Self-Stabilizing Depth-First Token Passing on Rooted Networks *

Colette Johnen[1] Gianluigi Alari[2] Joffroy Beauquier[1]
Ajoy K. Datta[3]

[1] L.R.I., C.N.R.S. URA 410, Université de Paris-Sud, France
[2] Unité d'Informatique, Université catholique de Louvain, Belgium
[3] Department of Computer Science, University of Nevada Las Vegas

Abstract

We present a deterministic distributed depth-first token passing protocol on a rooted network. This protocol does not use either the processor identifiers or the size of the network, but assumes the existence of a distinguished processor, called the root of the network. The protocol is self-stabilizing, meaning that starting from an arbitrary state (in response to an arbitrary perturbation modifying the memory state), it is guaranteed to reach a state with no more than one token in the network. The protocol implements a strictly fair token circulation—during a round, every processor obtains the token exactly once. The proposed protocol has extremely small memory requirement—only $O(1)$ bits of memory per incident network link.

Keywords: Mutual exclusion, self-stabilization, spanning tree, token passing.

1 Introduction

Robustness is one of the most important requirements of modern distributed systems. Various types of faults are likely to occur at various parts of the system. These systems go through the transient states because they are exposed to constant change of their environment. The concept of self-stabilization [5] is the most general technique to design a system to tolerate arbitrary transient faults. A self-stabilizing system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time.

The token circulation problem is similar to the mutual exclusion problem. A solution to the problem of mutual exclusion on a (uni-directional) ring is to implement a token circulating from one processor to the next in one direction; the token moves around the ring and a processor having the token is granted access to the shared resource and may execute the code in the critical section.

Related Work. Dijkstra introduced the property of self-stabilization in distributed systems by applying it to algorithms for mutual exclusion on a ring [5]. Several self-stabilizing token passing algorithms for different topologies have been proposed in the literature [16]: Dijkstra [5, 6], Burns and Pachl [4], Flatebo and Datta [8], and Flatebo, Datta, and Schoone [9] for a ring; Brown, Gouda, and Wu [3] and Ghosh [10] for a linear array of processors, Kruijer [15] for tree network,

*Contact author: Colette Johnen, LRI, Université de Paris-Sud, Bat. 490, Campus d'Orsay, F-91405 Orsay Cedex, France. phone : +33 1 69 15 67 02. fax : +33 1 69 15 65 86. email : colette@lri.fr

and Tchuente [17] on general networks. Recently, Huang and Chen [12] presented a token circulation protocol for a connected network in non-deterministic depth-first-search order, and Dolev, Israeli, and Moran [7] gave a mutual exclusion protocol on a tree network under the model whose actions only allow read/write atomicity. A memory-efficient token passing protocol on general network is presented in [14].

All the solutions to the token passing problem mentioned above use a distinguished processor whose program is different than the other processors in the network. Although this is undesirable in a fault-tolerant distributed system, but as Dijkstra commented in [6] that, there is no uniform deterministic self-stabilizing ring with a composite number of processors. Burns and Pachl [4] showed that there is a uniform self-stabilizing ring for every prime $n \geq 3$.

One of the important performance issues of self-stabilizing algorithms is the memory requirement per processor. The previous solutions to the token circulation problem on general networks have a space complexity of $O(\log n)$, where n is the number of processors. In these protocols, each processor maintains its *distance* to the distinguished processor. Awerbuch and Ostrovsky [1] proposed a $O(\log^* n)$ algorithm by using a data structure to maintain the *distance* variables in a distributed manner. Itkis and Levin [13] introduced another data structure based on Thue-Morse sequence and presented a $O(1)$ bits per link solution.

Contributions. In this paper, we present a self-stabilizing depth-first token circulation scheme on a general network with a distinguished root, called Algorithm $\mathcal{TP}$. One of the desirable features of the protocols written on large distributed systems is that they should not depend on the global properties, such as, network size, which can change over time. Algorithm $\mathcal{TP}$ has this feature and requires $O(1)$ bits of memory per link. When the network size changes, the Algorithm $\mathcal{TP}$ does not need to be modified. The code at a processor needs to be modified only when the degree of the processor changes (a locally checkable property). Our algorithm uses neither the *distance* variable nor any special data structure to achieve the $O(1)$ bits per link memory requirement. Algorithm $\mathcal{TP}$ also implements a strictly fair circulation of token.

The algorithm presented in [14] has the same space complexity. But, in this algorithm, a processor needs the knowledge of the state of the neighbors of its neighbors. Since the algorithm assumes the atomic execution of the actions, this requirement makes the atomic step bigger: in one atomic step, a processor reads the state of its neighbors, the neighbors of its neighbors, and finally changes its own state. In Algorithm $\mathcal{TP}$, a processor only reads the state of its neighbors in an atomic step. Thus, Algorithm $\mathcal{TP}$ has a smaller atomicity than that in [14].

2 Preliminaries

In this section, we define the distributed systems and programs considered in this paper, and state what it means for a protocol to be self-stabilizing. We then present the statement of the token passing problem and its properties.

2.1 Self-Stabilizing System

System. Let $\mathcal{DS} = (PR, M)$ be a *distributed system*, where PR is a set of processors and M is a set of bidirectional communication links. We will denote the processors by $i :: i \in \{1..n\}$ and the root processor by r. A communication link (i, j) exists iff i and j are neighbors. Each processor

i maintains its set of neighbors, denoted as $N.i$. Every processor owns a shared register (defined in the following paragraph). The processors may only communicate with their neighbors using the shared registers.

Programs. Each processor executes a program and the processors execute their programs asynchronously. The program consists of a set of variables and a finite set of actions. The processors have two types of variables: *local variables* and *field variables*. The field variables are part of the shared register which is used to communicate with the neighbors. The local variables defined in the program of processor i are used strictly locally, meaning that they cannot be accessed by the neighbors of i. A processor can only write to its own shared register and can only read shared registers owned by the neighboring processors. So, the field variables of i can be accessed by i and its neighbors. The program of each processor consists of a finite set of actions. Each action is uniquely identified by a label and is of the following form:

$$< label >:: < guard > \longrightarrow < statement >$$

The guard of an action in the program of i is a boolean expression involving the local variables of i, and the field variables of i and its neighbors. The statement of an action of i updates zero or more local variables and field variables of i. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed: the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step. The atomic execution of an action of i is called a *step* of i; this is known as the *distributed daemon* [2].

The *state* of a processor is defined by the values of its field variables. The *state* of a system is a product of the states of all processors ($\in PR$). In the sequel, we refer to the state of a processor and system as a *local state* and *global state*, respectively. A *computation* of a protocol $\mathcal{P}$ is a *fair, maximal* sequence of global states $\Phi = (\delta_1, \delta_2, \ldots)$ such that for $i = 1, 2, \ldots$, the global state δ_{i+1} is reached from δ_i by a single *computation step*. During a computation step, one or more processors execute a step and a processor may take at most one step. *Fairness* of the sequence means that if any action in $\mathcal{P}$ is continuously enabled along the sequence, it is eventually chosen for execution. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final global state. All computations considered in this paper are assumed to be fair and maximal.

Let $\mathcal{L}$ be a global state predicate of a protocol $\mathcal{P}$ specified with respect to the problem specification $\mathcal{SP}$ that $\mathcal{P}$ implements. $\mathcal{L}$ holds at all global states reached by the computations of $\mathcal{P}$ that meet $\mathcal{SP}$. Thus, $\mathcal{L}$ characterizes the set of all global states reached in the "correct" computations of $\mathcal{P}$. The predicate $\mathcal{L}$ is called a *legitimacy predicate* or an *invariant* of $\mathcal{P}$.

The protocol $\mathcal{P}$ is *self-stabilizing* for the specification $\mathcal{SP}$ if (i) every computation of $\mathcal{P}$ starting from a global state where $\mathcal{L}$ holds preserves $\mathcal{L}$ (**closure**), and (ii) starting from any arbitrary global state, every computation of $\mathcal{P}$ reaches a global state where $\mathcal{L}$ holds (**convergence**).

2.2 Specification of the Token Passing Protocol

The legitimacy predicate $\mathcal{L}_{TP}$ of our token passing protocol is any global state such that (i) no two processors have simultaneous token access at any time (called the **Single Token** property), and (ii) for each computation that starts in such a global state, during a token circulation round, each processor obtains the token exactly once (called the **Strict Fairness** property).

We also require our solution to the token passing problem to be **self-stabilizing**.

3 Token Passing Protocol

In this section, we propose a self-stabilizing depth-first token circulation algorithm. We first present the general approach to the solution, followed by the definition of the local and field variable at each processor. Finally, we present the actions of the algorithm formally.

3.1 Idea of the Solution

The proposed algorithm has two major tasks: to circulate the token in the network in a deterministic depth-first order and to handle the abnormal situations (illegal states) due to the unpredictable initial states and transient errors. The actions of the program for the token circulation are formally presented in Section 3.3 and the actions for the error handling processor are given in Section 3.4.

The root r (in the sequel referred to as the *legal root*) initiates a depth-first *circulation round* with a color of 1 or 3. We will refer to the *circulation round* as *cround*, and the color used in a circulation round as r_color. The circulation rounds with the colors of 1 and 3 are referred to as $cround_1$ and $cround_3$, respectively. Once the system is stabilized, in one *cround*, the token traverses every processor in the network in the depth-first order. The branches created due to the traversal of the token are called the *depth-first branches*.

In the beginning of a *cround*, the processor r holds the token and it is the leaf of the depth-first branch. The leaf (in this case r) chooses one of the neighbors i as a child and passes the token to i. So, i is the leaf now. After i uses the token (or exits the critical section), i selects one of its neighbors j who did not get the token yet. So, the token traversal procedure extends the branch originating at r. This continues until a leaf is unable to find any unvisited neighbor. At this time, the leaf drops the token, allowing its parent to remove this branch, create another depth-first branch (choose another child), and pass the token to an unvisited neighbor (the child). If the parent cannot find an unvisited neighbor, the branch is shrunk. This token traversal continues until the branch is reduced to r, i.e, all processors of the network have been visited during this *cround*. This completes the *cround*. r now initiates another *cround* with the other color and the token traversal is repeated.

The system has an unpredictable initial state—illegal branches or cycles can may exist initially. There are mainly two error handling tasks: one to remove the illegal branches and the other to eliminate the cycles. Our approach to handling the illegal branches (which are not cycles and are not rooted to the legal root) is similar to the ideas in [12] and [14], and is formally presented in Section 3.4.1. The illegal roots detect their abnormal situation and change their status to E. The E status is propagated to their leaves, these E status leaves are de-linked from their parents, and finally, these detached processors are recovered (changing their status to Ok).

The cycle elimination strategy is similar to the one proposed in [14]. Typically, a *distance* variable is used for this purpose. But, we do not use such variables. The basic idea is to detect the cycle by a processor which does not belong to the cycle. The coloring scheme (discussed in detail in Section 3.2) is designed such that during $cround_1$ ($cround_3$), no processor should have a color of 3 (1). Therefore, if in $cround_1$ ($cround_3$), the processor having the token i has a 3-colored (1-colored) neighbor j, j is faulty (j may be inside a cycle). The leaf i chooses j as the child. The faulty processor j detects that it has two parents and changes its status to $E++$. All descendents of j change their status to E, Then the parent of the faulty processor inside the cycle drops its faulty child and breaks the cycle. As, the nodes inside a cycle cannot change their color, all cycles

are eventually destroyed.

3.2 Basic Structures

In this section, we define the field variables of the register and local variables at each processor.

The field variables are denoted as $field\_name.processor\_id$. For example, $D.i$ refers to the field D of processor i. If a field variable $D.i$ points a processor j (a neighbor of i), then $D.D.i$ refers to the field D of $D.i$, i.e., $D.D.i = j$.

The field variables of processor i are defined as follows:

- $D.i$:: The child pointer. It points to one of its neighbors or contains $NULL$, i.e., $D.i \in N.i$ or $D.i = NULL$. If $D.i = NULL$, then $D.D.i = NULL$.
- $S.i$:: The status. It contains a value $\in \{Ok, E, E++\}$. Once the system stabilizes, all processors have the Ok status. The E and $E++$ status are used during the error recovering processor and explained in the following sections.
- $C.i$:: The color. It takes value $\in \{0, 1, 2, 3\}$. All arithmetic operations ($+$ and $-$) on C are *modulo* 4. The legal root r initiates depth-first token circulation rounds $cround_1$ and $cround_3$ alternately (see Section 3.1). Once the system stabilized, all processors except the leaf, in the current depth-first branch, have an odd color of 1 or 3 corresponding to r_color. Other processors have an even color. The processors who are yet to receive the token during this circulation round are colored $r\_color - 1$. The processors who received the token, but are not in the depth-first branch (the branch expands and shrinks as explained in Section 3.1) are colored $r\_color + 1$. The idea of using a 4-state variable is an extension of the ideas in [5, 10] on a general graph.

Before evaluating their guards, the processors read the shared registers of their neighbors and update their local variables. The local variables of i are defined below:

- P_i:: The set of parents of i. Ideally, a processor should have at most one parent. But, due to the faults, a processor may have more than one parent. The parent-child relations satisfy $j \in P_i \Leftrightarrow D.j = i$.
- NP_i:: The number of parents of $i = |P_i|$.

In the next two sections, we present the actions of the Algorithm $\mathcal{TP}$. Our protocol consists of fourteen actions: TC1-TC4, IB1-IB5, CE1-CE4, and ER1.

3.3 Token Circulation

The actions for the token circulation use the following definitions:

- $EvenColor(i) \equiv (C.i = 0) \vee (C.i = 2)$
- $OddColor(i) \equiv (C.i = 1) \vee (C.i = 3)$
- $GoodLeaf(i) \equiv (D.i = NULL) \wedge (S.i = Ok) \wedge EvenColor(i)$
- $Token(i) \equiv GoodLeaf(i) \wedge ((i = r) \vee ((i \neq r) \wedge (NP_i = 1) \wedge (\exists j \in P_i :: (C.i = C.j - 1))))$
  The legal root r holds the token iff $GoodLeaf(r)$ holds. Processor $i \neq r$ holds a token iff $GoodLeaf(i)$ holds and its color is its parent's color minus 1.
  i may enter the critical section iff $Token(i)$ holds.

- $Anomalous(i,k) \equiv (k \in N.i) \wedge (k \neq r) \wedge (C.k = C.i + 3)$
  The processor i has a neighbor k which has an "unexpected" color with respect to i. The unexpected color is $C.i + 3$.
- $FirstChild(i,k) \equiv Token(i) \wedge (k \in N.i) \wedge (k \neq r) \wedge (S.k = S.i) \wedge (C.k = C.i) \wedge (\forall j \in N.i :: \neg Anomalous(i,j))$
  Processor i holds the token, in i's neighborhood there is no processor with an "unexpected color", and i has a neighbor k which could be a potential child.
- $DeadEnd(i) \equiv Token(i) \wedge (\forall j \in N.i :: \neg Anomalous(i,j) \wedge \neg FirstChild(i,j))$
  Processor i does not have any neighbor which can be a possible $FirstChild$.
- $ChildDone(i) \equiv (D.i \neq NULL) \wedge (S.i = Ok) \wedge OddColor(i) \wedge (C.i = C.D.i - 1)$
  Processor i has a child k who has used the token and is a leaf.
- $NVChild(i,k) \equiv (k \in N.i) \wedge (k \neq r) \wedge (S.k = Ok) \wedge (C.k = C.i - 1)$
  Processor i has a neighbor k who can be a possible child. k did not get the token yet in this $cround$.
- $NewChild(i,k) \equiv ChildDone(i) \wedge NVChild(i,k)$
  Processor i selects k as the next child.
- $Backtrack(i) \equiv ChildDone(i) \wedge (\forall j \in N.i : \neg NVChild(i,j))$
  Processor i does not have any neighbor which can be a possible child.

The actions of the program for token circulation at processor i are defined in Figure 1.

TC1::	$DeadEnd(i)$	$\longrightarrow$	$C.i = C.i + 2$
TC2::	$FirstChild(i,k)$	$\longrightarrow$	$C.i = C.i + 1;\ D.i = k$
TC3::	$NewChild(i,k)$	$\longrightarrow$	$D.i = k$
TC4::	$Backtrack(i)$	$\longrightarrow$	$D.i = NULL;\ C.i = C.i + 1$

Figure 1: Actions for Token Circulation.

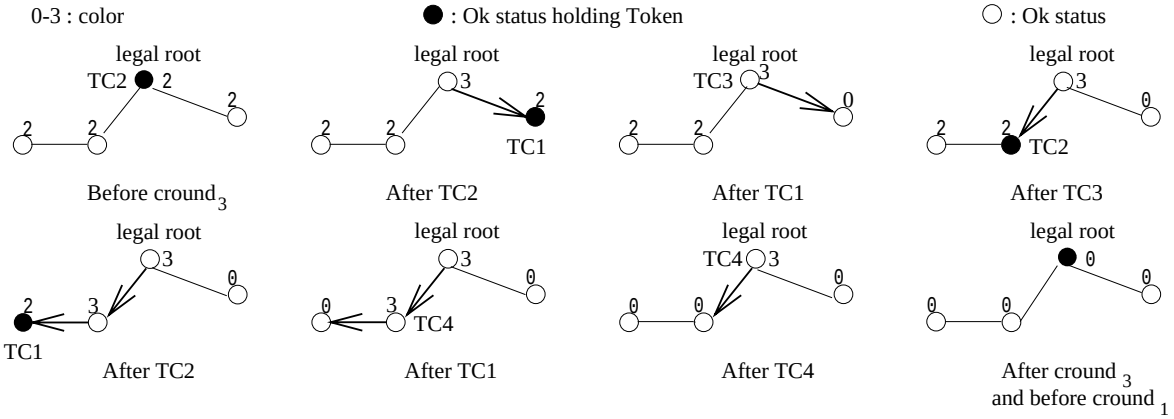

Figure 2: Token Circulation.

Figure 2 shows the circulation of a token starting from the legal root. The legal root r initiates a circulation round ($cround_3$ in the figure) by executing the action TC2. The token moves from r (previous leaf) to its first child which is now the leaf. Thus, the tree expands following a branch. The leaf uses the token, but cannot find a first child to pass the token to. So, it drops the token by action TC1. The legal root now finds another suitable unvisited child which can receive the token (action TC3). This child becomes the new leaf. If the current leaf has used the token and an unvisited processor does not exist, the branch is shrunk by the action TC4. When the branch is completely destroyed (e.g. the round is over), the legal root (r) has the token and it starts another *round* ($cround_1$ in Figure 2).

3.4 Error Handling

A distributed system has an unpredictable initial state where the D pointers may point to any neighbors or $NULL$. Thus, illegal branches or cycles may exist in the initial state. In this section, we present two error handling tasks: one to eliminate the illegal branches and the other to remove the cycles of D pointers. The actions defined in the following sections together with those defined in the previous section (Section 3.3) complete our solution to the Algorithm $\mathcal{TP}$.

3.4.1 Elimination of Illegal Branches

We use the following definitions in the actions to eliminate the illegal branches:

- $BadShape(i) \equiv ((D.i \neq NULL) \wedge (S.i = Ok) \wedge EvenColor(i)) \vee ((D.i = NULL) \wedge (S.i = Ok) \wedge OddColor(i)) \vee ((D.i = NULL) \wedge (S.i = E++))$
- $FastBacktrack(i) \equiv (D.i \neq NULL) \wedge (D.D.i = NULL) \wedge (S.D.i \neq Ok)$
  Processor i has a child whose status is *not* Ok and is a leaf.
- $Detached(i) \equiv (D.i = NULL \wedge (NP_i = 0)$
  Processor i has no child and no parent.
- $IllegalRoot(i) \equiv (i \neq r) \wedge (D.i \neq NULL) \wedge (NP_i = 0)$
  Processor i is the root of a branch, but is not the legal root r.
- $IllegalParent(i) \equiv (S.i = Ok) \wedge (\exists k \in N.i :: ((D.k = i) \wedge (S.k \neq Ok)))$
  The status of i's parent is not Ok.

The actions to eliminate the illegal roots and branches at processor i are defined in Figure 3.

IB1::	$BadShape(i)$	$\longrightarrow$	$D.i = NULL;\ S.i = E$
IB2::	$FastBacktrack(i)$	$\longrightarrow$	$D.i = NULL;\ S.i = E$
IB3::	$IllegalParent(i)$	$\longrightarrow$	$S.i = E$
IB4::	$Detached(i) \wedge (S.i \neq Ok)$	$\longrightarrow$	$S.i = Ok;\ C.i = 0$
IB5::	$IllegalRoot(i) \wedge (S.i \neq E)$	$\longrightarrow$	$S.i = E$

Figure 3: Actions to Eliminate the Illegal Roots and Illegal Branches.

Processor i which is in $BadShape$ but not in status E executes action IB1, changes its status to E and de-links itself from its child. Processor i's status is changed to Ok later by executing actions IB2, IB3, and IB4.

Figure 4 shows an example of eliminating an illegal root and an illegal branch. An illegal root executes action IB5 and changes its status to E. The E status propagates from a processor to its child by executing action IB3. A processor de-links an erroneous child that is a leaf (with status E) by executing action IB2. Thus, child get detached. The Erroneous detached processors are recovered by action IB4 by changing their status to Ok.

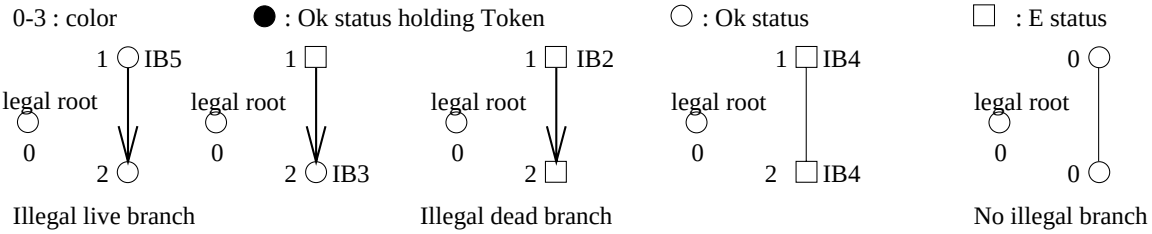

Figure 4: Elimination of An Illegal Root and Illegal Branch

3.4.2 Elimination of Cycles

The actions of the program to eliminate the cycles of D pointers are given in Figure 5 and are illustrated using an example in Figure 6. Processor i detects an anomalous processor k and becomes its new parent (action CE1). A processor having several parents and a child (thus, may be inside a cycle) changes its status to $E++$ (action CE2). The E status is assigned to a processor having several parents but no child (action CE3). The E status propagates to the descendants of the anomalous processor by repeated execution of action IB3. Then the parent of the anomalous processor (that is inside the cycle) executes action CE4 and breaks the cycle. Action IB2 is repeatedly executed until all the D pointers in the previous cycle are reset to $NULL$. Thus, all processors become detached. Finally, as explained in Figure 4, these detached processors are recovered by executing the action IB4 repeatedly.

CE1::	$Token(i) \wedge Anomalous(i,k)$	$\longrightarrow$	$D.i = k;\ C.i = C.i + 1$
CE2::	$(D.i \neq NULL) \wedge (S.i \neq E++) \wedge (NP_i \geq 2)$	$\longrightarrow$	$S.i = E++$
CE3::	$(D.i = NULL) \wedge (S.i \neq E) \wedge (NP_i \geq 2)$	$\longrightarrow$	$S.i = E$
CE4::	$(S.i \neq Ok) \wedge (S.D.i = E++)$	$\longrightarrow$	$D.i = NULL;\ S.i = E$

Figure 5: Actions to Eliminate Cycles.

3.4.3 Miscellaneous Error Handling

If processor i's D pointer points to the root r, then i removes that link (action ER1) because the root cannot have a parent.

- **ER1::** $D.i = r \longrightarrow D.i = NULL$

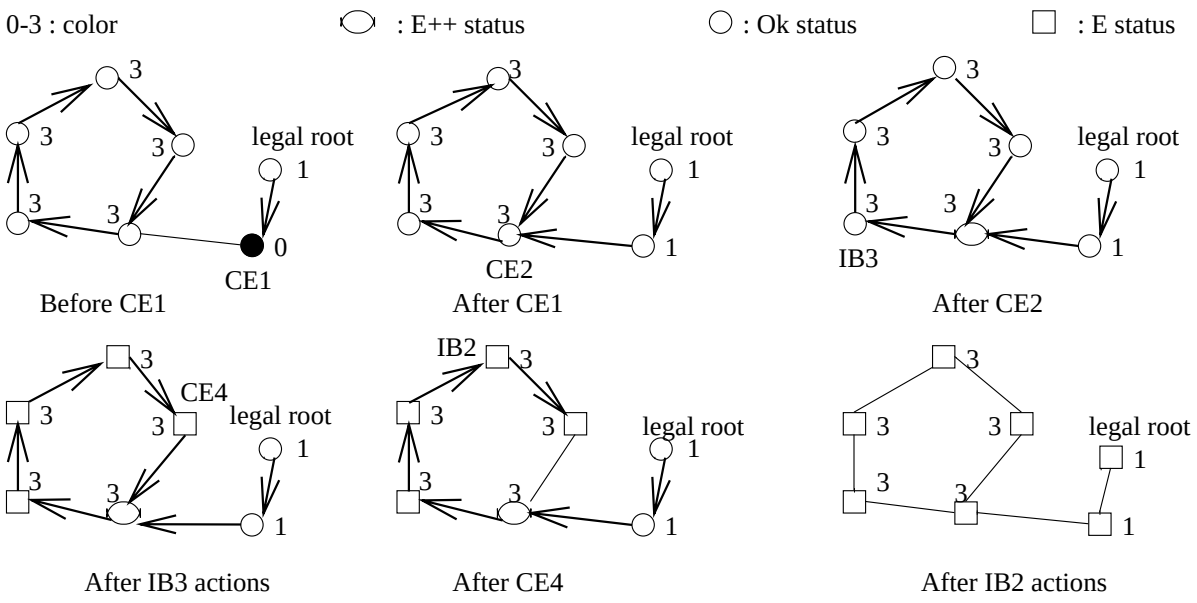

Figure 6: Elimination of Cycles.

4 Outline of the Correctness of the Token Passing Protocol

In this section, we present only the main results and the ideas of proving them. A few short proofs are given in the appendix.

Definition 4.1 (Attractor) *A global state predicate B of a protocol $\mathcal{P}$ is called an attractor for another global state predicate A of $\mathcal{P}$, if* (i) *B is closed in $\mathcal{P}$, i.e., once B holds in an arbitrary system computation in $\mathcal{P}$, it continues to hold subsequently, and* (ii) *upon starting at an arbitrary state in A, the system is guaranteed to reach a global state in B. We denote this relation as $B \triangleleft A$.*

The notation $\delta \vdash p$ means that the global state δ satisfies predicate p.

We apply the convergence stair method [11] to prove our protocol. We exhibit a finite sequence of state predicates $A_0, A_1, \ldots, A_m$, of Algorithm $\mathcal{TP}$ such that (i) $A_0 \equiv true$ (meaning any arbitrary state); (ii) $(A_m \equiv L_{TP}) \vee (A_m \vdash L_{TP})$; (iii) $\forall j : 0 \leq j < m :: A_{j+1} \triangleleft A_j$.

First, we prove that all maximal computations are infinite. We partition the global states into two sets depending on whether there exists a leaf or not. If there exists no leaf processor, then the legal root has a parent (ER1 is enabled), or there is an infinite path rooted at r (i.e., a processor inside this path has two parents). A processor in the path can execute CE2, CE4, or IB2 action. If there exists a leaf, then the leaf or its parent can execute one of the actions.

Theorem 4.1 *In any global state, at least one action of Protocol $\mathcal{TP}$ at a processor is enabled.*

The following two theorems easily follow from the actions of $\mathcal{TP}$.

Theorem 4.2 $A_1 \equiv NP_r = 0$. $A_1 \triangleleft A_0$.

Theorem 4.3 $A_2 \equiv A_1 \wedge (\forall i : i \in \{1, n\} :: \neg BadShape(i))$. $A_2 \triangleleft A_1$.

A branch not rooted at r and whose leaf is in Ok status, is called an *illegal and live branch*. This kind of branch may expand because the processors in this branch may execute the token circulation actions. We need to prove that all illegal and live branches will eventually be destroyed. One complication in the process of removing these branches is that new illegal and live branches may be created.

A new illegal and live branch is created when all parents of a processor i simultaneously execute CE4—i becomes the root of the new illegal branch. After the execution of CE4 by a parent j of i, j becomes a dead leaf, and the branch containing j and i is now a dead branch. If i is inside a dead branch, after the execution of CE4 by all parents of i, i is a root of a dead branch (there is not creation of a new illegal branch). If i is in several illegal and live branches, after the execution of CE4 by i's parents, all theses branches are now dead. Thus, the illegal branch rooted at i replaces several illegal branches. If i is in only one illegal and live branch, then i has $E++$ status and has only one parent. Such a situation may exist initially, or may happen if i had several parents and has already lost at least one parent (by a previous execution of CE4 by a parent of i): during the computation, several illegal and live branches are replaced by only one. Thus, only a finite number of illegal live branches may be created (this number depends on the initial global state). Then, by fair scheduling of IB3 and IB5, the illegal and live branches are converted to dead branches.

Theorem 4.4 $A_3 \equiv A_2 \wedge$ *(there exists no illegal and live branch).* $A_3 \triangleleft A_2$.

In A_3, there is at most one live leaf which is inside the legal branch. Thus, only one of the actions TC1, TC2, TC3, TC4, or CE1 is enabled at only one processor i (the leaf of the legal branch or its parent).

We call a legal branch *color consistent* if it is dead or all processors in it (except the leaf) are colored *r_color*. Once $Token(r)$ holds true, the legal branch is color consistent and stays color consistent. We can prove that all maximal computations contain an infinite number of states where $Token(r)$ holds. Let Φ be a computation with a finite number of states where $Token(r)$ holds. In A_3, after the execution of CE1, the legal branch ends in a cycle or a dead leaf. Then the legal branch will eventually destroy itself and $Token(r)$ will hold. Thus, Φ does not contain any execution of CE1. Eventually, along Φ, only the token circulation actions are executed infinitely many times. If Φ does not contain the execution of TC4 in r, then Φ is finite. After the execution of TC4, the system reaches a state where $Token(r)$ holds.

Theorem 4.5 *In A_3, all computations contain an infinite number of states where $Token(r)$ holds.*

Theorem 4.6 $A_4 \equiv A_3 \wedge$ *(the legal branch is color consistent).* $A_4 \triangleleft A_3$.

If the legal branch is not color consistent, then a cycle may be created. The live leaf of the legal branch may execute CE1 and choose a child which is in the legal branch and which is not colored *r_color*, creating a cycle. Therefore, if there is at least an illegal, live branch, or if the legal branch is color inconsistent. then a cycle can be created. But, in A_4, no cycle is created.

We need to prove that the cycles are eventually destroyed. We consider a computation Φ where some processors are in some cycles. Let $\mathcal{N}_\Phi$ denote the non-empty set of processors which are in a cycle in any state in Φ. We denote the distance between i and r as $\mathcal{D}is_i$, and the minimal distance

between r and a processor in $\mathcal{N}_\Phi$ by $\mathcal{D}is_\Phi$. The processors in $\mathcal{N}_\Phi$ are in *strict cycles*. A cycle is called a *strict cycle* if every processor in the cycle has only one parent and is not in $E{+}{+}$ status. By fair scheduling of CE2, IB3, and CE4, the cycles which are not strict cycles, will eventually become dead branches. Let $\mathcal{D}is_\Phi = \mathcal{D}is_i + 1$ and i has a neighbor $k \in \mathcal{N}_\Phi$. By induction on the distance between i and r, we prove that Φ contains an infinite number of states where $Token(i)$ holds. If $Token(i)$ holds, either $C.i = C.k+1$ or $C.i = Ck+3$. If $C.i = C.k+1$, then the next time $Token(i)$ holds, i's color will become $C.k+3$. The reason is that, in the meantime, i has executed TC1 or a sequence TC2TC3^{n}TC4, and k did not change its color. If $Token(i)$ holds and $C.i = C.k+3$, then CE1 is enabled at i. CE1 is the only action that can be executed in the protocol. After CE1 is executed at i, k is no longer inside a strict cycle.

Theorem 4.7 $A_5 \equiv A_4 \wedge (\forall i : i \in \{1,n\} :: \neg StrictCycle(i))$. $A_5 \triangleleft A_4$.

The cycles which are not strict cycles will become dead branches. The dead branches will destroy themselves by fair scheduling of IB2.

Theorem 4.8 $A_6 \equiv A_5 \wedge (\forall i : i \in \{1,n\} ::$ *i is inside the legal branch or Detached(i) holds).* $A_6 \triangleleft A_5$.

The detached processors that do not have the Ok status will recover (change to Ok status) by executing IB4.

Theorem 4.9 $A_7 \equiv A_6 \wedge (\forall i : i \in \{1,n\} :: S.i = Ok \wedge NP_i \leq 1)$. $A_7 \triangleleft A_6$.

In A_7, (i) only the token circulation actions are enabled, (ii) only one processor may execute one action, (iii) there is no cycle and no illegal branch, and (iv) all processors are in Ok status. Thus $A_7 \vdash L_{TP}$.

Let δ_0 be the global state where all processors are detached and are 0*_colored*, and δ_2 be the global state where all processors are detached and are 2*_colored*. At the end of a *cround*, more processors will be colored *r_color*. Thus, in any computation in A_7, δ_0 or δ_2 will be eventually reached.

Starting from δ_0 or δ_2, in one *cround*, all processors get the token exactly once, and at the end of the *cround*, the current state becomes δ_0 or δ_2. Thus, Protocol $\mathcal{TP}$ provides a strictly fair token circulating in the network after the system is stabilized.

5 Conclusion

We proposed a depth-first (strictly fair) token circulation protocol on rooted networks. The previous solutions for token circulation, except [14], on general network topology have a space complexity of $O(\log n)$, where n is the number of processors, because each processor stores its distance to the legal root. In Algorithm $\mathcal{TP}$, the distance variable is not used. The cycles are detected by processors who are not in the cycles. The size of variable D (child) of i is $O(\log \triangle_i)$ where $\triangle_i$ is the degree of i. The variables C (color) and S (status) are of constant size—4 bits total. The local variable NP (number of parents) takes $O(\log \triangle_i)$ space. The other local variable P (parents list) requires one bit per communication link. Thus, the space complexity of Algorithm $\mathcal{TP}$ is $O(1)$ per communication link.

References

[1] B. Awerbuch and R Ostrovsky, "Memory-efficient and self-stabilizing network reset," *Symposium on Principles of Distributed Computing*, Los Angeles, California, 1994, pp.254-263.

[2] J. Burns, M. Gouda, and R. Miller, "On Relaxing Interleaving Assumptions," *Proceedings of the MCC Workshop on Self-Stabilization*, Austin, Texas, November 1989.

[3] G. Brown, M. Gouda, and M. Wu, "Token Systems that Self-Stabilize," *IEEE Transactions on Computers*, Vol. 38, No. 6, June 1989, pp. 845-852.

[4] Burns J. and Pachl J. "Uniform Self-Stabilizing Rings," *ACM Transactions on Programming Language and Systems*, Vol. 11, No. 2, 1989, pp. 330-344.

[5] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM 17*, 1974, pp. 643-644.

[6] E. W. Dijkstra, "Self-Stabilization in Spite of Distributed Control," in *Selected writings on computing: a personal perspective*, Springer-Verlag, Berlin, 1982, pp. 41-46.

[7] S. Dolev, A. Israeli, and S. Moran, "Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity," *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec City, Canada, 1990, pp. 103-117; also *Distributed Computing* Vol. 7, 1993, pp. 3-16.

[8] M. Flatebo and A. K. Datta, "Two-State Self-Stabilizing Algorithms for Token Rings," *IEEE Transactions on Software Engineering*, June 1994, pp. 500-504.

[9] M. Flatebo, A. K. Datta, and A. A. Schoone, "Self-Stabilizing Multi-Token Rings," *Distributed Computing*, Vol. 8, 1995, pp. 133-142.

[10] S. Ghosh, "An Alternate Solution to a Problem on Self-Stabilization," *ACM Transactions on Programming Languages and Systems*, Vol. 15, No. 4, September 1993, pp. 735-742.

[11] M.G. Gouda and N. Multari, "Stabilizing Communication Protocols," *IEEE Transactions on Computing*, Vol. 40, No. 4, 1991, pp 448-458.

[12] S. Huang and N. Chen, "Self-Stabilizing Depth-First Token Circulation on Networks," *Distributed Computing*, Vol. 7, 1993, pp. 61-66.

[13] G. Itkis and L. Levin, "Fast and lean self-stabilizing asynchronous protocols," *35th Symposium on Foundations of Computer Science*, Santa Fe, New Mexico, 1994, pp. 226-239.

[14] C. Johnen and J. Beauquier, "Space-Efficient Distributed Self-Stabilizing Depth-First Token Circulation," *Proceedings of the 2nd Workshop on Self-Stabilizing Systems*, Las Vegas, Nevada, 1995, pp. 4.1-4.15.

[15] H.S.M. Kruijer, "Self-stabilizing (in spite of distributed control) in tree-structured systems," *Information Processing Letters*, 1979, 29:91–95.

[16] M. Schneider, "Self-Stabilization," *ACM Computing Surveys*, Vol. 25, No. 1, March 1993, pp. 45-67.

[17] M. Tchuente, "Sur l'auto-stabilisation dans un reseau d'ordinateurs," *RAIRO Informatique Theorique 15*, No. 1, 1981, pp.47-66.

A Definitions

We use the following definitions in the proofs of Protocol $\mathcal{TP}$:

Definition A.1 (Trap) *A predicate τ is called a trap in the global state δ if the following condition holds:* $(\forall i : i \in \{1, n\} :: \tau(i))$ *is closed in δ.*

- $Q_{0-k} \equiv (\exists$ (A sequence of processors $0, 1, \ldots, k) : \forall j \in [0, k[:: D.j = D.(j+1))$
- $Cycle(i) \equiv (\exists\ Q_{0-k} :\ (i = 0 = k))$
- $StrictCycle(i) \equiv (\exists\ Q_{0-k} :\ (i = 0 = k) \wedge (\forall j \in [0, k[:: NP_{j+1} = 1) \wedge (S.i \neq E++))$
  Processor i is in a cycle where every processor has only one parent and is not in $E++$ status.
- $InIllegalBranch(i)$ holds if the processor i is in a branch whose root is not r.
- $Leaf(i) \equiv (D.i = NULL) \wedge ((NP_i \geq 1) \vee (i = r))$
- $LiveLeaf(i) \equiv Leaf(i) \wedge (S.i = Ok)$
- $DeadLeaf(i) \equiv Leaf(i) \wedge (S.i \neq Ok)$
- $InLiveBranch(i)$ holds iff the processor i is in a branch ending with a live leaf.
- $InDeadBranch(i)$ holds iff the processor i is in a branch ending with a dead leaf or a cycle.
- $IllegalLiveRoot(i)$ holds if the processor i is an illegal root of a branch ending with a live leaf.
- $LiveEPProcessor(i) \equiv (S.i = E++) \wedge (NP_i = 1) \wedge InLiveBranch(i)$
- $SpLegalLive(i) \equiv InLegalLiveBranch(i) \wedge ((NP_i > 1) \vee (D.i = NULL) \vee (S.i = E++))$
  Processor i is a special processor in a live and legal branch, with some special properties.
- $IncLegalLiveBranch \equiv (\exists\ Q_{0-m} :\ (r = 0) \wedge InLiveBranch(r) \wedge SpLegalLive(m)$
  $\wedge (\forall i \in [0, m[:: \neg SpLegalLive(i)) \wedge (\exists\ i \in [0, m[::\ S.i = E))$
  $IncLegalLiveBranch$ is true if there exists a processor in a legal and live branch, with status E, between the root and the first $SpLegalLive$ processor in the legal branch.
- ILB = 1 if $IncLegalLiveBranch$ is *true*, otherwise, *false*.
- ILR = Number of processors which satisfy $IllegalLiveRoot$.
- LEP = Number of processors which satisfy $LiveEPProcessor$.
- $Sum_I = ILR + LEP + ILB$
- $IllegalProcessor(i)$ holds if the processor i is in a cycle, an illegal branch, or in a legal branch (in this case, i joined the legal branch by executing CE1).

Notations:

Action_a $\Uparrow i$ means that Action_a is enabled in the program of processor i at the current global state. (Action_a, Action_b, ... or Action_k) $\Uparrow i$ means that only one of the actions among Action_a, Action_b, ... or Action_k is enabled in the program of processor i at the current global state.

B Liveness of the Algorithm $\mathcal{TP}$

Theorem 4.1 *In any global state, at least one action of Algorithm $\mathcal{TP}$ at a processor is enabled.*

Proof: We consider two types of global states: there exists a leaf i and there exists no leaf. Assume that i is a leaf. If i has the $E++$ status, IB1 $\Uparrow i$. If i has the E status, IB2 $\Uparrow k$ $(k \in P_i)$. If $Token(i)$ holds, then (TC1, TC2, CE1) $\Uparrow i$. If $ChildDone(k)$ holds, then (TC3, TC4) $\Uparrow k$. Otherwise, $S.i \neq Ok$, $\neg Token(i)$, and $\neg ChildDone(k)$. Then IB1 $\Uparrow i$, IB3 $\Uparrow i$, CE3 $\Uparrow i$, or IB1 $\Uparrow k$.

Assume that there exists no leaf. If r has a parent l, then IB1 $\Uparrow l$. If r does not have a parent, then there is an infinite path rooted at r. This path must contain a processor (say i) sev-

eral times. Then CE2 $\Uparrow i$, IB3 $\Uparrow j$ (where j is inside the infinite path), or CE4 $\Uparrow k$ (where $k \in P_i$). □

C Destruction of Illegal Live Branches

Theorem 4.2 $A_1 \equiv NP_r = 0$. $A_1 \triangleleft A_0$.

Proof: A_1 is closed: No action creates a parent of r. Every computation reaches A_1: If A_1 does not hold in the current global state, at least one processor satisfies the guard of ER1. Every time ER1 is executed, the number of r's parents is reduced. Thus, by fair scheduling, A_1 will eventually hold. □

Theorem 4.3 $A_2 \equiv A_1 \wedge (\forall i : i \in \{1, n\} :: \neg BadShape(i))$. $A_2 \triangleleft A_1$.

Proof: After any action is executed by i, $BadShape(i)$ does not hold. Therefore A_2 is closed. If A_2 does not hold in the current global state, at least one processor satisfies the guard of IB1. □

Remark C.1

IB1, ER1 $\oslash A_2$.
$Token(i) \Leftrightarrow$ *(TC1, TC2, or CE1)* $\Uparrow i$.
$ChildDone(i) \Leftrightarrow$ *(TC3, TC4)* $\Uparrow i$.
In A_2, $(OddColor(i) \wedge (S, i = Ok)) \Leftrightarrow (D.i \neq NULL)$,
In A_2, $(EvenColor(i) \wedge (S, i = Ok)) \Leftrightarrow (D.i = NULL)$.
In A_2, *if* $(Token(i), i \neq r)$ *holds, then* $\neg ChildDone(k)$ *holds, where* $k \in P_i$.
In A_2, *a processor can execute TC1, TC2, or CE1 iff it is a live leaf.*
In A_2, *a processor can execute TC3 or TC4 iff it is in a live branch.*
The processors inside a cycle or a dead branch cannot change their color.
A dead branch cannot gain a new processor or a live leaf.

Theorem C.1 $A_3\prime \equiv A_2 \wedge (Sum_I = 0)$. $A_3\prime \triangleleft A_2$.

Proof: In A_2, only the execution of CE1 by the leaf of the legal branch changes the value of ILB from 0 to 1: the leaf chooses k as a child and $IllegalLiveRoot(k)$ holds. After the execution of CE1, we may have $ILB = 1$. But, $IllegalLiveRoot(k)$ does not hold now. Thus, Sum_I does not increase.

$IllegalLiveRoot(i)$ ($LiveEPProcessor(i)$) holds after the execution of an action even if it did not hold before, iff all parents of i (all parents except one) execute action CE4 and i was inside a live branch. Thus, in A_2, only CE4 executed by a processor in a live branch may increase the value of Sum_I. Let i be a processor in a live branch, and j be a parent of i that executes CE4. Before the execution of CE4, $IllegalLiveRoot(k)$ (where k is the root of the branch j is in), $IncLegalLiveBranch$, or $LiveEPProcessor(l)$ (where l is in the legal branch) holds. After the execution of CE4, none of these predicates holds. If i has only one parent j, then $LiveEPProcessor(j)$ was true before this step. But, after the execution of CE4, only $IllegalLiveRoot(i)$ holds and Sum_I has decreased. If i has several parents, and only j executes CE4, then after the execution of CE4, $LiveEPProcessor(i)$ may hold but Sum_I did not increase. If several parents of i execute CE4 during a computation step, then Sum_I decreases. Thus, Sum_I never increases, and decreases every time that ILR increases. Therefore, $A_3\prime$ is closed.

By fair scheduling of the actions IB2, IB3, and IB5, $A_3\prime$ will eventually hold. □

The following theorem follows from Theorem C.1:

Theorem 4.4 $A_3 \equiv A_2 \wedge$ (there is not illegal, live branch). $A_3 \triangleleft A_2$.

D Color Consistency of the Legal Branch

A legal branch is called *color consistent* if it does not end in a live leaf, or all processors in it, except the leaf, have the r_color.

Remark D.1
In A_3, there is no illegal live branch and there is at most one live leaf.
In A_3, only one of the actions TC1, TC2, TC3, TC4, or CE1 is enabled at only one processor i which is either the live leaf or the parent of the live leaf.
In A_3, if a processor i in a legal branch executes CE1, then the legal branch becomes dead and color consistent, i.e., the legal branch ends in a cycle or a dead leaf.

We define $A_4 \equiv A_3 \wedge$ (the legal branch is color consistent).

Theorem 4.5 *In A_3, all computations contain an infinitely many states where $Token(r)$ holds.*

Proof: Let Φ be a computation with a finite number of states where $Token(r)$ holds. After an action CE1, the legal branch ends in a cycle or a dead leaf (Remark D.1). Then the legal branch will eventually destroy itself and $Token(r)$ will hold. Thus, CE1 is not executed in Φ after the state where $Token(r)$ holds last. Then, no new processor will have several parents, and Φ will eventually reach a state where all processors have at most one parent. Once this state is reached, no new processor will get the $E++$ status in Φ. Thus, by fair scheduling of IB3, IB2, and CE4, the system will will reach a state in Φ, where no processor has the $E++$ status. So, the actions IB1, CE1, CE2, CE3, CE4, and ER1 are not executed in Φ. All the illegal branches are dead (they will delete themselves by executing IB2), and no new one will be created (only CE4 can create a new illegal root). Thus, only one of the token circulation actions is executed infinitely many times. Φ would be finite if TC4 is not executed by r. After the execution of CE4, the system reaches a state where $Token(r)$ holds. Thus, Φ does contain an infinitely many states where $Token(r)$ holds. □

Theorem 4.6 $A_4 \triangleleft A_3$.

Proof: If $Token(r)$ holds, the legal branch is color consistent. The token circulation actions ensure that a processor creating a child takes the same color as that of the parent. In A_3, after the execution of CE1, the legal branch becomes dead and color consistent (Remark D.1). IB4 changes the color of a processor inside the legal branch, iff it is executed by r. After IB4 is executed, the legal branch becomes color consistent. Other actions do not change the color. Thus, the legal branch, once color consistent, preserves the same property. □

E Destructions of Cycles

We define $A_5 \equiv A_4 \wedge (\forall i : i \in \{1, n\} :: \neg StrictCycle(i))$.

Lemma E.1 *$\neg StrictCycle$ is a trap in A_4.*

Proof: In A_4, no action creates a new cycle, and a non-strict cycle cannot become a strict cycle. □

After the execution of CE1, the legal branch may contain processors in $E++$ and E status. A processor in the legal branch may execute CE4 and create a new illegal branch (dead). Thus, $\neg InIllegalBranch$ is not a trap in A_4. But, all processors in this branch held $IllegalProcessor$ before executing CE4. Therefore, we can prove that $\neg IllegalProcessor$ is a trap in A_4.

Lemma E.2 *$IllegalProcessor(i) \wedge \neg StrictCycle(i)$ cannot hold forever.*

Proof: By fair scheduling of IB3, CE2, and CE4, the non-strict cycles will be broken into dead branches. The illegal branches are all dead and will eventually destroy themselves. After the execution of CE1, the legal branch ends in a non-strict cycle or a dead leaf. □

Lemma E.3 *$\neg IllegalProcessor$ is a trap in any state of A_4 reached by any computation where $Token(r)$ has held.*

Proof: Assume that $IllegalProcessor(i)$ is false and after the execution of some action, $IllegalProcessor(i)$ holds. (i) Assume i joined the legal branch by CE1. But, before this step, i was inside a dead branch or a cycle. Thus, $IllegalProcessor(i)$ was true. (ii) Assume that a new illegal branch is created by the execution of CE4. But, the processors in this new branch satisfied $IllegalProcessor$ before the step. Once $Token(r)$ is true, all processors that joined the legal branch by executing TC2 or TC3, except the leaf, have the Ok status and have at most one parent. Thus, none of them can execute CE4. Therefore, only processors that satisfy $IllegalProcessor$, may execute CE4. □

We prove that starting from an arbitrary state in A_4, the system is guaranteed to reach a state in A_5. We prove this by contradiction. We assume the contrary, i.e., there exists a computation Φ starting from a state in A_4 such that it does not reach A_5. Let $mathcalN_\Phi$ denote the non-empty set of processors which are in a strict cycle in every state in Φ. Let $mathcalD_\Phi$ indicate the minimal distance between r and a processor in $mathcalN_\Phi$. We define A_{41} as a global state where all processors in $mathcalN_\Phi$ are inside a cycle and other processors are not inside any cycle. $A_{42} \equiv A_{41} \wedge (IllegalProcessor(i) \Rightarrow StrictCycle(i))$. Once $Token(r)$ holds, the system will reach a state in A_{42} in the computation Φ and A_{42} will continue to hold in Φ (Lemmas E.2 and E.3). In A_{42}, CE1 breaks a strict cycle into a non-strict cycle. Therefore, CE1 cannot be executed in A_{42} in the computation Φ. As in the proof of Theorem 4.5, we can prove that the system will eventually have only the token circulation actions executed in Φ.

Lemma E.4 *Let i be a neighbor of r. If $\mathcal{D}is_\Phi > 1$, then in A_{42}, $Token(i)$ holds infinitely many times in the computation Φ.*

Proof: After TC4 is executed once in r, TC4 will be enabled again only after all its neighbors change their color. We know that TC4 is executed infinitely many times by r. Moreover, i can change its color only if $Token(i)$ holds. □

Similarly, we prove the following lemma:

Lemma E.5 *Let i be a processor such that $\mathcal{D}is_i < \mathcal{D}is_\Phi - 1$ and $Token(i)$ holds in infinitely many states in the computation Φ. Let k be a neighbor of i. Then in A_{42}, $Token(k)$ holds infinitely many times in the computation Φ.*

Lemma E.6 *All computations starting from a state in A_4, reach a state in A_5.*

Proof: Let i be a processor such that $\mathcal{D}is_\Phi = \mathcal{D}is_i + 1$ and $StrictCycle(k)$ holds, where k is a neighbor of i. By induction on the distance between the processors i and r, we can show that Φ contains an infinitely many states where $Token(i)$ holds (by Theorem 4.5, and Lemmas E.4 and E.5). When $Token(i)$ holds, either $C.i = C.k + 1$ or $C.i = C.k + 3$. If $C.i = C.k + 1$, the next time $Token(i)$ holds, i's color will become $C.k + 3$. The reason is that i executes TC1 or a sequence TC2TC3^{n}TC4, and k does not change its color. When $Token(i)$ holds and $C.i = C.k + 3$, i satisfies the guard of CE1. CE1 is the only action which can be executed in the protocol. Thus, CE1 is executed in Φ, proving the contradiction. □

The following theorem follows from Lemmas E.1 and E.6.

Theorem 4.7 $A_5 \triangleleft A_4$.

F Fair Token circulations

Theorem 4.8 $A_6 \equiv A_5 \wedge (\forall i : i \in \{1, n\} :: i$ is inside the legal branch or $Detached(i)$ holds). $A_6 \triangleleft A_5$.

Proof: The processors that are not detached, are inside the legal branch, or satisfy the $IllegalProcessor$ predicate (they are in a non-strict cycle or a dead branch). Therefore, $A_6 \equiv A_5 \wedge (\forall i : i \in \{1, n\} :: \neg IllegalProcessor(i))$.

As every computation contains infinitely many global states where $Token(r)$ holds (Theorem 4.5), the $\neg IllegalProcessor$ predicate is a trap in A_5 (Lemma E.3). So, A_6 is closed.

In A_5, a processor i which satisfies $IllegalProcessor(i)$ is not in a a strict cycle¿ By Lemma E.2, $IllegalProcessor(i)$ will not hold forever. □

Theorem 4.9 $A_7 \equiv A_6 \wedge (\forall i : i \in \{1, n\} :: S.i = Ok \wedge NP_i \leq 1)$. $A_7 \triangleleft A_6$.

Proof: Action $CE1$ is not executed by any computation in A_6. So, no new processor can have several parents. All computations will eventually reach a state where all processors have at most one parent. Once this state is reached, no new processor will get the $E++$ status. Thus, by fair scheduling of IB3, IB2, and CE4, the system will will reach a state, where no processor has the $E++$ status. So, the actions IB1, CE1, CE2, CE3, CE4, and ER1 are no more executed. All the illegal branches delete themselves by executing IB2, and no new one will be created (only CE4 can create a new illegal root). No more processor will have the E status. Eventually, the system will reach A_7. In A_7, only the token circulation actions may be executed. Thus, A_7 is closed. □