Being Efficient in Time, Space, and Workload: a Self-Stabilizing Unison and Its Consequences

Stéphane Devismes
Laboratoire MIS, Université de Picardie, 33 rue Saint Leu – 80039 Amiens cedex 1, France

David Ilcinkas
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Colette Johnen
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Frédéric Mazoit
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Abstract

We present a self-stabilizing algorithm for the unison problem which is efficient in time, workload, and space in a weak model. Precisely, our algorithm is defined in the atomic-state model and works in anonymous asynchronous connected networks in which even local ports are unlabeled. It makes no assumption on the daemon and thus stabilizes under the weakest one: the distributed unfair daemon.

In an n-node network of diameter D and assuming the knowledge $B \geq 2D + 2$, our algorithm only requires $\Theta(\log(B))$ bits per node and is fully polynomial as it stabilizes in at most $2D + 2$ rounds and $O(\min(n^2B, n^3))$ moves. In particular, it is the first self-stabilizing unison for arbitrary asynchronous anonymous networks achieving an asymptotically optimal stabilization time in rounds using a bounded memory at each node.

Furthermore, we show that our solution can be used to efficiently simulate synchronous self-stabilizing algorithms in asynchronous environments. For example, this simulation allows us to design a new state-of-the-art algorithm solving both the leader election and the BFS (Breadth-First Search) spanning tree construction in any identified connected network which, to the best of our knowledge, beats all existing solutions in the literature.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Distributed algorithms; Theory of computation → Design and analysis of algorithms

Keywords and phrases Self-stabilization, unison, time complexity, synchronizer

Digital Object Identifier 10.4230/LIPIcs.STACS.2025.30

Funding *David Ilcinkas, Colette Johnen, and Frédéric Mazoit*: This work was supported by the ANR project ENEDISC (`ANR-24-CE48-7768`). *Colette Johnen and Stéphane Devismes*: This work was supported by the ANR project SkyData (`ANR-22-CE25-0008`).

1 Introduction

1.1 Context

Self-stabilization is a general non-masking and lightweight fault tolerance paradigm [25, 3]. Precisely, a distributed system achieving this property inherently tolerates *any* finite number of transient faults.[1] Indeed, starting from an arbitrary configuration, which may be the result of such faults, a self-stabilizing system recovers within finite time, and without any external intervention, a so-called *legitimate configuration* from which it satisfies its specification.

[1] A *transient fault* occurs at an unpredictable time, but does not result in a permanent hardware damage. Moreover, as opposed to intermittent faults, the frequency of transient faults is considered to be low.

In this paper, we consider the most commonly used model in the self-stabilizing area: the *atomic-state* model [25, 3]. In this model, the state of each node is stored into registers and these registers can be directly read by neighboring nodes. Furthermore, in one atomic step, a node can read its state and that of its neighbors, perform some local computation, and update its state accordingly. In the atomic-state model, asynchrony is materialized by an adversary called *daemon* that can restrict the set of possible executions. We consider here the weakest (i.e., the most general) daemon: the *distributed unfair daemon*.

Self-stabilizing algorithms are mainly compared according to their *stabilization time*, i.e., the worst-case time to reach a legitimate configuration starting from an arbitrary one. In the atomic-state model, stabilization time can be evaluated in terms of *rounds* and *moves*. Rounds [13] capture the execution time according to the speed of the slowest nodes. Moves count the number of local state updates. So, the move complexity is rather a measure of work than a measure of time. It turns out that obtaining efficient stabilization times both in rounds and moves is a difficult task. Usually, techniques to design an algorithm achieving a stabilization time polynomial in moves make its round complexity inherently linear in n, the number of nodes (see, e.g., [2, 23, 19]). Conversely, achieving the asymptotic optimality in rounds, usually $O(D)$ where D is the network diameter, commonly makes the stabilization time exponential in moves (see, e.g., [22, 31]). Surprisingly, Cournier, Rovedakis, and Villain [14] manage to prove the first *fully polynomial* (i.e., with $Poly(n)$ move and $Poly(D)$ round complexities) silent[2] self-stabilizing algorithm. Their algorithm builds a BFS (Breadth-First Search) spanning tree in any rooted connected network and they prove that it stabilizes in $O(n^6)$ moves and $O(D^2)$ rounds using $\Theta(\log B + \log \Delta)$ bits per node, where B is an upper bound on D and Δ is the maximum degree of the network.

Up to now, fully polynomial self-stabilizing algorithms have only been proposed (see [14, 21]) for so called *static* problems [34], such as spanning tree constructions and leader election, which compute a fixed object in finite time. In this paper, we propose an algorithm for a fundamental *dynamic* (i.e., non static) problem: the *asynchronous unison* (*unison* for short). It consists in maintaining a local clock at each node. The domain of clocks can be bounded (like everyday clocks) or infinite. The liveness property of the problem requests each node to increment its own clock infinitely often. Furthermore, the safety property of the unison requires the difference between the clocks of any two neighbors to always be at most one increment. The usefulness of the unison comes from the fact that asynchrony often makes fault tolerance very difficult in distributed systems. The impossibility of achieving consensus in an asynchronous system in spite of at most one process crash [30] is a famous example illustrating this fact. Thus, fault tolerance, and in particular self-stabilization, often requires some kind of barrier synchronization, which the unison provides, to control the asynchronism of the system by making processes progress roughly at the same speed. Unison is thus a fundamental algorithmic tool that has numerous applications. Among others, it can be used to simulate synchronous systems in asynchronous environments [17], to free an asynchronous system from its fairness assumption (e.g., using the cross-over composition) [8], to facilitate the termination detection [9], to locally share resources [11], or to achieve infimum computations [10]. Thus, as expected, we also derive from our unison algorithm a *synchronizer* allowing us to obtain several new state-of-the-art self-stabilizing algorithms for various problems, including spanning tree problems and leader election.

[2] In the atomic-state model, a self-stabilizing algorithm is *silent* if all its executions terminate.

1.2 Related Work

Related Work on the Self-stabilizing Unison

The first self-stabilizing asynchronous unison for general graphs was proposed by Couvreur, Francez, and Gouda [15] in the link-register model (a locally-shared memory model without composite atomicity [27, 26]). However, no complexity analysis was given. Another solution, which stabilizes in $O(n)$ rounds, is proposed by Boulinier, Petit, and Villain [11] in the atomic-state model assuming a distributed unfair daemon. Its move complexity is shown in [24] to be in $O(Dn^3 + \alpha n^2)$, where α is a parameter of the algorithm that should satisfy $\alpha \geq L - 2$, where L is the length of the longest hole in the network. In his PhD thesis, Boulinier proposes a parametric solution that generalizes the solutions of both [15] and [11]. In particular, the time complexity analysis of this latter algorithm reveals an upper bound in $O(D \cdot n)$ rounds on the stabilization time of the atomic-state model version of the algorithm in [15]. Awerbuch, Kutten, Mansour, Patt-Shamir, and Varghese [4] propose a self-stabilizing unison that stabilizes in $O(D)$ rounds using an infinite state space. The move complexity of their solution is not analyzed. An asynchronous self-stabilizing unison algorithm is given in [23]. It stabilizes in $O(n)$ rounds and $O(\Delta \cdot n^2)$ moves using unbounded local memories. Emek and Keren [28] present in the stone age model a self-stabilizing unison that stabilizes in $O(B^3)$ rounds, where B is an upper bound on D known by all nodes. Their solution requires $\Theta(\log B)$ bits per node. Moreover, since node activations are required to be fair, the move complexity of their solution is unknown and may be unbounded.

Related Work on Simulations

Simulation is a useful tool to simplify the design of algorithms. In self-stabilization, simulation has been mainly investigated to emulate schedulers or to port solutions from a strong computational model to a weaker one. Awerbuch [7] introduced the concept of *synchronizer* in a non-self-stabilizing context. A synchronizer simulates a synchronous execution of an input algorithm into an asynchronous environment. The first two self-stabilizing synchronizers have been proposed in [4] for message-passing systems. Both solutions achieve a stabilization time in $O(D)$ rounds. The first solution is based on the previously mentioned unison, also proposed in the paper, that uses an infinite state space. To solve this latter issue, they then propose to mix it with the reset algorithm of [5] applied on links of a BFS spanning tree computed in $O(D)$ rounds. This reset algorithm is devoted, and so limits the approach, to locally checkable and locally correctable problems, and the BFS spanning tree construction uses a finite yet unbounded number of states per node and requires the presence of a distinguished node (a root). Again, the move complexity of their solutions is not analyzed. Awerbuch and Varghese [6] propose, still in the message-passing model, two synchronizers: the *rollback compiler* and the *resynchronizer*. The resynchronizer additionally requires the input algorithm to be locally checkable and assumes the knowledge of a common upper bound $\mathcal{D}$ on the network diameter. Using the rollback, resp. the resynchronizer, method, a synchronous non-self-stabilizing algorithm can be turned into an asynchronous self-stabilizing algorithm that stabilizes in $O(T)$ rounds, resp. $O(T + \mathcal{D})$ rounds, using $\Omega(T \times S)$ space, resp. $\Theta(S)$ space, per node where T, resp. S, is the execution time, resp. the space complexity, of the input algorithm. Again, the move complexity of these synchronizers is not analyzed. Now, the straightforward atomic-state model version of the rollback compiler is shown to achieve exponential move complexities in [21]. Finally, the synchronizer proposed in [21] works in the atomic-state model and achieves round and space complexities similar to those of the rollback

compiler, but additionally offers polynomial move complexity. Hence, it allows to design fully polynomial self-stabilizing solutions for static problems, but still with an important memory requirement (using $\Omega(T \times S)$ space).

Simulation has been also investigated in self-stabilization to emulate other schedulers. For example, the *conflict manager* proposed in [32] allows to emulate an unfair locally central scheduler in fully asynchronous settings. Another example is fairness that can be enforced using a unison algorithm together with the *cross-over* composition [8].

Concerning now model simulations, Turau proposes in [35] a general procedure allowing to simulate any algorithm for the distance-two atomic-state model in the (classical) distance-one atomic-state model assuming that nodes have unique identifiers. Finally, simulation from the atomic-state model to the link-register one and from the link-register model to the message-passing one are discussed in [26].

1.3 Contributions

Fully Polynomial Self-stabilizing Unison

We propose a fully polynomial self-stabilizing bounded-memory unison in the atomic-state model assuming a distributed unfair daemon. It works in any anonymous network of arbitrary connected topology, and stabilizes in $O(D)$ rounds and $O(n^3)$ moves using $\Theta(\log B)$ bits per node, where $B \geq 2D + 2$ (see Table 1 below). To the best of our knowledge, our algorithm vastly improves on the literature as other self-stabilizing algorithms have at least one of the following drawbacks: an unbounded memory, an $\Omega(n)$ round complexity, a restriction on the daemon (synchronous, fair, ...). Note also that the computational model we use is at least as general as the *stone age* model of Emek and Wattenhofer [29]: it does not require any local port labeling at nodes, or knowing how many neighbors a node has.

Overall, our unison achieves outstanding performance in terms of time, workload, and space, which also makes it the first fully polynomial self-stabilizing algorithm for a *dynamic* problem.

Self-stabilizing Synchronizer

From our unison algorithm, we straightforwardly derive a self-stabilizing synchronizer that efficiently simulates synchronous executions of an input self-stabilizing algorithm in an asynchronous environment. More precisely, if the input algorithm Alg_I is silent, then the output algorithm $Sync(Alg_I)$ is silent as well and satisfies the same specification as Alg_I. The specification preservation property also holds for any algorithm, silent or not, solving a static problem. We analyze the complexity of this synchronizer and show that it mostly preserves the round and space complexities of the simulated algorithm (see Table 1 for details). This synchronizer is thus a powerful tool to ease the design of efficient *asynchronous* self-stabilizing algorithms. Indeed, for many tasks, the usual lower bound on the stabilization time in rounds is $\Omega(D)$. Now, thanks to our unison, one just has to focus on the design of a *synchronous* $O(D)$-round self-stabilizing algorithm to finally obtain an asynchronous self-stabilizing solution asymptotically optimal in rounds, with a low overhead in space ($\Theta(\log B)$ bits per node) and a polynomial move complexity (i.e., a fully polynomial solution).

The transformer of [21] has similar round and move complexities. But this algorithm and ours are incomparable as they make different trade-offs. This paper prioritizes memory over generality, while the transformer of [21] makes the opposite choice by prioritizing generality over memory. More precisely, the transformer of [21] can simulate any synchronous algorithm (not necessarily self-stabilizing), by storing its whole execution. It thus has a much larger

space complexity than ours, which only stores two states of the simulated input algorithm. It turns out that the connections between our algorithm and the transformer of [21] are deeper than their move and round complexities. We further explain their similarities as well as their differences in Sections 3 and 4.5.

Implications of our Results

Using our synchronizer, one can easily obtain state-of-the-art (silent) self-stabilizing solutions for several fundamental distributed computing problems, e.g., BFS tree constructions, leader election, and clustering (see Table 1).

Table 1 Complexities of the Unison, the Synchronizer, and some consequences.

	Moves	Rounds	Space
Unison	$O\big(\min(n^2B, n^3)\big)$	$2D+2$	$\lceil \log B \rceil + 2$
Synchronizer	$O\big(\min(n^2B, n^3) + nT\big)$	$5D+3T$	$2M + \lceil \log B \rceil + 2$

Problem	Moves	Rounds	Space
BFS tree in rooted networks	$O(n^3)$	$O(D)$	$\Theta(\log B + \log \Delta)$
BFS tree in identified networks	$O(n^3)$	$O(D)$	$\Theta(\log N)$
Leader election	$O(n^3)$	$O(D)$	$\Theta(\log N)$
$O(\frac{n}{k})$-clustering	$O(n^3)$	$O(D)$	$\Theta(\log k + \log N)$

T and M are the synchronous time and space complexities of the input algorithm, and B and N are input parameters satisfying $B \geq 2D+2$ and $N \geq n$.

First, we obtain a new state-of-the-art asynchronous self-stabilizing algorithm for the BFS spanning tree construction in rooted and connected networks, by synchronizing the algorithm in [22] (which is a bounded-memory variant of the algorithm in [27]). This new algorithm converges in $O(n^3)$ moves and $O(D)$ rounds with $\Theta(\log B + \log \Delta)$ bits per node (the same round and space complexities as in [22]), where B is an upper bound on D and Δ is the maximum node degree. It improves both on the algorithm in [14], which only converges in $O(n^6)$ moves and $O(D^2)$ rounds, and on the algorithm in [21], which has similar complexities but uses $\Theta(B \cdot \log \Delta)$ bits per node.

In the following, we consider identified connected networks. In this setting, when nodes store identifiers, they usually know a bound k on the size of these identifiers. They thus know a bound $N = 2^k$ on n, and since N is a bound on D, we set $B = 2N + 2$.

In identified networks, a strategy to compute a BFS spanning tree is to compute a leader together with a BFS tree rooted at this leader. This is what the self-stabilizing algorithm in [33] actually does in a synchronous setting. Therefore, by synchronizing it, we obtain a new state-of-the-art asynchronous self-stabilizing algorithm for both the leader election and the BFS spanning tree construction in identified and connected networks. This new algorithm converges in $O(n^3)$ moves and $O(D)$ rounds with $\Theta(\log N)$ bits per node (i.e., the same round and space complexities as in [33]). To the best of our knowledge, no such efficient solutions exist until now in the literature. There are two incomparable asynchronous self-stabilizing algorithms that achieve an $O(D)$ round complexity [12, 1]. They operate in weaker models (resp. message-passing and link-register). However, their move complexity is not analyzed and the first one has a $\Theta(\log B \cdot \log N)$ space requirement (B being a known upper bound on D) while the second one uses an unbounded space.

Other memory-efficient fully polynomial self-stabilizing solutions can be easily obtained with our synchronizer, e.g., to compute the median or centers in anonymous trees by simulating algorithms proposed in [18]. Another application of our synchronizer is to remove fairness assumptions along with obtaining good complexities. For example, the silent self-stabilizing algorithm proposed in [16] computes a clustering of $O(\frac{n}{k})$ clusters in any rooted identified connected network. It assumes a distributed weakly fair daemon and its move complexity is unknown. With our synchronizer,[3] we achieve a fully polynomial silent solution that stabilizes under the distributed unfair daemon and without the rooted network assumption, in $O(D)$ rounds, $O(n^3)$ moves, and using $\Theta(\log k + \log N)$ bits per node.

Note that, by using the compiler in [21], one can obtain similar time complexities for all the previous problems, but with a drastically higher space usage.

1.4 Roadmap

The rest of the paper is organized as follows. Section 2 is dedicated to the computational model and the basic definitions. We develop the links between the present paper and [21] in Section 3, and we present our algorithm in Section 4. We sketch its correctness and its time complexity in Section 5. In Section 6, the self-stabilizing synchronizer derived from our unison algorithm is presented and its complexity is also sketched. We conclude in Section 7.

2 Preliminaries

2.1 Networks

We model *distributed systems* as simple graphs, that is, pairs $G = (V, E)$ where V is a set of *nodes* and E is a set of *edges* representing communication links. We assume that communications are bidirectional. The set $N(p) = \{q \mid \{p, q\} \in E\}$ is the set of *neighbors* of p, with which p can communicate, and $N[p] = N(p) \cup \{p\}$ is the closed neighborhood of p. A *path* (*from* p_0 *to* p_l) of *length* l is a sequence $P = p_0 p_1 \cdots p_l$ of nodes such that consecutive nodes in P are neighbors. We assume that G is connected, meaning that any two nodes are connected by a path. We can thus define the *distance* $d(p, q)$ between two nodes p and q to be the minimum length of a path from p to q. The *diameter* D of G is then the maximum distance between nodes of G.

2.2 Computational Model: the Atomic-state Model

Our unison algorithm works in a variant of the *atomic-state model* in which each node holds locally shared registers, called *variables*, whose values constitute its *state*. The vector of all node states defines a *configuration* of the system.

An algorithm consists of a finite set of *rules* of the form $label : guard \rightarrow action$. In the variant that we consider, a *guard* is a boolean predicate on the state of the node and on the set of states of its neighbors. The *action* changes the state of the node. To shorten guards and increase readability, priorities between rules may be set. A rule whose guard is *true* is *enabled*, and can be executed. By extension, a node with at least one enabled rule is also *enabled*, and $Enabled(\gamma)$ contains the enabled nodes in a configuration γ. Note that

[3] Also replacing the spanning tree construction used in [16] by the new BFS tree construction of the previous paragraph.

this model is quite weak. Indeed, in other variants, nodes may have, for example, distinct identifiers. In our case, the network is anonymous and since a node only accesses a set of states, it cannot even count how many neighbors it has.

An *execution* in this model is a maximal sequence of configurations $e = \gamma^0\gamma^1 \cdots \gamma^i \cdots$ such that for each transition (called *step*) $\gamma^i \mapsto \gamma^{i+1}$, there is a nonempty subset $\mathcal{X}^i$ of $Enabled(\gamma^i)$ whose nodes simultaneously and atomically execute one of their enabled rules, leading from γ^i to γ^{i+1}. We say that each node of $\mathcal{X}^i$ executes a *move* during $\gamma^i \mapsto \gamma^{i+1}$. Note that e is either infinite, or ends at a *terminal* configuration γ^f where $Enabled(\gamma^f) = \emptyset$. An algorithm with no infinite executions is *terminating* or *silent*.

A *daemon* $\mathfrak{D}$ is a predicate over executions. An execution which satisfies $\mathfrak{D}$ is said to be *an execution under* $\mathfrak{D}$. We consider the *synchronous* daemon, which is true if, at all steps, $\mathcal{X}^i := Enabled(\gamma^i)$, and the *fully asynchronous daemon*, also called *distributed unfair daemon* in the literature, which is always true. Note that under the distributed unfair daemon, a node may starve and may never be activated, unless it is the only enabled node.

In an execution, all the information in the states is not necessarily relevant for a problem. We thus use a *projection* to extract information (e.g., just an output boolean for the boolean consensus) from a node's state, and we canonically extend this projection to configurations and executions. A *specification* of a distributed problem is then a predicate over projected executions. A problem is *static* if its specification requires the projected executions to be constant, and it is *dynamic* otherwise.

An algorithm is *self-stabilizing* under a daemon $\mathfrak{D}$ if, for every network and input parameters, there exists a set of *legitimate* configurations such that (1) the algorithm *converges*, i.e., every execution under $\mathfrak{D}$ (starting from an arbitrary configuration) contains a legitimate configuration, and (2) the algorithm is *correct*, i.e., every execution under $\mathfrak{D}$ that starts from a legitimate configuration satisfies the specification.

We consider three complexity measures: *space*, *moves* which model the total workload, and *rounds* which model an analogous of the synchronous time by taking the speed of the slowest nodes into account. As done in the literature on the atomic-state model, the space complexity is the maximum space used by one node to store its own variables. As explained before, a move is the execution of a rule by a node. To define the round complexity of an execution $e = \gamma^0\gamma^1 \cdots$, we first need to define the notion of *neutralization*: a node p is *neutralized* in $\gamma^i \mapsto \gamma^{i+1}$, if p is enabled in γ^i and not in γ^{i+1}, but it does not apply any rule in $\gamma^i \mapsto \gamma^{i+1}$. 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' such that every node that is enabled in γ^0 either executes a move or is neutralized during a step of e'. If e' is finite, then let e'' be the suffix of e that starts from the last configuration of e'; the second round of e is the first round of e'', and so on. For every $i > 0$, we denote by γ^{r_i} the last configuration of the i-th round of e, if it exists and is finite; we also conventionally let $\gamma^{r_0} = \gamma^0$. Consequently, $\gamma^{r_{i-1}}$ is also the first configuration of the i-th round of e. The *stabilization time* of a self-stabilizing algorithm is the maximum time (in moves or rounds) over every execution possible under the considered daemon (starting from any initial configuration) to reach (for the first time) a legitimate configuration.

3 A Glimpse of our Research Process

3.1 An Unbounded Unison Algorithm

We started this work on the bounded unison problem when we observed that an unbounded solution can easily be derived from [21]. This can be seen as follows. The algorithm given in [21] simulates a synchronous non self-stabilizing algorithm in an asynchronous

self-stabilizing setting. To do so, it uses a very natural idea. It stores, at each node, the whole execution of the algorithm so far as a list of states. Given its list and the lists of its neighbors, a given node can check for inconsistencies in the simulation and correct them.

Now if we implement this idea in an asynchronous algorithm which is not self-stabilizing, then the length of the lists satisfy the unison property. Indeed, to compute its $(i+1)$-th value, a node must wait for all its neighbors to have computed at least their i-th value.

Obviously, in a self-stabilizing setting, we cannot expect the length of the lists of the nodes to initially satisfy the unison property. It turns out that the error recovery mechanism in [21] not only solves the initial inconsistencies of the simulation, but also recovers the unison property.

If we simulate an algorithm "that does nothing", we can compress the lists by only storing their lengths. We thus obtain a first (unbounded) unison algorithm, given below. Note that although we describe the whole algorithm, the reader does not need to fully understand it.

Each node p has a status $p.s \in \{E, C\}$ (Error/Correct) and a time $p.t \in \mathbb{N}$. Given these predicates,

$$\begin{aligned}
root(p) &:= \big(p.s = E \wedge \neg(\exists q \in N(p),\ q.s = E \wedge q.t < p.t)\big) \vee \\
&\quad\ \big(p.s = C \wedge \exists q \in N(p),\ (q.t \geq p.t + 2)\big) \\
activeRoot(p) &:= root(p) \wedge (p.t > 0 \vee p.s = C) \\
errProp(p,i) &:= \exists q \in N(p),\ q.s = E \wedge q.t < i < p.t \\
canClearE(p) &:= p.s = E \wedge \forall q \in N(p),\ \big(|q.t - p.t| \leq 1 \wedge (q.t \leq p.t \vee q.s = C)\big) \\
updatable(p) &:= p.s = C \wedge \big(\forall q \in N(p),\ p.t \leq q.t \leq p.t + 1\big)
\end{aligned}$$

the algorithm is defined by the following four rules

$$\begin{aligned}
R_R :&\quad activeRoot(p) &\longrightarrow&\quad p.t := 0\ ;\ p.s := E \\
R_P(i) :&\quad errProp(p,i) &\longrightarrow&\quad p.t := i\ ;\ p.s := E \\
R_C :&\quad canClearE(p) &\longrightarrow&\quad p.s := C \\
R_U :&\quad updatable(p) &\longrightarrow&\quad p.t := p.t + 1
\end{aligned}$$

in which R_R has the highest priority, and $R_P(i)$ has a higher priority than $R_P(i')$ for $i < i'$. The rules R_R, $R_P(i)$ and R_C are "error management" rules. Thus, once the algorithm has stabilized, the status of all nodes is C and only R_U is applicable.

This unbounded self-stabilizing unison algorithm is not really interesting by itself. Indeed, it converges in $2D+2$ rounds in an asynchronous setting, but in this regard, the algorithm in [4] converges twice as fast, is simpler and operates in the message-passing model, which is more realistic. However, whereas nobody has been able to derive a bounded version of the algorithm in [4], we hoped that this could be done with this new algorithm.

In the following subsections, we present a first very natural attempt, which ultimately failed, and a more complex version, which we detail and prove in the next sections of the paper.

3.2 A Failed Bounded Unison Algorithm

The most natural strategy to turn an unbounded unison into a bounded one is simply to count modulo a large enough fixed bound B. To outline this change of paradigm from an ever-increasing time to a circular clock, we rename the variable $p.t$ into $p.c$ for any node p.

We thus modify the rule R_U as follows:

$$updatable(p) := p.s = C \wedge \big(\forall q \in N(p),\ q.c \in \{p.c, p.c+1 \bmod B\}\big)$$
$$R_U : updatable(p) \longrightarrow p.c := p.c + 1 \bmod B$$

At first glance, we do not need to modify the other rules as their purpose is only to correct errors, but this intuition is wrong. Indeed, when two neighboring nodes p and q are such that $p.s = q.s = C$, $p.c = 0$ and $q.c = B - 1$, they satisfy the unison property, but p can apply the rule R_R, although there are no errors to correct. The problem comes from the term $\exists q \in N(p),\ (q.c \geq p.c + 2)$ in the *root* predicate which should detect out-of-sync neighbors. Hence, we must at least modify this predicate as follows:

$$root(p) := \big(p.s = E \wedge \neg(\exists q \in N(p),\ q.s = E \wedge q.c < p.c)\big) \vee$$
$$\big(p.s = C \wedge \exists q \in N(p),\ (q.c \geq p.c + 2) \wedge \underbrace{\neg(p.c = 0 \wedge q.c = B - 1)}\big).$$

Therefore, transforming the algorithm to implement this simple modulo-B idea is already not as straightforward as it may seem.

Moreover, even small modifications generally introduce new unforeseen behaviors, and the modified algorithm has no particular reasons to be efficient, or even correct. As a matter of fact, we failed to prove its correctness. To understand why, we must delve a bit into the proof scheme of [21].

An important observation is that rootless configurations (i.e., those without nodes satisfying the *root* predicate) satisfy the safety property of the unison. In [21], the correctness and the move complexity then follow from the key property that roots cannot be created, and that, in a "small" number of steps, at least one root disappears.

Sadly, this first attempt algorithm does not satisfy the "no root creations" property. To see this, consider a path $p - q - r$ and a configuration γ^a in which $p.c = q.c = B - 1$, $r.c = 3$, $p.s = q.s = C$ and $r.s = E$. In one step $\gamma^a \mapsto \gamma^b$,

- p applies the rule R_U and thus, in γ^b, $p.c = 0$ and $p.s = C$
- q applies the rule $R_P(4)$, and thus, in γ^b, $q.c = 4$.

Therefore, in γ^b, $p.s = C$ and p has a neighbor q such that $q.c \geq p.c + 2$ and $q.c \neq B - 1$. Thus, p is a root in γ^b, although it is not one in γ^a.

Note that the fact that roots can be created is not necessarily a problem. Indeed, if only a finite number of them appears, we recover the correctness of the algorithm. We actually believe that, for B large enough, any node can become a root only once per execution, and this would most likely imply that the move complexity remains polynomial. But n roots may appear sequentially, which would lead to an $\Omega(n)$ round complexity.

At this point, we cannot rule out that this algorithm is correct and has good properties. However, because of these problems, we took another approach.

3.3 Our Solution

In the end, our solution is obtained by a rather limited modification of the previous algorithm: we extend the range of the counters $p.c$ to the interval $[-B, B)$, but we restrict their range to $[-B, 0)$ when $p.s = E$.

Actually, this modification prevents *all* root creations. But, as with the previous attempt, we must be extra careful even with the smallest change, as proofs can easily break. We thus present the whole algorithm and its proofs in more details in the next sections, and further highlight the differences with [21] in Section 4.5.

4 A Unison Algorithm

4.1 Data Structures

Let $B \geq 2D+2$ be an integer. Each node p maintains a single variable $p.v \in \{(C,x) \mid x \in [-B,B)\} \cup \{(E,x) \mid x \in [-B,0)\}$. In the algorithm, $p.s$ and $p.c$, the *status* and the *clock* of p, respectively denote the left and right part of $p.v$. An assignment to $p.s$ or $p.c$ modifies the corresponding field of $p.v$.

We define the unison increment $a \oplus_B 1$ as $(B-1) \oplus_B 1 = 0$ and $a \oplus_B 1 = a+1$ if $a \in [-B, B-2]$. Two clocks are *synchronized* if they are at most one increment apart. We then define $a \oplus_B b$ as the result of b iterations of $\oplus_B 1$ over a. Note that, as hinted in Section 3.2, we also use the usual addition and subtraction.

4.2 Some Predicates

Apart from its state, a node p has only access to the set $\{q.v \mid q \in N(p)\}$ of its neighbors' variables. A guard should thus not contain a direct reference to a neighbor q of p. This may look like a problem for we have already used such references. Nevertheless, these uses are legitimate as, for any predicate Pred, the semantics of $\exists (s,c) \in \{q.v \mid q \in N(p)\},\ \mathrm{Pred}(s,c)$ is precisely $\exists q \in N(p),\ \mathrm{Pred}(q.s, q.c)$. We can similarly encode universal statements.

As a matter of fact, we use the following shortcuts to increase readability:

`Shortcut1` $\quad \exists q \in N(p),\ \mathrm{Pred}(q.s,q.c) := \exists (s,c) \in \{q.v \mid q \in N(p)\},\ \mathrm{Pred}(s,c)$

`Shortcut2` $\quad \forall q \in N(p),\ \mathrm{Pred}(q.s,q.c) := \forall (s,c) \in \{q.v \mid q \in N(p)\},\ \mathrm{Pred}(s,c)$

Below, we define the predicates used by our algorithm.

$$\begin{aligned} \mathit{root}(p) &:= \big(p.s = E \wedge \neg(\exists q \in N(p),\ q.s = E \wedge q.c < p.c)\big) \vee \\ &\qquad \big(p.s = C \wedge \exists q \in N(p),\ (q.c \geq p.c+2) \wedge \neg(p.c = 0 \wedge q.c = B-1)\big) \\ \mathit{activeRoot}(p) &:= \mathit{root}(p) \wedge (p.c \neq -B \vee p.s = C) \\ \mathit{errorPropag}(p,i) &:= i < 0 \wedge \exists q \in N(p),\ q.s = E \wedge q.c < i < p.c \\ \mathit{canClearE}(p) &:= p.s = E \wedge \forall q \in N(p),\ \big(|q.c - p.c| \leq 1 \wedge (q.c \leq p.c \vee q.s = C)\big) \\ \mathit{updatable}(p) &:= p.s = C \wedge \forall q \in N(p),\ q.c \in \{p.c, p.c \oplus_B 1\} \end{aligned}$$

A node p is a *root* if $\mathit{root}(p)$. An *error rule* is either the rule R_R or a rule $R_P(i)$.

4.3 The Algorithm

A unison algorithm is rarely used alone. It is merely a tool to drive another algorithm. It thus makes sense that our algorithm depends on some properties which are external to the unison algorithm and its variables. Our algorithm uses a predicate P_{aux} which is not yet defined. As a matter of fact, its influence on the complexity analysis of the algorithm is very limited. To prove the correctness of the unison, we set $P_{\mathrm{aux}} = \mathit{true}$, and we specialize P_{aux} differently in Section 6 when using our algorithm as a synchronizer.

$$\begin{aligned} R_R :&\quad \mathit{activeRoot}(p) &&\longrightarrow\quad p.c := -B\ ;\ p.s = E \\ R_P(i) :&\quad \mathit{errorPropag}(p,i) &&\longrightarrow\quad p.c := i\ ;\ p.s = E \\ R_C :&\quad \mathit{canClearE}(p) &&\longrightarrow\quad p.s := C \\ R_U :&\quad \mathit{updatable}(p) \wedge P_{\mathrm{aux}}(p) &&\longrightarrow\quad p.c := p.c \oplus_B 1 \end{aligned}$$

The rule R_R has the highest priority, and $R_P(i)$ has a higher priority than $R_P(i')$ for $i < i'$.

4.4 An Overview of the Algorithm

Contrary to [4] which proceeds by only locally synchronizing out-of-sync clocks, i.e., the clocks of two neighboring nodes that differ by at least two increments, we organize the synchronizations in *error broadcasts*. Every node p involved in such a broadcast is *in error* and its status is $p.s = E$. Otherwise, it is *correct* and $p.s = C$.

If p is correct, in sync with its neighbors, and if its clock $p.c$ is a local minimum, then p can apply the rule R_U to increment its clock.

There is a *cliff* between r and one of its neighbors p if their clocks are out-of-sync and $p.c > r.c$. If r is correct and has a cliff with a neighbor, then r is said to be a *root* and should initiate an error broadcast by applying the rule R_R, which respectively sets $r.c$ to $-B$ and $r.s$ to E.

If there is a cliff between r and p, r is in error, and $r.c < -1$, then p should propagate the broadcast by applying the rule R_P which sets $p.c$ to $r.c + 1$ and $p.s$ to E. If p has several such neighbors r, it applies R_P according to the one with the minimum clock.

As a consequence, any node p in error with $p.c > -B$ should have at least one neighbor in error with a smaller clock. This way, the structure of an error broadcast is a *dag* (directed acyclic graph). We therefore extend the definition of root to include nodes in error with no "parents" in the broadcast dag.

Note that a node may decrease its clock multiple times using R_P, and in doing so may consecutively join several error dags or several parts of them. This way, nodes reduce the height of the error dags, which is a key element to achieve the $O(D)$-round complexity. Furthermore, any node in error eventually has a clock smaller than $-B + D$ and all cliffs are eventually destroyed.

Finally, if p is in error, is not involved in any cliff (in which case an error must be propagated), and if all its neighbors with larger clocks are correct, then the broadcast from p is finished, and p can apply the rule R_C to switch back $p.s$ to C.

A key element to bound the move complexity is that a dag built during an error broadcast is cleaned from the larger clocks to the smaller, but nodes previously in the dag resume the "normal" increments (using the rule R_U) in the reverse order (i.e., from the smaller clocks to the larger). Indeed, a non-root node in an error broadcast is one increment ahead of its parents in the dag and so has to wait for their increment before being able to perform one itself. Hence, the first node in the dag that makes a R_U move after an error broadcast is its root.

4.5 Some Subtleties

Some statements and the corresponding proof arguments are very similar to the ones of [21] (rather its arXiv version [20]). However, the fact that the algorithm and its data structures are different imply that proofs are indeed different. As a matter of fact, we have tried but failed to unify both algorithms into a *natural* more general one.

Below, we outline subtleties which are specific to our algorithm.

- Since nodes in error are restricted to negative clocks, it is natural to expect that legitimate configurations require all clocks to be non-negative. This would suggest a $\Theta(B)$ round complexity, which is weaker than what we claim. But this intuition is false. For example, the configuration where all nodes are correct and all clocks are set to $-B$ is legitimate. This is one of the reasons for our $O(D)$ round complexity.
- In the unbounded unison algorithm above which we derive from [21], whenever two neighboring nodes p and q are such that $q.s = E$ and $p.c \geq q.c + 2$, the node p can always apply a rule $R_P(i)$. In our algorithm, this is not the case when $q.c = -1$. This could

introduce unexpected behaviors which could impact the complexities of our algorithm, or in the worst case, lead to deadlocks. We thus have to deal with this slight difference in the proofs.
- In [21], the proofs heavily rely on the fact that the counters increase when applying the rule R_U while they decrease when applying the rules R_R and $R_P(i)$. This monotony property is however not true in our setting. More generally, having two addition operators $+$ and $\oplus_B$ requires special care throughout the proofs.
- Finally, to bound the memory, the maximum clock is $B-1$, after which clocks go back to 0. Notice that to ensure the liveness property of the unison, we must have $B \geq 2D+2$ (an example of deadlock is presented for $B = 2D+1$ in Subsection 5.2).

5 Self-Stabilization and Complexity of the Unison Algorithm

As already mentioned, the unison algorithm corresponds to $P_{\text{aux}} = \mathit{true}$. However, since most proofs are valid regardless of the definition of P_{aux}, we only specify it when needed. We define the *legitimate* configurations as the configurations without roots. Let $e = \gamma^0\gamma^1\cdots$ be an execution. We respectively denote by $p.s^i$ and $p.c^i$ the value of $p.s$ and $p.c$ in γ^i.

5.1 Convergence and Move Complexity of the Unison Algorithm

Although it is tedious, it is straightforward to prove, by case analysis, that roots cannot be created. Since roots are obstructions to legitimate configurations, it is natural to partition the steps of e into *segments* such that each step in which at least one root disappears is the last step of a segment. There are thus at most n segments with roots, which constitute the *stabilization phase*, and at most one root-less segment. We now show that the stabilization phase is finite by providing a (finite) bound on its move complexity.

In the following, s is any segment of the stabilization phase. The key fact is that in s, a node p in error cannot apply the rule R_U until the end of s. We prove this by induction on $p.c$. If $p.c = -B$, then p is a root, and the only rule that p can apply is R_C, which removes its root status. The base case thus follows. Now let p be in error with $p.c > -B$. If p does not move in s, then our claim holds. Otherwise let $\gamma^a \mapsto \gamma^b$ be the first step in which p moves. If p applies the rule R_R, then $p.c^b = -B$, and for the remainder of s, the claim holds by induction. Otherwise, p has a neighbor q such that $q.s^a = E$ and $q.c^a < p.c^a < 0$. By induction, $q.c$ cannot increase until the end of s. As long as $p.c > q.c$, p cannot apply the rule R_U and if, at some point, $p.c \leq q.c$, then p must have applied an error rule, thus decreasing its clock, at which point the claim holds by induction.

Since roots cannot be created, the number of R_R-moves is at most n. Moreover, since between two R_C-moves, there has to be at least one error move (R_R- or R_P-move), we have $\#R_C$-moves $\leq n + \#R_R$-moves $+ \#R_P$-moves $\leq 2n + \#R_P$-moves. We thus only need to bound the number of R_U-moves and R_P-moves.

We now bound the number of R_U moves by a node in s. If a node q does not move between γ^a and γ^b in s with $a < b$, then a neighbor p can apply the rule R_U at most twice, to go from $q.c^a - 1$ to $q.c^a + 1$. More generally, if $p.c^b \geq p.c^a + 2 + i$ (we really mean the $+$ operator and not the $\oplus_B$ operator), then every neighbor q of p must increase its clock by at least i between γ^a and γ^b. By induction on d, if $p.c^b = p.c^a + 2d + i$, then every node q at distance d from p increases its clock by at least i between γ^a and γ^b. Since roots cannot increase their clocks, this implies that $p.c^b \leq p.c^a + 2D$.

From this "linear" bound, we now derive a "circular" bound which takes into account the fact that the clock of a node may decrease while applying the rule R_U (from $B-1$ to 0). In the worst case, p could apply the rule R_U $2D$ times to reach $p.c = B-1$, then apply

R_U once so that $p.c = 0$, then reapply R_U $2D$ more times (recall that $B \geq 2D + 2$). To summarize, p may apply R_U at most $4D + 1$ times in s. This gives an $O(n^2 D)$ bound on the number of R_U-moves done during the stabilization phase.

We now focus on the rule R_P in s. If a node p_0 applies a rule R_P in a step $\gamma^{j_1} \mapsto \gamma^{j_1+1}$ of s, it does so to "connect" to a neighbor p_1 which is already in error. Now p_1 may be in error in γ^{j_1} because it has applied a rule R_P in another step $\gamma^{j_2} \mapsto \gamma^{j_2+1}$ of s with $j_2 < j_1$, to connect to a neighbor p_2, and so on. This defines a *causality chain* $p_0 \cdots p_l$ for some l. Since, according to the key fact, rules R_P and R_U do not alternate in s, a node cannot appear twice in the causality chain, thus $l < n$. Moreover, when considering a maximal causality chain, $p_l.c^{j_l}$ is either the value of $p_l.c$ at the beginning of s, or $-B$ if p_l has applied the rule R_R. The clock $p_0.c$ can thus take at most $n(n + 1)$ distinct values in s, which implies that the rule R_P is applied at most $O(n^2)$ times in s by a given node. This gives an overall $O(n^4)$-bound on the number of rules R_P. Note that a more careful analysis gives an overall bound of $O(n^3)$ on the total number of R_P-moves.

We can also easily obtain a bound that involves B. Indeed, a node p has at most B R_P-moves and $4D + 1 = O(B)$ R_U-moves in s. This gives an $O(n^2 B)$-bound on the number of moves. To summarize, the stabilization phase terminates after at most $O(\min(n^3, n^2 B))$ moves.

Note that any configuration γ with at least one root contains at least one enabled node. Indeed, if any two neighboring clocks are at most one increment apart, then any root is in error, and the rule R_C is enabled at any node p in error with $p.c$ maximum. Otherwise, there exist two neighbors p and q such that $p.c$ and $q.c$ are more than one increment apart. We choose them with $q.c < p.c$ and $q.c$ minimum. $q.c$ being minimum, we can show that either q is a root that is enabled for R_R, or p can apply the rule R_P because q is in error and satisfies $q.c \leq -B + D < -1$. Thus, the last configuration of the stabilization phase is legitimate.

Also, note that since roots cannot be created, being legitimate is a closed property, meaning that in a step $\gamma^a \mapsto \gamma^b$, if γ^a is legitimate, then so is γ^b.

5.2 Correctness of the Unison Algorithm

We now show that any legitimate configuration γ satisfies the safety property of the unison. First, γ cannot contain nodes in error, because any such node p with $p.c$ minimum would be a root. Moreover, if the clocks of two correct neighbors differ by more than one increment, then the node with the smaller clock is a root.

To prove the liveness property of the unison, we set $P_{\text{aux}} = \mathit{true}$ in this paragraph. In legitimate configurations, since neighboring clocks differ by at most one increment, any two clocks differ by at most D increments. And since $B \geq 2D + 2$, there exists $c \in [0, B)$ which is not the clock of any node. This implies that there exists at least one node p whose clock is not the increment of any other clock. Thus, p satisfies *updatable* and can apply R_U. This proves that at least one node applies R_U infinitely often, and thus so do all nodes. Observe that $B \geq 2D + 2$ is tight. Indeed, when $B = 2D + 1$, the configuration of the cycle p_0, p_1, $\ldots p_{2D}$ in which all nodes are correct and $p_i.c = i$, is legitimate but is terminal.

5.3 Round Complexity of the Unison Algorithm

We claim that $\gamma^{r_{2D+2}}$ contains no roots and so is legitimate. Recall that for all $i \geq 1$, Round i is $\gamma^{r_{i-1}} \cdots \gamma^{r_i}$. We suppose that all γ^{r_i} with $i \leq 2D + 1$ contain roots otherwise our claim directly holds.

We now study the first $D+1$ rounds. Let r be any root in $\gamma^{r_{D+1}}$. Since there are no root creations, r is already a root in γ^0. By the end of the first round (using the rule R_R if needed), $r.c = -B$ and $r.s = E$. Now, since r is still a root in $\gamma^{r_{D+1}}$, it cannot make a move in the meantime, and its state does not change until $\gamma^{r_{D+1}}$. Furthermore, every neighbor p of r such that $p.c > -B+1$ can apply the rule R_P. So, by the end of Round 2, and as long as r does not increment its clock, $p.c \leq -B+1$. By induction on the distance $d(p,r)$ between p and r, we can prove that $p.c^{r_{D+1}} \leq -B + d(p,r)$ for every node p and every root r in $\gamma^{r_{D+1}}$.

We claim that $\gamma^{r_{D+1}}$ does not contain any *cliff*, i.e., a pair (q,p) of neighboring nodes whose clocks are out-of-sync and such that $q.c < p.c$. Suppose that (q,p) is a cliff in $\gamma^{r_{D+1}}$. The node q is in error as otherwise it would be a root not in error, which, as already mentioned, is impossible from γ^{r_1}. Moreover, we can prove by induction on $q.c$ that there is a root r in $\gamma^{r_{D+1}}$ such that $q.c \geq -B + d(q,r)$. Since $p.c \geq q.c + 2$, we have $p.c > -B + d(p,r)$, which contradicts the result of the previous paragraph.

We now consider the next $D+1$ rounds. Since $\gamma^{r_{D+1}}$ contains no nodes which can apply R_R, and no cliffs, nodes can only apply the rules R_U or R_C. Furthermore, among nodes in error, those with the largest clock can apply the rule R_C, which implies that roots no longer exist by the end of Round $2D+2$, and thus $\gamma^{r_{2D+2}}$ is legitimate.

6 A Synchronizer

Let us consider a variant of the atomic-state model which is at least as expressive as the model of our unison algorithm. This means that, in this model, we should be able to encode the shortcuts `Shortcut1` and `Shortcut2` (defined page 10).

In this model, let Alg_I be any silent algorithm which is self-stabilizing with a projection $proj$ for a static specification SP under the synchronous daemon. Using folklore ideas (see, e.g., [4] and [28]), we define in this section a synchronizer which uses our unison to transform Alg_I into an algorithm $Sync(Alg_I)$ which "simulates" synchronous executions of Alg_I in an asynchronous environment under a distributed unfair daemon.

6.1 The Synchronized Algorithm

On top of its unison variables, each node p stores two states of Alg_I, in the variables $p.old$ and $p.curr$. These variables ought to contain the last two states of p in a synchronous execution of Alg_I. When p applies the rule R_U, it also computes a next state of Alg_I. It does so by applying the function $\widehat{Alg_I}$ which selects $p.curr$ and, for each neighbor q, the variable $q.curr$ if $p.c = q.c$, and $q.old$ if $q.c = p.c \oplus_B 1$, and applies Alg_I on these values. We thus modify the rule R_U in the following way:

$$R_U : updatable(p) \wedge P_{\text{aux}}(p) \longrightarrow p.old := p.curr;\ p.curr := \widehat{Alg_I}(p);\ p.c := p.c \oplus_B 1.$$

The folklore algorithm corresponds to the case when $P_{\text{aux}}(p)$ is always *true*. In this case, the clocks of the unison constantly change. Thus, even if Alg_I is silent, its simulation is not. To obtain a silent simulation, we devise another strategy by defining $P_{\text{aux}}(p)$ as follows.

$$P_{\text{aux}}(p) = (\widehat{Alg_I}(p) \neq p.curr) \vee (\exists q \in N(p), q.c = p.c \oplus_B 1).$$

We define the legitimate configurations of $Sync(Alg_I)$ to be its terminal configurations. In the next sections, we sketch the proof that $Sync(Alg_I)$ is self-stabilizing for the same specification as Alg_I. As a matter of fact, our result is more general as the silent assumption is not necessary (we need a different definition for the legitimate configurations though).

6.2 Convergence and Move Complexity of the Synchronized Algorithm

In everyday life, we have a distinction between the value of a clock (modulo 24 hours) and the time. Both are obviously linked. We would like to make a similar distinction here. Let $e = \gamma^0\gamma^1\cdots$ be an execution of legitimate configurations. Since γ^0 is legitimate, every two neighboring clocks differ by at most one increment.

Since $B \geq 2D+2$, as already mentioned, at least one element of $[0, B)$ is the clock of no nodes in γ^0. This implies that there is a node x such that $x.c^0 \oplus_B 1$ is not the clock of any node. We extend this local synchronization property by uniquely defining $time(p)^0 \in [-D, 0]$ by (1) $time(x)^0 = 0$, (2) if $p.c^0 = q.c^0$, then $time(p)^0 = time(q)^0$, and (3) if $p.c \oplus_B 1 = q.c$, then $time(p)^0 = time(q)^0 - 1$. Moreover, we also define $time(p)^{i+1} = time(p)^i + 1$ if p applies R_U in $\gamma^i \mapsto \gamma^{i+1}$, and $time(p)^{i+1} = time(p)^i$ otherwise.

For any i, $j \geq 0$ such that $time(p)^j = i$, we set $\mathtt{st}_p^i := p.curr^j$. When $\mathtt{st}_p^i$ is defined for all p, let λ^i be the configuration in which the state of each node p is $\mathtt{st}_p^i$. A careful analysis shows that, by definition of P_{aux}, λ^i exists as soon as some $\mathtt{st}_p^i$ does, and $\Lambda = \lambda^0\lambda^1\cdots$ is precisely the synchronous execution of Alg_I from λ^0.

Suppose that T is a bound on the number of rounds that Alg_I needs to reach silence. Thus, $\Lambda = \lambda^0\cdots\lambda^H$ for some $H \leq T$. In the simulation phase, a node makes at most D moves to have a non-negative time, and then at most T moves to finish the simulation. Together with the stabilization time of the unison, our simulated algorithm is also silent with an $O(\min(n^3, n^2B) + nD + nT) = O(\min(n^3, n^2B) + nT)$ move complexity.

6.3 Correctness of the Synchronized Algorithm

In $Sync(Alg_I)$, we define the *restriction* $rest(s)$ of the state s of any node p to be $p.curr$, and we canonically extend $rest$ to configurations and executions. Let us consider a legitimate configuration γ of $Sync(Alg_I)$. This configuration is terminal, and therefore there exists a unique execution e of $Sync(Alg_I)$ starting at γ (the one restricted to γ alone). Besides, since γ is terminal, its restriction is terminal too (for Alg_I). Therefore $rest(\gamma)$ is legitimate, and $proj(rest(e))$ satisfies the specification SP. Hence, the algorithm $Sync(Alg_I)$ also satisfies SP (for the projection $proj \circ rest$).

6.4 Round Complexity of the Synchronized Algorithm

The round complexity is analyzed by considering two stages: a first stage to have all times non-negative, and a second stage to have all times equal to H.

To give an intuition of our proof, as it is the more complex, we first consider the second stage. Figure 1 is an illustration of the following explanation. Suppose that all times are 0 in γ^0, and only s_1 is such that $\mathtt{st}_{s_1}^0 \neq \mathtt{st}_{s_1}^1$. In the first round of the synchronous execution, s_1 applies R_U, and then, after each new round, nodes at distance 1 from s_1, then 2, and so on will increase their time to 1. Now suppose that only $s_2 \in N[s_1]$ is such that $\mathtt{st}_{s_2}^1 \neq \mathtt{st}_{s_2}^2$. As soon as all nodes in $N[s_2]$ have a time of 1, s_2 applies R_U. This happens at Round 3 if $s_2 = s_1$ and at Round 4 otherwise. After this, after each new round, nodes at distance 1 from s_2, then 2, and so on will increase their time to 2. If we consider some $s_3 \in N[s_2]$, and so on, then s_i increases its time to i at Round at most $3i - 2$, and all nodes do so at Round at most $3i + D - 2$. If nodes increase their time earlier, this only speed up the process.

Now, by definition of H, there is a node s_H whose state changes between λ_{H-1} and λ_H. If the states of all nodes in $N[p]$ were the same in λ_{H-2} and λ_{H-1}, then s_H would not have changed its state between λ_{H-1} and λ_H. There thus exists $s_{H-1} \in N[s_H]$ that changes its state between λ_{H-2} and λ_{H-1}. By repeating this process, we can prove that, unless $H = 0$,

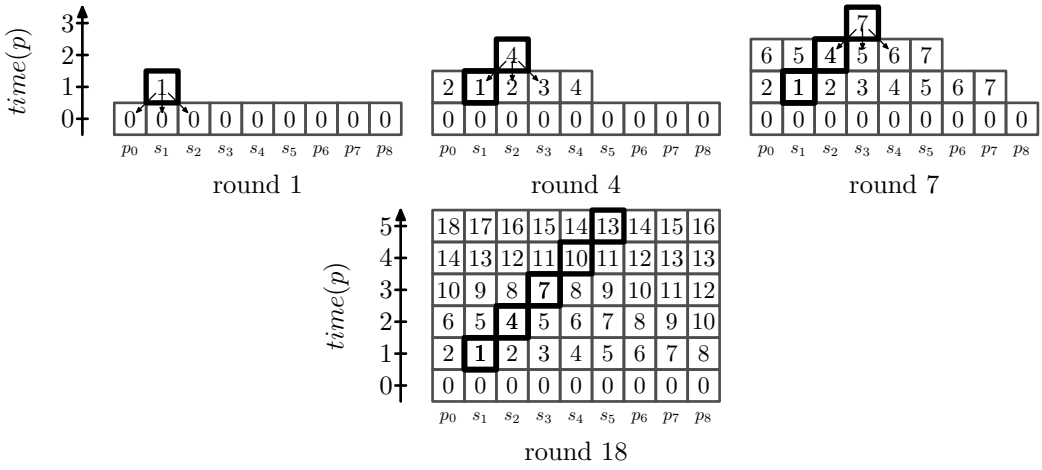

Figure 1 The intuition of the round complexity for the second stage.

Λ has a *starting sequence* that is a sequence $s_1 \cdots s_H$ verifying $\mathtt{st}_{s_i}^{i-1} \neq \mathtt{st}_{s_i}^{i}$ for $1 \leq i \leq H$, and $s_{i-1} \in N[s_i]$ for $1 < i \leq H$. We can then prove that, if all nodes have a positive time at Round X, then the algorithm becomes silent after at most $X + 3H + D - 2$ rounds.

Using similar ideas, we can prove that all times are non-negative after at most $X = 2D$ rounds. Taking into account the $3D + 2$ rounds of the stabilization phase, we obtain an overall $5D + 3T$ round complexity to reach the silence from any configuration.

7 Conclusion

We propose the first fully polynomial self-stabilizing unison algorithm for anonymous asynchronous bidirectional networks of arbitrary connected topology, and use it to obtain new state-of-the-art algorithms for various problems such as BFS constructions, leader election, and clustering.

A challenging perspective would be to generalize our approach to weaker models such as the message passing or the link-register models. We would also be curious to know the properties of the algorithm proposed in Section 3.2. Thirdly, although we could not do it, it would be nice to unify our result with that of [21] in a satisfactory manner. Finally, it would be interesting to know whether or not constant memory can be achieved by an asynchronous self-stabilizing unison for arbitrary topologies.

References

1 S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithms. In *13th Foundations of Software Technology and Theoretical Computer Science, (TSTTCS'93)*, volume 761, pages 400–410, 1993. doi:10.1007/3-540-57529-4_72.

2 K. Altisen, A. Cournier, S. Devismes, A. Durand, and F. Petit. Self-stabilizing leader election in polynomial steps. *Information and Computation*, 254(3):330–366, 2017. doi:10.1016/j.ic.2016.09.002.

3 K. Altisen, S. Devismes, S. Dubois, and F. Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2019. doi:10.2200/S00908ED1V01Y201903DCT015.

4 B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization. In *25th Annual Symposium on Theory of Computing, (STOC'93)*, pages 652–661, 1993. doi:10.1145/167088.167256.

5 B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *32nd Annual Symposium of Foundations of Computer, Science (FOCS'91)*, pages 268–277, 1991. doi:10.1109/SFCS.1991.185378.

6 B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *32nd Annual Symposium on Foundations of Computer Science, (FOCS'91)*, pages 258–267, 1991. doi:10.1109/SFCS.1991.185377.

7 Baruch Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, 1985. doi:10.1145/4221.4227.

8 J. Beauquier, M. Gradinariu, and C. Johnen. Cross-over composition - enforcement of fairness under unfair adversary. In *5th International Workshop on Self-Stabilizing Systems, (WSS'01)*, pages 19–34, 2001. doi:10.1007/3-540-45438-1_2.

9 L. Blin, C. Johnen, G. Le Bouder, and F. Petit. Silent anonymous snap-stabilizing termination detection. In *41st International Symposium on Reliable Distributed Systems, (SRDS'22)*, pages 156–165, 2022. doi:10.1109/SRDS55811.2022.00023.

10 C. Boulinier and F. Petit. Self-stabilizing wavelets and rho-hops coordination. In *22nd IEEE International Symposium on Parallel and Distributed Processing, (IPDPS'08)*, pages 1–8, 2008. doi:10.1109/IPDPS.2008.4536130.

11 C. Boulinier, F. Petit, and V. Villain. When graph theory helps self-stabilization. In *23rd Annual Symposium on Principles of Distributed Computing, (PODC'04)*, pages 150–159, 2004. doi:10.1145/1011767.1011790.

12 J. Burman and S. Kutten. Time optimal asynchronous self-stabilizing spanning tree. In *21st International Symposium on Distributed Computing, (DISC'07)*, volume 4731, pages 92–107, 2007. doi:10.1007/978-3-540-75142-7_10.

13 A. Cournier, A. K. Datta, F. Petit, and V. Villain. Snap-stabilizing PIF algorithm in arbitrary networks. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 199–206, 2002. doi:10.1109/ICDCS.2002.1022257.

14 A. Cournier, S. Rovedakis, and V. Villain. The first fully polynomial stabilizing algorithm for BFS tree construction. *Information and Computation*, 265:26–56, 2019. doi:10.1016/j.ic.2019.01.005.

15 J.-M. Couvreur, N. Francez, and M. G. Gouda. Asynchronous unison (extended abstract). In *12th International Conference on Distributed Computing Systems, (ICDCS'92)*, pages 486–493, 1992. doi:10.1109/ICDCS.1992.235005.

16 A. K. Datta, S. Devismes, K. Heurtefeux, L. L. Larmore, and Y. Rivierre. Competitive self-stabilizing k-clustering. *Theoretical Computer Science*, 626:110–133, 2016. doi:10.1016/j.tcs.2016.02.010.

17 A. K. Datta, S. Devismes, and L. L. Larmore. A silent self-stabilizing algorithm for the generalized minimal k-dominating set problem. *Theoretical Computer Science*, 753:35–63, 2019. doi:10.1016/j.tcs.2018.06.040.

18 A. K. Datta and L. L. Larmore. Leader election and centers and medians in tree networks. In *15th International Symposium on Stabilization, Safety, and Security of Distributed Systems, (SSS'13)*, pages 113–132, 2013. doi:10.1007/978-3-319-03089-0_9.

19 S. Devismes, D. Ilcinkas, and C. Johnen. Optimized silent self-stabilizing scheme for tree-based constructions. *Algorithmica*, 84(1):85–123, 2022. doi:10.1007/s00453-021-00878-9.

20 S. Devismes, D. Ilcinkas, C. Johnen, and F. Mazoit. Making local algorithms efficiently self-stabilizing in arbitrary asynchronous environments. *CoRR*, abs/2307.06635, 2023. doi:10.48550/arXiv.2307.06635.

21 S. Devismes, D Ilcinkas, C. Johnen, and F. Mazoit. Asynchronous self-stabilization made fast, simple, and energy-efficient. In *43rd Symposium on Principles of Distributed Computing, (PODC'24)*, pages 538–548, 2024. doi:10.1145/3662158.3662803.

22 S. Devismes and C. Johnen. Silent self-stabilizing BFS tree algorithms revisited. *Journal on Parallel Distributed Computing*, 97:11–23, 2016. doi:10.1016/j.jpdc.2016.06.003.

23 S. Devismes and C. Johnen. Self-stabilizing distributed cooperative reset. In *39th International Conference on Distributed Computing Systems, (ICDCS'19)*, pages 379–389, 2019. doi:10.1109/ICDCS.2019.00045.

24 S. Devismes and F. Petit. On efficiency of unison. In *4th Workshop on Theoretical Aspects of Dynamic Distributed Systems, (TADDS'12)*, pages 20–25, 2012. doi:10.1145/2414815.2414820.

25 E. W. Dijkstra. Self-stabilization in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

26 S. Dolev. *Self-Stabilization*. MIT Press, 2000.

27 S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7(1):3–16, 1993. doi:10.1007/BF02278851.

28 Y. Emek and E. Keren. A thin self-stabilizing asynchronous unison algorithm with applications to fault tolerant biological networks. In *40nd Symposium on Principles of Distributed Computing, (PODC'21)*, pages 93–102, 2021. doi:10.1145/3465084.3467922.

29 Y. Emek and R. Wattenhofer. Stone age distributed computing. In *32nd Symposium on Principles of Distributed Computing, (PODC'13)*, pages 137–146, 2013. doi:10.1145/2484239.2484244.

30 M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

31 C. Glacet, N. Hanusse, D. Ilcinkas, and C. Johnen. Disconnected components detection and rooted shortest-path tree maintenance in networks. *Journal of Parallel and Distributed Computing*, 132:299–309, 2019. doi:10.1016/j.jpdc.2019.05.006.

32 Maria Gradinariu and Sébastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *27th IEEE International Conference on Distributed Computing Systems (ICDCS 2007), June 25-29, 2007, Toronto, Ontario, Canada*, page 46. IEEE Computer Society, 2007. doi:10.1109/ICDCS.2007.95.

33 A. Kravchik and S. Kutten. Time optimal synchronous self stabilizing spanning tree. In *27th International Symposium on Distributed Computing, (DISC'13)*, pages 91–105, 2013. doi:10.1007/978-3-642-41527-2_7.

34 S. Tixeuil. *Vers l'auto-stabilisation des systèmes à grande échelle*. Habilitation à diriger des recherches, Université Paris Sud - Paris XI, 2006. URL: https://tel.archives-ouvertes.fr/tel-00124848/file/hdr_final.pdf.

35 V. Turau. Efficient transformation of distance-2 self-stabilizing algorithms. *Journal of Parallel and Distributed Computing*, 72(4):603–612, 2012. doi:10.1016/j.jpdc.2011.12.008.