Model Checking Population Protocols in PAT with Symmetry Reduction

Jun Pang and Shao Jie Zhang

Abstract The population protocol model has emerged as an elegant computation paradigm for describing mobile ad hoc networks, consisting of a number of mobile nodes that interact with each other to carry out a computation. The interactions of nodes are subject to a global fairness constraint, which plays a vital role in designing self-stabilizing population protocols. In this chapter, we focus on automatic analysis of population protocols with the model checker PAT, as model checking has established itself as an effective system analysis technique. However, applying model checking to practical systems is often limited by state space explosion. To enable our analysis, we first show that global fairness, unlike weak/strong fairness, can be combined with symmetry reduction. We then extend PAT with the proposed technique and demonstrate its usability by verifying a number of recently proposed population protocols.

1 Introduction

The field of distributed algorithms has enjoyed a rapid growth in the last two decades, due to the world-wide development and usage of mobile ad hoc networks. A great number of algorithms have been invented to solve hard problems in mobile ad hoc networks. However, these algorithms are only accessible to the distributed algorithms community, since their specifications and correctness arguments are often given at an informal level. This is insufficient to convince researchers outside the field of the validity of the arguments. If one wants to verify the correctness of some proofs, he has to prove substantial parts or entire sub-results, for which only
informal arguments were given. This has been observed and illustrated in [20] on formal reasoning about the correctness of a distributed consensus algorithm [10].

Formal verification, model checking in specific, has been recognized as an important method to prove the correctness of distributed algorithms formally and automatically. Model checking first builds a finite state space of a formal model of a system, and then verifies a property, written in some temporal logic, through an explicit state space search. Due to the finiteness of the state space, the search always terminates. Hence, model checking is largely automatic. It can produce an answer in a few minutes or even seconds for many models. A counter-example can be generated when the checked property fails to hold. Techniques such as symbolic representation, symmetry reduction, partial order reduction, and predicate abstraction [12], have been developed to deal with the state explosion problem and enhance the scalability of model checking. However, these techniques have not yet made impact on distributed algorithms, mainly because there have not yet been enough examples of non-trivial practical applications.

We apply model checking to self-stabilizing population protocols in this chapter. The population protocol model [2, 19, 5] has emerged as a new elegant computation paradigm for describing mobile ad hoc networks, consisting of a number of mobile nodes that interact with each other to carry out a computation. Each node has only a few states. One essential property of such protocols is that all nodes must eventually converge to the correct output value (or configuration), which is a typical liveness property (something good will eventually happen) in terms of formal verification. To guarantee that such kind of property can be achieved, the interactions of nodes in population protocols are subject to a fairness constraint. The fairness condition is imposed on the adversary to ensure that the protocol makes progress. In population protocols, the required fairness condition will make the system behave nicely eventually, although it can behave arbitrarily for an arbitrarily long period [5]. That is why for population protocols correctness arguments are always rephrased as a property to be satisfied eventually. A number of population protocols have been proposed and studied [2, 4, 19, 25, 3], but most of them only work if global fairness is imposed. For instance, it was shown that without global fairness uniform self-stabilizing leader election in rings is impossible [19].

In formal verification, fairness is typically used to rule out unrealistic runs due to non-determinism, and mainly concerns with a fair resolution of non-determinism in the models. There is a rich literature on how to handle fairness constraints in model checking, see e.g. [30, 27, 33]. However, existing verification algorithms/tools are ineffective with respect to fairness. One way to apply existing model checkers for verification under fairness is to re-formulate the property so that fairness assumptions become premises of the property. This practice is deficient though flexible. Typically, automata-based model checking relies on constructing a Büchi automaton from the property. The size of the Büchi automaton is exponential to the size the property. For example, Spin is a rather popular linear temporal logic (LTL) model checker [24]. The algorithm it uses for generating Büchi automata handles only a limited number of fairness constraints. Pang et al. [34, 35] applied the Spin model checker to establish the correctness of a family of population protocols. Only small
networks (i.e., with few nodes) were verified under weak fairness in Spin because of the problem discussed above. Verification under global fairness is infeasible in Spin. This situation calls for efficient model checking algorithms to deal with large LTL formulas. The work reported in [22] is closely related, but it still cannot be applied to population protocols. Therefore, it is important to have an alternative approach to handling stronger fairness constraints.

In our previous work [40], we developed a unified approach to model checking concurrent systems with a variety of fairness constraints. In particular, it shows the problem of model checking with global fairness can be reduced to the problem of searching for a terminal SCC which fails the given property. It was later applied to recently proposed population protocols [31] and previously unknown bugs are detected successfully. Nonetheless, it is limited by the state space explosion problem, like any model checking algorithm. Previous work has identified and solved the problem combining weak/strong fairness with state space reduction techniques like symmetry reduction [18] and partial order reduction [8]. In this work, we examine a combination of model checking with global fairness with symmetry reduction. Symmetry reduction is a natural choice to population protocols, or network protocols, which in general often contain many behaviorally similar or identical network nodes. Symmetry reduction has been investigated by many researchers for many years [11, 17, 6]. In [18, 21], it has been shown that combining weak/strong fairness with symmetry reduction is non-trivial. In this chapter, we prove that different from weak/strong fairness, symmetry reduction and global fairness can be integrated without extra effort. Adding symmetry reduction slightly changes the algorithm for model checking with global fairness. We present the combined reduction algorithm based on Tarjan’s strongly connected component algorithm [43]. We extend our home-grown PAT model checker with symmetry reduction and show its scalability by verifying recently proposed population protocols.

After the introduction, we first give a brief introduction to the population protocol model in Section 2 and the model checker PAT in Section 3, respectively. We define the semantic model and fairness notions used in PAT and describe how global fairness is handled in PAT in Section 4. We show our main result on how to combine symmetry reduction and global fairness in Section 5. Examples of population protocols and their model checking results are presented in Section 6. We discuss related works in Section 7 and conclude the chapter in Section 8.

2 The Population Protocol Model

As first proposed by Angluin et al. [2], a population protocol is a computing model for a distributed system of a large population of nodes under coarse control. Specifically, it describes a distributed network consisting of a finite but unbounded number of asynchronous agents. Every agent is (a) identical (one unified algorithm is designed for each agent irrespective of the network size); (b) simple with limited energy and memory (it is a finite-state machine and only performs simple computa-
tion); (c) passively mobile (it has no control over its own movement). The communication of agents (i.e., interaction) is pairwise, but the actual mechanism is abstracted away. Instead, an interaction graph is defined to describe which pairs of agents can possibly communicate with each other. The interaction is asymmetric, one as the initiator and the other as the responder. When two agents interact, they update their own states simultaneously according to a joint transition table. Usually global fairness condition is imposed on the running environment or the scheduler that each possible interaction occurs at an infinite number of times so that the whole system eventually makes progress. Rather than the termination of agents, a main concern is their convergence, i.e., the agents’ output eventually agrees to a common correct value (or configuration) after some time.

A population protocol model is formalized as below: First, the underlying network of the model can be described as a directed graph $G = (V, E)$ without multi-edges and self-loops. Each vertex represents a simple finite-state sensing device, and each edge $(u, v)$ means that $u$ as an initiator could possibly interact with $v$ as a responder. Second, a protocol is specified as a tuple $P(Q, \mathcal{C}, X, Y, O, \delta)$, which contains

- a finite set $Q$ of states,
- a set $\mathcal{C}$ of configurations,
- a finite set $X$ of input symbols,
- a finite set $Y$ of output symbols,
- an output function $O : Q \rightarrow Y$, and
- a transition function $\delta : (Q \times X) \times (Q \times X) \rightarrow 2^{Q \times Q}$.

If $(p', q') \in \delta((p, x), (q, y))$, then we write $((p, x), (q, y)) \rightarrow (p', q')$ and call it a transition. When $\delta$ always maps to a set that only contains a single pair of states, then we call the protocol deterministic.

A configuration $C$ is a mapping $C : V \rightarrow Q$ assigning to each node its internal state, and an input assignment $\alpha : V \rightarrow X$ specifies the input for each node. Let $C$ and $C'$ be configurations, $\alpha$ be an input assignment, and $u, v$ be different nodes. If there is a pair $(C'(u), C'(v)) \in \delta((C(u), \alpha(u)), (C(v), \alpha(v)))$, we say that $C$ goes to $C'$ via edge $e = (u, v)$ by transition $((C(u), \alpha(u)), (C(v), \alpha(v))) \rightarrow (C'(u), C'(v))$, abbreviated to $(C, \alpha) \xrightarrow{e} C'$. A pair of a transition $r$ and an edge $e$ constitutes an action $\sigma = (r, e)$. If $C$ goes to $C'$ via some edge, then $C$ can go to $C'$ in one step, written as $(C, \alpha) \rightarrow C'$. An execution is an infinite sequence of configurations and assignments $(C_0, \alpha_0), (C_1, \alpha_1), \ldots, (C_i, \alpha_i), \ldots$, such that $C_0 \in \mathcal{C}$ and for each $i$, $(C_i, \alpha_i) \rightarrow C_{i+1}$.

In the following, we summarize the fairness conditions for population protocols.

**Definition 1 (Global fairness).** For every $C, \alpha,$ and $C'$ such that $(C, \alpha) \rightarrow C'$, if $(C_i, \alpha_i) = (C, \alpha)$ for infinitely many $i$, then $(C_i, \alpha_i) = (C, \alpha)$ and $C_{i+1} = C'$ for infinitely many $i$, i.e., step $(C, \alpha) \rightarrow C'$ is taken infinitely many times in $E$.

**Definition 2 (Local fairness).** For every action $\sigma$, if $\sigma$ is enabled in $(C_i, \alpha_i)$ for infinitely many $i$, then $(C_i, \alpha_i) \xrightarrow{\sigma} C_{i+1}$ for infinitely many $i$. Hence, the action $\sigma$ is taken infinitely many times in $E$. 
It should be noticed that global fairness is strictly stronger than local fairness [19]. In population protocol model, steps specify how the whole protocol transforms from one configuration to another configuration, and actions specify the interactions between two nodes and only depend on the local states of the two interacting nodes. Global fairness requires that each step that can be taken infinitely often is actually taken infinitely often, while local fairness asserts that each action which is enabled infinitely often is actually taken infinitely often. Since one action can be enabled in different configurations, global fairness insists that an action must be taken infinitely often in all such configurations, whereas local fairness only requires that it occurs infinitely often in one of such configurations. Most of population protocols [2, 4, 19, 25, 3] will only work if global fairness is assumed. For instance, Fischer and Jiang [19] have proved that without global fairness uniform self-stabilizing leader election in rings is impossible.

3 A Brief Introduction to PAT

All our work mentioned in this chapter has been implemented in the model checker Process Analysis Toolkit (PAT)\(^1\) [39]. In the following we give a brief introduction to PAT.

PAT is a self-contained comprehensive verification framework for concurrent, real-time, probabilistic computing systems. It is composed of modeling, simulation and model checking tools integrated in one. For modeling, it comes with several expressive high-level specification languages. The main ones are from the Communicating Sequential Processes (CSP) [23] language family extended with data objects and data manipulation methods, such as CSP\# [39], timed CSP [41], probabilistic CSP [42]. For simulation, it provides several different simulation mechanisms to visualize system behaviors, such as a random or user-guided simulation. For verification, it supports various logic-based and behavior-based properties that are allowed to be checked by either explicit or symbolic model checking techniques, such as linear temporal logic (LTL) with various fairness assumptions, reachability, deadlock, refinement and divergence properties possibly with probabilities.

PAT is also a highly extensible and modularized framework for the technical and practical convenience of designing purpose specific model checkers. It decomposes modeling, abstraction techniques (if necessary), semantic representations of a state space and verification algorithms into four loosely coupled layers, so that the most advanced relevant techniques can be integrated into PAT with least effort. Moreover, it provides a common backplane that encapsulates a number of model checking algorithms and a wizard guide to interact with users for customizing the syntax and semantics of their own specification languages, verification algorithms, reduction and abstraction techniques and even graphical user interfaces.

\(^1\) http://www.patroot.com
In the following, we introduce the relevant subset of the syntax of CSP#, which is an event-based process algebra language for concurrent systems. CSP# is an extension of the classic CSP, that is, it adds shared data structures and operations on them, and low-level programming constructs such as assignments, if-then-else and while-loops.

In general, a CSP# model consists of four parts: constant definitions, variable declarations, process definitions and property specifications. A CSP# process definition, explained as follows, gives a name to a process expression that can be referenced in its or other process definitions.

**Definition 3 (Process).** A process P is defined using the following grammar:

\[
P ::= \text{Stop} | \text{Skip} | e\{\text{program}\} \rightarrow P | P\setminus X | P_1;P_2 | P_1\parallel P_2 | \text{if}(b)\{P_1\} \text{else } \{P_2\} | \text{if a}(b)\{P_1\} \text{else } \{P_2\} | P_1\parallel P_2 | \text{atomic}\{P\} | P_1\parallel P_2
\]

where \(P, P_1\) and \(P_2\) are processes, \(e\) is a name representing an event with an optional sequential program, \(X\) is a set of events, and \(b\) is a Boolean expression.

\(\text{Stop}\) is the process that communicates nothing, also called deadlock. \(\text{Skip} = \checkmark \rightarrow \text{Stop}\), where \(\checkmark\) is the termination event. Event prefixing \(e \rightarrow P\) performs event \(e\) and afterwards behaves as process \(P\). If \(e\) is attached with a sequential program, then the program is executed atomically together with the occurrence of the event. This sequential program can be a series of statements of modifying shared variables, or method calls of imported C# library classes, etc. Process \(P\setminus X\) hides all occurrences of the events in \(X\). An event is invisible if it is explicitly hidden by the hiding operator \(P\setminus X\). A user can also explicitly specify an invisible event by naming it \(\tau\). Sequential composition, \(P_1;P_2\), behaves as \(P_1\) until its termination and then behaves as \(P_2\). General choice \(P_1\parallel P_2\) behaves as either \(P_1\) or \(P_2\). Conditional choice \(\text{if}(b)\{P_1\} \text{else } \{P_2\}\) behaves as \(P_1\) if the Boolean expression \(b\) evaluates to true, and behaves as \(P_2\) otherwise. Atomic conditional choice \(\text{if a}(b)\{P_1\} \text{else } \{P_2\}\) performs the condition checking and the first operation of either \(P_1\) or \(P_2\) in one atomic step. Indexed interleaving \(P_1||P_2\) runs all processes independently except for communication through shared variables. The generalized form of interleaving is written as \(||x: \{0..n\}@P(x)\). An event may be in a compound form composed of variables and method calls. A process may be recursively defined, and may have parameters. The formal syntax and semantics of our language is presented in [38].

Regarding this algorithm, the most noticeable extension to CSP is that PAT provides a convenient and efficient mechanism to support user defined data type. It is difficult and inefficient to write complicate functions or advanced data structures using CSP’s syntax. To make this easier, PAT allows user to define functions and data types in C# programming language and use them in the models. These C# classes are built as DLL and loaded when models import them. Once they are defined, you can use them directly in any models.
4 Global Fairness in PAT

Nondeterminism is one fundamental feature of almost all models of distributed concurrent systems. It is due on one hand to the asynchronous executions of processes in various locations (process speed difference), and on the other hand to the asynchronous nature of communication channels (message delivery speed, delay or loss contention). Controlling nondeterminism is an essential task in the implementation of these systems. Fairness is a high-level construct that abstracts away the specific way in which nondeterminism is resolved. Informally, a fairness assumption is an abstract description of a class of schedulers [1].

In the area of system verification and model checking, liveness means “something good must eventually happen”. A counterexample to a liveness property is typically a loop (or a deadlock state which can be viewed as a trivial loop) during which good things never occur. Fairness is often necessary and important to prove liveness properties. Because fairness notions all restrict that a number of alternative events occur infinitely often in each infinite behavior of the system considered under certain conditions. Without fairness, verification of liveness properties often produces unrealistic infinite system executions during which one process or event is unfairly favored. It is crucial to systematically rule out those unfair counterexamples so as to identify real bugs.

Fairness in concurrent systems has been studied for decades. There are a variety of fairness notions proposed and discussed, e.g., unconditional fairness, weak fairness, strong fairness, hyperfairness and global fairness. In this section, we briefly review weak and strong fairness and then focus on the discussion on global fairness. In general, weak or strong fairness can work on either events or processes. For simplicity, we focus on event-level weak and strong fairness.

4.1 Labeled Kripke Structures

We present our work in the setting of Labeled Kripke structures (LKS) [9].

**Definition 4 (LKS).** An LKS is a 6-tuple $\mathcal{L} = (S, init, \Sigma, \rightarrow, AP, L)$ where: $S$ is a finite set of states; $init \in S$ is the initial state; $\Sigma$ is a finite set of events; $AP$ is a finite set of atomic state propositions; $\rightarrow: S \times \Sigma \times S$ is a transition-labeling relation with events; $L: S \rightarrow 2^{AP}$ is a state-labeling relation with atomic propositions.

For simplicity, we write $s \xrightarrow{e} s'$ to denote that $(s, e, s')$ is a transition in $\rightarrow$; $s \xrightarrow{e} s'$ to denote there exists some $e$ in $\Sigma$ such that $s \xrightarrow{e} s'$. Figure 4.1 shows an LKS, where transitions are labeled with event names and states are denoted by numbers, and 0 is the initial state. The dash-lined circles will be explained later.

We say that $\mathcal{L}$ is finite if and only if $S$ is finite. A run of $\mathcal{L}$ is a finite or infinite sequence of alternating states and events $\langle s_0, e_0, s_1, e_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all $i$. Because fairness affects infinite not finite system behaviors, we
focus on infinite system runs in this chapter. A state \( s \) is reachable if and only if there exists a finite run that reaches \( s \). Throughout the chapter, we assume that LKSs are always reduced, i.e., all states are reachable.

We assume properties are stated in the form of state/event linear temporal logic (SE-LTL) formulae \([9]\). Given an LKS \( \mathcal{L} = (S, \text{init}, \Sigma, \rightarrow, AP, L) \), an SE-LTL formula \( \phi \) can be constituted by not only atomic state propositions but also events.

\[
\phi ::= p | a | \neg \phi | \phi \land \phi | X\phi | F\phi | G\phi | \phi U \phi, \text{ where } p \in AP \text{ and } a \in \Sigma.
\]

**Definition 5.** Let \( \pi = \langle s_0, e_0, s_1, e_1, \cdots \rangle \) be a run in \( \mathcal{L} \) and \( \pi_i \) the suffix of \( \pi \) starting at \( s_i \). The path satisfaction relation is defined as follows:

\begin{itemize}
  \item \( \pi \models p \) iff \( s \) is the first state of \( \pi \) and \( p \in L(s) \).
  \item \( \pi \models a \) iff \( a \) is the first event of \( \pi \).
  \item \( \pi \models \neg \phi \) iff \( \pi \models \phi \).
  \item \( \pi \models \phi_1 \land \phi_2 \) iff \( \pi \models \phi_1 \) and \( \pi \models \phi_2 \).
  \item \( \pi \models X\phi \) iff \( \pi_1 \models \phi \).
  \item \( \pi \models F\phi \) iff there exists a \( k \geq 0 \) such that \( \pi_k \models \phi \).
  \item \( \pi \models G\phi \) iff for all \( i \geq 0 \) such that \( \pi_i \models \phi \).
  \item \( \pi \models \phi_1 U \phi_2 \) iff there exists a \( k \geq 0 \) s.t. \( \pi_k \models \phi_2 \) and for all \( 0 \leq j < k \), \( \pi_j \models \phi_1 \).
\end{itemize}

An example is \( G(d \implies F(x > 1)) \) where \( d \) is an event and \( x > 1 \) is an atomic proposition. The formula states that event \( d \) is always followed by a run such that \( x > 1 \) is eventually satisfied.

### 4.2 Fairness and Global Fairness

Event-level weak fairness \([28]\) states that if an event becomes continuously enabled after some steps, then it must be engaged infinitely often. An equivalent formulation is that every path should contain infinitely many positions at which the event is disabled or has occurred. Given the LKS presented in Figure 4.1, the path \( \langle 0, c, 1, g \rangle^\omega \) where the superscript \( \omega \) indicates an infinite number of repetitions does not satisfy

![Labeled Kripke structure](image-url)
event-level weak fairness because event \( d \) is always enabled (i.e., at both state 0 and 1) but never occurs during the path. The path which loops through state 3, 4 and 5 satisfies weak fairness as no event is enabled forever. Event-level strong fairness states that if an event is \textit{infinitely often} enabled, it must infinitely often occur. This type of fairness is particularly useful in the analysis of systems that use semaphores, synchronous communication, and other special coordination primitives. It has been identified by different researchers [29, 19, 37]. Given the LKS presented in Figure 4.1, the path which loops through state 3, 4 and 5 does not satisfy strong fairness because event \( g \) is infinitely often enabled but never occurs. It can be shown that strong fairness implies weak fairness.

**Definition 6 (Global fairness in LKSs).** Let \( E = (s_0, e_0, s_1, e_1, \ldots) \) be a path of an LKS \( \mathcal{L} \). \( E \) satisfies global fairness if and only if, for every \( s, e, s' \) such that \( s \xrightarrow{e} s' \), if 
\[ s = s_i \text{ for infinitely many } i, \text{ then } s_i = s \text{ and } e_i = e \text{ and } s_{i+1} = s' \text{ for infinitely many } i. \]

Global fairness\(^2\) was proposed by Fischer and Jiang in [19]. It is in fact a restricted form of extreme fairness proposed by Pnueli [36]. Global fairness states that if a \textit{step}\(^3\) (from \( s \) to \( s' \) by engaging event \( e \)) can be taken infinitely often, then it must actually be taken infinitely often. Many population protocols rely on global fairness [2, 19]. Compared to event-level strong fairness, global fairness requires that an infinitely often 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. Thus, global fairness is stronger than strong fairness. Their difference is illustrated in the following figure.

Under event-level strong fairness, state 2 in (a) may never be visited because all events occur infinitely often if the left loop is taken infinitely. With global fairness, all states in (a) must be visited infinitely often. Their difference when there is non-determinism is illustrated in (b). Both transitions labeled \( a \) must be taken infinitely with global fairness, which is not necessary with event-level strong or weak fairness. 

\textit{It can be shown that 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. In contrast to nontrivial combination of strong fairness and symmetry reduction [18], we show that model checking with global fairness can be straightforwardly combined with symmetry reduction.

\(^2\) In [19], it is called strong global fairness and defined for unlabeled transition systems. We slightly changed it so as to suit the setting of LKS.

\(^3\) Step and transition are used interchangeably in this chapter.
4.3 Model Checking with Fairness

Given an LKS $\mathcal{L}$ and a liveness property $\phi$, model checking is to search for a path of $\mathcal{L}$ which fails $\phi$. In automata-based model checking, the negation of $\phi$ is translated to an equivalent Büchi automaton $B$. Model checking with fairness is to search for a system path which is accepting by $B$ whilst satisfying the fairness constraint. In the following, we write $\mathcal{L} \models \phi$ to mean that $\mathcal{L}$ satisfies the property (without fairness assumption) and write $\mathcal{L} \models_{gf} \phi$ to mean that $\mathcal{L}$ satisfies the property with global fairness, i.e., every path of $\mathcal{L}$ which satisfies global fairness also satisfies $\phi$. We define a loop in the product of $\mathcal{L}$ and $B$ is a sequence of alternating states/events:

$$(s_0, b_0), e_0, (s_1, b_1), e_1, \cdots, (s_{n-1}, b_{n-1}), e_n, (s_n, b_n)$$

such that for all $0 \leq i \leq n$, $s_i$ is a state of $\mathcal{L}$, $b_i$ is a state of $B$, $(s_0, b_0)$ is reachable, $s_n = s_0$ and $b_n = b_0$. A loop is accepting if and only if there exists at least one accepting state of $B$ in $(b_0, b_1, \cdots, b_n)$. Furthermore, we define the following sets for a loop $l$ whose projection on $\mathcal{L}$ is $l_\mathcal{L} = (s_0, e_0, \cdots, s_{n-1}, e_{n-1}, s_0)$.

$\text{onceStep}(l) = \bigcup_{k=0}^{n-1} \text{enabled}(s_k)$
$\text{engagedStep}(l) = \bigcup_{k=0}^{n-1} \text{engaged}(s_k, l)$
$\text{enabled}(s) = \{(s, e, s') | s \xrightarrow{e} s'\}$
$\text{engaged}(s_k, l) = \{(s_k, e_k, s_{k+1}) | \text{\langle s_k, e_k, s_{k+1} \rangle is a subsequence of } l_\mathcal{L}\}$

Intuitively, $\text{onceStep}(l)$ is the set of steps which are enabled at least once during the loop, and $\text{engagedStep}(l)$ is the set of steps which are engaged during the loop. By definition, the proposition follows immediately.

**Proposition 1.** Let $E = m(l_\mathcal{L})$ be a path in $\mathcal{L}$ where $m$ is a finite path. $E$ satisfies global fairness if and only if $\text{onceStep}(l) = \text{engagedStep}(l)$. □

4.4 Algorithm for Model Checking with Global Fairness

Model checking with fairness can often be reduced to search for strongly connected components (SCC). In graph theory, an SCC is defined as a maximum subgraph such that every pair of vertices in the subgraph is connected by a path in the subgraph. A terminal SCC is an SCC such that all of its edges lead to vertices contained in the SCC. Naturally, an LKS can be viewed as a directed graph and therefore the concept of SCC can be extended to LKS. For instance, the LKS presented in Figure 4.1 contains four SCCs, indicated by dash-lined circles. Among the four, the one containing state 2 is terminal, whereas the one containing state 0 and 1 is not. For simplicity, we refer to a set of states of an LKS as an SCC if the subgraph containing the states and the transitions among them forms an SCC. We write that an SCC fails a liveness property $\phi$ as equivalent to that a path which reaches any state in the SCC
and infinitely often traverses through all states and transitions of the SCC fails $\phi$. For instance, the SCC containing state 2 fails the property $G(d \implies F(x > 1))$.

In our previous work [40], the problem of model checking with global fairness can be reduced to the problem of searching for a terminal SCC which fails the given property. Formally, it can be stated as the following theorem.

**Theorem 1.** Let $L$ be an LKS; $\phi$ be a property. $L \models_{gf} \phi$ if and only if there does not exist a terminal SCC $S$ in $L$ such that $S$ fails $\phi$.

Theorem 1 implies that we can use a simple procedure to find a counterexample by enumerating all terminal SCCs and then testing each one of them. The approach implemented in the PAT model checker is based on Tarjan’s algorithm for on-the-fly identification of SCCs. Its complexity is linear in the number of edges in the graph.

Given the LKS presented in Figure 4.1 with the property $G(d \implies F(x > 1))$, the SCC containing state 2 is identified as a counterexample with global fairness. Note that the SCC containing state 3, 4, and 5 is a counterexample only with no fairness or weak fairness. It is not a counterexample with global fairness because it does not satisfy global fairness, i.e., the step from state 5 to 6 by performing $g$ is enabled infinitely often but never occurs. The details of the algorithm can be found in [40].

### 5 Combining Symmetry Reduction with Global Fairness

We investigate the problem of model checking with global fairness and symmetry reduction. Symmetry reduction is a natural choice to population protocols, or network protocols, which in general often contain many behaviorally similar or identical network nodes. Symmetry reduction has been investigated by many researchers for many years [11, 17, 7]. In [18, 21], it has been shown that combining weak/strong fairness with symmetry reduction is non-trivial. In the section, we prove a surprising result that different from weak/strong fairness, symmetry reduction and global fairness can be integrated without extra effort. Adding symmetry reduction slightly changes the algorithm for model checking with global fairness. We present the combined reduction algorithm based on Tarjan’s strongly connected component algorithm [43].

#### 5.1 Model Checking with Symmetry Reduction

Any reduced state space can be regarded as an abstraction of the original one. It is useful only if there exists some kind of behavioral equivalence relation with the original state space, which guarantees that the reduced one is property-preserving with the original one. Thus it is pivotal to establish a certain behavioral equivalence for designing a particular state space reduction method. As for symmetry reduction,
the equivalence relation between the two state spaces can be specified by means of the notion of bisimulation given in the following.

Definition 7. Let $\mathcal{L}_i = (S_i, init_i, \Sigma_i, \rightarrow_i, AP_i, L_i)$, $i = 1, 2$, be two LKSs. A binary relation $R \in S_1 \times S_2$ between states of $(\mathcal{L}_1, \mathcal{L}_2)$ is a bisimulation if and only if whenever $(s, t) \in R$ and $\alpha \in \Sigma_1 \cup \Sigma_2$,

- if $(s, t) \in R$, then $L_1(s) = L_2(t)$.
- if $s \xrightarrow{\alpha_1} s'$ then $t \xrightarrow{\alpha_2} t'$ for some $t'$ such that $(s', t') \in R$ and
- if $t \xrightarrow{\alpha_2} t'$ then $s \xrightarrow{\alpha_1} s'$ for some $s'$ such that $(s', t') \in R$.

If there exists a bisimulation $R$ for $(S_1, S_2)$, then $\mathcal{L}_1$ and $\mathcal{L}_2$ are bisimulation equivalent, denoted $\mathcal{L}_1 \sim \mathcal{L}_2$.

Distributed/concurrent systems often contain a number of replicated components, which often leads to considerable symmetries in their corresponding state spaces.

Example 1. In [19], a self-stabilizing leader election protocol is proposed for complete networks. The system contains a lot of network nodes which interact with each other via following a number of simple rules. The system is modeled in the following form in CSP#.

$$
\text{System} = \text{Controller} || \text{Node}(0) || \text{Node}(1) || \cdots || \text{Node}(N-1)
$$

where Controller is a controlling process distinguished from the network nodes; Node($i$) models a network node with a unique identity $i$; $||$ denotes parallel composition. A node is marked as either a leader or not. Two nodes can interact according to the rules and start/quit being a leader. For instance, one of the rules states that if two interacting nodes are both leaders, then one of the nodes quits being a leader. One essential property of the protocol is that all nodes must eventually converge to the correct configuration. That is, eventually always there is one and only one leader in the network, i.e., $FG$ one leader.

In this example all network nodes (i.e., process Node($i$)) are indistinguishable and therefore they are all symmetric. Suppose $\sigma$ is a permutation on the set $\{0, 1, \cdots, N-1\}$ and a state of this protocol is written in the form $(s, s_0, \cdots, s_{N-1})$ where $s$ is the local state of Controller and $s_i$ is the local state of network node $i$. In terms of the convergence property, any pair of states $(s, s_0, \cdots, s_{N-1})$ and $(s, s_{\sigma(0)}, \cdots, s_{\sigma(N-1)})$ are equivalent, that is, one satisfies the property if and only if the other does. The symmetric permutation group on $\{0, 1, \cdots, N\}$ has $N!$ elements, so $\frac{100 \times (N-1)!}{N!}$ percent of the states are redundant equivalent ones at most. Symmetry reduction aims at subtracting these states from the exploration and often results in a significant saving in both time and space.

A permutation $\sigma$ is said to be an automorphism of an LKS $\mathcal{L}$ if and only if it preserves the transition relation and initial state. Formally, $\sigma$ satisfies the following condition.

$$(\forall s_1, s_2 \in S . \; \alpha \in \Sigma . \; s_1 \xrightarrow{\alpha} s_2 \Rightarrow \sigma(s_1) \xrightarrow{\sigma(\alpha)} \sigma(s_2)) \land \sigma(init) = init$$
A group $G$ is an automorphism group of $\mathcal{L}$ if and only if every $\sigma \in G$ is an automorphism of $\mathcal{L}$. A permutation $\sigma$ is said to be an invariance of an SE-LTL formula $\phi$ if and only if $\sigma(\phi) \equiv \phi$ where $\equiv$ denotes logical equivalence under all propositional interpretations [17]. For instance, given any permutation of process identities in the leader election example, the truth value of proposition one leader remains the same and therefore the permutation is an invariance of $FG$ one leader. A permutation $\sigma$ is said to be an invariance of $\mathcal{L}$ and property $\phi$ if and only if it is an automorphism of $\mathcal{L}$ and it is an invariance of $\phi$. $G$ is an invariance group of $\mathcal{L}$ and $\phi$ if and only if every $\sigma \in G$ is an invariance of $\mathcal{L}$ and $\phi$.

Given a state $s \in \mathcal{S}$ and the automorphism group $G$, the orbit of $s$ is the set $\theta(s) = \{ t \mid \exists \sigma \in G. \sigma(s) = t \}$, i.e., the set that contains all states equivalent to $s$. From each orbit of state $s$, a unique representative state $rep(s)$ can be picked such that for all $s$ and $s'$ in the same orbit, $rep(s) = rep(s')$. Intuitively, if $\sigma$ is an invariance of $\phi$, states of the same orbit are behaviorally indistinguishable with respect to $\phi$. For instance, the states of the 0-node being the only leader and the 1-node being the only leader in the leader election protocol are indistinguishable to the property $FG$ one leader. Based on this observation, an LKS can be turned into a quotient LKS where states in the same orbit are grouped together. Formally, a quotient LKS is defined as follows.

**Definition 8.** Let $\mathcal{L} = (S, init, \Sigma, \rightarrow, AP, L)$ be an LKS; $G$ be an automorphism group. The quotient LKS $\mathcal{L}_G = (S_G, init_G, \Sigma, \rightarrow_G, AP, L)$ is defined as follows:

- $S_G = \{ rep(s) \mid s \in S \}$ is the set of representative states of orbits.
- $init_G = \{ rep(init) \}$ is the initial representative state.
- $(r, e, r') \in \rightarrow_G$ iff there exists $r'' \in S$ such that $r \xrightarrow{e} r''$ and $rep(r'') = r'$.

It has been proved [12] that if $G$ is an invariance group of $\mathcal{L}$ and $\phi$, then $\mathcal{L}$ satisfies $\phi$ if and only if $\mathcal{L}_G$ satisfies $\phi$. Formally, it is stated as the following theorem. It is proved by showing that the relation $(s, \theta(s))$ is a bisimulation relation between $\mathcal{L}$ and $\mathcal{L}_G$.

**Theorem 2.** Let $\mathcal{L} = (S, init, \Sigma, \rightarrow, AP, L)$ be an LKS; $\phi$ be an SE-LTL formula. If $G$ be an invariance group of $\mathcal{L}$ and $\phi$, then $\mathcal{L} \models \phi$ if and only if $\mathcal{L}_G \models \phi$.

### 5.2 Symmetry Reduction with Global Fairness

In the following, we prove that global fairness is orthogonal with symmetry reduction by showing that there is a path which satisfies global fairness and fails $\phi$ in $\mathcal{L}$ if and only if there is a path which satisfies global fairness and fails $\phi$ in $\mathcal{L}_G$. For convenience, we fix that $\phi$ is an $SE$-LTL formula to be checked, $\mathcal{B}$ is the Büchi automaton constructed by the negation of $\phi$, $\mathcal{L}$ is LKS of the original system, $G$ is invariance group of $\mathcal{L}$ and $\phi$ and $\mathcal{L}_G$ is the LKS of the abstract system after applying symmetry reduction.

We first assume any event in $\Sigma$ is not allowed to be permuted.
Lemma 1. There exists a path \( p = (s_0, a_0, s_1, a_1, \cdots) \) in \( L \) if and only if there exists a path \( q = (r_0, a_0, r_1, a_1, \cdots) \) in \( L_G \) such that \( r_i = \text{rep}(s_i) \) for all \( i \).

Proof It follows from the proof of Lemma 3.1 in [17]. \( \square \)

Theorem 3. There exists an accepting loop in the product of \( L \) and \( B \) which satisfies global fairness if and only if there also exists an accepting loop in the product of \( L_G \) and \( B \) which satisfies global fairness.

Proof: (Sufficient condition) We first prove the sufficient condition. The proof is divided into two parts. In the first part, we prove (1) if there exists an accepting loop \( l' \) in the product of \( L_G \) and \( B \), then there exists an accepting loop \( l \) in the product of \( L \) and \( B \). Then we prove (2) if \( l' \) satisfies global fairness, so does \( l \).

Let \( l' = ((r_0, b_0), a_0, (r_1, b_1), a_1, \cdots, (r_{n-1}, b_{n-1}), a_{n-1}, (r_0, b_0)) \) be an accepting loop. Without loss of generality we assume that \( b_0 \) is an accepting state. Then there exists in the product of \( L_G \) and \( B \) a path arriving at \((r_0, b_0)\). By Lemma 1 there exists a corresponding path in the product of \( L \) and \( B \) to state \((s_0, b_0')\) where \( r_0 = \text{rep}(s_0) \). Because \( G \) is the invariance group of \( L \) and \( \phi \), \( b_0' = b_0 \) which is also an accepting state. By Lemma 1 again, for \( l' \) there exists in the product of \( L \) and \( B \) a path \( p_0 = ((s_0, b_0), a_0, (s_1, b_1), a_1, \cdots, (s_{n-1}, b_{n-1}), a_{n-1}, (s_0, b_0)) \) such that for all \( i \) in \( p_0 \) we have \( r_i = \text{rep}(s_i) \). Notice that \( p_0 \) is not necessarily a loop. Since \( r_0 = \text{rep}(s_0) \), we can unfold \( l' \) again according to Lemma 1, but this time beginning at \( s_0 \), which will produce the path \( p_1 = ((s_0, b_0), a_0, (s_1, b_1), a_1, \cdots, (s_{n-1}, b_{n-1}), a_{n-1}, (s_0, b_0)) \), and for all \( i \) in \( p_1 \) we still have \( r_i = \text{rep}(s_i) \). We can repeat this unfolding arbitrary many times which will give us a sequence of path \( p_0, p_1, p_2, \cdots \) with the corresponding end states \((s_0, b_0), (s_0, b_0'), (s_0, b_0'), \cdots \) which are all accepting. As the orbit of the states \( s_0, s_0', \cdots \) is finite, \( s_0' = s_i' \) for some \( i \) and \( j \). Obviously, the concatenation of the paths \( p' \) to \( p'^{i-1} \), say \( l \), is an accepting loop in the product of \( L \) and \( B \).

Because \( l' \) satisfies global fairness, \( \text{onceStep}(l') = \text{engagedStep}(l') \). We define a function \( \text{recover} \) such that given \( (s, e, s') \in \rightarrow_G \) and some permutation \( \sigma \in G \), \( \text{recover}((s, e, s'), \sigma) = (t, e', t') \) such that \( \sigma^{-1} = t \Rightarrow t' \). Intuitively, \( \text{recover} \) returns the corresponding transition of \( (s, e, s') \) in \( L \) with respect to a specific permutation \( \sigma \). For \( 0 \leq m \leq n \), \( r_m \) in loop \( l' \) corresponds to \( s'_m \) (i.e., \( r_m = s'_m \sigma_m \)) in each path \( p' (i \leq t < j) \). Then

- \( \text{enabled}(s'_m) = \text{recover}(\text{enabled}(r_m), \sigma_m') \);
- \( \text{engaged}(s'_m, p') = \text{recover}(\text{engaged}(r_m, l'), \sigma_m') \).

Thus, \( \text{onceStep}(l) = \{ \text{recover}(\text{enabled}(r_m), \sigma_m'), 0 \leq m < n, i \leq t < j \} \) and \( \text{engagedStep}(l) = \{ \text{recover}(\text{engaged}(r_m, p'), \sigma_m'), 0 \leq m < n, i \leq t < j \} \). Since \( \text{onceStep}(l') = \text{engagedStep}(l') \), \( \text{onceStep}(l') = \{ \text{enabled}(r_m), 0 \leq m \leq n, i \leq t < j \} \) and \( \text{engagedStep}(l') = \{ \text{engaged}(r_m, p'), 0 \leq m \leq n, i \leq t < j \} \), we have \( \text{onceStep}(l) = \text{engagedStep}(l) \).

(Necessary condition) Let \( l = ((s_0, b_0), a_0, (s_1, b_1), a_1, \cdots, (s_{n-1}, b_{n-1}), a_{n-1}, (s_0, b_0)) \) be an accepting loop in the product of \( L \) and \( B \). There exists a path arriving at \((s_0, b_0)\). Assume \( b_0 \) is an accepting state in \( B \). By Lemma 1 there exists a path in the product of \( L_G \) and \( B \) leading to state \((\text{rep}(s_0), b_0)\). By Lemma 1, there exists in the
product of $L_G$ and $\mathcal{B}$ a corresponding loop $l' = ((s_0, e_0, a_0), (s_1, e_1, a_1), \ldots, (s_n, e_n, a_n))$ such that $\sigma_i \in G$ and $rep(s_i) = s_i \sigma_i$ for all $0 \leq i < n$.

Because $l$ satisfies global fairness, $onestep(l) = engagedStep(l)$. We define a function $twist$ such that given $s \xrightarrow{\sigma} s'$, $twist(s, e, s') = rep(s) \xrightarrow{\sigma} G \cdot rep(s')$. Intuitively, $twist$ returns the corresponding transition in $L_\phi$ of $(s, e, s')$. For all $0 \leq i < n$, $s_i$ in loop $l$ corresponds to $rep(s_i)$ in $l'$. Then

- $enabled(rep(s_i)) = twist(enabled(s_i));$
- $engaged(rep(s_i), l') = twist(engaged(s_i, l)).$

Thus, $onestep(l') = \{twist(enabled(s_i)), 0 \leq i < n\}$ and $engagedStep(l') = \{twist(engaged(s_i, l')), 0 \leq i < n\}$. Because $onestep(l) = engagedStep(l)$, we have $onestep(l) = \{enabled(s_i), 0 \leq i < n\}$ and $engagedStep(l) = \{engaged(s_i, l), 0 \leq i < n\}$. Thus, we have $onestep(l') = engagedStep(l')$. □

Note that we did not allow the events to be permuted at the beginning of this subsection, which seems too restrictive. Now we relax the definition of permutation to permute states and events simultaneously. It is proved in [16] that the new definition is equivalent to the one given before. By a simple argument, it can be shown that Theorem 3 still holds.

Based on Theorem 3, we present a practical algorithm for searching the reduced state space for accepting globally fair loops, based on Tarjan’s SCC algorithm. Underlining shows the differences compared with the usual algorithm for model checking with global fairness. Assume that $G$ is a permutation group of process identities which is an invariance group of $L$ and $\phi$. Let $rep$ be a function which, given a state, returns a unique representative. Using function $rep$, we can tell whether two states are in the same orbit or not. Note that identifying an optimal representative function $rep$ can be non-trivial. We adopt the automata-theoretic approach and perform the following. Firstly, a Büchi automaton $\mathcal{B}$ is generated from the negation of $\phi$. Next, the synchronous product of $\mathcal{B}$ and $L$ is computed on-the-fly. Tarjan’s SCC algorithm is used to identify SCC in the product along the construction. Note that a state of the product is a pair $(s, b)$ where $s$ is a state of $L$ and $b$ is a state of $\mathcal{B}$. Assume that the initial state of the product is $(init_s, init_b)$ where $init_s$ is the initial state of $L$ and $init_b$ is the initial state of $\mathcal{B}$. For simplicity, we assume there is only one initial state in $\mathcal{B}$.4

The detailed algorithm is presented in Algorithm 1. It resembles the standard Tarjan’s SCC algorithm [43]. Note that we use the iterative version of Tarjan’s SCC algorithm in the practical implementation for performance reason. Three data structures are used to identify SCCs: $path$ is a stack containing states along a path from the initial state to the current one; $index$ and $lowlink$ are hash tables which assign two numbers to a state. A state is a root of an SCC if and only if the two numbers are equivalent. To apply symmetry reduction, instead of working with concrete states, Tarjan’s algorithm is applied to representatives of orbits. For instance, $path$ contains only $rep(v)$ (line 5) and $lowlink$ and $index$ map $rep(v)$ to numbers (line 2 and 3).

---

4 For simplicity, we assume there is only one initial state in $\mathcal{B}$.4
Algorithm 1 Tarjan’s algorithm with symmetry reduction

```plaintext
int counter := 0;
stack path := an empty stack;
hashable index := an empty hash table;
hashable lowlink := an empty hash table;
TarjanModelChecking((init, init));

1: procedure TarjanModelChecking(v)
2:   index[rep(v)] := counter;
3:   lowlink[rep(v)] := counter;
4:   counter := counter + 1;
5:   push rep(v) into path;
6:   for all v → v′ do
7:     if rep(v′) is not in index then
8:       TarjanModelChecking(v′);
9:       lowlink[rep(v)] = min(lowlink[rep(v)], lowlink[rep(v′)]);
10:      else if rep(v′) is in path then
11:         lowlink[rep(v)] = min(lowlink[rep(v)], index[rep(v′)]);
12:      end if
13:   end for
14:   if lowlink[rep(v)] = index[rep(v)] then
15:     set scc := an empty set;
16:     repeat
17:       pop an element v′ from path and add it into scc;
18:     until v′ = v
19:     if scc forms a terminal SCC in L and scc is accepting then
20:       generate a counterexample and return false;
21:     end if
22:   end if
23: end procedure
```

Whenever an SCC is identified (line 12), we check whether the SCC is terminal in L and accepting. If it is, then we prove the existence of at least one counterexample. We skip the details on generating a concrete counterexample. Note that an SCC is terminal in L if and only if, for every state (s, b) in the SCC, if s → s′, then there exists (s′, b′) in the SCC. An SCC is accepting if and only if it contains a state (s, b) such that b is an accepting state in B. The algorithm terminates when all states have been checked. The correctness of the algorithm follows from the theorems presented in previous sections. It is always terminating because the number of un-explored states are monotonically decreasing and the number of states are finite. Its complexity is linear in the edges of transitions in the product of L and B.

The correctness of the algorithm is established by the following theorem.

**Lemma 2.** In the product of L (resp. L_G) and B, there exists an accepting loop which satisfies global fairness if and only if there exists an accepting SCC which is also a terminal SCC in L (resp. L_G).

Proof: (Necessary Condition) Suppose l is an accepting loop which satisfies global fairness. so onceStep(l) = engagedStep(l). The states in l forms a strongly connected subgraph S in the product and S is a terminal SCC in L. Let S′ be the
SCC that contains the states in $S$. Suppose $l'$ be the loop which traverses all transitions in $S$. Because $S$ is a terminal SCC in $L$, $\text{onceStep}(l') = \text{engagedStep}(l') = \text{onceStep}(l)$. So $S'$ is also a terminal SCC in $L$. On the other hand, because $l$ is accepting, there is an accepting state in $S'$.

(Sufficient Condition) Suppose $S$ is an accepting SCC in the product of $L$ and $B$, and it is a terminal SCC in $L$. Let $l$ be the loop which traverses all transitions in $S$. We get $\text{onceStep}(l) = \text{engagedStep}(l)$. so $l$ is a globally fair loop. Since there is an accepting state in $l$, $l$ is an accepting loop which satisfies global fairness.

Using the same argument one can show the lemma holds for product of $L_G$ and $B$. □

**Theorem 4.** Let $\phi$ be an SE-LTL formula. If $G$ is an invariance group of $L$ and $\phi$, then $L \models_{gf} \phi$ if and only if $L_G \models_{gf} \phi$.

**Proof** By Theorem 1, $L \not\models_{gf} \phi$ if and only if there exists an accepting SCC in the product of $L$ and $B$ which is also a terminal SCC in $L$. Similarly, $L_G \not\models_{gf} \phi$ if and only if there exists an accepting SCC in the product of $L_G$ and $B$ which is also a terminal SCC in $L_G$. By Theorem 3 and Lemma 2, there exists an accepting SCC $S$ such that $S$ is a terminal SCC in $L$ if and only if there exists an accepting SCC $S'$ such that $S'$ is a terminal SCC in $L_G$, which proves the theorem. □

6 Verifying Population Protocols with PAT

In this section, we evaluate the effectiveness of our verification method. We extend the PAT model checker with our algorithms for model checking with global fairness and symmetry reduction. First, we introduce self-stabilizing population protocols for experiments and explain their corresponding PAT models. Second, we show the results of model checking these protocols without and with symmetry reduction.

6.1 Population protocols and their models

A distributed system or a population protocol is said to be self-stabilizing [15] if it satisfies the following two properties:

- **convergence**: starting from an arbitrary configuration, the system is guaranteed to reach a correct configuration;
- **closure**: once the system reaches a correct configuration, it cannot become incorrect any more.

This means that in our modeling of these protocols, we have to take all possible initial configurations into account, and the checked properties have the form of $FG$ property. We have selected protocols for two-hop coloring and orienting nodes.
and protocols for leader election and token passing. Note that all these protocols only work under global fairness.

In the population protocol model, one protocol consists of $N$ nodes, numbered from 0 to $N - 1$.\(^5\) A protocol is usually described by a set of interaction rules between an initiator $u$ and a responder $v$. Such rules have conditions on the state and the input of the initiator and the responder, and specify the state of the initiator and the responder if a transition can be taken.

### 6.1.1 Two-hop coloring

A protocol to make nodes to recognize their neighbors in a ring is presented in [3]. In fact, it is a general algorithm that enables each node in a degree-bounded graph to distinguish between its neighbors. The graph is colored such that any two nodes adjacent to the same node have different colors. More precisely, for each node $v$, if $u$ and $w$ are distinct neighbors of $v$, then $u$ and $w$ must have different colors. $(u, w)$ is called a two-hop pair. In the current paper, we restrict ourselves to rings, and three colors suffice the purpose (see [3]).

Each node $u$ in a ring has two state components, $\text{color}[u]$ encodes the color of node $u$ and $F[u]$ is a bit array, indexed by colors. Initially, $\text{color}[u]$ and $F[u]$ can have arbitrary values. The following description defines the interaction between an initiator $u$ and a responder $v$.

<table>
<thead>
<tr>
<th>Nondeterministic two-hop coloring protocol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $F[u][\text{color}[v]] \neq F[v][\text{color}[u]]$ then</td>
</tr>
<tr>
<td>$\text{color}[u] \leftarrow \text{color}'[u]$; $F[u][\text{color}[v]] = F[v][\text{color}[u]]$</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>$F[u][\text{color}[v]] = \neg F[u][\text{color}[v]]$; $F[v][\text{color}[u]] = \neg F[v][\text{color}[u]]$</td>
</tr>
<tr>
<td>endif</td>
</tr>
</tbody>
</table>

One edge (or interaction) $(u, v)$ is synchronized if $F[u][\text{color}[v]] = F[v][\text{color}[u]]$, then these two nodes do not change their color but flip their bits ($F[u][\text{color}[v]]$ and $F[v][\text{color}[u]]$). On the other hand, node $u$ is nondeterministically recolored, and it copies $F[v][\text{color}[u]]$ of node $v$ as its bit $F[u][\text{color}[v]]$. The statement $\text{color}[u] \leftarrow \text{color}'[u]$ means one of the three possible colors is nondeterministically assigned as the new color of $u$. The model of this protocol in PAT and its property to be checked are detailed in Figure 2.

Figure 2 presents (part of) its model in PAT to illustrate the modelling language. Line 1 defines two global constants ($N$ and $C$ of value 3) and global variables. $N$ models the network size, i.e., number of nodes and $C$ models the number of colors. Array $\text{color}$ models the color of each node. $F$ is a bit array for each node, indexed by colors. Next, line 2 to 10 defines how an initiator $u$ interacts with a responder $v$, which captures the essence of the protocol. Every time there is an interaction in the network, the initiator and responder must update themselves according to a

\(^5\) In the following discussion, we set $N$ as three for the sake of simplicity.
of the network. The property is $FG$ nodes’ interactions in the network. Which nodes can interact reflects the topology.

### Two-hop coloring

1. The predecessor of $v$ is the node that is two steps away from $v$.
2. For any two nodes $u$ and $v$, $u$ is the predecessor of $v$ if and only if $v$ is the successor of $u$.

### Orienting undirected rings

Given a ring colored by protocols in Section 6.1.1, it is possible to have a protocol that gives a sense of orientation to each node on an undirected ring [3]. After the orienting, (1) each node has exactly one predecessor and one successor, the predecessor and successor of a node are different; (2) for any two nodes $u$ and $v$ in the ring, $u$ is the predecessor of $v$ if and only if $v$ is the successor of $u$, for any edge $(u, v)$, either $u$ is the predecessor of $v$ or $v$ is the predecessor of $u$.

Each node $u$ in a ring has three state components: $color[u]$ encodes the color of node $u$, $precolor[u]$ the color of its predecessor, and $succolor[u]$ the color of its successor.

---

**Fig. 2** PAT model of the two-hop coloring protocol

---

```plaintext
#define N 3; #define C 3; var color[N]; var F[N][C];

Interaction(u,v) =
if (F[u][color[v]]! = F[v][color[u]]){
  act1.u.v(F[u][color[v]] = F[v][color[u]]; color[u] = 0;)
  Interaction(u,v)
}
else {
  act2.u.v(F[u][color[v]] = F[v][color[u]]; color[u] = 1;)
  Interaction(u,v)
  act3.u.v(F[u][color[v]] = F[v][color[u]]; color[u] = 2;)
}

Init() = ...

TowHopColoring() = Init(); (|| x : {0..N-1} @ Interaction(x, (x+1)%N))
  || (Interaction((x+1)%N,x));

#define twohopcoloring(color[0]! = color[2]&&color[1]! = color[2])

assert TowHopColoring() = FG twohopcoloring;
```

---

- A set of pre-defined rules. A rule is applicable only if the guarding condition (e.g., $F[u][color[v]]! = F[v][color[u]]$) is satisfied. An action (e.g., `act1.u.v`) may be attached to variables updating (e.g., $color[u] = 0$). Lines 12 and 13 model the two-hop coloring protocol as process `TowHopColoring`, which starts with process `Init` (which initializes the system in every possible configuration and is omitted here). After initialization, the system is the interleaving (modeled by the operator `||`) of nodes’ interactions in the network. Which nodes can interact reflects the topology of the network. The property is $FG$ `twohopcoloring` (defined as an assertion at line 16). `twohopcoloring` (defined at lines 14 and 15) is a proposition which states that the neighbors of a node in a ring have different colors (for rings of size three).

### 6.1.2 Orienting undirected rings

Given a ring colored by protocols in Section 6.1.1, it is possible to have a protocol that gives a sense of orientation to each node on an undirected ring [3]. After the orienting, (1) each node has exactly one predecessor and one successor, the predecessor and successor of a node are different; (2) for any two nodes $u$ and $v$, $u$ is the predecessor of $v$ if and only if $v$ is the successor of $u$, for any edge $(u, v)$, either $u$ is the predecessor of $v$ or $v$ is the predecessor of $u$.

Each node $u$ in a ring has three state components: $color[u]$ encodes the color of node $u$, $precolor[u]$ the color of its predecessor, and $succolor[u]$ the color of its successor.
successor. Initially, all nodes are two-hop colored (array color satisfies the two-hop coloring property), precolor[u] and succolor[u] can have arbitrary values. The following description defines the interaction between an initiator u and a responder v.

### Orienting an undirected ring protocol.

```
if color[v] = precolor[u] and color[v] ≠ succolor[u] then
  succolor[v] ← color[u]
elseif color[v] = succolor[u] and color[v] ≠ precolor[u] then
  precolor[v] ← color[u]
else
  precolor[u] ← color[v]; succolor[v] ← color[u]
endif
```

The PAT model of this protocol is shown in Figure 3. Lines 2-8 model how two nodes can interact. The initialization at line 9 makes sure that the nodes are initially two-hop colored. Lines 10 and 11 define a model of orienting an undirected ring, which takes two-hop coloring as inputs. The assertions that the protocol satisfies two properties are given at line 14 and 15. For example, property1 formalizes that the predecessor and successor of a node are different.

```c
#define N 3; #define C 3; var color[N]; var precolor[N]; var succolor[N];

Interaction(u, v) = if (color[v] == precolor[u] && color[v] != succolor[u]){
  act1.u.v{succolor[v] = mycolor[u];} -> Interaction(u, v)
} elseif (color[v] == succolor[u] && color[v] != precolor[u]){
  act2.u.v{precolor[v] = color[u];} -> Interaction(u, v)
} else {
  act3.u.v{precolor[u] = color[v]; succolor[v] = color[u];} -> Interaction(u, v)
}

Init() = ...

OrientingUndirected() = Init(); || x: {0..N-1}@ (Interaction(x, (x+1)\%N)

#define property1 (x: 0..N-1@ precolor[x] = succolor[x]);
#define property2 (...);

#assert OrientingUndirected() ⊨ FG property1;
#assert OrientingUndirected() ⊨ FG property2;
```

Fig. 3 PAT model of the orienting undirected ring protocol

### 6.1.3 Leader election

Leader election is a fundamental problem for distributed systems. A leader election protocol is used to choose a unique agent in the network as the leader. We choose
two population protocols of leader election, one in oriented odd rings and the other in complete graphs. In the following, we study a leader election protocol in oriented odd rings. The protocol in complete graphs [19] is quite similar and skipped for simplicity. The following description is partially taken from [25, 3]. 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 [25, 3].

**Leader election protocol for odd rings.**

```
if label[u] = label[v] then
  if probe[u] = 1 then leader[u] ← 1; probe[u] ← 0 endif
  bullet[v] ← 0
  if phase[u] = 0 then phase[u] ← 1; probe[v] ← 1
  elseif probe[v] = 0 then
    label[v] = ¬label[v]; phase[v] ← 0
  endif
else leader[v] = 1 then
  if bullet[v] = 1 then leader[v] ← 0
  else bullet[u] ← 1 endif
else
  if bullet[v] = 1 then bullet[v] ← 0; bullet[u] ← 1 endif
  if probe[u] = 1 then probe[u] ← 0; probe[v] ← 1 endif
endif
```

The PAT model of this protocol is shown in Figure 4. Lines 2-11 model how two nodes can interact. We have totally eleven (act1.u.v up to act11.u.v) cases separated 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 12, it captures any possible evaluations of the variables. Line 13 defines how nodes interact in an oriented ring. Line 14 defines a predicate that there is one leader in the network. Line 15 claims that the protocol eventually self-stabilizes to a unique leader existing in the network.
\#define N 3; \texttt{var} leader[N]; \texttt{var} label[N]; \texttt{var} probe[N]; \texttt{var} phase[N]; \texttt{var} bullet[N];

(1)

Interact(u, v) =

(2)

\[ [label[u] == label[v] \&\& probe[u] == 1 \&\& phase[u] == 0] \]

(3)

act1.u.v\{leader[u] = 1; probe[u] = 0; bullet[v] = 0; phase[u] = 1; probe[v] = 1; \} \rightarrow Interact(u, v)

(4)

\[ [phase[v] == 0;] \rightarrow Interact(u, v) \]

(5)

\[ [\ldots] \]

(6)

\[ [phase[v] == 0;] \rightarrow Interact(u, v) \]

(7)

\[ \ldots \]

(8)

Init() = ...

(9)

\texttt{LeaderElection()} = Init(); \{\{x : 0..N \rightarrow Interact(x, (x+1)\%N)\};

(10)


(11)

\#assert LeaderElection() \models FG leadeerection;

(12)

\texttt{Fig. 4} PAT model of the leader election protocol in odd rings

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. Then 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. Then 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 \( v = 0 \))


5. Then 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 \( bullet = [0, 0, 0], label = [0, 1, 0], leader = [1, 1, 0], phase = [0, 1, 1], probe = [1, 0, 0] \).\footnote{As the protocol is self-stabilizing, the counterexample can start directly from here. We keep the first part just to faithfully represent the infinite trace found by PAT.} 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 $\text{label}[2] = \text{label}[0]$, $\text{probe}[2] = 0$, $\text{phase}[2] = 1$ and $\text{probe}[0] = 1$, we have
   $\text{bullet}[0] \leftarrow 0$. ($u = 2$ and $v = 0$)
2. Then since $\text{label}[0]! = \text{label}[1]$, $\text{leader}[1] = 1$ and $\text{bullet}[1] = 0$, we have $\text{bullet}[0] \leftarrow 1$. ($u = 0$ and $v = 1$)
3. Then since $\text{label}[0]! = \text{label}[1]$, $\text{leader}[1] = 1$ and $\text{bullet}[1] = 0$, we have $\text{bullet}[0] \leftarrow 1$. ($u = 0$ and $v = 1$)
4. Then since $\text{label}[2] = \text{label}[0]$, $\text{probe}[2] = 0$, $\text{phase}[2] = 1$ and $\text{probe}[0] = 1$, we
   have $\text{bullet}[0] \leftarrow 0$. ($u = 2$ and $v = 0$)

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 [25], it is confirmed as a
valid counterexample which has escaped simulations of the protocol [26]. The rea-
son to the counterexample is the following [26]. 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 posi-
tion 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 de-
scription, this property is not preserved either. Modifications of the description have
been made in [3]. We also modeled the revised version of the protocol, and found no
counterexample. By this case study, we emphasize that without the newly developed
model checking algorithm [40] for efficient verification under (global) fairness, it is
impossible to find such an error in a pseudo code description of a population proto-
col, especially when a protocol tends to be intuitively more complicated.

6.1.4 Token circulation

The token circulation protocol in directed rings depicted below is proposed in [2, 3].
The desired behavior of this protocol can be described as follows: (1) there is only
one node who holds the token; (2) a node does not obtain again until every other
node has obtained a token once; (3) each node can have the token infinitely often.

| Token circulation protocol. |
| Rule 1. ($* b$, $N$), ($* b$, $L$) → (($b$), ($\bar{b}$)) |
| Rule 2. ($* b$, $*$), ($* b$, $N$) → (($b$), ($\bar{b}$)) |

It is assumed that every node passes the token to next one right after it has got it.
Furthermore, the protocol also requires the existence of a leader. Informally, there
is a static node with the leader mark $L$, and all other nodes have the non-leader
mark $N$ in every configuration. The state of each node is represented by a pair in
$\{-, +\} \times \{0, 1\}$. + means that the node is holding a token and − means the opposite.
#define N 3; var leader[N]; var label[N]; var token[N];

(1)

Rule 1 \((u, v) = (!leader[u] && leader[v] && label[u] = label[v])\) -> Rule 1(u, v);

(2)

Rule 2 \((u, v) = (!leader[v] && label[u] != label[v])\) -> Rule 2(u, v);

(3)

Init() = ...

(6)

TokenCirculation() = Init(); ((\(x : 0 .. N - 1\)) || (Rule1(x, (x + 1) \% N)) || (Rule2(x, (x + 1) \% N)));

(7)

#define onetoken(token[0] + token[1] + token[2] == 1);

(8)

#assert TokenCirculation() \(\equiv\) FG onetoken;

(9)

(10)

Fig. 5 PAT Model for the token circulation protocol

The second part of a state of a node is called the label. The * here denotes an always-matched symbol. On the left hand side, the symbol \(b\) matches either 0 or 1 and \(\bar{b}\) is its complement. It should be noticed that different occurrences of \(b\) in a same rule refer to the same value. The input for each node informs them who is leader, which is unique in the network. When two nodes interact, if the responder is the leader, it sets its label to the complement of the initiator’s label; otherwise the responder copies the label from the initiator. If an interaction triggers a label change, a token is passed from the initiator to the responder. If a token is not present at the initiator, a new token is generated.

The PAT model of this protocol is shown in Figure 5. We only give the assertion for the first property. The other two can be defined in a similar way. The states of the whole system are represented by three arrays of bits (\(leader[N]\), \(token[N]\) and \(label[N]\)).

6.2 Verification Results in PAT

The experiment data are presented in Table 1. The experiment testbed is a server with 2.813GHz Intel Xeon 64-bit CPU and 32 GB memory. In the table, ‘−’ means more than 3 hours. ‘States’ in the table means the number of states stored (not the number of states visited), and ‘Gain’ means the relative improvement on consumed time brought by symmetry reduction. We skip the statistics on memory consumption because the dynamic garbage collection facility in PAT makes the estimation inaccurate. Nonetheless, the number of states reflects the memory usage.

From the table, it is shown that symmetry reduction reduces both memory and time consumption as expected. As a result, PAT handles more network nodes with symmetry reduction for leader election. In most of the cases, the number of states without symmetry reduction is \(N\) times that with symmetry reduction, which is ex-
pected as the protocols are for network rings. Optimally, PAT with symmetry reduction is $N$ times better than without reduction time-wise. For the conducted experiments, the saving in terms of time is 75% of the optimal value in average. The computational overhead is mainly due to the orbit problem, i.e., to decide whether two states are in the same orbit. It is known that in general the orbit problem is as hard as the graph automorphism problem [12]. The current implementation is based on enumerating all states in an orbit explicitly. There are a number of optimizations which solve the problem efficiently in practice. It remains our future work to incorporate those techniques.

<table>
<thead>
<tr>
<th>Model</th>
<th>Network Size</th>
<th>Without Reduction</th>
<th>With Reduction</th>
<th>Gain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time (Sec)</td>
<td>States</td>
<td>Time (Sec)</td>
</tr>
<tr>
<td>two-hop coloring</td>
<td>3</td>
<td>122856</td>
<td>36.7</td>
<td>42182</td>
</tr>
<tr>
<td>orienting rings (prop 1)</td>
<td>3</td>
<td>19190</td>
<td>2.27</td>
<td>6398</td>
</tr>
<tr>
<td>orienting rings (prop 2)</td>
<td>3</td>
<td>19445</td>
<td>2.23</td>
<td>6503</td>
</tr>
<tr>
<td>orienting rings (prop 1)</td>
<td>4</td>
<td>1255754</td>
<td>267.2</td>
<td>313940</td>
</tr>
<tr>
<td>orienting rings (prop 2)</td>
<td>4</td>
<td>1206821</td>
<td>267.1</td>
<td>302071</td>
</tr>
<tr>
<td>orienting rings (prop 1)</td>
<td>5</td>
<td>11007542</td>
<td>9628.1</td>
<td>2201510</td>
</tr>
<tr>
<td>orienting rings (prop 2)</td>
<td>5</td>
<td>102236499</td>
<td>8322.6</td>
<td>2045935</td>
</tr>
<tr>
<td>leader election (odd)</td>
<td>3</td>
<td>55100</td>
<td>6.27</td>
<td>18561</td>
</tr>
<tr>
<td>leader election (complete)</td>
<td>3</td>
<td>6946</td>
<td>0.87</td>
<td>2419</td>
</tr>
<tr>
<td>leader election (complete)</td>
<td>4</td>
<td>63468</td>
<td>11.6</td>
<td>16758</td>
</tr>
<tr>
<td>leader election (complete)</td>
<td>5</td>
<td>598969</td>
<td>176.1</td>
<td>120021</td>
</tr>
<tr>
<td>token circulation</td>
<td>3</td>
<td>728</td>
<td>0.12</td>
<td>244</td>
</tr>
<tr>
<td>token circulation</td>
<td>4</td>
<td>4466</td>
<td>0.35</td>
<td>1138</td>
</tr>
<tr>
<td>token circulation</td>
<td>5</td>
<td>24847</td>
<td>1.86</td>
<td>4971</td>
</tr>
<tr>
<td>token circulation</td>
<td>6</td>
<td>129344</td>
<td>10.7</td>
<td>21559</td>
</tr>
<tr>
<td>token circulation</td>
<td>7</td>
<td>643666</td>
<td>77.2</td>
<td>91954</td>
</tr>
<tr>
<td>token circulation</td>
<td>8</td>
<td>3104594</td>
<td>740.8</td>
<td>388076</td>
</tr>
</tbody>
</table>

Table 1 Experiment results of model checking population protocols with global fairness and symmetry reduction in PAT

7 Related Work

Our work is related to research on applying formal verification techniques to population protocols. In our previous work [34, 35], we applied the Spin model checker to establish the correctness of a family of population protocols and showed that verification under global fairness is infeasible in Spin. As a consequence, we developed an algorithm for the problem of model checking with global fairness and proved that it can be reduced to the problem of searching for a terminal SCC which fails the given property [40]. The algorithm was implemented in the model checker PAT and later on applied to population protocols and previously unknown bugs are detected
The work presented in this chapter makes our effort in the verification of population protocols one step further, by combining symmetry reduction with global fairness in PAT. This enables us to deal with large instances of population protocols in PAT. Deng et al. [14] verified some self-stabilizing population protocols using the theorem prover Coq. Their results are somehow more general, as Coq does not need to fix the number of nodes in the protocols. However, on the other hand, their verification does not handle global fairness directly, as the verified protocols in [14] do not require global fairness. Clément et al. [13] applied counter abstraction to get an abstraction of the population protocol model, which can then be verified by the existing model checker. In their work, they proved sufficient conditions under which global fairness can be replaced by weak fairness in Spin. They also showed how to verify population protocols with global fairness using the probabilistic model checker PRISM. More recently, Méry and Poppleton verified two (simple) population protocols using Event-B and TLA [32].

Our work is also related to research on combining fairness and symmetry reduction. A solution for applying symmetry reduction under weak/strong fairness was discussed in [18]. Their method works by finding a candidate weak/strong fair path in the abstract transition system and then using annotations of state permutation details for each transition, in order to resolve the abstract path to a threaded structure which then determines whether there is a corresponding fair path in the concrete transition system. A similar approach was presented in [21]. Another close work is a nested depth first search algorithm that combines symmetry reduction with weak fairness [7]. Unfortunately, the combined algorithm cannot guarantee to preserve all behaviors under weak fairness and thus may produce false positives. We compare our algorithm with the one which handles strong fairness in [18] in more details. Since global fairness can be regarded as a kind of strong fairness, the algorithm is applicable to global fairness. It is the only algorithm for combining strong fairness and symmetry reduction that we could find in literature. First, Theorem 3.11 in [18] shows its time complexity is $O(|M| \times n^3 \times |g| \times a)$, where $|M|$ is the size of the reduced graph, $n$ is the number of processes, $|g|$ is the length of the checked property, and $a$ is the maximum size of the automaton for any basic modality of $g$. Our algorithm is almost identical to Tarjan’s SCC algorithm except for adding line 24, 25 in Figure 1. For a found SCC $c$ the condition checking in line 24 can be implemented in time linear in the number of edges in $c$. As a result our algorithm can be implemented in time $O(|M| \times |g| \times a)$. Second, in our approach it is not necessary to record permutations appearing on each path (unless unwinding an abstract counterexample) and to construct threaded structure for each strong connected subgraph $B$, of which the size is $O(|B| \times n)$. Hence our algorithm outperforms theirs in space and time. Further, an important practical advantage of our algorithm, unlike [18], is that our algorithm reuses the original algorithm for model checking with global fairness with slight changes.
8 Conclusion

In this chapter, first we have shown that unlike weak/strong fairness, global fairness, which plays a vital role in designing population protocols, can be combined with symmetry reduction. Next, we presented a practical fair model checking algorithm with symmetry reduction and demonstrated its usability by performing model checking of a number of population protocols. Additionally, in [44] we have also proved that the classic notion of partial order reduction [12] cannot guarantee to preserve properties with global fairness. An interesting line of future work is to identify sufficient condition that allows combination of fairness and abstraction in general. In the current implementation, symmetry relationships are assumed to be known or easily detected. In the future, we plan to develop symmetry detection technique (as well as reduction techniques) for hierarchical complex systems.

References


