Verification of Population Ring Protocols in PAT

Yang Liu[‡], Jun Pang[†], Jun Sun[‡], and Jianhua Zhao[⋆]

[†] Université du Luxembourg
Faculté des Sciences de la Technologie et de la Communication

[‡] National University of Singapore
School of Computing

[⋆] Nanjing University
State Key Laboratory of Novel Software Technology

Abstract

The population protocol model has emerged as an elegant paradigm for describing mobile ad hoc *networks, consisting of a number of nodes that interact with each other to carry out a computation. One essential property of self-stabilizing population protocols is that all nodes must eventually converge to the correct output value, with respect to all possible initial configurations. It has been shown that fairness constraints play a crucial role in designing population protocols. The* Process Analysis Toolkit *(PAT) has been developed to perform verifications under different fairness constraints efficiently. In particular, it can handle* global fairness*, which is required for the correctness of most of population protocols. It is an ideal candidate for automatically verifying population protocols. In this paper, we summarize our latest empirical evaluation of PAT on a set of self-stabilizing population protocols for ring networks. We report one previously unknown bug in a protocol for leader election identified using PAT.*

1 Introduction

In distributed computing, when designing or implementing protocols to achieve specific goals, such as mutual exclusion or leader election, it is important to be aware that the correctness of such protocols can only be guaranteed under certain kind of *fairness constraint*. Fairness, which is concerned with a fair resolution of non-determinism, is often necessary to establish liveness properties, meaning something good must eventually happen. Fairness is an abstraction of the fair scheduler in a multi-threaded programming environment or the relative speed of the processors in distributed systems. Without fairness, unrealistic behaviours of the protocols cannot be ignored. For example, without a fair scheduler, it is possible that one processor is infinitely faster than others. It is crucial to rule out these unrealistic behaviours in order to establish the correctness.

Recently, the population protocol model [4] has emerged as an elegant computation paradigm for describing mobile ad hoc networks, consisting of multiple mobile nodes which interact with each other to carry out a computation. Application domains of the protocols include wireless sensor networks and biological computers. One essential property of population protocols is that with respect to all possible initial configurations all nodes must eventually converge to the correct output values (or configurations), which is a typical liveness property. To guarantee that such kind of properties can be achieved, the interactions of nodes in population protocols are subject to fairness. The fairness constraint is imposed on the scheduler 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 [4]. 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 [1, 3, 9, 12, 2]. Fairness plays an important role in these protocols. Most of the protocols only work if *global fairness*[1] is imposed. For instance, it was shown that without global fairness uniform self-stabilizing leader election in rings is impossible [9].

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 if a property, written in some temporal logic, about the system holds or not through an explicit state space search. A counterexample can be generated when the checked property fails to hold, which explains why the formal model does not satisfy the property. In formal verification, fairness is typically used to rule out

[1] Definition of global fairness is given in Section 2.

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. [15, 14, 16]. 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 [11]. The algorithm it uses for generating Büchi automata handles only a limited number of fairness constraints [20]. Pang et al [17, 18] 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 [10] 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. A model checking tool, Process Analysis Toolkit (PAT), is developed to verify system with fairness efficiently and flexibly [20, 21, 19]. It supports different ways of applying fairness and has a unified on-the-fly model checking algorithm which handles a variety of fairness constraints. In particular, it can handle global fairness required for the correctness of most population protocols, which makes PAT an ideal candidate for automatically verifying population protocols.

In this paper, we summarize our latest empirical evaluation of PAT on a set of self-stabilizing population protocols for ring networks. The choice of ring topology makes it less demanding when we model the interactions of nodes and it also makes our models scale up easily. We select protocols for two-hop coloring and orienting nodes and protocols for leader election and token passing. All these protocols only work under global fairness. We report on our model checking results. Especially, we present one previously unknown bug in a leader election protocol [12], which can only be identified using PAT (as far as we know). This work is related to research on verifying distributed systems. It is also remotely related to our previous works on verification and model checking [5, 8, 7].

Roadmap In Section 2, we review the basic population protocol model and define fairness constraints required for population protocols. In Section 3, we briefly introduce PAT, its modelling language and its support for verification under fairness. The population protocols studied in this paper are presented in Section 4, with focus on the counterexample we have found on the leader election protocol. The model checking results are summarized in Section 5. Finally, Section 6 concludes the paper.

2 The Population Protocol Model

In this section, we briefly introduce the population protocol model. More details are available in [1, 9].

2.1 Model and definitions

In the model, the underlying network 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*.

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 $O : Q \to Y$, and
- a transition function $\delta : (Q \times X) \times (Q \times X) \to 2^{Q \times Q}$.

If $(p', q') \in \delta((p, x), (q, y))$, then we write $((p, x), (q, y)) \to (p', q')$ and call it a transition. When δ 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 $C : V \to Q$ assigning to each node its internal state, and an *input assignment* $\alpha : V \to X$ specifies the input for each node. Let C and C' be configurations, α 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 $((C(u), \alpha(u)), (C(v), \alpha(v))) \to (C'(u), C'(v))$, 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 $(C, \alpha) \to C'$.

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, $(C_i, \alpha_i) \to C_{i+1}$.

2.2 Fairness

Let $E = (C_0, \alpha_0), (C_1, \alpha_1), \ldots, (C_i, \alpha_i), \ldots$ be an execution. Two different fairness conditions [9] are defined below:

Global fairness For every C, α, 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.)

Local fairness For every action σ, if σ is enabled in (C_i, α_i) for infinitely many i, then $(C_i, \alpha_i) \xrightarrow{\sigma} C_{i+1}$ for infinitely many i. (Hence, the action σ is taken infinitely many times in E.)

It should be noticed that global fairness is strictly stronger than local fairness [9]. 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 [1, 3, 9, 12, 2] will only work if *global fairness* is assumed. For instance, Fischer and Jiang [9] have proved that without global fairness uniform self-stabilizing leader election in rings is impossible.

In the area of formal verification, there are usually two notions of fairness: weak and strong fairness. A strong fairness condition states that if *an activity* is infinitely often enabled then it has to be executed infinitely often. This can be mapped into the population protocol model as global fairness and local fairness, depending that the activity is either one step or one action. Note that global fairness is more expensive when model checking, as its definition takes all configurations where one action can be enabled into account.

3 Verification under Fairness in PAT

Process Analysis Toolkit (PAT[2]) is developed as an interactive toolkit to support composing, simulating and reasoning of concurrent systems. PAT is developed in C# for the benefit of Object-Oriented design and compatible performance. PAT supports a modeling language which mixes high-level specification language features (deterministic/nondeterministic choice, parallel, interleaving, interrupt, etc.) with programming language features (arrays, while, if-then-else, etc.), so that users are offered with great expressiveness and flexibility.

[2]http://pat.comp.nus.edu.sg

Example. A self-stabilizing population protocol for two-hop coloring is proposed [2]. This algorithm can guarantee that the neighbors of a node in a ring have different colors. Details of this protocol is given in Section 4.

Figure 1 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 *color* models the color of each node. F is a bit array for each node, indexed by colors. Next, line 2 to 9 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 set of predefined 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 with variables updating (e.g., $color[u] = 0$). Line 11 models the two-hop coloring protocol as process *TwoHopColoring*, 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 $\Diamond\Box$*twohopcoloring* (defined as an assertion at line 13), where $\Diamond$ and $\Box$ are modal operators which read as *eventually* and always respectively. *twohopcoloring* (defined at line 12) is a proposition which states that the neighbors of a node in a ring have different colors (for rings of size three).

PAT provides users friendly interfaces for system modeling. User input models can then be simulated using automatic animations. More importantly, PAT supports standard reachability analysis, deadlock checking, refinement checking and verification of LTL properties. PAT finds its strength in two unique aspects: (1) it is designed to verify systems under a variety of fairness assumptions, including local/gloabl fairness; (2) it supports mechanical refinement checking, possibly with data refinement relationships. In order to handle large systems, PAT is further improved with effective reduction techniques like partial order reduction. In this paper, we focus on its support for efficient verification under global fairness, which is partly motivated by recently developed population protocols (which only work under global fairness). The other motivation is that the current practice of verification is deficient under fairness.

Existing verification algorithms/tools are ineffective with respect to fairness. In automata based LTL model checking, the negation of a property is translated to an equivalent Büchi automaton, which is then composed with the automaton representing the system for analysis. The size of the Büchi automaton is exponential to the size of the property. Existing model checkers for verification under fairness is to re-formulate the property so that fairness as-

```
1. #define N 3; #define C 3; var color[N]; var F[N][C];
2. Interaction(u, v) =
3.       if (F[u][color[v]]! = F[v][color[u]]){
4.             act1.u.v{F[u][color[v]] = F[v][color[u]]; color[u] = 0; } -> Interaction(u, v)
5.           [] act2.u.v{F[u][color[v]] = F[v][color[u]]; color[u] = 1; } -> Interaction(u, v)
6.           [] act3.u.v{F[u][color[v]] = F[v][color[u]]; color[u] = 2; } -> Interaction(u, v)
7.       } else {
8.           act4.u.v{F[u][color[v]] = 1 − F[u][color[v]]; F[v][color[u]] = 1 − F[v][color[u]]; } -> Interaction(u, v)
9.       };
10. Init() = ...
11. TowHopColoring() = Init(); (||| x : {0..N − 1}@Interaction(x, (x + 1)%N) ||| (Interaction((x + 1)%N, x));
12. #define twohopcoloring(color[0]! = color[2]&&color[1]! = color[2]&&color[0]! = color[1]);
13. #assert TowHopColoring() |= <> []twohopcoloring;
```

Figure 1. PAT Model for Two Hop Coloring Protocol

sumptions become premises of the property. Model checking under fairness is then to search for an infinite execution which is accepting to the Büchi automaton and at the same time satisfies the fairness assumptions. This approach becomes impractical when fairness assumptions have complex structures. Especially, when model checking population protocols, global fairness must take all configurations where one action can be enabled into account, which often makes the size the property very large.

In [20], a unified algorithm is presented to verify whether a system is feasible under different fairness assumptions. It avoids the problem of constructing a Büchi automaton from a property with (global) fairness as the premise of the property. A system is *feasible* if and only if there exists at least one infinite execution which satisfies the fairness assumptions. Applied to the product of the system and the Büchi automaton, the algorithm can be easily extended to do model checking with fairness. Because of fairness, nested depth-first-search [11] is not feasible, the algorithm is therefore based on Tarjan's algorithm [22] for identifying strongly connected components. We refer interested readers to the paper [20] for detailed discussions.

4 Population Ring Protocols

In this section, we take a set of self-stabilizing population protocols for ring networks. A distributed system or a population protocol is said to be *self-stabilizing* [6] 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 modelling of these protocols, we have to take all possible initial configurations into account, and the checked properties have the form of $\Diamond\Box$*property*. The choice of ring topology makes it less demanding when we model the interactions of nodes (see the example in Section 3) and it also makes our models easily scale up to larger instances. 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 $N-1$.[3] 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.

4.1 Two hop coloring

A protocol to make nodes to recognize their neighbors in a ring is presented in [2]. 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 [2]).

Each node u in a ring has two state components, $color[u]$ encodes the color of node u and $F[u]$ is a bit array, indexed by colors. Initially, $color[u]$ and $F[u]$ can have arbitrary val-

[3] In the following discussion, we set N as three for simplicity.

ues. The following description defines the interaction between an initiator u and a responder v.

Nondeterministic two-hop coloring protocol.

if $F[u][color[v]] \neq F[v][color[u]]$ **then**
  $color[u] \leftarrow color'[u]$
  $F[u][color[v]] = F[v][color[u]]$
else $F[u][color[v]] = \neg F[u][color[v]]$
  $F[v][color[u]] = \neg F[v][color[u]]$ **endif**

One edge (or interaction) (u, v) is synchronized if $F[u][color[v]] = F[v][color[u]]$, then these two nodes do not change their color but flip their bits ($F[u][color[v]]$ and $F[v][color[u]]$). On the other hand, node u is nondeterministically recolored, and it copies $F[v][color[u]]$ of node v as its bit $F[u][color[v]]$. The statement $color[u] \leftarrow color'[u]$ 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 Section 3. In [2], a deterministic version of two-hop coloring is given as well (see below). Instead of nondeterministically assigning all possible colors to the initiator u, its color is updated as $color[u] \leftarrow (color[u] + r[u])$ mod C. The additional state component $r[u]$ is a local bit for node u that flits whenever u acts as the initiator of an interaction. We also model and analyse this protocol in PAT.

Deterministic two-hop coloring protocol.

if $F[u][color[v]] \neq F[v][color[u]]$ **then**
  $color[u] \leftarrow (color[u] + r[u])$ mod C
  $F[u][color[v]] = F[v][color[u]]$
else $F[u][color[v]] = \neg F[u][color[v]]$;
  $F[v][color[u]] = \neg F[v][color[u]]$ **endif**
$r[u] \leftarrow \neg r[u]$

4.2 Orienting undirected rings

Given a ring colored by protocols in Section 4.1, it is possible to have a protocol that gives a sense of orientation to each node on an undirected ring [2]. 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. 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 $color[v] \neq succolor[u]$ **then**
  $succolor[v] \leftarrow color[u]$
elseif $color[v] = succolor[u]$ and $color[v] \neq precolor[u]$ **then**
  $precolor[v] \leftarrow color[u]$
else $precolor[u] \leftarrow color[v]$; $succolor[v] \leftarrow color[u]$ **endif**

The PAT model of this protocol is shown in Figure 2 in the appendix. Lines 2-8 model how two nodes can interact. The initialization at line 9 makes sure that the nodes are initially two-hop colored. Line 10 defines 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 13 and 14. For example, $property1$ formalizes that the predecessor and successor of a node are different.

4.3 Leader election

In this section, we study a leader election protocol in oriented odd rings. The following description is partially taken from [12, 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 [12, 2].

The PAT model of this protocol is shown in Figure 3 in the appendix. Lines 2-9 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 10, it captures any possible evaluations of the variables. Line 11 defines how nodes interact in an oriented ring. Line 12 defines a predicate that there is one leader in the network. Line 13 claims that the protocol eventually self-stabilize to a unique leader existing in the network.

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

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. 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 $v = 0$)

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

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

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

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 [12], it is confirmed as a valid counterexample which has escaped simulations of the protocol [13]. The reason to the counterexample is the following [13]. 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 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 [20] 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.

4.4 Token circulation

The token circulation protocol in directed rings depicted below is proposed in [1, 2]. The desired behavior of this

[4] 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.

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. $((*\, b,\ \ N), (*\, b,\ \ L)) \ \rightarrow\ ((-\, b), (+\, \bar{b}))$
Rule 2. $((*\, b,\ \ *), (*\, \bar{b},\ \ N)) \ \rightarrow\ ((-\, b), (+\, b))$

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. 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 4 in the appendix. 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 $label[N]$). Without loss of generality, we assume that node 0 is always the leader. Therefore, we could simply set each node a fixed input ($leader[i]$) for leader election without considering complicated details of a dynamic leader election process, which we have analyzed in Section 4.3.

5 Verification Results in PAT

Table 1 collects the experimental results of all protocols presented in the paper. For the two-hop coloring protocol, there are two version: [1] for nondeterministic and [2] for deterministic. For the orienting undirected ring protocol, both properties in Figure 2 are checked. Leader election protocol is only checked for odd rings as required. All protocols, together with many other system models have been built inside the latest release of PAT 2.0[5]. The experiment testbed is a PC running Windows XP SP3 with 2.83GHz Intel Q9550 CPU and 4 GB memory. In the table, '$-$' means out of memory. Windows XP allocates maximum 2GB memory for each application, which limits the model checking for larger state spaces. We skip the statistics on memory consumption because the dynamic garbage collection facility in C# makes the estimation inaccurate. Nonetheless, the number of states and transitions reflect the memory usage.

From the table, firstly it shows that the number of states, transitions and running time increase rapidly (exponentially) with the number of nodes in rings, especially for two-hop coloring and leader election protocols. The reason is that these protocols use more state components than the others, e.g., the arrays. This conforms to the theoretical results. Secondly, we show that PAT is effective, it can handle millions states in hundreds of seconds (which is compatible to Spin [11]). Notice that Spin is infeasible for verifying the protocols because it does not support the fairness notions[6]. Although we are bound to check relatively small instances of the protocols, the newly developed verification techniques in [20], does complement existing model checkers with the improvement in terms of the performance and ability to handle different forms of fairness. It enables us to establish the correctness of these protocols under global fairness or, in the case of the leader election protocol, identify bugs. Readers can compare the result presented in [17] on a similar verification practice using the Spin model checker. The argument for using model checking techniques in general, is that, if there is a bug in the protocol design, probably it is present in a small network.

6 Concluding Remarks

In the literature, a number of population protocols have been proposed to solve problems in wireless sensor networks. The correctness of these protocols relies on global fairness, which makes their automatic verification using existing model checkers expensive or even infeasible. In this paper, we have applied PAT, a newly developed toolkit handling verification under fairness more efficiently, to a set of self-stabilizing population ring protocols. We have shown that the model checking algorithm [20] behind PAT allows us to successfully verify instances of these protocols. Moreover, it has helped us to identify one previously unknown bug in a leader election protocol.

During the analysis, we have faced the infamous state explosion problem (see Table 1). In future, we will explore how to combine different state space reduction techniques with the feasibility checking algorithm in [20]. For example, we want to explore how to perform abstraction with the presence of global fairness. The immediate future work is to apply PAT to other population protocols, such as self-stabilizing consensus protocols [3, 2].

[5] `http://www.comp.nus.edu.sg/~pat/`.

[6] Spin supports only process-level weak fairness.

Model	Property	Ring Size	Results	#States	#Transitions	Time (Sec)
two-hop coloring[1]	$\Diamond\Box$*twohopcoloring*	3	Yes	122856	1972174	43.3
two-hop coloring[1]	$\Diamond\Box$*twohopcoloring*	4	Yes	–	–	–
two-hop coloring[2]	$\Diamond\Box$*twohopcoloring*	3	Yes	983016	9473998	627
two-hop coloring[2]	$\Diamond\Box$*twohopcoloring*	4	Yes	–	–	–
orienting rings	$\Diamond\Box$*property*1	3	Yes	3200	28540	0.61
orienting rings	$\Diamond\Box$*property*2	3	Yes	3221	28163	0.64
orienting rings	$\Diamond\Box$*property*1	4	Yes	69766	883592	18.1
orienting rings	$\Diamond\Box$*property*2	4	Yes	66863	794662	17.5
orienting rings	$\Diamond\Box$*property*1	5	Yes	1100756	18216804	601
orienting rings	$\Diamond\Box$*property*2	5	Yes	1021851	15486265	536
leader election	$\Diamond\Box$*oneleader*	3	Yes	55100	216699	10.6
leader election	$\Diamond\Box$*oneleader*	5	Yes	–	–	–
token circulation	$\Diamond\Box$*onetoken*	3	Yes	244	655	0.07
token circulation	$\Diamond\Box$*onetoken*	4	Yes	1118	3870	0.13
token circulation	$\Diamond\Box$*onetoken*	5	Yes	4971	20838	0.58
token circulation	$\Diamond\Box$*onetoken*	6	Yes	21559	105577	2.86
token circulation	$\Diamond\Box$*onetoken*	7	Yes	91954	514703	14.9
token circulation	$\Diamond\Box$*onetoken*	8	Yes	388076	2446736	88.6

Table 1. Experiment Results

References

[1] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. In *OPODIS'05*, volume 3974 of *LNCS*, pages 103–117. Springer, 2005.

[2] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing Population Protocols. *ACM Transactions on Autonomous and Adaptive Systems*, 3(4):643–644, 2008.

[3] D. Angluin, M. J. Fischer, and H. Jiang. Stabilizing Consensus in Mobile Networks. In *DCOSS'06*, pages 37–50. IEEE Computer Society, 2006.

[4] J. Aspnes and E. Ruppert. An Introduction to Population Protocols. *Bulletin of the European Association for Theoretical Computer Science*, 93:98–117, 2007.

[5] C. Q. Chen, J. S. Dong, and J. Sun. A Verification System for Timed Interval Calculus. In *ICSE'08*, pages 271–280. ACM, 2008.

[6] E. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, 1974.

[7] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *ICFEM'06*, volume 4260 of *LNCS*, pages 342–359. Springer, 2006.

[8] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *ICFEM'06*, volume 4260 of *LNCS*, pages 226–245. Springer, 2006.

[9] M. J. Fischer and H. Jiang. Self-stabilizing Leader Election in Networks of Finite-state Anonymous Agents. In *OPODIS'06*, volume 4305 of *LNCS*, pages 395–409. Springer, 2006.

[10] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly LTL Model Checking. In *TACAS'05*, volume 3440 of *LNCS*, pages 191–205. Springer, 2005.

[11] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[12] H. Jiang. *Distributed Systems of Simple Interacting Agents*. PhD thesis, Yale University, 2007.

[13] H. Jiang. Personal communications, 2008.

[14] Y. Kesten, A. Pnueli, L. Raviv, and E. Shahar. Model Checking with Strong Fairness. *Formal Methods in System Design*, 28(1):57–84, 2006.

[15] T. Latvala and K. Heljanko. Coping with Strong Fairness. *Fundamenta Informaticae*, 43(1-4):175–193, 2000.

[16] M. Musuvathi and S. Qadeer. Fair stateless model checking. In *PLDI'08*, pages 362–371. ACM Press, 2008.

[17] J. Pang, Z. Luo, and Y. Deng. On Automatic Verification of Self-stabilizing Population Protocols. In *TASE'08*, pages 185–192. IEEE Computer Society, 2008.

[18] J. Pang, Z. Q. Luo, and Y. X. Deng. On Automatic Verification of Self-stabilizing Population Protocols. *Frontiers of Computer Science in China*, 2(4):357–367, 2008.

[19] J. Sun, Y. Liu, and J. S. Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *ISoLA'08*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008.

[20] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV'09*, 2009. to appear.

[21] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Specifying and Verifying Event-based Fairness Enhanced Systems. In *ICFEM'08*, volume 5256 of *LNCS*, pages 318–337. Springer, 2008.

[22] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 2:146–160, 1972.

Appendix

```
1. #define N 3; #define C 3; var color[N]; var precolor[N]; var succolor[N];
2. Interaction(u, v) = if (color[v] == precolor[u]&&color[v]! = succolor[u]){
3.                         act1.u.v{succolor[v] = mycolor[u]; }-> Interaction(u, v)
4.                     } elseif (color[v] == succolor[u]&&color[v]! = precolor[u]){
5.                         act2.u.v{precolor[v] = color[u]; }-> Interaction(u, v)
6.                     } else {
7.                         act3.u.v{precolor[u] = color[v]; succolor[v] = color[u]; }-> Interaction(u, v)
8.                     };
9. Init() = ...
10. OrientingUndirected() = Init(); ||| x : {0..N − 1}@(Interaction(x, (x + 1)%N) ||| Interaction((x + 1)%N, x));
11. #define property1 (x : 0..N − 1@precolor[x]! = succolor[x]));
12. #define property2 (...);
13. #assert OrientingUndirected() |= <> []property1;
14. #assert OrientingUndirected() |= <> []property2;
```

Figure 2. PAT Model for Orienting Undirected Ring Protocol

```
1. #define N 3; var leader[N]; var label[N]; var probe[N]; var phase[N]; var bullet[N];
2. Interact(u, v) =
3.     [label[u] == label[v]&&probe[u] == 1&&phase[u] == 0]
4.         act1.u.v{leader[u] = 1; probe[u] = 0; bullet[v] = 0; phase[u] = 1; probe[v] = 1;}-> Interact(u, v)
5.     [][label[u] == label[v]&&probe[u] == 1&&phase[u] == 1&&probe[v] == 0]
6.         act2.u.v{leader[u] = 1; probe[u] = 0; bullet[v] = 0; label[v] = 1 − label[v]; phase[v] = 0;}-> Interact(u, v)
7.     []...
8.     [][label[u]! = label[v]&&leader[v] == 0&&bullet[v] == 1&&probe[v] == 0]
9.         act11.u.v{bullet[u] = 1; bulllet[v] = 0;}-> Interact(u, v)
10. Init() = ...
11. LeaderElection() = Init(); (||| x : 0..N − 1@Interaction(x, (x + 1)%N));
12. #define leaderelection (leader[0] + leader[1] + leader[2] = 1);
13. #assert LeaderElection() |=<> []leaderelection;
```

Figure 3. PAT Model for Leader Election Protocol in Odd Rings

```
1. #define N 3; var leader[N]; var label[N]; var token[N];
2. Rule1(u, v) = [!leader[u]&&leader[v]&&label[u] = label[v]]
3.                     rule1.u.v{token[u] = 0; token[v] = 1; label[v] = 1 − label[u];}-> Rule1(u, v);
4. Rule2(u, v) = [!leader[v]&&label[u]! = label[v]]
5.                     {rule2.u.v{token[u] = 0; token[v] = 1; label[v] = label[u];}-> Rule2(u, v)};
6. Init() = ...
7. TokenCirculation() = Init(); (||| x : 0..N − 1@(Rule1(x, (x + 1)%N) ||| (Rule2(x, (x + 1)%N));
8. #define onetoken(token[0] + token[1] + token[2] == 1);
9. #assert TokenCirculation() |=<> []onetoken;
```

Figure 4. PAT Model for Token Circulation Protocol