From Self- to Snap- Stabilization

Alain Cournier* Stéphane Devismes† Vincent Villain‡
LaRIA, CNRS FRE 2733
Université de Picardie Jules Verne
33, rue Saint-Leu
80000, Amiens (France)

July 21, 2006

Abstract

A *snap-stabilizing* protocol, starting from any arbitrary initial configuration, always behaves according to its specification. In this paper, we propose a light semi-automatic method allowing to snap-stabilize self-stabilizing wave protocols for arbitrary networks with a unique initiator. To that goal, we consider such a self-stabilizing protocol $\mathcal{A}$. We then slightly update $\mathcal{A}$ to obtain a protocol $\mathcal{B}$ that can be automatically transformed, using a black box protocol, into a snap-stabilizing protocol. Protocol $\mathcal{B}$ is easy to obtain from $\mathcal{A}$ compared to the design of a snap-stabilizing protocol.

Keywords: Distributed systems, fault-tolerance, self-stabilization, snap-stabilization, wave protocols.

*alain.cournier@u-picardie.fr
†stephane.devismes@u-picardie.fr
‡vincent.villain@u-picardie.fr

1 Introduction

The quality of a distributed system depends on its tolerance to faults that may occur at various components of the system. Many fault-tolerant schemes have been proposed and implemented. The most general technique to design a system tolerating arbitrary transient faults is *self-stabilization* [10]. A self-stabilizing system, regardless of the initial states of the processors and messages initialy in the links, is guaranteed to converge into the intended behavior in finite time. Recently, a new paradigm called *snap-stabilization* has been introduced in [4]. A *snap-stabilizing* protocol guaranteed that, starting from any configuration, it always behaves according to its specification. In other words, a snap-stabilizing protocol is a self-stabilizing protocol which stabilizes in 0 time unit. Obviously, a *snap-stabilizing* protocol is optimal in stabilization time. Nevertheless, designing and proving self- or snap- stabilizing protocols is usually a complicated task. That is why some protocols, called *transformer*, was proposed in the literature, e.g., [14, 6], to automatically perform such a task. In [14], Katz and Perry design a protocol that transforms almost all non-self-stabilizing protocols into self-stabilizing protocols. In [6], the authors propose a *transformer* providing a snap-stabilizing version of any protocol which can be self-stabilized with the transformer of [14]. However, the transformers of [14, 6] use heavy mechanisms to transform an initial protocol into a self- or snap- stabilizing protocol and the overcost of the stabilization is often difficult to evaluate. In particular, they use snapshots to regulary evaluate a predicate defined on the variables of the protocol to transform. This predicate characterizes the normal configurations of the system. This technique is used for preventing the system from deadlocks and livelocks. The main drawbacks of these solutions are: (i) such a predicate is generally difficult to formalize; (ii) the number of snapshots used by the transformer protocol cannot be bounded compared to the number of actions of the initial protocol (i.e., the protocol to transform). In this paper, we propose a light semi-automatic method allowing to snap-stabilize self-stabilizing wave protocols for arbitrary networks with a unique initiator. To that goal, we consider such a self-stabilizing protocol $\mathcal{A}$. We then slightly update $\mathcal{A}$ to obtain a protocol $\mathcal{B}$ that can be automatically transformed, using a black box protocol, into a snap-stabilizing protocol. Protocol $\mathcal{B}$ is easy to obtain from $\mathcal{A}$ compared to the design of a snap-stabilizing protocol. In contrast with the solution in [6], our black box does not use any snapshot to snap-stabilize $\mathcal{B}$ and keeps the same fairness than the protocol to transform. Finally, to show the feasibility of our method, we propose to transform a self-stabilizing depth-first token circulation of Huang and Chen [12] into a snap-stabilizing token circulation.

The rest of the paper is organized as follows: in Section 2, we describe the model in which our protocols are written. In Section 3, we present how our black box works. In particular, we justify the property that a protocol must satisfy to be snap-stabilized by the black box. The correctness proofs and the complexity analysis are provided in the next section (Section 4). To valid our approach, we show in Section 5 how to snap-stabilize the protocol of [12] with our black box. Finally, we conclude in Section 6.

2 Preliminaries

We consider a *network* as an undirected connected rooted graph $G = (V,E,r)$ where V is a set of *processors*, E is the set of *bidirectional asynchronous communication links*, and $r \in V$. In such a network, we distinguish the particular processor r which is usually called *root*. The root processor corresponds to the protocol initiator. In the network, a communication link (p,q) exists if and only if p and q are neighbors. Every processor p can distinguish all its links. To simplify the presentation, we refer to a link (p,q) of a processor p by the *label* q. We assume that the labels of p, stored in the set $Neig_p$, are locally ordered by $\prec_p$. We also assume that $Neig_p$ is an input from the system and constant. In the paper, we also use the following notations: N is the size of the network (i.e., $|V| = N$), Δ is the degree of the network (i.e., the maximal value among the local degrees of the processors), and D is the diameter of the network. In our model, the protocols are *semi-uniform*, i.e., each processor executes the same program except r. Indeed, r is the unique initiator of the protocols and the behavior of a protocol initiator is generally different of the other processors. We consider the local shared memory model of computation introduced by Dijkstra in [10]. This model, called *state model*, is an abstraction of the message-passing model in a sense that the notion of messages is replaced by the fact that any processor can directly read the variables of its neighbors. In the state model, the program of every processor

consists in a set of *shared variables* (henceforth, referred to as variables) and an *ordered finite set of actions* inducing a *priority*. This priority follows the order of appearance of the actions into the text of the protocol. A processor can write to its own variable only, and read its own variables and that of its neighbors. Each action is constitued as follows: $< label > :: < guard > \rightarrow < statement >$. The guard of an action in the program of p is a boolean expression involving variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard is satisfied. The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors. We will refer to the state of a processor and the system as a (*local*) *state* and (*global*) *configuration*, respectively. We note $\mathcal{C}$ the set of all configurations of the system. Let $\gamma \in \mathcal{C}$ and A an action of p ($p \in V$). A is said *enabled* at p in γ if and only if the guard of A is satisfied by p in γ. Processor p is said to be *enabled* in γ if and only if at least one action is enabled at p in γ. When several actions are enabled simultaneously at a processor p: only the priority enabled action can be activated. Let a distributed protocol $\mathcal{P}$ be a collection of binary transition relations denoted by $\mapsto$, on $\mathcal{C}$. An *execution* of a protocol $\mathcal{P}$ is a *maximal* sequence of configurations $e = (\gamma_0,\gamma_1,...,\gamma_i,\gamma_{i+1},...)$ such that, $\forall i \geq 0$, $\gamma_i \mapsto \gamma_{i+1}$ (called a *step*) if γ_{i+1} exists, else γ_i is a terminal configuration. *Maximality* means that the sequence is either finite (and no action of $\mathcal{P}$ is enabled in the terminal configuration) or infinite. All executions considered here are assumed to be maximal. $\mathcal{E}$ is the set of all executions of $\mathcal{P}$. As we already said, each execution is decomposed into steps. Each step is shared into three sequential phases atomically executed: (i) every processor evaluates its guards, (ii) a *daemon* (also called *scheduler*) chooses some enabled processors, (iii) each chosen processor executes its priority enabled action. When the three phases are done, the next step begins. A *daemon* can be defined in terms of *fairness* and *distribution*. There exists several kinds of fairness assumption. In this paper, we consider the *strongly fairness*, *weakly fairness*, and *unfairness* assumption, respectively (these fairness assumptions are the most common of the literature). Under a *strongly fair* daemon, every processor that is enabled infinitively often is eventually chosen by the daemon to execute an action. When a daemon is *weakly fair*, every continuously enabled processor is eventually chosen by the daemon. Finally, the *unfair* daemon is the weakest scheduling assumption: it can forever prevent a processor to execute an action except if it is the only enabled processor. In the following, to simplify the notation, we will denote (when necessary) the strongly fair, weakly fair, and unfair daemon by SF, WF, and UF. Concerning the *distribution*, we assume that the daemon is *distributed* meaning that, at each step, if one or more processors are enabled, then the daemon chooses at least one of these processors to execute an action. We consider that any processor p executed a *disabling action* in the step $\gamma_i \mapsto \gamma_{i+1}$ if p was *enabled* in γ_i and not enabled in γ_{i+1}, but did not execute any protocol action in $\gamma_i \mapsto \gamma_{i+1}$. The disabling action represents the following situation: at least one neighbor of p changes its state in $\gamma_i \mapsto \gamma_{i+1}$, and this change effectively made the guard of all actions of p false in γ_{i+1}. To compute the time complexity, we use the definition of *round* [11]. This definition captures the execution rate of the slowest processor in any execution. Given an execution e ($e \in \mathcal{E}$), the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or a disabling action) of every enabled processor from the initial configuration. Let e'' be the suffix of e such that $e = e'e''$. The *second round* of e is the first round of e'', and so on.

Definition 1 (Wave Protocol [15]) *A* wave protocol *is a protocol $\mathcal{P}$ that satisfies the following requirements: (i) each execution of $\mathcal{P}$ (called* wave*) is finite and contains at least an action of* decision*; (ii) each action of decision is causally preceded by an action of each processor.*

Let $\mathcal{X}$ be a set. $x \vdash P$ means that $x \in \mathcal{X}$ satisfies the predicate P defined on $\mathcal{X}$.

Definition 2 (Snap-stabilization) *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_{\mathcal{T}}$ the specification of $\mathcal{T}$. The protocol $\mathcal{P}$ is snap-stabilizing for the specification $\mathcal{SP}_{\mathcal{T}}$ if and only if the following condition holds: $\forall e \in \mathcal{E} ::\ e \vdash \mathcal{SP}_{\mathcal{T}}$.*

Consider a wave protocol having a unique initiator, *r*, and performing a specific task in a safe environment. In the safe environment, starting from a pre-defined configuration called *normal starting configuration*, *r* initiates the protocol by executing a special action called *initialization action*. This initialization occurs upon an external (w.r.t. the protocol) request. Before this request, all the processors are "asleep" (i.e., disabled). In particular, *r* is on standby of a request. Similary, at the termination of the protocol, the processors become

asleep again until the next request occurs at the initiator. In contrast, in a self-stabilizing environnement, the protocols achieve a convergence to a specified behavior of the system in a finite time. So, the execution of first waves of such protocols may not satisfy its specification and, as a consequence, the waves have to be repeated so that the system eventually satisfies its specification. Hence, self-stabilizing protocols are inherently cyclic and the notion of request is simply kept in the background. On the contrary, the snap-stabilization guarantees that after the first initialization action, the execution of the protocol works as expected (i.e., according to its specification). Thus, snap-stabilization does not require to design cyclic protocols and the initialization of the protocols is similar to the one in a safe environnement, i.e., the initialization is assumed to occur only upon an external request (see [6] for further details). So, in our protocols, we will explicitly mention this external request using the shared variable $Request_r \in \{Wait, In, Out\}$ (noted $\mathcal{P}.Request_r$ for the specific protocol $\mathcal{P}$). We consider $Request_r$ as an input into the algorithm of the protocol initiator (*r*). $Request_r = Wait$ means that an execution of the protocol is required. When the initialization of the protocol occurs, $Request_r$ switches from $Wait$ to In meaning that *r* has taking in account of the request. Finally, $Request_r$ switches from In to Out at the termination of the wave meaning that the system is now ready to receive another request. Of course, the switching of $Request_r$ from $Wait$ to In and from In to Out is managed by the task it-self while the switching from Out to $Wait$ (which means that another execution of the protocol is required) is managed externally. Note that all other transitions (for instance, In to $Wait$) are forbidden. The external action, noted IR, that manages the switching from Out to $Wait$ is of the following form:

$$IR :: ApplicationRequest(r) \wedge (Request_r = Out) \rightarrow Request_r := Wait; ApplicationRelease_r;$$

$ApplicationRequest(r)$ is a predicate which is true when an application of the initiator *r* needs an execution of the snap-stabilizing protocol. $ApplicationRelease_r$ is a macro which contains the code of the application that has to be executed when the system takes the request into account. In particular, this macro has to make Predicate $ApplicationRequest(r)$ false. In the following, we will assume that, since satisfied, $ApplicationRequest(r)$ is continuously satisfied until IR is executed.

From Definitions 1, 2, and the above discussion, follows:

Remark 1 *Let $\mathcal{T}$ be a task, $\mathcal{SP}_{\mathcal{T}}$ a specification of $\mathcal{T}$, and $\mathcal{P}$ a wave protocol with a unique initiator,* r. *To prove that $\mathcal{P}$ is snap-stabilizing for $\mathcal{SP}_{\mathcal{T}}$, we have to show that every execution of $\mathcal{P}$ satisfies the two following conditions: (i) since* r *requests a $\mathcal{P}$ wave, the requested $\mathcal{P}$ wave is initiated in a finite time; (ii) from any configuration where* r *has initiated a $\mathcal{P}$ wave, the system computes $\mathcal{T}$ according to $\mathcal{SP}_{\mathcal{T}}$.*

3 The Approach

Principle. Let $\mathcal{A}$ be a self-stabilizing protocol with a unique initiator, *r*, designed for stabilizing to a specific task $\mathcal{T}$. In addition, assume that the decision actions are at the root only. We want to snap-stabilize $\mathcal{A}$ without using the snapshot techniques. In [14, 6], the snapshots are used for detecting deadlocks and livelocks in the execution of the initial protocol. Since $\mathcal{A}$ is self-stabilizing, we know that, starting from any configuration, it will never generates deadlocks or livelocks. We now propose to slightly modify $\mathcal{A}$ to obtain a protocol $\mathcal{B}$ which is automatically snap-stabilized by a black box protocol. From now on, we note $\mathcal{SSBB}(\mathcal{B})$ the snap-stabilizing version of $\mathcal{B}$ obtained with our Snap-Stabilizing Black Box ($\mathcal{SSBB}$). By Remark 1, the code of $\mathcal{B}$ must insure the following property:

(i) Starting from any configuration and upon an external request on *r*, *r* eventually initiates $\mathcal{SSBB}(\mathcal{B})$.

(ii) As soon as $\mathcal{SSBB}(\mathcal{B})$ is initiated, it executes the task $\mathcal{T}$ as expected.

First, by (i), starting from any configuration, the system must reach a configuration from which $\mathcal{SSBB}(\mathcal{B})$ can properly start. This implies that when the root requests an execution of $\mathcal{SSBB}(\mathcal{B})$, $\mathcal{SSBB}(\mathcal{B})$ must start in a finite time but without aborting a previous initiated computation of $\mathcal{T}$. One way to get this property is to use in $\mathcal{B}$ a variable $\mathcal{B}.End_r$ such that when *r* is ready to decide in $\mathcal{A}$, then *r* is also ready to decide in $\mathcal{B}$ and sets $\mathcal{B}.End_r$ to true. Also, since $\mathcal{B}.End_r$ is equal to true, the initialization actions of $\mathcal{B}$ (at *r*) has to be disabled

until $\mathcal{SSBB}(\mathcal{B})$ can execute the computation of $\mathcal{T}$ as expected (ii). To that goal, we have just to modify the guards of the initialization actions of $\mathcal{A}$ so that they become disabled when $\mathcal{B}.End_r = true$. $\mathcal{B}.End_r$ will be set to true by $\mathcal{SSBB}(\mathcal{B})$ when the system will be in a configuration from which $\mathcal{SSBB}(\mathcal{B})$ can execute the computation of $\mathcal{T}$ as expected (ii). Assuming the existence of $\mathcal{B}.End_r$ and the associated modifications in $\mathcal{B}$, we now just need to reset the variables of $\mathcal{B}$ since $\mathcal{B}.End_r$ is true in order to verify (ii). To that goal, for each processor p, all the variables assignments required to generate a *normal starting configuration* of $\mathcal{A}$ has to be stored in a macro of $\mathcal{B}$ noted $\mathcal{B}.Init_p$. For sake of clarity, we note $\mathcal{B}.Init$ the set of the macros $\mathcal{B}.Init_p$ defined on all the processors p. Using $\mathcal{B}.Init$, the reset phase is trivially initiated at the initialization action of $\mathcal{SSBB}(\mathcal{B})$ and, as soon as the reset terminates, $\mathcal{B}.End_r$ is set to false and $\mathcal{B}$ executes the task $\mathcal{T}$ as $\mathcal{A}$ in a non-faulty situation. In particular, this means that the initialization action of $\mathcal{SSBB}(\mathcal{B})$ corresponds to the the initialization action of reset and, of course, $\mathcal{SSBB}(\mathcal{B})$ will take in account of the requests for $\mathcal{B}$ (using $\mathcal{B}.Request_r$) instead of $\mathcal{B}$ it-self. $\mathcal{SSBB}(\mathcal{B})$ will reset the $\mathcal{B}$ variables (using $\mathcal{B}.Init$) so that the system reaches a *normal starting configuration* of $\mathcal{A}$ and, then, give the execution control to $\mathcal{B}$ so that it performs the task $\mathcal{T}$. A well-known technique to perform a reset in distributed systems is based on the *Propagation of Information with Feedback* (PIF). Many PIF protocol for arbitrary networks have been proposed in the self-stabilizing literature, e.g., [1, 2] as well as in the snap-stabilizing literature, e.g., [5, 3, 7]. A PIF scheme can be informally described as follows: the root processor initiates the protocol by broadcasting a message *m* (*broadcast phase*), then, every non-root processor will send an acknowledgment to the root for the receipt of *m* (*feedback phase*). Using the PIF scheme, the reset protocol can be performed as follows: (i) the root processor broadcasts an "abort" message, (ii) upon the reception of the message, the processors abort the execution of $\mathcal{B}$, (iii) finally, the processors reset their $\mathcal{B}$ variables during the feedback phase. To implement $\mathcal{SSBB}$, we need to use a snap-stabilizing PIF protocol working under a distributed unfair daemon (to apply our technique to self-stabilizing protocols working with any arbitrary daemon, we need a reset protocol that works with the most general daemon). Such a protocol is provided in [8].

Algorithm 1 $Reset(\mathcal{B})$ for $p = r$

Inputs: $Neig_p$: set of (locally) ordered neighbors of p;
Constants: $P_p = \perp$; $L_p = 0$;
Variables: $S_p \in \{B,F,P,C\}$; $Que_p \in \{Q,R,A\}$;
Macro: $Child_p = \{q \in Neig_p :: (S_q \neq C) \wedge (P_q = p) \wedge (L_q = L_p + 1) \wedge [(S_q \neq S_p) \Rightarrow (S_p \in \{B,P\} \wedge S_q = F)]\}$;

General Predicates:

$CFree(p) \equiv (\forall q \in Neig_p :: S_q \neq C)$
$Leaf(p) \equiv [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (P_q \neq p)]$
$BLeaf(p) \equiv (S_p = B) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (S_q = F)]$
$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$

Guards:

$Broadcast(p) \equiv (S_p = C) \wedge Leaf(p)$
$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$
$PreClean(p) \equiv (S_p = F) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (S_q \in \{F,C\})]$
$Cleaning(p) \equiv (S_p = P) \wedge Leaf(p)$
$Require(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))] \vee [(Que_p = A) \wedge (\exists q \in Neig_p :: (S_q \neq C) \wedge ((Que_q = Q) \vee (q \in Child_p \wedge Que_q = R)))]]$
$Answer(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q)]$

Actions:

PIF Part:

B-action	/* Initialization */	::	$(\mathcal{B}.Request_p = Wait) \wedge \mathcal{B}.End_p \wedge Broadcast(p)$	$\rightarrow$	$S_p := B$; $Que_p := Q$; $\mathcal{B}.Request_p := In$;
F-action		::	$Feedback(p)$	$\rightarrow$	$S_p := F$; $\mathcal{B}.Init_p$; $\mathcal{B}.End_p := false$;
P-action		::	$PreClean(p)$	$\rightarrow$	$S_p := P$;
C-action		::	$Cleaning(p)$	$\rightarrow$	$S_p := C$;
T-action	/* Termination */	::	$(\mathcal{B}.Request_p = In) \wedge \mathcal{B}.End_p \wedge (S_p = C)$	$\rightarrow$	$\mathcal{B}.Request_p := Out$;

Question Part:

QR-action		::	$Require(p)$	$\rightarrow$	$Que_p := R$;
QA-action		::	$Answer(p)$	$\rightarrow$	$Que_p := A$;

Snap-Stabilizing PIF. A snap-stabilizing PIF protocol satisfies the following specification: starting from any configuration, when *r* has a message *m* to broadcast, then it starts the broadcast in a finite time. Then, every other processor will both receive *m* and send an acknowledgment (for the receipt of *m*) which will reach *r* in a finite time.

Theorem 1 ([8]) *The PIF protocol proposed in [8] is snap-stabilizing under a distributed unfair daemon.*

As the distributed unfair daemon is the most general daemon, Theorem 1 implies that the protocol of [8], called in the following $\mathcal{PIF}$, works with any kind of daemon. The another important consequence of Theorem 1 is that, starting from any configuration, each PIF wave performed by $\mathcal{PIF}$ is bounded in terms of steps. We now roughly present the main actions and variables of $\mathcal{PIF}$ (see the technical report [7] for details). $\mathcal{PIF}$ is divided into three parts: the PIF, question, and correction parts, respectively. The PIF part is the most important part of the protocol because it contains the actions related to the three phases of a PIF wave: the broadcast phase, the feedback phase following the broadcast phase, and the cleaning phase which cleans the trace of the feedback phase so that the root is ready to broadcast a new message. The two other parts of the algorithm implement two mechanisms allowing the snap-stabilization of the PIF part. Due to the lack of space, we do not present these mechanisms here (see the technical report [7] for details). Informally, the PIF part maintains in every processor p two crucial variables for $\mathcal{SSBB}$:

- P_p. This variable points out to the processor from which p receives the broadcast message. So, $P_p \in Neig_p$ if $p \neq r$ and $P_p = \perp$ if $p = r$ (indeed, as *r* is the initiator of the broadcast, *r* never receives any message and P_r is a constant). A spanning tree of the network w.r.t. the P variables is dynamically built during the broadcast phase.
- S_p. Informally, S_p allows to know in which phase of the PIF the processor p is. S_p is set to B when p switches to the broadcast phase (B-$action$). Then, S_p is set to F when p switches to the feedback phase (F-$action$). The cleaning phase is managed with two states: P and C. After *r* detects the end of the feedback phase (*r* is the last processor which switches to the feedback phase), *r* initiates the propagation of the P value into the S variables following the computed spanning tree in order to inform all the processor of this termination (P-$action$). Then, the processors successively switches to C (C-$action$) in a bottom up fashion (from the leaves of the spanning tree to *r*) meaning that they now ready to receive another broadcast message. Hence, the PIF wave terminates when *r* sets S_r to C (C-$action$). Finally, note that two more states exists in S_p for $p \neq r$: EB and EF. But, they are used by the correction part only. So, we do not explain the goal of these states here.

We now recall some results about $\mathcal{PIF}$:

Property 1 *From [7], follows:*

1. *After* r *initiates a broadcast (B-$action$), the system eventually reaches a configuration where every processor is in the feedback phase associated to the broadcast of* r.
2. *From any configuration,* r *executes B-$action$ in at most $9N - 1$ rounds and $O(\Delta \times N^3)$ steps.*
3. *From any configuration, a complete PIF wave is executed in at most $15N - 3$ rounds and $O(\Delta \times N^3)$ steps.*

Remark 2 *By Property 1, from any configuration,* r *executes B-$action$ at most $9N - 1$ rounds. Actually, this time complexity corresponds to the following worst case: the maximal number of rounds starting from any configuration before the system reaches a configuration where B-$action$ at* r *is the only enabled action of the system (see [7] for details).*

We now explain how we build our $\mathcal{SSBB}$ protocol using the protocol of [7, 8].

Algorithm 2 $Reset(\mathcal{B})$ for $p \neq r$

Inputs: $Neig_p$: set of (locally) ordered neighbors of p;
Variables: $S_p \in \{B,F,P,C,EB,EF\}$; $P_p \in Neig_p$; $L_p \in \mathbb{N}$; $Que_p \in \{Q,R,W,A\}$;
Macros:

$Child_p = \{q \in Neig_p :: (S_q \neq C) \wedge (P_q = p) \wedge (L_q = L_p + 1) \wedge [(S_q \neq S_p) \Rightarrow ((S_p \in \{B,P\} \wedge S_q = F) \vee (S_p = EB))]\}$;
$Pre\_Potential_p = \{q \in Neig_p :: S_q = B\}$;
$Potential_p = \{q \in Neig_p :: \forall q' \in Pre\_Potential_p, L_q \leq L_{q'}\}$;

General Predicates:

$CFree(p) \equiv (\forall q \in Neig_p :: S_q \neq C)$
$Leaf(p) \equiv [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (P_q \neq p)]$
$BLeaf(p) \equiv (S_p = B) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (S_q = F)]$
$AnswerOk(p) \equiv (Que_p = A) \wedge [\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$
$GoodS(p) \equiv (S_p = C) \vee [(S_{P_p} \neq S_p) \Rightarrow ((S_{P_p} = EB) \vee (S_p = F \wedge S_{P_p} \in \{B,P\}))]$
$GoodL(p) \equiv (S_p \neq C) \Rightarrow (L_p = L_{P_p} + 1)$
$AbRoot(p) \equiv \neg GoodS(p) \vee \neg GoodL(p)$

Guards:

$EFAbRoot(p) \equiv (S_p = EF) \wedge AbRoot(p) \wedge [\forall q \in Neig_p :: (P_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF,C\})]$
$EBroadcast(p) \equiv (S_p \in \{B,F,P\}) \wedge [\neg AbRoot(p) \Rightarrow (S_{P_p} = EB)]$
$EFeedback(p) \equiv (S_p = EB) \wedge [\forall q \in Neig_p :: (P_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF,C\})]$
$Broadcast(p) \equiv (S_p = C) \wedge (Potential_p \neq \emptyset) \wedge Leaf(p)$
$Feedback(p) \equiv BLeaf(p) \wedge CFree(p) \wedge AnswerOk(p)$
$PreClean(p) \equiv (S_p = F) \wedge (S_{P_p} = P) \wedge [\forall q \in Neig_p :: (P_q = p) \Rightarrow (S_q \in \{F,C\})]$
$Cleaning(p) \equiv (S_p = P) \wedge Leaf(p)$
$Require(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))] \vee [(Que_p \in \{W,A\}) \wedge (\exists q \in Neig_p :: (S_q \neq C) \wedge ((Que_q = Q) \vee (q \in Child_p \wedge Que_q = R)))]]$
$Wait(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge (Que_p = R) \wedge (Que_{P_p} = R) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$
$Answer(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CFree(p)] \wedge (Que_p = W) \wedge (Que_{P_p} = A) \wedge (\forall q \in Child_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Neig_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$

Actions:

Correction Part:
$EC\text{-}action :: EFAbRoot(p) \rightarrow S_p := C;$
$EB\text{-}action :: EBroadcast(p) \rightarrow S_p := EB;$
$EF\text{-}action :: EFeedback(p) \rightarrow S_p := EF;$
PIF Part:
$B\text{-}action :: Broadcast(p) \rightarrow S_p := B; P_p := \min_{\prec_p}(Potential_p); L_p := L_{P_p} + 1; Que_p := Q;$
$F\text{-}action :: Feedback(p) \rightarrow S_p := F; \mathcal{B}.Init_p;$
$P\text{-}action :: PreClean(p) \rightarrow S_p := P;$
$C\text{-}action :: Cleaning(p) \rightarrow S_p := C;$
Question Part:
$QR\text{-}action :: Require(p) \rightarrow Que_p := R;$
$QW\text{-}action :: Wait(p) \rightarrow Que_p := W;$
$QA\text{-}action :: Answer(p) \rightarrow Que_p := A;$

SSBB Protocol. To build $\mathcal{SSBB}(\mathcal{B})$, we use the following composition technique. Let P_1 and P_2 be two protocols, the composition of P_1 and P_2, denoted $P_2 \circ_{|G} P_1$, will be the program satisfying the following conditions:

- $P_2 \circ_{|G} P_1$ contains all the variables and actions of P_1 and P_2.
- G is a predicate defined on the variables of P_1.
- Each action $L_i :: H_i \rightarrow S_i$ of P_2 is replaced into the protocol $P_2 \circ_{|G} P_1$ by $L_i :: G \wedge H_i \rightarrow S_i$.

Following these composition rules, $\mathcal{SSBB}(\mathcal{B}) = \mathcal{B} \circ_{|Ok(p)} Reset(\mathcal{B})$ where $Ok(p) \equiv (S_p = C)$. $Reset(\mathcal{B})$ is a slightly modified version of $\mathcal{PIF}$ (Algorithms 1 and 2). It is used for resetting the $\mathcal{B}$ variables when it is necessary. To that goal, we modify the guard of its initialization action: $B\text{-}action$ at r (the initialization action of $\mathcal{SSBB}(\mathcal{B})$) so that it is enabled only when a request for $\mathcal{B}$ occurs at the root ($\mathcal{B}.Request_r = Wait$) and $\mathcal{B}.End_r = true$ (in order to avoid the aborting a previous initiated wave of $\mathcal{B}$). Also, as previously explained, we modify the $F\text{-}action$ to reset the $\mathcal{B}$ variables using $\mathcal{B}.Init_p$ ($\forall p \in V$) and to set $\mathcal{B}.End_r$ to false (for the root only and so that the actions of $\mathcal{B}$ at r will be unlocked at the end of the reset) during the feedback phase. We use the predicate $Ok(p)$ into the composition so that each processor p aborts its local execution of $\mathcal{B}$ when receiving the reset ($B\text{-}action$) and until the local termination of the reset at p. Indeed, we already know that p continuously satisfies $S_p \neq C$ during its participation to a reset (see previous paragraph). So, while p participates to a reset, $Ok(p)$ is false and all actions of $\mathcal{B}$ in $\mathcal{SSBB}(\mathcal{B})$ are disabled at p. Finally, we

add an action, noted T-$action$, so that r switches $\mathcal{B}.Request_r$ from In to Out at the termination of each wave of $\mathcal{SSBB}(\mathcal{B})$.

4 Proof of Correctness

Let $\mathcal{A}$ be self-stabilizing wave protocol under a daemon $\mathcal{D}$ such that $\mathcal{A}$ has a unique initiator (r) and such that the decision actions of $\mathcal{A}$ are at the root only. Let $\mathcal{T}$ be the task solved by $\mathcal{A}$ in a self-stabilizing manner. Let $\mathcal{B}$ the modified version of $\mathcal{A}$ according to the explanation provided in the previous section. We now prove that the composite protocol $\mathcal{SSBB}(\mathcal{B})$ is snap-stabilizing for the specification of the task $\mathcal{T}$ under $\mathcal{D}$ ($\mathcal{D} \in \{SF,WF,UF\}$). First, as $\mathcal{A}$ is designed to solve the specific task $\mathcal{T}$ only, we make the following assumption about $\mathcal{B}$:

Assumption 1 *$\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables.*

We now prove $\mathcal{SSBB}(\mathcal{B})$ is a *fair composition* of Algorithms $Reset(\mathcal{B})$ and $\mathcal{B}$.

Definition 3 (Fair Execution [15]) *An execution e of the composite protocol $P_2 \circ_{|G} P_1$ is* fair *w.r.t. P_i ($i \in \{1,2\}$, if one of these conditions holds: (i) e is finite, (ii) e contains infinitively many steps of P_i, or (iii) e contains an infinite suffix in which no step of P_i is enabled.*

Theorem 2 *$\mathcal{SSBB}(\mathcal{B})$ is a fair composition of Algorithms $Reset(\mathcal{B})$ and $\mathcal{B}$.*

Proof. Assume, by the contradiction, that there exists at least one execution e of $\mathcal{SSBB}(\mathcal{B})$ which is not *fair*. Then, by Definition 3, e is infinite and two cases are possible:

- There exists an (infinite) suffix $\acute{e}$ of e where some actions of $\mathcal{B}$ are enabled but only actions of $Reset(\mathcal{B})$ are executed. Then, e' contains infinitively many steps of $Reset(\mathcal{B})$. Now, By Property 1 (Claim 2), each wave of $Reset(\mathcal{B})$ is performed in a finite number of steps. So, $Reset(\mathcal{B})$ executes waves infinitively often in $\acute{e}$. In e', the variables of $Reset(\mathcal{B})$ behaves as $\mathcal{PIF}$ which is snap-stabilizing by Theorem 1. So, during its first complete wave in $\acute{e}$, $Reset(\mathcal{B})$ performs, in particular, a complete feedback phase. Now, when r performs its feedback phase (F-$action$), it sets $\mathcal{B}.End_r$ to $false$ and, as no action of $Reset(\mathcal{B})$ can set $\mathcal{B}.End_r$ to $true$, $\mathcal{B}.End_r = false$ forever. This implies that the initialization action of $Reset(\mathcal{B})$ (i.e., B-$action$ of r) becomes disabled forever. So, $Reset(\mathcal{B})$ cannot execute waves infinitively often in $\acute{e}$, a contradiction.
- There exists an (infinite) suffix $\acute{e}$ of e where some actions of $Reset(\mathcal{B})$ are enabled but only actions of $\mathcal{B}$ are executed. In particular, this means that e' contains infinitively many steps of $\mathcal{B}$. Let $\gamma_{\acute{i}}$ be the first configuration of $\acute{e}$. According to the actions of $Reset(\mathcal{B})$ that are enabled in γ_i, we study the following three cases:
  (1) Assume that B-$action$ is enabled at r in γ_i, i.e., ($\mathcal{B}.Request_r$=$Wait$)$\wedge\mathcal{B}.End_r\wedge Broadcast(r)$ is satisfied in $\gamma_{\acute{i}}$. $\mathcal{B}.Request_r$ remains equal to $Wait$ until r executes B-$action$, so, $\mathcal{B}.Request_r$ is equal to $Wait$ forever from γ_i. Then, no action of $Reset(\mathcal{B})$ is executed in e' and $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 1). So, $Broadcast(r)$ is also satisfied from $\gamma_{\acute{i}}$ forever. Finally, F-$action$ is the only action of r that sets $\mathcal{B}.End_r$ to false, so, $\mathcal{B}.End_r = true$ forever from γ_i. Hence, B-$action$ is continuously enabled from γ_i and, by the contradiction, $\mathcal{D} = UF$ (i.e., $\mathcal{A}$ is self-stabilizing under an unfair daemon) because only the unfair daemon can prevent a continuously enabled action to be executed. Moreover, as $\mathcal{B}.End_r = true$ forever, all the initialization actions of $\mathcal{B}$ are disabled forever from γ_i. So, from γ_i, each step of the (infinite) execution contains some actions of $\mathcal{B}$ but no initialization action of $\mathcal{B}$. This execution of $\mathcal{B}$ corresponds to a possible execution of $\mathcal{A}$ under the unfair daemon. So, there exists a possible suffixe of execution of $\mathcal{A}$ in which $\mathcal{A}$ never starts. Hence, $\mathcal{A}$ is not self-stabilizing, a contradiction.
  (2) Assume that T-$action$ is enabled at r in γ_i, i.e., ($\mathcal{B}.Request_r = In$) $\wedge$ $\mathcal{B}.End_r \wedge (S_r = C)$ is satisfied in $\gamma_{\acute{i}}$. $\mathcal{B}.Request_r$ remains equal to In until r executes T-$action$, so, $\mathcal{B}.Request_r$ is equal to In forever from γ_i. Then, no action of $Reset(\mathcal{B})$ is executed in e' and $\mathcal{B}$ does not

write into the $Reset(\mathcal{B})$ variables (Assumption 1). So, S_r remains equal to C forever from γ_i. Finally, as $F\text{-}action$ is the only action of *r* that can set $\mathcal{B}.End_r$ to false, $\mathcal{B}.End_r = true$ forever from γ_i. Hence, $T\text{-}action$ is continuously enabled from γ_i and, by the contradiction, $\mathcal{D} = UF$ (i.e., $\mathcal{A}$ is self-stabilizing under an unfair daemon) because only the unfair daemon can prevent a continuously enabled action to be executed. Moreover, as $\mathcal{B}.End_r = true$ forever, all the initialization actions of $\mathcal{B}$ are disabled forever from γ_i and, as explained in (1), since actions of $\mathcal{B}$ are executed infinitively often while $\mathcal{B}.End_r$ remains equal to true, this means that $\mathcal{A}$ is not self-stabilizing under an unfair daemon, a contradiction.

(3) Assume that some actions of $Reset(\mathcal{B})$ different of $B\text{-}action$ and $T\text{-}action$ at *r* are enabled in γ_i. Such actions depend on internal variables of $Reset(\mathcal{B})$ only. So, as no action of $Reset(\mathcal{B})$ are executed in e' and $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 1), these actions are continuously enabled from γ_i and, by the contradiction, $\mathcal{D} = UF$. Moreover, $F\text{-}action$ is the only action of *r* that sets $\mathcal{B}.End_r$ to false, so, *r* never more sets $\mathcal{B}.End_r$ to false from γ_i (no action of $Reset(\mathcal{B})$ are executed in e'). So, if $\mathcal{B}.End_r = false$ at the beginning of e', then, as $\mathcal{B}$ executes actions infinitively many times, $\mathcal{B}$ eventually decides otherwise there exists a possible suffixe of execution of $\mathcal{A}$ in which $\mathcal{A}$ never decides and, in this case, $\mathcal{A}$ is not self-stabilizing, a contradiction. Now, $\mathcal{B}.End_r$ is set to true since $\mathcal{B}$ decides. So, as no action of $\mathcal{B}$ can set $\mathcal{B}.End_r$ to true, this means that $\mathcal{B}.End_r = true$ forever and, as explained in before, since actions of $\mathcal{B}$ are executed infinitively often while $\mathcal{B}.End_r$ remains equal to true, this means that $\mathcal{A}$ is not self-stabilizing, a contradiction.

□

The next lemma shows that from any configuration where the system is waiting for a computation of the task $\mathcal{T}$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in a finite time.

Lemma 1 *Starting from any configuration where $\mathcal{B}.Request_r = Wait$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in a finite time.*

Proof. Assume, by the contradiction, that $B\text{-}action$ is never executed. First, $B\text{-}action$ is the only action that can modify $\mathcal{B}.Request_r$ when it is equal to $Wait$. So, $\mathcal{B}.Request_r = wait$ forever. Then, by Theorem 2, Property 1 (Claim 2), and Assumption 1, the system reaches, in a finte number of steps, a configuration γ from which $\forall p \in V$, no action at p in $\mathcal{R}eset(\mathcal{B})$ is enabled forever. Actually, every processor are waiting for a new broadcast in γ_i, i.e., $\forall p \in V$, $S_p = C$ in γ_i but the broadcast is not initiated because $\mathcal{B}.End_r = false$ (remember $\mathcal{B}.Request_r = wait$ forever). In particular, this means that $Broadcast(r)$ is satisfied forever. Now, from γ_i, $\mathcal{B}$ behaves has $\mathcal{A}$ (in particular, no action of $\mathcal{B}$ is locked by $Ok(p)$). So, $\mathcal{B}$ eventually decides and $\mathcal{B}.End_r$ is set to true. Since, $\mathcal{B}.End_r = false$, $B\text{-}action$ is enabled, a contradiction. □

The next lemma shows that since *r* requests a computation of the task $\mathcal{T}$ (formally, since the predicate $ApplicationRequest(r)$ of the external action IR is satisfied), the system eventually takes this request into account by executing $\mathcal{B}.Request_r := Wait$.

Lemma 2 *Starting from any configuration where* r *requests a $\mathcal{SSBB}(\mathcal{B})$ wave,* r *executes IR in a finite time.*

Proof. Assume, by the contradiction, that, starting from a configuration γ_i, *r* requests a wave of $\mathcal{SSBB}(\mathcal{B})$ (i.e., $ApplicationRequest(r)$ is satisfied) but never executes IR (i.e., $\mathcal{B}.Request_r := Wait$).

- Assume that $\mathcal{B}.Request_r = Out$ in γ_i. In such a configuration, IR is enabled (the guard of IR is $ApplicationRequest(r) \wedge (Request_r = Out)$). Also, no action of $SSBB(\mathcal{B})$ can modify $\mathcal{B}.Request_r$ until *r* sets $\mathcal{B}.Request_r$ to $Wait$ by IR. So, IR is continuously enabled at *r* and, by the contradiction, $\mathcal{D} = UF$ (i.e., $\mathcal{A}$ is self-stabilizing under an unfair daemon). As IR is never executed, $\mathcal{B}.Request_r$ remains equal to Out forever. In particular, this means that $B\text{-}action$ is disabled at *r* forever. By Theorem 2 and Assumption 1, $\mathcal{B}$ does not perturb the behavior of $Reset(\mathcal{B})$. So, Property 1 (Claim 2) implies that the system eventually reaches a configuration γ_j from which no action of $Reset(\mathcal{B})$ are enabled forever (remember that *r* never executes $B\text{-}action$). Actually, every processor are waiting for a new broadcast in γ_j, i.e., $\forall p \in V$, $S_p = C$ in γ_j. From γ_j, only actions of $\mathcal{B}$ are executed infinitively

many times (otherwise IR is eventually chosen by the daemon). If $\mathcal{B}.End_r = false$ in γ_j, then $\mathcal{B}$ behaves as $\mathcal{A}$ (remember that $\forall p \in V$, $S_p = C$ forever) and $\mathcal{B}$ eventually decides. Now, $\mathcal{B}.End_r$ is set to true since $\mathcal{B}$ decides. So, as no action of $\mathcal{B}$ can set $\mathcal{B}.End_r$ to true, this means that $\mathcal{B}.End_r$ is eventually true forever. Moreover, since $\mathcal{B}.End_r = true$ forever, all the initialization actions of $\mathcal{B}$ are disabled forever. So, eventually, each step of the (infinite) execution contains actions of $\mathcal{B}$ only but no initialization action of $\mathcal{B}$. This execution of $\mathcal{B}$ corresponds to a possible execution of $\mathcal{A}$ under the unfair daemon. So, there exists a possible suffixe of execution of $\mathcal{A}$ in which $\mathcal{A}$ never starts. Hence, $\mathcal{A}$ is not self-stabilizing, a contradiction.

- Assume that $\mathcal{B}.Request_r = In$ in γ_i. From γ_i, $\mathcal{B}.Request_r$ will remain equal to In until *r* executes T-$action$ (see Algorithms 1 and 2). Assume that *r* eventually executes T-$action$. By T-$action$, $\mathcal{B}.Request_r := Out$ and, from the previous case, *r* eventually executes IR, a contradiction. So, assume, by the contradiction, that *r* never executes T-$action$ and $\mathcal{B}.Request_r$ remains equal to In forever. In particular, this means that B-$action$ is disabled at *r* forever from γ_i. By Theorem 2 and Assumption 1, we know that $\mathcal{B}$ does not perturb the behavior of $Reset(\mathcal{B})$. So, Property 1 (Claim 2) implies that the system eventually reaches a configuration γ_j from which no action of $Reset(\mathcal{B})$ are enabled forever (remember that *r* never executes B-$action$). This configuration corresponds to a configuration of $\mathcal{PIF}$ where every processor are waiting for a new broadcast, i.e., $\forall p \in V$, $S_p = C$. From γ_j, only actions of $\mathcal{B}$ are executed and, as in the previous case, $\mathcal{B}.End_r$ is eventually true forever. As the guard of T-$action$ is $(\mathcal{B}.Request_r = In) \wedge \mathcal{B}.End_p \wedge (S_p = C)$, T-$action$ is eventually continuously enabled and the only enabled action of $Reset(\mathcal{B})$. Theorem 2 implies then that T-$action$ is eventually executed, a contradiction.
- Assume that $\mathcal{B}.Request_r = Wait$ in γ_i. Then, by Lemma 1, we know that *r* eventually executes B-$action$. By B-$action$, $\mathcal{B}.Request_r := In$ and we retreive the previous case, a contradiction.

□

By Lemmas 1 and 2, the following theorem. This theorem means that, since *r* requests an execution of $\mathcal{SSBB}(\mathcal{B})$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in a finite time.

Theorem 3 *Starting from any configuration where* r *requests a* $\mathcal{SSBB}(\mathcal{B})$ *wave, the requested* $\mathcal{SSBB}(\mathcal{B})$ *wave is eventually initiated.*

The next theorem show that each computation of the task $\mathcal{T}$ initiated by *r* is executed as expected.

Theorem 4 *From any configuration where* r *initiates* $\mathcal{SSBB}(\mathcal{B})$*, the system computes the task* $\mathcal{T}$ *as expected.*

Proof. $\mathcal{SSBB}(\mathcal{B})$ starts by initiating a $Reset(\mathcal{B})$ wave with B-$action$ at *r*: *r* switches S_r to B. This wave terminates when *r* switches S_r to C by C-$action$ (see [8]). So, during the whole wave, *r* continuously satisfies $S_p \neq C$, i.e., $\neg Ok(r)$, and *r* cannot execute any action of $\mathcal{B}$. In particular, *r* cannot initiate $\mathcal{B}$ before $Reset(\mathcal{B})$ terminates (at *r*).

By Theorem 2 and Assumption 1, $\mathcal{B}$ does not perturb the behavior of $Reset(\mathcal{B})$. Now, by Property 1 (Claim 1), after *r* initiates a broadcast (B-$action$), the system eventually reaches a configuration γ_i where every processor is in the feedback phase associated to the broadcast of *r*. So, as the $\mathcal{B}$ variables are locally reset at p when p switches S_p to F and no action of $\mathcal{B}$ are executed by p while $S_p \neq C$ ($(S_p \neq C) \Rightarrow \neg Ok(p)$), γ_i corresponds to the configuration (w.r.t. the $\mathcal{B}$ variables) generated by $\mathcal{B}.Init$ (i.e., the normal starting configuration of $\mathcal{A}$). Also, note that $\mathcal{B}.End_r$ is set to false when *r* executes F-$action$. Owing the fact that the system reaches the configuration γ_i induces two other consequences:

1. From γ_i, no processor will execute an action of $\mathcal{B}$ before *r* initiates $\mathcal{B}$.
2. From γ_i, the system contains no abnormal behavior related to $Reset(\mathcal{B})$.

Hence, when the reset terminates at *r* by C-$action$ (i.e. $S_r := C$), the system is in a configuration γ_j where $\forall p \in V$, $S_p = C$ (the normal starting configuration of $Reset(\mathcal{B})$, see [8]), the $\mathcal{B}$ variables describe the configuration generated by $\mathcal{B}.Init$, and $\mathcal{B}.End_r = false$. From γ_j, no reset will be initiated before $\mathcal{B}.End_r$

(see the guard of $B\text{-}action$), i.e., before the decision associated to the execution of $\mathcal{B}$ that *r* will initiate. So, from γ_j and until $\mathcal{B}.End_r$, $\forall p \in V$, $S_p = C$ and only $\mathcal{B}$ works. Hence, $\mathcal{B}$ works as $\mathcal{A}$ when it is stabilized, i.e., from γ_j, $\mathcal{B}$ performs the specific task $\mathcal{T}$ as expected. □

By Remark 1, Theorems 3 and 4, follows:

Theorem 5 *$\mathcal{SSBB}(\mathcal{B})$ is snap-stabilizing for the specification of $\mathcal{T}$ under the daemon $\mathcal{D}$.*

4.1 Complexity Analysis

Space Complexity. Let $\mathcal{M}(\mathcal{A})$ be the memory requirement of $\mathcal{A}$. The variables of $\mathcal{B}$ differ from $\mathcal{A}$ by just a boolean at the root ($\mathcal{B}.End_r$). So, the memory requirement of $\mathcal{B}$ is in the same order than that of $\mathcal{A}$ and by taking into account the variables of $Reset(\mathcal{B})$, follows:

Theorem 6 *The memory requirement of $SSBB(\mathcal{B})$ is $O(\log(N) + \log(\Delta) + \mathcal{M}(\mathcal{A}))$ bits per processor.*

Time Complexity. For the following results, we assume that $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D}$ such that $\mathcal{D} \in \{WF,UF\}$ and we recall that the external request is managed by the action labelled IR.

Let $R_1(\mathcal{A})$ be the maximal number of rounds starting from any configuration before *r* decides in $\mathcal{A}$. Let $R_2(\mathcal{A})$ be the maximal number of rounds that $\mathcal{A}$ requires to perform the task $\mathcal{T}$ starting from the configuration generated by $\mathcal{B}.Init$.

Theorem 7 *If $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D}$ such that $\mathcal{D} \in \{WF,UF\}$, then, starting from any configuration where* r *requests a $\mathcal{SSBB}(\mathcal{B})$ wave, the requested $\mathcal{SSBB}(\mathcal{B})$ wave is initiated in $O(N+R_1(\mathcal{A})+R_2(\mathcal{A}))$ rounds.*

Proof. Assume that, from a configuration γ_i, *r* requests a wave of $\mathcal{SSBB}(\mathcal{B})$ (i.e., $ApplicationRequest(r)$ is satisfied). According to $\mathcal{B}.Request_r$, three cases are then possible:

- $\mathcal{B}.Request_r = Out$ in γ_i. In such a configuration, IR is enabled (the guard of IR is $Application\text{-}Request(r) \wedge (Request_r = Out)$). Also, no action of $SSBB(\mathcal{B})$ can modify $\mathcal{B}.Request_r$ until $\mathcal{B}.Request_r$ is set to $Wait$ by IR (see Algorithms 1 and 2). So, IR is continuously enabled at *r* and, *r* executes IR, i.e., $\mathcal{B}.Request_r := Wait$, in at most one round. Then, $\mathcal{B}.Request_r$ is continuously equal to $Wait$ until *r* initiates $\mathcal{SSBB}(\mathcal{B})$ by $B\text{-}action$ (see Algorithms 1 and 2). Also, as $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 1), in the worst case, the system reaches a configuration γ_j from which $Broadcast(r)$ is continuously satisfied and no action of $Reset(\mathcal{B})$ different of $B\text{-}action$ at *r* is enabled until *r* executes $B\text{-}action$ in at most $9N-2$ rounds by Property 1 (Claim 2) and Remark 2. This configuration corresponds to a configuration of $\mathcal{PIF}$ where every processor are waiting for a new broadcast, i.e., $\forall p \in V$, $S_p = C$. Now, in at most $R_1(\mathcal{A})$ rounds from γ_j, the system reaches a configuration γ_k from which $\mathcal{B}.End_r$ is continuously true. Thus, $B\text{-}action$ becomes continuously enabled at *r* from γ_k and *r* executes $B\text{-}action$ in the next round. Hence, starting from any configuration where $\mathcal{B}.Request_r = Out$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in at most $9N + R_1(\mathcal{A})$ rounds.
- $\mathcal{B}.Request_r = In$ in γ_i. As $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 1), in the worst case, the system reaches a configuration γ_j from which $Broadcast(r)$ is continuously satisfied and no action of $Reset(\mathcal{B})$ different of $B\text{-}action$ at *r* is enabled until *r* executes $B\text{-}action$ in at most $9N-2$ rounds by Property 1 (Claim 2) and Remark 2. This configuration corresponds to a configuration of $\mathcal{PIF}$ where every processor are waiting for a new broadcast, i.e., $\forall p \in V$, $S_p = C$. Then, in at most $R_1(\mathcal{A})$ rounds from γ_j, the system reaches a configuration from which $\mathcal{B}.End_r$ is continuously true. As $B\text{-}action$ is disabled until $\mathcal{B}.Request_r = Wait$, two rounds are necessary so that $\mathcal{B}.Request_r$ switches from In to Out by $T\text{-}action$ ($Broadcast(r) \Rightarrow (S_r = C)$) and from Out to $Wait$ by Action IR. Then, $B\text{-}action$ will be continuously enabled at *r* and *r* will execute it in the next round. Hence, starting from any configuration where $\mathcal{B}.Request_r = In$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in at most $9N + R_1(\mathcal{A}) + 1$ rounds.

- $\mathcal{B}.Request_r = Wait$ in γ_i. In this case, the system has to perform a complete $SSBB(\mathcal{B})$ wave before r satisfies $\mathcal{B}.Request_r = Out$. A $\mathcal{SSBB}(\mathcal{B})$ wave becomes by a reset of the $\mathcal{B}$ variables (a $Reset(\mathcal{B})$ wave). $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables by Assumption 1. So, actions of $Reset(\mathcal{B})$ are executed like in $\mathcal{PIF}$ except for B-$action$ at r (which now also depends on $\mathcal{B}.End_r$). So, compared to the round complexities of a complete PIF wave (at most $15N - 3$ rounds, by Property 1 (Claim 3) and similary to the previous cases, we have an additional cost of $R_1(\mathcal{A})$ rounds before $\mathcal{SSBB}(\mathcal{B})$ starts. After the initialization action (B-$action$ at r), $\mathcal{B}.Request_r = In$ and $Reset(\mathcal{B})$ works with a same cost than $\mathcal{PIF}$. So, the cost of the reset is globally at most $15N + R_1(\mathcal{A}) - 3$ rounds. After the reset, the system is in the configuration generated by $\mathcal{B}.Init$ ($\mathcal{SSBB}(\mathcal{B})$ is snap-stabilizing by Theorem 5) and $R_2(\mathcal{A})$ additional rounds are necessary to perform the specific task $\mathcal{T}$. Finally, after performing $\mathcal{T}$, $\mathcal{SSBB}(\mathcal{B})$ terminates the wave with T-$action$: $\mathcal{B}.Request_r := Out$ (this latter action is executed in at most one round). After T-$action$, the system is in a configuration where $\forall p \in V$, $S_p = C$, i.e., the normal starting configuration of $\mathcal{PIF}$ (indeed, Property 1 implies that the abnormal behavior related to $\mathcal{PIF}$ are erased from the system during the first wave), $\mathcal{B}.Request_r = Out$, and $\mathcal{B}.End_r = true$. From such a configuration, the root executes IR followed by B-$action$ in the two next steps (resp. rounds): they are the only enabled action of the system. Hence, starting from any configuration where $\mathcal{B}.Request_r = Wait$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in at most $15N + R_1(\mathcal{A}) + R_2(\mathcal{A})$ rounds.

□

Corollary 4.1 *If $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D}$ such that $\mathcal{D} \in \{WF, UF\}$, then, starting from any configuration, a complete requested $SSBB(\mathcal{B})$ wave is executed in $O(N + R_1(\mathcal{A}) + R_2(\mathcal{A}))$ rounds.*

For the following result, we assume that $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D} = UF$. We have proved that, if $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D} = UF$, then $SSBB(\mathcal{B})$ is snap-stabilizing under a daemon $\mathcal{D} = UF$. This means, in particular, that $\mathcal{B}$ can only execute a finite number of actions between each action of $Reset(\mathcal{B})$. Actually, this number of actions, noted $S(\mathcal{A})$, is equal to the maximal number of steps starting from any configuration so that $\mathcal{A}$ decides and then reaches a configuration from which r executes an initialization action (n.b., in the worst case, the unfair daemon prevents $\mathcal{A}$ to execute an initialization action until the system reaches a configuration where only the initialization actions are enabled). Actually, in $\mathcal{B}$, this number corresponds to the maximal number of actions that $\mathcal{B}$ can execute to set $\mathcal{B}.End_r$ to $true$ and then reaches a configuration where none of its actions are enabled (the initialization actions are disabled because $\mathcal{B}.End_r = true$).

Theorem 8 *If $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D} = UF$, then, starting from any configuration where* r *requests a $\mathcal{SSBB}(\mathcal{B})$ wave, the requested $\mathcal{SSBB}(\mathcal{B})$ wave is initiated in $O(\Delta \times N^3 \times S(\mathcal{A}))$ steps.*

Proof. In the proof of Theorem 7, we have seen that, in the worst case, a requested $\mathcal{SSBB}(\mathcal{B})$ wave is initiated after a complete non-requested wave of $\mathcal{SSBB}(\mathcal{B})$. By Property 1 (Claim 3), we know that this non-requested wave contains $O(\Delta \times N^3)$ actions of $Reset(\mathcal{B})$ (i.e., the steps complexities of $\mathcal{PIF}$ provided in [7] except for a constant factor due to the T-$action$ which is thus eliminated). Also, we have stated that at most $S(\mathcal{A})$ actions of $\mathcal{B}$ are executed between each action of $Reset(\mathcal{B})$. Hence, a loose estimate (in terms of steps) of the delay to start a requested $SSBB(\mathcal{B})$ wave corresponds to the product of these two complexities and the theorem holds. □

Corollary 4.2 *If $\mathcal{A}$ is self-stabilizing under a daemon $\mathcal{D} = UF$, then, starting from any configuration, a complete requested $SSBB(\mathcal{B})$ wave is executed in $O(\Delta \times N^3 \times S(\mathcal{A}))$ steps.*

5 Example

In this section, we present the self-stabilizing depth-first token circulation ($DFTC$) protocol of Huang and Chen [12] (due to the lack of space, the protocol has been moved in the appendix). We then explain how to modify it to obtain a protocol which can be snap-stabilized by our transformer ($\mathcal{SSBB}$).

The $DFTC$ protocol of Huang and Chen [12]. In arbitrary rooted networks, a $DFTC$ protocol works as follows: a token is first created at the root and, then, is passed from one processor to another in the depth-first order such that every processor eventually gets it during a single traversal. In [12], Huang and Chen present and prove the self-stabilizing $DFTC$ protocol noted $\mathcal{DFS}$ in the following. From [12], follows:

Theorem 9 ([12]) *$\mathcal{DFS}$ of [12] is self-stabilizing assuming a weakly fair daemon.*

Informally, $\mathcal{DFS}$ is divided ino two parts. The first part manages the token circulation strictly speaking. The other part handles abnormal behaviors due to the initial configuration. We focus first on the token circulation part. This part maintains two variables: D and C. D is a *descendant* pointor variable; C, a *color* variable. The token circulation uses two colors: 1 and 2. At the beginning of a new circulation, the root switches to a color different from the color of all the other processors. A processor having the token searches its neighbors to find one with a different color. The processor then passes the token to the neighbor if such a neighbor exists. Otherwise, it backtracks the token to its parent - the processor which passed the token to it. A processor changes its color to the color of its parent when its receives the token. In this way, all visited processors in the current circulation have the same color of the root, and all unvisited processors have a different color. The descendant relationship is indicated by variable D and this relationship is destroyed by letting $D_p := NULL$ when the token backtracks from p. Starting from the root and tracing through the descendant pointors, a *segment* of processors can be described with the token on the *front* of it. The segment lengthens when the token moves to an unvisited processor, and shrinks when the token backtracks. The token finally backtracks to the root when all the processors are visited. The root then changes its color and initiates a new circulation. We now explain the error handling strategy. First, due to the initial configuration, the D value of some processors may describe a cycle. A *level* variable L is thus used for detecting such cycles. The level of the root is fixed to 0. Levels of others processors have a value from 1 to $n-1$. During a circulation, a processor computes its level when it receive the curent token for the first time: its level is set to one plus the level of its parent. When a processor p is in a segment and does not satisfy $L_p = L_q + 1$ where q is its parent, it know that it is in a cycle. So, p break the circle by setting D_p to $NULL$. Then, the system may contain some illegal segments, i.e., the segment rooted at another processor than r. To erase such illegal segments, the protocol uses an additionnal color: $ERROR$. The root of an illegal segment knows it is in an error state and hence changes its color to $ERROR$. The error color then propagates along the D pointors to the front of the segment. When the parent of the front processor sees the color of the front processor is already changed to $ERROR$, it drops the front processor away by setting the pointor D to $NULL$. The dropped processor then recovers it self by changing its color to a normal one. Repeating the dropping and recovering process will correct the processors on the illegal segments.

How to snap-stabilize $\mathcal{DFS}$ using $\mathcal{SSBB}$. First, from the previous paragraph, we know that:

- $\mathcal{DFS}$ is a protocol with a unique initiator: r.
- The decision actions of $\mathcal{DFS}$ occurs at r only: the token finally backtracks to the root when all the processors are visited.

So, by Theorem 5, we know that a slightly modified version of $\mathcal{DFS}$ can be snap-stabilized by $\mathcal{SSBB}$. According to the principles exposed in Section 3, we now explain how to modify $\mathcal{DFS}$ into $\mathcal{DFS}prim$ such that $\mathcal{SSBB}(\mathcal{DFS}prim)$ is a snap-stabilizing $DFTC$ protocol assuming a weakly fair daemon:

1. A boolean variable End_r must be declared in $\mathcal{DFS}$.
2. In $\mathcal{DFS}$, r decides when setting D_r to $NULL$. So, we must modify each action of $\mathcal{DFS}$ such that $D_r := NULL$ appears in its statement so that each time $D_r := NULL$ is executed, $End_r := true$ is also executed.
3. Finally, from the previous paragraph, we know that a normal starting configuration of $\mathcal{DFS}$ satisfies $\forall p, q \in V, D_p = NULL \wedge C_p = C_q \wedge C_p \neq ERROR$. So, a normal starting configuration of $\mathcal{DFS}$ can be the following: $\forall p \in V, D_p = NULL \wedge C_p = 1$. Hence, we can define in $\mathcal{DFS}$ the macro $Init_p$ ($\forall p \in V$) with the following assignments: $D_p := NULL; C_p := 1$.

With such modifications, we obtain a protocol $\mathcal{DFS}prim$ and, by Theorem 5, the following theorem holds:

Theorem 10 *$\mathcal{SSBB}(\mathcal{DFS}prim)$ is a snap-stabilizing $DFTC$ protocol assuming a weakly fair daemon.*

$\mathcal{DFS}$ of Huang and Chen does not works assuming an unfair daemon. Indeed, under an unfair deamon, a possible execution of $\mathcal{DFS}$ is the following: the protocol can perform infinitively often uncomplete token circulation because some isolated processors p satisfying $D_p = NULL \wedge C_p = ERROR$ remains in the network. This is due to the fact that a processor that holds the token from the root simply ignores its neighbors such that $D = NULL \wedge C = ERROR$. However, starting from any configuration, if the unfair daemon eventually blocks the progression the legal segment, then this blocking can last only a finite number of steps because the number of actions that can be executed, the actions on the legal segment apart, is finite. So, this means that the unfair daemon cannot prevent forever the tokens from the root to circulate in the network. this also implies that the unfair daemon cannot prevent forever the root to decide. Transposed to $\mathcal{DFS}prim$, these properties insures that:

1. Only a finite number of actions of $\mathcal{DFS}prim$ can be executed before $End_r := true$.
2. Since $End_r = true$, only a finite number of actions of $\mathcal{DFS}prim$ before $Reset(\mathcal{DFS}prim)$ moves (indeed, since $End_r = true$, the initialization action of $\mathcal{DFS}prim$ are disabled until $F\text{-}action$ at r sets End_r to false).

Clearly, 1. and 2. implies the following theorem:

Theorem 11 *$\mathcal{SSBB}(\mathcal{DFS}prim)$ is a snap-stabilizing $DFTC$ protocol assuming an unfair daemon.*

We now focus on the complexity of $\mathcal{SSBB}(\mathcal{DFS}prim)$. By Theorem 7, we know that, starting from any configuration, a requested wave of $\mathcal{SSBB}(\mathcal{DFS}prim)$ is initiated in $O(N + R_1(\mathcal{DFS}) + R_2(\mathcal{DFS}))$ where $R_1(\mathcal{DFS})$ is the maximal number of rounds starting from any configuration before r decides in $\mathcal{DFS}$ and $R_2(\mathcal{DFS})$ be the maximal number of rounds that $\mathcal{DFS}$ requires to perform a depth-first token circulation starting from a configuration γ_1 where $\forall p \in V$, $D_p = NULL \wedge C_p = 1$. In the same way, by Corollary 4.1, starting from any configuration, a complete requested wave of $\mathcal{SSBB}(\mathcal{DFS}prim)$ is executed in $O(N + R_1(\mathcal{DFS}) + R_2(\mathcal{DFS})$. Clearly, starting from any configuration, r decides in $\mathcal{DFS}$ in $O(N)$ rounds and starting from γ_i, a depth-first token circulation is also performed in $O(N)$ rounds. So, $R_1(\mathcal{DFS})$ and $R_2(\mathcal{DFS})$ are both in $O(N)$ rounds and follows:

Theorem 12 *Starting from any configuration, a requested depth-first token circulation is initiated (resp. performed) using $\mathcal{SSBB}(\mathcal{DFS}prim)$ in $O(N)$ rounds.*

This latter result is very surprising because $\mathcal{DFS}$ alone stabilizes in $\Omega(D \times N)$ rounds. Actually, Theorems 11 and 12 show that our transformer ($\mathcal{SSBB}$) allows not only to snap-stabilize some self-stabilizing protocols but also, in some case, it enhances the fairness and the time complexity of the protocols. We conjecture that we can obtain the same results with the self-stabilizing protocols in [13, 9].

6 Conclusion

In this paper, we propose a semi-automatic method allowing to snap-stabilize self-stabilizing wave protocols for arbitrary networks with a unique initiator and such that their decision action are executed at the root only. The snap-stabilizing solution we obtain with our technique works at least with the same daemon than the self-stabilizing protocol we want to snap-stabilizing. But, in some case like the depth-first token circulation protocol of Huang and Chen [12], we obtain a solution working with a weaker scheduling assumption. Moreover, the solution we obtain may have better time complexities than the self-stabilizing protocol we want to transform. For instance, despite the depth-first token circulation protocol of Huang and Chen stabilizes in $\Omega(D \times N)$ rounds, its snap-stabilizing version executes a requested depth-first token circulation (as expected) in $O(N)$ rounds.

References

[1] Y Afek, S Kutten, and M Yung. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486*, pages 15–28, 1990.

[2] B Awerbuch, B Patt-Shamir, and G Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[3] L Blin, A Cournier, and V Villain. An improved snap-stabilizing pif algorithm. In *DSN SSS'03 Workshop: Sixth Symposium on Self-Stabilizing Systems (SSS'03)*, pages 199–214. LNCS 2704, 2003.

[4] A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Fourth Workshop on Self-Stabilizing Systems*, pages 78–85, Austin, Texas, USA, June 1999. IEEE Computer Society Press.

[5] A Cournier, AK Datta, F Petit, and V Villain. Snap-stabilizing PIF algorithm in arbitrary rooted networks. In *22st International Conference on Distributed Computing Systems (ICDCS-22)*, pages 199–206. IEEE Computer Society Press, 2002.

[6] A Cournier, AK Datta, F Petit, and V Villain. Enabling snap-stabilization. In *23th International Conference on Distributed Computing Systems (ICDCS 2003)*, pages 12–19, Providence, Rhode Island USA, May 19-22 2003. IEEE Computer Society Press.

[7] A Cournier, S Devismes, and V Villain. Snap-stabilizing pif and useless computations. Technical Report LaRIA-2006-04, LaRIA, CNRS FRE 2733, 2006. Available at www.laria.u-picardie.fr/∼devismes/LaRIA-2006-04.pdf.

[8] A Cournier, S Devismes, and V Villain. Snap-stabilizing pif and useless computations. In *The Twelfth International Conference on Parallel and Distributed Systems (ICPADS'06)*, Minneapolis, USA, 2006. To appear.

[9] AK Datta, C Johnen, F Petit, and V Villain. Self-stabilizing depth-first token circulation in arbitrary rooted networks. In *SIROCCO'98, The 5th International Colloquium On Structural Information and Communication Complexity Proceedings*, pages 229–243. Carleton University Press, 1998.

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

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

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

[13] 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, Las Vegas (UNLV), USA, May 28-29 1995. Chicago Journal of Theoretical Computer Science.

[14] S Katz and KJ Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.

[15] G Tel. *Introduction to distributed algorithms*. Cambridge University Press, Cambridge, UK, Second edition 2001.

A Algorithm $\mathcal{DFS}$ of Huang and Chen

Algorithm 3 $\mathcal{DFS}$ $\forall p \in V$

Input :

$Neig_p$: set of neighbors;

Variables:

$D_p \in (Neig_p \setminus \{r\}) \cup \{NULL\}$; $C_p \in \{0,1,ERROR\}$; $L_p \in \mathbb{N}$ **if** $p \neq r$, $L_p = 0$ **otherwise**;

Macros:

$Pred_p = \{q :: q \in Neig_p \wedge D_q = p\}$;
$P_p = (q :: q \in Pred_p)$ **if** $|Pred_p| = 1$, $NULL$ **otherwise**;
$NB_p = |Neig_p \setminus \{r\}|$;
$NP_p = |Pred_p|$;
$SEARCH_p = D_p := q$ **if** $\exists q \in NB_p :: (C_q \neq C_p) \wedge (C_q \neq ERROR) \wedge (D_q = NULL)$, $D_p := NULL$ **otherwise**;

Predicates:

$Token(p) \equiv (((p = r) \wedge (D_p = NULL) \wedge (C_p \neq ERROR)) \vee ((p = r) \wedge (NP_p = 1) \wedge (D_p = NULL) \wedge (C_{P_p} \neq C_p) \wedge (C_{P_p} \neq ERROR) \wedge (C_p \neq ERROR)))$
$BToken(p) \equiv ((D_p \neq NULL) \wedge (D_{D_p} = NULL) \wedge (C_p = C_{D_p}) \wedge (C_p \neq ERROR))$
$Discontinous(p) \equiv ((D_p \neq NULL) \wedge (D_{D_p} \neq NULL) \wedge (L_{D_p} \neq L_p + 1))$
$IllegalRoot(p) \equiv ((p \neq r) \wedge (NP_p = 0) \wedge (D_p \neq NULL))$

Actions :

R_0 :: $Token(r) \rightarrow C_r := (C_r + 1) \bmod 2$; $SEARCH_r$;
R_1 :: $(p = r) \wedge Token(p) \wedge (L_{P_p} < N - 1) \rightarrow (C_p,L_p) := (C_{P_p},L_{P_p} + 1)$; $SEARCH_p$;
R_2 :: $BToken(p) \rightarrow SEARCH_p$;
R_3 :: $Discontinous(p) \rightarrow D_p := NULL$;
R_4 :: $IllegalRoot(p) \wedge (C_p \neq ERROR) \rightarrow C_p := ERROR$;
R_5 :: $(\exists q \in Neig_p :: D_q = p \wedge C \wedge C_q = ERROR) \wedge (C_p \neq ERROR) \rightarrow C_p := ERROR$;
R_6 :: $(D_p \neq NULL) \wedge ((D_{D_p},C_{D_p}) = (NULL,ERROR)) \rightarrow D_p := NULL$;
R_7 :: $(NP_p = 0) \wedge ((D_p,C_p) = (NULL,ERROR)) \rightarrow C_p := 0$;