RAPPORT DE RECHERCHE

L R I

OBSERVING LOCALLY SELF-STABILIZATION

BEAUQUIER J / PILARD L / ROZOY B

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

05/2004

Rapport de Recherche N° 1387

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

Observing locally self-stabilization

Joffroy Beauquier beauquier@lri.fr
Laurence Pilard pilard@lri.fr
Brigitte Rozoy rozoy@lri.fr

Université Paris-Sud
Laboratoire de Recherche en Informatique
91405 Orsay Cedex, France

1 Abstract

A self-stabilizing algorithm cannot detect by itself that stabilization has been reached. For overcoming this drawback Lin and Simon introduced the notion of an external observer, i.e. a set of processes, one being located at each node, whose role is to detect stabilization.

We propose here a less expensive approach, where there is a single observing process located at a unique node. This process is not allowed to detect false stabilization and it must eventually detect that stabilization is reached. Moreover it must not interfere with the observed self-stabilizing algorithm. Our result is that there exists such an observer for any problem on a distinguished network having a synchronous self-stabilizing solution. Note that our proof is constructive.

2 Introduction

The notion of self-stabilization was introduced by Dijkstra [Dij74]. He defined an algorithm as self-stabilizing when "regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps". Such a property is very desirable for any distributed algorithm, because after any unexpected perturbation modifying the memory state, the algorithm eventually recovers and returns to a legitimate state, without any outside intervention.

Dijkstra's notion of self-stabilization, which originally had a very narrow scope of application, is proving to encompass a formal and unified approach to fault-tolerance under a model of transient failures for distributed algorithms (cf. books like [Dol00] or [Tel94] for overviews).

Traditionally, self-stabilization is supposed to suffer two main drawbacks. The first is that a self-stabilizing algorithm only eventually recovers, involving that during some time the behavior is not correct. The second is that a process can never know whether or not the algorithm is stabilized.

There is less that you can do against the first drawback because it is inherently bounded to the very definition of self-stabilization. You simply have to pay for a much smaller overhead than for the robust algorithm approach (consensus), and the price is the momentary loss of the safety.

Surprisingly enough, there is no study dealing with the second issue, apart the important paper by Lin and Simon [LS92]. If it is obvious that no detection of stabilization from the inside is possible (any local variable used for that could be corrupted). Nevertheless it is perfectly feasible to detect stabilization from the outside (for instance and although

it is just a theoretical remark, when a bound on the number of steps before stabilization is known simply by counting). "From the outside" can be replaced by "from the inside but using stable memory" (memory not subject to failures). By restricting their attention to ring networks, Lin and Simon propose "a new model, in which it is meaningful to say that a processor knows that the ring is stable". This model introduces the notion of a distributed observer, located at each node of the network and that is responsible for detecting stabilization and does not influence the self-stabilizing protocol. Lin and Simon did not address the implementation issue of their observer, but it is straightforward to see that implementing the observer involves, at each node, the presence of a stable memory. If such an assumption can be valid on small local area networks, in which strong security and reliability can be ensured, it is unrealistic for world wide, heterogeneous networks, in which implementing a stable memory on each machine would be, if ever possible, extremely costly.

In this paper we tried to solve the observation problem using much less safe memory. In particular, we raised the question to know whether or not stabilization detection is feasible, if only one node disposes of some stable memory.

Think of a network in which a self-stabilizing algorithm is executed. In a particular location, there is a special process, the observer, that, as its name indicates it, observes the behavior of the local module of the distributed algorithm. The observer disposes as a parameter of sequences of actions (or a machine generating such sequences) and simply tries to match one of these predetermined sequences to what the local process is executing. As long as no matching appears the observer does not say anything, but once there is one, it immediately announces "stabilization detected".

Obviously, the announcement obeys some conditions. For instance the observer is not allowed to announce false stabilization. At the opposite, if the algorithm is stabilized the observer must eventually announce. Moreover, the observer cannot be used by the algorithm for stabilizing : the behavior of the algorithm must be the same with or without observer.

The observer has the following features :

1. The observer is *located at one processor of the network.* If the network is anonymous, the location of the observer is arbitrary.

2. The observer is not allowed to detect stability with any information *depending on the network* (for instance the size).

3. *The observer cannot influence the algorithm*, which means that the execution of the algorithm is the same with or without the observer.

4. The observer is *not subject to any type of corruption.*

5. The observer observes the behavior of the local process (sequential sequence of instructions) and tries to *match part of this behavior with some predefined sequences.*

6. The announcement of the stabilization obeys some *safety* and *liveness* conditions.

We will prove the existence of such an observer, not only for particular network topologies like rings or trees, or for particular (local) problems, but for any problem on any distinguished network topology, provided that, obviously, this problem has a synchronous self-stabilizing solution.

Our main result can be stated as : if there exists a synchronous self-stabilizing distributed solution for some problem in some distinguished network, then there exists a

synchronous self-stabilizing distributed solution (not necessarily the same) for the same problem in the same network, that can be observed by an observer like above located at a particular node. Moreover the observable self-stabilizing solution can be effectively built from the simply self-stabilizing solution in a canonical way (and then be possibly built in a compiler).

As a matter of fact we offer a little bit more.

1. If the self-stabilizing algorithm is generic for a whole family of networks then the observer is the same for all networks in the family.

2. Two different observers for two different problems only differ by their parameter (the sequences) but their basic mechanism is the same (the matching)

Point 1 is useful for not changing the observer when network topology dynamically evolves (maintaining the same basic structure). Point 2 makes simpler the implementation, since only the parameters have to be changed for detecting stabilization of a new algorithm.

A last issue must be discussed : where to locate the observer? In the case where the network has a distinguished node (we will here assume this property), it is natural to locate it at this node.

The plan of the paper is the following. First, we introduce the formal definition of an observer. Then, we prove that any problem on a distinguished network that can be solved by a synchronous self-stabilizing solution, can also be solved by a synchronous *observable* self-stabilizing solution (not necessarily the same).

3 Model of the observer

We use a classical model for distributed algorithms.

Definition 1 (Distributed algorithm) *A distributed algorithm $\mathcal{A} = (C, A)$ is an automaton, where C is the set of all states (called* configurations*) of $\mathcal{A}$ and A is the set of all transitions (called* actions*) of $\mathcal{A}$.*

Definition 2 (Execution) *Let $\mathcal{A}$ be a distributed algorithm. An execution of $\mathcal{A}$, noted $E = c_1 a_1 c_2 a_2 \ldots$ is a maximal sequences of configurations and actions of $\mathcal{A}$ such as c_{i+1} is reached from c_i by the execution of the action a_i.*

The sequence is maximal if it is infinite, or finite but no action of $\mathcal{A}$ is possible in the last configuration.

Definition 3 (Self-stabilizing algorithm) *An algorithm $\mathcal{A}$ is self-stabilizing for a specification $\mathcal{P}$ iff it exists a sub-set $\mathcal{C}_{\mathcal{L}}$ of the set of the configurations of $\mathcal{A}$ such as : (i) every execution of $\mathcal{A}$ with an initial configuration in $\mathcal{C}_{\mathcal{L}}$ verifies $\mathcal{P}$ and $\mathcal{C}_{\mathcal{L}}$ is closed (ii) every execution of $\mathcal{A}$ contains at least a configuration in $\mathcal{C}_{\mathcal{L}}$.*

We call $\mathcal{C}_{\mathcal{L}}$, the set of legitimate configurations of $\mathcal{A}$.

Validation of the model It is difficult to describe a distributed algorithm by a global automaton. Usually, a processor is given by a set of guarded rules. Each rule has two parts : the guard part (condition) and the move part.

$\langle$ Algorithm $\rangle \equiv \langle$ Rule 1 $\rangle \mid \langle$ Rule 2 $\rangle \mid \ldots \mid \langle$ Rule n $\rangle$

$\langle$ Rule $\rangle \equiv \langle$ Guard $\rangle \rightarrow \langle$ Move $\rangle$

The guard part is defined as a boolean function of the processor's variables and of the variables of its neighbours. When a guard of a rule on a processor is true, this processor may take the corresponding move which changes the processor state into a new one. The relation between guarded rules and automaton is given by :

1. The variables of a processor and the program counter determine the states of this processor : each state represents a possible value of the program counter and the variables of the processor. We note $State(P)$, the set of all possible states of a processor P.

2. The code of the algorithm determines the transition function of the processor : each rule of the algorithm represents a transition of the processor. We note $Trans(P)$, the set of all transitions of a processor P.

We assume that rules are locally atomic and we note that a processor is a sequential machine, so it can only execute one rule at a time.

A distributed algorithm is composed by a set of processors. Let $\{P_1, \ldots, P_n\}$ be a set of processors. The relation between the processors $\{P_1, \ldots, P_n\}$ and the automaton of the distributed algorithm $\mathcal{A}$, composed by $\{P_1, \ldots, P_n\}$ is given by :

1. A configuration of $\mathcal{A}$ is a vector of states of all the processors of the algorithm, formally, if c is a configuration of $\mathcal{A}$, then $c \in State(P_1) \times \ldots \times State(P_n)$

2. An action $c \xrightarrow{a} c'$ of $\mathcal{A}$ is such as c and c' are configurations of the algorithm and a is a composition of some rules of processors of the algorithm such as : if $a = \{r_1, \ldots, r_n\}$, then $r_i \in Trans(P) \Rightarrow$ the state of P and the state of all its neighbours in c verify the guard of r_i and only the state of P is changed in c' according to the move of r_i.

We assume that algorithms are synchronous, which means that an action of an algorithm is the composition of at most one rule for each processor.

The relation between the automaton of a distributed algorithm and the processors of the same distributed algorithm allow us to talk about projection of an execution of a distributed algorithm over a processor :

Definition 4 (Projections of an execution over a processor)
Let $\mathcal{A}$ be a distributed algorithm and $E = c_1 a_1 c_2 a_2 \ldots$ an execution of $\mathcal{A}$. A projection of E over a processor p is :
$\Pi_p E = \Pi_p c_1 \ \Pi_p c_2 \ \Pi_p c_3 \ \ldots$ *with* $\forall i \geq 1, \Pi_p c_i \in State(p)$

Definition 5 (Projections of an execution over a set of variables of a processor)
Let $\mathcal{A}$ be a distributed algorithm and $E = c_1 a_1 c_2 a_2 \ldots$ an execution of $\mathcal{A}$. A projection of E over a processor p and a set of variables $v = \{v_1, \ldots, v_k\}$ such as $\forall i \in [1, k], v_i$ is a variable of p is : $\Pi_{pv} E = \Pi_{pv} c_1 \ \Pi_{pv} c_2 \ \Pi_{pv} c_3 \ \ldots$ with $\forall i \geq 1, \Pi_{pv} c_i$ is the projection of $\Pi_p c_i$ over the variables in v.

We adopt the general terminology of language theory, cf. for instance [Har78]. In particular, we will use the notion of *alphabet* (states), *word* (possibly infinite sequence of states), *factor* (finite sequence of states that appears in a word) and *sub-word* (finite sequence of factors that appears in a word and in the same order). For instance, let us consider the alphabet $\Sigma = \{a, b, c\}$. $w = ababca$ is a word on Σ, (ba) is a factor of w and $(ab\ ; ca)$ is a sub-word of w, but $(ca\ ; ab)$ is not a sub-word of w. Moreover, we introduce the letter '_' that represents any letter in Σ. Let u be a finite word over $\Sigma \cup \{\_\}$. We say

that u is a factor of a word w over the alphabet Σ iff the set of factors of w represented by u is not empty. For instance, we say that $(b\_)$ is a factors of w and by extension, we say that $(\_b\,;ca)$ is a sub-word of w, but (ca ;_b) is not. Note that a factor is also a sub-word.

We now define an *observer* and an *observable self-stabilizing algorithm.*

Definition 6 (observer)
An observer $\mathcal{O}(\Sigma, \mathcal{L})$ is a boolean function $\mathcal{O}(\Sigma, \mathcal{L}) : \Sigma^* \rightarrow \{true,false\}$ such as :

1. $\mathcal{O}(\Sigma, \mathcal{L})(\varepsilon) = false$
2. $\forall w \in \Sigma^*\ ,\ \mathcal{O}(\Sigma, \mathcal{L})(w) = true \Leftrightarrow \exists v \in \mathcal{L} \mid v$ *is a sub-word of* w

Definition 7 (observer for a self-stabilizing algorithm at a processor)
Let $\mathcal{A}$ be a self-stabilizing algorithm and let P be a processor in $\mathcal{A}$. We say that the observer $\mathcal{O}(\Sigma, \mathcal{L})$ is an observer for $\mathcal{A}$ at P iff

1. *there exists a set of variables $V = \{v_1, \ldots, v_k\}$ such as the domain of v_i is D_i, v_i is a variable of P and $D_1 \times \ldots \times D_k = \Sigma$.*
2. *for any left factor v of an execution w of $\mathcal{A}$, $\mathcal{O}(\Sigma, \mathcal{L})(\Pi_{PV} v) = true \Rightarrow v$ reaches a legitimate configuration of $\mathcal{A}$ (safety).*
3. *for any left factor v of an execution w of $\mathcal{A}$, v reaches a legitimate configuration of $\mathcal{A} \Rightarrow \exists\ u$ a left factor of w such as v is a left factor of u and $\mathcal{O}(\Sigma, \mathcal{L})(\Pi_{PV} u) = true$ (liveness).*

In this case, $\mathcal{O}(\Sigma, \mathcal{L})$ is noted $\mathcal{O}(V, \mathcal{L})$.

Remark 1 For an execution w of $\mathcal{A}$, a boolean value is associated by the observer to each left factor of w. Initially, for the empty sequence, the observer returns *false*. The first time it returns *true*, we will say that the observer "announces" the stabilization. Note that the condition in the definition above involves that an observer never announces false stabilization (safety) and eventually announces stabilization (liveness).

Definition 8 (observable self-stabilizing algorithm)
Let $\mathcal{A}$ be a self-stabilizing algorithm. We say that $\mathcal{A}$ is observable iff exists an observer $\mathcal{O}(V, \mathcal{L})$ such as :
if $\mathcal{A}$ is anonymous, then $\forall P$ in $\mathcal{A}$, $\mathcal{O}(V, \mathcal{L})$ is an observer for $\mathcal{A}$ at P and
if $\mathcal{A}$ is distinguished, then for P the leader of $\mathcal{A}$, $\mathcal{O}(V, \mathcal{L})$ is an observer for $\mathcal{A}$ at P .

Remark 2 (Non interference) In our model, the observer cannot influence the algorithm, which means that the execution of the algorithm is the same with or without the observer.

4 Result

In this section, we use the fair composition of self-stabilizing algorithms, described for instance in [Dol00].

Theorem 1 *All problem on a distinguished communication graph that can be solved by a synchronous self-stabilizing solution, can also be solved by a synchronous* observable *self-stabilizing solution (not necessarily the same).*

First, we explain the idea of the proof which is constructive.

Let $\mathcal{P}$ be a problem that can be solved by a self-stabilizing solution on a distinguished network. Let $\mathcal{A}$ be a self-stabilizing algorithm solving $\mathcal{P}$ and G a distinguished communication graph. We construct an observer for $\mathcal{A}$ in five steps :

1. We give a synchronous algorithm $\mathcal{B}$, that constructs a spanning tree over G and we prove that $\mathcal{B}$ is self-stabilizing.

2. We give a synchronous algorithm $\mathcal{C}$, that computes the size of a tree and we prove that $\mathcal{C}$ is self-stabilizing.

3. We consider the result $\mathcal{R}$ of the fair composition of $\mathcal{B}$ and $\mathcal{C}$. $\mathcal{R} = \mathcal{B} \circ \mathcal{C}$.
   $\mathcal{R}$ constructs a spanning tree over G and computes the size of the constructed tree.
   $\mathcal{B}$ and $\mathcal{C}$ are synchronous and self-stabilizing, thus $\mathcal{R}$ is synchronous and self-stabilizing.
   We prove that $\mathcal{R}$ can be observed (by constructing an observer).

4. Every round, we construct a global snapshot of the execution of an algorithm at the root of the tree. We give a synchronous algorithm $\mathcal{S}$ that computes this task and we prove that $\mathcal{S}$ is self-stabilizing. Moreover, we prove that if n is the size of the tree, then $\mathcal{S}$ converges in n rounds.

5. We considere the result $\mathcal{F}$ of the fair composition of $\mathcal{R}$ and $\mathcal{S}$. $\mathcal{F} = \mathcal{R} \circ \mathcal{S}$.
   $\mathcal{R}$ and $\mathcal{S}$ are synchronous and self-stabilizing, thus $\mathcal{F}$ is synchronous and self-stabilizing. We prove that $\mathcal{F}$ can be observe.
   $\mathcal{R}$ can be observed. When $\mathcal{R}$ is stabilized, the variable $size$ of the distinguished processor, called $size_{root}$, contains the size of the network. Let n be this size.
   $\mathcal{S}$ converges in $2n-1$ rounds, then $2 * size_{root} - 1$ rounds after the stabilization of $\mathcal{R}$, $\mathcal{S}$ is stabilized. Thus $\mathcal{F}$ can be observed.

   Note : n is not known at the beginning of the algorithm, but is computed in the variable $size_{root}$ during an execution.

6. We considere the result $\mathcal{H}$ of the fair composition of $\mathcal{A}$ and $\mathcal{F}$. $\mathcal{H} = \mathcal{A} \circ \mathcal{F}$.
   $\mathcal{F}$ and $\mathcal{A}$ are synchronous and self-stabilizing, thus $\mathcal{H}$ is synchronous and self-stabilizing. We prove that $\mathcal{H}$ can be observed.
   A self-stabilizing algorithm is related to the set of all its legitimate configurations. $\mathcal{F}$ can be observed. So once $\mathcal{F}$ is stabilized, when we observe a legitimate configuration of $\mathcal{A}$ in the distinguished processor, the observer announces the algorithm has being stable. We obtain an observer of $\mathcal{H}$ and especially an observer of $\mathcal{A}$.

Notations We present several self-stabilizing algorithms over a distinguished communication graph $G(V, E)$. We always use the same model : each node $v_i \in V$ represents the processor P_i, and each edge $(v_i, v_j) \in E$ indicates that P_i and P_j are neighbours ; i.e. they can communicate with each other. We use the shared memory model. A processor P_i communicates with its neighbours P_j by reading variables of P_j and by writing its own variables. Moreover, the system consists of n processors $P_1, P_2, \ldots, P_n$, where $P_2, \ldots, P_n$ run similar programs while P_1 is a special processor that possibly runs a different program, P_1 is called the *root* processor of the graph. Finally, we assume that rules are locally atomic.

We now develop this proof step by step.

4.1 Step 1 : spanning tree

We present the algorithm $\mathcal{B}$, which builds a spanning tree over a distinguished communication graph. Then, we prove that $\mathcal{B}$ is self-stabilizing. This algorithm is described in [Dol00]. In order to determine precisely what the observer will observe, we describe here this algorithm.

Presentation We present a self-stabilizing algorithm $\mathcal{B}$ for marking a breadth-first search (BFS) spanning tree over a distinguished communication graph $G(V, E)$. The program has an input parameter δ which is the number of adjacent links of the processor, each processor P_i numbers its links between 1 and δ_i.

Essentially the algorithm is a distributed BFS algorithm. Each processor is continuously trying to compute its distance from the root and to report this distance to all its neighbours by writing it in its variable. At the beginning of an arbitrary execution, the only processor guaranteed to compute the right distance is the root itself. Once this distance is written in the root's variable, the value stored in this variable will never change. Once processors at distance x from the root have completed computing their distance from the root correctly, and have written their variables, these variables remain constant throughout the execution, and processors at distance $x + 1$ from the root are ready to compute their own distance from the root, and so forth.

The output tree is encoded by means of the variables as follow : each variable $dist_i$ and $parent_i$ in which P_i writes and from which all neighbours of P_i reads, contains respectively the distance from the root to P_i and the number of the link of P_i that reaches the parent of P_i in the BFS tree ($parent_i \in [1 \dots \delta_i]$). There is no *parent* variable at the root.

The code of the algorithm $\mathcal{B}$, for the root and for the other processors is :

```
1  Root (P_1) :          do forever
2                          true → dist_1 := 0
3                        od
4  Other (P_i , i ≠ 1) : do forever
5                          true → dist_i := 1 + min{ dist_k | k ∈ [1 ... δ_i] }
6                                  parent_i := min{ k | k ∈ [1 ... δ_i] ∧ dist_k = dist_i − 1 }
7                        od
```

The state of a processor consists of the value of the program counter and the value of the internal variables : $dist_i$ and $parent_i$. A configuration of the system is a vector of the processor states.

The set $S_{\mathcal{B}}$ of legitimate sequences is defined as the set of all configuration sequences in which every configuration encodes a BFS tree of the communication graph. In fact, a particular BFS tree called the *first* BFS *tree* is encoded. Let $\alpha = (\alpha_1, \alpha_2, \dots, \alpha_n)$ be the arbitrary ordering of the edges incident to each node $v_i \in V$. The *first* BFS *tree* of a communication graph G is uniquely defined by the choice of the root v_1 and α. When a node v_i of distance $x + 1$ from v_1 has more than a single neighbour at distance x from v_1, v_i is connected to its first neighbour according to α_i, whose distance from v_1 is x. In the lemma below, we use the definition of the first BFS tree to characterize the set of legitimate configurations for the algorithm.

The algorithm $\mathcal{B}$ is self-stabilizing

This proof is presented in [Dol00]. We only give the following results. In this paper, we change the read/write atomicity by the rule locally atomicity, thus the lemma 1 is little different than in [Dol00].

The lemma below shows that, in every execution, a legitimate configuration is reached. We use the following definitions of *floating distances* and *smallest floating distance* in our proof.

Definition 9 *A floating distance in some configuration c is a value in a variable $dist_i$ that is smaller than the distance of P_i from the root. The smallest floating distance in some configuration c is the smallest value among floating distance.*

Lemma 1 *For every $k > 0$ and for every configuration that follows k rounds, it holds that*

1. *if there exists a floating distance, then the value of the smallest floating distance is at least k (i.e. $\geq k$)*
2. *the value of the variable dist of every processor that is within distance k from the root (i.e. $< k$) is equal to its distance from the root.*

Corollary 1 *The algorithm $\mathcal{B}$ presented above is self-stabilizing for $S_{\mathcal{B}}$.*

Let us consider the two following corollary. Assumes that the depth of the root is 1 and notes all BFS spanning trees over a same communication graph G, have the same depth.

Corollary 2 *Let G be a communication graph and p the depth of BFS spanning tree over G. Once p rounds, the algorithm $\mathcal{B}$ is stabilized for $\mathcal{S}_{\mathcal{B}}$.*

PROOF :
Let G be a communication graph. All BFS spanning trees over G have the same depth, let p be this depth. The longest distance between a processor and the root in a BFS spanning tree over G is $p - 1$.

According to lemma 1.2, for every $k > 0$ and for every configuration that follows k rounds, it holds that the value of the variable *dist* of every processor that is within distance k from the root (i.e. $< k$) is equal to its distance from the root.

Thus, the configuration c_p reached following the round p is such as the value of the variable *dist* of every processor that is within distance p from the root (i.e. $< p$) is equal to its distance from the root.

Every processor in a BFS spanning tree over G is within distance p from the root of this BFS spanning tree, so the configuration c_p is such as the value of the variable *dist* of every processor is equal to its distance from the root. □

Corollary 3 *Let G be a communication graph and n the size of G. Once n rounds, the algorithm $\mathcal{B}$ is stabilized for $\mathcal{S}_{\mathcal{B}}$.*

PROOF : If n is the size of G, then the depthest of G is n. □

4.2 Step 2 : size of a tree

We present the algorithm $\mathcal{C}$, which computes the size of a tree. Then, we will prove that $\mathcal{C}$ is self-stabilizing. This algorithm is described in [Dol00]. In order to determine precisely what the observer will observe, we describe here this algorithm.

Presentation We present a self-stabilizing algorithm $\mathcal{C}$ for computing the size of a tree $T(V, E)$. We associate each node $v_i \in V$ to a set $Child(v_i)$ which is the set of numbers of the links of v_i that reaches a child processor of v_i in T.

Each processor has a variable *size* that contains the size of *its sub-tree* (which means the sub-tree in which it is the root) and each processor is continuously trying to compute the size of its sub-tree, and to report this size to its father in the tree by writing it in its variable. At the beginning of an arbitrary execution, the only processors guaranteed to compute the right size of their sub-tree are the leafs of the tree. Once this size is written in the leafs's variables, the value stored in these variables will never change. Once all processors at distance x from a leaf have completed computing the size of there sub-tree correctly and have written it in their variables, their variables remain constant throughout execution, and processors at distance $x + 1$ from a leaf are ready to compute their own size of their sub-tree, and so forth.

The code of the algorithm $\mathcal{C}$ is :

```
1   (P_i) : do forever
2               true → size_i := 1 + Σ { size_j | j ∈ Child(P_i) }
3           od
```

The state of a processor consists of the value of the program counter and the value of its variable *size*. A configuration of the system is a vector of the processors states.

The set $S_{\mathcal{C}}$ of legitimate sequences is defined as the set of all configurations sequences in which each variable *size* in processors of every configurations contains the size of the sub-tree of this processor.

The algorithm $\mathcal{C}$ is self-stabilizing

We assume that the depth of the root is 1.

Lemma 2 *Let T be a tree and p be the depth of T. Once p rounds, all variables size of processors contain the size of the sub-tree of these processors.*

PROOF : Note that every round, each processor reads the variable *size* of all its children and writes to its variable. We prove the lemma by induction over p.

Base case : (proof for $p = 1$) If the depth of T is 1, then T contains only one processor : the root. As the root has no child and according to the line 2 of the algorithm, the configuration reached following the first round is such as the variable *size* of the root is 1. □

Induction step : (assumes correctness for $p \geq 1$ and proves for $p + 1$)
Let T be a tree such as its depth is $p + 1 \geq 2$ and r the root of T. r has at least one child. Let $\{r_1, \ldots, r_k \mid k \geq 1\}$ be the children of r and $\{T_1, \ldots, T_k \mid k \geq 1\}$ the sub-trees of T such as $\forall i \in [1, k], r_i$ is the root of T_i.

According to the line 2 of the algorithm, each round, the root writes $1 + \Sigma\{size_j \mid j \in Child(root)\}$ to its variable *size*. Thus, during the round $p+1$, the root writes $1 + \Sigma\{size_j \mid j \in \{r_1, \ldots, r_k\}\}$ to its variable *size*.
$\forall i \in [1,k]$, the depth of T_i is at most p. Moreover, according to the induction assumption, once p rounds, all variables *size* of processors in $\{T_1, \ldots, T_k \mid k \geq 1\}$ contains the size of their sub-tree. Thus, during the round $p+1$, the root writes the size of its sub-tree in its variable *size*. Therefore, once $p+1$ rounds, all variables *size* of processors in T contains the size of the sub-tree of these ones. □

Corollary 4 *The algorithm $\mathcal{C}$ is self-stabilizing for $S_{\mathcal{C}}$.*

Corollary 5 *Let T be a tree and n the size of this tree. Once n rounds, the algorithm $\mathcal{C}$ is stabilized for $S_{\mathcal{C}}$.*

PROOF : If n is the size of T and p the depth of T, then $p \leq n$. □

4.3 Step 3 : spanning tree and size of this tree

We compose the algorithms $\mathcal{B}$ and $\mathcal{C}$. Let $\mathcal{R}$ be the resulting algorithm of this composition. $\mathcal{B}$ and $\mathcal{C}$ are self-stabilizing (by Corollaries 1 and 4), thus $\mathcal{R}$ is self-stabilizing.
We prove that $\mathcal{R}$ can be observed.

Lemma 3 *Let G be a distinguished communication graph and p the depth of a BFS spanning tree over G. $\forall d \in [0, p-1]$, if n_d is the number of processors at distance less or equal than d from the root of G, then once the round $2(d+1)$, the value of the variable size of the root is greater or equal than n_d.*

PROOF : We prove this lemma by induction over d.

Base case : ($d = 0$) We must prove once the round 2, the value of the variable *size* of the root is greater or equal than 1. According to the line 2 of the algorithm $\mathcal{C}$, once the round 1, the variable *size* of the root is greater or equal than 1.□

Induction step : (Assumes correctness for $d \geq 0$ and proves for $d+1$.)
Let G be a distinguished communication graph and p the depth of a BFS spanning tree over G. Let P be a processor at distance $d \in [0, p-1]$ from the root of G.

According to lemma 1, once the round $d+1$, P is connected to the legitimate tree. Thus, $\forall d \in [0, p-1]$, once $d+1$ rounds, the algorithm $\mathcal{B}$ has at least construct a partially BFS spanning tree over G, let T be this tree. T is such as the set of processors in T is equal to the set of processors in G at distance less or equal than d from the root of G. Thus the depth of T is $d+1$.

According to lemma 2, $\forall d \in [0, p-1]$, once the round $2(d+1)$, the variable size of the root at least contains the size of T, which is the number of processors at distance less or equal than d from the root of G. □

Theorem 2 *We define $\mathcal{L}_{\mathcal{R}}$ as $\mathcal{L}_{\mathcal{R}} = \{ (v^{2(v+1)}) \mid v \in \mathbb{N}^+ \}$ and $V_R = \{size\}$.*
$\mathcal{O}_{\mathcal{R}}(V_R, \mathcal{L}_{\mathcal{R}})$ is an observer of $\mathcal{R}$.

PROOF : We must prove that $\mathcal{O}_{\mathcal{R}}(\mathcal{V}_{\mathcal{R}}, \mathcal{L}_{\mathcal{R}})$ never announces false stabilization (safety) and eventually announces stabilization (liveness).

Let G be a distinguished communication graph, P the root of G and E an execution of $\mathcal{R}$ over G.

- *Safety :* for any left factor E' of E, $\mathcal{O}_{\mathcal{R}}(V_R, \mathcal{L}_{\mathcal{R}})(\Pi_{PV_R}E') = \text{true} \Rightarrow E'$ reaches a legitimate configuration of $\mathcal{R}$.

Let E' a left factor of E, we assume that $\exists\ e \in \mathcal{L}_{\mathcal{R}} \mid e$ is a sub-word of $\Pi_{PV_R}E' \Leftrightarrow \mathcal{O}_{\mathcal{R}}(V_R, \mathcal{L}_{\mathcal{R}})(\Pi_{PV_R}E') = \text{true}$ (the variable $size$ of P during E' has consecutively taken the same value v, $2(v+1)$ times).
Let $size_P$ be the variable $size$ of P. We prove that $\mathcal{R}$ is stabilized after the execution of this sequence over the variable $size_P$. This proof is going to be done by refutation. We assume that after the execution of e over $size_P$, the algorithm $\mathcal{R}$ is not stabilized. Let n be the size of G. We have three cases :
1) $v > n$ 2) $v < n$ 3) $v = n$ and $\mathcal{R}$ is not stabilized.

Case 1 : $v > n$
According to the corollary 3, the algorithm $\mathcal{B}$, constructing a spanning tree over a communication graph, is stabilized once n rounds.
According to the corollary 5, the algorithm $\mathcal{C}$, computing the size of a tree, is stabilized once n rounds.
Thus, once $2n$ rounds, $\mathcal{R}$ is stabilized, so once $2n$ rounds, the value of the variable $size_P$ is equal to n.
We have, $0 \leq n < v$, so $0 \leq 2n < 2(v+1)$, thus we cannot observe the same value v, $2(v+1)$ times in $size_P$ if $v > n$.
Case 2 : $v < n$
We assume that the depth of the root is 1. Let p be the depth of a BFS spanning tree over G and n the size of G. Let $d \in [0, p-1]$ and n_d be the number of all processors at distance less or equal than d from the root of G.
According to lemma 3, once the round $2(d+1)$, $size_P \geq n_d$.

If $v < p \leq n$, then it exists at least $(v+1)$ processors at distance less or equal than v from the root of G. Thus, once $2(v+1)$ rounds, $size_P \geq v+1$, i.e. $size_P \neq v$.

If $p \leq v < n$, then once the round $2p$, $size_P = n$. $0 \leq p \leq v \Rightarrow 2p < 2(v+1)$, thus once the round $2(v+1)$, $size_P = n \neq v$.
Case 3 : $v = n$ and $\mathcal{R}$ is not stabilized
According to the corollary 3, the algorithm $\mathcal{B}$, constructing a spanning tree over a communication graph, is stabilized once n rounds.
According to the corollary 5, the algorithm $\mathcal{C}$, computing the size of a tree, is stabilized once n rounds.
Thus, once $2n$ rounds, $\mathcal{R}$ is stabilized, so once $2(n+1)$ rounds, $\mathcal{R}$ is stabilized. □

- *Liveness :* for any left factor E' of E, E' reaches a legitimate configuration of $\mathcal{R}$ $\Rightarrow \exists\ E''$ a left factor of E such as E' is a left factor of E'' and $\mathcal{O}_{\mathcal{R}}(V_R, \mathcal{L}_{\mathcal{R}})(\Pi_{PV}E'') = \text{true}$.

Let E' be a left factor of E such as E' reaches a legitimate configuration of $\mathcal{R}$. Let E'' be a left factor of E such as $|E''| = |E'| + 2(n+1)$. We have E' is a left factor of E''.
Once $\mathcal{R}$ is stabilized, the value of the variable $size_P$ will always equal to n, the size of the communication graph G, thus after E' is executed, $size_P = n$.
Thus $\mathcal{O}_{\mathcal{R}}(V_R, \mathcal{L}_{\mathcal{R}})(\Pi_{PV}E'') = \text{true}$. □.

4.4 Step 4 : global snapshots

We present the algorithm $\mathcal{S}$, which computes global snapshot of a distributed algorithm in the root of a tree. Then, we prove that $\mathcal{S}$ is self-stabilizing.

Presentation We present a synchronous self-stabilizing algorithm $\mathcal{S}$ for constructing global snapshots of a distributed algorithm $\mathcal{A}$ in a tree $T(V, E)$. We use the same model as previously and we associate each node $v_i \in V$ to a set $Child(v_i)$ which is the set of numbers of the links of v_i that reaches a child processor of v_i in T.

Each processor P has a variable *snap* that contains one couple for each processor Q in its sub-tree, this couple is composed by a local snapshot of Q and the distance between P and Q in T. Each processor is continuously computing its local snapshot and reading the variable *snap* of all its children in the tree. Then, it reports these ones, called the snapshot of its sub-tree, to its father, by writing it in its variable *snap*. At the beginning of an arbitrary execution, the only processors guaranteed to compute real snapshot of their sub-tree are the leafs of the tree. Once this snapshot is written in the leafs's variables, the value stored in these variables will always be correct. Once all processors at distance x from a leaf have completed computing the snapshot of there sub-tree correctly and have written it in their variables, their variables remain correct throughout execution, and processors at distance $x + 1$ from a leaf are ready to compute their own snapshot of their sub-tree, and so forth.

We must pay attention, because the snapshot of a sub-tree is composed by local snapshots of all processors in the sub-tree, but these local snapshots are not necessary for the same round.

The root has a variable *queue_snap* which is a queue and contains, in each line, the value of its variable *snap* : every round, the root write at the beginning of its variable *queue_snap* the value of its variable *snap* during this round. Moreover, the root has a variable *snap_global* which contains a global snapshot of the algorithm $\mathcal{A}$: each round, the root constructs a global snapshot of $\mathcal{A}$, with datas in *queue_snap*.

The code of the algorithm $\mathcal{S}$, for the root and for the other processors is :

```
1  Root (P_1) : do forever
2                 true → S := { snap_j | j ∈ Child(P_1) }
3                        snap_1 := (my_local_snapshot, 0) ∪
4                            { (local_snapshot, dist + 1) | (local_snapshot, dist) ∈ S }
5                        queue_snap := add_begin(snap_1, queue_snap)
6                        d_max := Max{ d | (e, d) ∈ line 0 of queue_snap }
7                        snap_global := { e | (e, d) ∈ line d_max − d of queue_snap }
8                 od
9  Other (P_i , i ≠ 1) :
10                do forever
11                true → S := { snap_j | j ∈ Child(P_i) }
12                       snap_i := (my_local_snapshot, 0) ∪
13                           { (local_snapshot, dist + 1) | (local_snapshot, dist) ∈ S }
14                od
```

The state of a processor consists of the value of the program counter and the value of its variable *snap*, S and if the processor is the root, the state also contains the value

of *queue_snap*, d_{max} and *snap_global*. A configuration of the system is a vector of the processors states.

$\mathcal{S}$ makes global snapshots of the algorithm $\mathcal{A}$, over a tree T. Let p be the depth of T (we assume that the depth of the root is 1). The set $S_{\mathcal{S}}$ of legitimate sequences is defined as the set of all configurations sequences in which $\forall i \geq 0$ during a round $2p-1+i$, the variable *snap_global* of the root exactly contains the global snapshot of $\mathcal{A}$, for the round $p+i$.

The algorithm $\mathcal{S}$ is self-stabilizing

Lemma 4 *Let T be a tree and P a processor at distance d from the farest leaf of its sub-tree in T. $\forall i \geq 0$, during the round $d+1+i$, the value of the variable snap of P exactly contains one couple (e_Q, d_Q) for every processor Q in the sub-tree of P such as :*

d_Q is the distance between P and Q.

e_Q is the local snapshot of Q during the round $d+1+i-d_Q$

PROOF : We prove this lemma by induction over d.

Base case : $(d=0)$ Let P be a processor at distance 0 from a leaf : P is a leaf. Every round $k \geq 1$, P reads the value of the variables *snap* of its children and writes them in S (line 11 or 2 if P is also the root). $Child(P) = \emptyset$, so $S = \emptyset$. Then P writes (local-snapshot-of-P-during-the-round-k , 0) in its variable *snap* (line 12 or 3 if P is also the root). Thus, $\forall k \geq 1$, during the round k, the variable *snap* of P exactly contains one couple (local-snapshot-of-P-during-the-round-k , 0). Thus, $\forall i \geq 0$, during the round $i+1$, the variable *snap* of P exactly contains (local-snapshot-of-P-during-the-round-$(i+1)$, 0).

Induction step : (assumes correctness for $d \geq 0$ and proves for $d+1$.) Let P be a processor at distance $d+1$ from the farest leaf of its sub-tree. Every round $k \geq 1$, P reads the value of the variables *snap* of its children (line 11 or 2 if P is also the root). Every children of P are at distance less or equal than d from the farest leaf of their sub-tree, so by induction assumption, for all $F \in Child(P)$, we have :
$\forall i \geq 0$, during the round $d+1+i$, the value of the variable *snap* of F exactly contains one couple (e_Q, d_Q) for every processor Q in the sub-tree of F such as :

d_Q is the distance between F and Q.

e_Q is the local snapshot of Q during the round $d+1+i-d_Q$

$\forall i \geq 0$, during the round $(d+1+i)+1$, P writes in S the value of variables *snap* of all its children. Thus, $\forall i \geq 0$, during the round $(d+1+i)+1$, S exactly contains one couple (e_Q, d_Q) for every processor Q in the sub-tree of P (without P) such as :

$d_Q + 1$ is the distance between P and Q.

e_Q is the local snapshot of Q during the round $d+1+i-d_Q$

Then P writes (local-snapshot-of-P-during-the-round-$(d+1+i)+1$, 0) $\cup$
$\{ (e_Q, d_Q+1) \mid (e_Q, d_Q) \in S \}$ in its variable *snap* (line 12 or 3 if P is also the root).
Thus, $\forall i \geq 0$, during the round $(d+1+i)+1$, the value of the variable *snap* of P exactly contains one couple (e_R, d_R) for each processor R in the sub-tree of P such as :

d_R is the distance between P and R.

e_R is such as

if $R \neq P$, then e_R is the local snapshot of R during the round $d+1+i-d_Q = d+1+i-(d_R-1) = (d+1+i-d_R)+1$

if $R = P$, then e_R is the local snapshot of R during the round (d+1+i)+1, with $d_P = 0$.
Thus e_R is the local snapshot of R during the round $(d+1+i-d_R)+1$. □

Corollary 6 *Let T be a tree and p its depth. $\forall i \geq 0$, during the round $p+i$, the value of the variable snap of the root exactly contains one couple (e_Q, d_Q) for each processor Q in T such as :*
d_Q is the distance between Q and the root.
e_Q is the local snapshot of Q during the round $p+i-d_Q$.

PROOF : The root of T is at distance $d = (p-1)$ from the farest leaf of T. □

We note $queue\_snap(l)$, the value of the variable $queue\_snap$ of the root at the line l.

Lemma 5 *Let $T = (V, E)$ be a tree and p its depth.*
$\forall i \geq 0$ and $\forall l \in [0, i]$, during the round $p+i$, $queue\_snap(l) = \{ (e_Q, d_Q) \mid \forall Q \in V$,
d_Q is the distance between Q and the root,
e_Q is the local snapshot of Q for the round $p+i-d_Q-l$ $\}$.

PROOF : This is due to the behavior of a queue :

Let T be a tree and p its depth. Let $snap_1$ be the variable $snap$ of the root.

$(i = 0)$; round p ; $l \in [0,0]$ $\rightarrow$
$queue\_snap(0)$ $= snap_1$ for the round p
$=$ one couple (e_Q, d_Q) for each processor Q in T such as
d_Q is the distance between Q and the root,
e_Q is the local snapshot of Q for the round $p - d_Q$
$= p + i - d_Q - l$, because $(l = i)$.
$(i = 1)$; round $p+1$; $l \in [0,1]$ $\rightarrow$
$\vdots$ $queue\_snap(1)$ $= snap_1$ for the round p
$=$ one couple (e_Q, d_Q) for each processor Q in T such as
$\vdots$ d_Q is the distance between Q and the root,
e_Q is the local snapshot of Q for the round $p - d_Q$
$\vdots$ $= p + i - d_Q - l$, because $(l = i)$.
$queue\_snap(0)$ $= snap_1$ for the round $p+1$
$\vdots$ $=$ one couple (e_Q, d_Q) for each processor Q in T such as
d_Q is the distance between Q and the root,
$\vdots$ e_Q is the local snapshot of Q for the round $(p+1) - d_Q$
$= p + i - d_Q - l$, because $(l = i - 1)$.
$\vdots$
(i) ; round $p+i$; $l \in [0,i]$ $\rightarrow$
$queue\_snap(i)$ $= snap_1$ for the round p
$=$ one couple (e_Q, d_Q) for each processor Q in T such as
d_Q is the distance between Q and the root,
e_Q is the local snapshot of Q for the round $p - d_Q$
$= p + i - d_Q - l$, because $(l = i)$.

$$
\begin{aligned}
queue\_snap(i-1) &= snap_1 \text{ for the round } p+1 \\
&\vdots \\
&= \text{one couple } (e_Q, d_Q) \text{ for each processor } Q \text{ in } T \text{ such as} \\
&\quad d_Q \text{ is the distance between } Q \text{ and the root,} \\
&\vdots \\
&\quad e_Q \text{ is the local snapshot of } Q \text{ for the round } (p+1)-d_Q \\
&\qquad = p+i-d_Q-l, \text{ because } (l=i-1). \\
&\vdots \\
queue\_snap(0) &= snap_1 \text{ for the round } p+i \\
&= \text{one couple } (e_Q, d_Q) \text{ for each processor } Q \text{ in } T \text{ such as} \\
&\quad d_Q \text{ is the distance between } Q \text{ and the root,} \\
&\quad e_Q \text{ is the local snapshot of } Q \text{ for the round } (p+i)-d_Q \\
&\qquad = p+i-d_Q-l, \text{ because } (l=0).
\end{aligned}
$$

Theorem 3 *Let T be a tree and p its depth. $\forall i \geq 0$, during the round $2p-1+i$, the variable snap_global of the root exactly contains a global snapshot of $\mathcal{A}$ for the round $p+i$.*

PROOF :
Let $T=(V,E)$ be a tree and p its depth.
$\forall i \geq 0$, during the round $2p-1+i$, the variable *snap_global* of the root exactly contains a global snapshot of $\mathcal{A}$ for the round $p+i$.
is equivalent to :
$\forall i \geq 0$, during the round $2p-1+i$,
snap_global $= \{e_Q \mid \forall Q \in V, e_Q$ is the local snapshot of P for the round $p+i\}$.
According to lemma 5, $\forall i \geq 0$, during the round $p+(p-1)+i$,
$\forall l \in [0,(p-1)+i]$, *queue_snap*$(l) = \{(e_Q, d_Q) \mid \forall Q \in V$,
d_Q is the distance between Q and the root,
e_Q is the local snapshot of Q for the round $(2p-1)+i-d_Q-l\}$.
Thus, $\forall i \geq 0$, during the round $p+(p-1)+i$,
$\{(e_Q, d_Q) \in$ *queue_snap*$(l) \mid l=(p-1)-d_Q\} = \{(e_Q, d_Q) \mid$
d_Q is the distance between Q and the root,
e_Q is the local snapshot of Q for the round $(2p-1)+i-d_Q-l =$
$(2p-1)+i-d_Q-(p-1-d_Q)=p+i\ \}$.
□

Corollary 7 *The algorithm $\mathcal{S}$ is self-stabilizing for $S_{\mathcal{S}}$.*

Corollary 8 *Let T be a tree and p its depth, once $2p-1$ rounds, the algorithm $\mathcal{S}$ is stabilized for $S_{\mathcal{S}}$.*

Corollary 9 *Let T be a tree and n its size, once $2n-1$ rounds, the algorithm $\mathcal{S}$ is stabilized for $S_{\mathcal{S}}$.*

4.5 Step 5 and 6 : $\mathcal{A}$ can be observed

Let $\mathcal{P}$ be a problem that can be solved by a synchronous self-stabilizing solution. $\mathcal{P}$ assumes a distinguished network.
Let $\mathcal{A}$ be a synchronous self-stabilizing algorithm solving $\mathcal{P}$ and G a distinguished communication graph.
We construct an observer of $\mathcal{A}$.

1. Let $\mathcal{B}$ be the self-stabilizing algorithm previously presented. $\mathcal{B}$ builds a spanning tree $T = (V, E_T)$ over a distinguished network $G = (V, E_G)$.

2. Let $\mathcal{C}$ be the self-stabilizing algorithm previously presented. $\mathcal{C}$ computes the size n of a tree $T = (V, E_T)$, $n = |V|$.

3. Let $\mathcal{R} = \mathcal{B} \circ \mathcal{C}$ be the self-stabilizing algorithm over a distinguished network $G = (V, E_G)$. $\mathcal{O}_{\mathcal{R}}(\{size\}, \{(v^{2(v+1)}) \mid v \in \mathbb{N}^+\})$ is an observer of $\mathcal{R}$.

4. Let $\mathcal{S}$ be the synchronous self-stabilizing algorithm previously presented. $\mathcal{S}$ computes global snapshots of $\mathcal{A}$ over a tree $T = (V, E_T)$.

5. let $\mathcal{F} = \mathcal{R} \circ \mathcal{S}$ be the synchronous self-stabilizing algorithm over a distinguished network $G = (V, E_G)$.

Theorem 4 *We define $\mathcal{L}_{\mathcal{F}}$ as $\mathcal{L}_{\mathcal{F}} = \{ (v^{2(v+1)} ; \_^{2v-1}) \mid v \in \mathbb{N}^+ \}$ and $V_F = \{size\}$. $\mathcal{O}_{\mathcal{F}}(V_F, \mathcal{L}_{\mathcal{F}})$ is an observer of $\mathcal{F}$.*

PROOF : We must prove that $\mathcal{O}_{\mathcal{F}}(V_F, \mathcal{L}_{\mathcal{F}})$ never announces false stabilization (safety) and eventually announces stabilization (liveness).

Let $G = (V, E_G)$ be a distinguished communication graph, P the root of G and E an execution of $\mathcal{F}$ over G.

Safety :
Let E' a left factor of E, we assume that:
$\exists\ e \in \mathcal{L}_{\mathcal{F}} = \{ (v^{2(v+1)} ; \_^{2v-1}) \mid v \in \mathbb{N}^+ \}$ such as e is a sub-word of $\Pi_{PV_F} E'$
$\Leftrightarrow \mathcal{O}_{\mathcal{F}}(V_F, \mathcal{L}_{\mathcal{F}})(\Pi_{PV_F} E') =$ true.
Let $size_P$ be the variable $size$ of P. We prove that $\mathcal{F}$ is stabilized after the execution of the sequence e on the variable $size_P$.

According to theorem 2, $\mathcal{O}_{\mathcal{R}}$ is an observer of $\mathcal{R}$ thus : the value of $size_P$ is v, during $2(v+1)$ rounds implies that the algorithm $\mathcal{R}$ is stabilized. Moreover,the algorithm $\mathcal{R}$ is stabilized implies that the algorithm $\mathcal{B}$ is stabilized (so $size_P = n = |V|$) and the algorithm $\mathcal{C}$ is stabilized, i.e. exists a spanning tree $T = (V, E_T)$ over G.

$\mathcal{S}$ is a synchronous self-stabilizing algorithm which computes global snapshots of $\mathcal{A}$ over a tree $T = (V, E_T)$. According to corollary 9, if n is the size of T, then once $2n-1$ rounds, the algorithm $\mathcal{S}$ is stabilized.

After the execution of $\{ (v^{2(v+1)}) \mid v \in \mathbb{N}^+ \}$ in $size_P$, $size_P = v = n$. Thus, after the execution of $\{ (v^{2(v+1)} ; \_^{2v-1}) \mid v \in \mathbb{N}^+ \}$ in $size_P$, $\mathcal{R}$ is stabilized and $size_P = v = n$ and $\mathcal{S}$ is stabilized. Thus, after the execution of $\{ (v^{2(v+1)} ; \_^{2v-1}) \mid v \in \mathbb{N}^+ \}$ in $size_P$, $\mathcal{F} = \mathcal{R} \circ \mathcal{S}$ is stabilized.

Liveness :
Let n be the size of G. Let E' be a left factor of E such as E' reaches a legitimate configuration of $\mathcal{F}$. Let E'' be a left factor of E such as $|E''| = |E'| + 2(n+1)$. According to theorem 2, $\mathcal{O}_{\mathcal{R}}(V_R, \mathcal{L}_{\mathcal{R}})(\Pi_{PV} E'') =$ true.
We have $V_R = \{size\} = V_F$ and $\mathcal{L}_{\mathcal{R}} = \{ (v^{2(v+1)} \mid v \in \mathbb{N}^+ \}$ and $\mathcal{L}_{\mathcal{F}} = \{ (v^{2(v+1)} ; \_^{2v-1}) \mid v \in \mathbb{N}^+ \}$.
Let E''' be a left factor of E such as $|E'''| = |E'| + 2(n+1) + (2n-1)$. We have E' is a left factor of E''' and $\mathcal{O}_{\mathcal{F}}(V_F, \mathcal{L}_{\mathcal{F}})(\Pi_{PV} E''') =$ true. $\square$.

6. Let $\mathcal{H} = \mathcal{A} \circ \mathcal{F}$ be the synchronous self-stabilizing algorithm over a distinguished network $G = (V, E_G)$. Let $S_{\mathcal{A}}$ be the set of all legitimate configurations of $\mathcal{A}$.

Theorem 5 *We define* $V_H = \{size, snap\_global\}$ *and* $\mathcal{L}_{\mathcal{H}}$ *as* $\mathcal{L}_{\mathcal{H}} = \{ ((v, \_)^{2(v+1)} ; (\_, \_)^{2v-1} ; (\_, c)) \mid c \in S_{\mathcal{A}} \wedge v \in \mathbb{N}^+ \}$. $\mathcal{O}_{\mathcal{H}}(V_H, \mathcal{L}_{\mathcal{H}})$ *is an observer of* $\mathcal{H}$.

PROOF : We must prove that $\mathcal{O}_{\mathcal{H}}(V_H, \mathcal{L}_{\mathcal{H}})$ never announces false stabilization (safety) and eventually announces stabilization (liveness).

Safety :

Let $G = (V, E_G)$ be a distinguished communication graph, P the root of G and E an execution of $\mathcal{H}$ over G.
Let E' be a left factor of E, we assume that :
$\exists e \in \mathcal{L}_{\mathcal{H}} = \{ ((v, \_)^{2(v+1)} ; (\_, \_)^{2v-1} ; (\_, c)) \mid c \in S_{\mathcal{A}} \wedge v \in \mathbb{N}^+ \}$ such as e is a sub-word of $\Pi_{PV_H} E' \Leftrightarrow \mathcal{O}_{\mathcal{H}}(V_H, \mathcal{L}_{\mathcal{H}})(\Pi_{PV_H} E') = \text{true}$.
We prove that $\mathcal{H}$ is stabilized after the execution of e on the variables $size_P$ and $snap\_global_P$.

According to theorem 4, $\mathcal{O}_{\mathcal{F}}(V_F, \mathcal{L}_{\mathcal{F}})$ is an observer of $\mathcal{F}$.
Let $\mathcal{L}_{\mathcal{F}}{}' = \{ ((v, \_)^{2(v+1)} ; (\_, \_)^{2v-1}) \mid v \in \mathbb{N}^+ \}$. We have : $\mathcal{O}_{\mathcal{F}}{}'(V_H, \mathcal{L}_{\mathcal{F}}{}')$ is an observer of $\mathcal{F}$. Thus, if $\exists\ e \in \mathcal{L}_{\mathcal{F}}{}' \in$ such as e is a sub-word of $\Pi_{PV_H} E'$ then $\mathcal{F}$ is stabilized after the execution of e on the variables $size_P$ and $snap\_global_P$.
Moreover, after the execution of $c \mid c \in S_{\mathcal{A}}$ in the variable $snap\_global_P$, $\mathcal{A}$ is stabilized.
Thus, if $\exists\ e \in \{ ((v, \_)^{2(v+1)} ; (\_, \_)^{2v-1} ; (\_, c)) \mid c \in S_{\mathcal{A}} \wedge v \in \mathbb{N}^+ \}$ such as e is a sub-word of $\Pi_{PV_H} E'$ then $\mathcal{F}$ and $\mathcal{A}$ are stabilized after the execution of e on the variables $size_P$ and $snap\_global_P$.

Liveness :

Let n be the size of G. Let E' be a left factor of E such as E' reaches a legitimate configuration of $\mathcal{H}$.

Let E'' be a left factor of E such as $|E''| = |E'| + 2(n+1) + 2n - 1$. According to theorem 4, $\mathcal{O}_{\mathcal{F}}(V_F, \mathcal{L}_{\mathcal{F}})(\Pi_{PV} E'') = \text{true}$.

We have $V_F = \{size\}$, $V_H = \{size, snap\_global\}$ and $\mathcal{L}_{\mathcal{F}} = \{ (v^{2(v+1)} ; \_^{2v-1}) \mid v \in \mathbb{N}^+ \}$ and $\mathcal{L}_{\mathcal{H}} = \{ ((v, \_)^{2(v+1)} ; (\_, \_)^{2v-1} ; (\_, c)) \mid c \in S_{\mathcal{A}} \wedge v \in \mathbb{N}^+ \}$.

Let E''' be a left factor of E such as $|E'''| = |E'| + 2(n+1) + (2n-1) + 1$. We have E' is a left factor of E''' and $\mathcal{O}_{\mathcal{H}}(V_H, \mathcal{L}_{\mathcal{H}})(\Pi_{PV} E''') = \text{true}$. □.

Thus $\mathcal{H}$ can be observed, and especially, $\mathcal{A}$ can be observed.

5 Conclusion

In this paper, we introduce the notion of a local observer for self-stabilizing algorithms on synchronous networks. The observer was defined in an abstract way and we did not consider implementation details.

Our result is that, provided the network has a distinguished node, any problem having a self-stabilizing solution has also a self-stabilizing solution that can be observed by an observer located at the distinguished node.

The next step will be to consider uniform networks in which the observer has no reason to be placed at some node rather than at some other.

References

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[Dol00] Shlomi Dolev. *Self-Stabilization.* MIT Press, Cambridge, MA, 2000. Ben-Gurion University of the Negev, Israel.

[Har78] Michael A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, 1 edition, 1978.

[LS92] Chengdian Lin and Janos Simon. Observing self-stabilization. In Maurice Herlihy, editor, *Proceedings of the 11th Annual Symposium on Principles of Distributed Computing*, pages 113–124, Vancouver, BC, Canada, August 1992. ACM Press.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge Uni Press, Cambridge, 1994.