune(scc) = \{s : scc \mid enabledEvt(s) \subseteq engagedEvt(scc)\}$$

In this setting, a state is bad if it enables an event which is not engaged in the SCC. It is clear that if the SCC contains a fair strongly connected subgraph, no state constituting the subgraph is pruned.

At line 9, a recursive call is made to check whether there is a fair strongly connected subgraph within the remaining states. The call terminates in two ways. One is that a fair subgraph is found (at line 6) and the other is that all states in *scc* are pruned (at line 14).

Example 1. Assume that the automaton shown in Figure 2 is the product of a LTS and a Büchi automaton. Further assume that state 2 is an accepting state, i.e. any traces which visits state 2 infinitely often is accepting the Büchi automaton. There are two SCCs, namely *scc*1 which is composed of state 1 only and *scc*2 which is composed of state 0, 2 and 3. State 0 is a bad state in *scc*2 under event-level strong fairness since $a \in enabledEvt(state\ 0)$ whereas

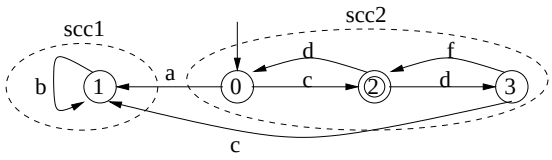


Fig. 2. ESF Model Checking Example

$a \notin engagedEvt(scc2)$. Notice that state 3 is not a bad state. As a result, state 0 is pruned. Next, in the recursive call, the SCC composed of state 2 and 3 are identified. However, state 3 becomes a bad state now because event c is now enabled but not engaged. State 3 is pruned then. Lastly, state 2 is pruned. Because $scc1$ does not contain an accepting state, it fails all *isFair* test. As a result, no counterexample is found. □

After a SCC has been fully examined (i.e., all pruned) at line 12, the algorithm repeats from line 2 to check the next SCC. We remark that the algorithm is a natural candidate for exploring multi-core parallelism. Firstly, examining weather a SCC is fair or whether it contains fair strongly connected subgraph is time consuming, and hence checking multiple SCCs in parallel is likely to generate significant saving. SCC fairness checking is linear in the number of transitions connecting states of the SCC. Checking whether a SCC contains a fair strongly connected subgraph is expensive. In the worst case, only one state is pruned each time and therefore the complexity is bounded by the number of transitions times the number of states. Secondly, different SCCs naturally exclude each other and therefore checking them in parallel will not cause significant computational or communication overhead.

4 Parallel Model Checking In Shared-Memory Platform

In this section, we present a parallel approach at the challenges of shared-memory architecture and its specific characteristics. We will detail the algorithm design, its complexity and correctness.

4.1 Shared-Memory Platform

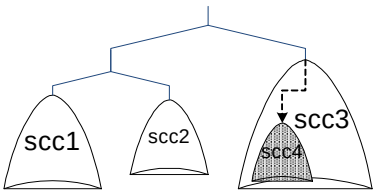
We work with a model based on threads that share all memory, although they have separate stacks in their shared address space and a special thread-local storage to store thread-private data. Our working environment is .NET framework (version 2.0) in Microsoft Windows platform, with its implementation of threads as lightweight processes. Switching contexts among different threads is cheaper than switching contexts among full-featured processes with separate address spaces, so using threads in the system incurs only a minor penalty.

Critical Sections, Locking and Lock Contention In a shared-memory setting, access to memory, that may be used for writing by more than a single thread, has to be controlled through the use of mutual exclusion, otherwise, race conditions will occur. This is generally achieved through use of a "mutual exclusion device", so-called mutex. A thread wishing to enter a critical section has to lock[2] the associated mutex, which may block the calling thread if the mutex is locked already by some other thread. An effect called resource or lock contention is associated with this behavior. This occurs, when two or more threads happen to need to enter the same critical section (and therefore lock the same mutex), at the same time. If critical sections are long or they are entered very often, contention starts to cause observable performance degradation, as more and more time is spent waiting for mutexes.

Memory Management and Thread Communication Microsoft .NET common language runtime requires that all resources be allocated from the managed heap. Objects are automatically freed when they are no longer needed by the application. The communication between threads can be achieved simply by object reference passing.

4.2 Parallel Fairness Model Checking Algorithm

The SCC-based verification algorithm presented in the previous section is recursive and employs a sequential DFS search, which exhibits some challenges in parallelism.



The sequential algorithm in Figure 1 can be illustrated in the figure above. When a SCC is detected, it will be analyzed and pruned until empty or there is a counterexample detected (*scc*4 in above graph). Taking a close look at the algorithm, we observe that there are four actions applied in each detected SCC: (1) fairness testing (line 4), (2) bad states pruning (line 8), (3) counterexample generation (line 5), (4) recursive sub-SCC detection (line 9). The first three actions are local to the detected SCC. Although the recursive sub-SCC detection is complicated, we can create a local copy of the Tarjan algorithm to search for "SCC" in the pruned states. In this way, each SCC can be processed independent. Therefore, we can put the workload of SCC analysis into separate

[2] In .NET framework, keyword **lock** is used to achieve this effect.

```
stopped = false;
procedure run(threadPool, States, Transitions)
1.  visited = ∅;
2.  while there are states in States but not in visited
3.        if stopped then {return; }
4.        let scc = tarjan(States, Transitions);
5.        visited = visited ∪ scc;
6.        if forking conditions then
7.              threadPool.forkWorkerThread(scc, Transitions);
8.        else
9.              process scc locally
10.       endif
11. endwhile
12. return;
```

Fig. 3. Tarjan Thread Implementation

threads to achieve concurrency. Inspired by these observations, we present a SCC-based parallel model checking algorithm with four parts: *Tarjan thread*, *SCC worker thread*, *SCC worker thread pool* and *parallel model checker*. The detailed algorithms are illustrated as follows.

Tarjan thread Figure 3 presents the implementation of *Tarjan thread*, which identifies all SCCs using Tarjan's algorithm. Tarjan thread has one public variable *stopped* and the thread starting procedure *run*. *stopped* is a control variable to stop this thread (line 3) as soon as a worker thread reports a counterexample. When *Tarjan thread* starts, the *run* procedure will perform a DFS to detect all SCCs in the search space using Tarjan's algorithm. This process is similar to *mc* procedure in Figure 1. When a SCC *scc* is detected at line 4, if the forking conditions at line 6 are satisfied, then a new SCC worker thread will be forked and added in to the worker thread pool (line 7). Otherwise *scc* will processed locally in the *Tarjan thread* (line 9). This local process is the same as the *SCC worker thread* (which will be explained later), which stops this thread if a counterexample is found. Forking conditions can be that the size of *scc* is bigger than some threshold or the thread pool is full. We add this checking to achieve better efficiency and workload balance. If the size of *scc* is small (e.g., only few nodes), the overhead of creating a thread is much bigger than processing it locally. If the thread pool is full, processing the found *scc* locally is probably more efficient than creating a long waiting queue in the thread pool.

SCC worker thread *SCC worker thread* works on a detected SCC to report whether the SCC contains a counterexample or not within the given SCC states and transitions. It basically resembles the code from line 4 to 12 (highlighted using _*) in Figure 1. If the detected SCC is not fair, it will prune the states

```
threadQueue = empty queue;
jobFinished = false;
procedure forkWorkerThread(States, Transitions)
1.  lock(threadQueue);
2.      if(!jobFinished)
3.          let wt = new workerThread(States, Transitions);
4.          register wt.termination to threadTermination procedure
5.          threadQueue.enqueue(wt);
6.      endif
7.  unlock(threadQueue);

procedure threadTermination(thread)
8.  lock(threadQueue);
9.      if thread produces counterexample ∧ !jobFinished then
10.         terminate all other threads
11.         terminate tarjan thread
12.         jobFinished = true;
13.     endif
14.     threadPool.remove(thread)
15. unlock(threadQueue);

procedure allThreadsJoin()
16. while(has running threads)
17.     busy wait
18. endwhile
```

Fig. 4. Thread Pool Implementation

according to the given fairness type. Otherwise it will terminate and return false. If the pruned *scc* has fewer states, a local copy of the Tarjan's algorithm will continue the searching. Upon the termination of *SCC worker thread*, a notification will send to the thread pool to notify the result.

SCC worker thread pool The implementation of the *SCC worker thread pool* is presented in Figure 4. The thread pool has a working queue *threadQueue*[3] to store all active worker threads. Private variable *jobFinished* indicates whether a counterexample has been found or not. Procedure *forkWorkerThread* creates a new worker thread (line 3) and puts it into the working queue (line 5), if the counterexample is not found (line 2). A lock is used on *threadQueue* (at line 1 and 7) to prevent *Tarjan thread* working too fast to add two or more

[3] In our implementation, *threadQueue* is realized by System.Threading.ThreadPool object in .NET Framework. The thread scheduling is managed by the thread pool automatically.

```
procedure pmc(States, Transitions)
1. initialize worker thread pool threadPool
2. let tarjan = tarjanThread.run(threadPool, States, Transitions);
3. tarjan.join();
4. threadPool.allThreadsJoin();
5. if counterexample is found then
6.       return false;
7. return true;
```

Fig. 5. Parallel Model Checker Implementation

threads at same time. This is possible because during the process of forking the first thread, *Tarjan thread* may find another SCC and want to fork a new thread. At line 4, we register the termination event of the *worker thread* to procedure *threadTermination*, which means upon the termination of the worker thread, the thread pool will be notified and procedure *threadTermination* will be triggered. When procedure *threadTermination* is triggered, if the termination thread has located a counterexample and no one does it before (line 9), thread pool will terminate[4] all other active threads (line 10) and *Tarjan thread* (by setting *stopped* flag to true) (line 11). Flag *jobFinished* is set to true at line 12, hence new threads shall not be forked anymore. !*jobFinished* checking in line 9 is necessary to prevent terminating same threads twice. In the end, the termination thread is removed from thread pool in line 12. During this process *threadQueue* is locked to prevent data race. Procedure *allThreadsJoin* does busy-waiting until all threads terminate.

Parallel model checker Lastly, *parallel model checker* is shown in Figure 5. It conducts the verification by creating the *Tarjan thread* and thread pool. Once *Tarjan thread* starts, it will wait for *Tarjan thread* to join (i.e., successfully terminate) (line 3). The termination can be that all states are explored, or a counterexample is found locally, or *stopped* flag is set to false. Afterwards, it will wait for thread pool to terminate (line 4). The procedure will return false if any counterexample is found in *tarjan thread* or any worker thread.

4.3 Complexity and Soundness

In this section, we discuss the complexity of the parallel model checking algorithm and prove its soundness.

For the sequential version of the algorithm, the time complexity for verification under no fairness, event-level or process-level weak fairness or strong global

[4] Thread termination can be achieved by thread killing or asking the thread to voluntarily give up. The second way is safer and adopted in our approach. One example is the stopped flag in *Tarjan thread*.

fairness are similar, i.e., all linear in the number of system transitions. All states in one SCC are discarded at once in all cases and, therefore, no recursive call is necessary. Furthermore, the *prune* function is linear in the number of transitions of a SCC. In comparison, SPIN's model checking algorithm under process-level weak fairness increases the run-time expense of a verification run by a factor that is linear in the number of running processes. Verification under event-level or process-level strong fairness is in general expensive. In the worst case (i.e., the whole system is strongly connected and only one state is pruned every time), the *prune* method may be invoked at most $\#S$ times, where $\#S$ is the number of system states. Thus, the time complexity is bounded by $\#S \times \#T$ where $\#T$ is the number of transitions. In practice, however, if the property is false, a counterexample is usually identified quickly, because our algorithm constructs and checks SCCs on-the-fly. Even if the property is true, our experience suggests that the worst case scenario is rare in practice.

For the parallel version of the algorithm, the time and space complexity is exactly same as the sequential version. This is not surprising because the parallel algorithm simply splits SCC analysis into worker threads. The parallel algorithm is designed for a shared memory framework, the SCCs and their transitions are shared between *Tarjan thread* and worker threads. There is no communication overhead. If to migrate this approach into distributed systems, we may consider to pass SCC only and let the worker threads to build the transitions locally to avoid the communication overhead. The is because the number of transitions of a SCC is often much larger than the number of vertices.

If the verification result is true, the number of states and transitions visited in the parallel and sequential version are same. If there is a counterexample, the parallel version may visit more states depending on when the counterexample is identified. If a counterexample is present in the first few SCCs encountered during the search, then the sequential version may find one quickly, while the parallel version may have forked multiple threads to search in more SCCs. Hence parallel version visits more states and transitions. On the other hand, if a counterexample is present only in the last few SCCs, the parallel version can be faster than the sequential version if the counterexample is identified quickly in one worker thread, which then terminates all other SCC checking. This is evidenced by the experiment results presented in Section 5. Notice that when there are more than one counterexamples in the system, it is possible that the parallel verification may produce different counterexample at different runs.

Regarding the soundness, the following theorem establishes correctness of the sequential algorithm. The proof for different fairness can be found in our technical report [30].

Theorem 1. *Let $\mathcal{L}$ be an LTS. Let ϕ be a property. Let F be a fairness type (i.e., EWF, PWF, ESF, PSF or SGF). $\mathcal{L} \vDash_F \phi$ if and only if the algorithm mc returns true.*

The following theorem states the correctness of the parallel algorithm *pmc*. We argue the total correctness of the parallel algorithm by showing it is terminating and equivalent to the sequential *mc* algorithm.

Theorem 2. *Let $\mathcal{L}$ be an LTS. Let ϕ be a property. Let F be a fairness type (i.e., EWF, PWF, ESF, PSF or SGF). $\mathcal{L} \models_F \phi$ if and only if the algorithm pmc returns true.*

Proof: Firstly, we show that the *pmc* algorithm is terminating. By the assumption, we know that the number of states is finite, so is the number of the SCCs. In *Tarjan thread*, the number of visited states and the pruned states are monotonically increasing, hence the Tarjan thread is terminating. Worker threads are terminating since they are working on the detected SCC and the number of pruned states are monotonically increasing. Since the number of SCC is finite, worker thread pool is terminating. Therefore *pmc* is terminating.

Secondly, we show that *pmc* returns the same result as *mc*. The key of this proof is to prove that each SCC analysis is independent of each other. If this true, then checking the SCCs in parallel is same as checking them sequentially. We have listed the four actions performed in the SCCs in Section 4.1, which can be applied independently.

Lastly, the correctness of data sharing and race condition prevention by using locks have been discussed in Section 4.2. We skip it here. □

Following the above theorem, we conclude that the sequential algorithm and the parallel algorithm are equivalent in terms of correctness. Therefore as long as the reduction is compatible with sequential algorithm, then it is compatible with the parallel algorithm. For example, our previous work [32] shows that partial order reduction is possible by employing fairness annotations on individual events, which means this technique can also be used with our parallel algorithm. We remark that *pmc* is orthogonal to state reduction techniques like partial order reduction, symmetry reduction or data abstraction. Intuitively, the parallel algorithm would perform better since it may utilize more CPU power. Nonetheless, thread forking/terminating or communication between threads can be costly. We present detailed analysis using real-world examples as well as hand craft examples in the next section.

5 Experimental Results

Process Analysis Toolkit (hereafter PAT) is designed for systematic validation of distributed/concurrent systems using state-of-art model checking techniques. Its main functionalities include simulation, explicit on-the-fly model checking, and verification under fairness. The model checker combines complementary model checking techniques for system verification. In the following, we show its performance on both benchmark systems as well as recently developed population protocols, which require fairness for correctness. All the models (with configurable parameters) are embedded in the PAT package and available online at our web site http://pat.comp.nus.edu.sg.

Regarding the threads scheduling, there are two approaches. The first approach is to manually assign a newly created thread to a free CPU-core. If all CPU-cores are used, the new thread is pushed into the working queue and

Model	Size	Avg SCC/	SCC	EWF			ESF			SGF		
		#SCC	Ratio	Result	mc	pmc	Result	mc	pmc	Result	mc	pmc
DP	5	67/13	0.36	No	0.08	0.08	Yes	0.22	0.20	Yes	0.19	0.19
DP	6	178/21	0.38	No	0.13	0.13	Yes	0.97	0.84	Yes	0.86	0.78
DP	7	486/31	0.4	No	0.38	0.37	Yes	4.62	3.39	Yes	4.42	3.38
DP	8	1368/43	0.41	No	1.41	1.33	Yes	29.28	19.49	Yes	32.90	22.14
LE_C	3	22/3	0.33	Yes	0.11	0.11	Yes	0.11	0.11	Yes	0.10	0.10
LE_C	4	24/15	0.47	Yes	0.53	0.47	Yes	0.52	0.47	Yes	0.46	0.45
LE_C	5	34/43	0.58	Yes	4.04	3.66	Yes	4.03	3.65	Yes	3.66	3.49
LE_C	6	48/103	0.64	Yes	23.12	21.39	Yes	23.05	21.54	Yes	21.91	20.14
LE_C	7	66/227	0.68	Yes	128.8	124.4	Yes	129.5	124.3	Yes	133.9	127.2
LE_C	8	86/479	0.71	Yes	604.3	600.5	Yes	615.8	606.6	Yes	721.9	684.4
LE_R	3	9/268	0.36	No	0.11	0.11	No	0.12	0.12	Yes	1.40	1.27
LE_R	4	9/2652	0.4	No	0.11	0.28	No	0.59	0.60	Yes	21.65	15.73
LE_R	5	9/25274	0.42	No	0.71	0.72	No	2.22	2.19	Yes	587.0	456.4
TC_R	4	16/1	0.01	No	0.06	0.07	No	0.07	0.06	Yes	0.11	0.12
TC_R	5	60/1	0.01	No	0.08	0.08	No	0.08	0.08	Yes	0.45	0.48
TC_R	6	84/2	0.01	No	0.11	0.11	No	0.11	0.11	Yes	2.20	2.38
TC_R	7	210/2	0.01	No	0.14	0.14	No	0.15	0.16	Yes	11.28	12.31
TC_R	8	330/3	0.01	No	0.19	0.20	No	0.25	0.23	Yes	69.55	72.98
TC_R	9	756/3	0.01	No	0.27	0.31	No	0.36	0.37	Yes	494.4	572.7

Table 1. Experiment results on a PC running Windows XP with 2.83 GHz quad-core Intel Q9550 CPU and 2 GB memory

wait.The second approach is to make each thread as operating system thread[5], and let the OS CPU scheduler to do the scheduling. We compared the two approaches, it shows that when the size of the SCCs is big, the two approaches have same results. When the number of SCCs is big, the second approach is more efficient. We applied second approach in our experiments.

In our experiments below, *Size* denotes the number processes in the models. Besides the execution time of the sequential algorithm (*mc*) and parallel algorithm (*pmc*), we present additional measurements which reflect the amount of workload *pmc* can put in parallel if the verification result is true[6]. One is the average size of nontrivial SCCs (denoted as *Avg SCC Size*) and the number of SCC (denoted as *#SCC*). A SCC is trivial if and only if it has only one state. Intuitively, the parallel algorithm gains more saving with larger and more SCCs. The other is the ratio of the number of states of all (non-trivial) SCCs and the whole state space (denoted as *SCC Ratio*). Intuitively, a higher *SCC Ratio* shall lead to more saving. The forking condition is that the SCC must have at least 100 states. '-' means out of memory. The unit of time measurement is second.

Table 1 summarizes the verification statistics on classic dinning philosophers problem (*DP*), and recently developed population protocols. The population

[5] In our implementation, we use System.Threading.ThreadPool object in .NET framework 2.0 to create system threads in Microsoft Windows system.

[6] When the property is false, *SCC Ratio* can be different for different runs.

Model	Size	Avg SCC/ #SCC	SCC Ratio	EWF			ESF			SGF		
				Result	mc	pmc	Result	mc	pmc	Result	mc	pmc
$PAR1$	5	10001/5	0.2	No	1.75	2.11	Yes	22.50	12.03	Yes	11.33	6.97
$PAR1$	6	10001/6	0.2	No	1.74	2.07	Yes	27.10	14.81	Yes	13.59	8.13
$PAR1$	7	10001/7	0.2	No	1.71	2.29	Yes	31.22	16.66	Yes	15.89	9.14
$PAR1$	8	10001/8	0.2	No	1.71	2.16	Yes	36.08	18.04	Yes	18.09	10.60
$PAR1$	9	10001/9	0.2	No	1.71	2.15	Yes	40.59	20.85	Yes	20.40	11.90
$PAR1$	10	10001/10	0.2	No	1.73	2.15	Yes	45.29	22.63	Yes	22.81	13.07
$PAR2$	4	20000/5	0.5	No	5.46	7.12	NA	-	-	Yes	8.87	5.52
$PAR2$	5	20000/6	0.5	No	6.05	9.53	NA	-	-	Yes	18.32	8.64
$PAR2$	6	20000/7	0.5	No	6.39	10.51	NA	-	-	Yes	21.37	9.32
$PAR2$	7	20000/8	0.5	No	6.90	11.41	NA	-	-	Yes	24.50	9.69
$PAR2$	8	20000/9	0.5	No	7.77	11.65	NA	-	-	Yes	27.86	11.82
$PAR2$	9	20000/10	0.5	No	8.06	12.76	NA	-	-	Yes	30.89	13.68
$PAR3$	7	2000/8	1	No	0.29	0.20	Yes	411.5	117.6	Yes	0.41	0.28
$PAR3$	8	2000/9	1	No	0.21	0.24	Yes	463.1	135.7	Yes	0.45	0.29
$PAR3$	9	2000/10	1	No	0.25	0.23	Yes	515.7	155.8	Yes	0.49	0.31

Table 2. Experiment results on a PC running Windows XP with 2.83 GHz quad-core Intel Q9550 CPU and 2 GB memory

protocols include leader election for complete networks (LE_C) [11], for network rings (LE_R) [11] and token circulation for network rings (TC_R) [1]. We modify the DP model so that it is deadlock-free (i.e., by letting one of the philosophers to pick up the forks in a different order). The property is that a philosopher never starves to death, i.e., $\Box\Diamond eat.0$, where $eat.0$ is the event of 0-th philosopher eating. The property for the leader election protocols is that eventually always there is one and only one leader in the network, i.e., $\Diamond\Box oneLeader$. Correctness of all these algorithms relies on different notions of fairness.

In Table 1, we can see that when the verification result is false, either *pmc* or *mc* can be faster, which is expected. When the verification result is true, *pmc* is faster in most of the cases, except in the case of model checking the TC_R example under strong global fairness. In this particular example, *SCC ratio* is very low (0.01), which means that there are many trivial SCCs. Furthermore, there are only few non-trivial SCCs. As a result, there is little work that can be separated out for the worker threads to speed up the model checking, and the communication overhead makes *pmc* slower. On the other hand, the *pmc* slowdown in this case is only several percents of *mc*, which shows that the communication overhead in *pmc* is low.

Table 2 summarizes the verification statistics on some hand craft examples to show the potential effectiveness of the parallel algorithm. We create three models ($PAR1$, $PAR2$ and $PAR3$) such that the their state space contains several SCCs, each of which has big number of states. As a result, worker threads can be dispatched with substantial workload. Correctness of all these algorithms requires ESF and SGF.

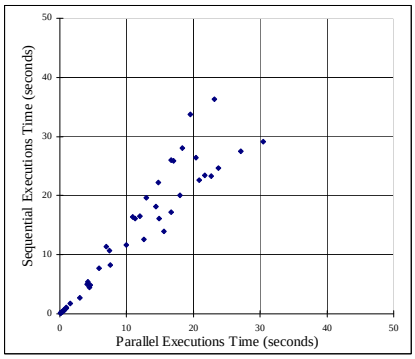


Fig. 6. Results on Intel Core2 6600 CPU

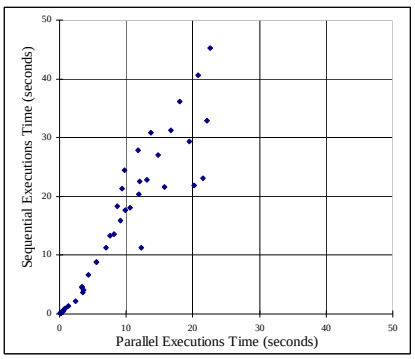


Fig. 7. Results on Intel Q9550 CPU

In Table 2, we can see that *pmc* is working well in $PAR1$ example, where the average SCC size is big and the SCC ratio is not very low. The performance is even better (60% speedup) when the SCC ratio increases to 0.5 in $PAR2$ example. The $PAR3$ example almost produces the ideal case (72% speedup) such that the four cores are fully loaded. Since there are more SCCs than cores, further speedup could be achieved if there were more cores. ESF case in $PAR2$ gives a worst case mentioned in Section 4.3 for strong fairness checking, hence it ends up with out of memory exception.

The experiment results in Table 1 and 2 confirm that the speedup of parallel verification relies on the size and the number of non-trivial SCCs. Each SCC has four analysis actions as described in Section 4.1. If the size of SCCs is big and/or the number of SCCs is more than the number of cores, each worker thread will make full use of the available CPU-cores. Overall, *pmc* performs better than *mc* for big average SCC size and high SCC ratio.

To study the scalability of our approach with different number of CPU cores, we conduct the same experiments (model checking examples in Table 1 and $PARA1$ example under strong global fairness)[7] on a dual-core CPU (Figure 6) and a quad-core CPU[8] (Figure 7). The coordinate of each point (x, y) in the graphs represents *mc* execution time and *pmc* execution time of a model correspondingly. From the figures we can see that, points in Figure 6 are scattered between line $y = x$ and $y = 2x$, while points in Figure 7 are scattered between line $y = 2x$ and $y = 3x$. The average speedup of the parallel algorithm is 22.9% for quad-core CPU and 11.2% for dual-core CPU. This suggests that our approach is scalable for more CPU cores in general.

Besides PAT, there are a number of model checkers which are designed for similar application domains. It is, however, not easy to compare PAT with

[7] $PARA2$ and $PARA3$ have high average SCC size and SCC ratio which is rare in real systems, so we exclude them in the salability testing.

[8] Since we calculate the speedup of *pmc* compared to *mc*, the absolute speed of the two CPUs is not important.

them. For instance, the refinement checker FDR does not support shared variables/arrays, and therefore, FDR's model is significantly different from PAT's. Further, FDR has no support for multi-core. The model checker SPIN supports verification of LTL properties. The multi-core parallel algorithm in SPIN is designed for model checking based on nested depth-first search. Nested depth-first-search works well for verification under no fairness. It can be twisted to perform model checking under fairness in the price of significant computational overhead, which has been shown in [31]. As a result, it makes little sense here to compare performance of our parallel algorithm with SPIN's.

6 Related Works

LTL parallel verification is an active research area due to the prevalence of the multi-core CPU and distributed systems. There are various approaches in the literature, as discussed below.

Holzmann proposed an extension of the SPIN model checker for dual-core machines in [16]. The algorithms keep their linear time complexity and the liveness checking algorithm supports full LTL. The algorithm for checking safety properties scales well to N-core systems. The algorithm for liveness checking, which is based on the original SPIN's nested DFS algorithm, can only be applied in dual-core systems. Furthermore, our approach handles different forms of fairness, while SPIN handles only process level weak fairness.

Lafuente [22] presented a cycle localization algorithm based on nested DFS, which is very similar to our ideas. In their approach, the main thread performs the first DFS to identify an accepting state, and the worker threads perform the second DFS to detect the fair cycle from the accepting state. Compared to this solution, our approach has the advantage that each SCC will be checked by one and only one worker thread.

A multi-core LTL model checking algorithm based on known distributed-memory algorithms is presented in [2]. This algorithm is linear for properties expressible as weak Büchi automata. However, the worst case complexity is quadratic. Our approach has no restriction on the types of LTL, and has linear time complexity in worst case.

A different approach to shared-memory model checking is presented in [19] based on CTL* translation to Hesitant Alternating Automata. The proposed algorithm uses non-emptiness game for deciding validity of the original formula and is therefore largely unrelated to the algorithms based on fair-cycle detection.

Barnat, Chaloupka and Pol gave a comprehensive survey in the distributed SCC decomposition algorithms [3]. We briefly list some of important ones in the following. These algorithms are designed for distributed systems and has quadratical or cubic order of complexity.

MAP The idea of the Maximal Accepting Predecessor algorithm [7, 8] relies on the fact that every accepting vertex inside an accepting cycle is its own predecessor. Direct implementation from this idea would give expensive computation

and store all proper accepting predecessors of all (accepting) vertices. To solve this problem, the MAP algorithm stores only a single representative of all proper accepting predecessor. The time complexity of the algorithm is $O(a^2 \times m)$, where a is the number of accepting vertices and m is the number of edges.

OWCTY The One Way Catch Them Young algorithm [12, 9] is to try to repeatedly remove vertices from the graph that cannot lie on an accepting cycle. The two removal rules of this algorithm are explained as follows: (1) a vertex is removed from the graph if it has no successors in the graph (the vertex cannot lie on a cycle), and (2) a vertex is removed if it cannot reach an accepting vertex (a potential cycle the vertex lies on is non-accepting). The algorithm continues the removal steps until there are more vertices to be removed. In the end, either there are some vertices remaining in the graph meaning that the original graph contained an accepting cycle, or all vertices have been removed meaning that the original graph had no accepting cycles. The time complexity of the algorithm is $O(h \times m)$ where h is the height of the SCC quotient graph. Here the factor m comes from the computation of elimination rules while the factor h relates to the number of global iterations the removal rules must be applied.

NEGC The idea behind the Negative Cycle Algorithm [6] is to transform the LTL model checking problem to the problem of negative cycle detection. Every edge of the graph outgoing from a non-accepting vertex is labeled with 0 while every edge outgoing from an accepting vertex is labeled with 1. Clearly, the graph contains a negative cycle if and only if it has an accepting cycle. The worst case time complexity of the algorithm is $O(n \times m)$, where n is the number of vertices and m is the number of edges.

OBF This algorithm is based on a recent technique OWCTY-BWD-FWD (OBF) [4, 5]. It identifies a number of independent subgraphs (called OBF slices) in $O(n+m)$ time, where n is the number of vertices and m is the number of edges. The slices are then decomposed using the FB algorithm. This algorithm assumes the input graph to be rooted, i.e., we have an initial vertex from which all other vertices are reachable. The time complexity of the algorithm is $O(n \times (n + m))$.

7 Conclusion

In this work, we proposed a parallel LTL-verification on fairness enhanced systems in multi-core shared-memory architecture. Based on the Tarjan's algorithm, our approach separated the SCC analysis into workers threads by careful algorithm design. Our approach is holistic, which does not only take care of LTL verification but also check the fairness constraints satisfaction in one goal. Fairness enhanced systems may contains big and complicated SCC structures in the state space. Our approach can split the workload to worker threads to achieve performance improvement. The solution is on-the-fly and the complexity is linear to the size of state space. We have implemented this technique in our home grown

model checker PAT. The experimental results on real world systems suggested our solution is efficient and scalable to multi-cores.

It is a well known fact, that a distributed-memory, parallel algorithm is straightforwardly transformed into a shared-memory one. But not vise verse. One of the future work is to migrate our approach into distributed systems with the aim of minimizing the communication overhead. Furthermore, we will conduct more experiments in the future to see both the scalability and limitations with more CPU cores.

References

1. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. In *9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, volume 3974 of *LNCS*, pages 103–117, 2005.
2. J. Barnat, L. Brim, and P. Ročkai. Scalable Multi-core LTL Model-Checking. In *Model Checking Software*, pages 187–203. Springer, 2007.
3. J. Barnat, J. Chaloupka, and J. Van De Pol. Distributed Algorithms for SCC Decomposition. *To appear in Journal of Logic and Computation*, 2009.
4. J. Barnat, J. Chaloupka, and J. van de Pol. Improved Distributed Algorithms for SCC Decomposition. *ENTCS*, 198(1):63–77, 2008.
5. J. Barnat and Pavel Moravec. Parallel Algorithms for Finding SCCs in Implicitly Given Graphs. In *Formal Methods: Applications and Technology*, volume 4346 of *LNCS*, pages 316–330. Springer, 2006.
6. L. Brim, I. Cerná, P. Krcál, and R. Pelánek. Distributed LTL Model Checking Based on Negative Cycle Detection. In *Foundations of Software Technology and Theoretical Computer Science (FST TCS 2001)*, volume 2245 of *LNCS*, pages 96–107. Springer, 2001.
7. L. Brim, I. Cerna, P. Moravec, and J. Simsa. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *5th International Conference of Formal Methods in Computer-Aided Design (FMCAD 2004)*, pages 352–366, 2004.
8. L. Brim, I. Cerna, P. Moravec, and J. Simsa. How to Order Vertices for Distributed LTL Model-Checking Based on Accepting Predecessors. In *4th International Workshop on Parallel and Distributed Methods in verifiCation*, pages 1–12, 2005.
9. I. Cerná and R. Pelánek. Distributed Explicit Fair Cycle Detection (Set Based Approach). In *10th International SPIN Workshop on Model Checking Software (SPIN 2003)*, volume 2648 of *LNCS*, pages 49–73. Springer, 2003.
10. C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, 1992.
11. M. J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *10th International Conference on Principles of Distributed Systems (OPODIS 2006)*, volume 4305 of *LNCS*, pages 395–409. Springer, 2006.
12. K. Fisler, R. Fraer, G. Kamhi, M. Y. Vardi, and Z. Yang. Is There a Best Symbolic Cycle-Detection Algorithm? In *Tools and Algorithms for Construction and Analysis of Systems (TACAS 2001)*, pages 420–434. Springer, 2001.
13. J. Geldenhuys and A. Valmari. More Efficient On-the-fly LTL Verification with Tarjan's Algorithm. *Theoritical Computer Science*, 345(1):60–82, 2005.

14. D. Giannakopoulou, J. Magee, and J. Kramer. Checking Progress with Action Priority: Is it Fair? In *7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 1999)*, pages 511–527, 1999.
15. M. R. Henzinger and J. A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *Scandinavian Workshop on Algorithm Theory (SWAT 1996)*, pages 16–27, 1996.
16. G. J. Holzmann and D. Bosnacki. The Design of a Multicore Extension of the SPIN Model Checker. *IEEE Trans. Softw. Eng.*, 33(10):659–674, 2007.
17. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.
18. G.J. Holzmann, D. Peled, and M. Yannakakis. On Nested Depth-first Search. In *The Spin Verification System*, pages 23–32, 1996.
19. C. P. Inggs and H. Barringer. CTL* Model Checking on a Shared-memory Architecture. *Form. Methods Syst. Des.*, 29(2):135–155, 2006.
20. Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model Checking with Strong Fairness. *Formal Methods and System Design*, 28(1):57–84, 2006.
21. Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton university press, 1995.
22. A. L. Lafuente. Simplified Distributed LTL Model Checking by Localizing Cycles. Technical report, Institute of Computer Science, Albert-Ludwings Universität Freiburg, 2002.
23. L. Lamport. Fairness and Hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.
24. Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
25. D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *8th Colloquium on Automata, Languages and Programming (ICALP 1981)*, pages 264–277, 1981.
26. M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 362–371. ACM, 2008.
27. A. Pnueli and Y. Sa'ar. All You Need Is Compassion. In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2008)*, volume 4905 of *LNCS*, pages 233–247, 2008.
28. J. H. Reif. Depth-First Search is Inherently Sequential. *Information Processing Letters*, 20(5):229–234, 1985.
29. J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2009)*, pages 127–135, 2009.
30. J. Sun, Y. Liu, J. S. Dong, and J. Pang. Towards a Toolkit for Flexible and Efficient Verification under Fairness. Technical Report TRB2/09, National Univ. of Singapore, Dec 2008. http://www.comp.nus.edu.sg/~pat/report.ps.
31. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *21th International Conference on Computer Aided Verification (CAV 2009)*, pages 702–708, Grenoble, France, 2009.
32. J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *10th International Conference on Formal Engineering Methods (ICFEM 2008)*, volume 5256 of *LNCS*, pages 318–337. Springer, 2008.
33. R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 2:146–160, 1972.