Size-Independent Self-Stabilizing Asynchronous Phase Synchronization in General Graphs*

Chi-Hung Tzeng†, Jehn-Ruey Jiang‡ AND Shing-Tsaan Huang‡
†*Department of Computer Science*
National Tsing Hua University, Hsinchu, Taiwan, 30013
‡*Department of Computer Science and Information Engineering*
National Central University, Chungli, Taiwan 32054.

Abstract

In this paper, we design a self-stabilizing phase synchronizer for distributed systems. The synchronizer enables a node transfer from one phase to the next one, subject to the condition that at most two consecutive phases can appear among all nodes. It does not rely on any system parameter like the number of nodes, and is thus fit for the system with dynamically changing number of nodes. Each node just maintains a few variables that are related to its neighborhood; all operations are decided based on local information rather than global information. The memory usage of the proposed algorithm is low; each node has only $O(\Delta K)$ states, where Δ is the maximum degree of nodes and $K > 1$ is the number of phases. To the best of our knowledge, there are no other such size-independent self-stabilizing algorithms for systems of general graph topology.

Keywords: General connected graph, Fault tolerance, Phase synchronization, Self-stabilization, Spanning tree

1. Introduction

A distributed system may become disordered due to unexpected transient faults, such as the corruptions of node states or joining/leaving of nodes. We can make it resilient to such faults by the concept of *self-stabilization*, which has two criteria: (1) *Convergence*: Starting from any initial configuration (possibly illegal), the system can converge to a legal one in finite time. (2) *Closure*: Once the system is in a legal configuration, it remains so henceforth [6]. When a self-stabilizing system encounters transient faults, it can be thought as in an arbitrary initial configuration. With the convergence property, it can reach a legal configuration; with the closure property, it can then function correctly henceforth.

The problems of synchronizing phases of nodes in self-stabilizing distributed systems are important. Such problems can be classified into three categories: clock synchronization, neighborhood synchronization, and phase synchronization. The *clock synchronization*

*This is an extended version of the paper entitled "Self-Stabilizing Asynchronous Phase Synchronization in General Graphs," presented in the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006), 2006.

requires that all nodes eventually have the same phase (or clock) value and concurrently advance the value by one per execution step [2, 7, 11, 12]. Clearly, clock synchronization algorithms work only under the fully synchronous execution model, in which every node makes a move in an execution step. The *neighborhood synchronization* requires that the phases of two neighboring nodes be different by at most one [4, 5, 10]. In this paper, we focus on *phase synchronization* as defined in [16–18] and is also known as *barrier synchronization*. A system node is assumed to be in one of circularly arranged phases: phase 0, phase 1,..., phase $K-1$, where K is the number of phases. The relation between any two nodes' phases must satisfy the following criteria:

Definition 1. (*Phase Synchronization*)

- *No node can proceed to phase* $k+1 \pmod K$ *until all nodes are in phase* k.
- *When all nodes are in phase* k, *each node eventually proceeds to phase* $k+1 \pmod K$.

One of the application of phase synchronization is to build a synchronous environment out of an asynchronous one. For example, algorithms such as *alternator* [10] require nodes to take its move concurrently and coherently. Yet some nodes may have more computing power; over time, they may take much more tasks than required if they run in an asynchronous environment. That would cause the algorithm to fail to meet its design purpose but such a problem can be avoided by introducing a phase synchronizer that synchronizes nodes.

There are many self-stabilizing phase synchronization algorithms proposed in the literature [1, 13, 14, 16, 17]. The algorithm in [17] is *uniform*; that is, it assumes that all system nodes have no distinguishable identification and have the same behavior. It is designed for uniform complete graphs and it demands a node to proceed to a proper phase by examine all others' phases. The algorithm in [16] is based on the idea of token circulation. It is designed for non-uniform rings, in which a node is identified as a special node. When receiving a token, the unique special node increments the phase number and forwards the token; on receiving the token, any other node proceeds to the new phase and then forwards the token. The algorithm in [1] is for rooted trees. It classifies nodes into the root node, internal nodes and leaf nodes. The root starts a new phase whenever it detects the end of the last phase, whereas any other node just copies that phase. The algorithm in [13] is for uniform rings of odd size. It also uses token circulation to carry out the synchronizer: a node receiving a token copies the sender's phase and increments the token's counter by one. When the counter value is equal to the number of nodes in the system, the token owner resets the counter, then proceeds to the next phase and sends out the token. The

algorithm in [14] is for uniform rings of any size. It views a ring as a set of segments of different phases and tries to make the number of segments decrease to one to achieve phase synchronization.

The proposed algorithm is designed for general connected graphs. It is semi-uniform; i.e., all system nodes, except a special node, have identical behavior. It does not rely on any system parameter like the number of nodes, and is thus fit for the systems with dynamically changing number of nodes. Its basic idea is to use the token circulation concept to achieve spanning tree construction and phase synchronization simultaneously. It is not just a combination [3] of a tree construction algorithm, such as those in [9], and a phase synchronization algorithm for trees, such as that in [1]. Unlike the algorithms in [13,14,16,17] relying the knowledge of the number of system nodes, the proposed algorithm is independent of any system parameters. It is thus fit for the systems with dynamically changing number of nodes. It is more flexible than (or as flexible as) related ones. The number K of phases can be any value larger than one for the proposed algorithm. However, it holds that $K = n \geq 3$, $K \geq n$, $K = 2$, K is even and $K \geq n-1$, where n is the number of system nodes, for the algorithms in [17], [16], [1], [13], and [14], respectively. It is also memory-efficient; its space complexity is $O(\Delta K)$ states, where Δ is the maximum degree of nodes. Its space complexity is the same as that of the algorithm in [1] and is better than $O(nK)$ of the algorithms in [13, 14]. To the best of our knowledge, there are no other such size-independent self-stabilizing algorithms for systems of general connected graph topologies.

We use a set of rules to describe our algorithm. The proposed algorithm can operate correctly in the parallel execution model, in which a distributed daemon selects an arbitrary subset of nodes to execute the rules in each execution step. That is a more general model than the serial execution model adopted by the algorithm in [13], in which a central daemon [6] randomly selects a node to execute exactly one rule in each execution step.

The rest of the paper is organized as follows. Section 2 presents the system model and some terms used throughout this paper. Section 3 shows the proposed algorithm. Its correctness proofs and time complexity are analyzed in section 4. Finally, section 5 concludes this paper, extending the idea proposed in this paper and showing some future work.

2. The System Model

We model the system by a connected undirected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges representing the communication links. Two nodes i and j are said to be neighbors if $(i, j) \in E$. Each node keeps a set of variables, to which it can write its own state and from which it can read the neighbors' states. Throughout this paper, we use the notation $VAR.i$ to denote the variable VAR maintained by node i.

The behavior of a node is defined by a set of rules of the form "$guard \rightarrow action$", where *guard* is a boolean formula and $action$ is a set of program statements about how to update the values of the variables. Once the guard of one rule is true for a node, we say that the node is *privileged* and the rule is *enabled*. The privileged node can execute the action of the enabled rule; we say that it executes a rule. A rule is executed in an atomic way: evaluating the guard and executing the corresponding action are done uninterruptedly. In this paper, we assume that the system is *semi-uniform*: each node except the special node r has the same set of rules.

We use the term $configuration$ to refer to a vector of all nodes' states for representing the system status. Given a configuration c and its successor c', the transition from c to c' is called a *computation step*, denoted by $c \rightarrow c'$. During $c \rightarrow c'$, one or more privileged nodes in the configuration c concurrently execute rules and each of them executes exactly one rule. After that, the system enters the configuration c' and the next computation step begins. In this paper, we assume a *parallel execution model*: There is a *daemon* selecting an arbitrary non-empty subset of privileged nodes to execute rules during every computation step.

The computation of the system can be expressed by a series of configurations $(c_0, c_1, \ldots)$, where c_0 is an arbitrary initial configuration and each $c_k \rightarrow c_{k+1}$ is a computation step. We use $c_k \rightsquigarrow c_{k+m}$ to denote m consecutive computation steps, where $m > 0$ and $k \geq 0$. Given a configuration, its successor may not be unique, depending on how the daemon selects privileged nodes. A self-stabilizing system must guarantee that it eventually reaches a legal configuration c_ℓ from any possible initial configuration c_0; that is, $c_0 \rightsquigarrow c_\ell$, where ℓ is a finite integer. This requirement is called *convergence*. Another requirement of self-stabilization is called *closure*: Given a legal configuration, its successor is also legal.

For the sake of simplicity, we use *round* instead of computation step to explain how the system converges to a legal configuration. Starting from a configuration c_k, a round is the least sequence of consecutive computation steps $c_k \rightsquigarrow c_{k+m}$ such that every privileged node in c_k has executed one or more rules when the system reaches c_{k+m}. The first round starts from c_0, and its ending configuration is the beginning of the second round, ..., and so on. In this paper, the time complexity is the number of rounds converging to the first legal configuration in the worst case.

3. The Phase Synchronization Algorithm

In this section, we develop a phase synchronization algorithm for semi-uniform systems under the parallel execution model. Our idea is to construct a spanning tree rooted at the special node r. The node r is responsible for starting a new phase when it detects the end of the last phase. Any other node simply copies the phase of its parent; thus the new

Variables:
P: Parent pointer
$D \in \{F, B\}$
$C \in \{0, 1, 2\}$

For the root node r, $P.r = r$ and $D.r = F$.
$\mathbb{R}0$: $(\forall j \in Child.r : D.j = B \wedge C.j = C.r) \rightarrow C.r = C.r + 1$;

For $i \neq r$:
$\mathbb{R}1$: $(D.i = F) \wedge (\forall j \in Child.i : D.j = B \wedge C.j = C.i) \rightarrow D.i = B$;
$\mathbb{R}2$: $(D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i) \rightarrow D.i = F; C.i = C.P.i$;

Figure 1: The token circulation for a static tree rooted at r

phase is propagated in a top-to-down manner and eventually all nodes proceed to the new phase.

To realize the above idea, we define a conceptual object called token circulating along only tree edges. (An edge is called tree edge if one of its ends is the other's parent.) There are two types of tokens: *forward tokens* and *backward tokens*. Forward tokens travel the tree from the root to leaf nodes, while backward tokens travel from leaf nodes to the root. During traveling, forward tokens help (1) propagate the current phase, and (2) construct the spanning tree. On the other hand, backward tokens simply help the root node know when to start a new phase; they are irrelevant to tree construction. In short, phase synchronization is done in tandem with spanning tree construction. In the initial configuration, the phase values may violate the requirements of Definition 1 and the tree edges may induce a forest rather than a spanning tree. As the algorithm runs over time, the system converges to legitimate configurations and we have consistent phase values together with a spanning tree

The proposed algorithm is developed on the basis of *token circulation* mechanism [15]. In [15], the number of tokens can be more than one, but adjacent nodes never hold tokens at the same time. That is, nodes holding tokens form an *independent set* and thus [15] solves *local mutual exclusion problem*. Here we extend idea of [15] and design a phase synchronizer for general connected graphs. For the sake of presentation, we rephrase [15] into three rules shown in Fig. 1. In the figure, $P.i$ is the parent of node i and $Child.i = \{j | P.j = i\}$ stands for the set of i's children nodes. In addition to the pointer variable P, every node keeps two scalar variables D and C. The variable D stands for the token's direction and its value is either B (Backward) or F (Forward). The variable C stands for the node's color and its value is 0, 1, or 2. Throughout this paper, the arithmetic operations on C are assumed to be under modulo 3 and such predicates as $(\forall j \in Child.i : D.j = B \wedge C.j = C.i)$ are assumed to be true if $Child.i = \emptyset$. A token is assumed to have the same color with its owner. A non-r node i is said to hold a forward token if $(D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i)$, and it is said to hold a backward token from its child j if

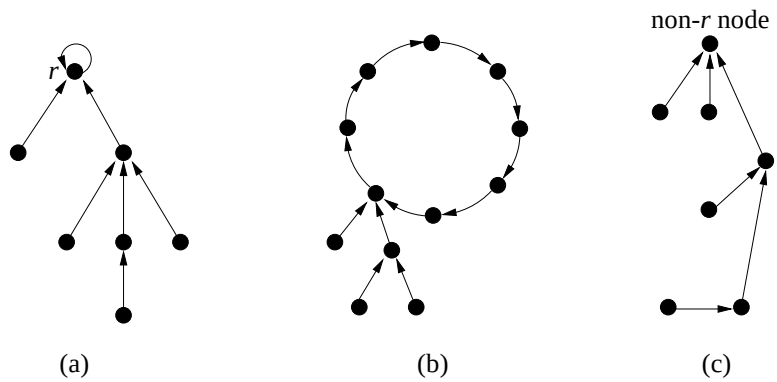

Figure 2: An example of connected components: (a) R-tree, (b) O-tree, and (c) Nil-tree.

$(D.i = F) \wedge (D.j = B) \wedge (C.i = C.j)$. For the root node r, the definition of holding a backward token is the same as that of a non-r node. However, the node r is assumed to hold a forward token once it holds backward tokens from all its children.

Consider the following *perfect state*: $\forall i \neq r : (C.i = C.r) \wedge (D.i = B) \wedge (D.r = F)$, in which only r has a forward token. From the perfect state, tokens circulate the tree as follows: By executing $\mathbb{R}0$, the root node r changes its color to $C.r + 1$, and propagates a forward token with this color to each of its children. When a non-r node receives a forward token, it executes $\mathbb{R}2$ to copy the parent's color and passes one forward token with the new color to each of its children. If this node is a leaf, it then executes R1. The forward token thus becomes a backward token and travels back to the root. When a non-r node receives backward tokens from all its children, it merges those tokens into one and passes the backward token to its parent by executing $\mathbb{R}1$. Once the node r receives backward tokens from all its children, a period of token circulation is assumed to be finished and the system enters another perfect state. Afterwards, the root node will start a new token circulation. Please note that during the time of the token circulation, the colors of nodes are either $C.r$ or $C.r - 1$. Therefore, the color variable C can be viewed as a kind of phase variable and Fig. 1 is actually a 3-phase synchronizer for static tree networks.

Because the network topology that we consider is a general connected graph instead of a tree, we have to further improve the algorithm in Fig. 1. First, the parent pointers are no longer constant variables. Therefore, we define that the root node r always points to itself and that any other node points to one of its neighbors or nil. By this setting, the system has three kinds of connected components induced by tree edges: *R-tree*, *O-tree* and *Nil-tree*. The R-tree is the tree rooted at the node r; an O-tree contains a cycle and branches pointing to the cycle; a Nil-tree is a tree rooted at a node pointing to nil. A Nil-tree of single node is called an *isolated node*. Fig. 2 gives an example of the three components,

When we apply the algorithm in Fig. 1 to a tree network, such as the R-tree, the system eventually reaches the perfect state from any arbitrary initial state and then tokens circulate the system infinitely often. However, when we apply the mechanism to an O-tree or a Nil-tree, there will be no token eventually. This is because there is no root node r generating and propagating tokens in O-trees/Nil-trees. In terms of phases, nodes in the R-tree keep changing their phases, whereas no node in O-trees/Nil-trees can change its phase. As will be shown later, this asymmetric property is useful to determine whether a node is in the R-tree.

Below, we start to develop the rules for our phase synchronizer. The basic idea is to break O-trees to be Nil-trees and then to be isolated nodes, and isolated nodes then join the R-tree and calibrate its phase value. Our solution requires a node to know whether each of its neighbors has a forward token or not, so the range of the variable D is extended to be $\{FT,F\}$ for the root node r and to be $\{FT,F,B\}$ for every non-r node. When $D.i = FT$ holds, it means that i holds a token of the direction "Forward". A node receiving a forward token first sets $D = FT$ and renews its color. Afterwards, the node sets $D = F$ and the token is sent out. Due to this setting, $\mathbb{R}0$ is divided into two rules (a) and (b):

(a) $(D.r = F) \wedge (\forall j \in Child.r : D.j = B \wedge C.j = C.r) \rightarrow D.r = FT; C.r = C.r + 1;$
(b) $(D.r = FT) \rightarrow D.r = F;$

Since the variable P of a non-r node i may point to nil, rule $\mathbb{R}1$ becomes rule (c) by adding the condition $(P.i \neq nil)$ to the guard. On the other hand, $\mathbb{R}2$ becomes two rules (d) and (e) as $\mathbb{R}0$ does.

(c) $(P.i \neq nil) \wedge (D.i = F) \wedge (\forall j \in Child.i : D.j = B \wedge C.j = C.i) \rightarrow D.i = B;$
(d) $(P.i \neq nil) \wedge (D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i) \rightarrow D.i = FT; C.i = C.P.i;$
(e) $(D.i = FT) \rightarrow D.i = F;$

Now, we explain how to use forward tokens to break O-trees. As we mentioned above, during the token circulation with color $C.r$ in the R-tree, the colors of nodes should be either $C.r$ or $C.r - 1$. Let i be a node in an O-tree or Nil-tree. Once node i detects that one of its neighbors j holds a forward token of color $C.i + 2$, it is aware that j is in the R-tree but it is not. Rather than pointing to j directly, node i points to nil first in order not to disturb the token circulation in the R-tree because the O-tree/Nil-tree where i lies may contain inconsistent colors with respect to the current token circulation, After setting $P.i = nil$, the O-tree/Nil-tree is broken. Thus we have the following rule (f), in which $N.i = \{j | (i, j) \in E\}$ is the set of i's neighbors.

(f) $(P.i \neq nil) \wedge (\exists j \in N.i : D.j = FT \wedge C.j = C.i + 2) \rightarrow P.i = nil;$

By rule (f), an O-tree is broken to be a Nil-tree. The next step is to break the Nil-tree to

be isolated nodes. The idea is to make Nil-tree collapse in a top-to-down manner, as rule (g) shows.

(g) $(P.i \neq nil) \wedge (P.P.i = nil) \rightarrow P.i = nil$;

The last step is to make isolated nodes join the R-tree with the help of forward tokens:

(h) $(P.i = nil) \wedge (Child.i = \emptyset) \wedge (\exists j \in N.i : D.j = FT \wedge P.j \neq nil) \rightarrow P.i = j$;

Below, we discuss the issues caused by an adversary daemon. Let $j \in N.r$ be a node not in the R-tree. When j evaluates the guard of rule (f) true, r must also evaluate the guard of rule (b) true at the same time. If r takes a move earlier than j does, j's privilege vanishes. An adversary daemon can make this always happen to prevent j from executing rule (f) and thus from joining the R-tree. Therefore, we must modify (b) to demand node r to wait until j takes a move.

(b*) $(D.r = FT) \wedge (\forall j \in N.r : C.j \neq C.r + 1) \rightarrow D.r = F$;

Similarly, rule (e) should be modified to be (e*):

(e*) $(D.i = FT) \wedge (\forall j \in N.i : C.j \neq C.i + 1) \rightarrow D.i = F$;

A Nil-tree root node i with $(D.i = FT)$ should reset $D.i$ unconditionally. Thus rule (e*) is modified to be rule (e**):

(e**) $(D.i = FT) \wedge ((\forall j \in N.i : C.j \neq C.i + 1) \vee (P.i = nil)) \rightarrow D.i = F$;

The last issue is to guarantee no disturbance in token circulation even when isolated nodes join the R-tree. To do so, a node i setting $P.i = j$ has to set $D.i = B$ and $C.i = C.j$ as well, as if i has already received a token of color $C.j$. Based on this reason, rule (h) is further modified into (h*) as follows.

(h*) $(P.i = nil) \wedge (Child.i = \emptyset) \wedge (\exists j \in N.i : D.j = FT \wedge P.j \neq nil) \rightarrow P.i = j; D.i = B; C.i = C.j$;

The above rules are sufficient to build a spanning tree as well as a 3-phase synchronizer by the variable C. To extend the rules to be a K-phase synchronizer, $K > 1$, each node maintain another variable $H \in \{0, 1, ..., K-1\}$ to denote its phase. We add $H.r = H.r + 1 \mod K$ to the action of rule (a) and add $H.i = H.P.i$ to that of rules (d) and (h*). The guards of all the rules remain unchanged. That is, a node updates its phase variable H whenever it changes its color. Therefore, the system satisfies definition 1 right after the spanning tree is constructed.

All the rules mentioned above constitute our algorithm, which is listed in Fig. 3. The root node r has two rules (a) and (b*), corresponding to R0 and R1 respectively. For non-r nodes, the rules (f) and (g) are combined into one rule R5, so it has five rules: R2 to R6.

Variables:
P: parent pointer
$C \in \{0,1,2\}$ // for denoting the color
$D \in \{FT,F,B\}$ // for denoting the direction
$H \in \{0,1,..,K-1\}$ // for denoting the phase

For the root node r: $P.r = r$ and $D.r \in \{FT,F\}$
R0: $(D.r = F) \wedge (\forall j \in Child.r : D.j = B \wedge C.j = C.r) \rightarrow D.r = FT; C.r = C.r + 1; H.r = H.r + 1;$
R1: $(D.r = FT) \wedge (\forall j \in N.r : C.j \neq C.r + 1) \rightarrow D.r = F;$

For $i \neq r$:
R2: $(P.i \neq nil) \wedge (D.i = F) \wedge (\forall j \in Child.i : D.j = B \wedge C.j = C.i) \rightarrow D.i = B;$
R3: $(P.i \neq nil) \wedge (D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i) \rightarrow D.i = FT; C.i = C.P.i; H.i = H.P.i;$
R4: $(D.i = FT) \wedge ((\forall j \in N.i : C.j \neq C.i + 1) \vee (P.i = nil)) \rightarrow D.i = F;$
R5: $(P.i \neq nil) \wedge ((\exists j \in N.i : D.j = FT \wedge C.j = C.i + 2) \vee (P.P.i = nil)) \rightarrow P.i = nil;$
R6: $(P.i = nil) \wedge (Child.i = \emptyset) \wedge (\exists j \in N.i : D.j = FT \wedge P.j \neq nil) \rightarrow P.i = j; D.i = B; C.i = C.j; H.i = H.j;$

Figure 3: The proposed algorithm.

We assume that each rule has a priority and a rule with a smaller number has a higher priority: When a privileged node is selected by the daemon to make a move, it executes the highest-priority enabled rule. As readers can check, the memory usage is low and is independent of any system parameter. Each node keeps only a pointer variable P, a phase variable H, and two scalar variables of totally 6 (resp., 9) states for the node r (resp., for a non-r node.) Let Δ denote the maximum degree of the graph; the space complexity of the variable P is $O(\Delta)$. When taking all the variables H, D, C, and P into consideration, the space complexity per node is $O(\Delta K)$ states.

4. Correctness and time complexity analysis

In this subsection, we show that the system stabilizes in $O(n^2)$ rounds, regardless of any arbitrary initial configuration, where $n = |V|$. We first define the legal configuration as below.

Definition 2. $(Legal\ Configuration)$
A configuration is legal if it satisfies the following three conditions:
(1) $\forall i \neq r : (C.i = C.r) \wedge (D.i = B)$.
(2) $D.r = F$.
(3) *the number of nodes in the R-tree is n.*
Furthermore, any configuration that arises from the one satisfying (1), (2) *and* (3) *by the completion of one or more moves is also a legal configuration.*

Before showing that the system eventually reaches a legal configuration, we must guarantee that at least one node is privileged for any arbitrary configuration. In other words, the system is never deadlocked.

Lemma 1. *For any configuration, at least one node is privileged.*

Proof. We prove this lemma by contradiction. We shall focus on only the R-tree. Assume that no node is privileged. According to the value of the variable D, there are two cases to be considered:
Case (1) every node has either $D = B$ or $D = F$:

Let $h(i) \geq 1$ be the height of a node i in the R-tree. We first use induction on $h(i)$ to show that every non-r node i in the R-tree has $D.i = B$. (Basis) Since a leaf node i ($h(i) = 1$) cannot execute R2, we have $D.i = B$. (Hypothesis) $D.j = B$ for any non-r node j with $h(j) < \lambda$. (Induction) Let i be a non-r node with $h(i) = \lambda$ and j be a child of i. We have $D.j = B$ by the hypothesis. Assume that $D.i = F$. If $C.j \neq C.i$ for some node j, then j can execute R3. If $C.j = C.i$ for any node j, node i can execute R2. Since no rule is enabled, the case of $D.i = F$ does not occur and we have $D.i = B$, as desired.

For the root node r, $D.r = F$ holds because the range of $D.r$ doesn't contain B. Since r cannot execute R0, it has a child j such that $C.j \neq C.r$, by which the node j can execute R3 due to $(P.j \neq nil) \wedge (D.j = B) \wedge (D.P.j = F) \wedge (C.j \neq C.P.j)$. Contradiction occurs.
Case (2) some node i has $D.i = FT$:

Because node i cannot execute R4, we have $P.i \neq nil$ and i has a neighbor j with $C.j = C.i + 1$ (node j may not be in the R-tree). We have two sub-cases to consider: (i) $P.j \neq nil$: j can execute R5 because $(D.i = FT) \wedge (C.i = C.j - 1 = C.j + 2)$. (ii) $P.j = nil$: Since j cannot execute R6, we have $Child.j \neq \emptyset$. Node j's child k can execute R5 because $(P.k \neq nil) \wedge (P.P.k = P.j = nil)$. We get a contradiction for each sub-case. □

Below, we begin to prove the convergence property. We first define tokens as follows. In a legitimate state, there is exactly one token in every path from the node r to the leaves.

Definition 3. (*Forward Token*)
The root node r is said to own a forward token iff

$(D.r = F \wedge (\forall j \in Child.r : D.j = B \wedge C.r = C.j)) \vee (D.r = FT)$,

whereas a non-r node is said to hold a forward token iff

$(P.i \neq nil) \wedge ((D.i = B \wedge D.P.i = F \wedge C.i \neq C.P.i) \vee D.i = FT)$.

In the above definition, the condition $(D.i = B \wedge D.P.i = F \wedge C.i \neq C.P.i)$ for a non-r node i means that $D.i$ has passed a forward token to i (by executing R4) but i has not yet executed R3.

Definition 4. (*Backward Token*)
A non-leaf node i is said to receive and hold a backward token from a child j iff

$(D.i = F) \wedge (D.j = B) \wedge (C.i = C.j)$,

whereas a non-isolated leaf node i is said to hold a backward token iff

$(D.i = F)$.

Our ultimate goal is to show that eventually there is a fixed spanning tree. To do this, we first prove some properties of tokens in lemmas 2, 3, 4, and 5. By these lemmas, we can infer the property of tokens in O-trees, Nil-trees, and the R-tree, as shown in lemmas 6, 7 and 8, respectively. Finally, lemmas 9 and 10 show how a system converges to the legal configuration.

Lemma 2. *In fixed components(R-tree, O-trees, or Nil-trees), a non-r node does not receive consecutive forward and consecutive backward tokens.*

Proof. We first show that a non-r node i never receives consecutive forward tokens. By definition, when node i owns a forward token, either $D.i = B$ or $D.i = FT$ holds and it can execute R3 and then R4 (if $D.i = B$), or simply execute R4 (if $D.i = FT$) to pass the token to its children. After the token passing, its status is $D.i = F$. Before owning a forward token again, node i has to execute R2 so that the value of $D.i$ becomes B from F. Since the guard of R2 implies the possession of (at least) a backward token, it means that node i must own (at least) a backward token before getting a forward token again. That is, node i never receives consecutive forward tokens.

Based on the same strategy, we can also prove that node i never receives consecutive backward tokens, so this proof is skipped. □

Lemma 3. *Once a forward token meets a backward token, one of them disappears.*

Proof. Consider two nodes i and j, $P.j = i$ and $P.i \neq nil$, such that j can execute R2 and i can execute R0 (if $i = r$) or R3 (if $i \neq r$). By definition, node j has a backward token, whereas node i has a forward token. We prove this lemma by checking how many tokens are left after the node pass the tokens.

Because the tokens meet by node i or node j or both executing the rules, we have the following three cases to consider:

Case (1) Only j passes the backward token by executing R2:

For this case, node j is of $D.j = B$ so the backward token disappears. On the other hand, node i still holds the forward token.

Case (2) Only i passes the forward token by executing R0 and R1 (or R3 and R4):

For this case, node i is of $D.i = F$ so the forward token disappears. On the other hand, node j still holds the backward token.

Case (3) Both i and j pass tokens:

After j executes R2 and i executes R0 and then R1 (or R3 and then R4), we have $D.j = B$, $D.i = F$ and $C.i = C.P.i$. According to the relation between $C.i$ and $C.j$, we have two sub-cases to consider: (i) $C.i = C.j$: Node i has $(D.j = B) \wedge (D.i = F) \wedge (C.i = C.j)$, so it has a backward token coming from j. On the other hand, node j does not own the

forward token. (ii) $C.i \neq C.j$: Here, node j has $(D.j = B) \wedge (D.P.j = F) \wedge (C.j \neq C.P.j)$ so it has a forward token. On the other hand, node i does not receive the backward token coming from j.

Because either the forward token or the backward token disappears in each case, this lemma holds. □

For normal token circulation, tokens bounce between the root node r and leaf nodes. That is, a backward token should become a forward token when it arrives at the node r, whereas a forward token should become a backward token when it arrives at a leaf node. However, in the beginning a token may change its direction at an internal node because of the unpredictable initial configuration. And we say that an internal node i performs an *illegal forward (resp., backward) token reversal* if it receives a forward (resp., backward) token but sends out a backward (resp., forward) token. In terms of rules, if node i perform an illegal forward token reversal, it executes R3, R4, and R2 consecutively. (Note that node i can execute R2 right after executing R4, but in normal situations it cannot do so.) Similarly, if node i perform an illegal backward token reversal, it executes R2, R3 and R4 consecutively.

Lemma 4. *Eventually no internal node can perform illegal forward or backward token reversal.*

Proof. To prove this lemma, we show that an internal node i can perform at most once illegal reversal of a forward token and of a backward token respectively.

Consider the case that node i performing an illegal forward token reversal. By definition, it executes R3, R4 and R2 consecutively. After executing the three rules, node i has $(D.i = B) \wedge (\forall j \in Child.i : C.j = C.i \wedge D.j = B)$. Let j be a child of i. The next time i receives a forward token, node i has $C.i \neq C.P.i$. After i executes R3 and R4 to pass the forward token, the condition $C.j \neq C.i$ holds so it cannot execute R2 immediately. That is, node i cannot reverse the forward token.

Now, consider the case that node i performs an illegal backward token reversal. Similarly, it means that node i executes R2, R3 and R4 consecutively. After executing the three rules, node i has $(D.i = F) \wedge (D.P.i = F) \wedge (C.i = C.P.i)$. The next time i owns a backward token and executes R2, it cannot execute R3 immediately because $C.i = C.P.i$ holds. That is, node i cannot reverse the backward token. □

Informally speaking, after a node executes R2 (resp., R3 and R4), its state is consistent with its children (resp., parent). For a node that has executed R2, any of its children never performs an illegal forward token reversal. Similarly, a node that has executed R3 and R4 never performs an illegal backward token reversal. Since an illegal forward (resp.,

backward) token reversal is caused by the executions of R3, R4, and R2 (resp., R2, R3, R4), we can infer that any non-r node performs at most one illegal forward (resp., backward) token reversal.

Now, consider the longest waiting time $T(d)$ for an internal node i to receive a forward token, where d is the depth of i. Because forward tokens move along only tree edges and because they transfer from a node to the next one in a constant number of rounds, the worst case happens when every of i's ancestors, except r, executes an illegal forward token reversal. Starting from the moment when the root holds a forward token, we can deduce $T(2) = 1$ and $T(d) = T(d-1) + O(d)$, and thus $T(d) = O(d^2) = O(n^2)$ rounds. Once node i holds a forward token, it executes R3 and R4 and its children will not be able to perform an illegal backward token traversal. In other words, every internal node cannot execute an illegal backward token traversal in $O(n^2)$ rounds.

Based on a similar reason, we can infer that every internal node cannot execute an illegal forward token reversal in $O(n^2)$ rounds. We have the following lemma:

Lemma 5. *After $O(n^2)$ rounds, no internal node can perform illegal forward or backward token reversal.*

In the following three lemmas, we show the behavior of tokens in O-trees, Nil-trees, and the R-tree, respectively.

Lemma 6. *After $O(n^2)$ rounds, there will be no token in O-trees.*

Proof. To prove this lemma, we show that, for any O-tree, tokens in the branches go into the cycle in $O(n)$ rounds and then disappear in $O(n^2)$ rounds. With the help of lemma 5, we assume that no illegal token reversal would occur.

First, focus on the tokens in the O-tree branches. In such components, a forward token becomes a backward when it arrives at a leaf node and the backward token either goes into the O-tree cycle or disappears. The time complexity for this is $O(n)$ rounds, including $O(n)$ rounds for a forward token to become a backward token and another $O(n)$ rounds for the backward token to go into the cycle.

Now, consider the tokens in the O-tree cycle. According to lemma 3, the number of tokens decreases when two tokens of different directions meet; hence eventually the tokens in the cycle are of the same direction, either forward or backward. The time complexity for this is $O(n) \times O(n) = O(n^2)$ rounds because there may be $O(n)$ tokens in the cycle and two tokens of different types meet in $O(n)$ rounds. Afterwards, these survival tokens disappear in $O(n)$ rounds, since a node never consecutively receives tokens of the same direction, according to lemma 2. In summary, all the tokens in the O-tree cycle disappear in $O(n^2)$ rounds, as desired. □

Lemma 7. *After $O(n)$ rounds, there will be no token in Nil-trees.*

Proof. By definition, an isolated node has no token. Therefore, we prove this lemma by showing that any Nil-tree becomes a set of isolated nodes in $O(n)$ rounds.

According to R5, a child of a Nil-tree root can point to nil, so the Nil-tree's height decreases by one every $O(1)$ rounds. Combining the fact that the tree height is $O(n)$, the Nil-tree becomes a set of isolated nodes in $O(n)$ rounds. □

Below, we show that the R-tree eventually reaches the *perfect state*; viz. $D.r = F$ and any non-r node i in the R-tree has $(C.i = C.r) \wedge (D.i = B)$.

Lemma 8. *After $O(n^2)$ rounds, the R-tree reaches the perfect state.*

Proof. To prove this lemma, we first show that in $O(n^2)$ rounds only one token exists in each tree path from a leaf node to the root node r. Afterwards, the R-tree enters the perfect state in $O(n)$ rounds. Similar to lemma 6, we assume that no illegal token reversal would occur.

Consider a tree path from a leaf node to the root node r. By the proof of lemma 1, there is at least one token in this path. Let the number of tokens in this path be $O(n)$. Because the tokens bounce between the root node and the leaf node, they meet one another in $O(n)$ rounds. By lemma 3, it means that the number of tokens decreases by one every $O(n)$ rounds, or, equivalently, decreases to one in $O(n) \times O(n) = O(n^2)$ rounds. If the last survival token is forward, it reaches the leaf node in $O(n)$ rounds and becomes a backward token traveling back to the root node.

Now we consider the configuration in which there is exactly one backward token in any tree path from a leaf node to the root node. For a non-r node i receiving all the backward tokens from its children, it executes R2 to pass the merged backward token to its parent. After the execution, node i has $D.i = B \wedge C.j = C.i \wedge D.j = B$, where $j \in Child.i$. Since every $O(1)$ rounds a backward token moves from the node of height k to the node of height $k+1$, the root node receives backward tokens from all the children in $O(n)$ rounds. By that time, the root node is of $D.r = F$ and any non-r node i in the R-tree is of $D.i = B \wedge C.i = C.P.i = C.r$. That is, the R-tree is in the perfect state.

According to the above proof, the R-tree enters the perfect state in $O(n^2)$ rounds. □

Lemma 9. *Once the R-tree reaches the perfect state and there is no token in O-trees/Nil-trees, the number of nodes in the R-tree is monotonically increasing.*

Proof. To prove this lemma, we show that a node i in the R-tree does not execute R5 to depart from the R-tree. Let j be a neighbor of i. Our attempt is to show that $C.j \neq C.i - 2$ holds when $D.j = FT$ holds. This node j must be in the R-tree; otherwise $D.j = FT$ cannot

hold since there is no token in O-trees/Nil-trees. Below, we consider a token circulation in the R-tree, observe how the color C changes, and prove the desired property: $C.j \neq C.i-2$.

Let's consider a token circulation starting from the perfect state, in which every node has the same color $C.r = \alpha - 1$. During this token circulation, the root node r executes exactly two rules R0 and R1, and any other R-tree node executes exactly three rules R2, R3, and R4. Because a node changes its color to be α only when it executes R0 or R3, and because these two rules set $D = FT$ as well, the condition $C.j = \alpha$ must hold when $D.j = FT$ holds. For node i, its color is either $C.i = \alpha$ or $C.i = \alpha - 1$. It is easy to check that $C.j \neq C.i-2$, as desired. □

Lemma 10. *Once the R-tree reaches the perfect state and there is no token in O-trees/Nil-trees, the R-tree spans all the nodes in $O(n^2)$ rounds.*

Proof. Let i and j be two adjacent nodes such that j is in the R-tree, while i is not. We can prove this lemma by showing that node i joins the R-tree in $O(n)$ rounds.

Consider the token circulations in the R-tree. During a token circulation propagating color 0, node j is of $D.j = FT$ and $C.j = 0$ at some time by executing R0 (if $j = r$) or R3 (if $j \neq r$). Similarly, during the token circulations propagating color 1 and color 2, node j is of $(D.j = FT) \wedge (C.j = 1)$ and $(D.j = FT) \wedge (C.j = 2)$ at some point, respectively. Because node i has no token and thus cannot change $C.i$, the condition $(D.j = FT) \wedge (C.j = C.i + 2)$ holds at some time within three consecutive token circulations. Since each token circulation finishes in $O(n)$ rounds, this condition holds within $3 * O(n)$ rounds. The next step is to prove that node i points to node j within $O(1)$ rounds when this condition holds.

By the components where i locates, there are three cases to consider:

Case (1) i is an isolated node:

Node i executes R6 to set $P.i = j$ to join the R-tree.

Case (2) i is in an O-tree:

Node i executes R5 to set $P.i = nil$ and becomes a Nil-tree root. Then its children, if any, execute R5 so node i becomes an isolated node. The remaining proof of this case is similar to that of Case (1).

Case (3) i is in a Nil-tree containing more than one node:

The proof of this case is similar to that of Case (2).

The actions in all the three cases take $O(1)$ rounds, so node i becomes a part of the R-tree in $O(1)$ rounds when $(D.j = FT) \wedge (C.j = C.i + 2)$ holds. Because this condition holds in $O(n)$ rounds, the number of nodes in the R-tree increases by one every $O(n)$ rounds, until the R-tree spans all the nodes. Thus the overall time complexity is $O(n) \times O(n) = O(n^2)$ rounds. □

Theorem 1 (convergence)**.** *The system enters legal configurations in $O(n^2)$ rounds.*

Proof. This is a direct consequence of lemmas 5, 6, 7, 8, 9 and 10. □

Theorem 2 (closure)**.** *Once the system is in a legal configuration, it remains so henceforth.*

Proof. Note that the criteria (1) and (2) of definition 2 are the conditions of perfect states. By lemma 9, no node can depart from the R-tree so the number of nodes in the R-tree is always n. □

5. Concluding Remarks

We propose a size-independent self-stabilizing algorithm for the phase synchronization problem in semi-uniform systems of general connected graph topologies. The algorithm runs under the parallel execution model and does not rely on any system parameter like the number of nodes. It is thus fit for the systems with dynamically changing number of nodes. Another advantage of the algorithm is the low space complexity: $O(\Delta K)$ states per node, where K is the number of phases and Δ is the maximum degree of system nodes. Moreover, the number of phases can be any number larger than one, which makes the proposed algorithm very flexible. To the best of our knowledge, there are no other such size-independent self-stabilizing algorithms for systems of general connected graph topologies.

In our algorithm, the scheduling daemon can be unfair for the following reasoning. Suppose that a node i is continuously privileged but not selected by the daemon. It is safe to say that the phase ($C.i$ or $H.i$) of i dose not change. By definition 1, other nodes cannot advance their phases eventually, or equivalently, they cannot execute R0 or R3. Those nodes can only execute R1, R2, R4, R5, or R6, and ultimately end up with $D = B$. Because they cannot execute R3, which is the only rule to reset the variable D, the condition $D = B$ remains and their privileges vanish. Therefore, node i eventually becomes the only privileged one. The daemon then must select i to execute.

We make an assumption that there is a unique root node r. This node is responsible for coordinating other nodes so that they know when to enter a new phase. If the node r departs from the system (due to events such as power failure), then the proposed algorithm no longer works. However, the assumption of the root node in this paper is for constructing a spanning tree deterministically during the convergence of the phase synchronizer, and it may be unnecessary for a phase synchronizer. Therefore, it is challengin task to remove this assumption and develop another self-stabilizing phase synchronizer for general graphs. We will take this as a future work.

References

1. L. O. Alima, J. Beauquier, A. K. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 102–109, 1998.

2. A. Arora, S. Dolev, and M. G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.

3. A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.

4. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *ACM Symposium on Theory of Computing*, pages 652–661, 1993.

5. C. Boulinier, F. Petit, and V. Villain. Synchronous vs. asynchronous unison. In *Self-Stabilizing Systems*, pages 18–32, 2005.

6. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

7. S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, 12(1):95–107, 1997.

8. S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE transations on Parallel and Distributed Systems*, 8(4):424–440, 1997.

9. F. C. Gärtner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, Swiss Federal Institution of Technology, 2003.

10. M. G. Gouda and F. F. Haddix. The alternator. In *WSS*, pages 48–53, 1999.

11. M. G. Gouda and T. Herman. Stabilizing unison. *Information Processing Letters*, 35:171–175, 1990.

12. S. T. Huang and T. J. Liu. Self-stabilizing 2^m-clock for unidirectional rings of odd size. *Distributed Computing*, 12:41–46, 1999.

13. S. T. Huang and T. J. Liu. Phase synchronization on asynchronous uniform rings with odd size. *IEEE Transactions on Parallel and Distributed System*, 12(6):638–652, 2001.

14. S. T. Huang, T. J. Liu, and S. S. Hung. Asynchronous phase synchronization in uniform unidirectional rings. *IEEE Transactions on Parallel and Distributed System*, 15(4):378–384, 2004.

15. H. S. M. Kruijer. Self-stabilization(in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8(2):91–95, 1979.

16. S. Kulkarni and A. Arora. Fine-grain multitolerant barrier synchronization. Technical report, Technical Report OSU-CISRC TR34, Ohio State University, 1997.

17. S. Kulkarni and A. Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, 1997.

18. J. Misra. Phase synchronization. *Information Processing Letters*, 38(2):101–105, 1991.