On Automatic Verification of Self-stabilizing Population Protocols

Jun Pang
University of Luxembourg
Computer Science and Communications
L-1359 Luxembourg

Zhengqin Luo Yuxin Deng
Shanghai Jiao Tong University
Department of Computer Science and Engineering
200240 Shanghai, China

Abstract

The population protocol model [2] 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 fairness constraint. One essential property of population protocols is that all nodes must eventually converge to the correct output value (or configuration). In this paper, we aim to automatically verify self-stabilizing population protocols for leader election and token circulation in the Spin model checker [11]. We report our verification results and discuss the issue of modeling strong fairness constraints in Spin.*

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 subresults, for which only informal arguments were given. This has been observed and illustrated in a recent paper [8] on formal reasoning about the correctness of a distributed consensus algorithm [4].

The last two decades have also seen an impressive amount of new techniques developed in the area of *formal verification* or *model checking*. 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, and predicate abstraction, 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.

Clearly, both two fields, distributed algorithms vs. formal verification, can benefit from each other. On the one hand, formal verification can offer techniques to well-understand distributed algorithms. On the other hand, distributed algorithms can offer challenging applications to formal verification.

We apply model checking to self-stabilizing population protocols in this paper. The population protocol model [2] 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 [2]. That is why for population protocols correctness arguments are always rephrased as a property to be satisfied eventually. In formal verification, fairness is typically used to rule out some *unrealistic* runs due to non-determinism, i.e., it mainly concerns with a fair resolution of non-determinism in the models. So unsurprisingly, fairness has been a research topic to both communities, see [3, 14, 17, 16, 5].

In this paper, we aim to automatically verify self-stabilizing population protocols for leader election and token circulation in the Spin model checker [11]. In next section we review the basic population protocol model and the fairness conditions which are required for population protocols. The general framework for modeling population protocols in Spin is given in Section 3. In Section 4 and Section 5, we discuss experiment results on automatic verification of self-stabilizing leader election for complete graphs [6] and token circulation for directed rings [1], respectively. For leader election in complete graphs, we show that the algorithm also works under a weaker fairness condition. For token circulation in directed rings, the algorithm is model checked in a two-phase manner. We first show that under a particular activation order of nodes, satisfying the *global fairness* condition, some pre-defined safe configurations will be eventually reached. Then we show that from these safe configurations eventually token circulation is stabilized. In Section 6, we demonstrate that *global fairness* generally assumed for population protocols is necessary. We present counter-examples that we have observed in Spin to show that self-stabilizing token circulation cannot be achieved with *local fairness*. Finally, we discuss the difficulty of modeling *global fairness* in Spin and conclude the paper by pointing out some possible future work in Section 7.

2 The Population Protocol Model

We briefly introduce the population protocol model in this section, more details are available in [1, 6].

2.1 Model and definitions

In our framework, the underlying network can be described by 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 communicate 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 is a transition. When δ always maps to a set that only contains a single pair of states, then we call the protocol and the transition function *deterministic*.

A *configuration* $\mathcal{C}$ 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 $\delta((C(u), \alpha(u)), ((C(v), \alpha(v))) \to (C'(u), C'(v))$, 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 $(C_0, \alpha_0), (C_1, \alpha_1)$ such that for each i, $(C_i, \alpha_i) \to C_{i+1}$.

2.2 Fairness conditions

In the following, we first summarize the fairness conditions for population protocols. Let $E = (C_0, \alpha_0), (C_1, \alpha_1), \ldots, (C_i, \alpha_i), \ldots$ be an execution.

Global fairness For every C, α, and C' such that $(C_i, \alpha_i) \to C_{i+1}$, 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.

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.

It should be noticed that the global fairness is strictly stronger than the local fairness [6]. The global fairness requires that each *step* that can be taken infinitely often is actually taken infinitely often, while the 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, the global fairness insists that an action must be taken infinitely often in all such configurations, whereas the local fairness only requires that it occurs infinitely often in one of such configurations.

In [6], two extra weak notions of fairness conditions are presented. The weak forms of fairness do not insist that particular steps occur infinitely often in E but only that the configurations that would result from those steps occur infinitely often. The relationship between these four kinds of fairness conditions are discussed thoroughly in [6].

As discussed before, in the area of formal verification, fairness is typically needed to prove liveness properties. It is concerned with a fair resolution of non-determinism, i.e., fairness conditions are used to rule out some *unrealistic* runs due to non-determinism. Usually, in formal verification 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 the global and local fairness above, depending that the activity is either one step or one action. There is another notion of weak fairness in verification, stating that if an activity is continuously often enabled (*no temporary disabling*) then it has to be executed infinitely often.

Weak fairness For every action σ, if there exists i and for all $j > i$, σ is is always enabled at (C_j, α_j) in E, then there exist C, α, C' such that $(C, \alpha) \xrightarrow{\sigma} C'$ occurs infinitely often.

This form of fairness is supported in Spin, see Section 3 for details. Note that we cannot have a similar fairness condition in which an action becomes enabled forever with respect to one concrete configuration, since the configuration of the system will be updated by other actions.

3 Automatic Verification in Spin

The Spin model checker is a popular tool set for verification of concurrent systems [11, 10]. A modeling language Promela (Process Meta Language) is used to specify a concurrent system consisting of some processes that are the basic dynamic system components. Given a model described in Promela, Spin can either run random simulations to check the validation of functional behavior, or generate efficient C programs to verify the correctness with respect to some constraint conditions. Its verifier can find non-progress cycles, or verify general properties which are expressed by linear temporal logic (LTL) formulas. The verifier also provides an option for weak fairness among processes, that is, if a process is eventually permanently enabled in the run, then the process is executed infinitely often in the run. The reader is referred to [12] for more details.

In the population protocol model, one protocol consists of N nodes, numbered from 0 to $N-1$. The protocol is usually described by a set of rewriting rules. On the left hand side of each rule, the state and the input of the initiator and the responder should be matched by the rules. On the right hand side, the rule specifies the state of the initiator and the responder after the transition has been taken.

Since the population protocols always depend on some kind of fairness condition, such as the global and local fairness, we attempt to use the weak fairness condition (on processes) in Spin to model a fairness condition in population protocols. However, the fairness condition in population protocol model is related to actions/steps but not to nodes (processes). Thus, if we use a single process in Promela to model a single node of the population protocol model, then the weak fairness condition only guarantees that if a node is from some moment onward always enabled, then it will be executed infinitely often. This obviously does not make any sense when verifying population protocols. Our strategy is to use a single process to represent an action which is related to an initiator, a responder and a rewriting rule. The process declaration for a transition in Promela is described in the following way:

```
proctype Rulen(int i; int r)
```

The parameters i and r are identities of an initiator and a responder. The process name Rule$n$ corresponds to some concrete rule n in the algorithm. The state of each node is stored in some global variable. The process will check the global configuration to decide the executability of its own. For example, Rule1(1,2) represents an action between node 1 as an initiator and node 2 as a responder according to rule 1. The entire system consists of all possible actions between every pair of nodes which can communicate with each other.

```
run Rule1(0,1);
run Rule1(1,0);
run Rule1(1,2);
...
run Rule2(2,0);
...
```

The different communication patterns are determined by different network topologies, such as complete graphs and directed rings.

By using the weak fairness condition in the Spin model checker, we immediately obtain a condition strictly weaker than the local fairness in the population protocol model. The condition is exactly the *weak* *fairness* as discussed in Section 2.2. Otherwise we have to use a large LTL formula and auxiliary variables to characterize the strong (global and local) fairness conditions which will increase the complexity of the model (see more discussion in Section 7). The weak fairness condition only assumes that if an action is permanently enabled from some point, then it will be taken infinitely many times. Most self-stabilizing population protocols require either global or local fairness. However, we have found that some of them also work properly under this weak fairness condition implemented in the Spin model checker. In the following sections, protocols will be verified under this weak fairness condition.

Once a model has been built, we could define a bunch of propositions which refer to different system states. Finally, LTL formulas over these propositions can be used to specify some desired behaviors of the protocol. The LTL formulas will be translated into never claims in Prolema automatically, and verified by the Spin model checker.

4 Self-stabilizing Leader Election in Complete Graphs

In this section, we show that self-stabilizing leader election in complete graphs can be achieved under the weak fairness with the help of an eventually correct leader detector. The algorithm was originally given in [6]. Every node has one bit memory which represents two states, being a leader or not. The leader detector gives each node an input true (T) or false (F) to indicate that whether there is a leader in the network. The detector may give wrong answers sometimes, but it will eventually return a correct answer permanently.

The algorithm is described by the three rewriting rules in Figure 1. On the left hand side, the state and input of an initiator and a responder should be matched. The symbol "$*$" denotes that the input can always be matched. On the right hand side, the state of the two nodes would be updated by the rule.

Rule 1. $((L,*),(L,*)) \rightarrow ((L),(-))$
Rule 2. $((-,F),(-,*)) \rightarrow ((L),(-))$
Rule 3. $((-,T),(-,*)) \rightarrow ((-),(-))$

Each node outputs its own state.

Figure 1. Algorithm for self-stabilizing leader election in complete graphs.

In [6], it has already been shown that the algorithm implements self-stabilizing leader election in complete graph under both global and local fairness, provided the existence of an eventual leader detector. Here we have new verification result to show that the algorithm is correct even under the weak fairness condition.

4.1 Modeling leader election in Spin

The model for leader election in complete graphs follows the general paradigm of population protocol in Section 3, only with some additional definition issues.

The states of the whole system are represented by an array of bits leader[N], N is the number of nodes in the network. When leader[i] equals to 1, it indicates that node i claims to be a leader. Since we are modeling self-stabilizing protocols, we have to ensure that the protocol is correct starting from any arbitrary initial configuration. We employ atomic sequences and case selection in Promela to assign all possible values for every state variable in a single step.

```
atomic{
if
    :: atomic{true -> leader[0]=0}
    :: atomic{true -> leader[0]=1}
fi;
...
}
```

Thus, at the beginning of the system, every state variable could be assigned all possible values. The verifier will check all these cases to ensure the self-stabilizing property of the system.

Besides, we have to model the eventually correct leader detector in the protocol. The detector is defined by two parts. First, there is a random process which randomly generates answers (encoded in the variable detector) when the detector is in "incorrect" state:

```
proctype RandomDetector() {
do
    :: (detectorcorrect == 0) -> detector = false;
    :: (detectorcorrect == 0) -> detector = true;
od
}
```

Then, we define another process that can switch the detector's state from "incorrect" to "correct" in a nondeterministic way. The progress label ensures that the transition will finally occur.

```
proctype DetectorCorrect() {
do
    :: (detectorcorrect == 0) ->
       progress: detectorcorrect = 1;
od
}
```

Once detectorcorrect becomes true, the value of detector will depend on whether the sum of leader[i] ($0 \leq i < N$) is greater than 0.

Having defined the model, the LTL formula which specifies the desirable system behavior is relatively small. Under the weak fairness, the LTL formula for leader election in complete graph is simply as follows:

$$<> [\ ]\ \mathsf{oneLeader}$$

This LTL formula says that along every path which satisfies the weak fairness condition a unique leader will eventually be elected. Here, oneLeader is the proposition stating that the sum of all leader[i] equals one. See [15] for the detailed model.

4.2 Verification results

It has been shown in [6] that the algorithm is valid under the local fairness condition, and the fact that the global fairness implies the local fairness condition yields that the algorithm is also valid under the global fairness. However, the weak fairness used in the verification model is weaker than both of them. Thus it is interesting to see if it is correct under such a weaker fairness condition. Surprisingly, the algorithm indeed implements self-stabilizing leader election in complete graphs. We have verified the model with size up to six. The detailed results are given in Figure 2.

	State size	Transition size	Time	Results
LE-3	558	92974	0.45s	valid
LE-4	1661	629905	5.29s	valid
LE-5	4856	3335330	41.71s	valid
LE-6	13629	14810700	264.07s	valid

Figure 2. Verification results of leader election algorithm under the weak fairness.

4.3 Correctness under weak fairness

In this section, we show that the self-stabilizing leader election algorithm is correct for any number of nodes ($N \geq 2$), under the weak fairness condition. Our proof follows the scheme in [6]. We call a configuration of the protocol *safe*, if there are at least one node that has become leader.

Lemma 4.1. *Let E be an execution of the algorithm starting from an arbitrary configuration. Then E contains a safe configuration.*

Theorem 4.2. *Given an eventually correct leader detector. Let E be a weak fair execution of the algorithm starting from an arbitrary configuration. Then eventually one unique leader will be elected.*

The proof of Lemma 4.1 is the same as in [6]. In the following, we give the proof of Theorem 4.2, which is also similar to the one in [6].

Proof. By Lemma 4.1, the algorithm can reach a safe configuration. By the rules of the algorithm, the number of leaders decreases by one only when two leaders interact via rule 1. So there is always at least one leader in subsequent configurations. By the eventually correct leader detector, eventually all nodes will receive T, after which no more new leaders can be generated. Now we prove that the number of leaders eventually decreases using the weak fairness. We only consider the case when there are more than one leader in the configuration. Assume node i and node j are leaders. Since the graph is complete, the interaction between i and j is enabled via rule 1. By subsequent steps in E,

- either they cannot change the state of either i or j, then the interaction between i and j keeps enabled. By weak fairness, finally the interaction will take place, and the number of leaders decreases;
- or they can change the state of either i or j, then this must be done via rule 1, since that is the only way to change the state of one node once the leader detector is correct. The interaction between i and j is disabled, and the number of leaders is decreased by one.

□

5 Token Circulation in Directed Rings

The token circulation protocol in directed rings is proposed in [1]. The desired behavior of this protocol can be described as follows:

- There is only one node who holds the token.
- A node does not obtain again until every other node has obtained a token once.
- Each node can have the token infinitely often.

The protocol is simple since we do not consider the case that some nodes are not willing to release the token. 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 common leader. 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 algorithm is given by the rewriting rules in Figure 3.

$$\textit{Rule } 1.\ ((*\, b, N), (*\, b, L)) \rightarrow ((-b), (+\, \bar{b}))$$
$$\textit{Rule } 2.\ ((*\, b, *), (*\, \bar{b}, N)) \rightarrow ((-\, b), (+\, b))$$

Figure 3. Algorithm for token circulation in directed rings.

The $*$ here still 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.

It has been proved in [13] that this algorithm implements a self-stabilizing token circulation in rings under the global fairness condition, provided that there is a unique leader.

5.1 Modeling token circulation in Spin

The model for token circulation protocol is similar to the one in Section 4, only with some minor adaptation. The states of the whole system are represented by three arrays of bits leader[N], token[N] and label[N], where N is the number of nodes in the network. Without loss of generality, we can 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.

We still use each single process in Promela to model a single action between each possible initiator and responder. However, now the network topologies are directed rings instead of complete graphs. So each node can only be the responder of its predecessor in a ring. Thus, the system is represented as below.

```
...
run Rule1(0,1);
run Rule1(1,2);
run Rule1(2,3);
...
```

The verification goal is represented by a conjunction of three LTL formulas, each for a goal of the protocol. For the first goal that there is only one token in the network, we use the LTL formula

```
<> [ ] oneToken
```

where oneToken is the predicate stating the sum of all token[i] ($0 \leq i < N$) equals one.

For the second goal that a node does not obtain again until every other node has obtained a token once, it is obviously equivalent to the one that when a node is holding a token, nobody could obtain the token until its successor has obtained it once. For example, if the network has four nodes, then the assertion for node 2 can be specified by the LTL formula

```
<> [ ] (token[2] -> (!token[0] & !token[1] U token[3])).
```

For the last goal that every node obtains the token infinitely often, we use a formula in the following form:

```
[ ] <> token[0] & ... & [ ] <> token[N-1]
```

We have done some experiments for the model under the weak fairness, and the results are mostly negative. The protocol is correct only when the size of the network is three. When it comes to a size greater than three, the Spin verifier complained about some failure traces which satisfy the weak fairness condition. Therefore, the weak fairness cannot guarantee the correctness of the protocol.

Since the token circulation in rings does not work properly under the weak fairness condition, we need to verify it under global fairness condition. However, with the ability limitation of Spin model checker (see more discussion in Section 7), it seems infeasible to explicitly model the global fairness. Here we use an alternative method to verify the protocol. The algorithm is model checked in a two-phase manner. We first show that under a particular activation order chosen by a scheduler (under the global fairness condition) safe configurations are eventually reached. Then we show from these safe configurations the token circulation is eventually stabilized (under the weak fairness). The idea follows the correctness proof of the protocol given in [13].

The safe configurations refer to those configurations in which all nodes have a same label. With a scheduler satisfying the global fairness, the order of activation for nodes complies the following sequence:

$$(0,1), (1,2), \ldots, (N-2, N-1)$$

If there is no possible interaction between nodes i and $i+1$ while the scheduler selects the activation $(i, i+1)$, then the scheduler just turns to the next pair of nodes $(i+1, i+2)$.

In the corresponding model in Spin, we employ a special variable turn to record the current activation pair. Every transition process will check whether it is its turn to do the transition. If it is the case, the process will possibly be able to do the transition according to the states of corresponding nodes. After the transition having been done, the turn flag variable will be increased by one. Those transitions which are not selected by the scheduler will block itself. A special watch-dog process is needed to handle the case when no transition is enabled.

```
:: timeout ->
   if
       :: turn <= (N-2) -> turn=turn+1;
   fi;
```

The watchdog process uses a timeout to detect the block state of the entire system. Thus, by using this model, we can carry out the first part of the verification, checking whether a safe configuration is reachable from any initial configuration.

For the second part of the verification, we only need to show that each of the three protocol goals is satisfied from the same-label initial configurations. Thus we only generate two possible initial configurations nondeterministically at the beginning of the system run. The modification is straightforward. See [15] for the detailed model.

5.2 Verification results

For the first phase, we can see from Figure 4 that it is indeed the case that some safe configuration is reachable under the particular activation order.

	State size	Transition size	Time	Results
TC-4	203	1114	0.01s	valid
TC-5	475	3466	0.01s	valid
TC-6	1083	11434	0.02s	valid

Figure 4. Verification results of token circulation algorithm (part I).

As to the second phase, the verification shows that the weak fairness is enough to ensure the correctness of the protocol after reaching a safe configuration. The results are shown in Figure 5. Note that the global fairness is needed for the overall verification task.

	State size	Transition size	Time	Results
TC-4	1525	10121	0.02s	valid
TC-5	7063	36831	0.13s	valid
TC-6	19287	111535	0.42s	valid

Figure 5. Verification results of token circulation algorithm (part II).

6 Counter-examples for Local Fairness

In this section, we give one counter-example [1] as evidence that self-stabilizing token circulation in directed rings does require the global fairness, the local fairness condition cannot guarantee its correctness.

In Section 5 we have shown that the algorithm is correct under the weak fairness for three nodes. Hence, the algorithm is also correct under local fairness for a network of three nodes. In order to get counter-examples under the local fairness condition, we need a network consisting of at least four nodes (and without using the particular activation order in Section 5).

We need to first present eight configurations which are involved in the counter-example, denoted by $C_1, C_2, \ldots, C_8$:

$$C_1 = \begin{pmatrix} \mathsf{Node}_0 : + & 1 \\ \mathsf{Node}_1 : + & 0 \\ \mathsf{Node}_2 : - & 1 \\ \mathsf{Node}_3 : + & 1 \end{pmatrix} C_2 = \begin{pmatrix} \mathsf{Node}_0 : + & 1 \\ \mathsf{Node}_1 : - & 0 \\ \mathsf{Node}_2 : + & 0 \\ \mathsf{Node}_3 : + & 1 \end{pmatrix}$$

$$C_3 = \begin{pmatrix} \mathsf{Node}_0 : - & 1 \\ \mathsf{Node}_1 : + & 1 \\ \mathsf{Node}_2 : + & 0 \\ \mathsf{Node}_3 : + & 1 \end{pmatrix} C_4 = \begin{pmatrix} \mathsf{Node}_0 : + & 0 \\ \mathsf{Node}_1 : + & 1 \\ \mathsf{Node}_2 : + & 0 \\ \mathsf{Node}_3 : - & 1 \end{pmatrix}$$

$$C_5 = \begin{pmatrix} \mathsf{Node}_0 : + & 0 \\ \mathsf{Node}_1 : + & 1 \\ \mathsf{Node}_2 : - & 0 \\ \mathsf{Node}_3 : + & 0 \end{pmatrix} C_6 = \begin{pmatrix} \mathsf{Node}_0 : + & 0 \\ \mathsf{Node}_1 : - & 1 \\ \mathsf{Node}_2 : + & 1 \\ \mathsf{Node}_3 : + & 0 \end{pmatrix}$$

$$C_7 = \begin{pmatrix} \mathsf{Node}_0 : - & 0 \\ \mathsf{Node}_1 : + & 0 \\ \mathsf{Node}_2 : + & 1 \\ \mathsf{Node}_3 : + & 0 \end{pmatrix} C_8 = \begin{pmatrix} \mathsf{Node}_0 : + & 1 \\ \mathsf{Node}_1 : + & 0 \\ \mathsf{Node}_2 : + & 1 \\ \mathsf{Node}_3 : - & 0 \end{pmatrix}$$

According to the rewriting rules of the protocol, the transitions enabled in each configuration are:

C_1: rule1(3,0), rule2(0,1), rule2(1,2)
C_2: rule1(3,0), rule2(0,1), rule2(2,3)
C_3: rule1(3,0), rule2(1,2), rule2(2,3)
C_4: rule2(0,1), rule2(1,2), rule2(2,3)
C_5: rule1(3,0), rule2(0,1), rule2(1,2)
C_6: rule1(3,0), rule2(0,1), rule2(2,3)
C_7: rule1(3,0), rule2(1,2), rule2(2,3)
C_8: rule2(0,1), rule2(1,2), rule2(2,3)

[1]For liveness properties, typically they are given as loops in executions.

We can construct a trace starting from initial configuration C_1, looping from C_1 to C_8.

$$C_1 \xrightarrow{\text{rule2(1,2)}} C_2 \xrightarrow{\text{rule2(0,1)}} C_3 \xrightarrow{\text{rule1(3,0)}} C_4 \xrightarrow{\text{rule2(2,3)}}$$

$$C_5 \xrightarrow{\text{rule2(1,2)}} C_6 \xrightarrow{\text{rule2(0,1)}} C_7 \xrightarrow{\text{rule1(3,0)}} C_8 \xrightarrow{\text{rule2(2,3)}} C_1$$

We can observe that actions enabled in these configurations are rule1(3,0), rule2(0,1), rule2(1,2) and rule2(2,3). Since each configuration occurs infinitely many times in the trace, these four actions are also enabled infinitely often. Clearly, the given trace satisfies the local fairness because these actions are actually taken infinitely often. However, there are more than one token in the infinite execution persistently, which does not meet the requirement that eventually there is only one token in the ring. [2]

Similarly, we can also show that for self-stabilizing leader election in rings [6] the local fairness is not a sufficient condition by counter-examples generated in Spin (see the Appendix).

7 Concluding Remarks

In this paper, we have reported our preliminary results on automatic verification of population protocols. We defined a weak form of fairness condition for population protocols, which is weaker than both the global and the local fairness as originally required for the population protocol model in [6]. This weak fairness can be supported by Spin. We have successfully model checked the self-stabilizing leader election in complete graphs under such a weaker fairness condition. (A formal correctness proof was also presented.) Although the global fairness is indeed necessary for the correctness of self-stabilizing token circulation in directed rings, we have still managed to model check the algorithm in Spin by a two-phase approach without explicitly encoding the global fairness. This approach follows the proof in [13]. More interestingly, counter-examples to show why local fairness is insufficient for token circulation have been automatically generated in Spin. This has improved our understanding of the population protocol model.

Directly encoding global fairness will require auxiliary variables in the model to characterize fairness situations, this will increase the complexity of the model. Furthermore, it will result in very large LTL formulas. Usually, a strong fairness condition in LTL has the following form:

```
<> [ ] enabled A => [ ] <> A
```

in which A stands for an activity in the model and enabled is the predicate specifying the condition when A can be enabled. The global fairness for population protocols requires that each *step* that can be taken infinitely often is actually taken infinitely often. Since one step can be enabled in many different configurations, this will require the enabled predicate to encode all such configurations when the step A can be enabled. Moreover, there are many more different steps in a population protocol. All these will end into a very large LTL formula. Spin cannot deal with large LTL formulas. The size of the formulas will also increase exponentially, when the number of nodes in the network increases. [3] Therefore, even we can model the global fairness, this approach does not scale up for verifying population protocols.

Our study in this paper gives rise to two interesting open questions: (1) Possibly we can find a fairness condition which is weaker than the global fairness but still strong enough to guarantee the correctness of population protocols. For such a fairness condition, hopefully the available model checkers can deal with. (2) For population protocols, we do not encounter the usual state explosion problem as in many other model checking exercises, since a node only has few states in the population protocol model. We in fact need efficient model checking algorithms to deal with large LTL formulas. The work reported in [9] is closely related, but it still cannot be applied to population protocols as we have checked in the paper. Our experience [7] of verification of self-stabilizing distributed algorithms in PVS suggests the possibility of using a theorem prover to check population protocols under global fairness, which is currently under our investigation.

Acknowledgments Jun Pang wants to thank Rena Bakhshi and Wan Fokkink for bringing his attention to the paper [6] on self-stabilizing population protocols. Yuxin Deng would like to thank Jing Cao for interesting discussions on the Spin model checker.

References

[1] D. Angluin, J. Aspnes, M. J. Fischer, and H. Jiang. Self-stabilizing population protocols. In *Proc. 9th Conference on Principles of Distributed Systems*, volume 3974 of *LNCS*, pages 103–117. Springer, 2005.

[2] J. Aspnes and E. Ruppert. An introduction to population protocols. *Bulletin of the European Association for Theoretical Computer Science, Distributed Computing Column*, 2007.

[2] In the trace, we ignore the inputs for each node, since we fix the node 0 as the unique leader. This information is stored in the array leader[i].

[3] The number of configurations will increase exponentially.

[3] P. C. Attie, N. Francez, and O. Grumberg. Fairness and hyperfairness in multi-party interactions. *Distributed Computing*, 6:245–254, 1993.
[4] T. D. Chandra and S. Toueg. Unreliable failure detector for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.
[5] F. Corradini, M. Di Berardini, and W. Vogler. Checking a mutex algorithm in a process algebra with fairness. In *Proc. 15th Conference on Concurrency Theory*, volume 4137 of *LNCS*, pages 142–157. Springer, 2006.
[6] M. J. Fischer and H. Jiang. Self-stabilizing leader election in networks of finite-state anonymous agents. In *Proc. 10th Conference on Principles of Distributed Systems*, volume 4305 of *LNCS*, pages 395–409. Springer, 2006.
[7] W. J. Fokkink, J.-H. Hoepman, and J. Pang. A note on K-state self-stabilization in a ring with $K{=}N$. *Nordic Journal of Computing*, 12(1):18–26, 2005.
[8] R. Fuzzati, M. Merro, and U. Nestmann. Distributed consensus, revisited. *Acta Informatica*, 44(6):377–425, 2007.
[9] M. Hammer, A. Knapp, and S. Merz. Truly on-the-fly LTL model checking. In *Proc. 11th Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 191–205. Springer, 2005.
[10] G. J. Holzmann. *On-The-Fly, LTL Model Checking with Spin*. `http://spinroot.com`.
[11] G. J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
[12] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
[13] H. Jiang. *Distributed Systems of Simple Interacting Agents*. PhD thesis, Yale University, 2007.
[14] L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.
[15] Z. Luo, J. Pang, and Y. Deng. *Promela source codes of self-stabilizing population protocols*. `http://basics.sjtu.edu.cn/~zhengqin/population`.
[16] H. Völzer. On conspiracies and hyperfairness in distributed computing. In *Proc. 19th Conference on Distributed Computing*, volume 3724 of *LNCS*, pages 33–47. Springer, 2005.
[17] H. Völzer, D. Varacca, and E. Kindler. Defining fairness. In *Proc. 14th Conference on Concurrency Theory*, volume 3653 of *LNCS*, pages 458–472. Springer, 2005.

Appendix: counter-examples for leader election in rings under local fairness

We have modeled the algorithm for self-stabilizing leader election in rings from [6], which is more complicated than the one for complete graphs. An eventually correct leader detector is also needed. In this algorithm, each node has three types of memory slots for tokens: a bullet slot (B), a leader mark slot (L), and a shield slot (S). $(-)$ represents an empty slot, and a full slot is denoted by its token. The order of slots in each node is (bullet, leader, shield). The leader detector gives each node an input true (T) or false (F) to indicate that whether there is a leader in the network. The algorithm is described by the following rules.

Rule 1. $((*\ *\ *, F), (*\ *\ *,\ *)) \to ((B\ L\ S), (*\ *\ *))$
Rule 2. $((*\ -S, T), (*\ *\ *,\ *)) \to ((*\ --), (-*\ S))$
Rule 3. $((*\ L\ S, T), (*\ *\ *,\ *)) \to ((B\ L\ -), (-*\ S))$
Rule 4. $((*\ L\ -, T), (-*\ *,\ *)) \to ((B\ L\ -), (-*\ *))$
Rule 5. $((*\ *-, T), (B*\ *,\ *)) \to ((B\ --), (-*\ *))$

Each node outputs its own leader slot.

Figure 6. Algorithm for self-stabilizing leader election in rings

It has been shown that leader election in rings does not work under the local fairness condition [6]. We used a model similar to the one in Section 4 and verified it in Spin. We have found several counter-examples which indicate that the algorithm does not work properly under the local fairness. Here we presented one of them. First, we present three configurations for a network with three nodes, denoted by C_1, C_2 and C_3:

$$C_1 = \begin{pmatrix} \mathsf{Node}_0 : B & L & - \\ \mathsf{Node}_1 : - & - & S \\ \mathsf{Node}_2 : - & L & S \end{pmatrix} C_2 = \begin{pmatrix} \mathsf{Node}_0 : - & L & S \\ \mathsf{Node}_1 : - & - & S \\ \mathsf{Node}_2 : B & L & - \end{pmatrix}$$

$$C_3 = \begin{pmatrix} \mathsf{Node}_0 : - & L & S \\ \mathsf{Node}_1 : - & - & - \\ \mathsf{Node}_2 : - & L & S \end{pmatrix}$$

According Figure 6, the actions enabled in each configuration are:

C_1: rule2(1,2), rule3(2,0), rule4(0,1)
C_2: rule2(1,2), rule3(0,1), rule4(2,0)
C_3: rule3(0,1), rule3(2,0)

We can construct a trace starting from initial configuration C_1, looping from C_1 to C_3.

$$C_1 \xrightarrow{\mathsf{rule4(0,1)}} C_1 \xrightarrow{\mathsf{rule3(2,0)}} C_2 \xrightarrow{\mathsf{rule4(2,0)}} C_2 \xrightarrow{\mathsf{rule2(1,2)}} C_3 \xrightarrow{\mathsf{rule3(0,1)}} C_1$$

Since each configuration occurs infinitely many times in the trace, those five actions are also enabled infinitely often. Clearly, the given trace satisfies the local fairness because these actions are actually taken infinitely often. However, there are two leaders in the infinite execution persistently. (In the trace, we ignore the inputs T for each node.)