Self-Stabilizing Distributed Cooperative Reset

Stéphane Devismes, Colette Johnen

Self-Stabilizing Distributed Cooperative Reset*

Stéphane Devismes† and Colette Johnen‡

† Université Grenoble Alpes, VERIMAG, UMR 5104, France
‡ Université Bordeaux, LaBRI, UMR 5800, France

Abstract

Self-stabilization is a versatile fault-tolerance approach that characterizes the ability of a system to eventually resume a correct behavior after any finite number of transient faults. In this paper, we propose a self-stabilizing reset algorithm working in anonymous networks. This algorithm resets the network in a distributed non-centralized manner, *i.e.*, it is multi-initiator, as each process detecting an inconsistency may initiate a reset. It is also cooperative in the sense that it coordinates concurrent reset executions in order to gain efficiency. Our approach is general since our reset algorithm allows to build self-stabilizing solutions for various problems and settings. As a matter of facts, we show that it applies to both static and dynamic specifications since we propose efficient self-stabilizing reset-based algorithms for the (1-minimal) (f, g)-alliance (a generalization of the dominating set problem) in identified networks and the unison problem in anonymous networks. Notice that these two latter instantiations enhance the state of the art. Indeed, in the former case, our solution is more general than the previous ones, while in the latter case, the complexity of our unison algorithm is better than that of previous solutions of the literature.

Keywords: Distributed algorithms, self-stabilization, reset, alliance, unison.

1 Introduction

In distributed systems, a *self-stabilizing* algorithm is able to recover a correct behavior in finite time, regardless of the *arbitrary* initial configuration of the system, and therefore also after a finite number of transient faults, provided that those faults do not alter the code of the processes.

After more than 40 years of researches, many self-stabilizing solutions has been proposed to solve various problems in various settings. Drawing on this experience, general methodologies for making distributed algorithms self-stabilizing have been proposed. In particular, Katz and Perry [26] give a characterization of problems admitting a self-stabilizing solution. Precisely, they describe a general algorithm that transforms almost any algorithm (specifically, those algorithms that can be self-stabilized) into its corresponding stabilizing version. However, this transformer is, by essence, inefficient both in terms of space and time complexities: its purpose is only to demonstrate the feasibility of the transformation.

Interestingly, many proposed general methods [26, 7, 4, 5] are based on reset algorithm.s Such algorithms are initiated when an inconsistency is discovered in the network, and aim at reinitializing the system to a correct (pre-defined) configuration.

*This study was partially supported by the ANR project DESCARTES : ANR-16-CE40-0023 and ANR project ESTATE : ANR-16 CE25-0009-03.

A reset algorithm may be centralized at a leader process (*e.g.*, see [4]), or fully distributed, meaning multi-initiator (as our proposal here). In the former case, either the reset is coupled with a snapshot algorithm (a *global checking* of the network), or processes detecting an incoherence (by *local checking* [6]) should request a reset to the leader. In the fully distributed case, resets at locally initiated by processes detecting inconsistencies. This latter approach is considered as more efficient, when the concurrent resets are coordinated. In other words, concurrent resets have to be *cooperative* [27] to ensure the fast convergence of the system to a consistent global state.

Self-stabilization makes no hypotheses on the nature (*e.g.*, memory corruptions or topological changes) or extent of transient faults that could hit the system, and the system recovers from the effects of those faults in a unified manner. Such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system are violated. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, the maximum duration of the stabilization phase.

General schemes and efficiency are usually understood as orthogonal issues. In this paper we tackle this problem by proposing an efficient self-stabilizing reset algorithm working in any anonymous connected network. Our algorithm is written in the locally shared memory model with composite atomicity, where executions proceed in atomic steps and the asynchrony is captured by the notion of *daemon*. The most general daemon is the *distributed unfair daemon*. Hence, solutions stabilizing under such an assumption are highly desirable, because they work under any other daemon assumption. The daemon assumption and time complexity are closely related. The *stabilization time* is usually evaluated in terms of rounds, which capture the execution time according to the speed of the slowest processes. But, another crucial issue is the number of local state updates, called *moves*. Indeed, the stabilization time in moves captures the amount of computations an algorithm needs to recover a correct behavior. Now, this complexity can be bounded only if the algorithm works under an unfair daemon. If an algorithm requires a stronger daemon to stabilize, *e.g.*, a *weakly fair* daemon, then it is possible to construct executions whose convergence is arbitrarily long in terms of atomic steps (and so in moves), meaning that, in such executions, there are processes whose moves do not make the system progress towards the convergence. In other words, these latter processes waste computation power and so energy. Such a situation should be therefore prevented, making solutions working under the unfair daemon more desirable. There are many self-stabilizing algorithms proven under the distributed unfair daemon, *e.g.*, [2, 13, 22]. However, analyses of the stabilization time in moves is rather unusual and this may be an important issue. Indeed, recently, several self-stabilizing algorithms which work under a distributed unfair daemon have been shown to have an exponential stabilization time in moves in the worst case, *e.g.*, the silent leader election algorithms from [13] (see [2]), the Breadth-First Search (BFS) algorithm of Huang and Chen [24] (see [14]).

1.1 Contribution

We propose an efficient self-stabilizing reset algorithm working in any anonymous connected network. Our algorithm is written in the locally shared memory model with composite atomicity, assuming a distributed unfair daemon, *i.e.*, the weakest scheduling assumption of the model. It is based on local checking and fully distributed (*i.e.*, multi-initiator). Concurrent resets are locally initiated by processes detecting inconsistencies, these latter being cooperative to gain efficient.

As a matter of facts, our algorithm makes an input algorithm recovering a consistent global state within at most $3n$ rounds, where n is the number of processes. Its move complexity is at most $3n + 3$. Our reset algorithm allows to build efficient self-stabilizing solutions for various problems and settings. In particular, it applies to both static and dynamic specifications. In the former case, the self-stabilizing solution is also

silent [19]: a silent algorithm converges within finite time to a configuration from which the values of the communication registers used by the algorithm remain fixed.

To show the efficiency of our method, we propose two reset-based self-stabilizing algorithms, respectively solving the 1-minimal (f, g)-alliance in identified networks and the unison problem in anonymous networks.

Our 1-minimal (f, g)-alliance algorithm is also silent, its stabilization time is in $O(n)$ rounds and $O(\Delta.n.m)$ moves, where D is the network diameter and m is the number of edges in the network. To the best of our knowledge, until now there was no self-stabilizing algorithm solving that problem without any restriction on f and g.

Our unison algorithm has a stabilization time is in $O(n)$ rounds and $O(\Delta.n^2)$ moves. Actually, its stabilization times in round matches the one of the best existing solution [9]. However, it achieves a better stabilization time in moves, since the algorithm in [9] stabilizes in $O(D.n^3+\alpha.n^2)$ moves (as shown in [15]), where α is greater than the length of the longest chordless cycle in the network.

1.2 Related Work

Several reset algorithms have been proposed in the literature. In particular, several solutions, *e.g.*, [7, 6], have been proposed in the I/O automata model. In this model, communications are implement using message-passing and assuming weakly fairness. Hence, move complexity cannot be evaluated in that model. [7, 6] additionally assume bound capacity links. In [7], authors introduce the notion of local checking, and propose a method that, given a self-stabilizing global reset algorithm, builds a self-stabilizing solution of any *locally checkable* problem (*i.e.*, problem where inconsistency can be locally detected) in an identified network. The stabilization time of obtained solutions, usually $O(n)$ rounds, depends on the input reset algorithm. In [6], authors focus on an restrictive class of locally checkable problems, those that are also locally correctable. A problem is locally correctable if the global configuration of the network can be corrected be applying independent correction on pair neighboring processes. Now, for example, the 1-minimal (f, g) alliance problem is not locally correctable since there are situations in which the correction of a single inconsistency may provoke a global correction in a domino effect reaction. Notice also that processes are not assumed to be identified in [6], however the considered networks are not fully anonymous either. Indeed, each link has one of its incident processes designated as leader. Notice also that authors show a stabilization time in $O(H)$ when the network is a tree, where H is the tree height.

Self-stabilization by power supply [1] also assumes message-passing with bounded capacity links and process identifiers. Using this technique, the stabilization time is in $O(n)$ rounds in general. Now, only static problems, *e.g.* leader election and spanning tree construction, are considered.

Fully anonymous networks are considered in [5] in message-passing systems with unit-capacity links and assuming weakly fairness. The proposed self-stabilizing reset has a memory requirement in $O(\log^\star(n)$ bits per process. But this small complexity comes at the price of a stabilization time in $O(n \log n)$ rounds.

Finally, Arora and Gouda have proposed a mono-initiator reset algorithm in the locally shared memory model with composite atomicity. Their self-stabilizing reset works in identified networks, assuming a distributed weakly fair daemon. The stabilization time of their solution is in $O(n + \Delta.D)$ rounds, where Δ is the degree of the network.

1.3 Roadmap

The remainder of the paper is organized as follows. In the next section, we present the computational model and basic definitions. In Section 3, we present, prove, and analyze the complexity of our reset algorithm. In

the two last sections, we propose two examples of efficient reset-based algorithms, respectively solving the 1-minimal (f, g)-alliance in identified networks and unison problems in anonymous networks.

2 Preliminaries

2.1 Network

We consider a distributed system made of n interconnected processes. Information exchanges are assumed to be bidirectional. Henceforth, the communication network is conveniently modeled by a simple undirected connected graph $G = (V, E)$, where V is the set of processes and E a set of m edges $\{u, v\}$ representing the ability of processes u and v to directly exchange information together. We denote by D the diameter of G, *i.e.*, the maximum distance between any two pairs of processes. For every edge $\{u, v\}$, u and v are said to be *neighbors*. For every process u, we denote by δ_u the degree of u in G, *i.e.*, the number of its neighbors. Let $\Delta = \max_{u \in V} \delta_u$ be the (maximum) degree of G.

2.2 Computational Model

We use the *composite atomicity model of computation* [16] in which the processes communicate using a finite number of locally shared registers, simply called *variables*. Each process can read its own variables and that of its neighbors, but can write only to its own variables. The *state* of a process is defined by the values of its variables. A *configuration* of the system is a vector consisting of the states of each process.

Every process u can access the states of its neighbors using a *local labeling*. Such labeling is called *indirect naming* in the literature [31]. All labels of u's neighbors are stored into the set $N(u)$. To simplify the design of our algorithms, we sometime consider the *closed neighborhood* of a process u, *i.e.*, the set including u itself and all its neighbors. Let $N[u]$ be the set of labels local to u designating all members of its closed neighborhood, including u itself. In particular, $N(u) \subsetneq N[u]$. We assume that each process u can identify its local label $\alpha_u(v)$ in the sets $N(v)$ of each neighbor v and $N[w]$ of each member w of its closed neighborhood. When it is clear from the context, we use, by an abuse of notation, u to designate both the process u itself, and its local labels (*i.e.*, we simply use u instead of $\alpha_u(v)$ for $v \in N[u]$).

A *distributed algorithm* consists of one local program per process. The *program* of each process consists of a finite set of *rules* of the form

$$\langle label \rangle \ : \ \langle guard \rangle \ \rightarrow \ \langle action \rangle$$

Labels are only used to identify rules in the reasoning. A *guard* is a Boolean predicate involving the state of the process and that of its neighbors. The *action* part of a rule updates the state of the process. A rule can be executed only if its guard evaluates to *true*; in this case, the rule is said to be *enabled*. A process is said to be enabled if at least one of its rules is enabled. We denote by *Enabled*(γ) the subset of processes that are enabled in configuration γ.

When the configuration is γ and *Enabled*$(\gamma) \neq \emptyset$, a non-empty set $\mathcal{X} \subseteq$ *Enabled*(γ) is selected by the so-called *daemon*; then every process of $\mathcal{X}$ *atomically* executes one of its enabled rules,[1] leading to a new configuration γ', and so on. The transition from γ to γ' is called a *step*. The possible steps induce a binary relation over the set of configurations, denoted by $\mapsto$. An *execution* is a maximal sequence of configurations $e = \gamma_0 \gamma_1 \cdots \gamma_i \cdots$ such that $\gamma_{i-1} \mapsto \gamma_i$ for all $i > 0$. The term "maximal" means that the execution is either infinite, or ends at a *terminal* configuration in which no rule is enabled at any process.

[1] In case of several enabled actions at the activated process, the choice of the executed action is nondeterministic.

Each step from a configuration to another is driven by a daemon. We define a daemon as a predicate over executions. In this paper we assume that the daemon is *distributed* and *unfair*. "Distributed" means that while the configuration is not terminal, the daemon should select at least one enabled process, maybe more. "Unfair" means that there is no fairness constraint, *i.e.*, the daemon might never select an enabled process unless it is the only enabled process.

2.3 Self-Stabilization and Silence

Let $\mathtt{A}$ be a distributed algorithm. Let P and P' be two predicates over configurations of $\mathtt{A}$. Let C and C' be two subsets of $\mathcal{C}_{\mathtt{A}}$, the set of $\mathtt{A}$'s configurations.

- P (resp. C) is *closed by* $\mathtt{A}$ if for every step $\gamma \mapsto \gamma'$ of $\mathtt{A}$, $P(\gamma) \Rightarrow P(\gamma')$ (resp. $\gamma \in C \Rightarrow \gamma' \in C$).
- $\mathtt{A}$ *converges from* P' (resp. C') *to* P (resp. C) if each of its executions starting from a configuration satisfying P' (resp. in a configuration of C') contains a configuration satisfying P (resp. a configuration of C).
- P (resp. C) is an *attractor* for $\mathtt{A}$ if P (resp. C) is closed by $\mathtt{A}$ and $\mathtt{A}$ converges from $true$ (resp. from $\mathcal{C}_{\mathtt{A}}$) to P (resp. to C).

Let SP be a specification, *i.e.*, a predicate over executions. Algorithm $\mathtt{A}$ is *self-stabilizing* for SP if there exists a non-empty subset of its configurations $\mathcal{L}$, called the *legitimate* configurations, such that $\mathcal{L}$ is an attractor for $\mathtt{A}$ and every execution of $\mathtt{A}$ that starts in a configuration of $\mathcal{L}$ satisfies SP. Configurations of $\mathcal{C}_{\mathtt{A}} \setminus \mathcal{L}$ are called the *illegitimate* configurations.

In our model, an algorithm is *silent* [19] if and only if all its executions are finite. Let SP' be an predicate over configurations of $\mathtt{A}$. Usually, silent self-stabilization is (equivalently) reformulated as follows. $\mathtt{A}$ is *silent and self-stabilizing* for the SP' if all its executions are finite and all its terminal configurations satisfy SP'. Of course, in silent self-stabilization, the set of legitimate configurations is chosen as the set of terminal configurations.

2.4 Time Complexity

We measure the time complexity of an algorithm using two notions: *rounds* [18, 11] and *moves* [16]. We say that a process *moves* in $\gamma_i \mapsto \gamma_{i+1}$ when it executes a rule in $\gamma_i \mapsto \gamma_{i+1}$.

The definition of round uses the concept of *neutralization*: a process v is *neutralized* during a step $\gamma_i \mapsto \gamma_{i+1}$, if v is enabled in γ_i but not in configuration γ_{i+1}, and it is not activated in the step $\gamma_i \mapsto \gamma_{i+1}$.

Then, the rounds are inductively defined as follows. The first round of an execution $e = \gamma_0\gamma_1\cdots$ is the minimal prefix $e' = \gamma_0 \cdots \gamma_j$, such that every process that is enabled in γ_0 either executes a rule or is neutralized during a step of e'. Let e'' be the suffix $\gamma_j\gamma_{j+1}\cdots$ of e. The second round of e is the first round of e'', and so on.

The *stabilization time* of a self-stabilizing algorithm is the maximum time (in moves or rounds) over every possible execution (starting from any initial configuration) to reach a legitimate configuration.

2.5 Composition

We denote by $\mathtt{A} \circ \mathtt{B}$ the composition of the two algorithms $\mathtt{A}$ and $\mathtt{B}$ which is the distributed algorithm where the local program $(\mathtt{A} \circ \mathtt{B})(u)$, for every process u, consists of all variables and rules of both $\mathtt{A}(u)$ and $\mathtt{B}(u)$.

3 Self-Stabilizing Distributed Reset Algorithm

3.1 Overview of the Algorithm

In this section, we present our distributed cooperative reset algorithm, called SDR. The formal code of SDR, for each process u, is given in Algorithm 1. This algorithm aims at reinitializing an input algorithm I when necessary. SDR is self-stabilizing in the sense that the composition I $\circ$ SDR is self-stabilizing for the specification of I. Algorithm SDR works in anonymous networks and is actually is multi-initiator: whenever a process u locally detects an inconsistency in I (*i.e.*, whenever the predicate $\neg\mathbf{P\_ICorrect}(u)$ holds), u initiates a reset. Hence, several resets may be executed concurrently. In this case, they are coordinated: a reset may be partial since we try to prevent resets to overlaps.

3.2 The variables

Each process u maintains two variables in Algorithm SDR: $st_u \in \{C, RB, RC\}$, the *status* of u with respect to the reset, and $d_u \in \mathbb{N}$, the *distance* of u in a reset.

Variable st_u. If u is not currently involved into a reset, then it has status C, which stands for *correct*. Otherwise, u has status either RB or RF, which respectively mean *reset broadcast* and *reset feedback*. Indeed, a reset consists in a (maybe partial) *Propagation of Information with Feedback (PIF)* during which processes reset their local state (using the macro $reset$) in I during the broadcast phase. When a reset locally terminates at process u (*i.e.*, when u goes back to status C by executing $\mathbf{rule\_C}(u)$), each member v of its closed neighborhood satisfies $\mathbf{P\_reset}(v)$, meaning that they are in a pre-defined initial state of I. At the global termination of a reset, every process u involved into that reset has a state in I which is consistent *w.r.t.* that of its neighbors, *i.e.*, $\mathbf{P\_ICorrect}(u)$ holds. Notice that, to ensure that $\mathbf{P\_ICorrect}(u)$ holds at the end of a reset and for liveness issues, we enforce each process u stops executing I whenever a member of its closed neighborhood (may be the process itself) is involved into a reset: whenever $\neg PClean(u)$ holds, u is not allowed to execute I.

Variable d_u. This variable is meaningless when u is not involved into a reset (*i.e*, when u has status C). Otherwise, the distance values are used to arrange processes involved into resets as a *Directed Acyclic Graph (DAG)*. This distributed structure allows to prevent both livelock and deadlock. Any process u initiating a reset (using rule $\mathbf{rule\_R}(u)$), takes distance 0. Otherwise, when a reset is propagated to u (*i.e.*, when $\mathbf{rule\_RB}(u)$ is executed), d_u is set to the minimum distance of a neighbor involved in a broadcast phase plus 1 (see the macro $compute(u)$).

3.3 Typical Execution

Assume the system starts from a configuration where, for every process u, $st_u = C$. A process u detecting an inconsistency in I (*i.e.*, when $\neg\mathbf{P\_ICorrect}(u)$ holds) initiates a reset using $\mathbf{rule\_R}(u)$, unless one of neighbor v is already broadcasting a reset, in which case it joins the broadcast of some neighbor by $\mathbf{rule\_RB}(u)$. To initiate a reset, u sets (st_u, d_u) to $(RB, 0)$ meaning that u is the root of a reset (see macro $beRoot(u)$), and resets its I's variables by executing the macro $reset(u)$. Whenever a process v as a neighbor involved in a broadcast phase of a reset (status RB), it also joins a reset using $\mathbf{rule\_RB}(u)$, even if its state in *I* is correct, (*i.e.*, even if $\mathbf{P\_ICorrect}(v)$ holds). In this case, u also switches its status to RB and resets its I's variables, yet u sets d_u to the minimum distance of its neighbors involved in a broadcast

phase plus 1 (see the macro $compute(u)$). Hence, if the configuration of I is not legitimate, then within at most n rounds, each process receives the broadcast of some reset. Meanwhile, processes (temporarily) stop executing I until the reset terminates in their closed neighborhood.

When a process u involved in the broadcast phase of some reset realizes that all its neighbors are involved into a reset (*i.e.*, have status EB or EF), it initiates the feedback phase by switching to status RF, using $\mathbf{rule\_RF}(u)$. The feedback phase is then propagated up in the DAG described by the distance value: a broadcasting process u switches to the feedback phase if each of its neighbors v has not status C and if $d_v > d_u$, then v has status EF. This way the feedback phase is propagated up into the DAG within at most n additional rounds. Once a root of some reset has status EF, it can initiate the last phase of the reset: all processes involves into the reset has to switch to status C meaning that the reset done using $\mathbf{rule\_C}$. The values C are propagated down into the reset DAGs within at most n additional rounds. A process u can executing I again when all members of its closed neighborhood (that is, including u itself) have status C, when it satisfies $\mathbf{P\_Clean}(u)$.

Hence, overall in this execution, the system reaches a configuration γ where all resets are done within at most $3n$ rounds. In γ, all processes have status C. However, process has not necessarily kept a state satisfying $\mathbf{P\_reset}$ in this configuration. Indeed, some process may have started executing I again before γ. However, the predicate $\mathbf{P\_Clean}$, which allows to block the local execution of I, ensures that no resetting process has been involved in this latter execution of I. Hence, SDR rather ensures that all processes are in states in I that are coherent with each other from γ. That is, γ is a so-called *normal configuration* where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds for every process u.

3.4 Stabilization of the Reset

If a process u is in an incorrect state of Algorithm SDR (*i.e.*, if $\mathbf{P\_R1}(u) \vee \mathbf{P\_R2}(u)$ holds), we proceed as for Algorithm I. Either it joins an existing reset (using $\mathbf{rule\_RB}(u)$) because at least one of its neighbors is in a broadcast phase, or it initiates its own reset using $\mathbf{rule\_R}(u)$. Notice also that starting from an arbitrary configuration, the system may contain some reset in progress. However, similarly to the typical execution, the system stabilizes within at most $3n$ rounds to a normal configuration.

Algorithm SDR is also efficient in moves. Indeed, in Sections 5 and 6 we will give two examples of composition I $\circ$ SDR that stabilize in a polynomial number of moves. Such complexities are mainly due to the coordination of the resets which, in particular, guaranteed that if a process u is enabled to initiate a reset ($\mathbf{P\_Up}(u)$) or the root of a reset with status RB, then it is in that situation since the initial configuration (*cf.*, Theorem 3, page 11).

3.5 Requirements on the input algorithm

According to the previous explanation, Algorithm I should satisfy the following prerequisites:

1. Algorithm I should not write into the variables of SDR, *i.e.*, variables st_u and d_u, for every process u.
2. For each process u, Algorithm I should provide the two input predicates $\mathbf{P\_ICorrect}(u)$ and $\mathbf{P\_reset}(u)$ to SDR, and the macro $reset(u)$. Those inputs should satisfy:
   (a) $\mathbf{P\_ICorrect}(u)$ does not involve any variable of SDR and is closed by Algorithm I.
   (b) $\mathbf{P\_reset}(u)$ involves neither a variable of SDR nor a variable of a neighbor of u.

(c) If $\neg\mathbf{P\_ICorrect}(u) \vee \neg\mathbf{P\_Clean}(u)$ holds (*n.b.* $\mathbf{P\_Clean}(u)$ is defined in SDR), then no rule of Algorithm I is enabled at u.

(d) If $\mathbf{P\_reset}(v)$ holds, for every $v \in N[u]$, then $\mathbf{P\_ICorrect}(u)$ holds.

(e) If u performs a move in $\gamma \mapsto \gamma'$, where, in particular, it modifies its variables in Algorithm I by executing $reset(u)$ (only), then $\mathbf{P\_reset}(u)$ holds in γ'.

Algorithm 1 Algorithm SDR, code for every process u

Inputs:

- $\mathbf{P\_ICorrect}(u)$: predicate from the input algorithm
- $\mathbf{P\_reset}(u)$: predicate from the input algorithm
- $reset(u)$: macro from the input algorithm

Variables:

- $st_u \in \{C, RB, RF\}$: the status of u
- $d_u \in \mathbb{N}$: the distance value associated to u

Predicates:

- $\mathbf{P\_Correct}(u) \equiv st_u = C \Rightarrow \mathbf{P\_ICorrect}(u)$
- $\mathbf{P\_Clean}(u) \equiv \forall v \in N[u], st_u = C$
- $\mathbf{P\_R1}(u) \equiv st_u = C \wedge \neg\mathbf{P\_reset}(u) \wedge (\exists v \in N(u) \mid st_v = RF)$
- $\mathbf{P\_RB}(u) \equiv st_u = C \wedge (\exists v \in N(u) \mid st_v = RB)$
- $\mathbf{P\_RF}(u) \equiv st_u = RB \wedge \mathbf{P\_reset}(u) \wedge (\forall v \in N(u), (st_v = RB \wedge d_v \leq d_u) \vee (st_v = RF \wedge \mathbf{P\_reset}(v)))$
- $\mathbf{P\_C}(u) \equiv st_u = RF \wedge (\forall v \in N[u], \mathbf{P\_reset}(v) \wedge ((st_v = RF \wedge d_v \geq d_u) \vee (st_v = C)))$
- $\mathbf{P\_R2}(u) \equiv st_u \neq C \wedge \neg\mathbf{P\_reset}(u)$
- $\mathbf{P\_Up}(u) \equiv \neg\mathbf{P\_RB}(u) \wedge (\mathbf{P\_R1}(u) \vee \mathbf{P\_R2}(u) \vee \neg\mathbf{P\_Correct}(u))$

Macros:

- $beRoot(u)$: $st_u := RB$; $d_u := 0$;
- $compute(u)$: $st_u := RB$; $d_u := \text{argmin}_{(v \in N(u) \wedge st_v = RB)}(d_v) + 1$;

Rules:

- $\mathbf{rule\_RB}(u)$: $\mathbf{P\_RB}(u) \rightarrow compute(u); reset(u);$
- $\mathbf{rule\_RF}(u)$: $\mathbf{P\_RF}(u) \rightarrow st_u := RF;$
- $\mathbf{rule\_C}(u)$: $\mathbf{P\_C}(u) \rightarrow st_u := C;$
- $\mathbf{rule\_R}(u)$: $\mathbf{P\_Up}(u) \rightarrow beRoot(u); reset(u);$

4 Correctness and Complexity Analysis

4.1 Partial Correctness

Lemma 1 *In any terminal configuration of* SDR, $\neg\mathbf{P\_R1}(u) \wedge \neg\mathbf{P\_R2}(u) \wedge \mathbf{P\_Correct}(u)$ *holds for every process* u.

Proof. Let u be any process and consider any terminal configuration of SDR. Since $\mathbf{rule\_RB}(u)$

and $\mathbf{rule\_R}(u)$ are disabled, $\neg\mathbf{P\_RB}(u)$ and $\neg\mathbf{P\_Up}(u)$ hold. Since $\neg\mathbf{P\_RB}(u) \wedge \neg\mathbf{P\_Up}(u)$ implies $\neg\mathbf{P\_R1}(u) \wedge \neg\mathbf{P\_R2}(u) \wedge \mathbf{P\_Correct}(u)$, we are done. □

Since $\neg\mathbf{P\_R2}(u) \equiv st_u = C \vee \mathbf{P\_reset}(u)$, we have the following corollary.

Corollary 1 *In any terminal configuration of* SDR*, $st_u = C \vee \mathbf{P\_reset}(u)$ holds for every process u.*

Lemma 2 *In any terminal configuration of* SDR*, $st_u \neq RB$ for every process u.*

Proof. Assume, by the contradiction, that some process u satisfies $st_u = RB$ in a terminal configuration of SDR. Without the loss of generality, assume u is a process such that $st_u = RB$ with d_u maximum. First, $\mathbf{P\_reset}(u)$ holds by Corollary 1. Then, every neighbor v of u satisfies $st_v \neq C$, since otherwise $\mathbf{rule\_RB}(v)$ is enabled. So, every v satisfies $st_v \in \{RB, RF\}$, $st_v = RB \Rightarrow dist_v \leq d_u$ (by definition of u), and $st_v = RF \Rightarrow \mathbf{P\_reset}(v)$ (by Corollary 1). Hence, $\mathbf{rule\_RF}(u)$ is enabled, a contradiction. □

Lemma 3 *In any terminal configuration of* SDR*, $st_u \neq RF$ for every process u.*

Proof. Assume, by the contradiction, that some process u satisfies $st_u = RF$ in a terminal configuration of SDR. Without the loss of generality, assume u is a process such that $st_u = RF$ with d_u minimum. First, every neighbor v of u satisfies $st_v \neq RB$, by Lemma 2. Then, every neighbor v of u such that $st_v = C$ also satisfies $\mathbf{P\_reset}(v)$, since otherwise $\mathbf{P\_R1}(v)$ holds, contradicting then Lemma 1. Finally, by definition of u and by Corollary 1, every neighbor v of u such that $st_v = RF$ both satisfies $dist_v \geq d_u$ and $\mathbf{P\_reset}(v)$. Hence, $\mathbf{rule\_C}(u)$ is enabled, a contradiction. □

Theorem 1 *For every configuration γ of* SDR*, γ is terminal if and only if $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds in γ, for every process u.*

Proof. Let u be any process and assume γ is terminal. By Lemmas 2 and 3, $st_u = C$ holds in γ. So, $\mathbf{P\_Clean}(u)$ holds in γ. Moreover, since $\mathbf{P\_Correct}(u)$ holds (Lemma 1), $\mathbf{P\_ICorrect}(u)$ also holds in γ, and we are done.

Assume now that for every process u, $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds in γ. Then, $st_u = C$ for every process u, and so $\mathbf{rule\_C}(u)$, $\mathbf{rule\_RB}(u)$, and $\mathbf{rule\_RF}(u)$ are disabled for every u. Then, since every process has status C, $\neg\mathbf{P\_R1}(u) \wedge \neg\mathbf{P\_R2}(u)$ holds, moreover, $\mathbf{P\_ICorrect}(u)$ implies $\mathbf{P\_Correct}(u)$, so $\mathbf{rule\_R}(u)$ is also disabled in γ. Hence γ is terminal, and we are done. □

4.2 Termination

From Requirements 1 and 2a, we know that Algorithm I does not write into st_u and $\mathbf{P\_ICorrect}(u)$ is closed by I, for every process u. Hence follows.

Remark 1 *For every process u, predicate $\mathbf{P\_Correct}(u)$ (defined in* SDR*) is closed by* I.

Requirements 1, 2b, and 2c ensures the following property.

Lemma 4 *For every process u, predicates $\neg\mathbf{P\_R1}(u)$, $\neg\mathbf{P\_R2}(u)$, and $\mathbf{P\_RB}(u)$ are closed by* I.

Proof. Let $\gamma \mapsto \gamma'$ be any step of I.

- Assume that $\neg\mathbf{P\_R1}(u)$ holds at some process p in γ.

  If $st_u \neq C \vee (\forall v \in N(u) \mid st_v \neq RF)$ in γ, then $st_u \neq C \vee (\forall v \in N(u) \mid st_v \neq RF)$ still holds in γ' by Requirement 1, and we are done.

  Otherwise, $st_u = C \wedge \mathbf{P\_reset}(u) \wedge (\exists v \in N(u) \mid st_v = RF)$ holds in γ. In particular, $\neg\mathbf{P\_Clean}(u)$ holds in γ. Hence, no rule of I is enabled at u in γ, by Requirement 2c. Consequently, $\mathbf{P\_reset}(u)$ still holds in γ'. Since, $\mathbf{P\_reset}(u)$ implies $\neg\mathbf{P\_R1}(u)$, we are done.

- Assume that $\neg\mathbf{P\_R2}(u)$ holds at some process u in γ. If $st_u = C$ holds in γ, then $st_u = C$ holds in γ' by Requirement 1, and so $\neg\mathbf{P\_R2}(u)$ still holds in γ'. Otherwise, $st_u \neq C \wedge \mathbf{P\_reset}(u)$ holds in γ. In particular, $\neg\mathbf{P\_Clean}(u)$ holds in γ. Hence, no rule of I is enabled at u, by Requirement 2c, and by Requirement 2b, $\mathbf{P\_reset}(u)$, and so $\neg\mathbf{P\_R2}(u)$, still holds in γ'.

- By Requirement 1, $\mathbf{P\_RB}(u)$ is closed by I.

□

Recall that two rules are *mutually exclusive* if there is no configuration γ and no process u such that both rules at enabled u in γ. Two algorithms are *mutually exclusive* if their respective rules are pairwise mutually exclusive. Now, whenever a process u is enabled in SDR, $\neg\mathbf{P\_ICorrect}(u) \vee \neg\mathbf{P\_Clean}(u)$ holds and, by Requirement 2c, no rule of I is enabled at u. Hence, follows.

Remark 2 *Algorithms* SDR *and* I *are mutually exclusive.*

Lemma 5 *Rules of Algorithm* SDR *are pairwise mutually exclusive.*

Proof. Since $\mathbf{P\_RB}(u)$ implies $st_u = C$, $\mathbf{P\_RF}(u)$ implies $st_u = RB$, and $\mathbf{P\_C}(u)$ implies $st_u = RF$, we can conclude that $\mathbf{rule\_RB}(u)$, $\mathbf{rule\_RF}(u)$, and $\mathbf{rule\_C}(u)$ are pairwise mutually exclusive.

Then, since $\mathbf{P\_Up}(u)$ implies $\neg\mathbf{P\_RB}(u)$, rules $\mathbf{rule\_R}(u)$ and $\mathbf{rule\_RB}(u)$ are mutually exclusive.

$\mathbf{P\_C}(u)$ implies $\mathbf{P\_Correct}(u) \wedge \mathbf{P\_reset}(u)$ which, in turn, implies $\neg\mathbf{P\_Up}(u)$. Hence, $\mathbf{rule\_R}(u)$ and $\mathbf{rule\_C}(u)$ are mutually exclusive.

$\mathbf{P\_RF}(u)$ implies $st_u = RB \wedge \mathbf{P\_reset}(u)$. Now, $\mathbf{P\_Up}(u)$ implies $st_u = C \vee \neg\mathbf{P\_reset}(u)$. Hence, $\mathbf{rule\_R}(u)$ and $\mathbf{rule\_RF}(u)$ are mutually exclusive. □

Lemma 6 *For every process u, predicates $\neg\mathbf{P\_R1}(u)$ and $\neg\mathbf{P\_R2}(u)$ are closed by* I ∘ SDR.

Proof. By Remark 2 and Lemma 4, to prove this lemma it is sufficient to show that $\neg\mathbf{P\_R1}(u)$ and $\neg\mathbf{P\_R2}(u)$ are closed by SDR, for every process u.

Predicate $\neg\mathbf{P\_R2}(u)$ only depends on variables of u by Requirement 2b. So, if u does not move, $\neg\mathbf{P\_R2}(u)$ still holds. Assume $\neg\mathbf{P\_R2}(u)$ holds in γ and u executes a rule of SDR in $\gamma \mapsto \gamma'$. If u executes $\mathbf{rule\_RB}(u)$ or $\mathbf{rule\_R}(u)$, then u modifies its variables in I by executing $reset(u)$. Hence, in both cases, $\mathbf{P\_reset}(u)$ holds in γ' by Requirement 2e and as $\mathbf{P\_reset}(u)$ implies $\neg\mathbf{P\_R2}(u)$, we are done. Otherwise, u executes $\mathbf{rule\_RF}(u)$ or $\mathbf{rule\_C}(u)$. In both cases, $\mathbf{P\_reset}(u)$ holds in γ and so in γ' by Requirement 2b, and we are done.

Assume now that the predicate $\neg\mathbf{P\_R1}(u)$ holds in γ and consider any step $\gamma \mapsto \gamma'$. Assume first that u moves in $\gamma \mapsto \gamma'$. If u executes $\mathbf{rule\_RB}(u)$, $\mathbf{rule\_RF}(u)$, or $\mathbf{rule\_R}(u)$, then $st_u \neq C$ in γ', hence $\neg\mathbf{P\_R1}(u)$ holds in γ'. If u executes $\mathbf{rule\_C}(u)$ in $\gamma \mapsto \gamma'$, u satisfies $\mathbf{P\_reset}(u)$ in γ, and so in γ' by Requirement 2b. Since $\mathbf{P\_reset}(u)$ implies $\neg\mathbf{P\_R1}(u)$, we are done. Assume now that u does not move in

$\gamma \mapsto \gamma'$. In this case, $\mathbf{P\_R1}(u)$ may become true only if at least a neighbor v of u switches to status RF, by executing $\mathbf{rule\_RF}(v)$. Now, in this case, $st_u \neq C$ in γ, and so in γ'. Consequently, $\neg\mathbf{P\_R1}(u)$ still holds in γ'. □

Theorem 2 *For every process u, $\mathbf{P\_Correct}(u) \vee \mathbf{P\_RB}(u)$ is closed by* I ∘ SDR.

Proof. By Remarks 1 and 2, and Lemma 4, to prove this lemma it is sufficient to show that $\mathbf{P\_Correct}(u) \vee \mathbf{P\_RB}(u)$ is closed by SDR, for every process u.

Let $\gamma \mapsto \gamma'$ be any step of SDR such that $\mathbf{P\_Correct}(u) \vee \mathbf{P\_RB}(u)$ holds in γ.

- Assume $\mathbf{P\_Correct}(u)$ holds in γ. By Requirement 2a, if $\mathbf{P\_ICorrect}(u)$ holds in γ, then $\mathbf{P\_ICorrect}(u)$ still holds in γ', and as $\mathbf{P\_ICorrect}(u)$ implies $\mathbf{P\_Correct}(u)$, we are done.

  Assume now $\neg\mathbf{P\_ICorrect}(u)$ holds in γ. Then, $\mathbf{P\_Correct}(u) \wedge \neg\mathbf{P\_ICorrect}(u)$ implies $st_u \neq C$ in γ. Since $\mathbf{P\_C}(u)$ implies $\mathbf{P\_ICorrect}(u)$ by Requirement 2d, $\mathbf{rule\_C}(u)$ is disabled in γ, and Consequently, $st_u \neq C$ in γ', which implies that $\mathbf{P\_Correct}(u)$ still holds in γ'.

- Assume $\mathbf{P\_RB}(u)$ holds in γ. If u moves in $\gamma \mapsto \gamma'$, then u necessarily executes $\mathbf{rule\_RB}(u)$ (see Lemma 5). In this case, $st_u = RB$ in γ', which implies $\mathbf{P\_Correct}(u)$ in γ'.

  If u does not move, then at least one neighbor of u should switch its status from RB to either C or RF so that $\neg\mathbf{P\_RB}(u)$ holds in γ'. Any neighbor v of u satisfying $st_v = RB$ may only change its status by executing $\mathbf{rule\_RF}(v)$ in $\gamma \mapsto \gamma'$. Now, $\mathbf{rule\_RF}(v)$ is necessarily disabled in γ since $st_u = C$. Hence, $\mathbf{P\_RB}(u)$ still holds in γ' in this case.

□

From Lemma 6 and Theorem 2, we can deduce the following corollary.

Corollary 2 *For every process u, $\neg\mathbf{P\_Up}(u)$ is closed by* I ∘ SDR.

4.2.1 Roots.

If the configuration is illegitimate *w.r.t.* the initial algorithm, then some processes locally detect the inconsistency by checking their state and that of their neighbors (using Predicate $\mathbf{P\_ICorrect}$). Such processes, called here *roots*, should initiates a reset. Then, each root u satisfies $st_u \neq C$ all along the reset processing. According to its status, a root is either *alive* or *dead*, as defined below.

Definition 1 *Let* $\mathbf{P\_root}(u) \equiv st_u = RB \wedge (\forall v \in N(u),\ st_v = RB \Rightarrow d_v \geq d_u)$.

- *A process u is said to be an* alive root *if* $\mathbf{P\_Up}(u) \vee \mathbf{P\_root}(u)$.
- *A process u is said to be an* dead root *if* $st_u = RF \wedge (\forall v \in N(u), status_v \neq C \Rightarrow d_v \geq d_u)$.

By definition, follows.

Remark 3 *For every process u, if $\mathbf{P\_C}(u)$ holds, then u is a dead root.*

The next theorem states that no alive root is created during an execution.

Theorem 3 *For every process u, $\neg\mathbf{P\_root}(u) \wedge \neg\mathbf{P\_Up}(u)$ is closed by* I ∘ SDR.

Proof. By Requirement 1 and Corollary 2, $\neg\mathbf{P\_root}(u) \wedge \neg\mathbf{P\_Up}(u)$ is closed by $\mathtt{I}$. Hence, by Remark 2, it is sufficient to show that $\neg\mathbf{P\_root}(u) \wedge \neg\mathbf{P\_Up}(u)$ is closed by $\mathtt{SDR}$.

Let $\gamma \mapsto \gamma'$ be any step of $\mathtt{SDR}$ such that $\neg\mathbf{P\_root}(u) \wedge \neg\mathbf{P\_Up}(u)$ holds in γ. By Corollary 2, $\neg\mathbf{P\_Up}(u)$ holds in γ'. To show that $\neg\mathbf{P\_root}(u)$ holds in γ', we now consider the following cases:

$st_u = RF$ **in** γ**:** In this case, $\mathbf{rule\_RB}(u)$ and $\mathbf{rule\_R}(u)$ are respectively disabled in γ since $st_u \neq C$ and $\neg\mathbf{P\_Up}(u)$ hold in γ. So, $st_u \neq RB$ in γ', which implies that $\neg\mathbf{P\_root}(u)$ still holds in γ'.

$st_u = RB$ **in** γ**:** Then, $\neg\mathbf{P\_root}(u)$ in γ implies that there is a neighbor v of u such that $st_v = RB \wedge d_v < d_u$ in γ. Let α be the value of d_v in γ. Since $\neg\mathbf{P\_Up}(u)$ holds in γ, u may only execute $\mathbf{rule\_RF}(u)$ in $\gamma \mapsto \gamma'$ and, consequently, $d_u > \alpha$ in γ'. Due to the value of d_u, v may only execute $\mathbf{rule\_R}(v)$ in $\gamma \mapsto \gamma'$. Whether or not v moves, $st_v = RB \wedge d_v \leq \alpha$ in γ'. Hence, $st_v = RB \wedge d_v < d_u$ in γ', which implies that $\neg\mathbf{P\_root}(u)$ still holds in γ'.

$st_u = C$ **in** γ**:** If u does not moves in $\gamma \mapsto \gamma'$, then $\neg\mathbf{P\_root}(u)$ still holds in γ'. Otherwise, since $\neg\mathbf{P\_Up}(u)$ holds in γ, u can only execute $\mathbf{rule\_RB}(u)$ in $\gamma \mapsto \gamma'$. In this case, $\mathbf{P\_RB}(u)$ implies that there is a neighbor v of u such that $st_v = RB$ in γ. Without the loss of generality, assume v is the neighbor of u such that $st_v = RB$ with the minimum distance value in γ. Let α be the value of d_v in γ. Then, $st_u = RB$ and $d_u = \alpha + 1$ in γ'. Moreover, since $st_u = C$ and $st_v = RB$ in γ, v may only execute $\mathbf{rule\_R}(v)$ in $\gamma \mapsto \gamma'$. Whether or not v moves, $st_v = RB \wedge d_v \leq \alpha$ in γ'. Hence, $st_v = RB \wedge d_v < d_u$ in γ', which implies that $\neg\mathbf{P\_root}(u)$ still holds in γ'.

□

4.2.2 Move Complexity.

Definition 2 (AR) *Let γ be a configuration of $\mathtt{I} \circ \mathtt{SDR}$. We denote by $AR(\gamma)$ the set of alive roots in γ.*

By Theorem 3, follows.

Remark 4 *Let $\gamma_0 \cdots \gamma_i \cdots$ be any execution of $\mathtt{I} \circ \mathtt{SDR}$. For every $i > 0$, $AR(\gamma_i) \subseteq AR(\gamma_{i-1})$.*

Based on the aforementioned property, we define below the notion of *segment*.

Definition 3 (Segment) *Let $e = \gamma_0 \cdots \gamma_i \cdots$ be any execution of $\mathtt{I} \circ \mathtt{SDR}$.*

- *If for every $i > 0$, $|AR(\gamma_{i-1})| = |AR(\gamma_i)|$, then the first* segment *of e is e itself, and there is no other segment.*
- *Otherwise, let $\gamma_{i-1} \mapsto \gamma_i$ be the first step of e such that $|AR(\gamma_{i-1})| > |AR(\gamma_i)|$. The* first segment *of e is the prefix $\gamma_0 \cdots \gamma_i$ and the* second segment *of e is the first segment of the suffix of e starting in γ_i, and so forth.*

By Remark 4, follows.

Remark 5 *Every execution of $\mathtt{I} \circ \mathtt{SDR}$ contains at most $n+1$ segments where n is the number of processes.*

We now study how a reset propagates into the network. To that goal, we first define the notion of *reset parent*. Roughly speaking, the parents of u in a reset are its neighbors (if any) that have caused its reset.

Definition 4 (Reset Parent and Children) *$RParent(v,u)$ holds for any two processes u and v if $v \in N(u)$, $st_u \neq C$, $\mathbf{P\_reset}(u)$, $d_u > d_v$, and $(st_u = st_v \vee st_v = RB)$.*

Whenever $RParent(v,u)$ holds, v (resp., u) is said to be a reset parent *of u in (resp., a* reset child *of v).*

Remark that in a given configuration, a process may have several reset parents. Below, we define the *reset branches*, which are the trails of a reset in the network.

Definition 5 (Reset Branch) *A* reset branch *is a sequence of processes $u_1, \ldots, u_k$ for some integer $k \geq 1$, such that u_1 is an alive or dead root and, for every $1 < i \leq k$, we have $RParent(u_{i-1}, u_i)$. The process u_i is said to be at* depth $i-1$ *and $u_i, \cdots, u_k$ is called a* reset sub-branch*. The process u_1 is the* initial extremity *of the reset branch $u_1, \ldots, u_k$.*

Lemma 7 *Let $u_1, \ldots, u_k$ be any reset branch.*

1. *$k \leq n$,*
2. *If $st_{u_1} = C$, then $k = 1$. Otherwise, $st_{u_1} \cdots st_{u_k} \in RB^*RF^*$.*
3. *$\forall i \in \{2, \ldots, k\}$, u_i is neither an alive, not a dead root.*

Proof. Let i and j such that $1 \leq i < j \leq k$. By definition, $d_{u_i} < d_{u_j}$ and so $u_i \neq u_j$. Hence, in a reset branch, each node appears at most once, and Lemma 7.1 holds.

Let $i \in \{2, \ldots, k\}$. Lemma 7.2 immediately follows from the following three facts, which directly derive from the definition of reset parent.

- $st_{u_i} \neq C$.
- $st_{u_i} = RB \Rightarrow st_{u_{i-1}} = RB$.
- $st_{u_i} = RF \Rightarrow st_{u_{i-1}} \in \{RB, RF\}$.

Lemma 7.3 immediately follows from those two facts.

- *u_i is not a dead root*, since $u_{i-1} \in N(u_i) \wedge st_{u_{i-1}} \neq C \wedge d_{u_{i-1}} < d_{u_i}$.
- *u_i is not an alive root*, indeed
  - *$\neg\mathbf{P\_root}(u)$ holds*, since $u_{i-1} \in N(u_i) \wedge (st_{u_i} = RB \Rightarrow st_{u_{i-1}} = RB) \wedge d_{u_{i-1}} < d_{u_i}$.
  - *$\neg\mathbf{P\_Up}(u_i)$ holds* since $\neg\mathbf{P\_R1}(u) \wedge \neg\mathbf{P\_R2}(u)$ holds because $\mathbf{P\_reset}(u)$ holds, and $\mathbf{P\_Correct}(u)$ holds because $st_u \neq C$.

□

Remark 6 *In a configuration, a process u may belong to several branches. Precisely, u belongs to at least one reset branch, unless $st_u = C \wedge \mathbf{P\_ICorrect}(u)$ holds.*

Lemma 8 *Let $\gamma_x \mapsto \gamma_{x+1}$ be a step of $\mathtt{I} \circ \mathtt{SDR}$. Let $u_1, \ldots, u_k$ be a reset branch in γ_x. If u_1 is an alive root in γ_{x+1}, then $u_1, \ldots, u_k$ is a reset branch in γ_{x+1}.*

Proof. We first show the following two claims:

Claim 1: *If u_1 moves in $\gamma_x \mapsto \gamma_{x+1}$, then u_1 necessarily executes* $\mathbf{rule\_R}$ *in $\gamma_x \mapsto \gamma_{x+1}$.*

Proof of the claim: Since u_1 is an alive root in γ_{x+1}, u_1 is an alive root in γ_x, by Theorem 3. Assume now, by the contradiction, that u_1 moves, but does not execute $\mathbf{rule\_R}$ in $\gamma_x \mapsto \gamma_{x+1}$. Then, $\neg\mathbf{P\_Up}(u_k)$ holds in γ_x, by Remark 2 and Lemma 5. So, by definition of alive root, $st_u = RB$ in γ_x and, from the code of SDR and Requirement 2c, u_1 executes $\mathbf{rule\_RF}$ in $\gamma_x \mapsto \gamma_{x+1}$. Consequently, $st_u = RF$. Now, $\neg\mathbf{P\_Up}(u_k)$ still holds in γ_{x+1}, by Corollary 2. Hence, u_1 is not an alive root in γ_{x+1}, a contradiction.

Claim 2: *For every $i \in \{2, \ldots, k\}$, if u_i moves $\gamma_x \mapsto \gamma_{x+1}$, then u_i executes* $\mathbf{rule\_RF}$ *in $\gamma_x \mapsto \gamma_{x+1}$ and in γ_x we have $st_{u_i} = RB$ and $i < k \Rightarrow st_{u_{i+1}} = RF$.*

Proof of the claim: We first show that only $\mathbf{rule\_RF}$ may be enabled at u_i in γ_x.

- By definition, $st_{u_i} \neq C$ and so $\neg\mathbf{P\_Clean}(u_i)$ holds in γ_x. Thus, by Requirement 2c, all rules of I that are disabled at u_i in γ_x.
- $\mathbf{rule\_RB}(u_i)$ is disabled in γ_x since $st_{u_i} \neq C$ (by definition).
- The fact that u_i is not a dead root in γ_x (Lemma 7) implies that $\mathbf{rule\_C}(u_i)$ is disabled in γ_x (Remark 3).
- $\mathbf{rule\_R}(u_i)$ is disabled in γ_x since u_i is not an alive root (Lemma 7).

Hence, u_i can only executes $\mathbf{rule\_RF}$ in $\gamma_x \mapsto \gamma_{x+1}$. In this case, $st_{u_i} = RB$ in γ_x (see the guard of $\mathbf{rule\_RF}(u_i)$). Moreover, if $i < k$, then $st_{u_i} = RB \wedge RParent(u_i, u_{i+1})$ implies that $u_{i+1} \in N(u_i)$, $st_{u_{i+1}} \in \{RB, RF\}$, and $d_{u_i} < d_{u_{i+1}}$. Now, if $u_{i+1} \in N(u_i)$, $st_{u_{i+1}} = RB$, and $d_{u_i} < d_{u_{i+1}}$, then $\mathbf{rule\_RF}(u_i)$ is disabled. Hence, if u_i moves $\gamma_x \mapsto \gamma_{x+1}$ and $i < k$, then u_i executes $\mathbf{rule\_RF}$ and so $st_{u_{i+1}} = RF$ in γ_x.

Then, we proceed by induction on k. The base case ($k = 1$) is trivial. Assume now that $k > 1$. Then, by induction hypothesis, $u_1, \ldots, u_{k-1}$ is a reset branch in γ_{x+1}. Hence, to show that $u_1, \ldots, u_k$ is a reset branch in γ_{x+1}, it is sufficient to show that $RParent(u_{k-1}, u_k)$ holds in γ_{x+1}. Since, $RParent(u_{k-1}, u_k)$ holds in γ_x, we have $st_{u_k} \neq C$ in γ_x. So, we now study the following two cases:

- $st_{u_k} = RB$ in γ_x. By Claim 2, u_k may only execute $\mathbf{rule\_RF}$ in $\gamma_x \mapsto \gamma_{x+1}$. Consequently, $st_{u_k} \in \{RB, RF\} \wedge \mathbf{P\_reset}(u_k) \wedge d_{u_k} = d$ holds in γ_{x+1}, where $d > 0$ is the value of d_{u_k} in γ_x. Consider now process u_{k-1}. Since $st_{u_k} = RB$ in γ_x, $st_{u_{k-1}} = RB$ too in γ_x (Lemma 7).

  If u_{k-1} does not move in $\gamma_x \mapsto \gamma_{x+1}$, then $RParent(u_{k-1}, u_k)$ still holds in γ_{x+1}, and we are done.

  Assume now that u_{k-1} moves in $\gamma_x \mapsto \gamma_{x+1}$. Then, we necessarily have $k = 2$ since otherwise, Claim 2 applies for $i = k - 1$: u_{k-1} moves in $\gamma_x \mapsto \gamma_{x+1}$ only if $st_{u_k} = RF$ in γ_x. Now, $k = 2$ implies that u_{k-1} executes $\mathbf{rule\_R}$ in $\gamma_x \mapsto \gamma_{x+1}$ (by Claim 1), so $st_{u_{k-1}} = RB$ and $d_{u_{k-1}} = 0 < d$ in γ_{x+1}. Consequently, $RParent(u_{k-1}, u_k)$ still holds in γ_{x+1}, and we are done.

- $st_{u_k} = RF$ in γ_x.

  By Claim 2, u_k does not move in $\gamma_x \mapsto \gamma_{x+1}$. So, $st_{u_k} = RF \wedge \mathbf{P\_reset}(u_k) \wedge d_{u_k} = d$ holds in γ_{x+1}, where $d > 0$ is the value of d_{u_k} in γ_x.

  If u_{k-1} does not move in $\gamma_x \mapsto \gamma_{x+1}$, we are done. Assume, otherwise, that u_{k-1} moves in $\gamma_x \mapsto \gamma_{x+1}$.

Then, if $k = 2$, then u_{k-1} executes $\mathbf{rule\_R}$ in $\gamma_x \mapsto \gamma_{x+1}$ (by Claim 1), so $st_{u_{k-1}} = RB$ and $d_{u_{k-1}} = 0 < d$ in γ_{x+1}, and so $RParent(u_{k-1}, u_k)$ still holds in γ_{x+1}.

Otherwise ($k > 2$), u_{k-1} necessarily executes $\mathbf{rule\_RF}$ in $\gamma_x \mapsto \gamma_{x+1}$ (by Claim 2): $st_{u_{k-1}} = RF$ and $d_{u_{k-1}} < d_{u_k}$ in γ_{x+1} (*n.b.*, neither $d_{u_{k-1}}$ nor d_{u_k} is modified in $\gamma_x \mapsto \gamma_{x+1}$). Hence, $RParent(u_{k-1}, u_k)$ still holds in γ_{x+1}, and we are done.

□

Lemma 9 *Let u be any process. During a segment $S = \gamma_i \cdots \gamma_j$ of execution of* I ∘ SDR*, if u executes the rule* $\mathbf{rule\_RF}$*, then u does not execute any other rule of* SDR *in the remaining of S.*

Proof. Let $\gamma_x \mapsto \gamma_{x+1}$ be a step of S in which u executes $\mathbf{rule\_RF}$. Let $\gamma_y \mapsto \gamma_{y+1}$ (with $y > x$) be the next step in which u executes its next rule of SDR. (If $\gamma_x \mapsto \gamma_{x+1}$ or $\gamma_y \mapsto \gamma_{y+1}$ does not exist, then the lemma trivially holds.) Then, since $\mathbf{rule\_RF}(u)$ is enabled in γ_x, $\neg\mathbf{P\_Up}(u)$ holds in γ_x, by Lemma 5. Consequently, $\neg\mathbf{P\_Up}(u)$ holds forever from γ_x, by Corollary 2. Hence, from the code of SDR and Requirement 2c, u necessarily executes $\mathbf{rule\_C}$ in $\gamma_y \mapsto \gamma_{y+1}$ since $st_u = RF \wedge \neg\mathbf{P\_Up}(u)$ holds in γ_y. In γ_x, since $st_u = RB$, u belongs to some reset branches (Remark 6) and all reset branches containing u have an alive root (maybe u) of status RB (Lemma 7). Let v be any alive root belonging to a reset branch containing u in γ_x. In γ_y, u is the dead root, since $\mathbf{P\_C}(u)$ holds (Remark 3). By Lemma 7, either $u = v$ or u no more belong to a reset branch whose initial extremity is v. By Lemma 8 and Theorem 3, v is no more an alive root in γ_y. Still by Theorem 3, the number of alive roots necessarily decreased between γ_x and γ_y: $\gamma_x \mapsto \gamma_{x+1}$ and $\gamma_y \mapsto \gamma_{y+1}$ belong to two distinct segments of the execution. □

Theorem 4 *The sequence of rules of* SDR *executed by a process u in a segment of execution of* I ∘ SDR *belongs to the following language:*

$$(\mathbf{rule\_C} + \varepsilon)\ (\mathbf{rule\_RB} + \mathbf{rule\_R} + \varepsilon)\ (\mathbf{rule\_RF} + \varepsilon)$$

Proof. From the code of SDR and Requirement 2c, we know that after any execution of $\mathbf{rule\_C}(u)$, the next rule of SDR u will execute (if any), is either $\mathbf{rule\_RB}$ or $\mathbf{rule\_R}$. Similarly, immediately after an execution of $\mathbf{rule\_RB}(u)$ (resp., $\mathbf{rule\_R}(u)$), $st_u = RB \wedge \mathbf{P\_reset}(u)$ holds (see Requirement 2e) and $\mathbf{P\_reset}(u)$ holds while u does not switch to status C (Requirements 2b and 2c). So the next rule of SDR u will execute (if any) is $\mathbf{rule\_RF}$. Finally, immediately after any execution of $\mathbf{rule\_RF}(u)$, $st_u = RF \wedge \mathbf{P\_reset}(u)$ holds until (at least) the next execution of a rule of SDR since $\mathbf{P\_reset}(u)$ holds while u does not switch to status C (Requirements 2b and 2c). Then, the next rule of SDR u will execute (if any) is $\mathbf{rule\_C}$. However, if this latter case happens, $\mathbf{rule\_RF}(u)$ and $\mathbf{rule\_C}(u)$ are executed in different segments, by Lemma 9. □

Since a process can execute rules of I only if its status is C, we have the following corollary.

Corollary 3 *The sequence of rules executed by a process u in a segment of execution of* I ∘ SDR *belongs to the following language:*

$$(\mathbf{rule\_C} + \varepsilon)\ \mathbf{words\_I}\ (\mathbf{rule\_RB} + \mathbf{rule\_R} + \varepsilon)\ (\mathbf{rule\_RF} + \varepsilon)$$

where $\mathbf{words\_I}$ *is any sequence of rules of* I.

From Remark 5 and Theorem 4, follows.

Corollary 4 *Any process u executes at most $3n + 3$ rules of* SDR *in any execution of* I $\circ$ SDR.

Let $S = \gamma_0 \cdots \gamma_j \cdots$ be a segment of execution of I $\circ$ SDR. Let $c_{\mathtt{I}}^S$ be the configuration of I such that for every process u,

1. $c_{\mathtt{I}}^S(u) = \gamma_{0|\mathtt{I}}(u)$ if u never satisfies $st_u = C$ in S,
2. $c_{\mathtt{I}}^S(u) = \gamma_{i|\mathtt{I}}(u)$ where γ_i is the first configuration such that $st_u = C$ in S otherwise.

The following lemma is a useful tool to show the convergence of I $\circ$ SDR.

Lemma 10 *Let $S = \gamma_0 \cdots \gamma_j \cdots$ be a segment of execution of* I $\circ$ SDR. *For every process u, let $\mathbf{words\_I}(u)$ be the (maybe empty) sequence of rules of* I *executed by u in S. There is a prefix of execution of* I *consisting in the executions of $\mathbf{words\_I}(u)$, for every process u.*

Proof. For every process u,

1. either u never satisfies $st_u = C$ in S and $c_{\mathtt{I}}^S(u) = \gamma_{0|\mathtt{I}}(u)$,
2. or $c_{\mathtt{I}}^S(u) = \gamma_{i|\mathtt{I}}(u)$ where γ_i is the first configuration such that $st_u = C$ in S.

In the former case, $\mathbf{words\_I}(v)$ is empty, for every $v \in N[u]$ (including u) since $\mathbf{P\_Clean}(v)$ never holds in S (see Requirement 2c).

In the latter case, if $\mathbf{words\_I}(u)$ is not empty, then the next rule u will execute from γ_i is the first rule of $\mathbf{words\_I}(u)$, by Corollary 3. Moreover, no neighbor v of u can start executing $\mathbf{words\_I}(v)$ before γ_i (see Requirement 2c). Then, the first rule of SDR executed by u in S after any rule of $\mathbf{words\_I}(u)$ (if any) is either $\mathbf{rule\_RB}(u)$ or $\mathbf{rule\_R}(u)$, by Corollary 3. After executing such a rule, $st_u \neq C$ until (at least) the end of S, by Corollary 3. Hence, from that point, every member x of the closed neighborhood of u (including u) satisfies $\neg\mathbf{P\_Clean}(x)$, and so is disabled *w.r.t.* I, until (at least) the end of S, by Requirement 2c, meaning that the execution of $\mathbf{words\_I}(x)$ is done.

Hence, keeping the relative order on execution of rules of I in S, we can build a possible prefix of execution of I that starts from $c_{\mathtt{I}}^S$ where every process u exactly executes the sequence $\mathbf{words\_I}(u)$, and we are done. □

4.2.3 Round Complexity.

Below, we use the notion of attractor, defined at the beginning of the section.

Definition 6 (Attractors)

- *Let $\mathcal{P}_1$ a predicate over configurations of* I $\circ$ SDR *which is true if and only if $\neg\mathbf{P\_Up}(u)$ holds, for every process u.*
- *Let $\mathcal{P}_2$ a predicate over configurations of* I $\circ$ SDR *which is true if and only if (1) $\mathcal{P}_1$ holds and (2) $\neg\mathbf{P\_RB}(u)$ holds, for every process u.*
- *Let $\mathcal{P}_3$ a predicate over configurations of* I $\circ$ SDR *which is true if and only if (1) $\mathcal{P}_2$ holds and (2) $st_u \neq RB$, for every process u.*

- $\mathcal{P}_4$ *a predicate over configurations of* I ∘ SDR *which is true if and only if (1)* $\mathcal{P}_3$ *holds and (2)* $st_u \neq RF$*, for every process* u*.*

  In the following, we call normal configuration *any configuration satisfying* $\mathcal{P}_4$*.*

Lemma 11 $\mathcal{P}_1$ *is an attractor for* I ∘ SDR*. Moreover,* I ∘ SDR *converges from* $true$ *to* $\mathcal{P}_1$ *within at most one round.*

Proof. First, for every process u, $\neg\mathbf{P}\_\mathbf{Up}(u)$ is closed by I ∘ SDR (Corollary 2). Consequently, $\mathcal{P}_1$ is closed by I ∘ SDR. Moreover, to show that I ∘ SDR converges from $true$ to $\mathcal{P}_1$ within at most one round, it is sufficient to show that any process p satisfies $\neg\mathbf{P}\_\mathbf{Up}(u)$ during the first round of any execution of I ∘ SDR. This property is immediate from the following two claims.

Claim 1: *If* $\mathbf{P}\_\mathbf{Up}(u)$*, then* u *is enabled.*

Proof of the claim: By definition of $\mathbf{rule\_R}(u)$.

Claim 2: *If* $\mathbf{P}\_\mathbf{Up}(u)$ *holds in* γ *and* u *moves in the next step* $\gamma \mapsto \gamma'$*, then* $\neg\mathbf{P}\_\mathbf{Up}(u)$ *holds in* γ'*.*

Proof of the claim: First, by Remark 2, Lemma 5, and the guard of $\mathbf{rule\_R}(u)$, $\mathbf{rule\_R}(u)$ is executed in $\gamma \mapsto \gamma'$. Then, immediately after $\mathbf{rule\_R}(u)$, $st_u = RB$ and $\mathbf{P\_reset}(u)$ holds (see Requirement 2e), now $st_u = RB$ and $\mathbf{P\_reset}(u)$ implies $\neg\mathbf{P}\_\mathbf{Up}(u)$.

□

Lemma 12 $\mathcal{P}_2$ *is closed by* I ∘ SDR*.*

Proof. By Requirement 1, $\neg\mathbf{P\_RB}(u)$ is closed by I, for every process u. So, by Lemma 11 and Remark 2, it is sufficient to show that for every step $\gamma \mapsto \gamma'$ of I ∘ SDR such that $\mathcal{P}_2$ holds in γ, for every process u, if u executes a rule of SDR in $\gamma \mapsto \gamma'$, then $\neg\mathbf{P\_RB}(u)$ still holds in γ'.

So, assume any such step $\gamma \mapsto \gamma'$ and any process u.

- If $st_u = C$ in γ, then $\forall v \in N(u)$, $st_v \neq RB$ in γ, since γ satisfies $\mathcal{P}_2$. Now, no rule $\mathbf{rule\_RB}$ or $\mathbf{rule\_R}$ can be executed in $\gamma \mapsto \gamma'$ since γ satisfies $\mathcal{P}_2$. So, $\forall v \in N(u)$, $st_v \neq RB$ in γ', and consequently $\neg\mathbf{P\_RB}(u)$ still holds in γ'.
- If $st_u \neq C$ in γ', then $\neg\mathbf{P\_RB}(u)$ holds in γ'.
- Assume now that $st_u \neq C$ in γ and $st_u = C$ in γ'. Then, u necessarily executes $\mathbf{rule\_C}$ in $\gamma \mapsto \gamma'$. In this case, $\forall v \in N(u)$, $st_v \neq RB$ in γ. Now, no rule $\mathbf{rule\_RB}$ or $\mathbf{rule\_R}$ can be executed in $\gamma \mapsto \gamma'$ since γ satisfies $\mathcal{P}_2$. So, $\forall v \in N(u)$, $st_v \neq RB$ in γ', and consequently $\neg\mathbf{P\_RB}(u)$ still holds in γ'.

Hence, in all cases, $\neg\mathbf{P\_RB}(u)$ still holds in γ', and we are done. □

Lemma 13 I ∘ SDR *converges from* $\mathcal{P}_1$ *to* $\mathcal{P}_2$ *within at most* $n-1$ *rounds.*

Proof. Let u be any process of status RB. Then, u belongs to at least one reset branch (Remark 6). Let $md(u)$ be the maximum depth of u in a reset branch it belongs to. Then, $md(u) < n$, by Lemma 7.

Consider now any execution $e = \gamma_0 \cdots \gamma_i \cdots$ of I ∘ SDR such that γ_0 satisfies $\mathcal{P}_1$. Remark first that from γ_0, $\mathbf{rule\_R}(v)$ is disabled forever, for every process v, since $\mathcal{P}_1$ is closed by I ∘ SDR (Lemma 11).

Claim 1: *If some process u satisfies $st_u = RB$ in some configuration γ_i ($i \geq 0$), then from γ_i, while $st_u = RB$, $md(u)$ cannot decrease.*

Proof of the claim: Consider any $\gamma_i \mapsto \gamma_{i+1}$ where $st_u = RB$ both in γ_i and γ_{i+1}. This in particular means that u does not move in $\gamma_i \mapsto \gamma_{i+1}$. Let $u_1, \ldots, u_k = u$ be a reset branch in γ_i such that $k = md(u)$. $\forall x \in \{1, \ldots, k-1\}$, u_x has a neighbor (u_{x+1}) such that $st_{x+1} = RB \wedge d_{x+1} > d_u$ in γ_i, by Lemma 7 and the definition of a reset branch. Hence, every u_x is disabled in γ_i. Consequently, u is still at depth at least k in a reset branch defined in γ_{i+1}, and we are done.

Claim 2: *For every process u that executes $\mathbf{rule\_RB}(u)$ in some step $\gamma_i \mapsto \gamma_{i+1}$ of the jth round of e, we have $st_u = RB \wedge md(u) \geq j$ in γ_{i+1}.*

Proof of the claim: We proceed by induction. Assume a process u executes $\mathbf{rule\_RB}(u)$ in some step $\gamma_i \mapsto \gamma_{i+1}$ of the first round of e. In γ_i, there is some neighbor v of u such that $st_v = RB$. Since $st_u = C$ in γ_i, v is disabled in γ_i. Consequently, $RParent(v, u)$ holds in γ_{i+1}, and so $st_u = RB \wedge md(u) \geq 1$ holds in γ_{i+1}. Hence, the claim holds for $j = 1$.

Assume now that the claim holds in all of the j first rounds of e, with $j \geq 1$.

Assume, by the contradiction, that some process u executes $\mathbf{rule\_RB}$ in a step $\gamma_i \mapsto \gamma_{i+1}$ of the $(j+1)^{th}$ round of e, and does not satisfy $st_u = RB \wedge md(u) \geq j+1$ in γ_{i+1}. Then, by definition of $\mathbf{rule\_RB}(u)$, $st_u = C$ in γ_i and $st_u = RB \wedge md(u) < j+1$ in γ_{i+1}. Let x be the value of $md(u)$ in γ_{i+1}. We have $x < j+1$. Without the loss of generality, assume that no process satisfies this condition before u in the $(j+1)^{th}$ round of e and any process v that fulfills this condition in the same step as u satisfies $st_v = RB \wedge x \leq md(v) < j+1$ in γ_{i+1}. Then, by definition $md(u)$ and Lemma 7, there is a neighbor v of u such $st_v = RB$ and $md(v) = x - 1 < j$ in γ_{i+1}. Moreover, by definition of u and Claim 1, $st_v = RB$ and $md(v) \leq x - 1 < j$ since (at least) the first configuration of the $(j+1)$th round of e, which is also the last configuration of the jth round of e. So, by induction hypothesis, $st_v = RB$ and $md(v) \leq x - 1 < j$ since (at least) the end of the $(x-1)$th round of e. If $st_u \neq C$ in the last configuration of the $(x-1)$th round of e, then $st_u \neq C$ continuously until γ_i (included) since meanwhile $\mathbf{rule\_C}(u)$ is disabled because $st_v = RB$. Hence, u cannot execute $\mathbf{rule\_RB}(u)$ in $\gamma_i \mapsto \gamma_{i+1}$, a contradiction. Assume otherwise that $st_u = C$ in the last configuration of the $(x-1)$th round of e. Then, u necessarily executes $\mathbf{rule\_RB}$ during the xth round of e, but not in the $(j+1)^{th}$ round of e since $st_v = RB$ continuously until γ_i (included), indeed, after the execution of $\mathbf{rule\_RB}(u)$ in the xth round, the two next rules executed by u (if any) are necessarily $\mathbf{rule\_RF}$ followed by $\mathbf{rule\_C}$, but $\mathbf{rule\_C}(u)$ is disabled while $st_v = RB$. Hence, $\mathbf{rule\_RB}(u)$ is not executed in $\gamma_i \mapsto \gamma_{i+1}$, a contradiction.

By Claim 2, no process executes $\mathbf{rule\_RB}$ during the nth round of e. Now, along e, we have:

- If $\mathbf{P\_RB}(u)$ holds, then u is enabled (see $\mathbf{rule\_RB}(u)$), and
- If $\mathbf{P\_RB}(u)$ holds in γ_i (with $i \geq 0$) and u moves in the next step $\gamma_i \mapsto \gamma_{i+1}$, then $\neg\mathbf{P\_RB}(u)$ holds in γ_{i+1}.

  Indeed, u necessarily executes $\mathbf{rule\_RB}(u)$ in $\gamma_i \mapsto \gamma_{i+1}$ (Remark 2 and Lemma 5) and, consequently, $st_u = RB$ in γ_{i+1}, which implies that $\neg\mathbf{P\_RB}(u)$ holds in γ_{i+1}.

Hence, we can conclude that the last configuration of the $(n-1)$th round of e satisfies $\mathcal{P}_2$, and we are done. □

Lemma 14 *$\mathcal{P}_3$ is closed by* $\mathtt{I} \circ \mathtt{SDR}$*. Moreover,* $\mathtt{I} \circ \mathtt{SDR}$ *converges from $\mathcal{P}_2$ to $\mathcal{P}_3$ within at most n rounds.*

Proof. By Requirement 1, no rule of $\mathtt{I}$ can set the status of a process to RB. Then, let γ be a configuration of $\mathtt{I} \circ \mathtt{SDR}$ such that $\mathcal{P}_3(\gamma)$ holds. Since $\mathcal{P}_1(\gamma)$ and $\mathcal{P}_2(\gamma)$ also holds, no rule $\mathbf{rule\_R}$ or $\mathbf{rule\_RB}$ is enabled in γ. Hence, after any step from γ, there is still no process of status RB and we can conclude that $\mathcal{P}_3$ is closed by $\mathtt{I} \circ \mathtt{SDR}$ since we already know that $\mathcal{P}_2$ is closed by $\mathtt{I} \circ \mathtt{SDR}$ (Lemma 12).

Let γ be any configuration satisfying $\mathcal{P}_2$ but not $\mathcal{P}_3$. To show the convergence from from $\mathcal{P}_2$ to $\mathcal{P}_3$ within at most n rounds, it is sufficient to show that at least one process u switches from $st_u = RB$ to $st_u \neq RB$ within the next round from γ, since we already know that once $st_u \neq RB$ after γ, $st_u \neq RB$ holds forever (recall that all configurations reached from γ satisfies $\mathcal{P}_2$, see Lemma 12).

Let mu be a process of status RB with a maximum distance value in γ. Since $\neg\mathbf{P\_Up}(mu) \wedge \neg\mathbf{P\_RB}(mu) \wedge st_{mu} \neq C$ holds, $\mathbf{P\_reset}(mu)$ holds in γ. Let v be any neighbor of mu. Since $\neg\mathbf{P\_RB}(v) \wedge st_{mu} = RB$ holds, $st_v \neq C$ in γ. Again, since $\neg\mathbf{P\_Up}(v) \wedge \neg\mathbf{P\_RB}(v) \wedge st_v \neq C$ holds, $\mathbf{P\_reset}(v)$ holds in γ. According to the definition of mu, we have $(st_v = RB \wedge d_v \leq d_{mu}) \vee st_v = RF$. We can conclude that along any execution from γ, $\mathbf{rule\_RF}(mu)$ is enabled until mu executes this rule. In this case, $\mathbf{rule\_RF}(mu)$ will be executed in the next move of mu by Remark 2 and Lemma 5. So, during the next round from γ, $\mathbf{rule\_RF}(mu)$ is executed, *i.e.*, st_{mu} is set to RF, and we are done. □

Let γ by any configuration of $\mathtt{I} \circ \mathtt{SDR}$. We denote by $\gamma_{|\mathtt{SDR}}$ the projection of γ over variables of $\mathtt{SDR}$. By definition, $\gamma_{|\mathtt{SDR}}$ is a configuration of $\mathtt{SDR}$.

Lemma 15 *For every configuration γ of* $\mathtt{I} \circ \mathtt{SDR}$*, $\gamma \in \mathcal{P}_4$ (*i.e.*, γ is a normal configuration) if and only if $\gamma_{|\mathtt{SDR}}$ is a terminal configuration of* $\mathtt{SDR}$*.*

Proof. Let γ be a configuration of $\mathtt{I} \circ \mathtt{SDR}$. By definition of $\mathcal{P}_4$, if $\gamma \in \mathcal{P}_4$, then $\gamma_{|\mathtt{SDR}}$ is a terminal configuration of $\mathtt{SDR}$. Then, if $\gamma_{|\mathtt{SDR}}$ is a terminal configuration of $\mathtt{SDR}$, then $\gamma \in \mathcal{P}_4$ by Lemmas 1, 2, and 3. □

Lemma 16 *$\mathcal{P}_4$ is closed by* $\mathtt{I} \circ \mathtt{SDR}$*. Moreover,* $\mathtt{I} \circ \mathtt{SDR}$ *converges from $\mathcal{P}_3$ to $\mathcal{P}_4$ within at most n rounds.*

Proof. By Requirement 1, no rule of $\mathtt{I}$ can set the status of a process to RF. So, by Lemma 15, we can conclude that $\mathcal{P}_4$ is closed by $\mathtt{I} \circ \mathtt{SDR}$.

Let γ be any configuration satisfying $\mathcal{P}_3$ but not $\mathcal{P}_4$. To show the convergence from from $\mathcal{P}_3$ to $\mathcal{P}_4$ within at most n rounds, it is sufficient to show that at least one process u switches from $st_u = RF$ to $st_u \neq RF$ within the next round from γ, since we already know that once $st_u \neq RF$ after γ, $st_u \neq RF$ holds forever (recall that all configurations reached from γ satisfies $\mathcal{P}_3$ by Lemma 14, and only process of status EB may switch to status EF).

Let mu be a process of status RF with a minimum distance value in γ. Since $\neg\mathbf{P\_Up}(mu) \wedge \neg\mathbf{P\_RB}(mu) \wedge st_{mu} \neq C$ holds, $\mathbf{P\_reset}(mu)$ holds in γ. Let v be any neighbor of mu. By definition of $\mathcal{P}_3$, $st_v \neq RB$ in γ. Moreover, since $\neg\mathbf{P\_Up}(v) \wedge \neg\mathbf{P\_RB}(v)$ and v has a neighbor of status EF (mu), $\mathbf{P\_reset}(v)$ holds in γ. According to the definition of mu, we have $(st_v = RF \wedge d_v \geq d_{mu}) \vee st_v = C$. We can conclude that along any execution from γ, $\mathbf{rule\_C}(mu)$ is enabled until mu executes this rule. In this case, $\mathbf{rule\_C}(mu)$ will be executed in the next move of mu by Remark 2 and Lemma 5. So, during the next round from γ, $\mathbf{rule\_C}(mu)$ is executed, *i.e.*, st_{mu} is set to C, and we are done. □

By Lemmas 11-16 and Theorem 1, follows.

Corollary 5 *$\mathcal{A}_4$ is an attractor for* $\mathtt{I} \circ \mathtt{SDR}$*. Moreover,* $\mathtt{I} \circ \mathtt{SDR}$ *converges from $true$ to $\mathcal{P}_4$ within at most $3n$ rounds. For every configuration γ of* $\mathtt{I} \circ \mathtt{SDR}$*, γ satisfies $\mathcal{A}_4$ (*i.e. *γ is a normal configuration) if and only if $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds in γ, for every process u.*

5 (f, g)-alliance

5.1 The Problem

The (f, g)-alliance problem has been defined by Dourado *et al.* [20]. Given a graph $G = (V, E)$, and two non-negative integer-valued functions on nodes f and g, a subset of nodes $A \subseteq V$ is an (f, g)-*alliance* of G if and only if every node $u \notin A$ has at least $f(u)$ neighbors in A, and every node $v \in A$ has at least $g(v)$ neighbors in A. The (f, g)-alliance problem is the problem of finding a subset of processes forming an (f, g)-*alliance* of the network. The (f, g)-*alliance* problem is a generalization of several problems that are of interest in distributed computing. Consider any subset S of processes/nodes:

1. S is a domination set [8] if and only if S is a $(1, 0)$-alliance;
2. more generally, S is a k-domination set [8] if and only if S is a $(k, 0)$-alliance;
3. S is a k-tuple dominating set [28] if and only if S is a $(k, k-1)$-alliance;
4. S is a global offensive alliance [29] if and only if S is a $(f, 0)$-alliance, where $f(u) = \lceil \frac{\delta_u+1}{2} \rceil$ for all u;
5. S is a global defensive alliance [30] if and only if S is a $(1, g)$-alliance, where $g(u) = \lceil \frac{\delta_u+1}{2} \rceil$ for all u;
6. S is a global powerful alliance [35] if and only if S is a (f, g)-alliance, such that $f(u) = \lceil \frac{\delta_u+1}{2} \rceil$ and $g(u) = \lceil \frac{\delta_u}{2} \rceil$ for all u.

We remark that (f, g)-alliances have applications in the fields of population protocols [3] and server allocation in computer networks [23].

Ideally, we would like to find a *minimum* (f, g)-alliance, namely an (f, g)-alliance of the smallest possible cardinality. However, this problem is $\mathcal{NP}$-hard, since the $(1, 0)$-alliance (*i.e.*, the domination set problem) is known to be $\mathcal{NP}$-hard [21]. We can instead consider the problem of finding a *minimal* (f, g)-alliance. An (f, g)-*alliance* is *minimal* if no proper subset of A is an (f, g)-*alliance*. Another variant is the 1-*minimal* (f, g)-*alliance*. A is a 1-*minimal* (f, g)-*alliance* if deletion of just one member of A causes A to be no more an (f, g)-*alliance*, *i.e.*, A is an (f, g)-*alliance* but $\forall u \in A$, $A \setminus \{u\}$ is not an (f, g)-*alliance*. Surprisingly, a 1-*minimal* (f, g)-*alliance* is not necessarily a *minimal* (f, g)-*alliance* [20]. However, we have the following property:

Property 1 ([20]) *Given two non-negative integer-valued functions f and g on nodes*

1. *Every minimal* (f, g)-alliance *is a* 1-*minimal* (f, g)-alliance*, and*
2. *if $f(u) \geq g(u)$ for every process u, then every* 1-*minimal* (f, g)-alliance *is a minimal* (f, g)-alliance*.*

5.2 Contribution

We first propose a distributed algorithm called FGA. Starting from a pre-defined configuration, FGA computes a 1-minimal (f, g)-alliance in any identified network where $\delta_u \geq max(f(u), g(u))$, for every process u. Notice that this latter assumption ensures the existence of a solution. FGA is not self-stabilizing, however we show that the composite algorithm FGA $\circ$ SDR is actually an efficient self-stabilizing 1-minimal (f, g)-alliance algorithm.

5.3 Related Work

Recall that the (f, g)-*alliance* problem has been introduced by Dourado *et al.* [20]. In that paper, the authors give several distributed algorithms for that problem and its variants, but none of them is self-stabilizing.

In [10], Carrier *et al.* proposes a silent self-stabilizing algorithm that computes a minimal (f, g)-*alliance* in an asynchronous network with unique node IDs, assuming that every node u has a degree at least $g(u)$ and satisfies $f(u) \geq g(u)$. Their algorithm is also *safely converging* in the sense that starting from any configuration, it first converges to a (not necessarily minimal) (f, g)-*alliance* in at most four rounds, and then continues to converge to a minimal one in at most $5n + 4$ additional rounds, where n is the size of the network. The algorithm is written in the locally shared memory model with composite atomicity. It is proven assuming an unfair (distributed) daemon and takes $O(n \cdot \Delta^3)$ moves to stabilize, where Δ is the degree of the network.

There are several other self-stabilizing solutions for particular instances of (minimal) (f, g)-alliances proposed in the locally shared memory model with composite atomicity, *e.g.*, [17, 25, 32, 33, 34, 35].

Algorithms given in [32, 34] work in anonymous networks, however, they both assume a central daemon. More precisely, Srimani and Xu [32] give several algorithms which compute minimal global offensive and 1-minimal defensive alliances in $O(n^3)$ moves. Wang *et al.* [34] give a self-stabilizing algorithm to compute a minimal k-dominating set in $O(n^2)$ moves.

All other solutions [17, 25, 33, 35] consider arbitrary identified networks. Turau [33] gives a self-stabilizing algorithm to compute a minimal dominating set in $9n$ moves, assuming an unfair distributed daemon. Yahiaoui *et al.* [35] give self-stabilizing algorithms to compute a minimal global powerful alliance. Their solution assumes an unfair distributed daemon and stabilizes in $O(n{\cdot}m)$ moves, where m is the number of edges in the network.

A safely converging self-stabilizing algorithm is given in [25] for computing a minimal dominating set. The algorithm first computes a (not necessarily minimal) dominating set in $O(1)$ rounds and then safely stabilizes to a *minimal* dominating set in $O(D)$ rounds, where D is the diameter of the network. However, a synchronous daemon is required. A safely converging self-stabilizing algorithm for computing minimal global offensive alliances is given in [17]. This algorithm also assumes a synchronous daemon. It first computes a (not necessarily minimal) global offensive alliance within two rounds, and then safely stabilizes to a *minimal* global offensive alliance within $O(n)$ additional rounds.

To the best of our knowledge, until now there was no self-stabilizing algorithm solving the 1-minimal (f, g)-alliance without any restriction on f and g.

5.4 Algorithm FGA

Overview. Recall that we consider any network where $\delta_u \geq max(f(u), g(u))$, for every process u. Moreover, we assume that the network is identified, meaning that each process u can be distinguished using a

Algorithm 2 Algorithm FGA, code for every process u

Inputs:

- $st_u \in \{C, RB, RF\}$: variable of SDR
- $\mathbf{P\_Clean}(u)$: predicate of SDR
- id_u : identifier of u, constant from the system

Variables:

- $color_u$: Boolean
- $Vscore_u \in \{-1, 0, 1\}$: The score of u
- $canQuit_u$: Boolean
- $ptr_u \in N[u] \cup \{\perp\}$: Closed Neighborhood Pointer

Predicates:

- $\mathbf{P\_ICorrect}(u) \equiv score(u) \geq 0 \wedge [(Vscore_u = score(u) = 1) \vee ptr_u = \perp \vee (ptr_u \neq \perp \wedge Vscore_u = 1 \wedge \neg color_{ptr_u})]$
- $\mathbf{P\_reset}(u) \equiv color_u \wedge ptr_u = \perp \wedge canQuit_u \wedge Vscore_u = 1$
- $\mathbf{P\_canQuit}(u) \equiv color_u \wedge \#Alliance(u) \geq f(u) \wedge (\forall v \in N(u), Vscore_v = 1)$
- $\mathbf{P\_toQuit}(u) \equiv \mathbf{P\_canQuit}(u) \wedge (\forall v \in N[u], ptr_v = u)$
- $\mathbf{P\_updPtr}(u) \equiv \neg \mathbf{P\_toQuit}(u) \wedge ptr_u \neq bestPtr(u)$

Macros:

- $reset(u)$: $color_u := true$; $ptr_u := \perp$; $canQuit_u := true$; $Vscore_u := 1$;
- $\#Alliance(u)$: $|\{w \in N(u) \mid color_w\}|$
- $\begin{cases} score(u) = -1 & if \quad \#Alliance(u) < f(u) \quad \wedge \quad \neg color_u \\ score(u) = -1 & if \quad \#Alliance(u) < g(u) \quad \wedge \quad color_u \\ score(u) = 0 & if \quad \#Alliance(u) = f(u) \quad \wedge \quad \neg color_u \\ score(u) = 0 & if \quad \#Alliance(u) = g(u) \quad \wedge \quad color_u \\ score(u) = 1 & if \quad \#Alliance(u) > f(u) \quad \wedge \quad \neg color_u \\ score(u) = 1 & if \quad \#Alliance(u) > g(u) \quad \wedge \quad color_u \end{cases}$
- $CmpVar(u)$: $Vscore_u := score(u)$; $canQuit_u := \mathbf{P\_canQuit}(u)$;
- $bestPtr(u)$: if $(Vscore_u \leq 0)$ return $\perp$; if $(\forall v \in N[u], \neg canQuit_u)$ then return $\perp$; else return $\text{argmin}_{(v \in N[u] | canQuit_u)}(id_u)$;
- $upd(u)$: $CmpVar(u)$; $ptr_u := bestPtr(u)$;

Rules:

$\mathbf{rule\_Clr}(u)$	:	$\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_toQuit}(u)$	$\rightarrow$	$color_u := false$; $upd(u)$;
$\mathbf{rule\_P1}(u)$	:	$\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_updPtr}(u) \wedge ptr_u \neq \perp$	$\rightarrow$	$ptr_u := \perp$; $CmpVar(u)$;
$\mathbf{rule\_P2}(u)$	:	$\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_updPtr}(u) \wedge ptr_u = \perp$	$\rightarrow$	$upd(u)$;
$\mathbf{rule\_Q}(u)$	:	$\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u) \wedge \neg \mathbf{P\_toQuit}(u) \wedge \neg \mathbf{P\_updPtr}(u) \wedge (Vscore_u \neq score(u) \vee canQuit_u \neq \mathbf{P\_canQuit}(u))$	$\rightarrow$	$CmpVar(u)$; if $score(u) \leq 0$ then $ptr_u = \perp$;

unique constant identifier, here noted id_u. The formal code of FGA is given in Algorithm 2. Informally, each process u maintains the following four variables.

$color_u$**:** a boolean variable, the output of FGA. Process u belongs to the (f, g)-alliance if and only if $color_u$.

$Vscore_u$**:** a variable, whose domain is $\{-1, 0, 1\}$. $Vscore_u \leq 0$ if and only if no u's neighbor can quit the alliance.

$canQuit_u$**:** a boolean variable. $\neg canQuit_u$ if u cannot quit the alliance (in particular, if u is out of the alliance).

ptr_u**:** a pointer variable, whose domain is $N[u] \cup \{\perp\}$. Either $ptr_u = \perp$ or ptr_u designates the member of its closed neighborhood of smallest identifier such that $canQuit_u$.

In the following, we assume that the system is initially in the configuration γ_{init} where every process u has the following local state:

$$color_u = true, Vscore_u = 1, canQuit_u = true, ptr_u = \perp, st_u = C.$$

In particular, this means that all processes are initially in the alliance. Then, the idea of the algorithm is reduced the alliance until obtaining a 1-minimal (f, g)-alliance. A process u leaves the alliance by executing $\mathbf{rule\_Clr}(u)$. To leave the alliance, u should have enough neighbors in the alliance ($\#Alliance(u) \geq f(u)$), *approve* itself, and have a *full* approval from all neighbors. Process v approves u if $ptr_v = u$. Moreover, the approval of v is *full* if $Vscore_v = 1$. Notice that, the ptr pointers ensure that removals from the alliance are locally central: in the closed neighborhood of any process, at most one process leaves the alliance at each step.

To ensure the liveness of the algorithm, a process u gives its approval (by executing $\mathbf{rule\_P2}(u)$, maybe preceded by $\mathbf{rule\_P1}(u)$) to the member of its closed neighborhood (that is, including u itself) having the smallest identifier among the ones requiring an approval (*i.e.*, processes satisfying $canQuit$).

To ensure that $score(v) \geq 0$ is a closed predicate, a process v gives its approval to another process u only if $score(v) = 1$ and none of its neighbor can leave the alliance (*i.e.*, $ptr_u \notin N(v)$). This latter condition ensures that no neighbor of v leaves the alliance simultaneously to a new approval of v. It is mandatory since otherwise the cause for which v gives its approval may be immediately outdated. Hence, any approval switching is done either in one step when the process leaves the alliance, or in two atomic steps where ptr_v first takes the value $\perp$ (rule $\mathbf{rule\_P1}(u)$) and then points to the suitable process, (rule $\mathbf{rule\_P2}(u)$).

Finally, the rule $\mathbf{rule\_Q}(u)$ refreshes the values of $Vscore_u$, ptr_u, and $canQuit_u$ after a neighbor left the alliance or updated its $Vscore$ variable.

Properties. Below, we show some properties of Algorithm FGA that will be used for showing both its correctness and the self-stabilization of its composition with Algorithm SDR.

First, by checking the rules of FGA, we can remark that each time a process u sets $Vscore_u$ to a value other than 1, it also sets ptr_u to $\perp$, in the same step. Hence, by construction we have the following lemma.

Lemma 17 *$Vscore_u = 1 \vee ptr_u = \perp$ is closed by* FGA*, for every process u.*

Since Algorithm FGA does not modify any variable from Algorithm SDR, we have

Remark 7 *$\mathbf{P\_Clean}(u)$ is closed by* FGA*, for every process u.*

Lemma 18 *Let u be any process. Let γ be a configuration where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds. Let γ' be any configuration such that $\gamma \mapsto \gamma'$. In γ', $score(u) \geq 0$.*

Proof. By definition of $\mathbf{P\_toQuit}$, at most one process of $N[u]$ executes $\mathbf{rule\_Clr}$ in $\gamma \mapsto \gamma'$. If no process of $N[u]$ executes $\mathbf{rule\_Clr}$, we are done. If $\mathbf{rule\_Clr}(u)$ is executed, then $\#Alliance(u) \geq f(u)$ in γ (see $\mathbf{P\_canQuit}(u)$) and so $\#Alliance(u) \geq f(u)$ in γ' too and thus $score(u) \geq 0$ in γ'. Otherwise, let $v \in N(u)$ such that $\mathbf{rule\_Clr}(v)$ is executed in $\gamma \mapsto \gamma'$. In γ, $color_v = true$ and $\mathbf{P\_toQuit}(v)$ holds with, in particular, $ptr_u = v \neq \perp$, *i.e.*, $color_{ptr_u}$ holds. Hence, $\mathbf{P\_ICorrect}(u)$, $ptr_u \neq \perp$, and $color_{ptr_u}$ imply $score(u) = 1$ in γ, and so $score(u) \geq 0$ in γ'. □

By definition of FGA, we have

Remark 8 *Let u be any process. Let γ be a configuration where $ptr_u = v$ with $v \neq u$. Let γ' be any configuration such that $\gamma \mapsto \gamma'$. In γ', we have $ptr_u \in \{v, \perp\}$.*

Lemma 19 *Let u be any process. Let γ be a configuration where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds. Let γ' be any configuration such that $\gamma \mapsto \gamma'$ is a step where $v \in N(u)$ executes $\mathbf{rule\_Clr}(v)$. $\mathbf{P\_ICorrect}(u)$ holds in γ'*

Proof. Since $\mathbf{rule\_Clr}(v)$ is enabled in γ, we have $ptr_u = v$ and $Vscore_u = 1$. So, if u does not move in $\gamma \mapsto \gamma'$, we have $ptr_u \neq \perp$, $Vscore_u = 1$, and $\neg color_{ptr_u}$. Hence, $\mathbf{P\_ICorrect}(u)$ holds in γ' by Lemma 18.

Otherwise, either $\mathbf{rule\_P1}(u)$ or $\mathbf{rule\_Q}(u)$ is executed in $\gamma \mapsto \gamma'$. In the former case, $ptr_u = \perp$ in γ' and by Lemma 18, $\mathbf{P\_ICorrect}(u)$ holds in γ'. In the latter case, in γ' either $ptr_u = v \neq \perp$, $Vscore_u = 1$ (by Lemma 17), and $\neg color_{ptr_u}$, or $ptr_u = \perp$. In either case, $\mathbf{P\_ICorrect}(u)$ holds in γ' by Lemma 18, and we are done. □

Lemma 20 *Let u be any process. Let γ be a configuration where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds. Let γ' be any configuration such that $\gamma \mapsto \gamma'$ is a step where u executes an action and none of its neighbor executes $\mathbf{rule\_Clr}$. $\mathbf{P\_ICorrect}(u)$ holds in γ'.*

Proof. Since no neighbor of u executes $\mathbf{rule\_Clr}$ in $\gamma \mapsto \gamma'$, we have $Vscore_u = score(u)$ in γ', and by Lemmas 17 and 18, we are done. □

By Remark 7 and Lemmas 19-20, follows.

Corollary 6 $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ *is closed by* FGA*, for every process u.*

Since u is disabled in FGA if $\neg\mathbf{P\_Clean}(u)$ holds. $\neg\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ is also closed by FGA, for every process u. Hence, follows.

Corollary 7 $\mathbf{P\_ICorrect}(u)$ *is closed by* FGA*, for every process u.*

Partial Correctness. Consider any terminal configuration of FGA where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds for every process u. By checking the code of Algorithm FGA, we can remark that every process u satisfies

$$Vscore_u = score(u) \wedge canQuit_u = \mathbf{P\_canQuit}(u) \wedge$$
$$ptr_u = bestPtr(u) \wedge \neg\mathbf{P\_toQuit}(u).$$

Based on this, one can easily establish that in such a terminal configuration, the set $A = \{u \in V \mid color_u\}$ is a 1-minimal (f, g) alliance of the network. Indeed, for every process u, since $\mathbf{P\_ICorrect}(u)$ holds, $score(u) \geq 0$, which in turn implies that A is an (f, g) alliance. Assume then, by the contradiction, that A is not 1-minimal in some terminal configuration of FGA where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds for every process u. Let m be the process of minimum identifier such that $A - \{m\}$ is an (f, g) alliance. First, by definition, $color_m \wedge \#Alliance(m) \geq f(m)$ holds. Then, $\forall u \in N[m], Vscore_u = score(u) = 1$ since $A - \{m\}$ is an (f, g) alliance. So, $\mathbf{P\_canQuit}(m)$ holds, which implies that $canQuit_m = true$. Finally, by minimality of the m's identifier, $\forall u \in N[m]$, $ptr_u = bestPtr(u) = m$. Hence, $\mathbf{P\_toQuit}(m)$ holds, which in turn implies that $\mathbf{rule\_Clr}(m)$ is enabled, a contradiction. Hence, follows.

Theorem 5 *In any terminal configuration of* FGA *where* $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ *holds for every process* u*, the set* $A = \{u \in V \mid color_u\}$ *is a 1-minimal* (f, g) *alliance of the network.*

Consider now any execution e of FGA starting from γ_{init}. In γ_{init}, we have $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ for every process u. Then, $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$, for every process u, is invariant in e, by Corollary 6. Hence, we have the following corollary.

Corollary 8 *Let* e *be any execution of* FGA *starting from* γ_{init}*. If* e *terminates, then the set* $A = \{u \in V \mid color_u\}$ *is a 1-minimal* (f, g) *alliance of the network in the terminal configuration of* e*.*

Termination. We now show that any execution of FGA (starting from any arbitrary configuration) eventually terminates. Let v be any process. Let e be any execution of FGA. First, $\mathbf{rule\_Clr}(v)$ switches $color_v$ from true to false and no rule of FGA sets $color_v$ from false to true. So,

(1) $\mathbf{rule\_Clr}(v)$ is executed at most once in e.

Moreover, this implies that

(2) the value of the macro $\#Alliance(v)$ is monotonically non-increasing in e.

If $score(u) < 0$ holds for some process u in some configuration of e, then $\mathbf{P\_ICorrect}(u)$ does not hold and so u is disabled. Moreover, by (2), u is disabled forever in e. Assume, otherwise, that $score(u) \geq 0$. Then, $score(u)$ may increase at most once in e: when $\mathbf{rule\_Clr}(u)$ is executed while $\#Alliance(u) > f(u)$. So,

(3) Every process u updates the value of $Vscore_u$ at most 4 times in e.

Hence, overall (by (1)-(3)), the value of $\mathbf{P\_canQuit}(v)$ changes at most $4\delta_v + 2$ in e, and thus

(4) v updates the value of $canQuit_v$ at most $4\delta_v + 3$ in e.

By (3) and (4)

(5) $\mathbf{rule\_Q}(v)$ is executed at most most $4\delta_v + 7$ times in e.

The value of $bestPtr(v)$ may change only when a process u in the closed neighborhood of v changes the value of its variable $canQuit_u$ or when v updates $Vscore_v$. So, by (3) and (4), the value of $bestPtr(v)$ changes at most $\delta_v.(4\Delta + 3) + 4\delta_v + 7$ times. So,

(6) v executes $\mathbf{rule\_P1}$ and $\mathbf{rule\_P2}$ at most $\delta_v.(4\Delta + 3) + 4\delta_v + 8$ times each in e.

Overall (by (1), (5), and (6)), follows.

Lemma 21 *A process v executes at most $8\delta_v\Delta + 18\delta_v + 24$ moves in an execution of* FGA*.*

Corollary 9 *Any execution of* FGA *contains at most $16.\Delta.m + 36.m + 24.n$ moves,* i.e.*, $O(\Delta.m)$ moves.*

By Corollaries 8 and 9 , we can conclude with the following theorem.

Theorem 6 FGA *is distributed (non self-stabilizing) 1-minimal (f, g)-alliance algorithm which terminates in at most $O(\Delta.m)$ moves.*

Round Complexity. We already know that in any execution of FGA, each process executes **rule_Clr** at most once. So, along any execution there are at most n steps containing the execution of some rule **rule_Clr**. We now say that a step is $Color$*-restricted*, if no rule **rule_Clr** is executed during that step. Similarly, we say that a round is $Color$*-restricted* if it only consists of $Color$-restricted steps. In the sequel, we show that any execution that starts from a configuration where $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds for every process u contains at most 4 consecutive $Color$-restricted rounds. Hence, the number of rounds in any execution starting from γ_{init} is bounded by $5n + 4$. To that goal, we first specialize the notion of closure. A predicate P over configurations of FGA is $Color$*-restricted closed* if for every $Color$-restricted step $\gamma \mapsto \gamma'$, $P(\gamma) \Rightarrow P(\gamma')$.

We now consider the following predicates over configurations of FGA.

- $\mathcal{P}_5$ is true if and only if every process u satisfies $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$.
- $\mathcal{P}_6$ is true if and only if $\mathcal{P}_5$ holds and every process u satisfies $Vscore_u = score(u)$.
- $\mathcal{P}_7$ is true if and only if $\mathcal{P}_6$ holds and every process u satisfies $canQuit_u = \mathbf{P\_canQuit}(u)$.
- $\mathcal{P}_8$ is true if and only if $\mathcal{P}_7$ holds and every process u satisfies $ptr_u \in \{bestPtr(u), \perp\}$.
- $\mathcal{P}_9$ is true if and only if $\mathcal{P}_8$ holds and every process u satisfies $ptr_u = bestPtr(u)$.

Lemma 22 *$\mathcal{P}_6$ is $Color$-restricted closed. Moreover, after one $Color$-restricted round from any configuration satisfying $\mathcal{P}_5$, a configuration satisfying $\mathcal{P}_6$ is reached.*

Proof. Let u be a process. The value of $score(u)$ stays unchanged during a $Color$-restricted step. So, the predicate $Vscore_u = score(u)$ is $Color$-restricted closed, and so $\mathcal{P}_6$ is. Let γ be a configuration satisfying $\mathcal{P}_5$. Recall that $\mathcal{P}_5$ is closed, by Corollary 6. So, $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds forever from γ. If $Vscore_u \neq score(u)$ in γ, then u is enabled in FGA. Now, if u moves in a $Color$-restricted step, then $Vscore_u = score(u)$ in the reached configuration. Hence, $Vscore_u = score(u)$ holds, for every process u, within at most one round from γ, and we are done. □

Lemma 23 *$\mathcal{P}_7$ is $Color$-restricted closed. Moreover, after one $Color$-restricted round from any configuration satisfying $\mathcal{P}_6$, a configuration satisfying $\mathcal{P}_7$ is reached.*

Proof. Let u be a process. Let $\gamma_i \mapsto \gamma_{i+1}$ be a $Color$-restricted step such that $\mathcal{P}_6(\gamma_i)$ holds. For every process v, the value of $color_v$, $Vscore_v$, and $\#Alliance(v)$ stay unchanged during $\gamma_i \mapsto \gamma_{i+1}$. Therefore, the value of $\mathbf{P\_canQuit}(u)$ stays unchanged during $\gamma_i \mapsto \gamma_{i+1}$ for every process u. So,

the predicate $\mathcal{P}_6 \wedge canQuit_u = \mathbf{P\_canQuit}(u)$ is $Color$-restricted closed, and so $\mathcal{P}_7$ is. Assume now that, $canQuit_u \neq \mathbf{P\_canQuit}(u)$ holds in γ_i. Then, u is enabled in γ_i and if u moves in $\gamma_i \mapsto \gamma_{i+1}$, $canQuit_u = \mathbf{P\_canQuit}(u)$ holds in in γ_{i+1}. Indeed, the value of $\mathbf{P\_canQuit}(u)$ stays unchanged during the step. Therefore, after at most one $Color$-restricted round from γ_i, we have $canQuit_u = \mathbf{P\_canQuit}(u)$ for every process u, and so $\mathcal{P}_7$ holds. □

Lemma 24 *$\mathcal{P}_8$ is $Color$-restricted closed. Moreover, after one $Color$-restricted round from any configuration satisfying $\mathcal{P}_7$, a configuration satisfying $\mathcal{P}_8$ is reached.*

Proof. Let u be a process. Let $\gamma_i \mapsto \gamma_{i+1}$ be a $Color$-restricted step such that $\mathcal{P}_7(\gamma_i)$ holds. During that step, only rules $\mathbf{rule\_P1}$ or $\mathbf{rule\_P2}$ are executed. Moreover, for every process v, the value of $color_v$ and $canQuit_v$ stay unchanged during $\gamma_i \mapsto \gamma_{i+1}$. So, the value of $bestPtr(u)$ stays unchanged as well. So, the predicate $\mathcal{P}_7 \wedge ptr_u \in \{bestPtr(u), \perp\}$ is $Color$-restricted closed, and so $\mathcal{P}_8$ is. Assume now that, $ptr_u \notin \{bestPtr(u), \perp\}$ holds in γ_i. Then, u is enabled in γ_i and if u moves in $\gamma_i \mapsto \gamma_{i+1}$, $ptr_u \in \{bestPtr(u), \perp\}$ in γ_{i+1}. Indeed, the value of $bestPtr(u)$ stays unchanged during the step. Therefore, after at most one $Color$-restricted round from γ_i, we have $ptr_u \in \{bestPtr(u), \perp\}$ for every process u, and so $\mathcal{P}_8$ holds. □

Lemma 25 *$\mathcal{P}_9$ is $Color$-restricted closed. Moreover, after one $Color$-restricted round from any configuration satisfying $\mathcal{P}_8$, a configuration satisfying $\mathcal{P}_9$ is reached.*

Proof. Let u be a process. Let $\gamma_i \mapsto \gamma_{i+1}$ be a $Color$-restricted step such that $\mathcal{P}_8(\gamma_i)$ holds. During that step, only rules $\mathbf{rule\_P2}$ are executed. Moreover, for every process v, the value of $color_v$ and $canQuit_v$ stay unchanged during $\gamma_i \mapsto \gamma_{i+1}$. So, the value of $bestPtr(u)$ stays unchanged as well. So, the predicate $\mathcal{P}_8 \wedge ptr_u = bestPtr(u)$ is $Color$-restricted closed, and so $\mathcal{P}_9$ is. Assume now that, $ptr_u \neq bestPtr(u)$ holds in γ_i. Then, u is enabled in γ_i and if u moves in $\gamma_i \mapsto \gamma_{i+1}$, $ptr_u = bestPtr(u)$ in γ_{i+1}. Indeed, the value of $bestPtr(u)$ stays unchanged during the step. Therefore, after at most one $Color$-restricted round from γ_i, we have $ptr_u = bestPtr(u)$ for every process u, and so $\mathcal{P}_8$ holds. □

Theorem 7 *Starting from any configuration satisfying $\mathcal{P}_5$, Algorithm* FGA *terminates in at most $5n + 4$ rounds.*

Proof. According to Lemmas 22-25, after 4 consecutive $Color$-restricted rounds, every process u satisfies:

$$Vscore_u = score(u) \wedge canQuit_u = \mathbf{P\_canQuit}(u) \wedge \neg \mathbf{P\_updPtr}(u)$$

So only the rule $\mathbf{rule\_Clr}(u)$ may be enabled at u. We conclude that an execution of $\mathbf{rule\_Clr}$ occurs at least every 5 rounds, unless the system reaches a terminal configuration. Since, along any execution, there are at most n steps containing the execution of some rule $\mathbf{rule\_Clr}$, the theorem follows. □

Since γ_{init} satisfies $\mathcal{P}_5$, we have the following corollary.

Corollary 10 *Starting from γ_{init}, Algorithm* FGA *terminates in at most $5n + 4$ rounds.*

5.5 Algorithm FGA ∘ SDR

Requirements. To show the self-stabilization of FGA ∘ SDR, we should first establish that FGA meets the requirements 1 to 2d, given in Subsection 3.5.

1. From the code of FGA, we can deduce that Requirements 1, 2b, 2c, and 2e are satisfied.
2. Requirement 2a is satisfied since $\mathbf{P\_ICorrect}(u)$ does not involve any variable of SDR and is closed by FGA (Corollary 7).
3. Finally, recall that $\delta_u \geq max(f(u), g(u))$, for every process u. So, if $\mathbf{P\_reset}(v)$ holds, for every $v \in N[u]$, then $score(u) = 1$ and so $\mathbf{P\_ICorrect}(u)$ holds, by definition. Hence, Requirement 2d holds.

Partial Correctness. Let γ be any terminal configuration of FGA ∘ SDR. Then, $\gamma_{|\mathtt{SDR}}$ is a terminal configuration of SDR and, by Theorem 1, $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds in γ, and so $\gamma_{|\mathtt{SDR}}$, for every process u. Moreover, $\gamma_{|\mathtt{FGA}}$ is a terminal configuration of FGA. Hence, by Theorem 5, follows.

Theorem 8 *In any terminal configuration of* FGA ∘ SDR*. the set* $\{u \in V \mid color_u\}$ *is a 1-minimal* (f, g) *alliance of the network.*

Termination and Self-Stabilization. Let u be a process. Let e be an execution of FGA ∘ SDR. By Corollary 3 (page 15), the sequence of rules executed by a process u in a segment of e belongs to the following language:

$$(\mathbf{rule\_C} + \varepsilon)\ \mathbf{words\_I}\ (\mathbf{rule\_RB} + \mathbf{rule\_R} + \varepsilon)\ (\mathbf{rule\_RF} + \varepsilon)$$

where $\mathbf{words\_I}$ be any sequence of rules of FGA. Moreover, by Lemma 10 (page 16) and Lemma 21, $\mathbf{words\_I}$ is bounded by $8\delta_u\Delta + 18\delta_u + 24$. Thus, u executes at most $8\delta_u\Delta + 18\delta_u + 27$ moves in any segment of e and, overall each segment of e contains at most $16m\Delta + 36m + 27n$ moves. Since, e contains at most $n + 1$ segments (Remark 5, page 12), e contains at most $(n + 1).(16m\Delta + 36m + 27n)$ moves, and we have the following theorem.

Theorem 9 *Any execution of* FGA ∘ SDR *terminates in* $O(\Delta.n.m)$ *moves.*

By Theorems 8 and 9, we can conclude:

Theorem 10 FGA ∘ SDR *is self-stabilizing for the 1-minimal* (f, g)*-alliance problem. Its stabilization time is in* $O(\Delta.n.m)$ *moves.*

Round Complexity. Corollary 5 (page 20) establishes that after at most $3n$ rounds a normal configuration of FGA ∘ SDR is reached. Then, since the set of normal configuration is closed (still by Corollary 5), all rules of SDR algorithm are disabled forever from such a configuration, by Lemma 15 (page 19). Moreover, in a normal configuration, $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ holds, for every process u (still by Corollary 5). Hence, after at most $5n + 4$ additional rounds, a terminal configuration of FGA ∘ SDR is reached, by Theorem 7, and follows.

Theorem 11 *The stabilization time of* FGA ∘ SDR *is at most* $8n + 4$ *rounds.*

6 Asynchronous Unison

6.1 The Problem

We now consider the problem of *asynchronous unison* (introduced in [12]), simply referred to as unison in the following. This problem is a clock synchronization problem: each process u holds a variable (usually an integer variable) called its *clock*, here noted val_u. Then, the problem is specified as follows:

Each process should increment its clock infinitely often (liveness).

The difference between clocks of every two neighbors should be at most one increment at each instant. (safety)

Notice that we consider here periodic clocks, *i.e.*, the clock incrementation is modulo a so-called *period*, here noted K.

6.2 Related Work

The first self-stabilizing asynchronous unison for general connected graphs was proposed by Couvreur *et al.* [12]. It uses at least n^2 states per processes, where n is the number of process. However, no complexity analysis was given. Another solution which stabilizes in $O(n)$ rounds has been proposed by Boulinier *et al.* in [9] using $m + \alpha$ states per process, where the period m should satisfy $m > C_G$ and α is a parameter that should satisfy $\alpha \geq T_G - 2$. C_G is the *cyclomatic characteristic* of the network G, *i.e.*, the length of the shortest maximal cycle of a cycle basis of G if G is cyclic, and 2 if G is a tree. T_G is the length of the longest chordless cycle in a graph G. Boulinier also proposed in his PhD thesis a parametric solution which generalizes both the solutions of [12] and [9]. In particular, the complexity analysis of this latter algorithm reveals an upper bound in $O(D.n)$ rounds on the stabilization time of the Couvreur *et al.*' algorithm, where D is the network diameter.

6.3 Contribution

We first propose a distributed algorithm called U. Starting from a pre-defined configuration, U implements the unison problem, providing that the input K satisfies $K > n$. U is not self-stabilizing, however we show that the composite algorithm U $\circ$ SDR is actually an efficient self-stabilizing unison algorithm. Indeed, its stabilization times in round matches the one of the best existing solution [9]. Moreover, it can be implemented using $O(\log n)$ bits per process, like the solution of [9]. Moreover, it achieves a better stabilization time in moves, since the algorithm in [9] stabilizes in $O(D.n^3 + \alpha.n^2)$ moves (as shown in [15]) while our solution stabilizes in $O(D.n^2)$ moves.

6.4 Algorithm U

Overview. We consider here anonymous (bidirectional) networks of arbitrary connected topology. Moreover, every process has the period K as input. K is required to be (strictly) greater than n, the number of processes. The formal code of Algorithm U is given in Algorithm 3. Informally, each process maintains a single variable, its clock val_u, using a single rule $\mathbf{rule\_U}(u)$.

In the following, we assume that the system is initially in the configuration γ_{init} where every process u satisfies $val_u = 0 \wedge st_u = C$. Basically, starting from γ_{init}, a process u can increment its clock val_u modulo K (using rule $\mathbf{rule\_U}(u)$) if it is on time or one increment late with all its neighbors, see predicate $\mathbf{P\_UP}(u)$.

Algorithm 3 Algorithm U, code for every process u

Inputs:

- $st_u \in \{C, RB, RF\}$: variable of SDR
- $\mathbf{P\_Clean}(u)$: predicate of SDR
- K : a constant from the system satisfying $K > n$

Variables:

- $val_u \in \mathbb{N}$: the clock of u

Predicates:

- $\mathbf{P\_agree}(u, v) \equiv val_v \in \{(val_u - 1) \bmod K, val_u, (val_u + 1) \bmod K\}$
- $\mathbf{P\_ICorrect}(u) \equiv (\forall v \in N(u), \mathbf{P\_agree}(u, v))$
- $\mathbf{P\_reset}(u) \equiv val_u = 0$
- $\mathbf{P\_UP}(u) \equiv (\forall v \in N(u), val_v \in \{val_u, (val_u - 1) \bmod K\})$

Macros:

- $reset(u)$: $val_u := 0$;

Rules:

$\mathbf{rule\_U}(u)$: $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_UP}(u) \rightarrow val_u := (val_u + 1) \bmod K$;

Correctness. Consider any two neighboring processes u and v such that $\mathbf{P\_agree}(u, v)$ in some configuration γ, *i.e.*, $val_v \in \{(val_u - 1) \bmod K, val_u, (val_u + 1) \bmod K\}$. Without the loss of generality, assume that $val_v \in \{val_u, (val_u + 1) \bmod K\}$ (otherwise switch the role of u and v). Let $\gamma \mapsto \gamma'$ be the next step. If $val_v = val_u$ in γ, then $val_v \in \{(val_u - 1) \bmod K, val_u, (val_u + 1) \bmod K\}$ in γ' since each clock increments at most once per step, *i.e.*, $\mathbf{P\_agree}(u, v)$ still holds in γ'. Otherwise, $val_v = (val_u + 1) \bmod K\}$ in γ, and so v is disable and only u may move. If u does not move, then $val_v = (val_u + 1) \bmod K\}$ still holds in γ', otherwise $val_v = val_u$ in γ'. Hence, in both cases, $\mathbf{P\_agree}(u, v)$ still holds in γ'. Hence, follows.

Lemma 26 $\mathbf{P\_ICorrect}(u)$ *is closed by* U*, for every process* u*.*

Since Algorithm U does not modify any variable from Algorithm SDR, we have

Remark 9 $\mathbf{P\_Clean}(u)$ *is closed by* U*, for every process* u*.*

Corollary 11 $\mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_Clean}(u)$ *is closed by* U*, for every process* u*.*

Corollary 12 *Any execution of* U*, that starts from a configuration where* $\mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_Clean}(u)$ *holds for every process* u*, satisfies the safety of the unison problem.*

Lemma 27 *Any configuration where* $\mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_Clean}(u)$ *holds for every process* u *is not terminal.*

Proof. Assume, by the contradiction, a terminal configuration γ where $\mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_Clean}(u)$ for every process u. Then, every process u has at least one neighbor v such that $val_v = (val_u + 1) \bmod K$. Since the number of processes is finite, in γ there exist elementary cycles $u_1, \ldots, u_x$ such that

1. for every $i \in \{1, \ldots, x-1\}$, u_i and u_{i+1} are neighbors, and $val_{u_i} = (val_{u_{i+1}} + 1) \bmod K$; and
2. u_1 and u_x are neighbors and $val_{u_x} = (val_{u_1} + 1) \bmod K$.

By transitivity, Case 1 implies that $val_{u_1} = (val_{u_x} + x - 1) \bmod K$. So, from Case 2, we obtain $val_{u_x} = (val_{u_x} + x) \bmod K$. Now, by definition $x \leq n$ and $K > n$ so $val_{u_x} \neq (val_{u_x} + x) \bmod K$, a contradiction. Hence, γ is not terminal. □

Lemma 28 *Any execution of* U*, that starts from a configuration where* $\mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_Clean}(u)$ *holds for every process* u*, satisfies the liveness of the unison problem.*

Proof. Let e be any execution of U that starts from a configuration where $\mathbf{P\_ICorrect}(u) \wedge \mathbf{P\_Clean}(u)$ holds for every process u. Assume, by the contradiction, that e does not satisfy the liveness of unison. Then, e contains a configuration γ from which some processes (at least one) never more executes $\mathbf{rule\_U}$. Let F be the non-empty subset of processes that no more move from γ. Let $I = V \setminus F$. By Lemma 27, I is not empty too. Now, since the network is connected, there are two processes u and v such that $u \in I$ and $v \in F$. Now, after at most 3 increments of u from γ, $\mathbf{P\_agree}(u, v)$ no more holds, contradicting Lemma 26. □

Consider now any execution e of U starting from γ_{init}. In γ_{init}, we have $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ for every process u. Hence, by Corollary 12 and Lemma 28, follows.

Theorem 12 U *is distributed (non self-stabilizing) unison algorithm.*

Properties of U**.** Consider any execution e of U starting from a configuration which does not satisfy $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ for every process u. Then, there exists at least one process u satisfying $\neg\mathbf{P\_Clean}(u) \vee \neg\mathbf{P\_ICorrect}(u)$ in γ. Then, u is disable forever from γ. Indeed, if $\neg\mathbf{P\_Clean}(u)$, then $\neg\mathbf{P\_Clean}(u)$ holds forever since U does not write into SDR's variables. If $\neg\mathbf{P\_ICorrect}(u)$ holds, then there is a neighbor v such that $\neg\mathbf{P\_agree}(u, v)$ holds, both u and v are disabled, hence so $\neg\mathbf{P\_agree}(u, v)$ holds forever, which implies that $\neg\mathbf{P\_UP}(u)$ forever.

Now, since u is disabled forever, each neighbor of u moves at most three times in e. Inductively, every node at distance d from u moves at most $3d$ times. Overall, we obtain the following lemma.

Lemma 29 *In any execution of* U *starting from a configuration which does not satisfy* $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ *for every process* u*, each process moves at most* $3D$ *times, where* D *is the network diameter.*

6.5 Algorithm U ∘ SDR

Requirements. To show the self-stabilization of U ∘ SDR, we should first establish that U meets the requirements 1 to 2d, given in Subsection 3.5.

Requirement 2a is satisfied since $\mathbf{P\_ICorrect}(u)$ does not involve any variable of SDR and is closed by U (Lemma 26). All other requirements directly follow from the code of U.

Self-stabilization and Move Complexity. We define the legitimate configurations of U ∘ SDR as the set of configurations satisfying $\mathbf{P\_Clean}(u) \wedge \mathbf{P\_ICorrect}(u)$ for every process u. This set actually corresponds to the set of normal configurations (see Corollary 5, page 20) and is closed by Algorithm U ∘

SDR, by Remark 2 (page 10), Theorem 1 (page 9), and Lemma 11. Then, from any normal configuration, the specification of the unison holds, by Corollary 12 and Lemma 28. So, it remains to show the convergence.

Let u be a process. Let e be an execution of U $\circ$ SDR. By Corollary 3 (page 15), the sequence of rules executed by a process u in a segment of e belongs to the following language:

$$(\mathbf{rule\_C} + \varepsilon)\ \mathbf{words\_I}\ (\mathbf{rule\_RB} + \mathbf{rule\_R} + \varepsilon)\ (\mathbf{rule\_RF} + \varepsilon)$$

where $\mathbf{words\_I}$ be any sequence of rules of U.

Let assume that e contains s segments. Recall that $s \leq n + 1$ (see Remark 5, page 12). Let call *regular* segment any segment that starts in a configuration containing at least one abnormal alive root. A regular segment contains no normal configuration, so, by Lemma 10 (page 16) and Lemma 29, the sequence $\mathbf{words\_I}$ of u is bounded by $3D$ in S. Thus, u executes at most $3D + 3$ moves in S and, overall a regular segment contains at most $(3D + 3).n$ moves and necessarily ends by a step where the number of abnormal alive root decreases. Hence, all $s - 1$ first segments are regular and the last one is not. Overall the last segment S_{last} starts after at most $(3D + 3).n.(s - 1)$ moves. S_{last} contains no abnormal alive root and so, the sequence of rules executed by u in S_{last} belongs to the following language: $(\mathbf{rule\_C} + \varepsilon)\ \mathbf{words\_I}$.[2] If the initial configuration of S_{last} contains no process of status EF, then it is a normal configuration and so $s = 1$, *i.e.*, e is initially in a normal configuration. Otherwise, let v be a process such that $st_v = EF$ in the initial configuration of S_{last} and no other process executes $\mathbf{rule\_C}$ later than v. Following the same reasoning as in Lemma 29, while v does not execute $\mathbf{rule\_C}$, each process other than v can execute at most $3D$ rules of U and one $\mathbf{rule\_C}$. Hence, there are at most $(3D + 1).(n - 1) + 1$ moves in S_{last} before the system reaches a normal configuration.

Since, in the worst case $s = n + 1$, overall e reaches a normal configuration in at most $(3D + 3).n^2 + (3D + 1).(n - 1) + 1$ moves, and we have the following theorem.

Theorem 13 U $\circ$ SDR *is self-stabilizing for the unison problem. Its stabilization time is in* $O(D.n^2)$ *moves.*

Round Complexity. By Corollary 5 (page 20), follows.

Theorem 14 *The stabilization time of* U $\circ$ SDR *is at most* $3n$ *rounds.*

References

[1] Yehuda Afek and Anat Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998, 1998.

[2] Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Inf. Comput.*, 254:330–366, 2017.

[3] Dana Angluin, James Aspnes, David Eisenstat, and Eric Ruppert. The computational power of population protocols. *Distributed Computing*, 20(4):279–304, 2007.

[4] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.

[2] Otherwise, $st_u = RB$ in some configuration of S_{last}, and that configuration contains an abnormal root, by Lemma 7 page 13, a contradiction

[5] Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network {RESET} (extended abstract). In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, pages 254–263. ACM, 1994.

[6] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 268–277. IEEE Computer Society, 1991.

[7] Baruch Awerbuch, Boaz Patt-Shamir, George Varghese, and Shlomi Dolev. Self-stabilization by local checking and global reset (extended abstract). In *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, volume 857 of *Lecture Notes in Computer Science*, pages 326–339. Springer, 1994.

[8] C. Berge. *The Theory of Graphs*. Dover books on mathematics. Dover, 2001.

[9] Christian Boulinier, Franck Petit, and Vincent Villain. When graph theory helps self-stabilization. In *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing (PODC), July 25-28, 2004*, pages 150–159, 2004.

[10] Fabienne Carrier, Ajoy Kumar Datta, Stéphane Devismes, Lawrence L. Larmore, and Yvan Rivierre. Self-stabilizing (f, g)-alliances with safe convergence. *J. Parallel Distrib. Comput.*, 81-82:11–23, 2015.

[11] Alain Cournier, Ajoy K. Datta, Franck Petit, and Vincent Villain. Snap-stabilizing PIF algorithm in arbitrary networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 199–206, 2002.

[12] Jean-Michel Couvreur, Nissim Francez, and Mohamed G. Gouda. Asynchronous unison (extended abstract). In *Proceedings of the 12nd International Conference on Distributed Computing Systems (ICDCS'92)*, pages 486–493, 1992.

[13] Ajoy K. Datta, Lawrence L. Larmore, and Priyanka Vemula. An $O(N)$-time self-stabilizing leader election algorithm. *JPDC*, 71(11):1532–1544, 2011.

[14] Stéphane Devismes and Colette Johnen. Silent self-stabilizing BFS tree algorithms revisited. *JPDC*, 97:11 – 23, 2016.

[15] Stéphane Devismes and Franck Petit. On efficiency of unison. In *4th Workshop on Theoretical Aspects of Dynamic Distributed Systems, (TADDS '12)*, pages 20–25, 2012.

[16] Edsger W. Dijkstra. Self-stabilizing Systems in Spite of Distributed Control. *Commun. ACM*, 17(11):643–644, 1974.

[17] Yihua Ding, James Z. Wang, and Pradip K. Srimani. Self-stabilizing minimal global offensive alliance algorithm with safe convergence in an arbitrary graph. In T. V. Gopal, Manindra Agrawal, Angsheng Li, and S. Barry Cooper, editors, *Theory and Applications of Models of Computation - 11th Annual Conference (TAMC)*, volume 8402 of *Lecture Notes in Computer Science*, pages 366–377, Chennai, India, April 11-13 2014. Springer.

[18] S Dolev, A Israeli, and S Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

[19] Shlomi Dolev, Mohamed G. Gouda, and Marco Schneider. Memory requirements for silent stabilization. *Acta Informatica*, 36(6):447–462, 1999.

[20] Mitre Costa Dourado, Lucia Draque Penso, Dieter Rautenbach, and Jayme Luiz Szwarcfiter. The south zone: Distributed algorithms for alliances. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, (SSS)*, volume 6976 of *Lecture Notes in Computer Science*, pages 178–192, Grenoble, France, October 10-12 2011. Springer.

[21] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[22] Christian Glacet, Nicolas Hanusse, David Ilcinkas, and Colette Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. In *SSS'14*, pages 120–134, 2014.

[23] Anupam Gupta, Bruce M. Maggs, Florian Oprea, and Michael K. Reiter. Quorum placement in networks to minimize access delays. In Marcos Kawazoe Aguilera and James Aspnes, editors, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 87–96, Las Vegas, NV, USA, July 17-20 2005. ACM.

[24] Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *IPL*, 41(2):109–117, 1992.

[25] Hirotsugu Kakugawa and Toshimitsu Masuzawa. A self-stabilizing minimal dominating set algorithm with safe convergence. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 8.–, Rhodes Island, Greece, 25-29 April 2006. IEEE.

[26] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7(1):17–26, 1993.

[27] Yonghwan Kim, Junya Nakamura, Yoshiaki Katayama, and Toshimitsu Masuzawa. A cooperative partial snapshot algorithm for checkpoint-rollback recovery of large-scale and dynamic distributed systems. In *CANDAR'18*, pages 285–291, 11 2018.

[28] Chung-Shou Liao and Gerard J. Chang. k-tuple domination in graphs. *Inf. Process. Lett.*, 87(1):45–50, 2003.

[29] J.M. Sigarreta and J.A. Rodríguez. On the global offensive alliance number of a graph. *Discrete Applied Mathematics*, 157(2):219 – 226, 2009.

[30] Jose Maria Sigarreta and Juan Alberto Rodríguez-Velazquez. On defensive alliances and line graphs. *Appl. Math. Lett.*, 19(12):1345–1350, 2006.

[31] M Sloman and J Kramer. *Distributed systems and computer networks.* Prentice Hall, 1987.

[32] Pradip K. Srimani and Zhenyu Xu. Distributed protocols for defensive and offensive alliances in network graphs using self-stabilization. In *International Conference on Computing: Theory and Applications (ICCTA)*, pages 27–31, Kolkata, India, March 5-7 2007. IEEE Computer Society.

[33] Volker Turau. Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler. *Inf. Process. Lett.*, 103(3):88–93, 2007.

[34] Guangyuan Wang, Hua Wang, Xiaohui Tao, and Ji Zhang. A self-stabilizing algorithm for finding a minimal k-dominating set in general networks. In Yang Xiang, Mukaddim Pathan, Xiaohui Tao, and Hua Wang, editors, *Data and Knowledge Engineering*, Lecture Notes in Computer Science, pages 74–85. Springer Berlin Heidelberg, 2012.

[35] Saïd Yahiaoui, Yacine Belhoul, Mohammed Haddad, and Hamamache Kheddouci. Self-stabilizing algorithms for minimal global powerful alliance sets in graphs. *Inf. Process. Lett.*, 113(10-11):365–370, 2013.