NOTICE: This is the technical report:
Towards a Toolkit for Flexible and Efficient Verification under Fairness

The technical report titled:
Model Checking Linearizability via Refinement
has been re-located to http://www.comp.nus.edu.sg/~pat/report.pdf

Towards a Toolkit for Flexible and Efficient Verification under Fairness

Jun Sun[1], Yang Liu[1], Jin Song Dong[1], and Jun Pang[2]

[1] School of Computing, National University of Singapore,
{sunj,liuyang,dongjs}@comp.nus.edu.sg
[2] Computer Science and Communications, University of Luxembourg,
jun.pang@uni.lu

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. Current practice of system analysis is, however, deficient under fairness constraints. In this work, we present a *Process Analysis Toolkit* (PAT) for flexible and efficient system analysis under fairness constraints. A unified algorithm is proposed to model check systems with different fairness (e.g., weak fairness, strong local fairness, strong global fairness) effectively. Partial order reduction which is effective for distributed system verification is extended to fair systems whenever possible. We show through empirical evaluation (on recent population protocols as well as benchmark systems) that PAT significantly outperforms the current practice of verification with fairness. We report that previously unknown bugs have been revealed using PAT against systems functioning under strong global fairness.

1 Introduction

In the area of distributed system/software 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 the network. A counterexample to a liveness property (against a finite state system) is typically 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 non-determinism, is often necessary and important to prove liveness properties. Fairness is an abstraction of the fair scheduler[3] in a multi-threaded programming environment or the relative speed of the processors in distributed systems. Without fairness, verification of liveness properties often produces unrealistic loops during which one process or event is infinitely ignored by the scheduler or one processor is infinitely faster than others. It is important to rule out those counterexamples and utilize the computational resource to identify the real bugs. However, systematically ruling out counterexamples due to lack of fairness is highly non-trivial. It requires flexible specification of fairness as well as efficient verification with fairness.

[3] e.g., the random scheduler which is a strongly fair scheduler.

In this work, we focus on formal system analysis under fairness assumptions. The objective is to deliver a toolkit which checks linear temporal logic (LTL) properties against distributed systems which function under a variety of fairness. Fairness and model checking with fairness have attracted much theoretical interests for decades [14, 24, 34]. Their practical implications in system/software design and verification have been discussed extensively. Recent development on distributed systems showed that there are a family of fairness notions which are crucial for designing self-stabilizing distributed algorithms [2, 3, 11, 5]. Some of the fairness notions are only recently formulated [11]. In order to verify (implementations of) those algorithms, model checking techniques must take fairness into account. However, current practice of formal verification is deficient with respect to the variety of fairness.

Recently, the population protocol model has emerged as an elegant computation paradigm for describing mobile ad hoc networks [2]. Such networks consist of multiple mobile nodes which interact with each other to carry out a computation. Application domain of the protocols include wireless sensor networks and biological computers. The interactions 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 [2, 3, 11, 5]. Fairness plays an important role in the protocols. For instance, in [11] 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 under *strong local fairness*, whereas the algorithm for rings, only works under *strong global fairness*. It has further been proved that with only strong local fairness or weaker, uniform self-stabilizing leader election in rings is impossible [11]. Verifying (implementations of) the algorithms thus must be carried out under the respective fairness constraints.

Existing verification algorithms/tools are ineffective with respect to fairness. One way to apply existing model checkers for verification under fairness constraints is to reformulate 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.

$$fairness\ assumptions \Rightarrow \phi \qquad \text{– F1}$$

This practice is, though flexible, deficient for two reasons. Firstly, 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 the formulas. Thus, it is infeasible to handle large formulas, whereas a typical system may have multiple fairness constraints. For example, SPIN is a popular LTL model checker [18]. The algorithm it uses for generating Büchi automata handles only a limited number of fairness constraints. Table 1 shows experiments on the time and space needed for SPIN to generate the automaton from standard notion of fairness, in particular, justice and compassion conditions [21] as explained later. The experiments are made on a 3.0GHz Pentium IV CPU and 1 GB memory executing SPIN 4.3. The results show that it takes a non-trivial amount of time to handle 5 fairness constraints.

Prop.	n	Time (Sec.)	Memory	#Büchi States
$(\bigwedge_{i=1}^{n} \Box\Diamond p_i) \Rightarrow \Box\Diamond q$	1	0.08	466Kb	74
same above	2	0.19	2Mb	286
same above	3	4.44	27MB	1052
same above	4	128.16	283Mb	3686
same above	5	more than 3600	more than 1Gb	–
$(\bigwedge_{i=1}^{n}(\Box\Diamond p_i \Rightarrow \Box\Diamond q_i)) \Rightarrow \Box\Diamond s$	1	0.13	487.268	134
same above	2	1.58	10123.484	1238
same above	3	30.04	55521.708	4850
same above	4	4689.24	more than 1Gb	–

Table 1. Experiments on LTL to Büchi Automata Conversion

Secondly, partial order reduction which is one of the most successful reduction techniques for model checking distributed systems becomes ineffective. Partial order reduction ignores/postpones invisible actions, whereas given $F1$ all actions/propositions presented in *fairness constraints* become visible and therefore cannot be ignored or postponed. In [30], Pang *et al* applied the SPIN model checker [18] to establish the correctness of a family of population protocols. Only protocols relying on a weak notion of fairness operating on very small networks were verified because of the problem discussed above. Protocols relying on stronger notion of fairness (e.g., strong local/global fairness) are beyond the capability of SPIN even for the smallest network (e.g., with 2 nodes). It is important to develop an alternative approach which 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. Nonetheless, the focus of existing model checkers has been on process-level fairness[4], which, informally speaking, states that every process shall make infinite progress if sufficiently possible (refer to detailed explanation later). For instance, SPIN has implemented a model checking algorithm which handles process-level weak 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 weak fairness may not be sufficient, e.g., for population protocols.

In this work, we present a *Process Analysis Toolkit* (PAT)[5] which is designed to verify system with fairness efficiently and flexibly. PAT supports different ways of applying fairness. An ordinary user may choose one of the fairness constraints (e.g., process-level fairness, weak fairness, strong local/global fairness) and apply it to the whole system. Or an advanced user may manually annotate different fairness with the relevant actions/events. The latter approach is motivated by population protocols and open system verification. It can be used to benefit the current model checking practice

[4] Or weak/strong fairness on top of CTL properties.

[5] Publicly available at http://pat.comp.nus.edu.sg.

in multiple ways. It is as flexible as encoding fairness in the property, only more efficient. In particular, we show that partial order reduction, which is proved in-feasible for verification under some fairness constraints, is feasible in such an approach. A unified on-the-fly model checking algorithm which handles a variety of fairness constraints is then applied to verify the fair system efficiently. The algorithm extends previous work on model checking based on finding strongly connected components. Using PAT, we identified *previously unknown bugs* in the implementation of population protocols [11, 19]. For experiments, we compare the two different approaches and SPIN over recent distributed algorithms as well as benchmark systems. We show that our approach handles fairness more flexibly and efficiently.

This work is related to research on categorizing and verifying fairness in general [14, 24, 11]. A rich set of fairness notions have been identified during the last decades, e.g., weak or strong fairness by Lamport [24], justice or compassion conditions by Pnueli [27] and strong global or local fairness recently by Fischer [11]. In this paper, we examine various kinds of fairness and propose a flexible verification framework which can be used to handle most of the fairness notions. In automata theory, fairness/liveness is often captured using the notion of accepting states. For instance, at least one accepting state must be visited infinitely often in Büchi automata. Our model checking algorithm is related to previous works on emptiness checking for Büchi automata (i.e., automata with justice conditions) and Streett automata (i.e., automata with compassion conditions) [21, 13, 26, 16]. In this work, we handle fairness constraints which not only concerns with states but also actions/events. In a way, our algorithm integrates the two algorithms presented in [21, 13] and improves them in a number of aspects to suit our purpose. For instance, our unified algorithm is very flexible with respect to different fairness. It is designed for automata-based explicit on-the-fly model checking with partial order reduction in mind. In addition, Section 4 is related to our previous work [34]. In [34], we annotate event-based process algebras with two kinds of fairness, namely weak/strong event fairness. In this work, we significantly extend [34]. Instead of process algebras under event-based fairness, we handle general programs under a variety of fairness constraints. For instance, strong local/global fairness which are very important for population protocols have been formulated and supported. The main contribution of this work is the new flexible verification algorithm and the toolkit which are designed to handle fairness efficiently. For instance, strong global fairness is handled much more efficiently in this work. This work is also related to previous work on CTL model checking with fairness, which too 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 [28] and independently developed in [10] for fair CTL. Nonetheless, our algorithm is designed for automata-based on-the-fly model checking, with a variety of fairness including the recently emerged strong global fairness. Different than the previous works [15, 22] 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 [29]. The fair scheduler presented [29] generates only partial fair executions, which works for testing but not formal verification. This work is remotely related to our previous works on verifying concurrent systems [9, 8, 33].

The remainder of the article is organized as follows. Section 2 reviews a concrete motivating example and our computational model, together with a family of different fairness. Section 3 presents our first contribution, i.e., the unified algorithm for verification under fairness. Section 4 discusses our second contribution, i.e., an alternative way for specifying and verifying action-based systems with fairness. Section 5 presents our third contribution, i.e., the PAT model checking system, and experiment results. Section 6 concludes the paper.

2 Background

In this section, we start with introducing the PAT's modeling language by modeling a concrete motivating example and then present our computational model as well as formal definitions of fairness. 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. Compared to SPIN and Promela, PAT supports more high-level compositional operators from classic process algebras [17] and similar programming-level constructs. Refer to the PAT web site for details on its input language.

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 [2, 19, 11], a number of algorithms have been proposed for self-stabilizing leader election. In particular, a self-stabilizing algorithm for ring networks is proposed [11]. This algorithm guarantees that one and only one leader will be eventually elected, given any initial configuration. The algorithm, however, only works under two assumptions. One is that the system satisfies a rather strong fairness constraint called strong global fairness [11]. The other is that there exists a leader detector which *eventually* detects whether or not there is a leader in the system. The detector is a diagnostic device which tests network nodes for certain information. The detector *eventually* (which is weaker than *immediately*) detects the presence/absence of a leader. Figure 1 presents the PAT model of the algorithm. Line 1 defines a global constant N of value 3. It models the network size, i.e., number of nodes. At line 2, a proposition named $exist$ is defined, which is the synonymy of the Boolean formula. The proposition may be used in the model or the assertions. Line 3 to 4 defines the global variables. In particular, $correct$ 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 presence of a leader and *false* for absence. Array $leader$, $bullet$ and $shield$ model the status of each node. For instance, the i bit of $leader[N]$ tells whether the i-th node is a leader or not. Note that PAT has a weak type system.

```
1. #define N 3;
2. #define exist (correct == 0 && guess) || (correct! = 0 && leader[0] + leader[1] + leader[2] > 0);
3. var correct = 0; var guess = false;
4. var leader[N]; var bullet[N]; var shield[N];
5. Node(i) = case {
6.       !exist : rule1.i.(i + 1)%N{bullet[i] = 1; leader[i] = 1; shield[i] = 1; } → Node(i)
7.       leader[i] == 0&&shield[i] == 1&&exist : rule2.i.(i + 1)%N{
8.                leader[i] = 0; shield[i] = 0; bullet[(i + 1)%N] = 0; shield[(i + 1)%N] = 1;
9.           } → Node(i)
10.      leader[i] == 1 && shield[i] == 1 && exist : rule3.i.(i + 1)%N{
11.               bullet[i] = 1; leader[i] = 1; shield[i] = 0; bullet[(i + 1)%N] = 0; shield[(i + 1)%N] = 1;
12.          } → Node(i)
13.      leader[i] == 1 && shield[i] == 0 && bullet[(i + 1)%N] == 0 && exist :
14.           rule4.i.(i + 1)%N{bullet[i] = 1; leader[i] = 1; shield[i] = 0; bullet[(i + 1)%N] = 0; } → Node(i)
15.      shield[i] == 0 && bullet[(i + 1)%N] == 1 && exist :
16.           rule5.i.(i + 1)%N{bullet[i] = 1; leader[i] = 0; shield[i] = 0; bullet[(i + 1)%N] = 0; } → Node(i)
17. };
18. Detector() = oracle{correct = 1; } → Detector() []
19.      guess1{guess = false; } → Detector() [] guess2{guess = true; } → Detector();
20. LeaderElection() = Init(); (Detector() ||| (||| x : 0..N − 1@Process(x)));

21. #define oneLeader (leader[0] + leader[1] + leader[2] == 1);
22. #assert LeaderElection() |= <> []oneLeader;
```

Fig. 1. Leader Election Protocol for Rings

Next, line 5 to 20 defines processes in the form of equations, which capture the essence of the algorithm. In particular, line 5 to 17 defines process $Node(i)$ which models the behaviors of a network node. Every time there is an interaction in the network, the initiator and responder must update themselves according to a set of 5 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** construct. Intuitively, a rule is applicable only if the guarding condition is satisfied. We skip the details of the rules and refer the readers to [11]. Line 18 and 19 define the leader $Detector$, where $[]$ is the choice operator borrowed from the classic CSP [17]. The detector has three choices. It may be enlightened (line 18) through the action $oracle$ and then detects correctly ever-after. Or, it may take a guess, randomly stating that there is a leader or there is not (line 19). Notice that $oracle$ is the action name and the assignment $correct = 1$ is executed together with the action. In general, an action may be attached with a sequential program (with loops, branches, etc.). The program executes atomically. Equivalently, $oracle$ can be viewed as a label of the program (for easy referencing). We remark that there is no guarantee that the detector will eventually detect correctly (e.g., the action $oracle$ may never happen) unless fairness is applied.

Line 20 models the leader election algorithm as process $LeaderElection$. The algorithm firstly invoked process $Init$, which initializes the system in every possible configuration, e.g., each element in the arrays may be either assign 0 or 1. We omit the details of $Init$ as it can be constructed straightforwardly using the choice operator $[]$. After initialization, the system is the interleaving (modeled by operator $|||$) of the leader detector and all the network nodes.

The property of particular interest to all leader election algorithms is $\Diamond\Box oneLeader$ (defined as an assertion at line 22), where $\Diamond$ and $\Box$ are modal operators which read as "eventually" and "always" respectively. $oneLeader$ (defined as line 21) is a proposition which states that there is one and only one leader in the network. In PAT, we support the state/event LTL [6]. The assertion is false under no fairness, process-level fairness, weak fairness or strong local fairness. Counterexamples can be generated efficiently for each of the cases using PAT. The assertion is true under 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 (like the one in [19]) satisfies the property, it must be verified under 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[6]. Let a be an action, which could be either an abstract event (e.g., a synchronization barrier if shared by multiple processes) or a data operation (e.g., a named sequential program). Let Σ be the set of all actions.

[6] by defining a complete set of operational semantics.

Definition 1 (LTS). *A Labeled Transition System is a 3-tuple $(S, init, \rightarrow)$ where S is a set of states, $init \in S$ is an initial state and $\rightarrow \subseteq S \times \Sigma \times S$ is a labeled transition relation.*

For simplicity, we write $s \xrightarrow{a} s'$ to denote that (s, a, s') is a transition in $\rightarrow$ and $s \rightarrow s'$ to denote that there exists some a such that $s \xrightarrow{a} s'$. $enabled(s)$ is the set of enabled actions at s, i.e., a is in $enabled(s)$ if and only if there exist s' such that $s \xrightarrow{a} s'$. We write $a(s)$ to denote the set of states reachable from s by engaging in a. Let $\#a(s)$ be the size of the set. Notice that $\#a(s)$ may be larger than 1 because of non-determinism.

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 infinite idling actions at the rear. Given an LTS $L = (S, init, \rightarrow)$, an execution is an infinite sequence of alternating states and actions $E = \langle s_0, a_0, s_1, a_1, \cdots \rangle$ where $s_0 = init$ and for all i such that $s_i \xrightarrow{a_i} s_{i+1}$. An LTS is feasible if and only if it has at least one execution. Without fairness constraints, a system may behave freely as long as it starts with an initial state and conforms to the transition relation. A fairness constraint restricts the set of system behaviors to only those fair ones. In the following, we review a variety of different fairness constraints and illustrates their differences using examples.

Definition 2 (Weak Fairness). *Let an execution E be $\langle s_0, a_0, s_1, a_1, \cdots \rangle$. E satisfies weak fairness, or is weak fair, if and only if for every action a, if a eventually becomes enabled forever in E, then $a = a_i$ for infinitely many i, i.e., $\Diamond\Box\ a\ is\ enabled \Rightarrow \Box\Diamond\ a\ is\ engaged$.*

Weak fairness is initially suggested by Lamport in [24]. It states that if an action 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 a is disabled or has just been taken. The latter is known as justice condition, suggested by Lehmann, Pnueli, and Stavi in [27]. Intuitively, it means that an enabled action shall not be ignored infinitely or equivalently some state must be visited infinitely often (e.g., accepting states in Büchi automata). Given a property ϕ, verification under weak fairness is to verify whether all weak fair executions satisfy ϕ. Weak fairness or justice conditions has been well studied and verification under weak fairness has been supported to some extent [18].

Given the LTS in Figure 2(a), the property $\Box\Diamond\ a$ is true under weak fairness assumption, whereas it is false under no fairness. Action a is always enabled and, hence, by definition it must be infinitely often engaged. Different than the related but different process-level weak fairness (supported in SPIN), weak fairness is not related to the system structure. Process-level weak fairness means that a process which is always enabled (to engage in some action) must eventually make a move. It guarantees that each process is only finitely faster than the others. Assuming that one LTS corresponds to one process (in Promela), verifying the same property against the LTS in Figure 2(a) in SPIN using its fairness option, however, returns false. The reason is that the process may make progress infinitely (by repeatedly engaging in b) without ever engaging in action a. Given the two LTSs in Figure 2(b), $\Box\Diamond\ a$ is true under SPIN's fairness because both processes must make infinite progress and therefore both a and b must be

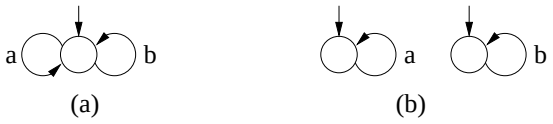

Fig. 2. weak fair vs. process-level fair

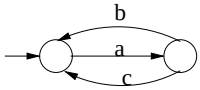

Fig. 3. weak fair vs. strong local fair

engaged infinitely. This example reveals that SPIN's fairness is associated with system structure whereas the ones we focused on are not. In this work, we concentrate on fairness notions which are structure independent. It can be shown that our approach can handle process-level weak/strong fairness straightforwardly. Furthermore, by a simple argument, it can be shown that weak fairness implies process-level weak fairness.

Definition 3 (Strong Local Fairness). *Let an execution E be $\langle s_0, a_0, s_1, a_1, \cdots \rangle$. E satisfies strong local fairness or is strong local fair if and only if, for every action a, if a is infinitely often enabled, then $a = a_i$ for infinitely many i, i.e., $\Box\Diamond\ a\ is\ enabled \Rightarrow \Box\Diamond\ a\ is\ engaged$.*

This notion of fairness has been identified by different researchers. In [25], it is named *strong fairness* by Lamport (by contrast to weak fairness defined above). In [11], it is named strong local fairness by Fischer (in comparison to strong global fairness defined below). It is also known as *compassion* condition, suggested by Pnueli [32]. Strong local fairness states that if an action 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. The process-level correspondence is process-level strong fairness which means that if a process is repeatedly enabled, it must eventually make some progress.

Strong local fairness is stronger than weak fairness (since $\Diamond\Box\ a\ is\ enabled$ implies $\Box\Diamond\ a\ is\ enabled$). Given the LTS in Figure 3, the property $\Box\Diamond\ b$ is false under weak fairness but true under strong local 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 under weak fairness. It is infinitely often enabled, and thus, the system must engage in b infinitely under strong local fairness by definition. Verification under strong local fairness or compassion conditions has been discussed previously, e.g., in the setting of Streett automata [13, 16], fair discrete systems [21] or programming codes [29]. Nonetheless, there are few established tool support for formal verification under strong local fairness to the best of our knowledge [14].

Definition 4 (Strong Global Fairness). *Let an execution E be $\langle s_0, a_0, s_1, a_1, \cdots \rangle$. E satisfies strong global fairness or is strong global fair if and only if, for every s, a, s'*

such that $s \xrightarrow{a} s'$*, if* $s = s_i$ *for infinite many* i*, then* $s_i = s$ *and* $a_i = a$ *and* $s_{i+1} = s'$ *for infinitely many* i*.*

Strong global fairness was suggested by Fischer and Jiang in [11]. It states that if a *step* (from s to s' by engaging in action a) can be taken infinitely often, then it must actually be taken infinitely often[7]. Different than the above notions of fairness, strong global fairness concerns about both actions and states, instead of actions only. It can be shown by a simple argument that strong global fairness is stronger than strong local fairness.

Strong global fairness requires that an infinitely enabled action must be taken infinitely often in *all* contexts, whereas strong local fairness only requires the enabled action to be taken in *one* context. Figure 4 illustrates the difference with two examples. Under strong local fairness, state 2 in Figure 4(a) may never be visited because all actions 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 Figure 4(a) must be visited infinitely often and therefore $\Box\Diamond\ state\ 2$ is true. Figure 4(b) illustrates their difference when there are non-determinism. Both transitions labeled a must be taken infinitely under strong global fairness, which is not necessary under strong local fairness or weak fairness. Thus, property $\Box\Diamond\ b$ is true under strong global fairness but not weak fairness or strong local fairness. A number of population protocols reply on strong global fairness, e.g., self-stabilizing leader election in ring networks [11] and token circulation in rings [2]. As far as the authors know, there are no previous works on verification under strong global fairness.

A number of other fairness notions have been discussed by various researchers, e.g., unconditional event fairness [23] which will be discussed in Section 4, hyper-fairness which is of only theatrical interests as stated in [25] and weak local/global fairness in [11]. We skip their definitions and remark that our approach can be extended to handle other kinds of fairness.

3 Verification under Fairness

Verification under fairness is to examine only fair executions of a given system and to decide whether certain property is true. Note that verifying whether a system is fair or not is relatively straightforward. For instance, to verify whether a system is weak fair with respect to a, we only need to verify the property $\Diamond\Box\ a\ is\ enabled \Rightarrow \Box\Diamond\ a\ is\ engaged$.

Given a property ϕ, model checking is all about searching for a counterexample. In automata-based model checking, the negation of ϕ is translated to an equivalent Büchi automaton, which is then composed with the LTS representing the system for analysis. There is a counterexample if and only if there exists an infinite execution which is accepting to the Büchi automaton. Model checking with fairness is to search for an infinite execution which is accepting to the Büchi automaton and at the same time satisfies the fairness constraints. In the following, we present a unified algorithm to verify whether a system is feasible under different fairness constraints. A system

[7] This definition is slightly changed from [11] as so to suit the setting of LTS. Nonetheless, both capture the same intuition.

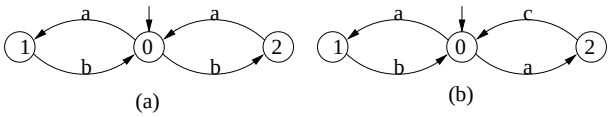

Fig. 4. strong local vs. strong global fair

is feasible if and only if there exists at least one infinite execution which satisfies the fairness constraints. Applied to the product of the system and the Büchi automaton, the algorithm can be easily extended to do model checking with fairness.

3.1 Feasibility Checking as Loop Searching

Without loss of generality, we assume that the systems contain only finite states. An LTS $L = (S, init, \rightarrow)$ contains a loop if and only if there exists a partial run $E = \langle s_0, a_0, s_1, a_1, \cdots, s_i, a_i, \cdots, s_{j-1}, a_{j-1}, s_j \rangle$ such that $s_0 = init$ and for all k which satisfies $0 \leq k < j$ such that $s_k \xrightarrow{a_k} s_{k+1}$ and $s_i = s_j$. By a simple argument, it can be shown that the system contains an infinite execution if and only if there exists a loop. To be feasible under certain fairness, the looping execution must satisfy additional constraints. The proof of the following propositions are straightforward.

Proposition 1. *An LTS $L = (S, init, \rightarrow)$ is feasible under weak fairness assumption, if and only if, there exists a loop $E = \langle s_0, a_0, s_1, a_1, \cdots, s_i, a_i, \cdots, s_{j-1}, a_{j-1}, s_j \rangle$ which satisfies the following: if action a is enabled at* every *state s_k where $i \leq k < j$, then there exists $a_m = a$ where $i < m < j$.*

The condition states that if an action a is enabled at *every* state during the loop, then it must be engaged during the loop. The proposition is easily proved by contradiction.

Proposition 2. *An LTS $L = (S, init, \rightarrow)$ is feasible under strong local fairness if and only if there exists a loop $E = \langle s_0, a_0, s_1, a_1, \cdots, s_i, a_i, \cdots, s_{j-1}, a_{j-1}, s_j \rangle$ which satisfies the following: if action a is enabled at* some *state s_k where $i \leq k < j$, then there exists $a_m = a$ where $i < m < j$.*

The condition states that if a is enabled at *some* state during the loop, then it must be engaged during the loop. Otherwise, a is infinitely often enabled but never engaged.

Proposition 3. *An LTS $L = (S, init, \rightarrow)$ is feasible under strong global fairness, if and only if, there exists a loop $E = \langle s_0, a_0, s_1, a_1, \cdots, s_i, a_i, \cdots, s_{j-1}, a_{j-1}, s_j \rangle$ which satisfies the following: if the step $s_k \xrightarrow{a} s'$ is enabled for some k such that $i \leq k < j$, then there exists m which satisfies $s_m = s_k$, $a_m = a$ and and $s_{m+1} = s'$.*

The condition states that if a step is enabled during the loop, the same step must be taken during the loop.

In addition, a loop is fair with respect to process-level weak fairness (process-level strong fairness) if and only if every process makes some progress during the loop if it

is always (sometimes) possible. Because of the propositions above, a system is feasible (with respect to certain fairness) if and only if there exists a loop which satisfies the fairness. Feasibility checking is hence reduced to loop searching. By a simple argument, it can be shown that a system is feasible if and only if there exists at least one *strongly connected subgraph* in the state graph which satisfies the respective fairness assumption. A strongly connected subgraph satisfies a fairness constraint if and only if the loop which traverses through all states and transitions of the subgraph satisfies the fairness assumption.

3.2 Feasibility Checking Algorithm

There are two groups of methods for loop searching. One is based on nested depth-first-search (DFS) and the other is based on identifying strongly connected components (SCC). Nested DFS has been implemented in SPIN. The basic idea is to perform one DFS first to reach a target state (i.e., an accepting state in the setting of Büchi automata) and then perform 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 [18]. Nonetheless, it is not suitable for verification under fairness assumptions [18], as 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 [13]. In this work, we extend the existing SCC-based model checking algorithms [13] to cope with fairness.

Figure 5 presents our generic algorithm for feasibility checking under fairness constraints. It is based on Tarjan's algorithm for identifying SCCs in linear time (in the number of graph edges). 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, the search is over. Otherwise, the SCC is partitioned into multiple smaller strongly connected subgraphs, which are then checked recursively one by one.

Assume for now, we have a set of states S and a set of transitions T. At line 1, a set $visited$, which stores the set of visited states, is initialized to be empty. Inside the main loop from line 1 to 13, at line 2 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 softwares 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 [13].

For instance, given the LTS presented in Figure 6, there are two SCCs, i.e., one composed of state 1 only and the other composed of state 0, 2 and 3. The order in which SCCs are found is irrelevant to the correctness of the algorithm. If state 2 is explored before state 1, at line 2, scc_states is the set of states in the SCC (i.e., state 0, 2 and 3).

At line 4, we mark scc_states as visited so that the SCC is not examined again. The method $prune$ (at line 5) is used to prune *bad states* from the SCC. Bad states are the reasons why the SCC is not fair. For instance, given the LTS in Figure 6 and the SCC composed of state 0, 2 and 3, state 0 (where the action a is enabled) is a bad state under strong local fairness because action a is never engaged in the SCC (i.e., no

```
procedure feasible(S, T)
0.  visited = ∅;
1.  while there are states not in visited        – while there are still un-explored states
2.        let scc_states = findSCC(S, T);        – identify one SCC at a time
3.        size = #scc_states;
4.        visited = visited ∪ scc_states;        – mark all states in the SCC visited
5.        prune(scc_states, T);                  – prune bad states
6.        if size == #scc_states;                – if no state in the SCC is pruned
7.              generate a feasible path;
8.              return true;                     – a loop is found; the system is feasible
9.        endif
10.       if feasible(scc_states, T)             – recursively check
11.             return true;                     – a fair SCC is found in the remaining
12.       endif
13. endwhile
14. return false;
```

Fig. 5. feasibility checking algorithm

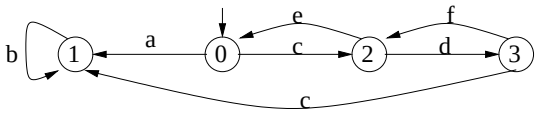

Fig. 6. feasibility checking example

a-transition in the SCC). State 3 is a bad state under 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. We highlight that by simply modifying the $prune$ method, the algorithm can be used to handles different fairness. For instance, Figure 7, 8 and 9 present the respective $prune$ method for weak fairness, strong local fairness and strong global fairness.

At line 1 in Figure 7, the set of actions which label some transition between two states in the SCC is collected into set $engaged$. Line 2 to 5 collects all actions which are enabled at every state in the SCC into set $alwaysEnabled$. If $alwaysEnabled$ is not a subset of $engaged$ (line 7), there exists some action which is always enabled but never engaged and hence the SCC is not weak fair. If the SCC does not satisfy weak fairness, its subgraphs do not either because the action is always enabled in any of its subgraphs. All states are then pruned at line 7 if the SCC is not weak fair. Otherwise, no state is. The pruning works differently for strong local fairness. At line 4 of Figure 8, a state is pruned if and only if there is an action enabled at this state but never engaged in the SCC. By pruning the state, the action may be never enabled in the SCC and therefore

not required to be engaged. The pruning method for strong global fairness is presented in Figure 9. At line 4 of Figure 9, all states are pruned if there is a transition from a state in the SCC to a state not in the SCC. It can be shown that if an SCC is not strong global fair, then its subgraphs are not either (since the subgraph must contain a step to a pruned state and therefore can not be strong global fair). Following the idea, a $prune$ method for process-level weak/strong fairness can be easily defined. The time complexity of the $prune$ methods are linear in the number of edges.

If the SCC does satisfy the fairness assumption, no state is pruned and thus the size of the SCC remains the same (line 6 of Figure 5). For instance, the maximum SCC containing state 1 in Figure 6 is fair (with respects to any kind of fairness). In such a case, a fair loop (which traverses all states/transitions in the SCC) is generated at line 7 and then we conclude 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 (refer to [21]). 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 8) and the other is all states are pruned (at line 14). If the recursive call returns false, there is no fair subgraph and we continue with another SCC until there is no state left.

Given the LTS in Figure 5, the path containing only state 1 is returned as a feasible path. As discussed above, under strong local fairness, state 0 is pruned from the SCC containing state 0, 2, and 3. After that the only remaining strongly connected subgraph contains state 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.

In the worst case (i.e., the whole system is strongly connected and only one state is pruned every time), a node may be visited 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. We remark that the time complexity for verification with no fairness, weak fairness and strong global fairness are similar, i.e., both linear in $\#T$ since all states in one SCC are discarded all together in all cases. SPIN's model checking algorithm under process-level weak fairness increases the run-time expense of a verification run by a factor that is linear in the number of running processes. In comparison, our algorithm is less expensive. Verification under strong local fairness is in general expensive. However, our experience suggests that the worse case scenario is rare in practice. Instead of the detailed complexity analysis[8], we show its efficiency using real systems in Section 5.

The algorithm is terminating because the number of visited states and pruned states are monotonically increasing. It can be shown that method $prune$ is sound, i.e., if there is a fair strongly connected subgraph in the given set of states, it remains in the pruned set of states. This can be established by Proposition 1, 2 and 3 (e.g., by contradiction). The following theorem then proves the soundness of the algorithm.

Theorem 1. *Given a fairness assumption, a finite-state system is feasible under the fairness iff* feasible *returns true.*

[8] One way is to build a tree where each node is an SCC in the system graph and then give a tighter bound in the height of the tree.

```
procedure prune(scc_states, T)
1. let engaged = GetEngagedActs(scc_state, T);
2. alwaysEnabled = Σ;
3. foreach s ∈ scc_states
4.       alwaysEnabled = alwaysEnabled ∩ enabled(s)
5. endeach
6. if alwaysEnabled ⊈ engaged
7.       scc_states = ∅;
8. endif
```

Fig. 7. $prune$ for weak fairness

```
procedure prune(scc_states, T)
1. let engaged = GetEngagedActs(scc_state, T);
2. foreach s ∈ scc_states
3.       if enabled(s) ⊈ engaged
4.             remove s from scc_states
5.       endif
6. endeach
```

Fig. 8. $prune$ for strong local fairness

Proof: As discussed above, the system is feasible if and only if there exists a loop which satisfies the fairness assumption. Equivalently, there exists such a loop if and only if there exists a strongly connected subgraph which satisfies the fairness assumption. If there is such a graph and itself is an SCC, it must be found (by the correctness of Tarjan's algorithm and the correctness of method $prune$) and the algorithm returns true and, the theorem holds. If it is contained in one (and only one) SCC, by the correctness of method $prune$, 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 true and the theorem holds. It is straightforward to show that the algorithm only returns true when a strongly connected subgraph which satisfies the fairness assumption is found. Thus, the theorem is true. □

Based on the above theorem, the existing automata-based on-the-fly model checking algorithm [18] can be extended straightforwardly to perform model checking under fairness. Given an LTL formula, we negate it and translate it into an equivalent Büchi automaton using existing approaches [12]. We then compute on-the-fly the synchronous product of the Büchi automaton and the system (in which the system and the automaton must make a move simultaneously). The algorithm presented in Figure 5 is then used to identify one fair strongly connected subgraph at a time. If a fair subgraph is accepting

```
procedure prune(scc_states, T)
1. foreach s ∈ scc_states
2.     foreach (s, a, s') such that s -a-> s'
3.         if (s, a, s') is not a transition in scc
4.             scc_states = ∅;
5.             return;
6.         endif
7.     endeach
8. endeach
```

Fig. 9. $prune$ for strong global fairness

to the Büchi automaton (by checking whether the subgraph contains an Büchi accepting state), it means that the subgraph satisfies the fairness constraints and yet fails the LTL property. Thus, we generate the feasible path as a counterexample. If no fair subgraph is accepting, the property is true. We remark *un-fair* loops which represents unrealistic system executions are eliminated before checking whether they are accepting or not.

4 Action Annotated Fairness

In this section, we present an alternative (and more flexible) approach, which allows users to associate fairness to only part of the systems or associate different parts with different fairness constraints. The motivation is twofold.

Firstly, previous approaches treat every action or state equally, i.e., fairness is applied to every action/state. In verification practice, it may be that only certain actions are meant to be fair. For instance, when verifying open systems, fairness/liveness assumptions are often associated with input events from the environment as a way to capture assumptions on the environment. Given the leader detector presented in Section 2.1, if no fairness is applied, the action $oracle$ may never happen and thus violate the requirement (i.e., eventually the detector detects correctly). If weak fairness or even stronger fairness is applied, $oracle$, $guess1$ and $guess2$ all must occur infinitely often since they are always enabled. This is clearly overwhelming. Our remedy is to allow users to associate fairness constraints with individual actions. For instance, the eventual leader detector can be modeled as in Figure 10. The action $oracle$ is annotated as $wf(oracle)$, which stands for weak fairness. It is used to capture the requirement that $oracle$ (if always enabled) must eventually occur. We remark that $guess1$ and $guess2$ are not annotated because there are no such requirements. In PAT, a number of different fairness constraints may be used to annotate actions. In the following, we examine three of them.

- Unconditional action fairness is written as $f(a)$. An execution of the system is fair if and only if a occurs infinitely often.
- Weak action fairness is written as $wf(a)$. An execution of the system is fair if and only if a occurs infinitely often given it is always enabled from some point on.

$$
\begin{aligned}
Detector() = \ & \mathbf{wf}(oracle)\{correct = 1;\ \} \rightarrow Detector() \\
& [] \ guess1\{guess = true;\ \} \rightarrow Detector() \\
& [] \ guess2\{guess = false;\ \} \rightarrow Detector()
\end{aligned}
$$

Fig. 10. annotated leader detector

– Strong action fairness is written as $sf(a)$. An execution of the system is fair if and only if a occurs infinitely often given it is enabled infinitely often.

Unconditional action fairness does not depend on whether the action is enabled or not, and therefore, is stronger than weak/strong action fairness. It may be used to annotate actions which are known to be periodically engaged. For instance, the following process models a natural clock.

$$Clock() = f(tick)\{x = x + 1;\ \} \rightarrow Clock();$$

where x is a discrete clock variable. By annotating $tick$ with unconditional fairness, we require that the clock must progress infinitely and the system (in which the clock and other components execute in parallel) disallows unrealistic *timelock*, i.e., execution of infinite actions which takes finite time. Unconditional fairness (like other actions annotations) can be used to mechanically reduce the size of the property. For instance, given the property $\Box\Diamond\ a \Rightarrow \Box\Diamond\ b$. We may mechanically annotate a in the model with unconditional fairness and verify $\Box\Diamond\ b$ instead. The semantics of weak (strong) action fairness is similar to weak (strong local) fairness defined in Section 2.2 except it is associated with individual actions (by contrast to all actions)[9]. Action annotated fairness may be viewed as the dual image of accepting states in automata theory, e.g., same as only selected states are marked accepting, only selected actions are annotated.

The other motivation of action annotated fairness is that it makes partial order reduction possible (to some extent) for model checking with strong local/global fairness. The feasibility algorithm in Figure 5, an SCC-based explicit model checking algorithm, undoubtedly suffers from state space explosion, especially when the whole system is strongly connected. Partial order reduction is one of the most effective techniques to tackle the problem, which sometimes works surprisingly well for distributed systems. The idea behind partial order reduction is that actions may be independent of each other and the ordering of their occurrences in an execution is irrelevant to the truth of certain property. As a result, it might not be necessary to consider all enabled actions at a given state, but only a certain subset which are independent of the rest. Thus, instead of working with the full state graph, a reduced state graph is constructed.

For instance, assume that action a and b are independent and the property to verify is deadlock-freeness, it is sufficient to explore only one of the two outgoing transitions at state 0 in the LTS of Figure 4(a). For classic model checking, a set of conditions which the subset of enabled actions has to fulfill have been proposed to guarantee sound

[9] Because strong global fairness concerns with both actions and states. No corresponding action annotation is defined.

verification against 'X'-free LTL formulas. Efficient heuristic algorithms which calculates an (over-)approximation of the subset are explored as well [7]. One such heuristic algorithm has been implemented in PAT.

However, the conditions and algorithms may not work for verification under fairness. Following results proved in [4], it can be shown that partial order reduction is applicable to verification under weak fairness. However, though every strong-local-fair execution in the full state graph has an equivalent execution (up to re-ordering of independent actions) in the reduced state graph, it may not be strong-local-fair and thus verification result over the reduced state graph may not be valid. Worse, with strong global fairness the reduced state graph may not be feasible even if the full state graph is. For instance, let the system be the LTS in Figure 4(a) and assume a and b are independent. The reduced graph may only contain state 0 and 1, which is not feasible under strong global fairness. In [31], it was suggested that by considering actions dependent to each other if they can enable or disable each other, partial order reduction can be applied to some extent for verification under fairness. Nonetheless, in previous approaches, because all actions must be considered, virtually all events become inter-dependent and therefore no reduction is possible. In PAT, partial order reduction is disabled for model checking under strong local/global fairness. Nonetheless, for systems with action annotated fairness, it remains possible to apply partial order reduction to actions which are irrelevant to the fairness annotations.

The algorithm presented in Figure 5 can be applied to check systems with action annotated fairness with slight modification. The basic idea remains, i.e., finding a loop which satisfies the fairness constraints. Only actions with fairness annotations are considered this time (by contrast to all actions). We remark that annotating all actions with weak (strong) fairness is equivalent to associate weak (strong local) fairness with the whole system. The method $findSCC$ is modified to cope with partial order reduction, following the heuristic function in [7]. In addition, we define an action to be *fairness visible* if it enables or disables an action annotated with fairness and require that if the chosen set of actions are a strict subset of enabled actions, the subset must not contain fairness visible actions. The intuition is that independent actions which are irrelevant to the fairness constraints are subject to partial order reduction. Notice that this checking has time complexity linear in the number of enabled actions. The soundness follows from the discussion in [4, 31].

Algorithm $prune$ is also modified to examine only the annotated actions. Figure 11 shows the modified algorithm. An SCC is fair with respect to the action annotated fairness if and only if: all actions which are annotated with unconditional action fairness are contained in the set $engaged$; if an action is annotated with weak action fairness and is enabled at *every* state in the SCC, then the action is contained in $engaged$; and if an action is annotated with strong action fairness and is enabled at *some* state in the SCC, then the action is contained in $engaged$. If an SCC does not satisfy unconditional or weak action fairness, it is abandoned all together (line 11). If a state enables an action annotated with strong action fairness which is never engaged in the SCC, then it is pruned (line 6 to 8). For instance, given the LTS in Figure 6, if action a is annotated with strong local fairness, then state 0 is a bad state. It is not if it is annotated with

procedure $prune(scc\_states,\ T)$
1. **let** $engaged = GetEngagedActs(scc\_states,\ T)$;
2. **let** $unconditional = \{e \mid \text{exists } f(e)\}$;
3. **let** $weak = \Sigma$;
4. **foreach** $s \in scc\_states$
5. $\quad weak = weak \cap \{e \mid wf(e) \in enabled(s)\}$;
6. $\quad$ **if** $\{e \mid sf(e) \in enabled(s)\} \not\subseteq engaged$
7. $\quad\quad$ remove s from scc_states;
8. $\quad$ **endif**
9. **endeach**
10. **if** $unconditional \not\subseteq engaged$ or $weak \not\subseteq engaged$
11. $\quad scc\_states = \varnothing$;
12. **endif**

Fig. 11. $prune$ for action annotated fairness

no or weak fairness. By a similar argument (to the proof of Theorem 1), we show the soundness of the algorithm.

5 Implementation and Experiments

PAT is designed for systematic validation of distributed systems using state-of-art model checking techniques. It has three main components. Its main functionalities include simulation, explicit on-the-fly model checking, and verification with fairness. 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. Figure 12 shows the user interface of the editor and the model checker.

5.1 Experiments on Population Protocols

In the following, we show its performance over both benchmark systems as well as recently developed systems where fairness is required. 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 2 summarizes the verification statistics over recently developed population protocols. Notice that $-$ means either out of memory or more than 4 hours. The protocols include leader election for complete networks (LE_C) [11], for rooted trees (LE_T) [5], for odd sized rings (LE_OR) [19], for network rings (LE_R) [11] and token circulation for network rings (TC_R) [2]. Correctness of all these algorithms relies on fairness. Notice that fairness is applied to the whole system for simplicity. For soundness reasons, partial order reduction is applied for verification under no or weak fairness, but not strong local fairness or strong global fairness.

As discussed in Section 2.2, process-level weak fairness (supported in SPIN) is different than weak fairness. In order to compare PAT with SPIN for verification with weak

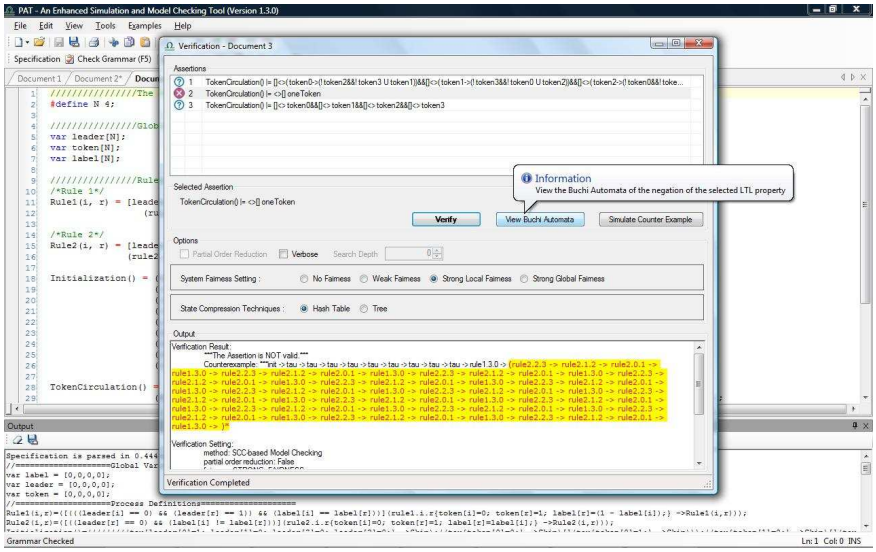

Fig. 12. process analysis toolkit

fairness, we twist the models so that each action in population protocols is modeled as a SPIN process. By a simple argument, it can be shown that for such models, weak fairness is equivalent to process-level weak fairness. However, checking under process-level weak fairness in SPIN increases the verification time by a factor that is linear in the number of processes. By modeling each action as a process, we increase the number of processes and therefore un-avoidably increase the SPIN verification time by a factor that is constant (in the number of actions per process for network rings) or linear (in the number of network nodes for complete network). SPIN has no support for strong local or global fairness. Thus, the only way to do verification under strong local/global fairness in SPIN is to encode the fairness constraints as part of the property. However, even for the smallest network (with 3 nodes), SPIN needs significant amount of time constructing (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 local fairness can be transformed to weak fairness by paying the price of one Boolean variable [21]. Nonetheless, the property again needs to be augmented with additional clauses after the translation, which is again infeasible.

We remark that fairness does play an important in these protocols. 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 weak fairness, whereas the rest of the algorithms require strong global fairness. It is thus important to be able to verify systems under strong local/global fairness. Notice that the token circulation algorithm for network rings (TC_R) functions correctly for a network of size 3 under weak fairness. Nonetheless, weak fairness is not sufficient for a network wit more nodes, as shown in the table. The reason is that a particular sequence of message exchange which satisfies

Model	Property	Size	Weak Fair			Strong Local Fair			Strong Global Fair		
			Result	PAT	SPIN	Result	PAT	SPIN	Result	PAT	SPIN
LE_C	$\Diamond\Box oneleader$	3	Yes	0.1	0.4	Yes	0.1	–	Yes	0.2	–
LE_C	$\Diamond\Box oneleader$	4	Yes	0.8	4.3	Yes	0.9	–	Yes	0.7	–
LE_C	$\Diamond\Box oneleader$	5	Yes	5.2	35.7	Yes	6.1	–	Yes	4.9	–
LE_C	$\Diamond\Box oneleader$	6	Yes	31.6	229	Yes	38.2	–	Yes	29.7	–
LE_C	$\Diamond\Box oneleader$	7	Yes	167.8	1190	Yes	199.5	–	Yes	158.3	–
LE_C	$\Diamond\Box oneleader$	8	Yes	819.6	5720	Yes	863.4	–	Yes	785.3	–
LE_T	$\Diamond\Box oneleader$	3	Yes	0.1	< 0.1	Yes	0.1	–	Yes	0.1	–
LE_T	$\Diamond\Box oneleader$	5	Yes	0.3	0.7	Yes	0.3	–	Yes	0.3	–
LE_T	$\Diamond\Box oneleader$	7	Yes	1.9	7.6	Yes	1.9	–	Yes	1.8	–
LE_T	$\Diamond\Box oneleader$	9	Yes	13.3	62.3	Yes	13.5	–	Yes	13.0	–
LE_T	$\Diamond\Box oneleader$	11	Yes	91.1	440	Yes	93.1	–	Yes	88.7	–
LE_T	$\Diamond\Box oneleader$	13	Yes	688.1	3200	Yes	738.7	–	Yes	671.4	–
LE_OR	$\Diamond\Box oneleader$	3	No	0.2	0.3	No	0.2	–	Yes	17.7	–
LE_OR	$\Diamond\Box oneleader$	5	No	1.9	8.7	No	2.4	–	–	–	–
LE_OR	$\Diamond\Box oneleader$	7	No	38.3	95	No	27.9	–	–	–	–
LE_R	$\Diamond\Box oneleader$	3	No	0.2	< 0.1	No	0.2	–	Yes	3.2	–
LE_R	$\Diamond\Box oneleader$	4	No	0.5	< 0.1	No	1.1	–	Yes	51.4	–
LE_R	$\Diamond\Box oneleader$	5	No	1.3	< 0.1	No	4.6	–	Yes	1359.0	–
LE_R	$\Diamond\Box oneleader$	6	No	0.5	0.2	No	2.2	–	–	–	–
LE_R	$\Diamond\Box oneleader$	7	No	1.3	0.6	No	6.6	–	–	–	–
LE_R	$\Diamond\Box oneleader$	8	No	3.3	1.7	No	19.2	–	–	–	–
TC_R	$\Diamond\Box onetoken$	3	Yes	0.1	< 0.1	Yes	0.1	–	Yes	0.1	–
TC_R	$\Diamond\Box onetoken$	5	No	0.1	< 0.1	No	0.1	–	Yes	0.7	–
TC_R	$\Diamond\Box onetoken$	7	No	0.2	0.1	No	0.3	–	Yes	18.9	–
TC_R	$\Diamond\Box onetoken$	9	No	0.5	0.2	No	0.5	–	Yes	982.7	–
TC_R	$\Diamond\Box onetoken$	11	No	0.9	0.8	No	1.1	–	–	–	–
TC_R	$\Diamond\Box onetoken$	13	No	1.7	1.6	No	2.2	–	–	–	–

Table 2. population protocols experiments: with Core 2 CPU 6600 at 2.40GHz and 2GB RAM

the weak fairness constraint only needs the participation of at least 4 network nodes. This suggests that our improvement in terms of the performance and ability to handle different forms of fairness has its practical values. We highlight that previously unknown bugs in implementation of the leader election algorithms for odd-sized ring [19] have been revealed using PAT (see next section).

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., LE_R and TC_R under weak fairness or strong local 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^{5*N} where N is network size) and a counterexample is only present for particular initial configurations. Secondly, verification with strong local fairness is more expensive than verification with no fair, 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 outperforms the current practice of verification under fairness. PAT offers comparably better performance on verification with weak fairness (e.g., LE_C and LE_T) and makes it feasible to verify with strong local/gloabl fairness. This allowed us to discover bugs in systems functioning with strong fairness. Experiments on LE_C and LE_T (for which the property is true under any fairness) show minor computational overhead for handling a stronger fairness.

5.2 Bug Report

In this section, we study the leader election protocol in oriented odd rings in detail and report the bug that has been discovered for the first time. The following description is taken from [19, 2]. Supposing each node has a *label* bit, a maximal sequence of alternating labels is called a segment. According to the orientation of the ring, the head and tail of a segment can be defined in a natural way. One edge of the form $(0, 0)$ or $(1, 1)$ connecting the tail of one segment to the head of another segment is called a *barrier* edge. For a node u in a ring, it has four state components: $leader[u]$ states whether the node is a leader, $label[u]$ is its label, $probe[u]$ is 1 if u holds a probe token, and $phase[u]$ alternates between 0 and 1 to make each barrier alternate between firing a probe and moving forward. The protocol consists of several parts. In the basic part, the barriers move clockwise around the ring. Each barrier advances by flipping the label bit of the second node on the barrier (the head of the next segment). When two barriers collide, they cancel out each other. Because the ring size is odd, there is always at least one barrier. In the rest of the protocol, the leader bullet and probe marks are manipulated. Probes are sent out by the barrier in a clockwise direction and absorbed by any leader they run into. If a probe meets the barrier on its way back, it is converted to leader. Leaders fire *bullets* counter-clockwise around the ring. Bullets are absorbed by the barrier, but they kill any leaders they encounter along the way. More detailed discussion of the protocol is referred to [19, 2]. The description of an interaction between an initiator u and a responder v in the protocol (taken from [19], p.66) is presented as follows:

Leader election protocol for odd rings.

if $label[u] = label[v]$ **then**
  if $probe[u] = 1$ **then** $leader[u] \leftarrow 1$; $probe[u] \leftarrow 0$ **endif**
  $bullet[v] \leftarrow 0$
  if $phase[u] = 0$ **then** $phase[u] \leftarrow 1$; $probe[v] \leftarrow 1$
  elseif $probe[v] = 0$ **then**
    $label[v] = \neg label[v]$; $phase[v] \leftarrow 0$
  endif
elseif $leader[v] = 1$ **then**
  if $bullet[v] = 1$ **then** $leader[v] \leftarrow 0$
  else $bullet[u] \leftarrow 1$ **endif**
else
  if $bullet[v] = 1$ **then** $bullet[v] \leftarrow 0$; $bullet[u] \leftarrow 1$ **endif**
  if $probe[u] = 1$ **then** $probe[u] \leftarrow 0$; $probe[v] \leftarrow 1$ **endif**
endif

The protocol is modeled in PAT. We have totally eleven ($act1.u.v$ up to $act11.u.v$) case splits according to the protocol description. For example, the condition of the action $act1.u.v$ collects the conditions at the first, second and fourth line in the description and the updates of variables at the second, third, and fourth line, correspondingly. The initialization of the model is taken care of at line 17, it captures any possible evaluations of the variables. Line 9 defines how nodes interact in an oriented ring. Line 20 defines a predicate that there is one leader in the network. Line 21 claims that the protocol eventually self-stabilize to a unique leader existing in the network.

Counterexample. We have analyzed this protocol in PAT, and found one counterexample. We consider a ring of size three, nodes are numbered as 0, 1 and 2. The counterexample found by PAT can be described as follows: it is an infinite execution containing a loop, u is the node for the initiator and v for the responder of one interaction according to the protocol description. The execution can start with a configuration $bullet = [1,1,1], label = [1,1,1], leader = [1,1,0], phase = [1,1,1], probe = [1,1,0]$.

1. since $label[2] = label[0]$, $probe[2] = 0$, $phase[2] = 1$ and $probe[0] = 1$, we have $bullet[0] \leftarrow 0$. ($u = 2$ and $v = 0$)
2. since $label[0] = label[1]$, $probe[0] = 1$, $phase[0] = 1$ and $probe[1] = 1$, we have $leader[0] \leftarrow 1$, $probe[0] \leftarrow 0$, and $bullet[1] \leftarrow 0$. ($u = 0$ and $v = 1$)
3. since $label[2] = label[0]$, $probe[2] = 0$, $phase[2] = 1$ and $probe[0] = 0$, we have $bullet[0] \leftarrow 0$, $label[0] \leftarrow 1 - label[0]$, and $phase[0] \leftarrow 0$. ($u = 2$ and $v = 0$)
4. since $label[1] = label[2]$, $probe[1] = 1$, $phase[1] = 1$ and $probe[2] = 0$, we have $leader[1] \leftarrow 1$, $probe[1] \leftarrow 0$, $bullet[2] \leftarrow 0$, $label[2] \leftarrow 1 - label[2]$ and $phase[2] \leftarrow 0$. ($u = 1$ and $v = 2$)
5. since $label[2] = label[0]$, $probe[2] = 0$ and $phase[2] = 0$, we have $bullet[0] \leftarrow 0$, $phase[2] \leftarrow 1$ and $probe[0] \leftarrow 1$. ($u = 2$ and $v = 0$)

Now, we have reached a configuration with $bullet = [0,0,0], label = [0,1,0], leader = [1,1,0], phase = [0,1,1], probe = [1,0,0]$.[10] From here, we have a loop. Within this loop, all actions enabled at reachable configurations of the loop are executed. But these configurations contain more than two leaders. Hence, this infinite execution is global fair but not self-stabilizing for leader election. The loop is given below.

1. since $label[2] = label[0]$, $probe[2] = 0$, $phase[2] = 1$ and $probe[0] = 1$, we have $bullet[0] \leftarrow 0$. ($u = 2$ and $v = 0$)
2. since $label[0]! = label[1]$, $leader[1] = 1$ and $bullet[1] = 0$, we have $bullet[0] \leftarrow 1$. ($u = 0$ and $v = 1$)
3. since $label[0]! = label[1]$, $leader[1] = 1$ and $bullet[1] = 0$, we have $bullet[0] =\leftarrow 1$. ($u = 0$ and $v = 1$)
4. since $label[2] = label[0]$, $probe[2] = 0$, $phase[2] = 1$ and $probe[0] = 1$, we have $bullet[0] =\leftarrow 0$. ($u = 2$ and $v = 0$)

[10] As the protocol is self-stabilizing, the counterexample can start directly from here. We keep the first part just to faithly represent the infinite trace found by PAT.

Model	Property	Result	Fairness	PAT	SPIN
dp(10)	$\Box\Diamond eat0$	No	no	< 0.1	< 0.1
dp(13)	$\Box\Diamond eat0$	No	no	0.1	< 0.1
dp(15)	$\Box\Diamond eat0$	No	no	0.1	< 0.1
dp(10)	$\Box\Diamond eat0$	No	strong global fairness (whole system)	< 0.1	–
dp(13)	$\Box\Diamond eat0$	No	strong global fairness (whole system)	0.1	–
dp(15)	$\Box\Diamond eat0$	No	strong global fairness (whole system)	0.1	–
ms(10)	$\Box\Diamond work0$	Yes	strong local fairness (whole system)	11.4	–
ms(12)	$\Box\Diamond work0$	Yes	strong local fairness (whole system)	132.2	–
ms(100)	$\Box\Diamond work0$	Yes	strong local fairness (annotations)	5.5	–
ms(200)	$\Box\Diamond work0$	Yes	strong local fairness (annotations)	27.4	–
peterson(3)	bounded bypass	Yes	weak fairness (whole system)	0.1	1.25
peterson(4)	bounded bypass	Yes	weak fairness (whole system)	2.4	> 671
peterson(5)	bounded bypass	Yes	weak fairness (whole system)	112.6	–

Table 3. experiment results on benchmarks

The last step in the loop leads us back to the starting configuration of the loop. We have communicated this counterexample to the author of [19], it is confirmed as a valid counterexample which has escaped simulations of the protocol [20]. The reason to the counterexample is the following [20]. In the explanation of the protocol, it says that "probes are sent out by the barrier in a clockwise direction and absorbed by any leader they run into". The second half of the sentence is missing from the pseudo code description. The protocol also requires consistent ordering of the position of tokens within each node (in the order of leader, bullet, and probe clockwise). A barrier edge should only generate a probe at the responder if the responder is not a leader. Otherwise, the probe would be able to pass the leader token. In the description, this property is not preserved either. Modifications of the description have been made in [2]. We also model the revised version of the protocol, and find no counterexample. By this case study, we emphasize that without the newly developed model checking algorithm for efficient verification under (global) fairness, it is impossible to find such an error in a pseudo code description of a population protocol, especially when a protocol tends to be intuitively more complicated.

5.3 Experiments on Benchmark Systems

Table 3 shows verification statistics of benchmark systems to show other aspects of PAT. 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 identifying a counterexample. The reason is both due to the particular order of exploration and the difference between model checking based on nested depth-first-search and model checking based on identifying SCCs. PAT's algorithm relies on identifying SCCs. If a large portion of the system is strongly con-

nected, it takes 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. Milner's cyclic scheduler algorithm ($ms(N)$ for N processes) is a showcase for the effectiveness of partial order reduction. We apply fairness in two different ways, i.e., one applying strong local fairness to the whole system and the other applying only to inter-process communications. In the latter case, partial order reduction allows us to prove the property over a much larger number of processes (e.g., 200 vs 12). Peterson's mutual exclusive algorithm ($peterson(N)$) requires process-level weak fairness to guarantee bounded by-pass [1], i.e., if a process requests to enter the critical section, it eventually will. The property is verified under weak fairness in PAT and process-level weak fairness in SPIN, with modifying the model. PAT outperforms SPIN in this setting as well.

6 Conclusion

The contribution of the paper is threefold. Firstly, we improved and unified SCC-based model checking algorithms to handle a variety of fairness constraints. In particular, we studied how to perform model checking effectively with strong global fairness, which has never been studied in the model checking community. Secondly, two different ways of applying fairness have been investigated and supported, namely, one applying fairness to the whole system and the other applying fairness to only the relevant actions. Both approaches have been motivated by practical reasons. Thirdly, we significantly extend PAT (started as a testbed for [34]) to be a reliable efficient environment for verification with or without fairness. PAT has been applied to a variety of distributed systems and previously-unknown flaws have been identified.

We are actively developing PAT. One future work of particular interest is to investigate refinement with fairness constraints. The motivations are that refinement under a fair scheduler or in a distributed system is rather different and interesting. For instance, trace refinement with weak fairness prevents removing a transition which is always enabled and trace refinement under strong global fairness prevents removing a nondeterministic choice. The consequence of a fair scheduler over program refinement is worth investigating. Another future work is to identify develop efficient algorithms for refinement checking with fairness. Other possible future works include investigating how to handle infinite data states, migrating the algorithms to a general software model checker, etc.

Acknowledgements

We thank Michael Fischer and Jiang Hong for the discussion on fairness and leader election protocols, and Deng Yu Xin for his comments on an early version of the article. This work is partially supported by the research project "Sensor Networks Specification and Validation" (R-252-000-320-112) funded by Ministry of Education, Singapore.

References

1. K. Alagarsamy. Some Myths About Famous Mutual Exclusion Algorithms. *SIGACT News*, 34(3):94–103, 2003.
2. D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. In *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS 2005)*, volume 3974 of *Lecture Notes in Computer Science*, pages 103–117, 2005.
3. D. Angluin, M. J. Fischer, and H. Jiang. Stabilizing Consensus in Mobile Networks. In *Proceedings of the 2006 International Conference on Distributed Computing in Sensor Systems (DCOSS'06)*, volume 4026 of *Lecture Notes in Computer Science*, pages 37–50, 2006.
4. L. Brim, I. Cerná, P. Moravec, and J. Simsa. On Combining Partial Order Reduction with Fairness Assumptions. In *Proceedings of the 11th International Workshop Formal Methods: Applications and Technology (FMICS 2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 84–99, 2006.
5. D. Canepa and M. Potop-Butucaru. Stabilizing Token Schemes for Population Protocols. *Computing Research Repository (CoRR)*, abs/0806.3471, 2008.
6. S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/Event-Based Software Model Checking. In *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*, pages 128–147, 2004.
7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.
8. J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 342–359. Springer, 2006.
9. J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *Lecture Notes in Computer Science*, pages 226–245. Springer, 2006.
10. E. A. Emerson and C.-L. Lei. Modalities for Model Checking: Branching Time Logic Strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.
11. M. J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS 2006)*, volume 4305 of *Lecture Notes in Computer Science*, pages 395–409. Springer, 2006.
12. P. Gastin and D. Oddoux. Fast LTL to Büchi Automata Translation. In *Proc. of the 13th Inter. Conf. on Computer Aided Verification (CAV 2001)*, pages 53–65. Springer, 2001.
13. J. Geldenhuys and A. Valmari. More efficient on-the-fly LTL verification with Tarjan's algorithm. *Theoritical Computer Science*, 345(1):60–82, 2005.
14. D. Giannakopoulou, J. Magee, and J. Kramer. Checking Progress with Action Priority: Is it Fair? In *Proceedings of the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 1999)*, volume 1687 of *Lecture Notes in Computer Science*, pages 511–527, 1999.
15. R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A New Heuristic for Bad Cycle Detection Using BDDs. *Formal Methods in System Design*, 18(2):131–140, 2001.
16. M. R. Henzinger and J. A. Telle. Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory (SWAT 1996)*, pages 16–27, 1996.
17. C. A. R. Hoare. *Communicating Sequential Processes*. International Series on Computer Science. Prentice-Hall, 1985.

18. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003.
19. H. Jiang. *Distributed Systems of Simple Interacting Agents*. PhD thesis, Yale University, 2007.
20. H. Jiang. Personal communications, 2008.
21. Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model Checking with Strong Fairness. *Formal Methods and System Design*, 28(1):57–84, 2006.
22. N. Klarlund. An n log n Algorithm for Online BDD Refinement. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, pages 107–118. Springer-Verlag, 1997.
23. M. Z. Kwiatkowska. Event Fairness and Non-interleaving Concurrency. *Formal Aspects of Computing*, 1(3):213–228, 1989.
24. L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.
25. L. Lamport. Fairness and Hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.
26. T. Latvala and K. Heljanko. Coping with Strong Fairness. *Fundamenta Informaticae*, 43(1–4):175–193, 2000.
27. D. J. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In *Proceedings of the 8th Colloquium on Automata, Languages and Programming (ICALP 1981)*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277, 1981.
28. O. Lichtenstein and A. Pnueli. Checking That Finite State Concurrent Programs Satisfy Their Linear Specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'85)*, pages 97–107. ACM, 1985.
29. M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 362–371. ACM, 2008.
30. J. Pang, Z. Q. Luo, and Y. X. Deng. On Automatic Verification of Self-stabilizing Population Protocols. In *Proceedings of the Second IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE 2008)*, pages 185–192. IEEE, 2008.
31. D. Peled. All from One, One for All: on Model Checking Using Representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, 1993.
32. A. Pnueli and Y. Sa'ar. All You Need Is Compassion. In *Proceedings of the Ninth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2008)*, volume 4905 of *Lecture Notes in Computer Science*, pages 233–247, 2008.
33. J. Sun, Y. Liu, and J. S. Dong. Model checking csp revisited: Introducing a process analysis toolkit. In *Proceedings of the Third International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.
34. J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and verifying event-based fairness enhanced systems. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM 2008)*, volume 5256 of *Lecture Notes in Computer Science*, pages 318–337. Springer, Oct 2008.