Self-Stabilizing Depth-First Token Circulation In Arbitrary Rooted Networks *

Ajoy K. Datta,[1] Colette Johnen,[2] Franck Petit,[3]
Vincent Villain[3]
[1] Department of Computer Science, University of Nevada, Las Vegas
[2] L.R.I./C.N.R.S., Université de Paris-Sud, France
[3] LaRIA, Université de Picardie Jules Verne, France

Abstract: We present a deterministic distributed depth-first token passing protocol on a rooted network. This protocol uses neither the processor identifiers nor the size of the network, but assumes the existence of a distinguises hed 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. Our protocol implements a fair token circulation scheme, i.e., in every round, every processor obtains the token at least once. The proposed protocol has extremely small state requirement—only $3(\Delta+1)$ states per processor, i.e., $O(log\Delta)$ bits per processor, where Δ is the degree of the network. The protocol can be used to implement a fair distributed mutual exclusion in any rooted network. This protocol can also be used to construct a DFS spanning tree.

Keywords: Distributed 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 [Dij74] 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 depth-first token circulation problem is to implement a token circulating from one processor to the next in the depth-first order such that every processor gets the token at least once in every round (defined more formally later). In this paper, the token is initiated by the root of the network.

Related Work. Dijkstra introduced the property of self-stabilization in distributed systems by applying it to algorithms for mutual exclusion on a ring [Dij74]. Several deterministic self-stabilizing token passing algorithms for different topologies have been proposed in the literature: [BD95, BP89,

[1]Type of Submission: Research. Contact author: Vincent Villain. LaRIA, Université de Picardie Jules Verne, France. Phone: +33 3 22 82 78 74. Fax: +33 3 22 82 76 54. Email: villain@laria.u-picardie.fr.

Dij74, FDS94] for a ring; [BGW89, Gho93, GH96, ?] for a linear array of processors, and [Kru79, ?] for tree network. Huang and Chen [HC93] presented a token circulation protocol for a connected network in non-deterministic depth-first-search order, and Dolev, Israeli, and Moran [DIM93] gave a mutual exclusion protocol on a tree network under the model whose actions only allow read/write atomicity.

One of the important performance issues of self-stabilizing algorithms is the memory requirement per processor. The memory requirement of a processor depends on the total number of states of the processor. Most of the previous solutions to the token circulation problem on general networks require $O(\log n)$ bits per processor, where n is the number of processors. In these protocols, each processor maintains its *distance* to the distinguished processor. Awerbuch and Ostrovsky [AO94] and Itkis and Levin [IL94] used some special data structures to store the distance. Thus, their space complexity on each processor is $O(\log^* n)$ and $O(1)$ bits per edge, respectively.

A state-efficient token passing protocol on general network is presented in [JB95]. In this protocol, a processor p_i needs to maintain $3(\Delta_i + 1)$ states ($\lceil \log(3(\Delta_i + 1)) \rceil$), where Δ_i is the degree of p_i. Subsequently, this result was improved by Petit and Villain [PV97a] to $2(\Delta_i + 1)$ states for a processor p_i. Both of these two protocols use neither the *distance* variable nor any special data structure to achieve the low memory requirement. But, in these algorithms, a processor needs the knowledge of the state of the neighbors of its neighbors. Since the algorithms assume 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 state of the neighbors of its neighbors, and finally changes its own state. This drawback has been removed in [JABD97]. In this protocol, a processor only reads the state of its neighbors in an atomic step. Thus, this algorithm has a smaller atomicity than that in [JB95, PV97a]. The state requirement of this protocol is $12(\Delta_i + 1)$ states for a processor p_i. Petit and Villain [PV97c] and [PV97b] adapted the result of [JB95] and [PV97a], respectively, in the message passing model.

Contributions. In this paper, we present a self-stabilizing depth-first token circulation scheme on a general network with a distinguished root, called Algorithm $\mathcal{TC}$. Algorithm $\mathcal{TC}$ has all the desirable features of the algorithm in [JABD97]. In addition, we reduced the state requirement for a processor p to $3(\Delta_p + 1)$ states (only $2(\Delta_p + 1)$ states on the root). Also, our algorithm is simpler (less number of actions) than that in [JABD97]. Algorithm $\mathcal{TC}$ implements a fair circulation of token. Our algorithm can also be used to implement the distributed mutual exclusion among the processors on a rooted network.

Outline of the Paper. The token passing problem is formally defined in Section 2.2. The rest of the paper is organized as follows: In Section 2, we describe the distributed systems and the model in which our token circulation scheme is written, and give a formal statement of the token passing problem solved in this paper. In Section 3, we present the token passing protocol, and in the following section (Section 4), we give the proof of correctness of the protocol. Some of the formal proofs are moved to the appendix due to the lack of space. The state complexity of the protocol is given in Section 5. In Section 6, we discuss the fairness issues implemented by our protocol and also, the use of our protocol to implement the mutual exclusion protocol. Finally, we make concluding remarks in Section 7.

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. A *distributed system* is an undirected connected graph, $S = (V, E)$, where V is a set of nodes ($|V| = n$) and E is the set of edges. Nodes represent *processors* and edges represent *bidirectional communication links*. We consider networks which are *asynchronous* and *rooted*, i.e., all processors, except the root are *anonymous*. We denote the processors by $p :: p \in \{1..n\}$ and the root processor by r. The numbers, $1..n$, are used to identify the processors to present our ideas here, but no processor, except the root (identified by r), has any identity. A communication link (p, q) exists iff p and q are neighbors. Each processor p maintains its set of neighbors, denoted as N_p. We assume that N_p is a constant and is maintained by an underlying protocol.

Programs. Each processor executes the same program except the root r. The program consists of a set of *shared variables* (henceforth referred to as variables) and a finite set of actions. A processor can only write to its own variables and can only read its own variables and variables owned by the neighboring processors. So, the variables of p can be accessed by p and its neighbors.

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 p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates zero or more variables of p. 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 p is called a *step* of p.

The *state* of a processor is defined by the values of its variables. The *state* of a system is a product of the states of all processors ($\in V$). In the sequel, we refer to the state of a processor and system as a (*local*) *state* and *configuration*, respectively. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$, the set of all possible configurations of the system. A *computation* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0, \gamma_1, ..., \gamma_i, \gamma_{i+1}, ...)$, such that for $i \geq 0, \gamma_i \mapsto \gamma_{i+1}$ (a single *computation step*) if γ_{i+1} exists, or γ_i is a terminal configuration. *Maximality* means that the sequence is either infinite, or it is finite and no action of $\mathcal{P}$ is enabled in the final configuration. All computations considered in this paper are assumed to be fair and maximal. During a computation step, one or more processors execute a step and a processor may take at most one step. This execution model is known as the *distributed daemon* [BGM89]. We use the notation *Enable* (A, p, γ) to indicate that the guard of the action A is true at processor p in the configuration γ. A processor p is said to be *enabled* at γ $(\gamma \in \mathcal{C})$ if there exists an action A such that *Enable* (A, p, γ). We assume a *weakly fair* daemon, meaning that if a processor, p is continuously *enabled*, p will be eventually chosen by the daemon to execute an action.

The set of computations of a protocol $\mathcal{P}$ in system S starting with a particular configuration $\alpha \in \mathcal{C}$ is denoted by $\mathcal{E}_\alpha$. The set of all possible computations of $\mathcal{P}$ in system S is denoted as $\mathcal{E}$.

Predicates. $x \vdash P$ means that x satisfies the predicate P. We define a special predicate true as follows: *for any* $e \in \mathcal{E}$, $e \vdash \mathsf{true}$.

Self-Stabilization. We use the following term, *attractor* in the definition of self-stabilization.

Definition 2.1 (Attractor). *Let X and Y be two predicates of a protocol $\mathcal{P}$ defined on $\mathcal{C}$ of system $\mathcal{S}$. Y is an attractor for X if and only if the following condition is true:*
$\forall \alpha \vdash X :: \forall e \in \mathcal{E}_\alpha :: e = (\gamma_0, \gamma_1, ...) :: \exists i \geq 0, \forall j \geq i, \gamma_j \vdash Y$. *We denote this relation as $Y \triangleleft X$.*

Definition 2.2 (Self-stabilization). *The protocol $\mathcal{P}$ is self-stabilizing for the specification predicate $\mathcal{SP}$ on $\mathcal{E}$ if and only if there exists a predicate $\mathcal{L}$ defined on $\mathcal{C}$ such that the following conditions hold:*

1. $\forall \alpha \vdash \mathcal{L} :: \forall e \in \mathcal{E}_\alpha :: e \vdash \mathcal{SP}$ *(correctness).*
2. $\mathcal{L} \triangleleft$ *true (closure and convergence).*

2.2. Specification of the Depth-First Token Passing Protocol

We will use the following term to specify the protocol:

Definition 2.3 (Token Circulation Round). *We define a computation in the protocol $\mathcal{TC}$ starting from a state δ_1 to another state δ_2 as a token circulation round (in the sequel referred to as cround) if the following conditions are true:*
(i) r holds a token in both δ_1 and δ_2.
(ii) There is at least one state in between δ_1 and δ_2 such that the token is passed among the processors in the depth-first search order[1].

The legitimacy predicate $\mathcal{L}_{\mathcal{TC}}$ of our token passing protocol is any configuration such that (*i*) exactly one processor has a token at any time (called the *Single Token* property), and (*ii*) for each computation that starts in such a configuration, during a token circulation round, each processor obtains the token at least once (called the *Fairness* property).

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

3. Depth-First Token Passing Algorithm

In this section, we propose the self-stabilizing depth-first token circulation algorithm. We first present the data structure used by the processors. Then we present the formal algorithm. Next, we define some terms to be used later in the paper. We then explain the process of token circulation, followed by the method of error correction. In particular, we do not use the distance variable used in [HC93] to destroy the cycles. We use a method similar to the one introduced in [JB95] to remove the cycles in the network.

[1] We assume that the network has at least one processor other than r.

3.1. Data Structures and Algorithm $\mathcal{TC}$

To distinguish each token round, each processor p uses a variable C_p, called the *round color*, which contains a value $\in \{0, 1\}$ when the system is stabilized. A third color E, called the *Error* color, is used by processors, except the root, during the stabilization. The descendant relationship is indicated by the variable D_p ($D_p \in N_p \cup \{\perp\}$). To choose its descendant, each processor p locally distinguishes each neighbor by some ordering, denoted as $\succ_p$.

The self-stabilizing depth-first token circulation is shown in Algorithm 3.1. To make the algorithm readable, we present it in three parts: the *macros*, *predicates*, and *actions*. The macros are not variables and they are dynamically evaluated. Anc_p denotes the set of ancestors of p, i.e., $Anc_p = \{q \in N_p \mid D_q = p\}$. UV_p is the set of neighbors not visited by the token. $Search_p$ chooses the next neighbor from UV_p. In the following, $\sharp Anc_p$ denotes the current number of ancestors of p. If $\sharp Anc_p = 1$, then the only ancestor of p is denoted as a_p.

Algorithm 3.1 ($\mathcal{TC}$) Self-Stabilizing Depth-First Token Circulation in Rooted Network.

Macro

$$
\begin{aligned}
Anc_p &= \{q \in N_p : D_q = p\} \\
UV_p &= \left\{ q \in N_p : \begin{pmatrix} (q \succ_p D_p) \wedge (C_q \neq C_p) \wedge (D_q \neq p) \wedge (q \neq r) \\ \wedge\ ((C_q \neq E) \vee (D_q \neq \perp)) \end{pmatrix} \right\} \\
Search_p &= \begin{cases} \min_{\succ_p}(UV_p) \textbf{ if } (UV_p \neq \emptyset) \\ \perp \textbf{ otherwise} \end{cases}
\end{aligned}
$$

Predicates

$$
\begin{aligned}
Forward\,(p) &\equiv (D_p = \perp) \wedge \big((p = r) \vee \big((\sharp Anc_p = 1) \wedge (C_p \neq E) \wedge \big(C_{a_p} = (C_p + 1) \bmod 2\big)\big)\big) \\
Backtrack\,(p) &\equiv (D_p \neq \perp) \wedge (D_p \neq r) \wedge \big(D_{D_p} = \perp\big) \wedge \big(C_{D_p} = C_p\big) \wedge (C_p \neq E) \\
&\quad \wedge\ ((p = r) \vee (\sharp Anc_p = 1)) \\
Break\,(p) &\equiv (p \neq r) \wedge (D_p \neq \perp) \\
&\quad \wedge \begin{pmatrix} (D_p = r) \\ \vee\ \big((C_p = E) \wedge \big(\big(D_{D_p} = \perp\big) \vee \big((\sharp Anc_p > 1) \wedge \big(C_{D_p} = E\big)\big)\big)\big) \\ \vee\ \big(\big(D_{D_p} = \perp\big) \wedge (\sharp Anc_p = 0) \wedge \big(C_{D_p} \neq (C_p + 1) \bmod 2\big)\big) \end{pmatrix} \\
EDetect\,(p) &\equiv (p \neq r) \wedge (C_p \neq E) \\
&\quad \wedge \begin{pmatrix} \big((D_p \neq \perp) \wedge \big(C_{D_p} = E\big) \wedge (\sharp Anc_p = 1)\big) \\ \vee\ ((D_p \neq r) \wedge (\sharp Anc_p > 1)) \end{pmatrix} \\
EEnd\,(p) &\equiv (p \neq r) \wedge (C_p = E) \wedge (D_p = \perp) \wedge ((\sharp Anc_p = 0) \vee (Anc_p = \{r\}))
\end{aligned}
$$

Actions

$$
\begin{aligned}
TC1 &:: Forward(p) \longrightarrow C_p := (C_p + 1) \bmod 2;\ D_p := Search_p; \\
TC2 &:: Backtrack(p) \longrightarrow D_p := Search_p; \\
EC1 &:: Break\,(p) \longrightarrow D_p := \perp; \\
EC2 &:: EDetect\,(p) \longrightarrow C_p := E; \\
EC3 &:: EEnd\,(p) \longrightarrow \textbf{if } (\sharp Anc_p = 0) \textbf{ then } C_p := 0; \textbf{ else } C_p := C_r;
\end{aligned}
$$

The predicates are used to describe the guards of the actions in Algorithm 3.1. Actions $TC1$ and $TC2$ implement the *token circulation*, i.e., the correct behavior of the system. The token circulates in the network according to the Definition 2.3. Actions $EC1$, $EC2$, and $EC3$ implement the *error correction* of the system, i.e., they are used to bring the system from an illegitimate configuration to a legitimate one. All these predicates will be explained in detail in Section 3.2.

When the system stabilizes, the system must contain only one token which circulates in the DFS order. In such a configuration, a processor can make a move only if it holds the token. Hold

the token means $Forward(p)$ or $Backtrack(p)$ is true. Formally:

$$Token(p) \equiv Forward(p) \vee Backtrack(p)$$

3.2. Informal Explanation of Algorithm $\mathcal{TC}$

The proposed algorithm has two major tasks: (i) to circulate the token in the network in a deterministic depth-first order and (ii) to handle the abnormal situations (illegal configurations) due to the unpredictable initial configurations and transient errors. The tasks (i) and (ii) are explained with examples in Paragraphs **Token Circulation** and **Error Correction**, respectively.

Some Definitions. A path μ_p is a sequence $(p_1, p_2, ..., p_l)$ such that (i) $p = p_1$, (ii) $l \geq 2$, (iii) $\forall i \in [1, l[$, $D_{p_i} = p_{i+1}$ and (iv) $D_{p_l} = \perp$ or $D_{p_l} \in \{p_1, p_2, ..., p_{l-1}\}$. $\forall i \in [1, l], p_i$ is said to *belong to* the path μ_p and is denoted as $p_i \in \mu_p$.

If $Anc_p = \emptyset$, then μ_p is called a *rooted path* (the path is rooted at p). A path μ_p rooted at $p \neq r$ is called an *illegal rooted path* and p is called *illegal root*. A path μ_p rooted at r is called the *legal* (*rooted*) *path*.

The processor $p_l \in \mu_p$ is termed as a *leaf* if $D_{p_l} = \perp$. The leaf of a legal (respectively, illegal) rooted path is called *legal* (respectively, *illegal*) *leaf*. A leaf, p' is termed as a *live* (respectively, *dead*) *leaf*, if $C_{p'} \neq E$ (respectively, $C_{p'} = E$).

The path μ_p is called a *cycle* if $D_{p_l} \in \{p_1, p_2, ..., p_{l-1}\}$. A cycle μ_p is called a *strict cycle* if $\forall i \in [2, l]$, p_{i-1} is the only ancestor of p_i ($Anc_{p_i} = \{p_{i-1}\}$) and p_l is the only ancestor of p_1 ($Anc_{p_1} = \{p_l\}$). Otherwise, there exists at least one rooted path μ_q such that all processors in the cycle μ_p belong to μ_q i.e., $\exists p_i \in \mu_p$ such that $i \in [2, l]$ and $\sharp Anc_{p_i} > 1$. Such a rooted path μ_p is called a *rooted cycle*.

A rooted path with a live leaf is termed as a *live rooted path*. All other rooted paths are called *dead rooted paths*.

Every processor p such that $Anc_p = \emptyset$ and $D_p = \perp$ is called *path-free*, meaning p does not belong to any path.

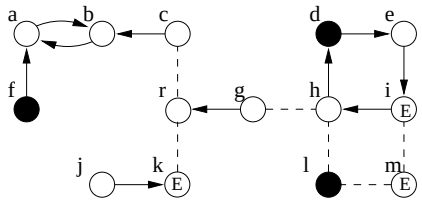

Figure 3.1: A Possible Configuration.

These definitions are illustrated in Figure 3.1. Processors c, f, g, and j are illegal roots. d, e, i, h form a strict cycle. f and c are roots of a rooted cycle. k is a dead leaf. l and m are path-free.

Token Circulation. The root r initiates the token circulation round. The token then traverses all processors during a token circulation round (Definition 2.3).

We use $\delta_{c0} \in \mathcal{C}$ to denote a configuration where every processor in the system is path-free and has the color 0. Similarly, δ_{c1} denotes the configuration in which every processor is path-free and has the color 1. Both δ_{c0} and δ_{c1} are among the possible configurations from where the algorithm behaves correctly, i.e., starting from δ_{c0} (respectively, from δ_{c1}), Algorithm $\mathcal{TC}$ circulates the token (represented by the predicate $Token()$) in the depth-first search order to reach δ_{c1} (respectively, δ_{c0}). This is called one token circulation round (*cround*). ¿From δ_{c1} (respectively, δ_{c0}), the system reaches δ_{c0} (respectively, δ_{c1}) again in the same manner. After stabilization, the system repeats the *crounds* forever. The *cround* is implemented by Actions $TC1$ and $TC2$. Every suffix of the computation starting from δ_{c0} or δ_{c1} is a legitimate configuration.

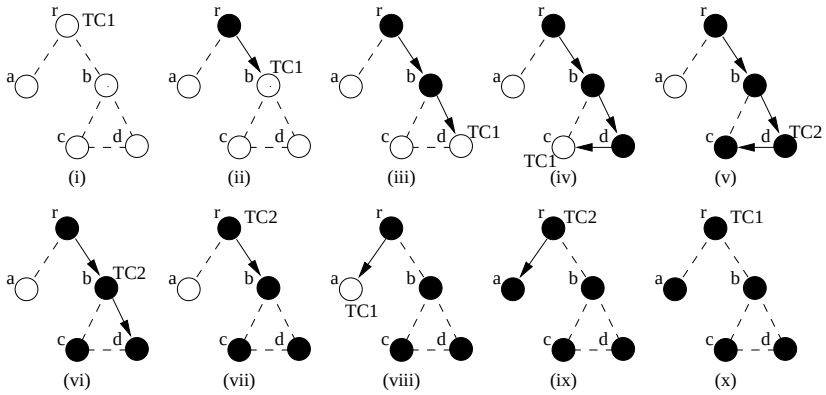

Figure 3.2: Depth-First Search Token Circulation.

Consider the example in Figure 3.2. Step (i) corresponds to the configuration δ_{c0}. In this configuration, $Forward(r)$ is true and the only process *enabled* is r and the only action enabled at r is $TC1$. The root changes its color ($C_r := (C_r + 1) \bmod 2$) and builds μ_r by choosing a descendant (*Search* predicate). The root chooses the processor b as the descendant. This is shown in Step (ii). Similarly, b changes its color and chooses a descendant (Step (iii)). This process of extending the path continues until c executes Action $TC1$. c does not have any neighbor to choose from. So, c executes $D_c := \perp$ (*Search*). This indicates to its ancestor d that the token has traversed all nodes reachable from c in the DFS tree (Step (v)). Now, $Backtrack(d)$ becomes true and d can execute $TC2$. Since d has no more unvisited neighbors, D_d becomes equal to $\perp$ (Step (vi)). Actions $TC1$ and $TC2$ are repeated until all processors are visited by the token (Steps (vii) to (x)). Step (x) corresponds to δ_{c1}. Now, r changes its color to 0 and starts a new round with this color.

Error Correction. We now consider the transient failures. An example of an illegitimate configuration was shown in Figure 3.1.

Actions $EC1$, $EC2$ and $EC3$ are used to bring the system into a legitimate configuration. Illegal configurations are locally detected by the predicates *Break*, *EDetect*, and *EEnd*. We split the predicates *Break* and *EDetect* into simpler predicates in Figure 3.3 to help explain them better.

First consider the illegal configuration in which r has ancestors. For every ancestor p of r, p satisfies $BrkA$ ($D_p = r$) and hence, $Break(p)$. Upon executing Action $EC1$, p eventually destroys

$$
\begin{array}{rcl}
BrkA\,(p) & \equiv & (D_p = r) \\
BrkB\,(p) & \equiv & (C_p = E) \wedge (\sharp Anc_p > 1) \wedge \left(C_{D_p} = E\right) \\
BrkC\,(p) & \equiv & (C_p = E) \wedge \left(D_{D_p} = \bot\right) \\
BrkD\,(p) & \equiv & \left(D_{D_p} = \bot\right) \wedge (\sharp Anc_p = 0) \wedge \left(C_{D_p} \neq (C_p + 1) \bmod 2\right) \\
Break\,(p) & \equiv & (p \neq r) \wedge (D_p \neq \bot) \wedge (BrkA\,(p) \vee BrkB\,(p) \vee BrkC\,(p) \vee BrkD\,(p)) \\
EDetectA\,(p) & \equiv & (D_p \neq \bot) \wedge \left(C_{D_p} = E\right) \wedge (\sharp Anc_p = 1) \\
EDetectB\,(p) & \equiv & (D_p \neq r) \wedge (\sharp Anc_p > 1) \\
EDetect\,(p) & \equiv & (p \neq r) \wedge (C_p \neq E) \wedge (EDetectA\,(p) \vee EDetectB\,(p))
\end{array}
$$

Figure 3.3: Predicates $Break$ and $EDetect$.

the descendant pointer to r and since, r cannot be chosen as descendant in the algorithm (see macro UV_p, $q \neq r$), r eventually does not belong to any illegal paths.

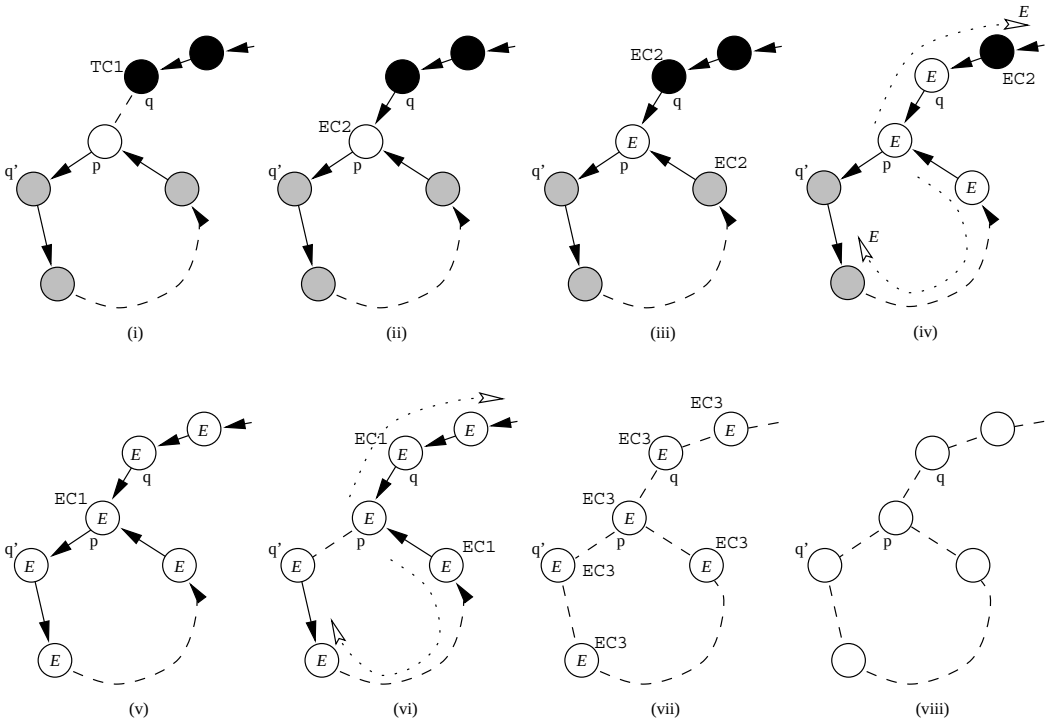

Figure 3.4: Cycles Destruction.

The Strict cycles are destroyed by the token circulation mechanism. We explain this strategy using the example in Figure 3.4. In the configuration in Step (i), p belongs to a cycle. The grey processors in the figure can have any color. Assume that p is 0-colored. The token circulation consists of successive *crounds*, alternately colored with 0 and 1. In the next *cround*, p is eventually chosen as a descendant by one of its neighbors. Let q be that neighbor. q eventually executes the macro $Search_q$ (Action $TC1$ or $TC2$) and chooses p as the descendant. This is shown in Step (ii). In this configuration, p detects that it has more than one ancestor ($EDetectB(p)$ is true) and executes $EC2$ to become a E-colored processor (Step (iii)).

The key point of our strategy is that the color E is propagated along the *backtrack* path, i.e., the color E is propagated from a descendant to its ancestors. In our example, for each ancestor q of p, $EDetectA(q)$ is true. The ancestors of the E colored processors execute $EC2$ to implement the propagation of the color E through all the paths attached to q (Steps (iii) and (iv)).

Since p belongs to a cycle, its descendant (q' in the figure is eventually E-colored (Step (v)). p then satisfies $BrkB$, executes $EC1$, and detaches q' to break the cycle (Step (vi)). Next, for every ancestor $q \neq r$ of p, either $BrkD(q)$ or $BrkC(q)$ becomes true depending on q is an illegal root or not a root. Every E-colored path is eventually self-destroyed using $EC1$ (Step (vii)). Finally, the E-colored, path-free processors are made 0-colored by Action $EC3$ (Step (viii)).

It is easy to see that the destruction of the rooted cycles is implemented using the same mechanism (Step (ii) and the following steps).

Finally, the protocol must destroy all illegal rooted paths. If a rooted path μ_p is a rooted cycle, it is destroyed using the cycle destruction mechanism described above. Otherwise, it has a live (not E-colored) or dead (E-colored) leaf. In the first case, μ_p is self-destroyed as above (Step (vi) and the following steps). In the second case, μ_p is self-destroyed by just allowing the token to circulate. When p executes $Search_p$, D_p strictly increases. So, every processor belonging to μ_p will eventually be without any descendant and $BrkD(p)$ becomes true. p then executes $EC1$ and the illegal rooted path μ_p is destroyed.

4. Correctness of the Token Passing Protocol $\mathcal{TP}$

We apply the convergence stair method [GM91] to prove our protocol. We exhibit a finite sequence of state predicates $\mathcal{A}_0, \mathcal{A}_1, \ldots, \mathcal{A}_m$, of Protocol $\mathcal{TP}$ such that the following conditions hold:

(i) $\mathcal{A}_0 \equiv true$ (meaning any arbitrary state)

(ii) $(\mathcal{A}_m \equiv \mathcal{L}_{TP}) \vee (\mathcal{A}_m \vdash \mathcal{L}_{TP})$

(iii) $\forall j : \ 0 \leq j < m :: \ \mathcal{A}_{j+1} \triangleleft \mathcal{A}_j$

The proof outline is as follows:

In Section 4.1, we show that eventually no processor has the root as a descendant. Then, we prove that a locked processor (which never executes any action) cannot be the root, and either it does not have any descendant, or it belongs to a strict cycle (Section 4.2). This last result trivial leads to the proof of liveness of the algorithm (Section 4.3). It also implies that all illegal live rooted paths are eventually destroyed (Section 4.4). That amounts to the fact that, once no live rooted path exists, the system contains only one token. In Section 4.5, we show that as the root changes its color infinitely often, the legal path is eventually colored with the color of the root. Then in Section 4.6, we prove that all cycles are eventually detected and destroyed. Finally, in Section 4.7, we prove that the system reaches a configuration which satisfies $\mathcal{L}_{\mathcal{TC}}$. We give some proofs in the appendix due to the lack of space.

4.1. Root Without An Ancestor

In this section, we show that the system trivially reaches a configuration in which r does not have any ancestor.

We define $\mathcal{A}_1 \equiv (\forall p \neq r : D_p \neq r)$.

Theorem 4.1. $\mathcal{A}_1 \triangleleft \mathcal{A}_0$.

Proof: $\mathcal{A}_1$ is closed: The root r can not be chosen as a descendant by a process $p \neq r$ (see macro UV_p, $q \neq r$). Hence, $\sharp Anc_r$ cannot increase.

Every computation leads to $\mathcal{A}_1$: $(\forall p \in \{1..n\}, \forall \alpha \in \mathcal{C} :: D_p = r) \Rightarrow Enable\,(EC1, p, \alpha)$. p executes $EC1$ in the configuration α or β, where $\alpha \mapsto \beta$ and $Enable\,(EC1, p, \beta)$. By fairness, $\exists \beta : \alpha \rightsquigarrow \beta$ such that p executes $EC1$. Hence, $\sharp Anc_r$ decreases. Since $\sharp Anc_r$ cannot increase, $\exists \gamma \in \mathcal{C} : \alpha \rightsquigarrow \gamma :: Anc_r = \emptyset$. □

4.2. Properties of Locked Processors

We need the following term throughout this section:

A processor p is said to be *Locked* in a configuration α, if in all configuration reachable from α, C_p and D_p are constants. Formally:

> $Locked(p, \alpha) \equiv (\forall \beta : \alpha \rightsquigarrow \beta :: C_{p_\alpha} = C_{p_\beta} \wedge D_{p_\alpha} = D_{p_\beta}$, where V_{p_γ} denotes the value of V_p in the configuration $\gamma)$

Since the daemon is weakly fair, $Locked\,(p, \alpha)$ implies that p is not continuously *enabled* in all configurations reachable from α.

We now prove that a locked processor can not be the root, and either it does not have any descendant, or it belongs to a cycle.

Lemma 4.2. $\forall p, q \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ (Locked\,(p, \alpha) \wedge (D_p = q)) \Rightarrow (\exists\, \beta :\ \alpha \rightsquigarrow \beta ::\ Locked\,(q, \beta))$.

Proof Outline: We prove this by contradiction. If q is not *Locked*, then q eventually executes an action such that q has no descendant and has the same color as p, or the color E. Thus, p would be *enabled* forever, which contradicts the hypothesis.

For the detail proof, please see Lemma A.1 in Appendix A.

Lemma 4.3. $\forall p, q \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ (Locked\,(p, \alpha) \wedge (D_p = q) \wedge (p \neq r)) \Rightarrow$
$(\exists\, \beta :\ \alpha \rightsquigarrow \beta :\ \forall \beta' : \beta \rightsquigarrow \beta' ::\ D_q \neq \perp)$

Proof Outline: Following the similar reasoning as in the previous lemma, we can show that q will eventually reach a state where it has a descendant and will maintain it forever.

For the detail proof, please see Lemma A.2 in Appendix A.

Lemma 4.4. $\forall p \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ Locked\,(p, \alpha) \Rightarrow p$ *does not belong to a rooted cycle.*

Proof Outline: We prove this lemma by contradiction. If p belongs to a rooted cycle, it is clear that a processor q in that cycle has at least two ancestors. The processor q will eventually get the color E and the color E will be propagated along the cycle in the direction from the descendants towards the ancestors. So, q's descendant will also be E-colored. Then, q will break the cycle. By induction, every process in the rooted cycle will eventually detach its descendant. Thus, p is not *Locked*.

For the detail proof, please see Lemma A.3 in Appendix A.

Theorem 4.5. $\forall p \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ (\mathit{Locked}\,(p,\alpha) \wedge (p \neq r))$
$\Rightarrow ((D_p = \perp) \vee (p \textit{ belongs to a strict cycle}))$.

Proof: Assume that $D_p = q$. By Lemmas 4.2 and 4.3, $\exists \beta : \alpha \rightsquigarrow \beta ::\ \mathit{Locked}\,(q,\beta)$ and $D_q \neq \perp$. By induction, the descendant of q will also be eventually locked, and so on. Since the graph S is finite, p belongs to a cycle. By Lemma 4.4, p cannot belong to a rooted cycle. Thus, p belongs to a strict cycle. □

Theorem 4.6. $\forall p \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ \mathit{Locked}\,(p,\alpha) \Rightarrow (p \neq r)$

Proof Outline: In order to be *Locked*, r must be in a strict cycle (see Theorem 4.5). So, r must have an ancestor, which cannot be true in any configuration $\vdash \mathcal{A}_1$.

For the detail proof, please see Theorem A.4 in Appendix A.

4.3. Liveness

The following Lemma follows directly from Theorem 4.6:

Lemma 4.7 (liveness). *In any configuration $\vdash \mathcal{A}_1$, at least one processor is enabled.*

4.4. Destruction of Live Illegal Rooted Paths

In this section, we show that all live illegal rooted paths are destroyed.

Lemma 4.8. *Every illegal rooted path is eventually destroyed, or becomes a strict cycle.*

Proof: Assume the contrary, i.e., there exists an illegal rooted path μ_p that does not disappear and also does not become a cycle. Then, p is never chosen as a descendant. Otherwise, p is not an illegal root and μ_p is destroyed. By Theorem 4.5, p is not *Locked*. So, there exists a configuration $\alpha \vdash \mathcal{A}_1$ such that p executes an action. So, p executes $EC1$ at α (only $Break(p)$ can be true). After Action $EC1$ is executed, the illegal rooted path rooted at p disappears, which contradicts our assumption. □

Let us denote the number of live illegal leaves by LIL.

Lemma 4.9. *$\forall \alpha \vdash \mathcal{A}_1$, $\forall \beta$ such that $\alpha \mapsto \beta$, the value of LIL in β is less than or equal to the value of LIL at α.*

Proof Outline: Algorithm $\mathcal{TC}$ cannot create a new live illegal leaf for the following reasons: (i) A dead illegal leaf cannot become a live illegal leaf. (ii) A path cannot be split creating a live illegal leaf. (iii) A path-free processor cannot create a new illegal rooted path.

For the detail proof, please see Lemma B.1 in Appendix B.

We define $\mathcal{A}_1 \ \equiv\ \mathcal{A}_1 \wedge (LIL = 0)$.

Theorem 4.10. $\mathcal{A}_2 \triangleleft \mathcal{A}_1$.

Proof: Follows from Lemmas 4.9 and 4.8. □

Corollary 4.11. *In any configuration $\vdash \mathcal{A}_2$, if there exists a live leaf, then it must be a legal leaf.*

4.5. Color Consistency

In this section, we show that eventually, either the system contains no live leaf, or every processor in the legal path (except the leaf) has the same color as r has. In such a configuration, the legal path cannot create a new cycle.

We define a predicate $ColorConsistent$ in a configuration γ such that it is true if any of the following conditions is true: $(CC1)$ $D_r = \perp$, $(CC2)$ The leaf of the legal path is a live leaf and all processors on the legal path, except the leaf, are *r-colored* (with the same color as r), $(CC3)$ The legal path does not have a leaf (rooted cycle), or has a dead leaf.

We define $\mathcal{A}_3 \equiv \mathcal{A}_2 \wedge ColorConsistent$.

Theorem 4.12. $\mathcal{A}_3 \triangleleft \mathcal{A}_2$.

Proof Outline: By Theorem 4.6, r executes its actions infinitely often. So, r starts a new *cround* with a new color infinitely often. If the legal path, μ_r does not meet any illegal path, then it remains color consistent (all processors, except the leaf, have the same color). Otherwise, when μ_r meets an illegal path, its leaf becomes dead and it remains color consistent.

For the detail proof, please see Theorem C.2 in Appendix C.

4.6. Cycle Destruction

In this section, we prove that all cycles are eventually destroyed. The process of destruction is as follows: All strict cycles are merged with the legal path and thus, become rooted cycles. Then by the repeated application of $EC1$ and $EC2$, the rooted cycles will be destroyed.

The *first DFS tree* [CD94] of the graph G is defined as the DFS spanning tree rooted at r, created by traversing the graph in the DFS manner, and visiting the adjacent edges of every processor in the order induced by $\succ_p$. We defined the macro $Search_p$ such that Algorithm $\mathcal{TC}$ circulates the token in the first DFS tree.

Lemma 4.13. *Starting from any configuration $\vdash \mathcal{A}_3$, all nodes which do not belong, either to the legal path, or to any strict cycles, will be eventually path-free.*

Proof: *Follows directly from Lemma 4.8.* □

Lemma 4.14. *Starting from any configuration $\vdash \mathcal{A}_3$, every processor which is path-free and E-colored, will be eventually path-free and 0-colored.*

Proof: *By fairness, all E-colored and path-free processors eventually execute $EC3$ because none of its neighbors can choose it as a descendant (see UV_p, q cannot be chosen if $C_q = E$ and $D_q = \perp$).* □

Lemma 4.15. *Starting from any configuration $\vdash \mathcal{A}_3$, every strict cycle will be eventually transformed into a rooted cycle.*

Proof: By Lemma 4.8, for all configurations $\vdash \mathcal{A}_3$, there exists no live leaf of an illegal rooted path. So, our responsibility is to show that eventually a node on every strict cycle in the system

will be selected as a descendant by the leaf of the legal path. Assume the contrary, i.e., there exists one strict cycle which will never be reached by the legal path.

So, there exists $\alpha \vdash \mathcal{A}_3$ such that all processors between r and the strict cycle on the first DFS-tree are path-free (by Lemma 4.13), or they belong to the legal path (r-colored in $\mathcal{A}_3$). By Lemma 4.14, $\exists \alpha' : \alpha \rightsquigarrow \alpha'$ such that every processor between r and the strict cycle is $0-$ or 1-colored. Also, by successive *crounds*, $\exists \alpha'' : \alpha' \rightsquigarrow \alpha''$ such that every processor between r and the strict cycle has the same color k (0 or 1).

Let q be the first processor in the strict cycle that is on the first DFS tree. Let $p \in N_q$ be the ancestor of q in the first DFS tree. Since no strict cycle is reachable by the legal path (by assumption), $C_q = k$. Otherwise, p will eventually select q as a descendant, which will contradict our assumption. But, in the next *cround*, p will choose q as a descendant because C_p will be equal to $(k+1) \bmod 2$. Thus, we arrive at the contradiction. □

Lemma 4.16. *Starting from any configuration $\vdash \mathcal{A}_3$, every cycle is destroyed.*

Proof: By Lemma 4.15, every strict cycle is eventually transformed into a rooted cycle. By Lemma 4.8, every rooted cycle is eventually destroyed. □

Let NC denote the number of cycles in the system.
We define $\mathcal{A}_4 \equiv \mathcal{A}_3 \wedge (NC = 0)$.

Theorem 4.17. $\mathcal{A}_4 \triangleleft \mathcal{A}_3$.

Proof: $\mathcal{A}_4$ is closed: All processors which belong to the legal path have the same color. So, by the definition of $Search$, $Forward$, and Action $TC1$, the leaf of the legal path chooses a descendant of a different color. So, no new cycle can be created in $\mathcal{A}_3$. Hence, NC cannot increase.

Every computation starting from a configuration in $\mathcal{A}_3$ leads to a state in $\mathcal{A}_4$: Follows from Lemma 4.16. □

4.7. Legitimacy Predicate

It is easy to prove that the legitimacy predicate $\mathcal{L}_{\mathcal{TC}}$ eventually holds, i.e., exactly one processor has a token at any time (Single Token Property) and for each computation that starts in such a configuration, during a token circulation round, each processor obtains the token at least once (Fairness Property).

For the detail proofs, please refer to Appendix D.

5. State Complexity

A processor p in Algorithm $\mathcal{TC}$ uses two variables, D_p and C_p. The variable C_p, for a processor $p \neq r$, can have 3 different values (0, 1, and E), whereas C_r can have only 2 values (0 or 1). The variable D_p can have Δ_p ($|N_p|$) plus one ($\perp$) values. So, a processor, $p \neq r$, needs to maintain $3 \times (\Delta_p + 1)$ states and r needs $2 \times (\Delta_r + 1)$. Thus, the total number of configurations of the whole network is

$$2\,(\Delta_r + 1) \times \prod_{p \in [1..n], p \neq r} 3\,(\Delta_p + 1)$$

It is worth mentioning here that all the previous papers computed the space complexity in terms of the number of *bits* only, not in terms of the *states*. We feel that the measurement in terms of the number of states is more accurate.

6. Fairness and Mutual Exclusion

The token circulation problem is similar to the mutual exclusion problem. A solution to the problem of mutual exclusion in a network is to implement a token circulating from one processor to the next following some pattern. The token moves around the network. A processor having the token is granted access to the shared resource and can execute the code in the critical section.

Our solution to the depth-first token circulation problem can be used to solve the *mutual exclusion problem*. After stabilization, in Algorithm $\mathcal{TC}$, in each token circulation round, a processor p holds the token as many times as its degree Δ_p—once while satisfying $Forward(p)$ and $\Delta_p - 1$ while $Backtrack(p)$ is true. Since the degrees of the processors in the network may not be the same, Algorithm $\mathcal{TC}$ may not implement a *strictly fair* token circulation (and mutual exclusion). By *strict fairness*, we mean that in every round, all processors will obtain the token (enjoy the critical section access) *exactly once*. But, it is easy to implement the strict fairness in Algorithm $\mathcal{TC}$ as follows: A processor p can use the token (access the critical section) if and only if $Forward(p)$ is true. Then, in each token circulation round, each processor obtains the token exactly once.

7. Conclusions

We presented a self-stabilizing depth-first token circulation scheme on a general network with a distinguished root. Algorithm $\mathcal{TC}$ and its proof are much simpler than the earlier algorithm [JABD97]. Our algorithm implements a fair token circulation. This algorithm can be used to implement a fair distributed mutual exclusion algorithm. Our algorithm can also be used to construct a DFS spanning tree just by maintaining the descendant pointers instead of destroying them.

References

[AO94] B Awerbuch and R Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC94 Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 254–263, 1994.

[BD95] J Beauquier and O Debas. An optimal self-stabilizing algorithm for mutual exclusion on bidirectional non uniform rings. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 17.1–17.13, 1995.

[BGM89] JE Burns, MG Gouda, and RE Miller. On relaxing interleaving assumptions. In *Proceedings of the MCC Workshop on Self-Stabilizing Systems, MCC Technical Report No. STP-379-89*, 1989.

[BGW89] GM Brown, MG Gouda, and CL Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38:845–852, 1989.

[BP89] JE Burns and J Pachl. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*, 11:330–344, 1989.

[CD94] Z Collin and S Dolev. Self-stabilizing depth-first search. *Information Processing Letters*, 49:297–301, 1994.

[Dij74] EW Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[DIM93] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[FDS94] M Flatebo, AK Datta, and AA Schoone. Self-stabilizing multi-token rings. *Distributed Computing*, 8:133–142, 1994.

[GH96] MG Gouda and FF Haddix. The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*, 35:43–48, 1996.

[Gho93] S Ghosh. An alternative solution to a problem on self-stabilization. *ACM Transactions on Programming Languages and Systems*, 15:735–742, 1993.

[GM91] MG Gouda and N Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.

[HC93] ST Huang and NS Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

[IL94] G Itkis and L Levin. Fast and lean self-stabilizing asynchronous protocols. In *FOCS94 Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 226–239, 1994.

[JABD97] C Johnen, G Alari, J Beauquier, and AK Datta. Self-stabilizing depth-first token passing on rooted networks. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings, Springer-Verlag LNCS:1320*, pages 260–274, 1997.

[JB95] C Johnen and J Beauquier. Space-efficient distributed self-stabilizing depth-first token circulation. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 4.1–4.15, 1995.

[Kru79] HSM Kruijer. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8:91–95, 1979.

[PV97a] F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation. In *I-SPAN'97, Third International Symposium on Parallel Architectures, Algorithms and Networks Proceedings, IEEE Computer Society Press*. IEEE Computer Society Press, 1997. To appear.

[PV97b] F Petit and V Villain. Color optimal self-stabilizing depth-first token circulation protocol for asynchronous message-passing. In *PDCS-97 10th International Conference on Parallel and Distributed Computing Systems Proceedings*, pages 227–233. International Society for Computers and Their Applications, 1997.

[PV97c] F Petit and V Villain. A space-efficient and self-stabilizing depth-first token circulation protocol for asynchronous message-passing systems. In *Euro-par'97 Parallel Processing, Proceedings LNCS:1300*, pages 476–479. Springer-Verlag, 1997.

A. Properties of Locked Processors

Lemma A.1. $\forall p, q \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ (Locked\,(p, \alpha) \wedge (D_p = q)) \Rightarrow (\exists\, \beta :\ \alpha \rightsquigarrow \beta ::\ Locked\,(q, \beta))$.

Proof: We will prove this by contradiction. We assume the contrary, i.e.,
$\exists p, q \in 1..n, \exists \alpha \vdash \mathcal{A}_1 :\ (Locked\,(p, \alpha) \wedge (D_p = q)) \wedge (\forall\, \beta :\ \alpha \rightsquigarrow \beta ::\ \neg Locked\,(q, \beta))$.
As p is locked, $\forall \beta : \alpha \rightsquigarrow \beta :: \ p \in Anc_q$.

1. Assume that $\exists \beta :\ \alpha \rightsquigarrow \beta :: C_q = E$. Thus, for all $\beta' : \beta \rightsquigarrow \beta'$, q can execute only $EC1$ and $EC3$.

   a Assume that $D_q = \perp$ at β. Then, q can execute only $EC3$ in β. Anc_q does not increase while q does not execute $EC3$ because no neighbor of q can select q (see macro UV_p). Since q is not locked, $\exists \beta' :\ \beta \rightsquigarrow \beta' ::\ Enable(EC3, q, \beta')$. Since $p \in Anc_q$, $Enable(EC3, q, \beta')$ implies $[p = r$ and $Anc_q = \{r\}]$ in β'. Execution of $EC3$ makes $[C_p := C_r$ and $Enable(TC2, r, \beta')]$ ($Backtrack(r)$ is true). In all β'' such that $\beta' \rightsquigarrow \beta''$, $Enable(TC2, p, \beta'')$. By fairness, p eventually runs $TC2$ which contradicts the assumption, $Locked(p, \alpha)$.

   b Assume that $D_q \neq \perp$ in β. Then, q can execute $EC1$ only in β. Since q is not locked, $\exists \beta' : \beta \rightsquigarrow \beta'$ such that q eventually executes $EC1$ in β'. After the execution of $EC1$ by q, $D_q = \perp$. Thus, we arrive at the assumed state of Case 1a.

2. Assume that $\forall \beta :\ \alpha \rightsquigarrow \beta ::\ C_q \neq E$. Thus, $\forall \beta' : \beta \rightsquigarrow \beta'$, q will not execute $EC1$, $EC2$, and $EC3$.

   a Assume that q executes $TC2$ infinitely often. Since D_q strictly increases with respect to $\succ_q$ (see macro $Search_p$), $\exists \beta :\ \alpha \rightsquigarrow \beta ::\ D_q = \perp$. In this case, $\forall \beta' :\ \beta \rightsquigarrow \beta'$, $Enable(TC2, p, \beta')$, $Enable(EC1, p, \beta')$, or $Enable(EC2, p, \beta')$ depending on $\sharp Anc_p$ and C_p. By fairness, p eventually executes $TC2$, $EC1$, or $EC2$, which contradicts the assumption, $Locked(p, \alpha)$.

   b Assume that q executes $TC2$ a finite number of times only. So, $\exists \beta : \alpha \rightsquigarrow \beta$ after which q executes only $TC1$. But, after the execution of $TC1$, $C_q = C_p$. If $D_q = \perp$, then the system reaches Case 2a. If $D_q \neq \perp$, then q is locked and we prove the contradiction.

□

Lemma A.2. $\forall p, q \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ (Locked\,(p, \alpha) \wedge (D_p = q) \wedge (p \neq r)) \Rightarrow$
$(\exists\, \beta :\ \alpha \rightsquigarrow \beta :\ \forall \beta' : \beta \rightsquigarrow \beta' ::\ D_q \neq \perp)$

Proof: By Lemma 4.2, $\exists \beta : \alpha \rightsquigarrow \beta :: \ Locked(q, \beta)$. So, $\forall \beta' : \beta \rightsquigarrow \beta'$, D_q remains unchanged.
Assume the contrary, i.e., $D_q = \perp$ in β. We will consider the following cases to arrive at the contradiction.

1. Assume that $C_p = E$. Then, irrespective of the color of q, $\forall \beta' : \beta \rightsquigarrow \beta' :: Break(p)$ ($C_p = E$ and $D_{D_p} = \perp$). By fairness, p eventually executes $EC1$. But, that is not possible since p is locked.

2. Assume that $C_p \neq E$.

    1. Assume that $C_q = E$. Then $\forall \beta' : \beta \rightsquigarrow \beta'$, $EDetect(p)$ or $Break(p)$ is true depending on the value of $\sharp Anc_p$. By fairness, p eventually executes either $EC1$ or $EC2$, which is not possible since p is locked.
    2. Assume that $C_q = C_p \neq E$. Then, $\forall \beta' : \beta \rightsquigarrow \beta'$, $Backtrack(p)$, $Break(p)$, or $EDetect(p)$ is true, if $\sharp Anc_p = 1, 0$, or > 1, respectively. By fairness, p eventually executes $TC2$, $EC1$, or $EC2$, which is not possible since p is locked.

3. Assume that $C_q = (C_p + 1) \bmod 2$. Then either $[\exists \beta' : \beta \rightsquigarrow \beta' :: \sharp Anc_q > 1]$, or $[\forall \beta' : \beta \rightsquigarrow \beta' :: Anc_q = \{p\}]$. If $\sharp Anc_q = 1$, then $Enable(TC1, q, \beta')$. If $\sharp Anc_q > 1$, $Enable(EC2, q, \beta')$. By fairness, q eventually executes either $TC1$ or $EC2$, both of which are not possible since q is locked.

□

Lemma A.3. $\forall p \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ Locked\,(p, \alpha) \Rightarrow p$ *does not belong to a rooted cycle.*

Proof: Let $\mu_q = (p_1, p_2, ..., p_{l-1}, p_l)$ be a rooted cycle, where $p \in \mu_q$ and $D_{p_l} = p_i,\ i \in [2, l[$ ($\sharp Anc_{p_i} > 1$). We will prove this lemma by contradiction by assuming the contrary, i.e., there is a processor p such that $Locked(p, \alpha)$ is true and p belongs to the rooted cycle μ_p.

1. Assume that $p = p_i$ and $C_{p_i} \neq E$. Then $EDetect(p_i)$ is true. Thus, $Enable(EC2, p_i, \alpha)$ and $\forall \beta : \alpha \rightsquigarrow \beta$, $Enable(EC2, p_i, \beta)$ remains true because no action allows the ancestor of p_i to detach it because $D_p \neq \perp$. By fairness, p_i eventually executes $EC2$ which contradicts the assumption (p is *Locked*).

2. Assume that $p = p_i$ and $C_p = E$. Since p is *Locked* (according to our assumption), by Lemma 4.2, all processors $p_j, j \in [i, l]$, also are *Locked*.

    a Assume that $C_j \neq E$, $D_j = i$, and $j \in\ ]i, l]$. Then $EDetect(p_j)$ is true. Thus, $Enable(EC2, p_j, \alpha)$, and $\forall \beta : \alpha \rightsquigarrow \beta$, $Enable(EC2, p_j, \beta)$ remains true while p_j does not execute $EC2$, which contradicts the assumption by fairness. Following the similar reasoning, we can show that all processors $\forall j \in [i, l], p_j$, must have the color E to satisfy our assumption.

    b Following the similar reasoning as in Case 2a, all processors $\forall j \in [i, l], p_j$, must have the color E to satisfy our assumption. Then $Break(p_i)$ is true. Thus, $Enable(EC1, p_i, \alpha)$, and remains true while p_i does not execute $EC1$, which contradicts our assumption by fairness.

3. From Cases 1, 2a, and 2b, $p \neq p_i$, i.e., p_i cannot be *Locked*. Assume that $p = p_j, j \in]i, l]$. Then by Lemma 4.2, all processors $p_k, k \in [i, l]$, are also *Locked*, which contradicts the fact that p_i is not *Locked*. Thus, all processors $p_j, j \in [i, l]$, are not *Locked*.

4. Assume that $p = p_j, j \in [1, i-1]$. Then by Lemma 4.2, all processors $p_k, k \in [j+1, l]$, are also *Locked*, which is not possible according to Case 3.

□

Theorem A.4. $\forall p \in 1..n,\ \forall \alpha \vdash \mathcal{A}_1 :\ Locked(p, \alpha) \Rightarrow (p \neq r)$

Proof: We will prove by contradiction. Assume that $\exists \alpha \vdash \mathcal{A}_1 :\ Locked(p, \alpha) \wedge p = r$.

1. Assume that r has no descendant. Then, $\forall \beta : \alpha \rightsquigarrow \beta ::\ Enable(TC1, r, \alpha)$. Thus, by fairness, r is not *Locked*.

2. Assume that r has a descendant, q. Then by Lemma 4.2, q is also *Locked*. So, by Theorem 4.5, q is either inside a strict cycle or $D_q = \perp$.

   a $D_q = \perp$.

   (i) Assume that $\sharp Anc_q = 1$ ($Anc_q = \{r\}$). If $C_q = E$, then $\forall \beta : \alpha \rightsquigarrow \beta ::\ Enable(EC3, q, \beta)$. So, by fairness, q is not *Locked*. If $C_q \neq E$, then $\forall \beta : \alpha \rightsquigarrow \beta ::\ Enable(TC1, q, \beta) \vee Enable(TC2, r, \beta)$. Thus, by fairness, either q or r is not *Locked*.

   (ii) Assume that $\sharp Anc_q > 1$. Assume that $\exists \beta : \alpha \rightsquigarrow \beta ::\ \sharp Anc_q = 1$. Thus, $Anc_q = \{r\}$ and by Case 2a(i), this is not possible. So, $\exists \beta : \alpha \rightsquigarrow \beta ::\ \sharp Anc_q = 1$. If $C_q \neq E$, then $Enable(EC2, q, \beta)$ and q is not *Locked*. If $C_q = E$, then the ancestors of q ($\neq r$) can only execute Actions $EC1$ or $EC2$ until q remains their descendent. These ancestors of q can execute $EC2$ at most once (to get the color E). After this execution of $EC2$, the ancestors can only execute $EC1$. Because $C_q = E$ and $D_q = \perp$, it cannot get a new ancestor. Thus, after repeated execution of $EC1$, eventually, q will have no ancestors except r. This contradicts the assumption.

   b q has a descendant and is inside a strict cycle. Since $\forall \alpha \vdash \mathcal{A}_1$, r has no ancestor, q must belong to a rooted cycle (Theorem 4.1), which contradicts the assumption.

□

B. Destruction of Live Illegal Rooted Paths

Lemma B.1. *$\forall \alpha \vdash \mathcal{A}_1$, $\forall \beta$ such that $\alpha \mapsto \beta$, the value of LIL in β is less than or equal to the value of LIL at α.*

Proof: Assume the contrary, i.e., LIL in β is greater than LIL in α. Then one of the following is true: (1) A dead illegal leaf becomes a live illegal leaf, (2) A path is broken creating a live illegal leaf, and (3) A processor, other than the root, becomes the root of an illegal rooted path.

1. For any p such that $C_p = E$, only $EC3$ changes C_p. If p executes $EC3$, then one of the following two conditions must be true: (i) $Anc_p = \emptyset$ and p is not a leaf, and (ii) $Anc_p = \{r\}$ and p is not a leaf of an illegal path. Both (i) and (ii) contradict our assumption.

2. In order to break a path $\mu_p = (p_1, p_2, ..., p_l)$ so that a live leaf is created, $\exists p_i \in \mu_p$ such that p_i executes an action in α and p_i becomes a live leaf in β. Since, $D_{p_i} \neq \perp$, p_i can execute $TC2$, $EC1$, and $EC2$ in α.

   If p_i executes $EC2$, then C_{p_i} becomes equal to E. So, p_i is not a live leaf.

   If $(EC1, p_i, \alpha)$, then $C_{p_i} = E$ in α because $\sharp Anc_{p_i} > 0$. Since the execution of $Ec1$ does not change the color, p_i cannot become a live leaf.

   If $(TC2, p_i, \alpha)$, then $D_q = \perp$ in α. Thus, after the execution of $TC2$, p_i becomes a live leaf, but q is no more a leaf.

3. A processor, $p \neq r$, without an ancestor, cannot select a new descendant because both $Forward(p)$ and $Backward(p)$ are disabled for p.

We proved the contradiction in all three cases. □

C. Color Consistency

Lemma C.1. *The root r changes its color infinitely often.*

Proof: By Theorem 4.6, r executes an action infinitely often. r can execute only $TC1$ and $TC2$. If r executes $TC1$ infinitely often, then r changes its color infinitely often and hence, the lemma is proven. Assume that r does not execute $TC1$ infinitely often. This implies that r executes $TC2$ infinitely often (by Theorem 4.6). Then, by the definition of $Search_r$, eventually $D_r = \perp$ must be true. This will enable r to execute $TC1$, which contradicts our assumption. □

Theorem C.2. $\mathcal{A}_3 \triangleleft \mathcal{A}_2$.

Proof: $\mathcal{A}_3$ is closed:

1. Assume that $D_r = \perp$. By Lemma C.1, r changes its color infinitely often. r chooses a descendant by executing $TC1$. If r chooses a descendant which belongs to a cycle or to an illegal rooted path with a dead leaf, then *ColorConsistent* remains true ($CC3$). If r selects a path-free descendant p, then p becomes the new live leaf of the legal path, μ_r, and thus, *ColorConsistent* is preserved ($CC2$).

2. Assume that $D_r \neq \perp$. The only processor which can choose a descendant is the live leaf of the legal path by executing $TC1$. Assume that p is the live leaf. If p chooses a path-free processor as the descendant, then all processors except the leaf, are r-colored. Thus, *ColorConsistent* remains true ($CC2$). If p selects a path-free descendant q, then q becomes the new live leaf of the legal path, μ_r, and thus, *ColorConsistent* is preserved ($CC2$).

Every computation starting from a configuration satisfying $\mathcal{A}_2$ leads to a configuration in $\mathcal{A}_3$: The proof follows from Corollary 4.11. □

D. Legitimacy Predicate

Theorem D.1 (Single Token Property). $\forall \alpha \vdash \mathcal{A}_4$, *exactly one processor has a token at any time.*

Proof: Follows from Corollary 4.11. □

Recall from Section 3.2 that δ_{c_0} and δ_{c_1} denote the configurations where every processor is path-free and has the color 0 and 1, respectively. Thus, both δ_{c_0} and δ_{c_1} satisfy $\mathcal{L}_{\mathcal{TC}}$.

Lemma D.2. $\forall \alpha \vdash \mathcal{A}_4$, $\alpha \rightsquigarrow \delta_{c_0}$.

Proof: By Lemma 4.13, $\exists \alpha' : \alpha \rightsquigarrow \alpha'$ such that all processors in the system are path-free or belong to the legal path. By Lemma 4.14, $\exists \alpha'' : \alpha' \rightsquigarrow \alpha''$ such that every path-free processor is 0– or 1-colored. Then by successive *crounds*, $\exists \beta : \alpha'' \rightsquigarrow \beta$ such that every processor is path-free and has the same color k (0 or 1). If $k = 0$ in β, then the lemma is proven ($\beta = \delta_{c_0}$). If $k = 1$, then, in the next round, k becomes equal to 0. □

Theorem D.3. $\mathcal{L}_{\mathcal{TC}} \triangleleft \mathcal{A}_4$.

Proof: $\mathcal{L}_{\mathcal{TC}}$ is closed: Follows from Actions $TC1$ and $TC2$.

Every computation starting from a configuration in $\mathcal{A}_4$ leads to a state in $\mathcal{L}_{\mathcal{TC}}$: Follows from Lemma D.2. □

Theorem D.4 (Fairness Property). *Starting from any configuration* $\in \mathcal{L}_{\mathcal{TC}}$, *in every cround, every processor obtains the token at least once.*

Proof: Follows from the Definition 2.3, the definitions of δ_{c_0} and δ_{c_1}, the definition of *token* (Section 3.1), and Actions $TC1$ and $TC2$. □