Fair Model Checking with Process Counter Abstraction

Jun Sun, Yang Liu, Abhik Roychoudhury, Shanshan Liu and Jin Song Dong*

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

Abstract. Parameterized systems are characterized by the presence of a large (or even unbounded) number of behaviorally similar processes, and they often appear in distributed/concurrent systems. A common state space abstraction for checking parameterized systems involves not keeping track of process identifiers by grouping behaviorally similar processes. Such an abstraction, while useful, conflicts with the notion of fairness. Because process identifiers are lost in the abstraction, it is difficult to ensure fairness (in terms of progress in executions) among the processes. In this work, we study the problem of fair model checking with process counter abstraction. Even without maintaining the process identifiers, our on-the-fly checking algorithm enforces fairness by keeping track of the local states from where actions are enabled / executed within an execution trace. We enhance our home-grown PAT model checker with the technique and show its usability via the automated verification of several real-life protocols.

1 Introduction

Parameterized concurrent systems consist of a large (or even unbounded) number of behaviorally similar processes of the same type. Such systems frequently arise in distributed algorithms and protocols (*e.g.*, cache coherence protocols, control software in automotive / avionics) — where the number of behaviorally similar processes is unbounded during system design, but is fixed later during system deployment. Thus, the deployed system contains fixed, finite number of behaviorally similar processes. However during system modeling/verification it is convenient to not fix the number of processes in the system for the sake for achieving more general verification results. A parameterized system represents an infinite family of instances, each instance being finite-state. Property verification of a parameterized system involves verifying that *every finite state instance of the system* satisfies the property in question. In general, verification of parameterized systems is undecidable [2].

A common practice for analyzing parameterized systems can be to fix the number of processes to a constant. To avoid state space explosion, the constant is often small, compared to the size of the real applications. Model checking is then performed in the hope of finding a bug which is exhibited by a fixed (and small) number of processes. This practice can be incorrect because the real size of the systems is often unknown during system design (but fixed later during system deployment). It is also difficult to

* This research has been supported in part by National University of Singapore FRC grant R-252-000-385-112 and R-252-000-320-112.

fix the number of processes to a "large enough" constant such that the restricted system with fixed number of processes is observationally equivalent to the parameterized system with unboundedly many processes. Computing such a *large enough constant* is undecidable after all, since the parameterized verification problem is undecidable.

Since parameterized systems contain process types with large number of behaviorally similar processes (whose behavior follows a local finite state machine or FSM), a natural state space abstraction is to group the processes based on which state of the local FSM they reside in [23, 7, 24]. Thus, instead of saying "process 1 is in state s, process 2 is in state t and process 3 is in state s" — we simply say "2 processes are in state s and 1 is in state t". Such an abstraction reduces the state space by exploiting a powerful state space symmetry (concrete global states with different process identifiers but the same count of the processes in the individual local states get grouped into the same abstract global state), as often evidenced in real-life concurrent systems such as a caches, memories, mutual exclusion protocols and network protocols. Verification by traversing the abstract state space here produces a sound and complete verification procedure. However, if the total number of processes is unbounded, the aforementioned counter abstraction still does not produce a finite state abstract system. The count of processes in a local state can still be ω (unbounded number), if the total number of processes is ω. To achieve a finite state abstract system, we can adopt a *cutoff number*, so that any count greater than the *cutoff* number is abstracted to ω. This yields a finite state abstract system, model checking which we get a sound but incomplete verification procedure — any linear time Temporal Logic (LTL) property verified in the abstract system holds for all concrete finite-state instances of the system, but not vice-versa.

Contributions In this paper, we study the problem of fair model checking with process counter abstraction. Imagine a bus protocol where a large / unbounded number of processors are contending for bus access. If we do not assume any fairness in the bus arbitration policy, we cannot prove the non-starvation property, that is, bus accesses by processors are eventually granted. In general, fairness constraints are often needed for verification of such liveness properties — ignoring fairness constraints results in unrealistic counterexamples (e.g. where a processor requesting for bus access is persistently ignored by the bus arbiter for example) being reported. These counterexamples are of no interest to the protocol designer. To systematically rule out such unrealistic counterexamples (which never happen in a real implementation), it is important to verify the abstract system produced by our process counter abstraction under fairness. We do so in this paper. However, this constitutes a significant *technical challenge* — since we do not even keep track of the process identifiers, how can we ensure a fair scheduling among the individual processes!

In this work, we develop a novel technique for model checking parameterized systems under (weak or strong) fairness, against linear temporal logic (LTL) formulae. We show that model checking under fairness is feasible, even without the knowledge of process identifiers. This is done by systematically keeping track of the local states from which actions are enabled / executed within any infinite loop of the abstract state space. We develop necessary theorems to prove the soundness of our technique, and also present efficient on-the-fly model checking algorithms. Our method is realized within our home-grown PAT model checker [26]. The usability / scalability of PAT is demonstrated via (i) automated verification of several real-life parameterized systems and (ii) a quantitative comparison with the SPIN model checker [17].

2 Preliminaries

We begin by formally defining our system model.

Definition 1 (System Model). *A system model is a structure* $\mathcal{S} = (Var_G, init_G, Proc)$ *where* Var_G *is a finite set of global variables,* $init_G$ *is their initial valuation and* $Proc$ *is a parallel composition of multiple processes* $Proc = P_1 \parallel P_2 \parallel \cdots$ *such that each process* $P_i = (S_i, init_i, \rightarrow_i)$ *is a transition system.*

We assume that all global variables have finite domains and each P_i has finitely many local states. A local state represents a program text together with its local context (e.g. valuation of the local variables). Two local states are equivalent if and only if they represent the same program text and the same local context. Let $State$ be the set of all local states. We assume that $State$ has finitely many elements. This disallows unbounded non-tail recursion which results in infinite different local states. $Proc$ may be composed of infinitely many processes. Each process has a unique identifier. In an abuse of notation, we use P_i to represent the identifier of process P_i when the context is clear. Notice that two local states from different processes are equivalent only if the process identifiers are irrelevant to the program texts they represent. Processes may communicate through global variables, (multi-party) barrier synchronization or synchronous/asynchronous message passing. It can be shown that parallel composition $\parallel$ is symmetric and associative.

Example 1. Fig. 1 shows a model of the readers/writers problem, which is a simple protocol for the coordination of readers and writers accessing a shared resource. The protocol, which we refer to as RW, is designed for arbitrary number of readers and writers. Several readers can read concurrently, whereas writers require exclusive access. Global variable $counter$ records the number of readers which are currently accessing the resource; $writing$ is true if and only if a writer is updating the resource. A transition is of the form $[guard]name\{assignments\}$, where $guard$ is a guard condition which must be true for the transition to be taken and $assignments$ is a simple sequential program which updates global variables. The following are properties which are to be verified.

$$\begin{array}{ll} \Box!(counter > 0 \wedge writing) & -\ Prop_1 \\ \Box\Diamond counter > 0 & -\ Prop_2 \end{array}$$

Property $Prop_1$ is a safety property which states that writing and reading cannot occur simultaneously. Property $Prop_2$ is a liveness property which states that always eventually the resource can be accessed by some reader.

In order to define the operational semantics of a system model, we define the notion of a configuration to capture the global system state during the execution, which is referred to as *concrete configurations*. This terminology distinguishes the notion from the state space abstraction and the abstract configurations which will be introduced later.

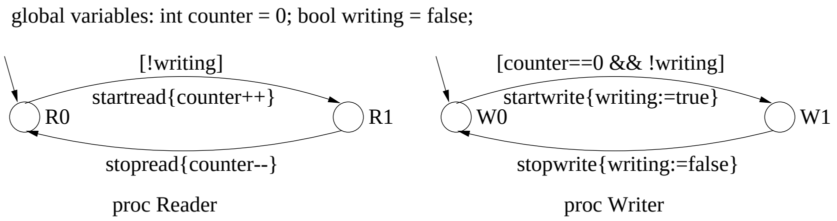

global variables: int counter = 0; bool writing = false;
[!writing]
startread{counter++}
R0
R1
stopread{counter--}
proc Reader
[counter==0 && !writing]
startwrite{writing:=true}
W0
W1
stopwrite{writing:=false}
proc Writer

Fig. 1. Readers/writers model

Definition 2 (Concrete Configuration). *Let $\mathcal{S}$ be a system model. A concrete configuration of $\mathcal{S}$ is a pair $(v, \langle s_1, s_2, \cdots \rangle)$ where v is the valuation of the global variables (channel buffers may be viewed as global variables), and $s_i \in S_i$ is the local state in which process P_i is residing.*

A system transition is of the form $(v, \langle s_1, s_2, \cdots \rangle) \rightarrow_{Ag} (v', \langle s_1', s_2', \cdots \rangle)$ where the system configuration after the transition is $(v', \langle s_1', s_2', \cdots \rangle)$ and Ag is a set of participating processes. For simplicity, set Ag (short for $agent$) is often omitted if irreverent. A system transition could be one of the following forms:

(i) a local transition of P_i which updates its local state (from s_i to s_i') and possibly updating global variables (from v to v'). An example is the transition from $R0$ to $R1$ of a reader. In such a case, P_i is the participating process, i.e., $Ag = \{P_i\}$.

(ii) a multi-party synchronous transition among processes $P_i, \cdots, P_j$. Examples are message sending/receiving through channels with buffer size 0 (e.g., as in Promela [17]) and alphabetized barrier synchronization in the classic CSP. In such a case, local states of the participating processes are updated simultaneously. The participating processes are $P_i, \cdots, P_j$.

(iii) process creation of P_m by P_i. In such a case, an additional local state is appended to the sequence $\langle s_1, s_2, \cdots \rangle$, and the state of P_i is changed at the same time. Assume for now that the sequence $\langle s_1, s_2, \cdots \rangle$ is always finite before process creation. It becomes clear in Section 5 that this assumption is not necessary. In such a case, the participating processes are P_i and P_m.

(iv) process deletion of P_i. In such case, the local state of P_i is removed from the sequence $(\langle s_1, s_2, \cdots \rangle)$. The participating process is P_i.

Definition 3 (Concrete Transition System). *Let $\mathcal{S} = (\mathit{Var}_G, \mathit{init}_G, \mathit{Proc})$ be a system model, where $\mathit{Proc} = P_1 \parallel P_2 \parallel \cdots$ such that each process $P_i = (S_i, \mathit{init}_i, \rightarrow_i)$ is a local transition system. The concrete transition system corresponding to $\mathcal{S}$ is a 3-tuple $T_{\mathcal{S}} = (C, \mathit{init}, \hookrightarrow)$ where C is the set of all reachable system configurations, init is the initial concrete configuration $(\mathit{init}_G, \langle \mathit{init}_1, \mathit{init}_2, \cdots \rangle)$ and $\hookrightarrow$ is the global transition relation obtained by composing the local transition relations $\rightarrow_i$ in parallel.*

An execution of $\mathcal{S}$ is an infinite sequence of configurations $E = \langle c_0, c_1, \cdots, c_i, \cdots \rangle$ where $c_0 = \mathit{init}$ and $c_i \hookrightarrow c_{i+1}$ for all $i \geq 0$. Given a model $\mathcal{S}$ and a system configuration c, let $\mathit{enabled}_{\mathcal{S}}(c)$ (or $\mathit{enabled}(c)$ when the context is clear) be the set of processes which is ready to make some progress, i.e., $\mathit{enabled}(c) = \{P_i \mid \exists\, c',\ c \hookrightarrow_{Ag} c' \wedge$

$P_i \in Ag\}$. The following defines two common notions of fairness in system executions, i.e., weak fairness and strong fairness.

Definition 4 (Weak Fairness). *Let $\mathcal{S}$ be a system model. An execution $\langle c_1, c_2, \cdots \rangle$ of $T_{\mathcal{S}}$ is weakly fair, if and only if, for every P_i there are infinitely many k such that $c_k \rightarrow_{Ag} c_{k+1}$ and $P_i \in Ag$ if there exists n so that $P_i \in enabled(c_m)$ for all $m > n$.*

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, weak fairness guarantees that each process is only finitely faster than the others.

Definition 5 (Strong Fairness). *Let $\mathcal{S}$ be a system model. An execution $\langle c_1, c_2, \cdots \rangle$ of $T_{\mathcal{S}}$ is strongly fair, if and only if, for every P_i there are infinitely many k such that $c_k \rightarrow_{Ag} c_{k+1}$ and $P_i \in Ag$ if there are infinitely many n such that $P_i \in enabled(c_n)$.*

Strong fairness states that if a process is infinitely often enabled, it must be infinitely often engaged. This type of fairness is particularly useful in the analysis of systems that use semaphores, synchronous communication, and other special coordination primitives. Clearly, strong fairness guarantees weak fairness.

In this work, we assume that system properties are expressed as *LTL formulae constituted by propositions on global variables*. One way to state property of a single process is to migrate part of its local context to global variables. Let ϕ be a property. $\mathcal{S}$ satisfies ϕ, written as $\mathcal{S} \vDash \phi$, if and only if every execution of $T_{\mathcal{S}}$ satisfies ϕ. $\mathcal{S}$ satisfies ϕ under weak fairness, written as $\mathcal{S} \vDash_{wf} \phi$, if and only if, every weakly fair execution of $T_{\mathcal{S}}$ satisfies ϕ. T satisfies ϕ under strong fairness, written as $T \vDash_{sf} \phi$, if and only if, every strongly fair execution of T satisfies ϕ.

Given the RW model presented in Fig. 1, it can be shown that $RW \vDash Prop_1$. It is, however, not easy to prove it using standard model checking techniques. The challenge is that many or unbounded number of readers and writers cause state space explosion. Also, RW fails $Prop_2$ without fairness constraint. For instance, a counterexample is $\langle startwrite, stopwrite \rangle^{\infty}$, i.e., a writer keeps updating the resource without any reader ever accessing it. This is unreasonable if the system scheduler is well-designed or the processors that the readers/writers execute on have comparable speed. To avoid such counterexamples, we need to perform model checking under fairness.

3 Process Counter Representation

Parameterized systems contain behaviorally similar or even identical processes. Given a configuration $(v, \langle \cdots, s_i, \cdots, s_j, \cdots \rangle)$, multiple local states[1] may be equivalent. A natural "abstraction" is to record only how many copies of a local state are there.

Let $\mathcal{S}$ be a system model. An alternative representation of a concrete configuration is a pair (v, f) where v is the valuation of the global variables and f is a total function from a local state to the set of processes residing at the state. For instance, given that $R0$ is a local state in Fig. 1, $f(R0) = \{P_i, P_j, P_k\}$ if and only if reader processes P_i, P_j and P_k are residing at state $R0$. This representation is sound and complete because processes

[1] The processes residing at the local states may or may not have the same process type.

at equivalent local states are behavioral equivalent and $\|$ composition is symmetric and associative (so that processes ordering is irrelevant).

Furthermore, given a local state s and processes residing at s, we may consider the processes indistinguishable (as the process identifiers must be irrelevant given the local states are equivalent) and abstract the process identifiers. That is, instead of associating a set of process identifiers with a local state, we only keep track of the number of processes. Instead of setting $f(R0) = \{P_i, P_j, P_k\}$, we now set $f(R0) = 3$. *In this and the next section, we assume that the total number of processes is bounded.*

Definition 6 (Abstract Configuration). *Let $\mathcal{S}$ be a system model. An abstract configuration of $\mathcal{S}$ is a pair (v, f) where v is a valuation of the global variables and $f : State \rightarrow \mathbb{N}$ is a total function*[2] *such that $f(s) = n$ if and only if n processes are residing at s.*

Given a concrete configuration $cc = (v, \langle s_0, s_1, \cdots \rangle)$, let $\mathcal{F}(\langle s_0, s_1, \cdots \rangle)$ returns the function f (refer to Definition 6) — that is, $f(s) = n$ if and only if there are n states in $\langle s_0, s_1, \cdots \rangle$ which are equivalent to s. Further, we write $\mathcal{F}(cc)$ to denote $(v, \mathcal{F}(\langle s_0, s_1, \cdots \rangle))$. Given a concrete transition $c \rightarrow_{Ag} c'$, the corresponding abstraction transition is written as $a \hookrightarrow_{Ls} a'$ where $a = \mathcal{F}(c)$ and $a' = \mathcal{F}(c')$ and Ls (short for local-states) is the local states at which processes in Ag are. That is, Ls is the set of local states from which there is a process leaving during the transition. We remark that Ls is obtained similarly as Ag is.

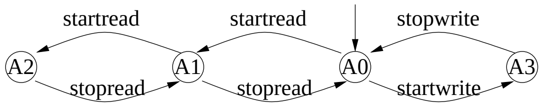
Given a local state s and an abstract configuration a, we define $enabled(s, a)$ *to be true if and only if* $\exists\, a',\ a \hookrightarrow_{Ls} a' \wedge s \in Ls$, i.e., a process is enabled to leave s in a. For instance, given the transition system in Fig. 2, $Ls = \{R0\}$ for the transition from $A0$ to $A1$ and $enabled(R0, A1)$ is true.

Definition 7 (Abstract Transition System). *Let $\mathcal{S} = (Var_G, init_G, Proc)$ be a system model, where $Proc = P_1 \parallel P_2 \parallel \cdots$ such that each process $P_i = (S_i, init_i, \rightarrow_i)$ is a local transition system. An abstract transition system of $\mathcal{S}$ is a 3-tuple $A_{\mathcal{S}} = (C, init, \hookrightarrow)$ where C is the set of all reachable abstract system configurations, $init \in C$ is $(init_G, \mathcal{F}(init_G, \langle init_1, init_2, \cdots \rangle))$ and $\hookrightarrow$ is the abstract global transition relation.*

We remark that the abstract transition relation can be constructed *without* constructing the concrete transition relation, which is essential to avoid state space explosion. Given the model presented in Fig. 1, if there are 2 readers and 2 writers, then the abstract transition system is shown in Fig. 2.

A concrete execution of $T_{\mathcal{S}}$ can be uniquely mapped to an execution of $A_{\mathcal{S}}$ by applying $\mathcal{F}$ to every configuration in the sequence. For instance, let $X = \langle c_0, c_1, \cdots, c_i, \cdots \rangle$ be an execution of $T_{\mathcal{S}}$ (i.e., a concrete execution), the corresponding execution of $A_{\mathcal{S}}$ is $L = \langle \mathcal{F}(c_0), \mathcal{F}(c_1), \cdots, \mathcal{F}(c_i), \cdots \rangle$ (i.e., the abstract execution). In an abuse of notation, we write $\mathcal{F}(X)$ to denote L. Notice that the valuation of the global variables are preserved. Essentially, no information is lost during the abstraction. It can be shown that $A_{\mathcal{S}} \vDash \phi$ if and only if $T_{\mathcal{S}} \vDash \phi$.

[2] In PAT, the mapping from a local state to 0 is always omitted for memory saving.



A0: ((writing,false),(counter,0),(R0,2),(R1,0),(W0,2),(W1,0))
A1: ((writing,false),(counter,1),(R0,1),(R1,1),(W0,2),(W1,0))
A2: ((writing,false),(counter,2),(R0,0),(R1,2),(W0,2),(W1,0))
A3: ((writing,true),(counter,0),(R0,2),(R1,0),(W0,1),(W1,1))

Fig. 2. Readers/writers model

4 Fair Model Checking Method

Process counter abstraction may significantly reduce the number of states. It is useful for verification of safety properties. However, it conflicts with the notion of fairness. A counterexample to a liveness property under fairness must be a fair execution of the system. By Definition 4 and 5, the knowledge of which processes are enabled or engaged is necessary in order to check whether an execution is fair or not. In this section, we develop the necessary theorems and algorithms to show that model checking under fairness constraints is feasible even without the knowledge of process identifiers.

By assumption the total number of processes is finite, the abstract transition system $A_{\mathcal{S}}$ has finitely many states. An infinite execution of $A_{\mathcal{S}}$ must form a loop (with a finite prefix to the loop). Assume that the loop starts with index i and ends with k, written as $L_i^k = \langle c_0, \cdots, c_i, c_{i+1}, \cdots, c_{i+k}, c_{i+k+1} \rangle$ where $c_{i+k+1} = c_i$. We define the following functions to collect loop properties and use them to define fairness later.

$$\begin{aligned}
&\mathit{always}(L_i^k) = \{s : \mathit{State} \mid \forall\, j : \{i, \cdots, i+k\},\ \mathit{enabled}(s, c_j)\}\\
&\mathit{once}(L_i^k) \quad = \{s : \mathit{State} \mid \exists\, j : \{i, \cdots, i+k\},\ \mathit{enabled}(s, c_j)\}\\
&\mathit{leave}(L_i^k) \quad = \{s : \mathit{State} \mid \exists\, j : \{i, \cdots, i+k\},\ c_j \hookrightarrow_{Ls} c_{j+1} \wedge s \in Ls\}
\end{aligned}$$

Intuitively, $always(L_i^k)$ is the set of local states from where there are processes, which are ready to make some progress, throughout the execution of the loop; $once(L_i^k)$ is the set of local states where there is a process which is ready to make some progress, at least once during the execution of the loop; $\mathit{leave}(L_i^k)$ is the set of local states from which processes leave during the loop. For instance, given the abstract transition system in Fig. 2, $X = \langle A0, A1, A2 \rangle^\infty$ is a loop starting with index 0 and ending with index 2. $always(X) = \varnothing$; $once(X) = \{R0, R1, W0\}$; $\mathit{leave}(X) = \{R0, R1\}$.

The following lemma allows us to check whether an execution is fair by only looking at the abstract execution.

Lemma 1. *Let $\mathcal{S}$ be a system model; X be an execution of $T_{\mathcal{S}}$; $L_i^k = \mathcal{F}(X)$ be the respective abstract execution of $A_{\mathcal{S}}$.* (1). *$always(L_i^k) \subseteq \mathit{leave}(L_i^k)$ if X is weakly fair;* (2). *$once(L_i^k) \subseteq \mathit{leave}(L_i^k)$ if X is strongly fair.*

Proof: (1). Assume X is weakly fair. By definition, if state s is in $always(L_i^k)$, there must be a process residing at s which is enabled to leave during every step of the loop. If it is the same process P, P is always enabled during the loop and therefore, by definition 4, P must participate in a transition infinitely often because X is weakly fair. Therefore, P must leave s during the loop. By definition, s must be in $\mathit{leave}(L_i^k)$. If there are different processes enabled at s during the loop, there must be a process leaving s, so that $s \in \mathit{leave}(L_i^k)$. Thus, $always(L_i^k) \subseteq \mathit{leave}(L_i^k)$.

(2). Assume X is strongly fair. By definition, if state s is in $once(L_i^k)$, there must be a process residing at s which is enabled to leave during one step of the loop. Let P be the process. Because P is infinitely often enabled, by Definition 4, P must participate in a transition infinitely often because X is strongly fair. Therefore, P must leave s during the loop. By definition, s must be in $leave(L_i^k)$. □

The following lemma allows us to generate a concrete fair execution if an abstract *fair* execution is identified.

Lemma 2. *Let $\mathcal{S}$ be a model; L_i^k be an execution of $A_{\mathcal{S}}$. (1). If $always(L_i^k) \subseteq leave(L_i^k)$, there exists a weakly fair execution X of $T_{\mathcal{S}}$ such that $\mathcal{F}(X) = L_i^k$; (2). If $once(L_i^k) \subseteq leave(L_i^k)$, there exists a strongly fair execution X of $T_{\mathcal{S}}$ such that $\mathcal{F}(X) = L_i^k$.*

Proof: (1). By a simple argument, there must exist an execution X of $T_{\mathcal{S}}$ such that $\mathcal{F}(X) = L_i^k$. Next, we show that we can unfold the loop (of the abstract fair execution) as many times as necessary to let all processes make some progress, so as to generate a weakly fair concrete execution. Assume P is the set of processes residing at a state s during the loop. Because $always(L_i^k) \subseteq leave(L_i^k)$, if $s \in always(L_i^k)$, there must be a transition during which a process leaves s. We repeat the loop multiple times and choose a different process from P to leave each time. The generated execution must be weakly fair.

(2). Similarly as above. □

The following theorem shows that we can perform model checking under fairness by examining the abstract transition system only.

Theorem 1. *Let $\mathcal{S}$ be a system model. Let ϕ be an LTL property. (1). $\mathcal{S} \vDash_{wf} \phi$ if and only if for all executions L_i^k of $A_{\mathcal{S}}$ we have $always(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$; (2). $\mathcal{S} \vDash_{sf} \phi$ if and only if for all execution L_i^k of $A_{\mathcal{S}}$ we have $once(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$.*

Proof: (1). **if** part: Assume that for all L_i^k of $A_{\mathcal{S}}$ we have $L_i^k \vDash \phi$ if $always(L_i^k) \subseteq leave(L_i^k)$, and $\mathcal{S} \nvDash_{wf} \phi$. By definition, there exists a weakly fair execution X of $T_{\mathcal{S}}$ such that $X \nvDash \phi$. Let L_i^k be $\mathcal{F}(X)$. By lemma 1, $always(L_i^k) \subseteq leave(L_i^k)$ and hence $L_i^k \vDash \phi$. Because our abstraction preserves valuation of global variables, $L_i^k \nvDash \phi$ as $X \nvDash \phi$. We reach a contradiction.
only if part: Assume that $\mathcal{S} \vDash_{wf} \phi$ and there exists L_i^k of $A_{\mathcal{S}}$ such that $always(L_i^k) \subseteq leave(L_i^k)$, and $L_i^k \nvDash_{wf} \phi$. By lemma 2, there must exist X of $T_{\mathcal{S}}$ such that X is weakly fair. Because process counter abstraction preserves valuations of global variables, $X \nvDash \phi$. Hence, we reach contradiction.

(2). Similarly as above. □

Thus, in order to prove that $\mathcal{S}$ satisfies ϕ under fairness, we need to show that there is no execution L_i^k of $A_{\mathcal{S}}$ such that $L_i^k \nvDash \phi$ and the execution satisfies an additional constraint for fairness, i.e., $always(L_i^k) \subseteq leave(L_i^k)$ for weak fairness or $once(L_i^k) \subseteq leave(L_i^k)$ for strong fairness. Or, if $\mathcal{S} \nvDash_{wf} \phi$, then there must be an execution L_i^k of $A_{\mathcal{S}}$ such that L_i^k satisfies the fairness condition and $L_i^k \nvDash \phi$. In such a case, we can generate a concrete execution.

Following the above discussion, fair model checking parameterized systems is reduced to searching for particular loops in $A_{\mathcal{S}}$. There are two groups of methods for loop searching. One is based on nested depth-first-search (DFS) [17] and the other is based on identifying strongly connected components (SCC) [12]. It has been shown that the nested DFS is not suitable for model checking under fairness assumptions, as whether an execution is fair depends on the path instead of one state [17]. In this work, we extend the approaches presented in [12, 27] to cope with weak or strong fairness and process counter abstraction. Given $A_{\mathcal{S}}$ and a property ϕ, model checking involves searching for an execution of $A_{\mathcal{S}}$ which fails ϕ. In automata-based model checking, the negation of ϕ is translated to an equivalent Büchi automaton $\mathcal{B}_{\neg\phi}$, which is then composed with $A_{\mathcal{S}}$. Notice that a state in the produce of $A_{\mathcal{S}}$ and $\mathcal{B}_{\neg\phi}$ is a pair (a, b) where a is an abstract configuration of $A_{\mathcal{S}}$ and b is a state of $\mathcal{B}_{\neg\phi}$. Model checking under fairness involves searching for a fair execution which is accepted by the Büchi automaton.

Given a transition system, a strongly connected subgraph is a subgraph such that there is a path connecting any two states in the subgraph. An MSCC is a maximal strongly connected subgraph. Given the product of $A_{\mathcal{S}}$ and $\mathcal{B}_{\neg\phi}$, let scg be a set of states which, together with the transitions among them, forms a strongly connected subgraph. We say scg is accepting if and only if there exists one state (a, b) in scg such that b is an accepting state of $\mathcal{B}_{\neg\phi}$. In an abuse of notation, we refer to scg as the strongly connected subgraph in the following. The following lifts the previously defined functions on loops to strongly connected subgraphs.

$$\begin{array}{ll} always(scg) & = \{y : State \mid \forall\, x : scg,\ enabled(y, x)\} \\ once(scg) & = \{y : State \mid \exists\, x : scg,\ enabled(y, x)\} \\ leave(scg) & = \{z : State \mid \exists\, x, y : scg,\ z \in leave(x, y)\} \end{array}$$

$always(scg)$ is the set of local states such that for any local state in $always(scg)$, there is a process ready to leave the local state for every state in scg; $once(scg)$ is the set of local states such that for some local state in $once(scg)$, there is a process ready to leave the local state for some state in scg; and $leave(scg)$ is the set of local states such that there is a transition in scg during which there is a process leaving the local state. Given the abstract transition system in Fig. 2, $scg = \{A0, A1, A2, A3\}$ constitutes a strongly connected subgraph. $always(scg) =$ nil; $once(scg) = \{R0, R1, W0, W1\}$; $leave(scg) = \{R0, R1, W0, W1\}$.

Lemma 3. *Let $\mathcal{S}$ be a system model. There exists an execution L_i^k of $A_{\mathcal{S}}$ such that $always(L_i^k) \subseteq leave(L_i^k)$ if and only if there exists an MSCC scc of $A_{\mathcal{S}}$ such that $always(scc) \subseteq leave(scc)$.*

Proof: The **if** part is trivially true. The **only if** part is proved as follows. Assume there exists execution L_i^k of $A_{\mathcal{S}}$ such that $always(L_i^k) \subseteq leave(L_i^k)$, there must exist a strongly connected subgraph scg which satisfies $always(scg) \subseteq leave(scg)$. Let scc be the MSCC which contains scg. We have $always(scc) \subseteq always(scg)$, therefore, the MSCC scc satisfies $always(scc) \subseteq always(scg) \subseteq leave(scg) \subseteq leave(scc)$. □

The above lemma allows us to use MSCC detection algorithms for model checking under weak fairness. Fig. 3 presents an on-the-fly model checking algorithm based on Tarjan's algorithm for identifying MSCCs. The idea is to search for an MSCC scg such that $always(scg) \subseteq leave(scg)$ and scg is accepting. The algorithm terminates in two ways, either one such MSCC is found or all MSCCs have been examined (and it returns true). In the former case, an abstract counterexample is generated. In the latter case, we successfully prove the property. Given the system presented in Fig. 2, $\{A0, A1, A2, A3\}$

```
procedure $checkingUnderWeakFairness(A_{\mathcal{S}}, \mathcal{B}_{\neg\phi})$
1. while there are un-visited states in $A_{\mathcal{S}} \otimes \mathcal{B}_{\neg\phi}$
2.      use the improved Tarjan's algorithm to identify one SCC, say $scg$;
3.      if $scg$ is accepting to $\mathcal{B}_{\neg\phi}$ and $always(scg) \subseteq leave(scg)$
4.           generate a counterexample and return false;
5.      endif
6. endwhile
7. return $true$;
```

Fig. 3. Model checking algorithm under weak fairness

constitutes the only MSCC, which satisfies $always(scg) \subseteq leave(scg)$. The complexity of the algorithm is linear in the number of transitions of $A_{\mathcal{S}}$.

Lemma 4. *Let $\mathcal{S}$ be a system model. There exists an execution L_i^k of $A_{\mathcal{S}}$ such that $once(L_i^k) \subseteq leave(L_i^k)$ if and only if there exists a strongly connected subgraph scg of $A_{\mathcal{S}}$ such that $once(scg) \subseteq leave(scg)$.*

We skip the proof of the lemma as it is straightforward. The lemma allows us to extend the algorithm proposed in [27] for model checking under strong fairness. Fig. 4 presents the modified algorithm. The idea is to search for a strongly connected subgraph scg such that $once(scg) \subseteq leave(scg)$ and scg is accepting. Notice that a strongly connected subgraph must be contained in one and only one MSCC. The algorithm searches for MSCCs using Tarjan's algorithm. Once an MSCC scg is found (at line 2), if scg is accepting and satisfies $once(scg) \subseteq leave(scg)$, then we generate an abstract counterexample. If scg is accepting but fails $once(scg) \subseteq leave(scg)$, instead of throwing away the MSCC, we prune a set of *bad states* from the SCC and then examinate the remaining states (at line 6) for strongly connected subgraphs. Intuitively, *bad states* are the reasons why the SCC fails the condition $once(scg) \subseteq leave(scg)$. Formally,

$$bad(scg) = \{x : scg \mid \exists\, y,\ y \notin leave(scg) \land y \in enabled(y, x)\}$$

That is, a state s is bad if and only if there exists a local state y such that a process may leave y at state s and yet there is no process leaving y given all transitions in scg. By pruning all bad states, there might be a strongly connected subgraph in the remaining states which satisfies the fairness constraint.

The algorithm is partly inspired by the one presented in [16] for checking emptiness of Streett automata. Soundness of the algorithm follows the discussion in [27, 16]. It can be shown that any state of a strongly connected subgraph which satisfies the constraints is never pruned. As a result, if there exists such a strongly connected subgraph scg, a strongly connected subgraph which contains scg or scg itself must be found eventually. Termination of the algorithm is guaranteed because the number of visited states and pruned states are monotonically increasing. The complexity of the algorithm is linear in $\#states \times \#trans$ where $\#states$ and $\#trans$ are the number of states and transitions of $A_{\mathcal{S}}$ respectively. A tighter bound on the complexity can be found in [16].

```
procedure $checkingUnderStrongFairness(A_{\mathcal{S}}, \mathcal{B}_{\neg\phi}, states)$
while there are un-visited states in $states$
    use Tarjan's algorithm to identify a subset of $states$ which forms an SCC, say $scg$;
    if $scg$ is accepting to $\mathcal{B}_{\neg\phi}$
        if $once(scg) \subseteq leave(scg)$
            generate a counterexample and return false;
        else if $checkingUnderStrongFairness(A_{\mathcal{S}}, \mathcal{B}_{\neg\phi}, scg \setminus bad(scg))$ is false
            return false;
        endif
    endif
endwhile
return $true$;
```

Fig. 4. Model checking algorithm under strong fairness

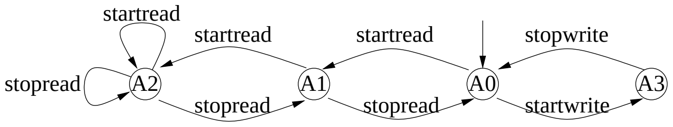
5 Counter Abstraction for Infinitely Many Processes

In the previous sections, we assume that the number of processes (and hence the size of the abstract transition system) is finite and bounded. If the number of processes is unbounded, there might be unbounded number of processes residing at a local state, e.g., the number of reader processes residing at $R0$ in Fig. 1 might be infinite. In such a case, we choose a *cutoff* number and then apply further abstraction. In the following, we *modify* the definition of abstract configurations and abstract transition systems to handle unbounded number of processes.

Definition 8. *Let $\mathcal{S}$ be a system model with unboundedly many processes. Let K be a positive natural number (i.e., the cutoff number). An abstract configuration of $\mathcal{S}$ is a pair (v, g) where v is the valuation of the global variables and $g : State \rightarrow \mathbb{N} \cup \{\omega\}$ is a total function such that $g(s) = n$ if and only if $n(\leq K)$ processes are residing at s and $g(s) = \omega$ if and only if more than K processes are at s.*

Given a configuration $(v, \langle s_0, s_1, \cdots \rangle)$, we define a function $\mathcal{G}$ similar to function $\mathcal{F}$, i.e., $\mathcal{G}(\langle s_0, s_1, \cdots \rangle))$ returns function g (refer to Definition 8) such that given any state s, $g(s) = n$ if and only if there are n states in $\langle s_0, s_1, \cdots \rangle$ which are equivalent to s and $g(s) = \omega$ if and only if there are more than K states in $\langle s_0, s_1, \cdots \rangle$ which are equivalent to s. Furthermore, $\mathcal{G}(c) = (v, \mathcal{G}(\langle s_0, s_1, \cdots \rangle))$.

The abstract transition relation of $\mathcal{S}$ (as per the above abstraction) can be constructed without constructing the concrete transition relation. We illustrate how to generate an abstract transition in the following. Given an abstract configuration (v, g), if $g(s) > 0$, a *local transition* from state s to state s', creating a process with initial state *init* may result in different abstract configurations (v, g') depending on g. In particular, g' equals g except that $g'(s) = g(s) - 1$ and $g'(s') = g(s') + 1$ and $g'(init) = g(init) + 1$ assuming $\omega + 1 = \omega$, $K + 1 = \omega$ and $\omega - 1$ is either ω or K. We remark that by assumption $State$ is a finite set and therefore the domain of g is always finite. This allows us to drop the assumption that the number of processes must be finite before process creation. Similarly, we abstract synchronous transitions and process termination.



A0: ((writing,false),(counter,0),(R0,inf),(R1,0),(W0,inf),(W1,0))
A1: ((writing,false),(counter,1),(R0,inf),(R1,1),(W0,inf),(W1,0))
A2: ((writing,false),(counter,inf),(R0,inf),(R1,inf),(W0,inf),(W1,0))
A3: ((writing,true),(counter,0),(R0,inf),(R1,0),(W0,inf),(W1,1))

Fig. 5. Abstract readers/writers model

The *abstract transition system* for a system model $\mathcal{S}$ with unboundedly many processes, written as $R_{\mathcal{S}}$ (to distinguish from $A_{\mathcal{S}}$), is now obtained by applying the aforementioned abstract transition relation from the initial abstract configuration.

Example 2. Assume that the cutoff number is 1 and there are infinitely many readers and writers in the readers/writers model. Because $counter$ is potentially unbounded and, we mark $counter$ as a special process counter variable which dynamically counts the number of processes which are reading (at state $R1$). If the number of *reading* processes is larger than the cutoff number, $counter$ is set to ω too. The abstract transition system A_{RW} is shown in Fig. 5. The abstract transition system may contain spurious traces. For instance, the trace $\langle start, (stopread)^{\infty} \rangle$ is spurious. It is straightforward to prove that $A_{RW} \vDash Prop_1$ based on the abstract transition system.

The abstract transition system now has only finitely many states even if there are unbounded number of processes and, therefore, is subject to model checking. As illustrated in the preceding example, the abstraction is sound but incomplete in the presence of unboundedly many processes. Given an execution X of $T_{\mathcal{S}}$, let $\mathcal{G}(X)$ be the corresponding execution of the abstract transition system. An execution L of $R_{\mathcal{S}}$ is spurious if and only if there does not exist an execution X of $T_{\mathcal{S}}$ such that $\mathcal{G}(X) = L$. Because the abstraction only introduces execution traces (but does not remove any), we can formally establish a simulation relation (but not a bisimulation) between the abstract and concrete transition systems, that is, $R_{\mathcal{S}}$ simulates $T_{\mathcal{S}}$. Thus, while verifying an LTL property ϕ we can conclude $T_{\mathcal{S}} \vDash \phi$ if we can show that $R_{\mathcal{S}} \vDash \phi$. Of course, $R_{\mathcal{S}} \vDash \phi$ will be accomplished by model checking under fairness.

The following re-establishes Lemma 1 and (part of) Theorem 1 in the setting of $R_{\mathcal{S}}$. We skip the proof as they are similar to that of Lemma 1 and Theorem 1 respectively.

Lemma 5. *Let $\mathcal{S}$ be a system model, X be an execution of $T_{\mathcal{S}}$ and $L_i^k = \mathcal{G}(X)$ be the corresponding execution of $R_{\mathcal{S}}$. We have* (1). *$always(L_i^k) \subseteq leave(L_i^k)$ if X is weakly fair;* (2).*$once(L_i^k) \subseteq leave(L_i^k)$ if X is strongly fair.*

Theorem 2. *Let $\mathcal{S}$ be a system model and ϕ be an LTL property.* (1). *$\mathcal{S} \vDash_{wf} \phi$ if for all execution traces L_i^k of $R_{\mathcal{S}}$ we have $always(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$;* (2). *$\mathcal{S} \vDash_{sf} \phi$ if for all execution traces L_i^k of $R_{\mathcal{S}}$ we have $once(L_i^k) \subseteq leave(L_i^k) \Rightarrow L_i^k \vDash \phi$;*

The reverse of Theorem 2 is not true because of spurious traces. We remark that the model checking algorithms presented in Section 4 are applicable to $R_{\mathcal{S}}$ (as the abstraction function is irrelevant to the algorithm). By Theorem 2, if model checking of $R_{\mathcal{S}}$ (using the algorithms presented in Section 4 under weak/fairness constraint) returns true, we conclude that the system satisfies the property (under the respective fairness).

6 Case Studies

Our method has been realized in the *Process Analysis Toolkit* (PAT) [26]. PAT is designed for systematic validation of distributed/concurrent systems using state-of-the-art model checking techniques. In the following, we show the usability/scalability of our method via the automated verification of several real-life parameterized systems. All the models are embedded in the PAT package and available online. The experimental results are summarized in the following table, where NA means not applicable (hence not tried, due to limit of the tool); NF means not feasible (out of 2GB memory or running for more than 4 hours). The data is obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 2GB RAM. We compared PAT with SPIN [17] on model checking under no fairness or weak fairness. Notice that SPIN does not support strong fairness and is limited to 255 processes.

Model	#Proc	Property	No Fairness			Weak Fairness			Strong Fairness		
			Result	PAT	SPIN	Result	PAT	SPIN	Result	PAT	Spin
LE	10	$\Diamond\Box$ $one\ leader$	false	0.04	0.015	true	0.06	320	true	0.06	NA
LE	100	$\Diamond\Box$ $one\ leader$	false	0.04	0.015	true	0.27	NF	true	0.28	NA
LE	1000	$\Diamond\Box$ $one\ leader$	false	0.04	NA	true	2.26	NA	true	2.75	NA
LE	10000	$\Diamond\Box$ $one\ leader$	false	0.04	NA	true	23.89	NA	true	68.78	NA
LE	∞	$\Diamond\Box$ $one\ leader$	false	0.06	NA	true	264.78	NA	true	463.9	NA
KV	2	$Prop_{Kvalue}$	false	0.05	0	true	0.6	1.14	true	0.6	NA
KV	3	$Prop_{Kvalue}$	false	0.05	0	true	4.56	61.2	true	4.59	NA
KV	4	$Prop_{Kvalue}$	false	0.05	0.015	true	29.2	NF	true	30.24	NA
KV	5	$Prop_{Kvalue}$	false	0.06	0.015	true	174.5	NF	true	187.1	NA
KV	∞	$Prop_{Kvalue}$	false	0.12	NA	?	NF	NA	?	NF	NA
$Stack$	5	$Prop_{stack}$	false	0.06	0.015	false	0.78	NF	false	0.74	NA
$Stack$	7	$Prop_{stack}$	false	0.06	0.015	false	11.3	NF	false	12.1	NA
$Stack$	9	$Prop_{stack}$	false	0.06	0.015	false	158.6	NF	false	191.8	NA
$Stack$	10	$Prop_{stack}$	false	0.05	0.015	false	596.1	NF	false	780.3	NA
ML	10	$\Box\Diamond$ $access$	true	0.11	21.5	true	0.11	107	true	0.11	NA
ML	100	$\Box\Diamond$ $access$	true	1.04	NF	true	1.04	NF	true	1.04	NA
ML	1000	$\Box\Diamond$ $access$	true	11.04	NA	true	11.08	NA	true	11.08	NA
ML	∞	$\Box\Diamond$ $access$	true	13.8	NA	true	13.8	NA	true	13.8	NA

The first model (LE) is a self-stabilizing leader election protocol for complete networks [11]. Mobile ad hoc networks consist of multiple mobile nodes which interact with each other. The interactions among the nodes are subject to fairness constraints. One essential property of a self-stabilizing population protocols is that all nodes must eventually converge to the correct configurations. We verify the self-stabilizing leader election algorithm for complete network graphs (i.e., any pair of nodes are connected). The property is that eventually always there is one and only one leader in the network, i.e., $\Diamond\Box$ $one\ leader$. PAT successfully proved the property under weak or strong fairness for many or unbounded number of network nodes (with cutoff number 2). SPIN took much more time to prove the property under weak fairness. The reason is that the fair model checking algorithm in SPIN copies the global state machine $n+2$ times (for n processes) so as to give each process a fair chance to progress, which increases the verification time by a factor that is linear in the number of network nodes.

The second model (KV) is a K-valued register [3]. A shared K-valued multi-reader single-writer register R can be simulated by an array of K binary registers. When the single writer process wants to write v into R, it will set the v-th element of B to 1 and then set all the values before v-th element to 0. When a reader wants to read the value, it will do an upwards scan first from 0 to the first element u whose value is 1, then do a downwards scan from u to 0 and remember the index of the last element with value 1, which is the return value of the reading operation. A *progress* property is that $Prop_{Kvalue} = \Box(read\_inv \rightarrow \Diamond read\_res)$, i.e., a reading operation ($read\_inv$) eventually returns some valid value ($read\_res$). With no fairness, both PAT and SPIN identified a counterexample quickly. Because the model contains many local states, the size of $A_{\mathcal{S}}$ increases rapidly. PAT proved the property under weak/strong fairness for 5 processes, whereas SPIN was limited to 3 processes with weak fairness.

The third model ($Stack$) is a lock-free stack [28]. In concurrent systems, in order to improve the performance, the stack can be implemented by a linked list, which is shared by arbitrary number of processes. Each push or pop operation keeps trying to update the stack until no other process interrupts. The property of interest is that a process must eventually be able to update the stack, which can be expressed as the LTL $Prop_{stack} = \Box(push\_inv \rightarrow \Diamond push\_res)$ where event $push\_inv$ ($push\_res$) marks the starting (ending) of $push$ operation. The property is false even under strong fairness.

The fourth model (ML) is the Java meta-lock algorithm [1]. In Java language, any object can be synchronized by different threads via synchronized methods or statements. The Java meta-locking algorithm is designed to ensure the mutually exclusive access to an object. A synchronized method first acquires a lock on the object, executes the method and then releases the lock. The property is that always eventually some thread is accessing the object, i.e., $\Box\Diamond\ access$, which is true without fairness. This example shows that the computational overhead due to fairness is negligible in PAT.

In another experiment, we use a model in which processes all behave differently (so that counter abstraction results in no reduction) and each process has many local states. We then compare the verification results with or without process counter abstraction. The result shows the computational and memory overhead for applying the abstraction is negligible. In summary, the enhanced PAT model checker complements existing model checkers in terms of not only performance but also the ability to perform model checking under weak or strong fairness with process counter abstraction.

7 Discussion and Related Work

We studied model checking under fairness with process counter abstraction. The contribution of our work is twofold. First, we presented a fully automatic method for property checking of under fairness with process counter abstraction. We showed that fairness can be achieved without the knowledge of process identifiers. Secondly, we enhanced our home-grown PAT model checker to support our method and applied it on large scale parameterized systems to demonstrate its scalability. As for future work, we plan to investigate methods to combine well-known state space reduction techniques (such as partial order reduction, data abstraction for infinite domain data variables) with the process counter abstraction so as to extend the applicability of our model checker.

Verification of parameterized systems is undecidable [2]. There are two possible remedies to this problem: either we look for restricted subsets of parameterized systems for which the verification problem becomes decidable, or we look for sound but not necessarily complete methods. The first approach tries to identify a *restricted subset* of parameterized systems and temporal properties, such that if a property holds for a system with up to a certain number of processes, then it holds for any number of processes in the system. Moreover, the verification for the reduced system can be accomplished by using model checking. This approach can be used to verify a number of systems [13, 18, 8]. The sound but incomplete approaches include methods based on synthesis of invisible invariant (*e.g.*, [10]); methods based on network invariant (*e.g.*, [21]) that relies on the effectiveness of a generated invariant and the invariant refinement techniques; regular model checking [19] that requires acceleration techniques. Verification of liveness properties under fairness constraints have been studied in [15, 17, 20]. These works are based on SCC-related algorithms and decide the existence of an accepting run of the product of the transition system and Büchi automata, Streett automata or linear weak alternating automaton.

The works closest to ours are the methods based on *counter abstraction* (*e.g.*, [7, 24, 23]). In particular, verification of liveness properties under fairness is addressed in [23]. In [23], the fairness constraints for the abstract system are generated manually (or via heuristics) from the fairness constraints for the concrete system. Different from the above work, our method handles one (possibly large) instance of parameterized systems at a time and uses counter abstraction to improve verification effectiveness. In addition, fairness conditions are integrated into the on-the-fly model checking algorithm which proceeds on the abstract state representation — making our method fully automated.

Our method is related to work on symmetry reduction [9, 5]. A solution for applying symmetry reduction under fairness is discussed in [9]. Their method works by finding a candidate fair path in the abstract transition system and then using special annotations to resolve the abstract path to a threaded structure which then determines whether there is a corresponding fair path in the concrete transition system. A similar approach was presented in [14]. Different from the above, our method employs a specialized form of symmetry reduction and deals with the abstract transition system only and requires no annotations. Additionally, a number of works on combining abstraction and fairness, were presented in [6, 22, 29, 4, 25]. Our work explores one particular kind of abstraction and shows that it works with fairness with a simple twist.

References

1. O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y.S. Ramakrishna, and D. White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization. In *OOPSLA*, pages 207–222, 1999.
2. K.R. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent systems. *Inf. Process. Lett.*, 22(6):307–309, 1986.
3. H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. John Wiley & Sons, Inc., Publication, 2nd edition, 2004.
4. D. Bosnacki, N. Ioustinova, and N. Sidorova. Using Fairness to Make Abstractions Work. In *SPIN'04*, volume 2989 of *LNCS*, pages 198–215. Springer, 2004.

5. E. M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. In *CAV'93*, volume 697 of *LNCS*, pages 450–462. Springer, 1993.
6. D. Dams, R. Gerth, and O. Grumberg. Fair Model Checking of Abstractions. In *VCL'2000*.
7. G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols. In *CAV*, pages 53–68, 2000.
8. E. A. Emerson and K. S. Namjoshi. On Reasoning About Rings. *Int. J. Found. Comput. Sci.*, 14(4):527–550, 2003.
9. E. A. Emerson and A. P. Sistla. Utilizing Symmetry when Model-Checking under Fairness Assumptions: An Automata-Theoretic Approach. *ACM Trans. Program. Lang. Syst.*, 19(4):617–638, 1997.
10. Y. Fang, K. L. McMillan, A. Pnueli, and L. D. Zuck. Liveness by Invisible Invariants. In *FORTE*, volume 4229 of *LNCS*, pages 356–371, 2006.
11. M.J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *OPODIS*, volume 4305 of *LNCS*, pages 395–409, 2006.
12. J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoretical Computer Science*, 345(1):60–82, 2005.
13. S.M. German and A.P. Sistla. Reasoning about Systems with Many Processes. *J. ACM*, 39(3):675–735, 1992.
14. V. Gyuris and A. P. Sistla. On-the-Fly Model Checking Under Fairness That Exploits Symmetry. In *CAV*, volume 1254 of *LNCS*, pages 232–243. Springer, 1997.
15. M. Hammer, A. Knapp, and S. Merz. Truly On-the-Fly LTL Model Checking. In *TACAS*, volume 3440 of *LNCS*, pages 191–205, 2005.
16. M.R. Henzinger and J.A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *SWAT*, pages 16–27, 1996.
17. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.
18. C. N. Ip and D. L. Dill. Verifying Systems with Replicated Components in Mur&b.phiv;. *Formal Methods in System Design*, 14(3):273–310, 1999.
19. B. Jonsson and M. Saksena. Systematic Acceleration in Regular Model Checking. In *CAV*, volume 4590 of *LNCS*, pages 131–144, 2007.
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. D. Lesens, N. Halbwachs, and P. Raymond. Automatic Verification of Parameterized Linear Networks of Processes. In *POPL*, pages 346–357, 1997.
22. U. Nitsche and P. Wolper. Relative Liveness and Behavior Abstraction (Extended Abstract). In *PODC'97*, pages 45–52, 1997.
23. A. Pnueli, J. Xu, and L.D. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *CAV*, pages 107–122, 2002.
24. F. Pong and M. Dubois. A New Approach for the Verification of Cache Coherence Protocols. *IEEE Trans. Parallel Distrib. Syst.*, 6(8):773–787, 1995.
25. F. Pong and M. Dubois. Verification Techniques for Cache Coherence Protocols. *ACM Comput. Surv.*, 29(1):82–126, 1996.
26. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.
27. J. Sun, Y. Liu, J.S. Dong, and H.H. Wang. Specifying and verifying event-based fairness enhanced systems. In *ICFEM*, volume 5256 of *LNCS*, pages 318–337. Springer, 2008.
28. R.K. Treiber. Systems programming: Coping with parallelism. Technical report, 1986.
29. U. Ultes-Nitsche and S. St. James. Improved Verification of Linear-time Properties within Fairness: Weakly Continuation-closed Behaviour Abstractions Computed from Trace Reductions. *Softw. Test., Verif. Reliab.*, 13(4):241–255, 2003.