Front. Comput. Sci., 2014, 8(1): 1–16
DOI 10.1007/s11704-013-3091-5

RESEARCH ARTICLE

Model checking with fairness assumptions using PAT

Yuanjie SI[1], Jun SUN(✉)[2], Yang LIU[3], Jin Song DONG[4], Jun PANG[5], Shao Jie ZHANG[2], Xiaohu YANG[1]

1 College of Computer Science, Zhejiang University, Hangzhou 310027, China
2 Information System Technology and Design, Singapore University of Technology and Design, Singapore 138682, Singapore
3 School of Computer Engineering, Nanyang Technological University, Singapore 639798, Singapore
4 School of Computing, National University of Singapore, Singapore 117417, Singapore
5 Faculty of Science, Technology and Communication, University of Luxembourg, Luxembourg L-1359, Luxembourg

Abstract Recent development on distributed systems has shown that a variety of fairness constraints (some of which are only recently defined) play vital roles in designing self-stabilizing population protocols. Existing model checkers are deficient in verifying the systems as only limited kinds of fairness are supported with limited verification efficiency. In this work, we support model checking of distributed systems in the toolkit PAT (process analysis toolkit), with a variety of fairness constraints (e.g., process-level weak/strong fairness, event-level weak/strong fairness, strong global fairness). It performs on-the-fly verification against linear temporal properties. We show through empirical evaluation (on recent population protocols as well as benchmark systems) that PAT has advantage in model checking with fairness. Previously unknown bugs have been revealed against systems which are designed to function only with strong global fairness.

Keywords model checking, fairness, PAT, verification tool, formal methods

1 Introduction

In the area of distributed and concurrent system verification, liveness means something "good" must eventually happen. For instance, a typical requirement for leader election protocols is that one and only one leader must be elected eventually in a network. A counterexample to a liveness property (against a finite state system) is a loop (or a deadlock state, which can be viewed as a trivial loop) during which the "good" thing never occurs. For instance, the network nodes may repeatedly exchange a sequence of messages and never elect a leader.

Fairness, which is concerned with a fair resolution of nondeterminism, is often necessary and important to prove liveness properties. 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 computational resources 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 with fairness. In this work, we focus on formal system analysis with fairness assumptions. The objective is to deliver a toolkit which model checks linear temporal logic (LTL) properties against distributed systems functioning with a variety of fairness constraints.

Fairness and model checking with fairness have attracted

Received March 18, 2013; accepted October 10, 2013

E-mail: sunjun@sutd.edu.sg

much theoretical interest for decades [1–5]. Their practical implications in system/software design and verification have been discussed extensively. Recent development on distributed systems show that there are a family of fairness notions, including a newly formulated fairness notion named strong global fairness [6], which are crucial for designing self-stabilizing distributed algorithms [6–9]. The population protocol model has emerged as an elegant computation paradigm for describing mobile ad hoc networks [7]. Such networks consist of multiple mobile nodes which interact with each other to carry out a computation. Application domains of the protocols include wireless sensor networks and biological computers. The interaction events among the nodes are subject to fairness constraints. One essential property of population protocols is that all nodes must eventually converge to the correct output values (or configurations). A number of population protocols have been proposed and studied [6–9]. Fairness plays an important role in the protocols. For instance, in [6] it was shown that with the help of an eventual leader detector (see details in Section 2), self-stabilizing algorithms can be developed to handle two natural classes of network graphs: complete graphs and rings. The algorithm for the complete graph works with *event-level strong fairness*, whereas the algorithm for rings, only works with *strong global fairness*. It has been proved that with only event-level strong fairness or weaker, uniform self-stabilizing leader election in rings is impossible [6]. Because the algorithms are designed to function with fairness, model checking of (implementations of) the algorithms thus must be carried out with the respective fairness constraints.

Existing model checkers are ineffective with respect to fairness. One way to apply existing model checkers for verification with fairness constraints is to re-formulate the property so that fairness constraints become premises of the property. A liveness property ϕ is thus verified by showing the truth value of the following formula:

$$fairness\ assumptions \implies \phi$$

This practice is, though flexible, deficient for the reason below. Model checking is PSPACE-complete in the size of the formula. In particular, automata-based model checking relies on constructing a Büchi automaton from the LTL formula. The size of the Büchi automaton is exponential to the size of the formulas. Thus, it is infeasible to handle large formulas, whereas a typical system may have multiple or many fairness constraints. There has been dedicated research on handling large LTL formulae [10, 11].

In [12], Pang et al. applied the SPIN model checker to establish the correctness of a family of population protocols. Only protocols relying on a notion of weak fairness operating on very small networks were verified because of the problems discussed above. Protocols relying on a notion of stronger fairness (e.g., strong fairness or strong global fairness) are beyond the capability of SPIN even for the smallest network (e.g., with three nodes). It is important to develop an alternative approach and toolkit which can handle larger networks because real counterexamples may only be present in larger networks, as shown in Section 5.

An alternative method is to design specialized verification algorithms which take fairness into account while performing model checking. The focus of existing model checkers has been on process-level weak fairness, which, informally speaking, states that every process shall make infinite progress if always possible (see detailed explanation in Section 2). For instance, SPIN has implemented a model checking algorithm which handles this kind of fairness. The idea is to copy the global reachability graph $K + 2$ times (for K processes) so as to give each process a fair chance to progress. Process-level strong fairness is not supported because of its complexity. It has been shown that process-level fairness may not be sufficient, e.g., for population protocols.

In this work, we enhance a toolkit PAT (process analysis toolkit) to support on-the-fly model checking with a variety of fairness constraints including process-level weak/strong fairness, event-level weak/strong fairness, strong global fairness, etc. Our model checking algorithm unifies previous work on model checking based on finding strongly connected components (SCC) and extends it to handle newly proposed notions like strong global fairness and event-labeled fairness efficiently. After enhancement, PAT supports a rich modeling language and an easy way of applying fairness. A PAT user can choose one of the fairness constraints and apply it to the whole system. Using PAT, we identified *previously unknown bugs* in the implementation of population protocols [6, 13]. For experiments, we compare PAT with SPIN over recent distributed algorithms as well as benchmark systems. We show that our approach handles fairness more flexibly and efficiently.

The main contribution of this work is the enhancement of PAT to support modeling, simulation and model checking of distributed algorithms with fairness constraints. PAT is applied to a number of case studies including recently proposed population protocols and have successfully found previously unknown bugs. Another contribution is that we extend previously developed model checking algorithm with weak/strong fairness to model check with a variety of fairness, including

strong global fairness and event-labeled fairness.

The remainder of the article is organized as follows. Section 2 shows a concrete motivating example and our computational model, together with a family of fairness constraints. Section 3 develops necessary theories for model checking. Section 4 presents the algorithm for verification with fairness. Section 5 presents the PAT model checking system, and experiment results. Section 6 discusses the related work. Section 7 concludes the paper.

2 Background

In this section, we start with introducing a concrete motivating example and then present our computational model as well as formal definitions of fairness.

2.1 Motivating examples

Leader election is a fundamental problem in distributed systems. The problem is easily solved with the help of a central coordinator. Nonetheless, there may not be a central coordinator in domains like wireless sensor networks. Self-stabilizing algorithms do not require initialization in order to operate correctly and can recover from transient faults that obliterate all state information in the system. In [6, 7, 13], a number of algorithms have been proposed for the self-stabilizing leader election problem. In particular, Fischer and Jiang propose a self-stabilizing algorithm for ring networks [6], which guarantees that one and only one leader will be eventually elected given any initial configuration. The algorithm relies on two assumptions. One is that the system satisfies a rather strong fairness constraint called *strong global fairness* [6]. The other one is that there exists a leader detector. The detector is a diagnostic device which tests network nodes for certain information, which is required to *eventually* (not necessarily *immediately*) detect the presence/absence of a leader.

Figure 1 presents the PAT model of the algorithm presented in [6]. PAT supports a modeling language which mixes high-level specification language features (e.g., deterministic or nondeterministic choice, alphabetized parallel, interleaving, interrupt, etc.) with low-level programming language features (arrays, while, if-then-else, etc.), so that the users are offered with great expressiveness as well as flexibility. We skip the details of the language as it is not the focus of this work. Interested readers are referred to [14] for its design principles. Lines 1 – 6 of Fig. 1 declares constants and variables of the model. In particular, constant N (of value 3) to models the network size, i.e., the number of network nodes; *correct* is an indicator which is of value 1 if and only if the leader detector has started detecting correctly. *guess* is a Boolean flag representing the current diagnostic result of the detector, i.e., *true* for the presence of a leader and *false* for the absence. Array *ld* (short for leader), *bu* (short for bullet), and *sh* (short for

```
1. #define N 3;
2. var correct = 0;
3. var guess = false;
4. var ld[N];
5. var bu[N];
6. var sh[N];
7. #define oneLeader (ld[0] + ld[1] + ld[2] == 1);
8. #define exist (correct==0 && guess) || (correct!=0 && ld[0]+ld[1]+ld[2]>0);

9. Node(i) = case {
10.        !exist :
11.                rule1.i.(i+1)%N{bu[i]=1; ld[i]=1; sh[i]=1;} -> Node(i)
12.        ld[i] == 0 && sh[i] == 1 && exist :
13.                rule2.i.(i+1)%N{ld[i]=0;sh[i]=0;bu[(i+1)%N]=0;sh[(i+1)%N]=1;} -> Node(i)
14.        ld[i] == 1 && sh[i] == 1 && exist :
15.                rule3.i.(i+1)%N{bu[i] = 1; ld[i] = 1; sh[i] = 0; bu[(i+1)%N] = 0;
                                   sh[(i+1)%N] = 1;} -> Node(i)
16.        ld[i] == 1 && sh[i] == 0 && bu[(i+1)%N] == 0 && exist :
17.                rule4.i.(i+1)%N{bu[i] = 1; ld[i] = 1; sh[i]=0; bu[(i+1)%N] = 0;} -> Node(i)
18.        sh[i] == 0 && bu[(i+1)\%N] == 1 && exist :
19.                rule5.i.(i+1)%N{bu[i] = 1; ld[i] = 0; sh[i] = 0; bu[(i+1)%N] = 0;} -> Node(i)
20. };
21. Detector() = oracle{correct = 1;} -> Detector()
22.                [] guess1{guess = false;} -> Detector()
23.                [] guess2{guess = true;} -> Detector();
24. LeaderElection() = Init(); (Detector() ||| (|||x: {0..N-1 } @ Process(x)));
25. #assert LeaderElection() |= <>[] oneLeader;
```

Fig. 1 Leader election protocol for rings

shield) model the status of each node. For instance, the i bit of $ld[N]$ tells whether the ith node is a leader or not. We skip the intuition behind these variables and refer the readers to [6].

Lines 7 and 8 declares two propositions. A proposition is a synonymy of a Boolean formula, which may be used in the model or the assertions. In particular, proposition *oneLeader* is defined to be true if and only if there is one and only one leader (i.e., $ld[0] + ld[1] + ld[2]$ is 1). Proposition *exist* denotes the leader detector *believes* there is a leader in the network. That is, if it has not starting detecting correctly (i.e., *correct* is 0), then it guesses there is a leader (i.e., *guess* is 1); otherwise if it detects correctly, then there is a leader in the network (i.e., $ld[0] + ld[1] + ld[2] > 0$).

Lines 9 – 24 define three processes in the form of equations, which capture the essence of the algorithm. In particular, Lines 9 – 20 define process $Node(i)$ which models the behaviors of a network node. Every time there is an interaction event in the network, the initiator and responder must update themselves according to a set of five pre-defined rules, e.g., to become a leader if there is no leader (according to the leader detector), to stop being a leader if both the initiator and the responder are leaders, etc. The rules are specified using a case statement of the following form,

case{
$b_0 : P_0$
$b_1 : P_1$
$\cdots$
$b_k : P_k$
}

where b_i is a condition and P_i is a process expression. The conditions are evaluated in sequence until the first one which evaluates to true, then the corresponding process is executed. For instance, if the leader detector believes there is no leader in the network, then the condition at Line 10 is true, and therefore the process expression at Line 11 is executed. The process expression takes the form of $e\{\text{program}\} \rightarrow P$ where e is an event name, program is a sequential program attached with the event and P is a process. Intuitively, the event is taken and at the same time the program is executed atomically together. Next, process P is executed. In general, an event may be attached with a sequential program (with loops, branches, etc.), which executes atomically. Equivalently, *oracle* can be viewed as a label of the program (for easy referencing). For instance, if the process expression at Line 11 is taken, event $\text{rule1}.i.(i + 1)\%N$ is generated. At the same time, $bu[i]$, $id[i]$ and $sh[i]$ are all set to be 1. Afterwards, process $Node(i)$ is invoked so that the network node repeats from the beginning. We skip the details of the rules and refer the readers to [6].

Lines 21 – 23 define the leader *Detector*, where [] is the unconditional choice operator borrowed from the classic CSP [15]. The detector has three choices. It may be enlightened (Line 21) through the event *oracle* and then detects correctly ever-after. Or, it may take a guess, randomly stating that there is a leader (Line 23) or there is not (Line 22). We remark that there is no guarantee that the detector will eventually detect correctly (e.g., the event *oracle* may never happen) unless fairness is applied.

Line 24 models the leader election algorithm as process *LeaderElection*. The algorithm firstly invokes process *Init*, which initializes the system in every possible configuration, e.g., each element in the arrays may be either assigned 0 or 1. We omit the details of *Init* as it can be constructed straightforwardly using the unconditional choice operator []. After the initialization, the system behaves as the interleaving (modeled by operator |||) of the leader detector and all the network nodes.

The property of interest to all leader election algorithms is $\diamond\Box$*oneLeader* (defined as an assertion at Line 25), where $\diamond$ and $\Box$ are modal operators which read as "eventually" and "always" respectively. In PAT, we support the state/event LTL [16]. We remark that the assertion is false with no fairness, process-level weak/strong fairness or event-level weak/strong fairness. Counterexamples can be generated efficiently for each of the cases using PAT. The assertion is true with strong global fairness, as proved by PAT for bounded networks. However, verifying the original algorithm is only as useful as confirming the theorem proved. In order to show that an implementation of the algorithm satisfies the property, it must be verified with strong global fairness. To the best of our knowledge, PAT is the only tool which is capable of finding bugs in such a setting.

2.2 Models and definitions

We present the approaches in the setting of labeled transition systems (LTS). Models in PAT are interpreted as LTSs implicitly by defining a complete set of operational semantics. Let e be an event, 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.

Definition 1 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.

We write $s \xrightarrow{e} s'$ to denote that (s, e, s') is a transition in $\rightarrow$. $enabledEvt(s)$ is the set of enabled events at s, i.e., e is in $enabledEvt(s)$ if and only if there exists 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 $\{e\}$.

Because our targets are nonterminating distributed systems, and fairness affects infinite not finite system behaviors, we focus on infinite system executions in the following. Finite behaviors are extended to infinite ones by appending infinitely many idling events at the rear. Given an LTS $\mathcal{L} = (S, init, \rightarrow)$, an execution is an infinite sequence of alternating states and events $E = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ where $s_0 = init$ and for all i $s_i \xrightarrow{e_i} s_{i+1}$. 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. Given a property ϕ, verification with fairness is to verify whether all fair executions of the system satisfy ϕ. In the following, we review a variety of different fairness constraints and illustrate their differences using examples.

Definition 2 Let $E = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ be an execution. E satisfies event-level weak fairness, if and only if for every event 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.

Event-level weak fairness (EWF) [2] states that if an event becomes enabled forever after some steps, then it must be engaged infinitely often. An equivalent formulation is that every computation should contain infinitely many positions at which e is disabled or has just been engaged. The latter is known as justice condition [17]. Intuitively, it means that an enabled event shall not be ignored infinitely. Or equivalently some state must be visited infinitely often (e.g., accepting states in Büchi automata).

Definition 3 Let $E = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ be an execution. 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, i.e., $\diamond\Box p$ is enabled $\Rightarrow \Box \diamond p$ is engaged.

Process-level weak fairness (PWF) 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 (or justice condition) has been well studied and verification with weak fairness has been supported to some extent, e.g., process-level weak fairness is supported by the SPIN model checker [18]. Given the LTS in Fig. 2(a), the property $\Box \diamond$ a is engaged is true with event-level weak fairness. Action a is always enabled and, hence, by definition it must be infinitely often engaged. The property is, however, false with no fairness or process-level weak fairness. The reason that it is false with process-level weak fairness is that the process W may make progress infinitely (by repeatedly engaging in b) without ever engaging in event a. Alternatively, if the system is modeled using two processes as shown in Fig. 2(b), $\Box \diamond$ a is engaged becomes true with process-level weak fairness (or event-level weak fairness) because both processes must make infinite progress and therefore both a and b must be engaged infinitely. This example suggests that, different from process-level weak fairness, event-level weak fairness is not related to the system structure. In general, process-level fairness may be viewed a special case of event-level fairness. By a simple argument, it can be shown that process-level weak fairness can be achieved by labeling all events in a process with the same name and applying event-level weak fairness. For simplicity, in the rest of the paper, we write $\Box\diamond$ a (or $\diamond\Box$ a) to denote $\Box \diamond$ a is engaged (or $\diamond\Box$ a is engaged) unless otherwise stated.

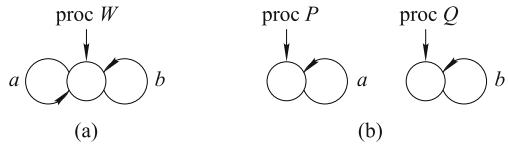


Fig. 2 Event-level vs. process-level weak fairness

Definition 4 Let $E = \langle s_0, e_0, s_1, e_1, \ldots \rangle$ be an execution. E satisfies event-level strong fairness if and only if, for every event e, if e is infinitely often enabled, then $e = e_i$ for infinitely many i, i.e., $\Box \diamond$ e is enabled $\Rightarrow \Box \diamond$ e is engaged.

Event-level strong fairness (ESF) has been identified by different researchers. It is named *strong fairness* in [19] (by contrast to weak fairness defined above). In [6], it is named strong local fairness (in comparison to strong global fair-

ness defined below). It is also known as *compassion* condition [20]. Event-level strong fairness states that if an event 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. Given the LTS in Fig. 3(a), the property $\Box \diamond\ b$ is false with event-level weak fairness but true with event-level strong fairness. The reason is that b is not always enabled (i.e., it is disabled at the left state) and thus the system is allowed to always take the c branch with event-level weak fairness. It is infinitely often enabled, and thus, the system must engage in b infinitely with event-level strong fairness.

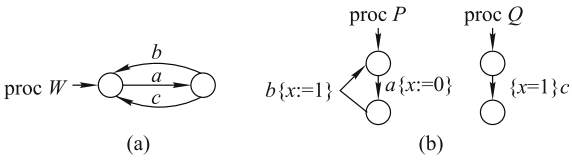


Fig. 3 Event-level vs. process-level strong fairness

Definition 5 Let $E = \langle s_0, e_0, s_1, e_1, \ldots\rangle$ be an execution. E satisfies process-level strong fairness 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, i.e., $\Box \diamond\ p$ is enabled $\Rightarrow \Box \diamond\ p$ is engaged.

The process-level correspondence is process-level strong fairness (PSF). Intuitively, process-level strong fairness means that if a process is repeatedly enabled, it must eventually make some progress. Given the LTS in Fig. 3(b), the property $\Box \diamond\ c$ is false with process-level weak fairness but true with process-level strong fairness. The reason is that event c is guarded by condition $x = 1$ and therefore is only repeatedly enabled.

Verification with (event-level/process-level) strong fairness (or compassion condition) has been discussed previously, e.g., in the setting of Streett automata [21,22], fair discrete systems [23] or programming codes [24]. Nonetheless, there is few established tool support for formal verification with strong fairness [1] to the best of our knowledge. The main reason is the computational complexity. For instance, it is claimed too expensive to support in SPIN (page 182 of [18]). We, however, show that reasonably efficient model checking with strong fairness can be achieved (refer to experiment results in Section 5).

Definition 6 Let $E = \langle s_0, e_0, s_1, e_1, \ldots\rangle$ be an execution. E satisfies strong global fairness 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.

Strong global fairness (SGF) was suggested in [6]. 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[1)]. 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. *It can be shown that strong global fairness coincides event-level strong fairness when every transition is labeled with a different event*. This observation implies that we can uniquely label all transitions with different events and then apply model checking algorithm for strong fairness to deal with global fairness. We show however, model checking with global fairness can be solved using a more efficient approach.

Figure 4 illustrates the difference with two examples. Under event-level strong fairness, state 2 in Fig. 4(a) may never be visited because all events are engaged infinitely often if the left loop is taken infinitely. As a result, property $\Box \diamond$ state 2 is false. Under strong global fairness, all states in Fig. 4(a) must be visited infinitely often and therefore $\Box \diamond$ state 2 is true. Figure 4(b) illustrates their difference when there is non-determinism. Both transitions labeled a must be taken infinitely with strong global fairness, which is not necessary with event-level strong fairness or weak fairness. Thus, property $\Box \diamond\ b$ is true only with strong global fairness. Many population protocols rely on strong global fairness, e.g., protocols presented in [6,7]. As far as the authors know, there are no previous works on model checking with strong global fairness.

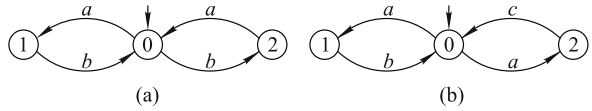


Fig. 4 Strong global fairness

3 Loop/SCC searching

Given an LTS $\mathcal{L}$ and an LTL formula ϕ, model checking

[1)] The definition in [6] is for unlabeled transition systems. We slightly changed it so as to suit the setting of LTS.

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 with fairness is to search for an infinite execution which is accepted by the Büchi automaton and at the same time satisfies the fairness constraints. In the following, we write $\mathcal{L} \vDash \phi$ to mean that the LTS satisfies the property with no fairness, i.e., every execution of $\mathcal{L}$ satisfies ϕ. We write $\mathcal{L} \vDash_{\text{EWF}} \phi$ ($\mathcal{L} \vDash_{\text{PWF}} \phi$) to mean that $\mathcal{L}$ satisfies ϕ with event-level (process-level) weak fairness; $\mathcal{L} \vDash_{\text{ESF}} \phi$ ($\mathcal{L} \vDash_{\text{PSF}} \phi$) to mean that $\mathcal{L}$ satisfies ϕ with event-level (process-level) strong fairness, and $\mathcal{L} \vDash_{\text{SGF}} \phi$ to mean that $\mathcal{L}$ satisfies ϕ with strong global fairness.

We assume that $\mathcal{L}$ contains only finite states. By a simple argument, it can be shown that $\mathcal{L}$ contains an infinite execution if and only if there exists a loop. A lasso in the product of $\mathcal{L}$ and $\mathcal{B}$, denoted as R_i^j, is a sequence of alternating states/events

$$\langle (s_0, b_0), e_0, \ldots, (s_i, b_i), e_i, \ldots, (s_j, b_j), e_j, (s_{j+1}, b_{j+1}) \rangle,$$

such that s_i is a state of $\mathcal{L}$, b_i is a state of $\mathcal{B}$, $s_i = s_{j+1}$ and $b_i = b_{j+1}$. We skip the details on constructing the product and refer the readers to [18]. R_i^j is accepting if and only if the sequence $\langle b_0, e_0, b_1, e_1, \ldots, b_k, e_k, \ldots \rangle$ is accepting to $\mathcal{B}$, i.e., the sequence visits at least one accepting state of $\mathcal{B}$ infinitely often. R_i^j is fair with certain notion of fairness if and only if the sequence $\langle s_0, e_0, s_1, e_1, \ldots, s_k, e_k, \ldots \rangle$ is. Furthermore, we define the following sets:

$$\begin{aligned}
&\mathit{alwaysEvt}(R_i^j) = \{e | \forall k : \{i, \ldots, j\}\ e \in \mathit{enabled}(s_k)\},\\
&\mathit{alwaysPro}(R_i^j) = \{p | \forall k : \{i, \ldots, j\}\ p \in \mathit{enabledPro}(s_k)\},\\
&\mathit{onceEvt}(R_i^j) = \{e | \exists k : \{i, \ldots, j\}\ e \in \mathit{enabled}(s_k)\},\\
&\mathit{oncePro}(R_i^j) = \{p | \exists k : \{i, \ldots, j\}\ p \in \mathit{enabledPro}(s_k)\},\\
&\mathit{engagedEvt}(R_i^j) = \{e | \exists k : \{i, \ldots, j\}\ e = e_k\},\\
&\mathit{engagedPro}(R_i^j) = \{p | \exists k : \{i, \ldots, j\}\ p \in \\
&\qquad\qquad \mathit{engagedPro}(s_k, e_k, s_{k+1})\}.
\end{aligned}$$

Intuitively, set $\mathit{alwaysEvt}(R_i^j)$ ($\mathit{alwaysPro}(R_i^j)$) is the set of events (processes) which are always enabled during the loop. Set $\mathit{onceEvt}(R_i^j)$ ($\mathit{oncePro}(R_i^j)$) is the set of events (processes) which are enabled at least once during the loop. Set $\mathit{engagedEvt}(R_i^j)$ ($\mathit{engagedPro}(R_i^j)$) is the set of events (processes) which are engaged during the loop.

Lemma 1 Let $\mathcal{L} = (S, \mathit{init}, \rightarrow)$ be an LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of an LTL formula ϕ.

- $\mathcal{L} \vDash_{\text{EWF}} \phi$ if and only if there does not exist R_i^j s.t. $\mathit{alwaysEvt}(R_i^j) \subseteq \mathit{engagedEvt}(R_i^j)$ and R_i^j *is accepting*.
- $\mathcal{L} \vDash_{\text{PWF}} \phi$ if and only if there does not exist R_i^j s.t. $\mathit{alwaysPro}(R_i^j) \subseteq \mathit{engagedPro}(R_i^j)$ and R_i^j *is accepting*.
- $\mathcal{L} \vDash_{\text{ESF}} \phi$ if and only if, there does not exist R_i^j s.t. $\mathit{onceEvt}(R_i^j) \subseteq \mathit{engagedEvt}(R_i^j)$ and R_i^j *is accepting*.
- $\mathcal{L} \vDash_{\text{PSF}} \phi$ if and only if there does not exist R_i^j s.t. $\mathit{oncePro}(R_i^j) \subseteq \mathit{engagedPro}(R_i^j)$ and R_i^j *is accepting*.

The lemma can be proved straightforwardly by contradiction. By the lemma, a system fails the property with certain fairness if and only if there exists a loop which satisfies the fairness and at the same time fails the property. Modeling checking with fairness is hence reduced to loop searching.

Given a transition system, a subgraph is strongly connected if there is a path from each state in the subgraph to every other state in the subgraph. A SCC is a maximal strongly connected subgraph. Given the product of $\mathcal{L}$ and $\mathcal{B}$, let S be a set of states which, together with the transitions among them, forms a strongly connected subgraph. In an abuse of notations, we refer to S as the strongly connected subgraph in the following. To further abuse notations, we write $\mathit{alwaysEvt}(S)$ ($\mathit{alwaysPro}(S)$, $\mathit{onceEvt}(S)$, $\mathit{oncePro}(S)$, $\mathit{engagedEvt}(S)$ or $\mathit{engagedPro}(S)$) to denote the set obtained by applying the function to a loop which traverses through all states of the subgraph. We say S is accepting if and only if there exists one state (s, b) in S such that b is an accepting state of $\mathcal{B}$.

Lemma 2 Let $\mathcal{L}$ be an LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of an LTL formula ϕ.

- $\mathcal{L} \vDash_{\text{ESF}} \phi$ if and only if there does not exist a reachable strongly connected subgraph S in the product of $\mathcal{L}$ and $\mathcal{B}$ such that S is accepting and $\mathit{onceEvt}(S) \subseteq \mathit{engagedEvt}(S)$.
- $\mathcal{L} \vDash_{\text{PSF}} \phi$ if and only if there does not exist a reachable strongly connected subgraph S in the product of $\mathcal{L}$ and $\mathcal{B}$ such that S is accepting and $\mathit{oncePro}(S) \subseteq \mathit{engagedPro}(S)$.

The above lemma can be proved by a simple argument. It shows that model checking with strong fairness can be reduced to *strongly connected subgraph* (not SCC) searching.

Lemma 3 Let $\mathcal{L}$ be an LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of an LTL formula ϕ.

- $\mathcal{L} \vDash_{\text{EWF}} \phi$ if and only if there does not exist a reachable SCC S in the product of $\mathcal{L}$ and $\mathcal{B}$ such that S is

accepting and $alwaysEvt(S) \subseteq engagedEvt(S)$.

- $\mathcal{L} \vDash_{\text{PWF}} \phi$ if and only if there does not exist a reachable SCC S in the product of $\mathcal{L}$ and $\mathcal{B}$ such that S is accepting and $alwaysPro(S) \subseteq engagedPro(S)$.

The proof of the lemma follows previous results (see for example Chapter 6 of [25]). In the following, we argue the event-level weak fairness part of the lemma and remark that the other part can be proved similarly. It can be shown that a system fails ϕ with event-level weak fairness if and only if there exists one reachable strongly connected subgraph C such that C is accepting and $alwaysEvt(C) \subseteq engagedEvt(C)$. Hence, it is sufficient to show that there exists such a C if and only if there exists a reachable SCC S such that S is accepting and $alwaysEvt(S) \subseteq engagedEvt(S)$. If there exists such an SCC S, then we simply let C be S. If there exists such subgraph C, the SCC S which contains C is accepting and satisfies $alwaysEvt(S) \subseteq engagedEvt(S)$ since $alwaysEvt(S) \subseteq alwaysEvt(C)$ and $engagedEvt(C) \subseteq engagedEvt(S)$. This concludes the proof.

The lemma shows that model checking with weak fairness can be reduced to SCC searching. The following lemma reduces model checking with strong global fairness to searching for a terminal SCC in $\mathcal{L}$. An SCC is terminal if and only if any transition starting from a state in the SCC must end with a state in the SCC. Notice that a terminal SCC in the product of $\mathcal{L}$ and $\mathcal{B}$ may not be constituted by a terminal SCC in $\mathcal{L}$.

Lemma 4 Let $\mathcal{L}$ be an LTS; $\mathcal{B}$ be a Büchi automaton equivalent to the negation of an LTL formula ϕ. $\mathcal{L} \vDash_{\text{SGF}} \phi$ if and only if there does not exist a reachable SCC S in the product of $\mathcal{L}$ and $\mathcal{B}$ such that S is accepting and for all $(s,b) \in S$, if there exists e and s' such that $s \xrightarrow{e} s'$, then there exists $(s',b') \in S$ such that $(s,b) \xrightarrow{e} (s',b')$.

Proof $\mathcal{L}$ fails ϕ with strong global fairness if and only if there exists R_i^j in the product of $\mathcal{L}$ and $\mathcal{B}$ such that R_i^j satisfies strong global fairness and R_i^j is accepting. $\mathcal{L}$ fails ϕ with strong global fairness if and only if there exists a reachable strongly connected subgraph C such that a loop which travels through all states/transitions in C (and no other states/transitions) infinitely often satisfies strong global fairness and C is accepting. Hence, it is sufficient to show that there exists such a subgraph C if and only if there exists an SCC S such that S which satisfies the constraint.

if: This is proved trivially.

only if: Assume that there exists such a subgraph C. Let $x(C) = \{s | \exists b\ (s,b) \in C\}$ be the states of $\mathcal{L}$ which constitute C and $t(C) = \{(s,e,s') | s \in x(C) \wedge s' \in x(C) \wedge \exists (s,b),(s,b') : C\ (s,b) \xrightarrow{e} (s',b')\}$ be the transitions of $\mathcal{L}$ which constitute the strongly connected subgraph. By contradiction, it can be shown that $x(C)$ (together with the transitions in $t(C)$) forms one terminal SCC in $\mathcal{L}$. Let S be the SCC containing C. It can be shown that $x(S)$ (together with the transitions in $t(S)$) forms the same terminal SCC. Therefore, S must satisfy the constraint.

4 Modeling checking with fairness

There are two groups of methods for loop searching. One is based on nested depth-first-search (DFS) and the other one is based on identifying SCCs. Nested DFS has been implemented in SPIN. The basic idea is to perform a first DFS to reach a target state (i.e., an accepting state in the setting of Büchi automata) and then perform a second DFS from that state to check whether it is reachable from itself. It has been shown the nested DFS works efficiently for model checking with no fairness [18]. Nonetheless, it is not suitable for verification with fairness because whether an execution is fair depends on the whole path instead of one state. In recent years, model checking based on SCC has been re-investigated and it has been shown that it yields comparable performance [26]. In this work, we extend the existing SCC-based model checking algorithms to cope with different notions of fairness. The algorithm is inspired by early work on emptiness check of Streett automata [22].

4.1 A unified algorithm

Figure 5 presents the algorithm. It is based on Tarjan's algorithm for identifying SCCs (which is linear time in the number of graph edges [27]). It searches for fair strongly connected subgraph on-the-fly. The basic idea is to identify one SCC at a time and then check whether it is fair or not. If it is,

```
procedure mc (S,T,F)
1.  visited =Ø;
2.  while there are states in S but not in visited
3.        let scc_states = findSCC (S,T);
4.        visited = visited ∪ scc_states;
5.        if scc_states is accepting
6.              prunedSCC = prune(scc_states,T,F);
7.              if #prunedSCC == #scc_states
8.                    return false;
9.              else if mc(prunedSCC,T,F) == false
10.                   return false;
11.             end if
12.       end if
13. end while
14. return true;
```

Fig. 5 Algorithm for model checking with fairness

the search is over. Otherwise, the SCC is partitioned into multiple smaller strongly connected subgraphs, which are then checked recursively one by one. Figure 6 presents a running example, i.e., the product of an LTS and a Büchi automaton. Notice that state 2 is an accepting state.

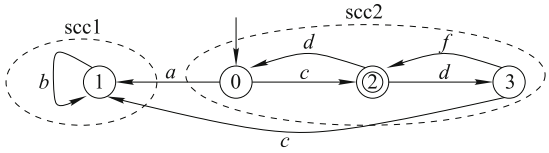


Fig. 6 Model checking example

Assume for now that S is the set of states and T is the set of transitions. The method takes three inputs, i.e., S, T and a fairness type F (of value either EWF, PWF, ESF, PSF or SGF). At Line 1, a set *visited*, which stores the set of visited states, is initialized to be empty. The main loop is from Lines 2–13. At Line 3 Tarjan's algorithm (improved and implemented as method findSCC(S, T)) is used to identify one SCC within S and T. Identifying S and T for compositional systems or software programs requires reachability analysis. In order to perform on-the-fly verification, findSCC is designed in such a way that if no S and T are given, it will explore states and transitions on-the-fly until one SCC is identified. We skip the details of findSCC as it largely resembles the algorithm presented in [21]. Given the LTS presented in Fig. 6, findSCC identifies two SCCs, i.e., scc1 which is composed of state 1 only and scc2 which is composed of state 0, 2, and 3. The order in which SCCs are found is irrelevant to the correctness of the algorithm.

At Line 4, we mark scc_states as visited so that the SCC is not examined again. At Line 5, we check whether the SCC is accepting. Only accepting SCCs may contain a bad loop which constitutes a counterexample. For instance, scc1 is not accepting and therefore ignored. Function *prune* (at Line 6) is used to prune *bad states* from the SCC. Bad states are the reasons why the SCC is not fair. For instance, state 0 (where the event a is enabled) is a bad state in scc2 with event-based strong fairness because event a is never engaged in scc2 (i.e., a is not in *engagedEvt*(ssc2)). State 3 is a bad state with strong global fairness because the step from state 3 to state 1 via c is not part of the SCC. The intuition behind the pruning is that there may be a fair strongly connected subgraph in the remaining states after eliminating the bad states. By simply modifying the *prune* method, the algorithm can be used to handle different fairness constraints. Refer to details in Section 2.

If the SCC does satisfy the fairness assumption, no state is pruned and thus the size of the SCC remains the same (Line 7 of Fig. 5). In such a case, a fair loop which traverses all states/transitions in the SCC is generated as a counterexample and we conclude that the property is not true at Line 8. We skip the details on generating the path in this paper and remark that it could be a non-trivial task (see [23]). If some states have been pruned, a recursive call is made to check whether there is a fair strongly connected subgraph within the remaining states. The recursive call terminates in two ways. One is that a fair subgraph is found (at Line 9) and the other is all states are pruned (at Line 14). If the recursive call returns true, there is no subgraph satisfying the fairness condition and we continue with another SCC until all states are checked.

4.2 Coping with different notions of fairness

In this section, we show how to customize the *prune* function so as to handle different fairness constraints. Let S be a strongly connected subgraph and T be the transition among the states. The following defines the pruning methods for event-based weak fairness.

$$\begin{aligned} &prune(S, T, \text{EWF}) = S, \\ &\qquad \text{if } alwaysEvt(S) \subseteq engagedEvt(S); \\ &prune(S, T, \text{EWF}) = \emptyset, \\ &\qquad \text{otherwise.} \end{aligned}$$

If there exists an event e which is always enabled (i.e., $e \in alwaysEvt(S)$) but never engaged (i.e., e not in $engagedEvt(S)$), by definition S does not satisfy event-level weak fairness. If an SCC does not satisfy event-level weak fairness, none of its subgraphs do, because e is always enabled in any of its subgraphs but never engaged. As a result, either all states are pruned or none of them is. Similarly, the following defines the pruning function for process-level weak fairness.

$$\begin{aligned} &prune(S, T, \text{PWF}) = S, \\ &\qquad \text{if } alwaysPro(S) \subseteq engagedPro(S); \\ &prune(S, T, \text{PWF}) = \emptyset, \\ &\qquad \text{otherwise.} \end{aligned}$$

In the case of event-level (process-level) strong fairness, a state is pruned if and only if there is an event (process) enabled at this state but never engaged in the subgraph. By pruning the state, the event (process) may become never enabled and therefore not required to be engaged. The following defines the pruning function for event-level and process-level

strong fairness.

$$prune(S, T, \text{ESF}) = \{s : S \mid enabledEvt(s) \subseteq engagedEvt(S)\};$$
$$prune(S, T, \text{PSF}) = \{s : S \mid enabledPro(s) \subseteq engagedPro(S)\}.$$

By Lemma 4, an SCC may constitute a counterexample to a property with strong global fairness if and only if the SCC satisfies strong global fairness and is accepting. Therefore, we prune all states if it fails strong global fairness.

$$prune(S, T, \text{SGF}) = S, \quad \text{if } \forall (s, b) \in S \ \ \forall e, s' \quad s \xrightarrow{e} s' \Rightarrow s' \in S;$$
$$prune(S, T, \text{SGF}) = \emptyset, \quad \text{otherwise.}$$

We remark that the time complexity of the *prune* functions are all linear in the number of edges of the SCC. Given the example in Fig. 6, with event-level strong fairness, state 0 is pruned from scc2 because $enabledEvt(\text{state } 0) = \{a, c\} \not\subseteq engagedEvt(\text{scc2})$. After that the only remaining strongly connected subgraph contains states 2 and 3, now state 3 where c is enabled is considered as a bad state because c is not engaged in the subgraph. State 2 is then pruned for being a trivial strongly connected subgraph which fails event-level strong fairness.

4.3 Complexity and soundness

The time complexity for verification with no fairness, event-level or process-level weak fairness or strong global fairness are similar, i.e., all linear in the number of transitions in the product automaton. 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 an SCC. SPIN's model checking algorithm with 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. In comparison, our algorithm is less expensive for weak fairness. This becomes evident by the experiment results presented in Section 5. Verification with 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 find-SCC 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 worse case scenario is rare in practice. Instead of performing detailed complexity analysis (see the discussion presented in [22]), we illustrate the performance of our algorithm using real systems in Section 5.

Next, we argue the total correctness of the algorithm. The algorithm is terminating because by assumption, the number of states is finite, and the number of visited states and pruned states are monotonically increasing.

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). $\mathcal{L} \models_F \phi$ if and only if the algorithm returns true.

Proof

Case EWF: By Lemma 1, $\mathcal{L} \models_{\text{EWF}} \phi$ if and only if there does not exist an SCC S such that $alwaysEvt(S) \subseteq engagedEvt(S)$ and S is accepting. Given any SCC S, the algorithm returns false if and only if it does so at Line 8 because the recursive call at Line 9 always returns true (by the definition of $prune(S, T, \text{EWF})$). Therefore, it returns false if and only if S is accepting (so that the condition at Line 5 is true) and $alwaysEvt(S) \subseteq engagedEvt(S)$ (so that the condition at Line 7 is true). If there does not exist such an SCC, the algorithm returns true. If the algorithm returns true, there must not be such an SCC. Therefore, $\mathcal{L} \models_{\text{EWF}} \phi$ if and only if the algorithm returns true.

Case PWF: Similar to the above.

Case ESF: By Lemma 3, $\mathcal{L} \models_{\text{ESF}} \phi$ if and only if there does not exist a strongly connected subgraph C such that $onceEvt(C) \subseteq engagedEvt(C)$ and C is accepting. If C itself is an SCC, it must be found (by the correctness of Tarjan's algorithm and function $prune(S, T, \text{ESF})$) and the algorithm returns false if C is accepting. If it is contained in one (and only one) SCC, by the correctness of $prune(S, T, \text{ESF})$, its states are never pruned. As a result, it is identified when all other states in the SCC are pruned or a fair strongly connected subgraph containing all its states is identified. In either case, the algorithm returns false if and only if such a fair strongly connected subgraph is found. Equivalently, it returns true if and only if there are no such subgraphs. Therefore, $\mathcal{L} \models_{\text{ESF}} \phi$ if and only if the algorithm returns true.

Case PSF: Similar to the above.

Case SGF: By Lemma 4, $\mathcal{L} \models_{\text{SGF}} \phi$ if and only if there does not exist an SCC S such that S satisfies strong global fairness and S is accepting. The algorithm returns false if and only if it is at Line 8 because the recursive call at Line 9 always

returns true (by the definition of $prune(S, T, \text{SGF})$). By definition of $prune(S, T, \text{SGF})$, the control reaches Line 8 if and only if the SCC is terminal and is accepting. Thus, $\mathcal{L} \vDash_{\text{SGF}} \phi$ if and only if it returns true.

5 Implementation and experiments

The main contribution of the work is the model checking with fairness capability in the PAT model checker. 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 with fairness. It has three main components. The editor features all standard text editing functionalities. The simulator allows users to interactively drive the system execution. 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.

Table 1 summarizes the verification statistics on recently developed population protocols. The experiments are made on a 3.0 GHz Pentium IV CPU and 2 GB memory executing SPIN 4.3. Notice that "–" means either out of memory or more than four hours; column "Size" represents the number of network nodes; result "Yes" means there is no counterexample; result "No" means there is a counterexample; the verification times for PAT and SPIN are measured in seconds. We compare PAT with the SPIN model checker mainly because SPIN targets at similar systems and it adopts similar model checking approaches (i.e., automata-based on-the-fly LTL model checking). Note that there are improvements on model checking techniques which have not been incorporated in SPIN, e.g., on handling large LTL formulae [10, 11] or on nested DFS [26].

The protocols include leader election for complete networks (*LE_C*) [6], for rooted trees (*LE_T*) [9], for odd sized rings (*LE_OR*) [13], for network rings (*LE_R*) [6] and token circulation for network rings (*TC_R*) [7]. The descriptions of these protocols are built in the PAT model checker [28]. The property 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. For simplicity, fairness is applied to the whole system. We remark that event-level fairness or strong global fairness is required for these examples.

Table 1 Experiment results on population protocols

Model	Size	EWF			ESF		SGF	
		Result	PAT	SPIN	Result	PAT	Result	PAT
LE_C	5	Yes	4.7	35.7	Yes	4.7	Yes	4.1
LE_C	6	Yes	26.7	229	Yes	26.7	Yes	23.5
LE_C	7	Yes	152.2	1 190	Yes	152.4	Yes	137.9
LE_C	8	Yes	726.6	5 720	Yes	739.0	Yes	673.1
LE_T	5	Yes	0.2	0.7	Yes	0.2	Yes	0.2
LE_T	7	Yes	1.4	7.6	Yes	1.4	Yes	1.4
LE_T	9	Yes	10.2	62.3	Yes	10.2	Yes	9.6
LE_T	11	Yes	68.1	440	Yes	68.7	Yes	65.1
LE_T	13	Yes	548.6	3 200	Yes	573.6	Yes	529.6
LE_OR	3	No	0.2	0.3	No	0.2	Yes	11.8
LE_OR	5	No	1.3	8.7	No	1.8	–	–
LE_OR	7	No	15.9	95	No	21.3	–	–
LE_R	3	No	0.1	< 0.1	No	0.2	Yes	1.5
LE_R	4	No	0.3	< 0.1	No	0.7	Yes	19.5
LE_R	5	No	0.8	< 0.1	No	2.7	Yes	299.0
LE_R	6	No	1.8	0.2	No	4.6	–	–
LE_R	7	No	4.7	0.6	No	9.6	–	–
LE_R	8	No	11.7	1.7	No	28.3	–	–
TC_R	3	Yes	< 0.1	< 0.1	Yes	< 0.1	Yes	< 0.1
TC_R	5	No	< 0.1	< 0.1	No	< 0.1	Yes	0.6
TC_R	7	No	0.2	0.1	No	0.2	Yes	13.7
TC_R	9	No	0.4	0.2	No	0.4	Yes	640.2

As discussed in Section 2, process-level weak fairness is different from event-level weak fairness. In order to compare PAT with SPIN for verification with event-level weak fairness, we twist the models so that each event in population protocols is modeled as a process. By a simple argument, it can be shown that for such models, event-level weak fairness is equivalent to process-level weak fairness. Nonetheless, model checking with process-level weak fairness in SPIN increases the verification time by a factor that is linear in the number of processes. By modeling each event as a process, we increase the number of processes and therefore unavoidably increase the SPIN verification time by a factor that is constant (in the number of events per process for network rings) or linear (in the number of network nodes for complete network). SPIN has no support for event-level/process-level strong fairness or strong global fairness. Thus, the only way to model check with strong fairness or strong global fairness in SPIN is to encode the fairness constraints as part of the property. However, even for the smallest network (with three nodes), SPIN needs significant amount of time to construct (very large) Büchi automata from the property. Therefore, we conclude that it is infeasible to use SPIN for such a purpose and omit the experiment results from the table. We remark that in theory, strong fairness can be transformed to weak

fairness by paying the price of one Boolean variable [23]. Nonetheless, the property needs to be augmented with additional clauses after the translation, which is again infeasible. In [4], we carried out some experiments on using the approach in [23] and the results show that it is very inefficient.

All of the algorithms fail to satisfy the property without fairness. The algorithm for complete networks (*LE_C*) or trees (*LE_T*) requires at least event-level weak fairness, whereas the rest of the algorithms require strong global fairness. It is thus important to be able to verify systems with strong fairness or strong global fairness. Notice that the token circulation algorithm for network rings (*TC_R*) functions correctly for a network of size 3 with event-level weak fairness. Nonetheless, event-level weak fairness is not sufficient for a network with more nodes, as shown in the table. The reason is that a particular sequence of message exchange which satisfies event-level weak fairness only needs the participation of at least four network nodes. This suggests that our improvement in terms of the performance and ability to handle different forms of fairness has its practical value. We highlight that previously unknown bugs in implementation of the leader election algorithms for odd-sized ring [13] have been revealed using PAT.

A few conclusions can be drawn from the results in the table. Firstly, in the presence of counterexamples, PAT usually finds one quickly (e.g., on *LE_R* and *TC_R* with event-level weak fairness or strong fairness). It takes longer to find a counterexample for *LE_OR* mainly because there are too many possible initial configurations of the system (exactly 2^{5N} where N is network size) and a counterexample is only present for particular initial configurations. Secondly, verification with event-level strong fairness is more expensive than verification with no fair, event-level weak fairness or strong global fairness. This conforms to theoretical time complexity analysis. The worse case scenario is absent from these examples (e.g., there are easily millions of transitions/states in many of the experiments). Lastly, PAT is worse than SPIN on *LE_R* and *TC_R* with event-level weak fairness. This is however not indicative as when there is a counterexample, the verification time depends on the searching order. PAT outperforms the current practice of verification with fairness. PAT offers comparably better performance on verification with weak fairness (e.g., on *LE_C* and *LE_T*) and makes it feasible to verify with strong fairness or strong global fairness. This allows us to discover bugs in systems functioning with strong fairness. Experiments on *LE_C* and *LE_T* (for which the property is only false with no fairness) show minor computational overhead for handling a stronger fairness.

Table 2 shows verification statistics of benchmark systems to show other aspects of our algorithm. Notice that the property "bounded bypass" means a process which is interested in getting into the critical section will be in the critical section; the verification times for PAT and SPIN are measured in seconds. Because of the deadlock state, the dining philosophers model ($dp(N)$ for N philosophers and forks) does not guarantee that a philosopher always eventually eats ($\Box \diamond eat0$) whether with no fairness or strong global fairness. This experiment shows PAT takes little extra time for handling the fairness assumption. We remark that PAT may spend more time than SPIN on identifying a counterexample in some cases. This is both due to the particular order of exploration and the difference between model checking based on nested DFS and model checking based on identifying SCCs. PAT's algorithm relies on SCCs. If a large portion of the system is strongly connected, it takes more time to construct the SCC before testing whether it is fair or not. In this example, the whole system contains one large SCC and a few trivial ones including the deadlock state. If PAT happens to start with the large one, the verification may take considerably more time.

Table 2 Experiment results on benchmark systems

Model	Property	Result	Fairness	PAT	SPIN
dp(10)	$\Box \diamond eat0$	No	No	0.8	< 0.1
dp(13)	$\Box \diamond eat0$	No	No	9.8	< 0.1
dp(15)	$\Box \diamond eat0$	No	No	56.1	< 0.1
dp(10)	$\Box \diamond eat0$	No	SGF	0.8	–
dp(13)	$\Box \diamond eat0$	No	SGF	9.8	–
dp(15)	$\Box \diamond eat0$	No	SGF	56.0	–
ms(10)	$\Box \diamond work0$	Yes	ESF	9.3	–
ms(12)	$\Box \diamond work0$	Yes	ESF	105.5	–
peterson(3)	bounded bypass	Yes	PWF	0.1	1.25
peterson(4)	bounded bypass	Yes	PWF	1.7	> 671
peterson(5)	bounded bypass	Yes	PWF	58.9	–

Milner's cyclic scheduler algorithm ($ms(N)$ for N processes) is a showcase for the effectiveness of PAT, in which we apply event-level strong fairness to the whole system. Peterson's mutual exclusive algorithm ($peterson(N)$) requires at least process-level weak fairness to guarantee bounded bypass [29], i.e., if a process requests to enter the critical section, it eventually will. The property is verified with process-level weak fairness in PAT and process-level weak fairness in SPIN. PAT outperforms SPIN in this setting as well.

6 Related work

This work is the full version of the tool paper about the PAT model checker published at CAV09 [30]. In this article, we

present a comprehensive list of fairness notations and how they are supported efficiently in the PAT model checker. The research on categorizing fairness/liveness, motivated by system analyzing of distributed or concurrent systems, has a long history [2,17,31,32]. A rich set of fairness notions have been identified during the last decades, e.g., weak or strong fairness in [2], justice or compassion conditions in [17], hyperfairness in [19, 33], strong global or local fairness recently in [6], etc. This work summarizes a number of fairness notions which are closely related to distributed system verification and provides a framework to model check with different fairness constraints. Other works on categorizing fairness/liveness have been evidenced in [20, 34–36].

This work is related to research on system verification with fairness [1,2,6]. In automata theory, fairness/liveness is often captured using the notion of accepting states. For instance, an execution of a Büchi automaton (which is an automaton with justice conditions) must visit at least one accepting state infinitely often. An execution of a Streett automaton (which is an automata with compassion conditions) must infinite states in a set F if infinitely many states in E is visited, where E and F are subsets of states. Our model checking algorithm is related to previous works on emptiness checking for Büchi automata [18,21] and Streett automata [21–23,37]. Two different groups of methods have been proposed for checking emptiness of Büchi automata, i.e., one based in nested DFS [38] and the other based on identifying SCCs [21]. As discussed in Section 4, nested DFS is not feasible for verification with strong fairness. Similar to [21], our work is based on Tarjan's algorithm for identifying SCCs. We generalize the idea to handle different fairness constraints. In [22], an algorithm for checking emptiness of Streett automata is proposed. In this work, we apply the idea to the automata-based model checking framework and generalize it to handle different fairness constraints. In this way, our algorithm integrates the two algorithms presented in [21,22] and extends them in a number of aspects to suit our purpose. Furthermore, model checking algorithms with strong fairness have been proposed in [21] and [37]. In both works, a similar pruning process is involved. Besides, some research has been done on combining fairness and state reduction techniques, e.g., combining symmetry reduction with global fairness assumptions [39], applying partial order reduction during model checking with fairness assumption [4], etc.

This work is also related to previous work on CTL model checking with fairness, which also relies on identifying a fair strongly connected component. For instance, the basic fixed-point computation algorithm for the identification of fair executions was presented in [40] and independently developed in [41] for fair CTL. Nonetheless, our algorithm is designed for automata-based on-the-fly model checking, with a variety of fairness constraints including the recently emerged strong global fairness. Different from the previous works [42,43] on symbolic model checking with fairness, our approach is designed for LTL model checking. This work is also related to the recent work on designing a strong fair scheduler for concurrent programs testing presented [24]. The fair scheduler presented [24] generates only partial fair executions, which works for testing but not formal verification. In addition, this work is remotely related to the work on handling large LTL formulae [11] as well as our previous work on verifying concurrent systems [44–46].

7 Conclusion and future work

In summary, we developed a method and a self-contained toolkit for (distributed) system analysis with a variety of fairness constraints. We showed that our method and the toolkit are effective enough to prove or disprove not only benchmark systems but also newly proposed distributed algorithms.

We are actively developing PAT. One future work of particular interest is to investigate refinement with fairness constraints. Refinement checking is an alternative way of system verification. Instead of showing that a system implementation satisfies some critical property, refinement checking may be used to show that the implementation satisfies its own specification (often in the same language). The main motivation is that refinement with a fair scheduler or in a distributed system is different from refinement under scheduling with no fairness. For instance, trace refinement with event-level weak fairness prevents removing a transition which is always enabled and trace refinement with strong global fairness prevents removing a nondeterministic choice. The consequence of a fair scheduler over program refinement is worth investigating. In order to handle non-trivial systems, efficient algorithms for refinement checking with fairness must be developed.

Another line of future work is to investigate how to verify infinite-state systems with fairness. For instance, how to integrate our method with abstraction techniques which are used to build finite state models. In particular, we will investigate abstract schema which are proposed for parameterized systems so that our method can be extended to verify population protocols with many or even infinite network nodes.

Acknowledgements This work was partially supported by the research

grant NAP project “Formal Verification on Cloud” (M4081155.020), “Automatic Checking and Verification of Security Protocol Implementations” (M4080996.020), and the National Natural Science Foundation of China (Grant Nos. 91118003, 61272106).

References

1. Giannakopoulou D, Magee J, Kramer J. Checking progress with action priority: is it fair? In: Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering. 1999, 511–527
2. Lamport L. Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering, 1977, 3(2): 125–143
3. Kurshan R P. Computer-aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, 1995
4. Sun J, Liu Y, Dong J S, Wang H H. Specifying and verifying event-based fairness enhanced systems. In: Proceedings of the 10th International Conference on Formal Engineering Methods. 2008, 318–337
5. Sun J, Liu Y, Roychoudhury A, Liu S, Dong J. Fair model checking with process counter abstraction. FM 2009: Formal Methods, 2009, 123–139
6. Fischer M J, Jiang H. Self-stabilizing leader election in networks of finite-state anonymous agents. In: Proceedings of the 10th International Conference on Principles of Distributed Systems. 2006, 395–409
7. Angluin D, Aspnes J, Fischer M J, Jiang H. Self-stabilizing population protocols. In: Proceedings of the 9th International Conference on Principles of Distributed Systems. 2005, 103–117
8. Angluin D, Fischer M J, Jiang H. Stabilizing consensus in mobile networks. In: Proceedings of the 2006 International Conference on Distributed Computing in Sensor Systems. 2006, 37–50
9. Canepa D, Potop-Butucaru M. Stabilizing Token Schemes for Population Protocols. Computing Research Repository, 2008, abs/0806.3471
10. Rozier K Y, Vardi M Y. LTL Satisfiability Checking. In: Proceedings of the 14th International SPIN Workshop. 2007, 149–167
11. Hammer M, Knapp A, Merz S. Truly on-the-fly LTL model checking. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2005, 191–205
12. Pang J, Luo Z Q, Deng Y X. On automatic verification of self-stabilizing population protocols. In: Proceedings of the 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering. 2008, 185–192
13. Jiang H. Distributed Systems of Simple Interacting Agents. PhD thesis, Yale University, 2007
14. Sun J, Liu Y, Dong J S, Chen C Q. Integrating specification and programs for system modeling and verification. In: Proceedings of the third IEEE International Symposium on Theoretical Aspects of Software Engineering. 2009, 127–135
15. Hoare C A R. Communicating Sequential Processes. International Series on Computer Science. Prentice-Hall, 1985
16. Chaki S, Clarke E M, Ouaknine J, Sharygina N, Sinha N. State/event-based software model checking. In: Proceedings of the 4th International Conference on Integrated Formal Methods. 2004, 128–147
17. Lehmann D J, Pnueli A, Stavi J. Impartiality, justice and fairness: the ethics of concurrent termination. In: Proceedings of the 8th Colloquium on Automata, Languages and Programming. 1981, 264–277
18. Holzmann G J. The SPIN Model Checker: Primer and Reference Manual. Addison Wesley, 2003
19. Lamport L. Fairness and hyperfairness. Distributed Computing, 2000, 13(4): 239–245
20. Pnueli A, Sa’ar Y. All you need is compassion. In: Proceedings of the 9th International Conference on Verification, Model Checking and Abstract Interpretation. 2008, 233–247
21. Geldenhuys J, Valmari A. More efficient on-the-fly LTL verification with Tarjan’s algorithm. Theoritical Computer Science, 2005, 345(1): 60–82
22. Henzinger M R, Telle J A. Faster algorithms for the nonemptiness of streett automata and for communication protocol pruning. In: Proceedings of the 5th Scandinavian Workshop on Algorithm Theory. 1996, 16–27
23. Kesten Y, Pnueli A, Raviv L, Shahar E. Model checking with strong fairness. Formal Methods and System Design, 2006, 28(1): 57–84
24. Musuvathi M, Qadeer S. Fair stateless model checking. In: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation. 2008, 362–371
25. Baier C, Katoen J. Principles of Model Checking. The MIT Press, 2008
26. Schwoon S, Esparza J. A note on on-the-fly verification algorithms. In: Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2005, 174–190
27. Tarjan R. Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1972, 2: 146–160
28. Liu Y, Pang J, Sun J, Zhao J. Verification of population ring protocols in pat. In: Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering. 2009, 81–89
29. Alagarsamy K. Some myths about famous mutual exclusion algorithms. SIGACT News, 2003, 34(3): 94–103
30. Sun J, Liu Y, Dong J S, Pang J. PAT: towards flexible verification under fairness. Lecture Notes in Computer Science, 2009, 5643: 709–714
31. Apt K R, Francez N, Katz S. Appraising fairness in languages for distributed programming. Distributed Computing, 1988, 2(4): 226–241
32. Francez N. Fairness. Texts and Monographs in Computer Science. Springer-Verlag, 1986
33. Attie P C, Francez N, Grumberg O. Fairness and hyperfairness in multiparty interactions. Distributed Computing, 1993, 6(4): 245–254
34. Queille J P, Sifakis J. Fairness and related properties in transition systems — a temporal logic to deal with fairness. Acta Informaticae, 1983, 19: 195–220
35. Kwiatkowska M Z. Event fairness and non-interleaving concurrency. Formal Aspects of Computing, 1989, 1(3): 213–228
36. Völzer H, Varacca D, Kindler E. Defining fairness. In: Proceedings of the 16th International Conference on Concurrency Theory. 2005, 458–472
37. Latvala T, Heljanko K. Coping with strong fairness. Fundamenta Informaticae, 2000, 43(1–4): 175–193
38. Courcoubetis C, Vardi M Y, Wolper P, Yannakakis M. Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design, 1992, 1(2/3): 275–288
39. Zhang S, Sun J, Pang J, Liu Y, Dong J. On combining state space reductions with global fairness assumptions. FM 2011: Formal Methods, 2011, 432–447

40. Lichtenstein O, Pnueli A. Checking that finite state concurrent programs satisfy their linear specification. In: Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 1985, 97–107
41. Emerson E A, Lei C L. Modalities for model checking: branching time logic strikes back. Science of Computer Programming, 1987, 8(3): 275–306
42. Hardin R H, Kurshan R P, Shukla S K, Vardi M Y. A new heuristic for bad cycle detection using BDDs. Formal Methods in System Design, 2001, 18(2): 131–140
43. Klarlund N. An n log n Algorithm for Online BDD Refinement. In: Proceedings of the 9th International Conference on Computer Aided Verification. 1997, 107–118
44. Dong J S, Liu Y, Sun J, Zhang X. Verification of computation orchestration via timed automata. In: Proceedings of the 8th International Conference on Formal Engineering Methods. 2006, 226–245
45. Dong J S, Hao P, Sun J, Zhang X. A reasoning method for timed CSP based on constraint solving. In: Proceedings of the 8th International Conference on Formal Engineering Methods. 2006, 342–359
46. Sun J, Liu Y, Dong J S. Model checking csp revisited: introducing a process analysis toolkit. In: Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. 2008, 307–322

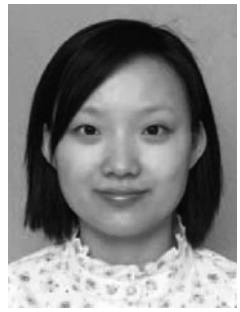
Yuanjie Si is a PhD candidate in the College of Computer Science at Zhejiang University, China. He received BS in Software Engineering from Zhejiang University in China. His current research interests include software engineering and formal verification, in particular, model checking and software reliability evaluation techniques.

Jun Sun received BS and PhD degrees in Computing Science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship in School of Computing of NUS. Since 2010, he joined Singapore University of Technology and Design (SUTD) as an assistant professor. He was a visiting scholar at MIT from 2011–2012. Jun's research focuses on software engineering and formal methods, in particular, system verification and model checking. He is the co-founder of the PAT model checker.

Yang Liu graduated in 2005 with a BS of Computing in the National University of Singapore (NUS). In 2010, he obtained his PhD and continued with his post doctoral work in NUS. Since 2012, he joined Nanyang Technological University as an assistant professor. His research focuses on software engineering, formal methods and security. Particularly, he specializes in software verification using model checking techniques. This work led to the development of a state-of-the art model checker, Process Analysis Toolkit.

Jin Song Dong received BS and PhD degrees in Computing from University of Queensland, Australia in 1992 and 1996. From 1995-1998, he was Research Scientist at CSIRO in Australia. Since 1998 he has been in the School of Computing at the National University of Singapore (NUS) where he is currently Associate Professor and a member of PhD supervisors at NUS Graduate School. He is on the editorial board of Formal Aspects of Computing and Innovations in Systems and Software Engineering. His research interests include formal methods, software engineering, pervasive computing, and semantic technologies.

Jun Pang received his PhD degree from Free University of Amsterdam, Netherlands. Currently, he is a senior researcher with the Faculty of Science, Technology and Communication in the University of Luxembourg. His main research areas include formal methods, model checking, security and privacy.



Shao Jie Zhang received her PhD degree in computer science from National University of Singapore in 2013. Currently, she is a postdoctoral researcher in Singapore University of Technology and Design, Singapore. Her current research interests include software engineering and formal verification, in particular model checking and state reduction techniques.

Xiaohu Yang received his PhD degree of Computer Science at Department of Computer Science and Technology in Zhejiang University, China in 1993. He is currently a professor in College of Computer Science, Zhejiang University, China. His main research areas include very large-scale information system, software engineering, and financial informatics.