Space-efficient, distributed and self-stabilizing depth-first token circulation

Colette Johnen, Joffroy Beauquier
L.R.I./C.N.R.S.
Université de Paris-Sud
Bat. 490, Campus d'Orsay
F-91405 Orsay Cedex, France

tel : (+33) 1 69 41 66 29
fax : (+33) 1 69 41 65 86
colette@lri.fr, jb@lri.fr

Abstract

The notion of self-stabilization was introduced by Dijkstra. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". Such a property is very desirable for any distributed system, because after any unexpected perturbation modifying the memory state, the system eventually recovers and returns to a legitimate state, without any outside intervention.

In this paper, we are interested in a distributed self-stabilizing depth-first token circulation protocol on an uniform rooted network (no identifiers, but a distinguished root).

As already noted, a search algorithm together with a deterministic enumeration of the node's neighbors yields an algorithm determining a spanning tree.

Our contribution is improving the best up to now known space complexity for this problem, from $O(\log(N))$ to $O(\log(D))$ where N is number of nodes and D is the network's degree. Moreover, we give a full proof of the algorithm correctness assuming the existence of a distributed demon.

Keywords : fault-tolerant distributed algorithms, self-stabilization, spanning tree, mutual-exclusion, distributed demon.

1 Introduction

The notion of self-stabilization was introduced by Dijkstra [5]. He defined a system as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". Such a property is very desirable for any distributed system, because after any unexpected perturbation modifying the memory state, the system eventually recovers and returns to a legitimate state, without any outside intervention. Self-stabilizing has been since studied by various researchers and Dijkstra's original notion, which had a very narrow scope of application, has

proved to encompass a formal and unified approach to fault-tolerance, under a model of transient failures for distributed systems.

In this paper, we are interested in the construction of a distributed Self-stabilizing for depth-first token circulation in an uniform rooted network (no identifier, but a distinguished root). As a token circulation algorithm, our algorithm provides a fair mutual-exclusion protocol (the node having the token is the one authorized to enter into critical section).
Several authors [5], [6], and [3] have presented token circulation algorithm on ring networks ; Brown, Gouda, and Wu [2] have presented one on linear chains ; Kruijer [11] have presented one on tree networks. Huang and Chen have presented an algorithm [10] on general networks with a distributed demon.

As noted in [10], a token circulation algorithm together with a deterministic enumeration of the node's neighbors yields an algorithm determining a spanning tree. The task of spanning tree construction is a basic primitive in communication networks. Many crucial network tasks, such as network reset (and thus any input/output task), leader election, broadcast, topology update, and distributed database maintenance, can be efficiently carried out in the presence of a tree defined on the network nodes spanning the entire network. Improving the efficiency of the underlying spanning tree algorithm usually also correspondingly improves the efficiency of the particular task at hand.

Note that other constructions of spanning trees in a self-stabilizing way are known. Some authors (as in [1] and [4]) have presented algorithms with a central demon. Huang and Chen [9] construct a minimal spanning tree with a distributed demon. Sur and Srimani [12] have presented a similar algorithm but the correctness proof is substantially simpler, based on graph theoretical reasoning. Dolev, Israeli, and Moran [7] have reported a minimal spanning tree construction with read-write atomicity (then the system is fully asynchronous). Finally, Tsai, and Huang [13] have presented an algorithm that constructs a minimal spanning tree with a fully distributed demon.

There are two principal measures of efficiency for self-stabilizing algorithms : stabilization time, which is the maximum time taken for the algorithm to converge to a legitimate state, starting from an arbitrary state and the space required at each node (e.g. size of local memory needed). We are interested here in reducing the value of the second parameter. The goal of producing systems with a small number of states per processor/node is of particular interest because such processors may have direct implementations in hardware.

The existing solutions for token circulation or spanning tree construction on general network topology have a space complexity in $O(\log(N))$, N being the number of nodes. Our contribution is a new algorithm that achieves the goal in $O(\log(D))$ states per node, D being the upper bound of node's degree.

On the other hand, Burns and Pachl [3] showed that does not exist uniform self-stabilizing token circulation on a composite ring. The best that can be proposed is a semi-uniform algorithm, as our algorithm.

Moreover, our protocol does not need to know the number of nodes in the network. Therefore, it works for any connected network and even for dynamic networks, in which the topology of the network may change during the execution (nevertheless, the upper bound of the node's degree should not increase to keep constant the required memory space at each node).

We give the extensive proof of our algorithm within the distributed model where several nodes can simultaneously perform a move.

The remainder of the paper has been organized as follows ; an informal description of the proposed protocol is provided in section 2. The formal model is described in section 3 ; protocol formal

description is given in section 4 ; its correctness is proven in section 5.

2 Informal description of the protocol

As a model of computation, we choose the following model, that is an extension of Dijkstra's original model for rings to arbitrary graphs. Consider a connected graph G(V, E), in which V is a set of nodes and E is a set of edges. Such a graph is used to model a distributed system with N nodes, N = |V|, in which each node represents a processor. In the graph, directly connected nodes are called each other's neighbors. Our goal is to design a self-stabilizing algorithm that performs a depth-first search on the graph.

The proposed self-stabilizing algorithm is encoded as a set of rules. Each processor has several rules. Each rule has two parts : the privilege (condition) part, and the move part. The privilege part is defined as a boolean function of the processor's own state and of the states of its neighbors. When the privilege of a rule on a processor is true, we say that the processor has the privilege. A processor having the privilege may then make the corresponding move which changes the processor state into a new one that is a function of its old state and of the states of its neighbors.

We assume the existence of a *distributed* demon and we assume that the computation proceeds in steps. The distributed demon [5] chooses *several* privileged nodes and one enabled rule on each chosen node at a time. Hence, in each computation step, several processors make a move. The privileges for the next move depend on the states resulting from the previous moves. The rules are atomic : the processors cannot evaluate their privilege at a time and then make the move later with in between other moves.
To ensure the correctness of the protocol, the demon is regarded as an adversary and the protocol is required to be correct in all possible executions. Nevertheless, the demon is fair, a node does not hold forever a privilege on a rule without being chosen by the demon.

The proposed algorithm has two parts. One circulates the token among the nodes in an indeterministic depth-first order. This part is identical to the one in the Huang-Chen algorithm [10]. The other part handles abnormal situation due to unpredictable initial states or transient failures.

We name r the distinguished node that initiates the depth-first circulation rounds, and chooses the round color (0 or 1). Each node has a color among three values : 0, 1, and E (for error). The node having the token, takes the round color and searches among its neighbors one which has not been visited during this round (an isolated node having the color different of the round color and of E color). If it finds a suitable node then it passes to this node the token ; else it backtracks the token to its parent. When the token has backtrack to r ; the round is over. r initiates a new round with the other color.

There are two error-handling strategies : one for destroying the illegal branches that are not cycles and the other for the cycles. The treatment of illegal branches (branches which are not cycles and which are not rooted to the legal root) is similar to the one used by Huang and Chen. The illegal roots detect their abnormal situation and color themselves to E. The E color is propagated to their leaf ; then, the E-colored leaves are dropped ; and the detached E-colored nodes are recovered by changing color. The repetition of dropping and recovering processes will correct all nodes inside illegal branches (there is a finite number of creations of new illegal branches). The cycle destruction strategy is completely different from the one used in [10]. Our solution does not use a variable level. The key point is the detection of cycles by outside nodes that will provoke the correction. The root initializes successive depth-first searches alternatively colored 0 and 1. Note that, due to a bad initialization, such a depth-first search can only be partial.

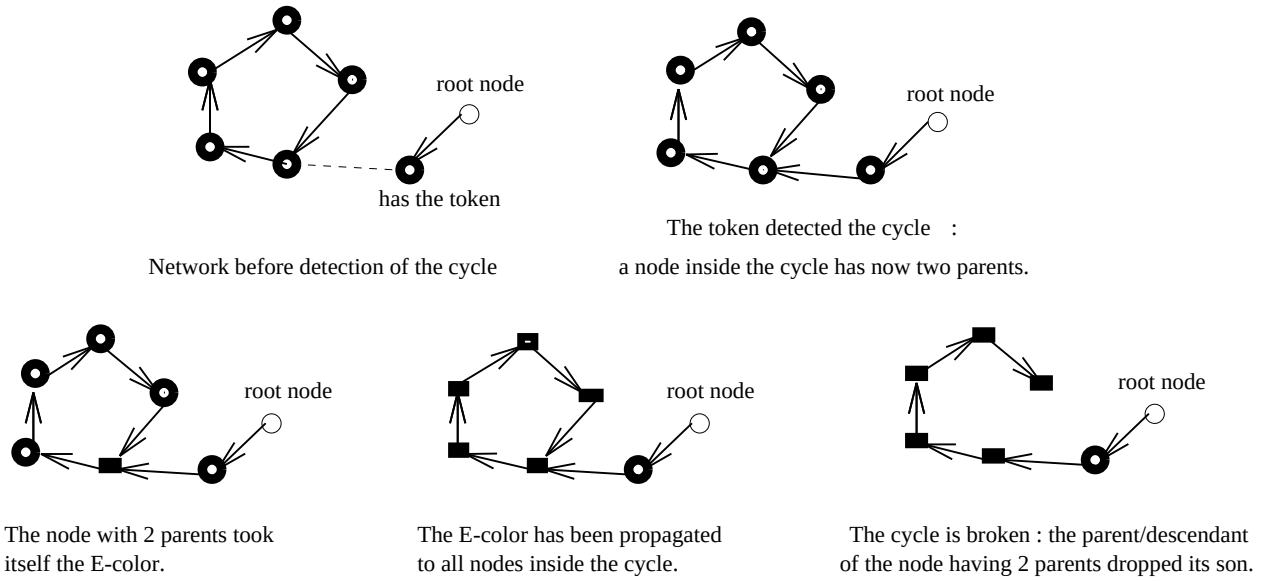

Figure 1: Destruction of a cycle

During a 0-colored round, all nodes inside the branch are 0-colored. If during a 0-colored round, the leaf has an 1-colored neighbor which is inside a cycle, there must be an error somewhere. Then the leaf chooses the faulty node as son (figure 1). The faulty node detects that it has two parents, and then colors itself E. The E color propagates to the descendant-parent of the faulty node. At this point, this node can drop its son (the faulty node) and break the cycle.
Obviously, the same holds if a node inside a cycle during an 1-colored round is 0-colored.

The nodes inside a cycle can change their color only to become E-colored (by a R8 move). This move can be performed at most once on a node inside a cycle. Thus, such nodes stop changing color. We can also prove that the cycles are eventually destroyed.

3 Formal model

Let S be a system defined by a set of states and a set of transitions where each transition is an ordered pair of states.

A *computation* is a sequence of system states (s_1, s_2, ... , s_n, ...) where each couple (s_i, s_{i+1}) is a system transition.
A system state is defined by the local variable values of each node. If the simultaneous moves of several rules modify the system state from s_1 to s_2 then (s_1, s_2) is a system transition in the case where (i) at most one move by a node is performed ; (ii) and in s_1 the rule privileges are satisfied on the nodes which perform the corresponding rule moves.

A *region* of a system is a subset of system states. A region *REG* is *closed* if for every transition (s_1, s_2) where s_1 in *REG* then s_2 is in *REG*.
A computation *C leads* to a region *REG*, if *C* has a state in *REG*.
A region *REG* is an *attractor of a computation* (s_1, s_2, ...) if there is an integer n such that for all i > n, s_i belongs to *REG*.
A region *REG* is an *attractor* if it is closed and all computations lead to *REG*.
A predicate P over system states defines the region *REG*(P) as the set of states where P is satisfied. Shorten, we said that P is closed (resp. attractor) if and only if *REG*(P) is closed (resp. an attractor).

A predicate P_n over node states is a *trap* if for any node i, the predicate "$P_n(i)$ = true" over system states is closed.

A *legitimate states set* verifies several properties [5] : (i) it is closed, (ii) in each legitimate state, one and only one node holds one privilege, (iii) each legitimate state is reachable from any other legitimate state, and (iv) each node has a legitimate state where it holds a privilege.

We call a system *self-stabilizing* if and only if regardless of the initial state and regardless of the computation, the system is guaranteed to reach the legitimate states set after a finite number of moves.

4 Protocol formal specification

Notation X.i is read X of i; notation X.Y.i : X of Y of i.
Each node i maintains the following variables :

- D.i : a pointer pointing to one of its neighbors (called i's son) or pointing to *NULL*.
- C.i : the color of node i taking value in the set {0, 1, E}.

The required space at each node can be evaluated. Under the hypothesis that the graph under consideration has a fixed upper bounded degree D, independent from the number N of nodes, the size of son variable is log(D) ; the color variable has also a fixed size (2 bits). Then, the space complexity of the algorithm at each node is $O(\log(\mathrm{D}))$.

Other used notations are :

- P.i : the set of i's parents.
- NB.i : the set of i's neighbors with r excluded.
- NP.i : the number of i's parents.

4.1 Token circulation rules

We define some predicates used in the definition of the token circulation rules :
Token.i holds if i is a live leaf and i's color is not the same as i's parent color. A live leaf is a leaf whose color is not E.
BToken.i holds if i's son is a live leaf whose color is the same as i's color.
Anomalous(i,k) holds if k has a parent and does not have the expected color for an inside node with respect to i (the expected color is either C.i if BToken.i or C.i+1 mod2 if Token.i).
Detached.i holds if i is a node without son and without parent.
PotentialFirstSon(i,k) holds if Token.i holds, there is no anomalous node with respect to i, and k is a potential first i's son (k is a detached node with the right color : i's color).
DeadEnd.i holds if Token.i holds, there is no anomalous node with respect to i, and it does not exist a potential first i's son.
PotentialNewSon(i,k) holds if BToken.i holds, there is no anomalous node with respect to i, and k is a potential new i's son. (k is a live detached node with the right color : different from i's color).
Backtrack.i holds if BToken.i holds, there is no anomalous node with respect to i, and it does not exist a potential new i's son.

We formally define the predicates :

- Token.i = $\big[\big((i = r) \wedge (\mathrm{C}.i \neq \mathrm{E}) \wedge (\mathrm{D}.i = \mathrm{NULL})\big) \vee \big((i \neq r) \wedge (\mathrm{D}.i = \mathrm{NULL}) \wedge$

$$(\mathrm{NP}.i = 1) \wedge (\mathrm{C}.i \neq \mathrm{E}) \wedge (\mathrm{C.P}.i \neq \mathrm{E}) \wedge (\mathrm{C.P}.i \neq \mathrm{C}.i)\big)\big]$$

- $\mathrm{BToken}.i = [(\mathrm{D}.i \neq \mathrm{NULL}) \wedge (\mathrm{D.D}.i = \mathrm{NULL}) \wedge (\mathrm{C.D}.i = \mathrm{C}.i) \wedge (\mathrm{C}.i \neq \mathrm{E})]$
- $\mathrm{Anomalous}(i,k) = \big[\exists\, k \in \mathrm{NB}.i \mid (\mathrm{NP}.k \geq 1) \wedge \big((\mathrm{Token}.i \wedge (\mathrm{C}.k \neq \mathrm{C}.i{+}1 \bmod 2)) \vee (\mathrm{BToken}.i \wedge (\mathrm{C}.k \neq \mathrm{C}.i))\big)\big]$
- $\mathrm{Detached}.i = [(\mathrm{D}.i = \mathrm{NULL}) \wedge (\mathrm{NP}.i = 0)]$
- $\mathrm{PotentialFirstSon}(i,k) = \big[\mathrm{Token}.i \wedge \big(\forall\, \mathrm{j} \in \mathrm{NB}.i \mid \neg\mathrm{Anomalous}(i,k)\big) \wedge \big(\exists\, k \in \mathrm{NB}.i \mid \mathrm{Detached}.k \wedge (\mathrm{C}.k = \mathrm{C}.i)\big)\big]$
- $\mathrm{DeadEnd}.i = \big[\mathrm{Token}.i \wedge \big(\forall\, \mathrm{j} \in \mathrm{NB}.i \mid \neg\mathrm{Anomalous}(i,k) \wedge \neg\mathrm{PotentialFirstSon}(i,k)\big)\big]$
- $\mathrm{PotentialNewSon}(i,k) = \big[\mathrm{BToken}.i \wedge \big(\forall\, \mathrm{j} \in \mathrm{NB}.i \mid \neg\mathrm{Anomalous}(i,k)\big) \wedge \big(\exists\, k \in \mathrm{NB}.i \mid \mathrm{Detached}.k \wedge (\mathrm{C}.k = \mathrm{C}.i{+}1 \bmod 2)\big)\big]$
- $\mathrm{Backtrack}.i = \big[\mathrm{BToken}.i \wedge \big(\forall\, \mathrm{j} \in \mathrm{NB}.i \mid \neg\mathrm{Anomalous}(i,k) \wedge \neg\mathrm{PotentialNewSon}(i,k)\big)\big]$

On a node i, the token circulation rules are :

R0 : $\mathrm{PotentialFirstSon}(i,k) \wedge (i = r) \rightarrow \mathrm{C}.r = \mathrm{C}.r{+}1 \bmod 2\,;\ \mathrm{D}.r = k$

R1 : $\mathrm{PotentialFirstSon}(i,k) \wedge (i \neq r) \rightarrow \mathrm{C}.i = \mathrm{C.P}.i\,;\ \mathrm{D}.i = k$

R2 : $\mathrm{DeadEnd}.i \wedge (i \neq r) \rightarrow \mathrm{C}.i = \mathrm{C.P}.i\,;\ \mathrm{D}.i = \mathrm{NULL}$

R3 : $\mathrm{PotentialNewSon}(i,k) \rightarrow \mathrm{D}.i = k$

R4 : $\mathrm{Backtrack}.i \rightarrow \mathrm{D}.i = \mathrm{NULL}$

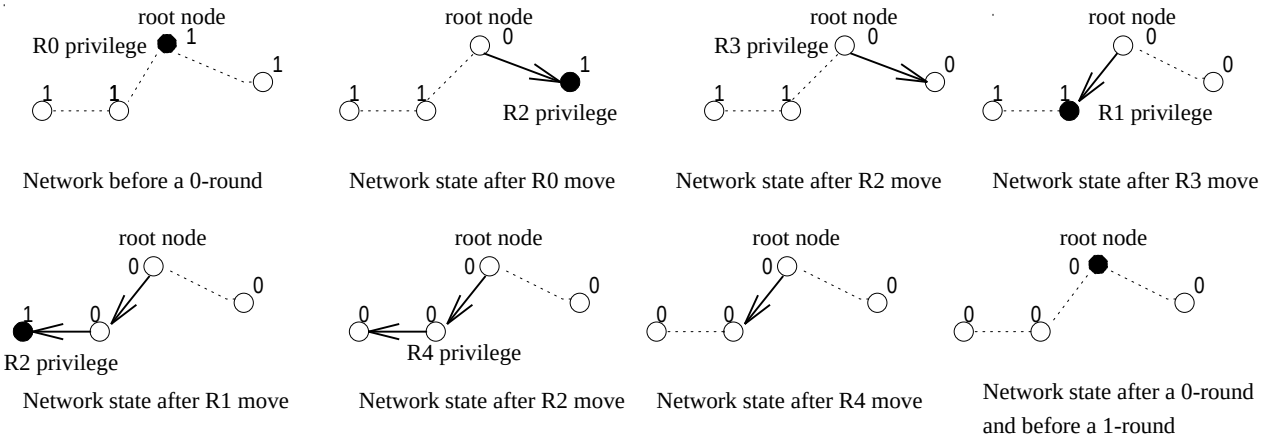

Figure 2: Token circulation

The rule R0 initiates a regular circulation round : the node r changes the round color and chooses a son that gains the token. By a R1 move, the token passes from the previous leaf to its new son (that is now the leaf). So, the branch lengthens. If the leaf cannot find a suitable son (a neighbor that had not been visited during the current round) the leaf drops its token, by a R2 move. A R3 move, substitutes a new leaf (a node that had not been visited during the current round) for the current one (that does not have the token) ; this new leaf gains the token. If the current leaf does have the token and a new suitable leaf cannot be found, the branch is shrunk by a R4 move. When the branch is completely destroyed (e.g. the round is over), the node r has the token, and can perform a R0 move.

Evaluation of any privilege necessitates two communications round : each node has to get the two local variable values from its neighbors. Then, each node can compute its number of parents and transmit this value to all its neighbors.

4.2 Error handling rules

A self-stabilizing system has an unpredictable initial state. In such a state, the D pointers point to any neighbors or NULL. Thus, illegal branches or cycles can exist in the initial state. The following rules delete illegal branches and transform cycles into branches. Thus, the system eventually reaches a legitimate state.

4.2.1 Illegal branch destruction

We define some predicates used in the definition of the illegal branch destruction rules :

- FBToken.$i = [(\text{D}.i \neq \text{NULL}) \wedge (\text{D.D}.i = \text{NULL}) \wedge (\text{C.D}.i = \text{E})]$
- IllegalRoot.$i = [(i \neq r) \wedge (\text{D}.i \neq \text{NULL}) \wedge (\text{NP}.i = 0)]$

FBToken.i holds if i's son is a dead leaf (an E-colored leaf).
IllegalRoot.i holds if i is a branch root without being the node r.

On a node i, the rules that destruct the illegal branches are :

R7 : FBToken.$i \rightarrow \text{C}.i = \text{E}$; $\text{D}.i = \text{NULL}$

R8 : $\exists\ k \in \text{NB}.i \mid \text{D}.k = i \wedge \text{C}.k = \text{E} \wedge \text{C}.i \neq \text{E} \rightarrow \text{C}.i = \text{E}$

R9 : Detached.$i \wedge \text{C}.i = \text{E} \rightarrow \text{C}.i = 0$

R10 : IllegalRoot.$i \wedge \text{C}.i \neq \text{E} \rightarrow \text{C}.i = \text{E}$

The illegal branch destructions are processed as follows : R10 colors illegal roots E. R8 propagates the E color toward the leaf (the E color is propagated only from parent to son). R7 drops the E-colored leaf (the new leaf will have also the color E). R9 recovers the detached erroneous nodes.

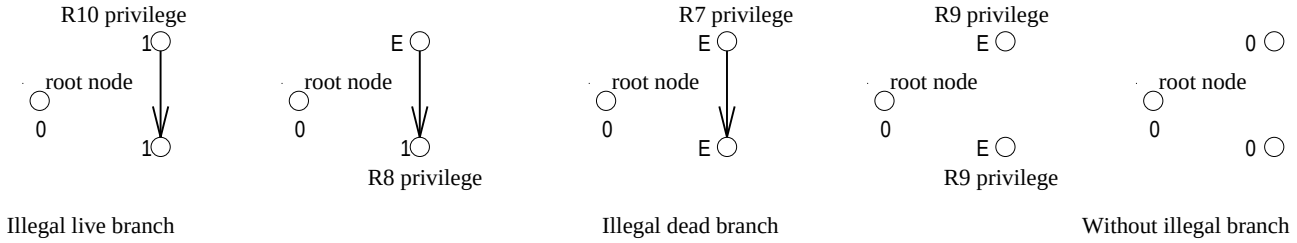

Figure 3: Destruction of an illegal branch

4.2.2 Cycle destruction

R6 : Anomalous(i,k) $\rightarrow \text{D}.i = k$

R11 : $\text{NP}.i \geq 2 \wedge \text{C}.i \neq \text{E} \rightarrow \text{C}.i = \text{E}$

R12 : $\text{NP.D}.i \geq 2 \wedge \text{C}.i = \text{E} \wedge \text{C.D}.i = \text{E} \rightarrow \text{D}.i = \text{NULL}$

The cycle destructions are done as follows : R6 detects an anomalous node, and becomes its new parent (an anomalous node has a parent and does not have the expected color for a node having a parent). Now, R11 can color E the anomalous node (R11 colors E a node having several parents). The E color is propagated to the descendants of the anomalous node by R8 moves. Either the anomalous node is inside a branch (see above), or the anomalous node is inside a cycle. Then a R12 move on the parent/descendant of the anomalous node breaks the cycle (R12 disconnects an E-colored node with its son if its son has several parents and is also E-colored). After that, we have a branch whose leaf is E-colored.

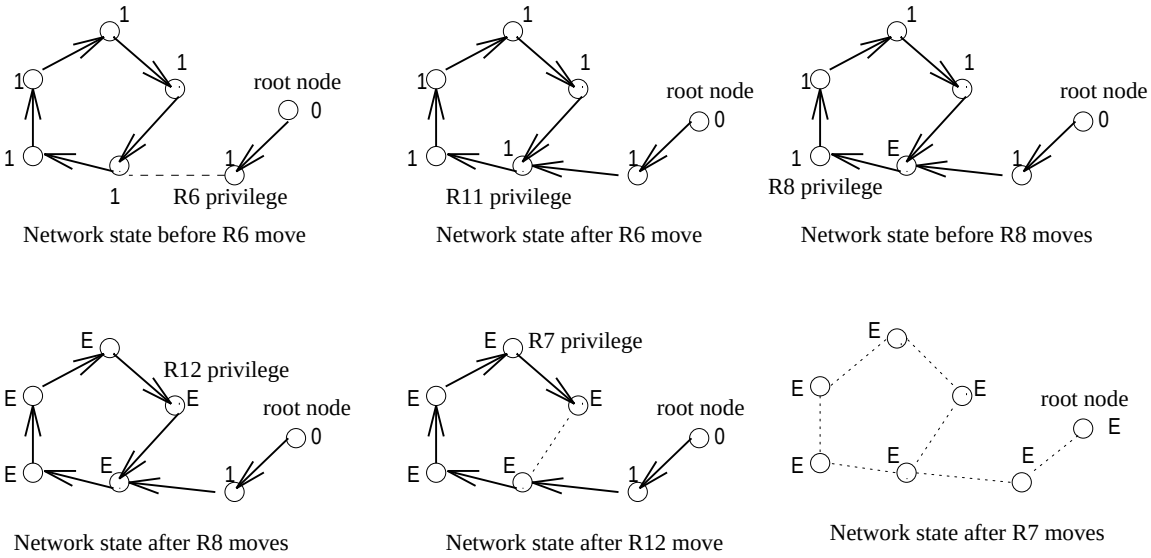

Figure 4: Cycle destruction rules

4.2.3 Miscellaneous error handling

R5 : DeadEnd.$i \wedge (i = r) \rightarrow$ C.r = C.r+1 mod2 ; D.r = NULL

R13 : D.$i = r \rightarrow$ D.i = NULL

The rule R5 initiates a quick round (the only move is the r's color changing). R13 breaks the links parent-son with the node r (r should not have parent).

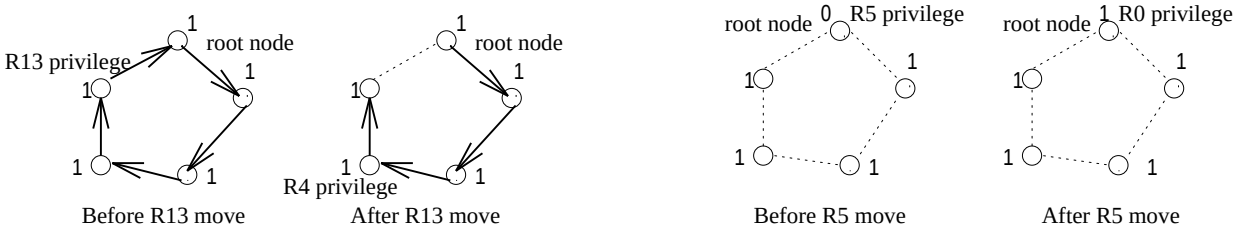

Figure 5: Miscellaneous error handling rules

5 Correctness of the protocol

We name LS the set of states where (i) only one node holds a privilege (ii) the satisfied privilege is R0, R1, R2, R3 or R4 (iii) there is no cycle and no illegal branch.

We will prove that LS is a valid legitimate states set, and that, in LS, the token circulates in depth-first order.

To prove the correctness of our algorithm, we use the *convergent stair* [8] method. We show that there is a sequence of predicates on the system states such that all computations lead to the regions defined by these predicates step by step (w.r.t. each region is an attractor, and each region is a subset of the previous one).

First, we prove that all computations are infinite. Then, we establish that there is a finite number of creations of illegal and live branches (e.g. branch whose root is r and whose leaf is not E-colored), in

any computation. The fairness scheduling of the rules R10 and R8 provokes the dead of the illegal branches (e.g. their leaves get the E color). At this point, we show that no more node will join an illegal branch ; and the illegal branches will eventually destroy themselves (by fairness scheduling of the rule R7). We prove that the legal branch will unavoidably become and stay sound (see the following predicate definition). We demonstrate that after the legal branch is sound, no more cycle is created ; and that the cycles are eventually destroyed.

We conclude in showing that in *LS* the protocol provides a token circulation in depth-first order.

5.1 Predicate definitions

We define some predicates used in the correctness proofs :
Cycle.i holds if i belongs to a cycle : one of i's descendant is a i's parent.
StrictCycle.i holds if i belongs to a cycle and all nodes in this cycle have only one parent.
IllegalNode.i holds if i belongs to a branch whose root is not r.
IllegalLiveRoot.i holds if i is an illegal root whose branch ends in a dead leaf.
DeadLeaf.i holds if i is an erroneous leaf.
Unsound.i holds if i is an inside node (no leaf) of the legal branch that does have the same color as its parent and the legal branch ends in a live leaf. If a such node i exists, we said that the legal branch is unsound.

- Cycle.i = [there is a series of nodes p_0, ... , p_n such that $p_0 = i = p_n \wedge$ ($\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1}$)]
- StrictCycle.i = [there is a series of nodes p_0, ... , p_n such that $p_0 = i = p_n \wedge$ ($\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1} \wedge$ NP.$p_j = 1$)]
- IllegalNode.i = [there is a series of nodes p_0, ... , p_n such that IllegalRoot.$p_0 \wedge p_n = i \wedge$ ($\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1}$)]
- IllegalLiveRoot.i = [IllegalRoot.$i \wedge$ there is a series of nodes p_0, ..., p_n such that $p_0 = i \wedge$ D.p_n = NULL $\wedge$ C.$p_n \neq$ E $\wedge$ ($\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1}$)]
- DeadLeaf.i = $\big[$(D.i = NULL) $\wedge$ (C.i = E) $\wedge$ $\big($(i = r) $\vee$ (N.P.$i \geq 1$)$\big)\big]$
- Unsound.i = [there is a series of nodes p_0, ... , p_n such that $p_0 = r \wedge$ D.p_n = NULL $\wedge$ C.$p_n \neq$ E $\wedge$ ($\forall$ j such that $0 \leq$ j $<$ n $\mid$ D.$p_j = p_{j+1}$) $\wedge$ ($\exists$ j $\mid 0 \leq$ j $<$ n $\wedge$ $i = p_j$) $\wedge$ C.$i \neq$ C.P.i]

5.1.1 Algorithm Liveness

Theorem 1 *In any system state, at least one node holds a privilege.*

Proof
We consider the two possible global configurations :

1 there is a leaf i $\big($D.i = NULL $\wedge$ (N.P.$i \neq 0 \vee i = r$)$\big)$. Either, the leaf holds the R0, R1, R2, R5, R6, R8, R9, or R11 privilege. Or, the parent of the leaf holds the R3, R4, R6, R7 or R13 privilege.

2 There is no leaf.

- NP.$r \neq 0$ - all r's parents hold the R13 privilege.
- NP.$r = 0$ - Let the series p_0, ... p_n such that $p_0 = r \wedge$ ($\forall$ j such that $0 <$ j $<$ n there is : D.$p_j = p_{j+1}$). There is no leaf, thus the series is infinite. But, there is a finite number of nodes ; thus some nodes are several times in the series. Let be p_0, ... p_k the smaller prefix of the series with a node appearing twice. It exists $0 <$ i $<$ k such that $p_i = p_k$.

p_k has two parents (p_{k-1} and p_{i-1}).

- $\star$ C.$p_k \neq$ E - p_k holds R11 privilege.
- $\star$ $\exists$ j such that i < j < k $\wedge$ C.p_{j-1} = E $\wedge$ C.$p_j \neq$ E - p_j holds R8 privilege.
- $\star$ $\forall$ j such that i < j $\leq$ k there is C.p_j = E - p_{i-1} and p_{k-1} holds R12 privilege. □

5.2 Destruction of illegal branches

5.2.1 Destruction of illegal and live branches

We present how all illegal and live branches are eventually destroyed, whatever computation is performed.

Lemma 1 *REG1 = { NP.r = 0 } is an attractor.*

Proof : *REG1* is closed : The rules R8, R9, R10, and R11 have no effect on parent/son relationships. The move of the rules R2, R4, R5, R7, R12 and R13 on i put D.i value to NULL. The move of R0, R1, R3 and R6 on i changes the i's son value ; the new son belongs to NB.i (thus it cannot be r). Every computation leads to *REG1* : A R13 move decreases the number of r's parents ; and as long as *REG1* is not reached, a node satisfies the R13 privilege. By fairness scheduling, *REG1* will be reached. □

Let us define CorrectLegalBranch as a boolean function of the system state. This function is true if there is a series of nodes p_0, ..., p_n such that

$$p_0 = r \wedge (\forall \text{ i such that } 0 \leq \text{i} < \text{n} \mid \text{D}.p_i = p_{i+1}) \wedge \Big(\text{DeadLeaf}.p_n \vee (\exists \text{ i} \mid 0 \leq \text{i} < \text{n} \wedge p_i = p_n) \vee \big((\text{NP}.p_n > 1 \vee \text{D}.p_n = \text{NULL}) \wedge (\forall \text{ i such that } 0 \leq \text{i} < \text{n} \mid \text{C}.p_i \neq \text{E})\big)\Big)$$

CorrectLegalBranch is true when the branch whose root is r ends in a dead leaf or in a cycle, or when all nodes of this branch between the root and a suitable node are not E-colored (a node is suitable if it is a leaf or if it has several parents).

Let us define the number X as following (if CorrectLegalBranch is false then IncorrectLegalBranch = 1 otherwise IncorrectLegalBranch = 0) :

X = number of illegal and live roots + IncorrectLegalBranch

First, we will prove that X value decreases at each creation of an illegal live root ; then we will establish that X value never increases. We will conclude that there is a finite number of illegal live root creations, in any computation. At this point, we will be able to prove that each computation is attracted by system states where there is no live and illegal branch.

Lemma 2 *In REG1, at each creation of an illegal and live root, X decreases.*

Proof : There is creation of an illegal root, only when all parents of one node perform a R12 move. This node becomes a new illegal root by losing all its parents. Let us name i this node.

- ***i* was inside a cycle**. A parent of i was also a i's descendant. After the R12 moves, this descendant is a dead leaf and it is still a i's descendant. i is an illegal and dead root.
- ***i* was inside a dead branch**. After the R12 moves, the i's branch still ends in dead leaf. i is an illegal and dead root.
- ***i*'s branch ended in a cycle**. After the R12 moves, the i's branch still ends in a cycle.
- ***i*'s was inside several illegal and live branches**. After the R12 moves, all these branches end in a dead leaf (one of i's parents). Several illegal and live branches are replaced by one illegal and live branch whose root is i. X decreases.

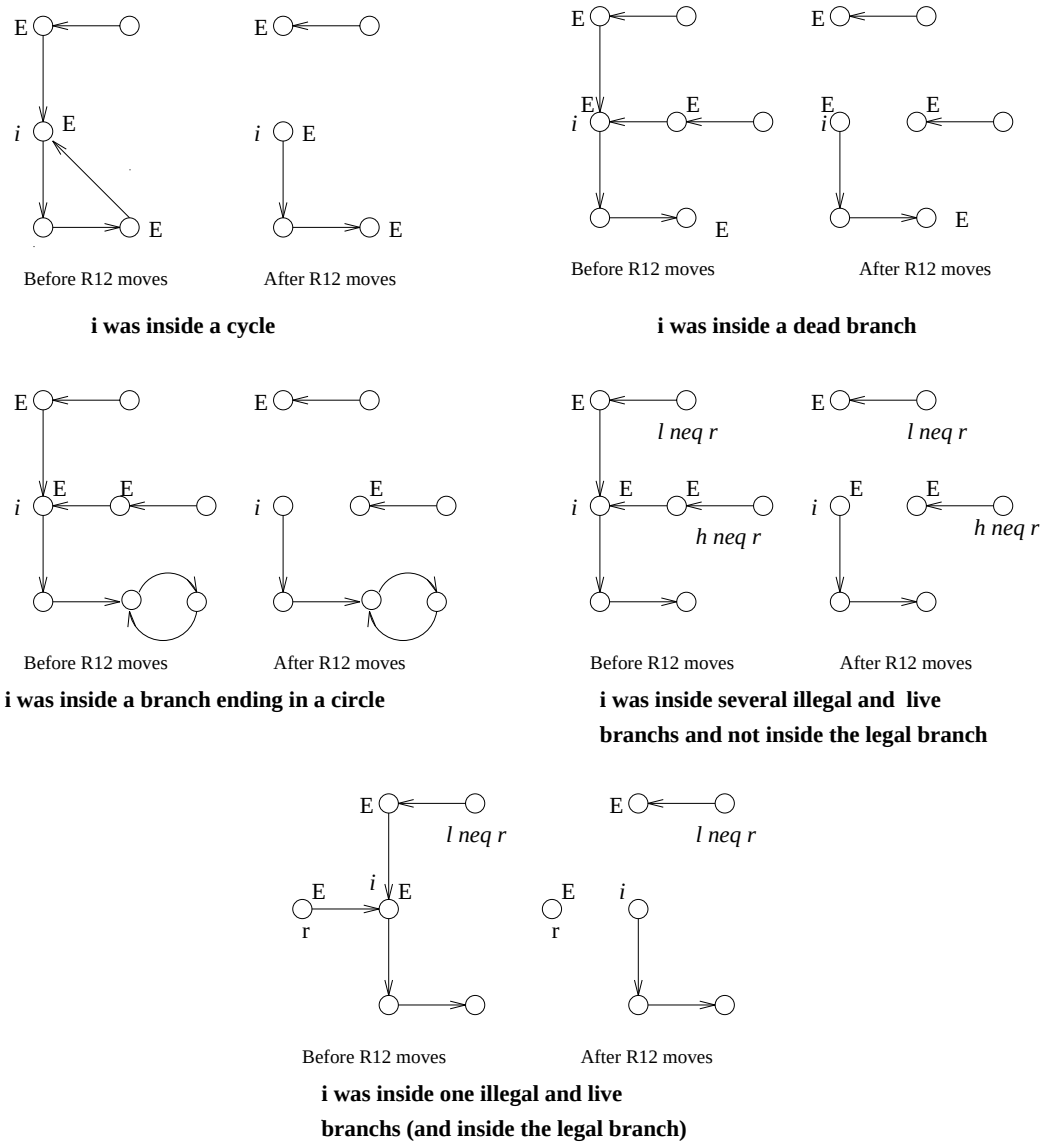

Figure 6: All cases where a node i lost all its parents

- ***i*'s was inside one illegal and live branch and was not inside a cycle or a dead branch**. i was also inside the legal branch (because i had several parents). We had IncorrectLegalBranch = 1 (the legal branch ended in a live leaf, all nodes between r and i had only one parent, and at least one of them was E-colored). After the R12 moves, the number of illegal and live branches did not change (we substituted i for one root). But nevertheless, X decreased because now IncorrectLegalBranch = 0 . □

Lemma 3 *In REG1, X never increases.*

Proof : There are only two cases where X increases : Either the number of illegal and live roots increases ; thus a new illegal and live root has been created. The lemma 2 establishes that X does not increase in this case. Or, the legal branch reaches an incorrect state from a correct one. Let us study this case :

- The legal branch ends in a cycle. The only move that changes that is the R12 move on a node of this cycle. The legal branch get a dead leaf.
- The legal branch ends in a dead leaf. The only move that changes that is the R9 move on r. After this move the legal branch is reduced to r.

- (i) The legal branch ends in a live leaf ; (ii) all inside nodes have only one parent and are not E-colored. Only a R6 move changes that (and gives several parents to a node that we call i).
- All nodes between r and i are not E-colored and have only one parent. i has several parents. Several cases are possible :
  - The legal branch ends in a cycle (as above) -
  - The legal branch ends in a dead leaf (as above) -
  - The legal branch still ends in a live leaf - Thus, i is inside one illegal and live branch. Either, after a R6 move the legal branch will end by a cycle or a dead leaf. Or, a series of R8 moves will give a dead leaf to the legal branch. Or all i's parents except the parent inside the legal branch will perform a R12 move. The legal branch will be in an incorrect state. But all illegal branches that contained i will end in a dead leaf. The number of illegal and live root will decrease ; thus X will not increase. □

Theorem 2 *REG2 = REG1* $\cap$ { $\forall$ *i* $\neg$*IllegalLiveRoot.i* } *is an attractor.*

Proof : X decreases at each creation of illegal and live root (lemma 2) and X never increases. At some point, there will be no more creation of illegal and live roots. Then, by fairness scheduling of the R8 and R10 moves, *REG2* will be reached. □

In *REG2* there is at most one live leaf (the leaf of the legal branch). There are some illegal branches but all of them are dead.

5.2.2 Destruction of illegal and dead branches

We present how all illegal and dead branches are eventually destroyed, whatever computation be performed.

Lemma 4 *REG3 = REG2* $\cap$ { $X = 0$ } *is an attractor.*

Proof : *REG3* is closed (lemma 3). In *REG2* X $\leq$ 1, X = 1 if the legal branch is in incorrect state (the legal branch has a E-colored node and its leaf is not E-colored) ; a series of R8 moves can change that by giving a dead leaf to the legal branch. By fairness scheduling, these R8 moves will eventually be performed . □

Lemma 5 *In REG3, the predicate* $\neg$*IllegalNode is a trap.*

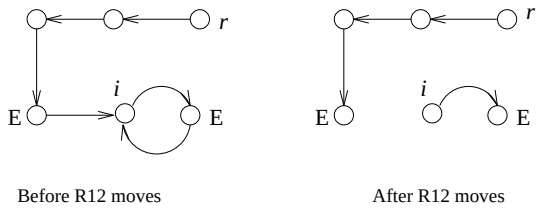

Figure 7: A node inside the legal branch and inside a cycle

Proof : The illegal branches cannot be extended because they are dead. The only way that nodes become illegal is the creation of a new illegal branch whose root was not already an illegal node. The new root was inside the legal branch and was inside a cycle ; and its both parents performed a R12 moves (see figure 7). But in this case, the legal branch was in an incorrect state : all nodes between r and i had one parent, and at least one was E-colored. We have a contradiction ; the legal branch is always in a correct state in *REG3*. □

Theorem 3 *REG4 = REG3* $\cap$ { $\forall$ *i* $\neg$*IllegalNode.i* } *is an attractor.*

Proof : By fairness scheduling of the R7 moves, the illegal branches will destroy themselves. □

5.3 Destruction of cycles

We show that in all computations, the cycles are eventually destroyed.

5.3.1 Soundness of the legal branch

We show that whatever computation be performed, it leads to system states where the legal branch is and stays sound (when the legal branch ends in a live leaf, all inside nodes of the legal branch have the same color).

Lemma 6 *In REG4, The predicate ¬Unsound is a trap.*

Proof : Let us study all the legal branch configurations :

- If the legal branch ends in a cycle : a R12 move will be performed that will give an E-colored leaf to the legal branch.
- If the legal branch ends in a dead leaf : only the R9 move on r gives to the legal branch a live leaf (the legal branch is reduced to r).
- The legal branch ends in a live leaf (All inside nodes have the same color ($\neq$ E) and all of them have only one parent). The nodes of the legal branch cannot perform a R7, R8, R9, R10, R11, R12 or R13 move. R1 colors the new inside node like its parent. R2, R3, and R4 do not change the color of the inside nodes. After R0 or R5, the legal branch is sound. R6(i,k) can only be performed if i is the live leaf of the legal branch and if :
  - k is inside a cycle. After the R6 move, the legal branch ends in a cycle.
  - k is inside the legal branch. After the R6 move, the legal branch ends in a cycle. (remark : before this move, the legal branch was unsound). □

Remark : $REG5 = REG4 \cap \{ \forall \text{ i} : \neg\text{Unsound.i} \}$ is closed.

In order to prove that any execution leads to *REG5*; we will prove that all computations have to lead to *REG5*.

Let C be a computation which does not reach *REG5*. Thus, there is a no-empty set of nodes which are unsound all along C. Let is name $\mathcal{N}_C$ this set. These nodes are and stay in the legal branch along C. We call $\mathcal{REG}4$ the subregion of *REG4* where all nodes of $\mathcal{N}_C$ are unsound and others are not.

In $\mathcal{REG}4$, R6 privilege holds only on the live leaf of the legal branch. As described in the proof of the lemma 6, after a R6 move on the legal leaf, the legal branch is sound. Therefore, C does not contain a R6 move.

Lemma 7 $\mathcal{REG}4a = \mathcal{REG}4 \cap \{ \forall\ i : NP.i \leq 1 \}$ *is an attractor of C.*

Proof : In $\mathcal{REG}4$, there is only one leaf. At a time, only one node can pick up a new son. Thus, any node cannot gain several parents, in one step. Only after a R6 move, a node having a parent get a second one. A R6 move is never performed by C; thus $\mathcal{REG}4$ is closed. By fairness scheduling of the rules R11, R8, or R12, $\mathcal{REG}4a$ will be reached by C. □

Lemma 8 $\mathcal{REG}4b = \mathcal{REG}4a \cap \{ \forall\ i : Cycle.i \vee C.i \neq E \}$ *is an attractor of C.*

Proof : By fairness scheduling of R7, R8 and R9 rules, $\mathcal{REG}4b$ will be reached in C. Any rule moves that can be performed in $\mathcal{REG}4b$ does not color E a node outside cycles. □

Remark : in $\mathcal{REG}4b$, C does not contain R6, R7, R9, R10, R11, R12, and R13 moves. in $\mathcal{REG}4b$, the move R8 is performed a finite number of times (at most one time on each node inside a cycle). After a R0, or R5 move the legal branch is sound. C contains only an infinity of R1, R2, R3, or R4 moves in $\mathcal{REG}4b$.

Let I_i be an integer function of system states defined as :

I_i	=	4	x	number of detached nodes of color different from C.i
	+	3	x	number of nodes in legal branch after i that have a son
	+	2	x	number of leaf whose color differs from C.i
	+	1	x	number of leaf whose color is C.i

Lemma 9 *Let* i *be the farthest node of* $\mathcal{N}_C$ *on the legal branch. In* $\mathcal{REG}4b$, *the* C *computation contains a R4 move on* i.

Proof : All nodes inside the legal branch after i have the same color as i, except the leaf. Until a R4 move on i, I_i is strictly decreased by R1, R2, R3, and other R4 moves. Assume that C does not contain a R4 move on i, the C computation would be finite, in contradiction with the theorem 1. □
After this R4 move, the farthest node of $\mathcal{N}_C$ is the leaf and is sound. Thus, there is a contradiction with the hypothesis *the nodes* $\mathcal{N}_C$ *are unsound all along* C. We conclude that all computations reach *REG5*. The following theorem is a consequence of the lemmas 6 and 9.

Theorem 4 *REG5 is an attractor.*

5.3.2 Destruction of strict cycles

We show that in all computations, the strict circles are eventually destroyed.

Lemma 10 *In REG5, the predicates* $\neg$*Cycle and* $\neg$*StrictCycle are traps.*

Proof : A R8, R9, R10, or R11 move does not modify the previously existing parent/son relations. After a R2, R4, R5, R7, R12 or R13 move on i, i is not within a cycle (D.i = NULL). After a R0, R1, or R3(i,k) move on i, i and k are not within a cycle (D.D.i = NULL) and (D.k = NULL).
The R6(i,k) privilege holds in *REG5* if k is within a strict cycle and i is the leaf. After the R6 move, k is still within the cycle but not within a strict cycle, and i is not within a cycle. □

Remark : *REG6* = REG5 ∩ {∀ i : ¬StrictCycle.i} is closed (lemma 10).

In order to prove that any execution lead to *REG6*, we prove that it does not exist a computation not leading to *REG6*.
Let C be a computation which does not lead to *REG6*. Thus there is a no-empty set of nodes which are and stay inside a strict cycle along C; let us name $\mathcal{N}_C$ this set. We call $\mathcal{REG}5$ the subregion of *REG5* where all nodes of $\mathcal{N}_C$ are inside a cycle and others nodes are not inside a cycle. In *REG5*, after a R6(i,k) move, k which was previously inside a strict cycle, is no more within a strict cycle. Therefore, C cannot contain R6 move in $\mathcal{REG}5$.
The proof of the following lemma is similar to the proof of the lemmas 7 and 8.

Lemma 11 $\mathcal{REG}5b = \mathcal{REG}5 \cap \{ \forall\ i : NP.i \leq 1 \} \cap \{ \forall\ i : Cycle.i \vee C.i \neq E \}$ *is an attractor of C.*

Proof : similar to the proof of the lemma 7 and 8. **Remark :** C contains only an infinity of R0, R1, R2, R3, R4 moves in $\mathcal{REG}5b$.

Lemma 12 *In $\mathcal{REG}5b$, the C computation contains an infinity of R0 moves.*

Proof : Between two R4 moves on r, there is a R0 move. Assume that C contains a finite number of R0 moves on r. At some point, C does not contain R0 and R4 move on r. After that I_r is strictly decreased by all possible moves. Thus the C computation would be finite, in contradiction with the theorem 1. □

Let $\mathcal{D}_i$ be the the minimal distance between r and i defined as :

$$\mathcal{D}_i = \text{Min } \{ \text{ n} \in \mathcal{N} \mid \exists \text{ a node series } p_0, \ldots, p_n \text{ such that } p_0 = r \wedge p_n = i \wedge (\forall \text{ j such that } 0 \leq \text{j} < \text{n} \mid p_{j+1} \in \text{NB}.p_j) \}$$

Let $\mathcal{D}_C$ be the minimal distance between r and a node of $\mathcal{N}_C$. Formally, we define $\mathcal{D}_C$ as :

$$\mathcal{D}_C = \text{Min } \{ \text{ n} \in \mathcal{N} \mid \exists \text{ a node series } p_0, \ldots, p_n \text{ such that } p_0 = r \wedge \text{StrictCycle}.p_n \wedge (\forall \text{ j such that } 0 \leq \text{j} < \text{n} \mid \neg\text{StrictCycle}.p_j \wedge p_{j+1} \in \text{NB}.p_j) \}$$

Lemma 13 *If $\mathcal{D}_C > 1$, then in $\mathcal{REG}5b$, the C computation contains an infinity of R1 or R2 moves performed on each r's neighbor.*

Proof : After a R0 move, r cannot perform a new R0 move until all its neighbors have the same color as its own. (i) There are an infinity of R0 moves ; (ii) the R0 moves are the only moves to change the color of r ; (iii) and the only moves which change the r's neighbors color are R1 and R2 moves. □

Similarly, we prove the following lemma.

Lemma 14 *Let i be a node such that $\mathcal{D}_C > \mathcal{D}_i$ and such that the C computation contains an infinity of R1 or R2 moves performed by i. C contains also an infinity of R1 or R2 moves performed by each i's neighbor.*

Lemma 15 *There does not exist a computation which does not lead to REG6.*

Proof : By induction on the distance between the node i and r, the lemmas 13 and 14 establish that C contains an infinity of i's R1 or R2 moves if $\mathcal{D}_C > \mathcal{D}_i$.
Let i be a node such that $\mathcal{D}_C = D_i{+}1$ and such that i has a neighbor k verifying StrictCycle.k. In $\mathcal{REG}5b$, system states where Token.i is satisfied, are infinity often reached along C (because a R1 or R2 move on i is performed only from a system state where Token.i is satisfied). At some point, k cannot change the color ; the only move (R8 move) that could change k's color had been already performed. After that, when Token.i is satisfied, two cases are possible :

- C.k = C.i+1 mod2, R1 or R2 privilege is satisfied. After the R1 or R2 move on i, C.$k \neq$ C.i+1 mod2. Until Token.i is satisfied, k and i do not change their color. (see second case).
- C.$k \neq$ C.i+1 mod2, the R6 privilege on i is satisfied. This R6 move is the only move which can be performed at that time. C *computation contains a R6 move in* $\mathcal{REG}$5b, in contradiction with the hypothesis. □

The following theorem is a consequence of the lemmas 10 and 15.

Theorem 5 *REG6 is an attractor.*

5.3.3 Destruction of un-strict cycles

The strict cycle have been deleted ; Thus, the is at most one (un-strict) cycle. Now, we establish that the last circle is eventually destroyed.

Theorem 6 *REG7 = REG6 $\cap$ $\{ \forall\ i : \neg Cycle.i \}$ is an attractor.*

Proof : The lemma 10 establishes that the region *REG7* is closed. Let i be a node belonging to a cycle. i does not belong to a strict cycle. Thus, a node of its cycle holds the R11, R8 or R12 privilege. At each system state of *REG6*, only one node holds a privilege. At each step, the only enable move (R11, R8 or R12) is performed until the cycle is destroyed by the R12 move. □

5.4 Legitimate state set

The proof of the following theorem is similar to the proof of lemma 8.

Theorem 7 *$LS = REG7 \cap \{ \forall\ i :\ C.i \neq E\}$ is an attractor.*

In *LS*, (i) only one node has a privilege ; (ii) only the R0, R1, R2, R3, or R4 moves are performed ; and (iii) any node does not verify Cycle or IllegalNode predicates.
From any state of *LS*, we can reach the system state s_0 where all nodes are detached and have the color 0. It is quite obvious that from s_0, any state of *LS* can be reached.

The lemmas 12, 13, and 14 establish that each node i has several legitimate states where Token.i is true. In these states, i holds the R0, R1, or R2 privilege.

The privilege of R2, the rule that stops the branch growing is held if and only if the branch cannot lengthen. The privilege of R4, the rule that shrinks the branch, is held if and only if the branch cannot lengthen and cannot change its way (e.g. to change leaf). Thus, as long as it is possible, the current branch lengthens and the token goes further off r. In LS, the token circulation is done in a depth-first order.

We have proved that (i) *LS* is a valid legitimate state set ; (ii) *LS* is an attractor ; and (iii) in *LS*, our protocol provides a token circulating in the network in depth-first order.

References

[1] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Int. Workshop on Distributed Algorithms*, volume 486, pages 15–28. Springer-Verlag, 1990.

[2] Geoffrey M. Brown, Mohamed G. Gouda, and Chuan lin Wu. Token systems that self-stabilize. *IEEE Transactions on Computers*, 38(6):845–852, 1989.

[3] James E. Burns and Jan Pachl. Uniform self-stabilizing rings. *ACM Trans. on Programming Languages and Systems*, 11(2):330–344, 1989.

[4] Nian-Shing Chen, Hwey-Pyng Yu, and Shing-Tsaan Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39:147–151, 1991.

[5] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Com-munications of the A.C.M.*, 17(11):643–644, 1974.

[6] Edsger W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1:5–6, 1986.

[7] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuring only read/write atomicy. *Distributed Computing*, 7:3–16, 1993.

[8] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, 1991.

[9] Shing-Tsan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.

[10] Shing-Tsan Huang and Nian-Shing Chen. Self-stabilizing depth-first token circulation on networks. *Distributed Computing*, 7:61–66, 1993.

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

[12] Sumit Sur and Pradip K. Srimani. A self-stabilizing distributed algorithm to construct bfs spanning trees on a symetric graph. *Parallel Processing Letters*, 2(2/3):171–179, 1992.

[13] Ming-Shin Tsai and Shing-Tsaan Huang. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters*, 4(1):65–72, 1994.