From Self- to Snap- Stabilization

Alain Cournier, Stéphane Devismes, and Vincent Villain

LaRIA CNRS FRE 2733
University of Picardie Jules Verne, Amiens, France
`firstname.lastname@u-picardie.fr`
`http://www.laria.u-picardie.fr/~lastname`

Abstract. A *snap-stabilizing* protocol, starting from any 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. $\mathcal{B}$ is easy to obtain from $\mathcal{A}$ compared to the design of a snap-stabilizing protocol.

1 Introduction

The quality of a distributed system depends on its tolerance to faults. Many fault-tolerant schemes have been proposed. For instance, *self-stabilization* [1] allows to design a system tolerating arbitrary transient faults. 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 [2]. A *snap-stabilizing* protocol guarantees 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. Designing and proving self- or snap- stabilizing protocols is usually a complicated task. That is why some protocols, called *transformers*, were proposed to automatically perform such a task, e.g., [3,4]. In [3], Katz and Perry design a protocol that transforms almost all non-self-stabilizing protocols into self-stabilizing protocols. In [4], the authors propose a *transformer* providing a snap-stabilizing version of any protocol which can be self-stabilized with the transformer of [3], but, this transformer is designed in a higher level model than the one used in [3]. The transformers of [3,4] 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. Indeed, 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. 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. $\mathcal{B}$ is easy to obtain from $\mathcal{A}$ compared to the design of a snap-stabilizing protocol. In contrast with the solution in [4], our black box does not use any snapshot to snap-stabilize $\mathcal{B}$ and keeps the same fairness as 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 [5] into a snap-stabilizing token circulation.

The rest of the paper is organized as follows. In Section 2, we describe the model. In Section 3, we present and justify how our black box works. A sketch of proof and the complexity analysis are provided in Section 4. We show in Section 5 how to snap-stabilize the protocol of [5]. 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 $r \in V$. The particular processor r, called *root*, 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 Ng_p, are locally ordered by $\prec_p$. We also use the following notations: respectively, N is the size, Δ the degree, and D the diameter of the network. Our protocols are *semi-uniform*, i.e., each processor executes the same program except r. We consider a local shared memory model of computation (see [6]) where 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 simultaneously enabled 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* 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. Under a *strongly fair* daemon, every processor that is enabled infinitively often is chosen by the daemon infinitively often 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. 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 is *neutralized* 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 action in $\gamma_i \mapsto \gamma_{i+1}$. To compute the time complexity, we use the definition of *round* [7]. This definition captures the execution rate of the slowest processor in any execution. The 1^{st} *round* of $e \in \mathcal{E}$, noted e', is the minimal prefix of e containing the execution of one action or the neutralization of every enabled processor from the initial configuration. Let e'' be the suffix of e such that $e = e'e''$. The 2^{nd} *round* of e is the 1^{st} round of e'', and so on.

Definition 1 (Wave Protocol [6]). *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.*

Definition 2 (Snap-stabilization). *Let $\mathcal{T}$ be a task, and $\mathcal{S}_\mathcal{T}$ a specification of $\mathcal{T}$. A protocol $\mathcal{P}$ is snap-stabilizing for $\mathcal{S}_\mathcal{T}$ if and only if $\forall e \in \mathcal{E}$, e satisfies $\mathcal{S}_\mathcal{T}$.*

Consider a wave protocol having a unique initiator, r, and performing a specific task in a safe system. In the safe system, 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 system, the protocols achieve a convergence to a specified behavior of the system in a finite time. So, the execution of the first waves of such a protocol 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 system, i.e., the initialization is assumed to occur only upon an external request (see [4] for further details). So, in our protocols, we will explicitly mention this external request using the shared variable $Req_r \in \{W,I,O\}$ (noted $\mathcal{P}.Req_r$ for the specific protocol $\mathcal{P}$). We consider Req_r as an input into the algorithm of the protocol initiator (r). $Req_r = W$ means that an execution of the protocol is required. When the initialization of the protocol occurs, Req_r switches from W to I meaning that r has taking in account of the request. Finally, Req_r switches from I to O at the termination of the wave meaning that the system is now ready to receive another request. Of course, the switching of Req_r from W to I and from I to O is managed by the task itself while the switching from O to W (which means that another execution of the protocol is required) is managed externally. Note that all other transitions (for instance, I to W) are forbidden. The external action, noted IR, that manages the switching from O to W is of the following form:

$$IR :: AppliReq(r) \wedge (Req_r = O) \rightarrow Req_r := W; AppliRelease_r;$$

$AppliReq(r)$ is a predicate which is true when an application of the initiator r needs an execution of the snap-stabilizing protocol. $AppliRelease_r$ 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 $AppliReq(r)$ false. In the following, we will assume that, since satisfied, $AppliReq(r)$ is continuously satisfied until IR is executed.

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

Remark 1. Let $\mathcal{T}$ be a task, $\mathcal{S}_\mathcal{T}$ a specification of $\mathcal{T}$, and $\mathcal{P}$ a wave protocol with one initiator, r. To prove that $\mathcal{P}$ is snap-stabilizing for $\mathcal{S}_\mathcal{T}$, we must show that any execution of $\mathcal{P}$ satisfies two 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 $\mathcal{S}_\mathcal{T}$.

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

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

```
Input: $Ng_p$: 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: $Cld_p=\{q\in Ng_p::(S_q\neq C)\wedge(P_q=p)\wedge(L_q=L_p+1)\wedge[(S_q\neq S_p)\Rightarrow(S_p\in\{B,P\}\wedge S_q=F)]\}$;
Predicates:
$CF(p)$      $\equiv (\forall q \in Ng_p :: S_q \neq C)$
$Leaf(p)$    $\equiv [\forall q \in Ng_p :: (S_q \neq C) \Rightarrow (P_q \neq p)]$
$BLeaf(p)$   $\equiv (S_p = B) \wedge [\forall q \in Ng_p :: (P_q = p) \Rightarrow (S_q = F)]$
$AnsOk(p)$   $\equiv (Que_p = A) \wedge [\forall q \in Ng_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$
$Bst(p)$     $\equiv (S_p = C) \wedge Leaf(p)$
$Fck(p)$     $\equiv BLeaf(p) \wedge CF(p) \wedge AnsOk(p)$
$PreC(p)$    $\equiv (S_p = F) \wedge [\forall q \in Ng_p :: (P_q = p) \Rightarrow (S_q \in \{F,C\})]$
$Clean(p)$   $\equiv (S_p = P) \wedge Leaf(p)$
$Requi(p)$   $\equiv (S_p\in\{B,F\})\wedge[(S_p=B)\Rightarrow CF(p)]\wedge[[(Que_p=Q)\wedge(\forall q\in Ng_p::(S_q\neq C)\Rightarrow(Que_q\in\{Q,R\}))]$
             $\vee [(Que_p=A) \wedge (\exists q\in Ng_p::(S_q\neq C) \wedge ((Que_q=Q)\vee(q\in Cld_p\wedge Que_q=R)))]]$
$Ans(p)$     $\equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CF(p)] \wedge (Que_p = R)$
             $\wedge (\forall q\in Cld_p::Que_q\in\{W,A\}) \wedge [\forall q\in Ng_p::(S_q\neq C)\Rightarrow(Que_q\neq Q)]$
Actions:
 PIF Part:
  $B$-action :: $(\mathcal{B}.Req_p = W) \wedge \mathcal{B}.End_p \wedge Bst(p)$   $\rightarrow S_p := B$; $Que_p := Q$; $\mathcal{B}.Req_p := I$;
  $F$-action :: $Fck(p)$                                                          $\rightarrow S_p := F$; $\mathcal{B}.Init_p$; $\mathcal{B}.End_p := false$;
  $P$-action :: $PreC(p)$                                                         $\rightarrow S_p := P$;
  $C$-action :: $Clean(p)$                                                        $\rightarrow S_p := C$;
  $T$-action :: $(\mathcal{B}.Req_p = I) \wedge \mathcal{B}.End_p \wedge (S_p = C)$ $\rightarrow \mathcal{B}.Req_p := O$;
 Question Part:
  $QR$-action :: $Requi(p)$                                                       $\rightarrow Que_p := R$;
  $QA$-action :: $Ans(p)$                                                         $\rightarrow Que_p := A$;
```

techniques. In [3,4], 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 previously 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) have 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 false 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, $\forall p \in V$, all the variables assignments required to generate a *normal starting configuration* of $\mathcal{A}$ have 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 $\mathcal{B}.Req_r$) instead of $\mathcal{B}$ itself. $\mathcal{SSBB}(\mathcal{B})$ 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). Some PIF protocols for arbitrary networks have been proposed in the snap-stabilizing literature, e.g., [8,9]. A PIF scheme can be informally described as follows: the initiator, r, starts the protocol by broadcasting a message m (*broadcast phase*), then, $\forall p \in V \setminus \{r\}$, p will send an acknowledgment to r for the receipt of m (*feedback phase*). Using the PIF scheme, the reset protocol can be performed as follows: (i) r 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. Indeed, we want to apply our technique to self-stabilizing protocols working with any daemon, so, we need a reset protocol that works with the most general daemon. Such a protocol is provided in [9].

Snap-Stabilizing PIF. A snap-stabilizing PIF protocol satisfies the following specification: starting from any configuration, when r has a message m to broadcast, it starts the broadcast in a finite time. Then, $\forall p \in V \setminus \{r\}$, p will both receive m and send an acknowledgment (for the receipt of m) which will reach r in a finite time.

Theorem 1 ([9]). *The PIF protocol proposed in [9] 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 [9], called $\mathcal{PIF}$, works with any 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 [9] for details). $\mathcal{PIF}$ is divided in three parts: the PIF, question, and correction parts. 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. Informally, the PIF part maintains in every processor p a variable crucial for $\mathcal{SSBB}$: S_p. Indeed, 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.

Property 1. From [9], 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 costs 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 the technical report for details [10]).

SSBB Protocol. To build $\mathcal{SSBB}(\mathcal{B})$, we use the following composition technique. This composition technique is closed to the *hierarchical composition* of Gouda and Herman [11]. Let P_1 and P_2 be two protocols. The composition of P_1 and P_2, noted $P_2 \circ_{|G} P_1$, is 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.
- Any action $L_i :: H_i \rightarrow S_i$ in P_2 is replaced by $L_i :: G \wedge H_i \rightarrow S_i$ in $P_2 \circ_{|G} P_1$.

Following these rules, $\mathcal{SSBB}(\mathcal{B}) = \mathcal{B} \circ_{|Ok(p)} Reset(\mathcal{B})$ with $Ok(p) \equiv (S_p = C)$. $Reset(\mathcal{B})$ is a slightly modified $\mathcal{PIF}$ (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-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 ($\mathcal{B}.Req_r = W$) and $\mathcal{B}.End_r = true$ (to avoid the aborting a previous initiated wave of $\mathcal{B}$). Also, we modify the *F-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 $Ok(p)$ in the composition so that any processor p aborts its local execution of $\mathcal{B}$ when receiving the reset 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. So, while p participates to a reset, $Ok(p)$ is false and any action of $\mathcal{B}$ in $\mathcal{SSBB}(\mathcal{B})$ is disabled at p. Finally, we add an action, noted *T-action*, so that r switches $\mathcal{B}.Req_r$ from I to O at the termination of each wave of $\mathcal{SSBB}(\mathcal{B})$.

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

```
Input: $Ng_p$: set of (locally) ordered neighbors of $p$;
Variables: $S_p \in \{B,F,P,C,EB,EF\}$; $P_p \in Ng_p$; $L_p \in \mathbb{N}$; $Que_p \in \{Q,R,W,A\}$;
Macros:
  $Cld_p = \{q \in Ng_p :: (S_q \neq C) \wedge (P_q = p) \wedge (L_q = L_p + 1) \wedge [(S_q \neq S_p) \Rightarrow ((S_p \in \{B,P\} \wedge S_q = F) \vee (S_p = EB))]\}$;
  $PPot_p = \{q \in Ng_p :: S_q = B\}$;
  $Pot_p = \{q \in Ng_p :: \forall q' \in PPot_p, L_q \leq L_{q'}\}$;
Predicates:
  $CF(p) \equiv (\forall q \in Ng_p :: S_q \neq C)$
  $Leaf(p) \equiv [\forall q \in Ng_p :: (S_q \neq C) \Rightarrow (P_q \neq p)]$
  $BLeaf(p) \equiv (S_p = B) \wedge [\forall q \in Ng_p :: (P_q = p) \Rightarrow (S_q = F)]$
  $AnsOk(p) \equiv (Que_p = A) \wedge [\forall q \in Ng_p :: (S_q \neq C) \Rightarrow (Que_q = A)]$
  $GoodS(p) \equiv (S_p = C) \vee [(S_{P_p} \neq S_p) \Rightarrow ((S_{P_p} = EB) \vee (S_p = F \wedge S_{P_p} \in \{B,P\}))]$
  $GoodL(p) \equiv (S_p \neq C) \Rightarrow (L_p = L_{P_p} + 1)$
  $AbR(p) \equiv \neg GoodS(p) \vee \neg GoodL(p)$
  $EFAbR(p) \equiv (S_p = EF) \wedge AbR(p) \wedge [\forall q \in Ng_p :: (P_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF,C\})]$
  $EBst(p) \equiv (S_p \in \{B,F,P\}) \wedge [\neg AbR(p) \Rightarrow (S_{P_p} = EB)]$
  $EFck(p) \equiv (S_p = EB) \wedge [\forall q \in Ng_p :: (P_q = p \wedge L_q > L_p) \Rightarrow (S_q \in \{EF,C\})]$
  $Bst(p) \equiv (S_p = C) \wedge (Pot_p \neq \emptyset) \wedge Leaf(p)$
  $Fck(p) \equiv BLeaf(p) \wedge CF(p) \wedge AnsOk(p)$
  $PreC(p) \equiv (S_p = F) \wedge (S_{P_p} = P) \wedge [\forall q \in Ng_p :: (P_q = p) \Rightarrow (S_q \in \{F,C\})]$
  $Clean(p) \equiv (S_p = P) \wedge Leaf(p)$
  $Requi(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CF(p)] \wedge [[(Que_p = Q) \wedge (\forall q \in Ng_p ::$
      $(S_q \neq C) \Rightarrow (Que_q \in \{Q,R\}))] \vee [(Que_p \in \{W,A\}) \wedge (\exists q \in Ng_p :: (S_q \neq C)$
      $\wedge ((Que_q = Q) \vee (q \in Cld_p \wedge Que_q = R)))]]$
  $Wait(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CF(p)] \wedge (Que_p = R) \wedge (Que_{P_p} = R)$
      $\wedge (\forall q \in Cld_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Ng_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$
  $Ans(p) \equiv (S_p \in \{B,F\}) \wedge [(S_p = B) \Rightarrow CF(p)] \wedge (Que_p = W) \wedge (Que_{P_p} = A)$
      $\wedge (\forall q \in Cld_p :: Que_q \in \{W,A\}) \wedge (\forall q \in Ng_p :: (S_q \neq C) \Rightarrow (Que_q \neq Q))$
Actions:
  Correction Part:
    EC-action :: $EFAbR(p) \rightarrow S_p := C$;
    EB-action :: $EBst(p) \rightarrow S_p := EB$;
    EF-action :: $EFck(p) \rightarrow S_p := EF$;
  PIF Part:
    B-action :: $Bst(p) \rightarrow S_p := B$; $P_p := \min_{\prec_p}(Pot_p)$; $L_p := L_{P_p} + 1$; $Que_p := Q$;
    F-action :: $Fck(p) \rightarrow S_p := F$; $\mathcal{B}.Init_p$;
    P-action :: $PreC(p) \rightarrow S_p := P$;
    C-action :: $Clean(p) \rightarrow S_p := C$;
  Question Part:
    QR-action :: $Requi(p) \rightarrow Que_p := R$;
    QW-action :: $Wait(p) \rightarrow Que_p := W$;
    QA-action :: $Ans(p) \rightarrow Que_p := A$;
```

4 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 r 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 Section 3. We now prove that $\mathcal{SSBB}(\mathcal{B})$ is snap-stabilizing for the specification of $\mathcal{T}$ 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 remark about $\mathcal{B}$:

Remark 3. $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables.

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

Definition 3 (Fair Execution [6]). *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.*

From Assumption 3, it is easy to see that the number of steps of each protocol in a wave of the composition is finite, so we can deduce the following theorem.

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

Since Algorithm $\mathcal{A}$ allows r to restart the protocol infinitely often it is clear that $\mathcal{B}$ sets $\mathcal{B}.End_r$ to $false$ in a finite time. So, the system needs a computation of $\mathcal{T}$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in a finite time and the two next lemmas are proved.

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

The next lemma shows that since r requests a computation of $\mathcal{T}$, the system eventually takes this request into account by executing $\mathcal{B}.Req_r := W$.

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

By Lemmas 1 and 2, the following theorem holds. 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 shows that each computation of $\mathcal{T}$ initiated by r is executed as expected. This result is based on the snap-stabilizing reset of $\mathcal{B}$ when r initiates $\mathcal{SSBB}(\mathcal{B})$.

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

4.1 Complexity Analysis

Space Complexity. Let $\mathcal{M}(\mathcal{A})$ be the memory requirement of $\mathcal{A}$. $\mathcal{B}$ differs from $\mathcal{A}$ by just a boolean at r. So, the memory requirement of $\mathcal{B}$ is in the same order than $\mathcal{A}$ and by taking into account 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. In the following, we assume that $\mathcal{A}$ is self-stabilizing under $\mathcal{D}$ such that $\mathcal{D} \in \{WF, UF\}$. So, let $R_1(\mathcal{A})$ be the maximal number of rounds starting from any configuration before r decides in $\mathcal{A}$ and let $R_2(\mathcal{A})$ be the maximal number of rounds that $\mathcal{A}$ requires to perform $\mathcal{T}$ starting from the configuration generated by $\mathcal{B}.Init$.

Theorem 7. *If $\mathcal{A}$ is self-stabilizing under $\mathcal{D}$ 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., $AppliReq(r)$ is satisfied). According to $\mathcal{B}.Req_r$, three cases are possible:

- $\mathcal{B}.Req_r = O$ in γ_i. In such a configuration, IR is enabled (the guard of IR is $AppliReq(r) \wedge (B.Req_r = O)$). Also, no action of $SSBB(\mathcal{B})$ can modify $\mathcal{B}.Req_r$ until $\mathcal{B}.Req_r$ is set to W by IR (see Algorithms 1 and 2). So, IR is continuously enabled at r and, r executes IR, i.e., $\mathcal{B}.Req_r := W$, in at most one round. Then, $\mathcal{B}.Req_r$ is continuously equal to W until r initiates $\mathcal{SSBB}(\mathcal{B})$ by *B-action* (see Algorithms 1 and 2). Also, as $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 3), in the worst case, the system reaches a configuration γ_j from which $Bst(r)$ is continuously satisfied and no action of $Reset(\mathcal{B})$ different of *B-action* at r is enabled until r executes *B-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-action* becomes continuously enabled at r from γ_k and r executes *B-action* in the next round. Hence, starting from any configuration where $\mathcal{B}.Req_r = O$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in at most $9N + R_1(\mathcal{A})$ rounds.
- $\mathcal{B}.Req_r = I$ in γ_i. As $\mathcal{B}$ does not write into the $Reset(\mathcal{B})$ variables (Assumption 3), in the worst case, the system reaches a configuration γ_j from which $Bst(r)$ is continuously satisfied and no action of $Reset(\mathcal{B})$ different of *B-action* at r is enabled until r executes *B-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-action* is disabled until $\mathcal{B}.Req_r = W$, two rounds are necessary so that

$\mathcal{B}.Req_r$ switches from I to O by *T-action* ($Bst(r) \Rightarrow (S_r = C)$) and from O to W by Action IR. Then, *B-action* will be continuously enabled at r and r will execute it in the next round. Hence, starting from any configuration where $\mathcal{B}.Req_r = I$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in at most $9N + R_1(\mathcal{A}) + 1$ rounds.
- $\mathcal{B}.Req_r = W$ in γ_i. In this case, the system has to perform a complete $SSBB(\mathcal{B})$ wave before r satisfies $\mathcal{B}.Req_r = O$. 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 3. 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), $\mathcal{B}.Req_r = I$ 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*: $\mathcal{B}.Req_r := O$ (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), $\mathcal{B}.Req_r = O$, 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 $\mathcal{B}.Req_r = W$, $\mathcal{SSBB}(\mathcal{B})$ is initiated in at most $15N + R_1(\mathcal{A}) + R_2(\mathcal{A})$ rounds.

□

Corollary 1. *If $\mathcal{A}$ is self-stabilizing under $\mathcal{D}$ 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 $\mathcal{D} = UF$. We have proved that, if $\mathcal{A}$ is self-stabilizing under $\mathcal{D} = UF$, then $SSBB(\mathcal{B})$ is snap-stabilizing under $\mathcal{D} = UF$. 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 $\mathcal{D} = UF$, 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 [10] except for a constant factor due to the *T-action*). 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 of the delay to start a requested $SSBB(\mathcal{B})$ wave is the product of these two complexities and the theorem holds. □

Corollary 2. *If $\mathcal{A}$ is self-stabilizing under $\mathcal{D} = UF$, 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 propose to snap-stabilize the self-stabilizing depth-first token circulation ($DFTC$) protocol of Huang and Chen [5] using our transformer. In the following, the protocol of Huang and Chen will be denoted by $\mathcal{DFS}$.

Protocol $\mathcal{DFS}$. 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. From [5], follows:

Theorem 9 ([5]). *$\mathcal{DFS}$ is a self-stabilizing $DFTC$ protocol assuming a weakly fair daemon.*

Informally, $\mathcal{DFS}$ is divided in two parts. The first part manages the token circulation strictly speaking. The other part handles abnormal behaviors due to the initial configuration. We first focus on the token circulation part. This part maintains two variables: D and C. D is a *descendant* pointer 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 pointers, 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 current 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 pointers 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 pointer D to $NULL$. The dropped processor then recovers itself 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, 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 $\mathcal{DFS}'$ such that $\mathcal{SSBB}(\mathcal{DFS}')$ 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. We add the condition $\neg End_r$ at the guard of the initialization of the token circulation action at r.
4. Finally, 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 $\mathcal{DFS}'$ and, by Theorem 5, the following theorem holds:

Theorem 10. *$\mathcal{SSBB}(\mathcal{DFS}')$ 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 daemon, 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 $\mathcal{DFS}'$, these properties insures that:

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

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

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

By Theorem 7, we know that, starting from any configuration, a requested wave of $\mathcal{SSBB}(\mathcal{DFS}')$ 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 $DFTC$ starting from a configuration γ_i where $\forall p \in V$, $D_p = NULL \wedge C_p = 1$. In the same way, by Corollary 1, starting from any configuration, a complete requested wave of $\mathcal{SSBB}(\mathcal{DFS}')$ 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 $DFTC$ 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 DFTC is initiated (resp. performed) using $\mathcal{SSBB}(\mathcal{DFS}')$ 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 [12,13].

6 Conclusion

We propose a semi-automatic method to snap-stabilize self-stabilizing wave protocols for arbitrary networks with one initiator and such that their decision actions are at the initiator only. The snap-stabilizing solution we obtain with our technique works at least with the same daemon than the self-stabilizing protocol to snap-stabilizing. But, in some case like the $DFTC$ protocol of [5], we obtain a solution working with a weaker scheduling assumption. Also, the solution we obtain could be better in time complexities than the self-stabilizing protocol we want to transform. For instance, despite the $DFTC$ protocol of [5] stabilizes in $\Omega(D \times N)$ rounds, its snap-stabilizing version executes a requested $DFTC$ (as expected) in $O(N)$ rounds.

References

1. Dijkstra, E.: Self stabilizing systems in spite of distributed control. Communications of the Association of the Computing Machinery **17** (1974) 643–644
2. Bui, A., Datta, A., Petit, F., Villain, V.: State-optimal snap-stabilizing PIF in tree networks. In: Proceedings of the Fourth Workshop on Self-Stabilizing Systems, Austin, Texas, USA, IEEE Computer Society Press (1999) 78–85
3. Katz, S., Perry, K.: Self-stabilizing extensions for message-passing systems. Distributed Computing **7** (1993) 17–26
4. Cournier, A., Datta, A., Petit, F., Villain, V.: Enabling snap-stabilization. In: 23th International Conference on Distributed Computing Systems (ICDCS 2003), Providence, Rhode Island USA, IEEE Computer Society Press (2003) 12–19
5. Huang, S., Chen, N.: Self-stabilizing depth-first token circulation on networks. Distributed Computing **7** (1993) 61–66
6. Tel, G.: Introduction to distributed algorithms. Cambridge University Press, Cambridge, UK (Second edition 2001)
7. Dolev, S., Israeli, A., Moran, S.: Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems **8**(4) (1997) 424–440
8. Blin, L., Cournier, A., Villain, V.: An improved snap-stabilizing PIF algorithm. In: DSN SSS'03 Workshop: Sixth Symposium on Self-Stabilizing Systems (SSS'03), LNCS 2704 (2003) 199–214
9. Cournier, A., Devismes, S., Villain, V.: Snap-stabilizing PIF and useless computations. In: The Twelfth International Conference on Parallel and Distributed Systems (ICPADS'06). Volume 1., Minneapolis, USA, IEEE Computer Society Press P2612 (2006) 39–46
10. Cournier, A., Devismes, S., Villain, V.: 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.
11. Gouda, M.G., Herman, T.: Adaptive programming. IEEE Trans. Softw. Eng. **17**(9) (1991) 911–921
12. Johnen, C., Beauquier, J.: Space-efficient distributed self-stabilizing depth-first token circulation. In: Proceedings of the Second Workshop on Self-Stabilizing Systems, Las Vegas (UNLV), USA, Chicago Journal of Theoretical Computer Science (1995) 4.1–4.15
13. Datta, A., Johnen, C., Petit, F., Villain, V.: Self-stabilizing depth-first token circulation in arbitrary rooted networks. In: SIROCCO'98, The 5th International Colloquium On Structural Information and Communication Complexity Proceedings, Carleton University Press (1998) 229–243