Asynchronous Self-stabilization Made Fast, Simple, and Energy-efficient

Stephane Devismes, David Ilcinkas, Colette Johnen, Fréderic Mazoit

Asynchronous Self-stabilization Made Fast, Simple, and Energy-efficient

Stéphane Devismes
Univ. de Picardie Jules Verne, MIS
Amiens, France
Stephane.Devismes@u-picardie.fr

David Ilcinkas
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Talence, France
David.Ilcinkas@labri.fr

Colette Johnen
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Talence, France
Colette.Johnen@u-bordeaux.fr

Frédéric Mazoit
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800
Talence, France
Frederic.Mazoit@u-bordeaux.fr

Abstract

Distributed systems are ubiquitous, and their distributed nature make them particularly vulnerable to faults. Being able to automatically recover from these faults is of utmost importance, and self-stabilization is a general and lightweight approach to tackle this problem. However, fully asynchronous self-stabilizing algorithms (FASS) are notoriously difficult to design and prove. It thus makes sense to create and prove a transformer that turns synchronous algorithms into FASSes.

The *Rollback Compiler* of Awerbuch and Varghese (FOCS 1991) is such a transformer. The problem is that although it produces FASSes that are fast (time being evaluated in rounds), we prove that their energy requirement (measured through the number of state changes) can be exponential in the number n of nodes. Actually, regardless of the problem, the literature only contains a few FASSes which are asymptotically optimal time-wise. Moreover, several of these algorithms have been shown to require an exponential amount of energy, and no such algorithms are known to be energy-efficient.

In this paper, we introduce the first transformer that turns any terminating synchronous algorithm into a fully asynchronous self-stabilizing algorithm with essentially the same time complexity and with a low energy requirement (polynomial in n). However, as for the rollback compiler, this comes at the cost of a (reasonable) memory increase. Our approach is compatible with most models, ranging from the *LOCAL* model (a powerful synchronous fault-free model), down to models such as the Stone Age model, in which a node cannot even know how many neighbors it has.

In particular, we can transform extremely fast algorithms such as the classical $\Theta(log^* n)$-time ring coloring algorithm by Cole and Vishkin (FOCS 1986) into fast and energy-efficient FASSes. We also provide the best FASSes so far for many classical distributed problems such as leader election and spanning tree constructions (*e.g.*, BFS, shortest-path).

PODC '24, June 17–21, 2024, Nantes, France

ACM ISBN 979-8-4007-0668-4/24/06
https://doi.org/10.1145/3662158.3662803

CCS Concepts

• **Theory of computation → Distributed computing models**; **Distributed algorithms**; **Design and analysis of algorithms**.

Keywords

self-stabilization, Local Model, transformer, time complexity

ACM Reference Format:
Stéphane Devismes, David Ilcinkas, Colette Johnen, and Frédéric Mazoit. 2024. Asynchronous Self-stabilization Made Fast, Simple, and Energy-efficient. In *ACM Symposium on Principles of Distributed Computing (PODC '24), June 17–21, 2024, Nantes, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3662158.3662803

1 Introduction

1.1 Context

Distributed computing is everywhere in our daily lives (Internet, smartphones, aircraft and industrial control systems, ...) and fault-tolerance (*i.e.*, the ability of a system to automatically withstand failures) is a major concern in this area. However, designing distributed algorithms, and in particular asynchronous ones, is notoriously difficult (see *e.g.*, the literature on race conditions and deadlocks). Fault tolerance adds another layer of difficulty to the point that some problems become outright impossible to solve (see *e.g.*, [33]). Furthermore, when it can be achieved, it often comes at the price of sacrificing efficiency (in time, space, or energy) or versatility (see *e.g.*, [16, 20]).

Self-stabilization [27] is a popular paradigm for dealing with these distributed systems in which errors may occur. Self-stabilization focuses on *transient faults*, which occurs at an unpredictable time, but do not result in permanent hardware damage. As opposed to intermittent faults, the frequency of transient faults is considered to be low. After a finite number of transient faults, the configuration of the system may be arbitrary, and may thus violate the safety properties of the system. Now, starting from such an arbitrary configuration, a self-stabilizing algorithm recovers within finite time a *legitimate configuration* from which an intended specification is satisfied. Self-stabilization is commonly considered [4, 29] as a general approach for tolerating such faults. Indeed, it makes no assumptions on the extent or the nature of transient faults that could hit the system (*e.g.*, memory corruption, message loss, topological changes) and a self-stabilizing system recovers from those faults in a unified manner. However, such versatility comes at a price, *e.g.*, after transient faults,

there is a finite period of time during which the safety properties of the system are violated.

One obviously wants "fast" algorithms. The running time of asynchronous algorithms is usually evaluated in terms of *rounds*, which capture the execution time according to the speed of the slowest nodes. Since, to finish its computation, a node may need some information stored in another node far away, the diameter D of the network is often a lower bound for the number of rounds. Hence, looking for solutions achieving round complexities that are polynomial, or even linear, in D is natural. It even becomes quite desirable considering the fact that, in many large-scale networks, D is logarithmic in n, the number of nodes in the network. As an illustrative example, the IPv6 protocol, which allows for up to 2^{128} machines, assumes that the diameter of the network is at most 255 (the TTL is stored on 8 bits).

Energy consumption is also becoming a more and more crucial issue, both for scalability and ecological reasons. We model the energy requirement with the *move complexity*, which counts the total number of state changes. This choice is a bit unusual. Indeed, although local computations do require energy, this energy is often negligible. Messages between nodes are much more costly in this regard. However both are related. Indeed, a node only starts a local computation when it receives information (via messages) about its neighbors, and a node only emits messages when moving. The number of moves is thus a lower bound on the number of messages. Thus if the number of moves is, say, exponential, then so is the number of messages. Conversely, if the number of moves is polynomial, then so is the number of messages. Of course, we must also consider the size of the messages, and take into account the fact that any self-stabilizing algorithm must regularly check for errors. (We will expand more on energy in Section 6.) Since each node usually makes at least one move, a natural parameter for this complexity is n.

1.2 A Difficult Balancing Act

Designing fully asynchronous self-stabilizing algorithms is difficult because of the huge variety of possible interleavings. In particular, fairness is not guaranteed. A node may starve and never be allowed to move as long as some other node can be activated. Requiring that these algorithms are both fast and energy-efficient is thus even more challenging. Indeed, for the algorithm to be fast, nodes should exploit parallelism as much as possible and thus perform their computation as early as possible. But then, a very fast node p may do a lot of computation which turns out to be useless, typically because p has not received a crucial information from a slow node. Thus, fast self-stabilizing algorithms tend to behave poorly energy-wise: their move complexity is often exponential; see *e.g.*, [26]. A natural idea is then to slow down nodes by removing some parallelism to prevent them from doing computations whose dependencies are not sound. This may, in turn, lead to a slow algorithm.

To try to solve this problem, a popular approach is then to use an error mechanism in which nodes that detect "major errors" launch a reset of possibly faulty nodes. Such a partial reset can be done in many ways. If this reset allows nodes to resume their computation too early, then the exponential moves problem most likely remains. A common way to perform a "slow" and accurate reset is to use at least two phases: a broadcast phase in which involved nodes are frozen, and a feedback phase in which involved nodes reset and confirm the good propagation of the freezing. The computation resumes after this second phase. Those two phases organize reset nodes into directed acyclic graphs (DAG) or even trees. Since the freezing is propagated following such underlying error DAGs, the round complexity of the feedback phase depends on their depth, which may be $\Omega(n)$. One may then want to allow nodes to restructure error DAGs until their depth is $O(D)$. But, this shortening process is a computation in itself, and while the "real" computation may be under control, the shortening may become costly, even exponential. This moving of the goal post further suggest that being both fast and energy-efficient is inherently hard.

In fact, this intuition is reflected in the literature. Indeed, just having a round complexity polynomial in D is rare in a fully asynchronous setting. Most algorithms that focus on rounds have a $\Omega(n)$ complexity (we focus here on the fully asynchronous setting). Notable exceptions are the Rollback compiler of Awerbuch and Varghese [10], Dolev's Breadth-First Search (BFS) algorithm [28], and Bellman-Ford-based spanning tree constructions [7, 15]. However, all these algorithms have an exponential move complexity; see Section 7 for the Rollback compiler, [26] for Dolev's BFS, and [6] for Bellman-Ford shortest-path constructions. On the other hand, and as expected, techniques used so far to optimize the move complexity produce round complexities inherently linear in n (*e.g.*, [3, 18, 24]). Another compelling example is the best (and probably optimal) algorithm with respect to moves for tree construction [38] (with a $O(nD)$ move complexity). We can show that this algorithm has a round complexity in $\Omega(n)$. The general consensus seems to be that being both efficient in rounds and moves is only rarely possible.

Cournier, Rovedakis, and Villain [19] tackle the existence of non-trivial algorithms whose round and move complexities are bounded by polynomials in respectively D and n. They prove that such *fully polynomial* self-stabilizing algorithms do exist by providing the only fully polynomial algorithm so far: a $O(n^6)$-move and $O(D^2)$-round BFS spanning tree algorithm for rooted connected networks. They also ask whether (1) the spanning tree problem can be solved by a fully polynomial algorithm with optimal $O(D)$ round complexity, and (2) other typical distributed problems, such as leader election, admit fully polynomial solutions.

1.3 Our Contributions

In this paper, we mostly respond to the two aforementioned questions in the affirmative. Indeed, we prove that every synchronous algorithm which terminates in $O(D)$ rounds can be turned, by the mean of a transformer, into a fully asynchronous self-stabilizing algorithm that converges in $O(D)$ rounds and $O(n^2)$ moves per node (thus, $O(n^3)$ moves in total). Now, it is well known that classical problems such as spanning tree constructions and leader election admit such synchronous solutions; see Section 5 for illustrative examples.

Actually, our transformer is even more general. It operates in the atomic-state model (the most commonly used model in the self-stabilizing area) and has several inputs: (1) a terminating synchronous algorithm $AlgI$, (2) an upper bound B on the execution time T of $AlgI$, which may be set to $+\infty$ if unknown, (3) a boolean flag `greedy`, indicating whether the transformer runs in "greedy" or "lazy" mode. The output is a fully asynchronous *silent* self-stabilizing algorithm

(FASSS) $Trans(AlgI)$ that simulates $AlgI$. *Silent* algorithms are such that the state of each node eventually remains fixed [30]. Silence is a desirable property as, for example, such algorithms can be more easily composed, thus helping the design of algorithms solving more complex tasks [4].

The complexity bounds (in moves and rounds) and the memory requirement of $Trans(AlgI)$ according to the used mode (greedy or lazy) are summarized in Table 1. Intuitively, the error recovery phase ends when everything is back to normal, except that the simulation may not be finished yet. A more precise definition will be given at the beginning of Section 4.

	Move complexity	Round complexity
Lazy mode	$O(\min(n^3+nT,n^2B))$	$O(D+T)$
Greedy mode	$O(\min(n^3+nB,n^2B))$	$O(B)$
Common features		
Error recovery	$O(\min(n^3,n^2B))$	$O(\min(D,B))$
Space complexity	$O(B \cdot S)$ bits	

- T and S are respectively the time and space complexities of $AlgI$.
- $B \in [T,+\infty)$.

Table 1: Complexities of $Trans(AlgI)$.

Notice also that our method is compatible with most variants of the atomic-state model. In particular, our transformer neither requires node identifiers nor local port numbers. It can thus be used in strong models with node identifiers like the *LOCAL* model [41], down to models such as the *Stone Age* model [32]. Our results have the following three direct implications:

(1) In lazy mode, $Trans(AlgI)$ is fully-polynomial whenever $AlgI$ has a round complexity polynomial in D, regardless of the value of B. As a consequence, all problems which can be solved in the *LOCAL* model admits a fully-polynomial FASSS in the asynchronous corresponding model.
(2) Any problem whose optimal complexity is $\Omega(D)$ in the synchronous setting admits a FASSS with the same time complexity and just an additional $O(n^2)$ energy consumption per node (using our transformer in lazy mode). As a consequence, we have time-optimal energy-efficient FASSSes for many classical problems, such as spanning tree constructions and leader election.
(3) When the provided bound B is linear in T, our transformer in greedy mode performs the simulation in $O(T)$, even when T is sublinear in D. In particular, we can transform extremely fast algorithms that use a bound as input, such as the classical $\Theta(log^*n)$-time ring 3-coloring algorithm by Cole and Vishkin [17].

Overall, our solution is very efficient in terms of time and energy but at the price of multiplying the memory cost of the original algorithm by its execution time. However, this is often not a real problem. Indeed, being able to focus on synchronous algorithms with a controlled initialization makes it much easier to design algorithms whose execution times are quite small, achieving complexities such as $O(D)$ or even $O(\log^* n)$ (which is admittedly extreme). Recall that, in large-scale networks such as the Internet, D is often logarithmic in n. Furthermore, using nonces and hashes, one may reduce in practice the communication cost to almost the one of the original algorithm. Finally, our transformer always modifies its state in a way that can be described with limited information, linear in the time and memory complexity of the simulated algorithm, and this can be also used to limit the volume of exchanged information. We elaborate on those two latter points in more detail in Section 6.

To summarize, our transformer allows to drastically simplify the design of energy-efficient FASSSes by overcoming two main sources of uncertainty: the arbitrary initial configuration and the asynchronism.

1.4 Related Works

Transformers are far from being new, and can be used in several ways.

One way is as a (constructive) tool to prove that solutions to a given problem can be assumed to have an additional property. Alongside a proof that the property cannot be achieved outside the class, this gives a full characterization. For example, Katz and Perry [37] have addressed the expressivity of self-stabilization in message-passing systems where links are reliable and have unbounded capacity, and nodes are both identified and equipped with infinite local memories. Several transformers, *e.g.*, [14, 37], build time-efficient self-stabilizing solutions yet work on synchronous systems only. Boldi and Vigna [14] propose a universal transformer for synchronous networks. As in [37], the transformer allows to self-stabilize any behavior for which there exists a self-stabilizing solution. However, the transformer in [14] is slow, stabilizes in at most $n+D$ rounds, and is costly in terms of local memories (basically, each node collects and stores information about the whole network). In the same vein, Afek and Dolev [1] propose to collect pyramids of views of the system to detect incoherences and correct the behavior of a synchronous system. In [13], Blin, Fraigniaud and Patt-Shamir propose two transformers to construct silent self-stabilizing algorithms. Their goal is to minimize memory consumption. The transformers are therefore slow. The first one may need an exponential number of rounds. The second one stabilizes in $O(n)$ rounds.

Transformers have been also used to compare the expressiveness of computational models. Equivalence (in terms of computational power) between the atomic-state model and the register one and between the register model and message passing are discussed in [29].

In the previous examples, transformers are only used as tools in a proof, and in this context, performance is not an issue. Indeed, all aforementioned transformers use heavy (in terms of memory and/or time) mechanisms such as global snapshots and resets in order to be generic. For example, the transformer of Katz and Perry [37] requires infinite local memories and endlessly computes (costly) snapshots of the network even after the stabilization.

Lighter transformers have been proposed but at the price of reducing the class of problems they can handle. For example, locally checkable problems are considered in the message-passing model [2]. The proposed transformer constructs solutions that stabilize in $O(n^2)$ rounds. The more restrictive class of locally checkable and locally correctable problems is studied in [8], still in message-passing. Finally, Awerbuch and Varghese [10] propose, in the message-passing

model, to transform synchronous terminating algorithms into self-stabilizing asynchronous algorithms. They propose two methods: 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, the output algorithm stabilizes in $O(T)$ rounds, resp. $O(T+\mathcal{D})$ rounds, using $O(T \times S)$ space, resp. $O(S)$ space, per node where T, resp. S, is the execution time, resp. the space complexity, of the input algorithm. Notice however that the straightforward atomic-state model version of the rollback compiler (the work closest to our contribution) achieves exponential move complexities, as shown in Section 7. Finally, notice that the rollback compiler of Awerbuch and Varghese is revisited in the position paper of Lenzen, Suomela, and Wattenhofer [40] to show the links, that we exploit here, between the *Local model* [41] and self-stabilization.

1.5 Roadmap

In the next section, we define the computational model. In Section 3, we present our transformer. In Section 4, we sketch its correctness and its complexity. We illustrate the versatility and efficiency of our transformer by presenting several of its instances in Section 5. In Section 6, we justify why reducing the move complexity of self-stabilizing algorithms allows to make them energy-efficient. In Section 7, we establish an exponential lower bound in moves for the rollback compiler, the closest related work. We make concluding remarks in Section 8. Precise statements and proofs can be found in the technical report online [25].

2 Preliminaries

2.1 Networks

We consider *distributed systems* made of $n \geq 1$ interconnected nodes. Each node can directly communicate through channels with a subset of other nodes, called its neighbors. We assume that the network is connected and that communication is bidirectional. More formally, we model the topology by a connected simple graph $G = (V,E)$, where V is the set of *nodes* and E is the set of *edges*. If $\{p,q\}$ is an edge, then q is a *neighbor* of p. We denote by $N(p)$ the set of neighbors of p, and let $N[p] = N(p) \cup \{p\}$.

A *path* is a finite sequence $P = p_0 p_1 \cdots p_l$ of nodes such that consecutive nodes in P are neighbors. We say that P is *from* p_0 *to* p_l. The *length* of the path P is the number l. Since we assume that G is *connected*, then for every pair of nodes p and q, there exists a path from p to q. 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 the maximum distance between nodes of G.

2.2 Computational Model: the Atomic-state Model

Formally, we do not provide a single transformer but a family of transformers. Indeed, since the algorithms that we deal with work under different models, so does our transformers. In this section, we present the weakest model required for our algorithms. We will explain how to cope with other (stronger) models after having presented the transformer, in Section 3.

Our algorithms run on a variant of the *atomic-state model* in which nodes communicate using a finite number of locally shared registers, called *variables*. The *state* of a node is defined by the values of its local variables. A *configuration* of the system consists of the states of each node.

An algorithm is described as a finite set of *rules* of the form $label : guard \longrightarrow action$. A *guard* is a boolean predicate involving the state of the node and the set of states of its neighbors. The *action* part of a rule updates the state of the node. If the guard of a rule evaluates to *true*, the rule is *enabled* and can be executed. By extension, a node is *enabled* if at least one of its rules is enabled, and $Enabled(\gamma)$ denotes the set of enabled nodes in γ. Note that this model is quite weak. Indeed, a node only receives the *set* of states of its neighbors, so it may not even be able to count how many neighbors it has.

In the model, executions proceed as follows. Given a configuration γ with $Enabled(\gamma) \neq \emptyset$, a so-called *daemon* selects a nonempty set of enabled nodes $\mathcal{X}$ that simultaneously and atomically execute one of their enabled rules, leading to a new configuration γ'. The transition $\gamma \mapsto \gamma'$ is a *step*, and we say that each node of $\mathcal{X}$ executes a *move* during $\gamma \mapsto \gamma'$. 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 $i > 0$. Thus, an execution starts from an arbitrary configuration and is either infinite, or ends at a *terminal* configuration γ^f where $Enabled(\gamma^f) = \emptyset$. An algorithm which admits no infinite executions is said to be *terminating* or *silent*.

We consider two particular daemons: the *synchronous* one, which always selects all enabled nodes, and the *fully asynchronous* one (also called the distributed unfair daemon), which has no constraints except that at each step it must select a nonempty set of enabled nodes. The latter daemon may never select a specific enabled node unless it is the only enabled node.

We use two units of measurement to evaluate the complexity: *moves* for the energy consumption, and *rounds* for the time. The definition of a round uses the concept of *neutralization*: a node p is *neutralized* during a step $\gamma^i \mapsto \gamma^{i+1}$, if p is enabled in γ^i but not in configuration γ^{i+1}, and does not execute any move 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' 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. Note that in a synchronous execution, steps and rounds coincide, unless a terminal configuration is reached. The *complexity* of a silent self-stabilizing algorithm is the maximum number of moves or rounds over every possible execution under the considered daemon.

3 The Algorithm

We propose a transformer that takes as an input a terminating synchronous distributed algorithm *AlgI* and transforms it into an efficient fully asynchronous silent self-stabilizing algorithm which simulates it.

Our algorithm is a major improvement on the *Rollback Compiler* algorithm of Awerbuch and Varghese [10]. The *Rollback Compiler* is based on a standard approach (already developed in [9], for example) which consists in storing the whole synchronous execution of the input algorithm *AlgI*. Every node p thus has a list L such that for any cell i, ultimately, $p.L[i] = \mathtt{st}_p^i$, where $\mathtt{st}_p^i$ is the state of p at Round i

of the synchronous execution of *AlgI*. Since the cell $p.L[i+1]$ is computed using the cells $q.L[i]$ for node q in the closed neighborhood of p (*i.e.*, p and its neighbors), we say that $p.L[i+1]$ *depends* on the cells $q.L[i]$. In [10], when p is activated, it finds its faulty cells, and corrects all of them. Moreover, if $p.L[i]$ does not exist but is necessary (*e.g.*, the synchronous execution of *AlgI* contains at least i rounds) and all its dependencies exist, then it creates $p.L[i]$. This gives a simple algorithm which has a good round complexity but has an exponential move complexity; see Section 7 for details.

In our transformer, to control the number of moves, a node that discovers a problem launches a partial reset by the mean of an error broadcast / error feedback mechanism that builds error DAGs. Nodes in DAGs are *frozen* (*i.e.*, they cannot execute *AlgI*) until the broadcast and feedback have terminated in their DAGs.

In the following, we give the actual algorithm (Section 3.1), after which we explain in detail the principles of the algorithm, and in particular how the error broadcasts work (Section 3.2).

3.1 Definition of the Algorithm

Let T be the execution time in synchronous rounds of *AlgI*. Apart from the algorithm *AlgI* that we simulate, our transformer needs two additional parameters as inputs:

- an upper bound B on T, which can be set to $+\infty$ when such a bound is not available;
- a boolean parameter `greedy` indicating which mode the transformer uses.

Precisely, `greedy` is true when the transformer runs in "greedy" mode and false when it runs in "lazy" mode. In greedy mode, the transformer simulates B rounds of the input algorithm while, in lazy mode, a new round is simulated only if necessary. In greedy mode, since B is an upper bound on T, we may end up simulating, say, 18 rounds of an algorithm which terminates in 10 rounds. To do so, we assume that the 8 last rounds "do nothing".

The state `st` of a node consists of the following shared variables:

- $\texttt{st}.init$: an initial state in the simulated algorithm; it cannot be modified and constitutes the read-only part of the state;
- $\texttt{st}.s$: the status, which can take two values: C or E;
- $\texttt{st}.L$: a list of at most B elements containing states of *AlgI*.

Given a node p with (local) state $\texttt{st}_p$, $p.init$, $p.s$, and $p.L$ respectively denote $\texttt{st}_p.init$, $\texttt{st}_p.s$, and $\texttt{st}_p.L$. A node p such that $p.s=E$ is said to be *in error*. We denote by $p.h$ the length of $p.L$, which we call the *height of* p, and by $p.L[i]$ with $i>0$ the i^{th} element of $p.L$. To simplify the design, we conventionally set $p.L[0]:=p.init$. Finally, in an execution $\gamma^0\gamma^1\cdots$, we respectively denote by $p.s^i$, $p.L^i$ and $p.h^i$ the value of $p.s$, $p.L$ and $p.h$ in γ^i.

As already stated at the beginning of the section, we ultimately want that $p.L[i]=\texttt{st}_p^i$. We thus must be able to call the simulated algorithm on the cells of $p.L$ and those of its neighbors. For this purpose, we define $\widehat{\texttt{algo}}(p,i)$ to be the result of the application of *AlgI* by node p when each node q has the state $q.L[i]$. More precisely, in the model presented in Section 2.2, we have:

$$\widehat{\texttt{algo}}(p,i):=AlgI\big(p.L[i],\{\texttt{st}.L[i]\mid \texttt{st}\in\{\texttt{st}_q\mid q\in N(p)\}\}\big)\,.$$

In this weak model, apart from its state, a node p has only access to the set $\{\texttt{st}_q\mid q\in N(p)\}$ of its neighbors' states. A guard should thus not contain a direct reference to a neighbor q of p. However, the semantics of $\exists \texttt{st}\in\{\texttt{st}_q\mid q\in N(p)\},\text{Pred}(\texttt{st}.init,\texttt{st}.s,\texttt{st}.L)$, for some predicate Pred, is precisely $\exists q\in N(p),\text{Pred}(q.init,q.s,q.L)$. We can also encode similar universal statements. To increase readability, we will thus make heavily use of the following shortcuts:

$$\texttt{Shortcut1}:\exists q\in N(p),\text{Pred}(q.init,q.s,q.L):=\\ \exists \texttt{st}\in\{\texttt{st}_q\mid q\in N(p)\},\text{Pred}(\texttt{st}.init,\texttt{st}.s,\texttt{st}.L)$$

$$\texttt{Shortcut2}:\forall q\in N(p),\text{Pred}(q.init,q.s,q.L):=\\ \forall \texttt{st}\in\{\texttt{st}_q\mid q\in N(p)\},\text{Pred}(\texttt{st}.init,\texttt{st}.s,\texttt{st}.L)\,.$$

Below, we define the predicates used by our algorithm.

$$algoErr(p):=\exists i,1\le i\le p.h,(\forall q\in N(p),q.h\ge i-1)\\ \wedge p.L[i]\neq\widehat{\texttt{algo}}(p,i-1)$$

$$depErr(p):=\big(p.s=E\wedge\neg(\exists q\in N(p),q.s=E\wedge q.h<p.h)\big)\\ \vee\big(p.s=C\wedge\exists q\in N(p),q.h\ge p.h+2\big)$$

$$root(p):=algoErr(p)\vee depErr(p)$$

$$errProp(p,i):=\exists q\in N(p),q.s=E\wedge q.h<i<p.h$$

$$canClearE(p):=p.s=E\wedge\forall q\in N(p),\big(|q.h-p.h|\le 1\\ \wedge(q.h\le p.h\vee q.s=C)\big)$$

$$updatable(p):=p.s=C\wedge p.h<B\wedge\big(\forall q\in N(p),p.h\le q.h\le p.h+1\big)\\ \wedge\big(\texttt{greedy}\vee p.L[p.h]\neq\widehat{\texttt{algo}}(p,p.h)\vee\exists q\in N(p),q.h>p.h\big)$$

Our algorithm uses these predicates and is defined as the following four rules.

- $R_R:(p.h>0\vee p.s=C)\wedge root(p)\longrightarrow p.h:=0\,;p.s:=E$
- $R_P(i):errProp(p,i)\longrightarrow p.h:=i\,;p.s:=E$
- $R_C:canClearE(p)\longrightarrow p.s:=C$
- $R_U:updatable(p)\longrightarrow p.L[p.h+1]:=\widehat{\texttt{algo}}(p,p.h)$

We set that R_R has the highest priority, and $R_P(i)$ has a higher priority than $R_P(i+l)$ for $l>0$.

3.2 Principles of the Algorithm

Let us now explain the algorithm. First of all, at a very high level, the rules are used as follows. The rule R_R is applied by a node which detects a major error, and thus initiates an error broadcast. The rule R_P is used both to propagate error broadcasts and to shorten the corresponding DAGs. The rule R_C allows a node to leave an error DAG. Finally, R_U is used to perform a simulation move.

We start our explanation of the algorithm with the rule R_U. Informally, this rule is used by a node p when the situation locally looks normal, and p's next cell can be computed. More precisely, the first part of the *updatable* predicate expresses that p is in the correct status C, its list is not full, and the lists of its neighbors are long enough to allow p to compute its next cell, but not too long either.

In greedy mode, p computes its next cell with *AlgI* applied on the relevant cells of its neighbors. In the lazy mode however, p performs this computation only after checking that, either the algorithm has not yet terminated (*i.e.*, the new simulated state would be different from the last one), or some neighbor q has already a value in its cell $q.L[p.h+1]$.

Let us now focus on the three other rules, which deal with error recovery. When a node r detects a major error, it is a *root* (*cf.* Predicate $root(r)$, to be explained later) and it initiates an error broadcast. To do so, r empties its list $r.L$ (which removes all its possible faulty cells) and switches to the error status E by applying the rule R_R.

The cells removed by r most likely create cells with missing dependencies in a neighbor p. Since these cells are most likely faulty, p removes them with the rule R_P, and it switches its status to error to inform its neighbors of the ongoing error broadcast. As a matter of fact, the rule R_P does not rely on the fact that r applied the rule R_R. It only needs the neighbor to be in error. This allows the rule R_P to be more aggressive. Indeed, p removes all its cells with missing dependencies in any of its neighbors in error.

There is thus a causal link between p and its neighbor(s) in error of minimal height. We generalize these links by saying that p is a *child* of a neighbor q if both p and q are in error and $p.t > q.t$. This defines a DAG among the nodes in error. Note also that a given node may apply the rule R_P several times, thus compressing the error DAG.

We can now properly define what a root is; we will explain the rule R_C afterward. Obviously, contents of cells should correspond to the application of $\widehat{\texttt{algo}}$ on the depending cells. The predicate *algoErr* checks whether any inconsistency of this type exists. Furthermore, any node in error which did not initiate an error broadcast should have a neighbor in error with a shorter list. Any node in error without such a "parent" is also considered to be a root (first part of predicate *depErr*). Finally, since the error rules R_R and R_P remove cells, we expect cells to have missing dependencies. However, such missing dependencies should only appear in neighbors which are in error. More formally, if p has a neighbor q such that $q.h \geq p.h+2$, then q has a missing dependency in p. The node p should be in error. If this is not the case, then p is also considered to be a root (second part of predicate *depErr*).

We implement the classical feedback mechanism without a specific status, but instead by switching the status of a node from E to C. Any node without children nor would-be children (*i.e.*, neighbors which could apply the rule R_P and become a child) can leave the DAG by switching its status to C (rule R_C).

The purpose of this broadcast/feedback mechanism is to forbid nodes involved in a broadcast from a root r to apply the rule R_U until r has been notified that its broadcast has ended: r will the first of those nodes to resume the simulation by R_U. And indeed, whenever a node p (which is not a root) enters the error DAG, it does so by applying the rule R_P. It thus has a neighbor q in the DAG such that $q.h < p.h$. Because of this neighbor, p cannot apply the rule R_U. We prove that such a node always exists as long a r has not moved, even if p has got back Status C. Hence, the "freeze" property is achieved.

3.3 Dealing with Stronger Models

Accesses to neighbors' local states by our algorithm are only performed through $\widehat{\texttt{algo}}$, `Shortcut1`, and `Shortcut2`. Moreover, all our proofs only rely on their semantics, not on their implementation. Therefore, our claims about our transformer will still hold in any model where it is possible to implement $\widehat{\texttt{algo}}$, `Shortcut1` and `Shortcut2` while respecting their semantics.

Stronger models include classical ones such as models in which nodes have unique identifiers (like the *LOCAL model*), semi-uniform models in which some nodes are distinguished, and models in which port numbers allow nodes to locally distinguish their neighbors. The encoding is usually straightforward. For example, in the case of node identifiers, we can assume that nodes receive pairs $(\texttt{id},\texttt{st})$ containing the identifier and the state of each neighbor. The $\widehat{\texttt{algo}}$ macro can simply be encoded as follows:

$$\widehat{\texttt{algo}}(p,i) := AlgI\big((\texttt{id}_p, p.L[i]), \{(\texttt{id},\texttt{st}.L[i]) \mid (\texttt{id},\texttt{st}) \in \{(id_q,\texttt{st}_q) \mid q \in N(p)\}\}\big).$$

4 Correctness and Complexity Analysis

In the following, we consider an execution $e = \gamma^0\gamma^1\cdots$ and we partition the steps $\gamma^i \mapsto \gamma^{i+1}$ into *segments* such that each step in which at least one *root* (*i.e.*, a node p satisfying $root(p)$) applies the rule R_C (and thus disappears) is the last step of a segment. Although it is a bit tedious, it is straightforward to prove that roots cannot be created. As a consequence, there are at most n segments in which roots disappear. They constitute the *error recovery phase*. Furthermore, there is at most one rootless segment (the last one) which constitutes what we call the *simulation phase*.

4.1 Terminal Configurations

Every configuration with a root r contains a node which can be activated. Indeed, if $r.s = C$, then r can be activated, otherwise there exists nodes (such as r) in error. Let p be in error with maximum height. If p has a neighbor q such that $p.h \geq q.h+2$, then either $q.s = C$ and q can apply the rule R_R or $q.s = E$ and p can apply the rule R_P. If p has a neighbor q such that $q.h \geq p.h+2$, then q can apply the rule R_P. And if neither previous cases occur, then $p.h$ being maximum implies that p can apply the rule R_C.

As a consequence, *terminal* configurations (*i.e.*, in which no nodes can be activated) contain no roots, and thus, by induction on i, we have $p.L[i] = \texttt{st}_p^i$ for every node p and every $i \leq p.h$. Moreover, in such a configuration, for all nodes p and q, $p.h = q.h$. Indeed, otherwise, there exists p with $p.h$ minimum and with a neighbor q such that $q.h = p.h+1$. This node can apply the rule R_U.

Recall that we assume that $B \geq T$. In greedy mode, the common final height is B, and thus, the algorithm does not terminate if $B = +\infty$. In lazy mode, if initially all heights are less than or equal to T, then the final height is T. Indeed, when $AlgI$ is not silent yet, there exists p such that $\texttt{st}_p^i \neq \texttt{st}_p^{i+1}$, and thus p can apply the rule R_U. Otherwise, the final height is at most the maximum initial height of a node.

4.2 Move Complexity

To bound the number of moves, we individually bound the number of R_R-moves, R_P-moves, R_C-moves, and R_U-moves. Since no roots can be created, we can easily bound the number of R_R-moves by n. The bound on R_C-moves is also easy. Indeed, between two applications of the rule R_C, a node p has to apply either the rule R_R or the rule R_P. The number of R_C-moves is thus at most $n + \#\{R_R\text{-moves}\} + \#\{R_P\text{-moves}\}$.

A path $P = v_0 \cdots v_l$ is *decreasing* in a given configuration if $v_i.h > v_{i+1}.h$ for $0 \leq i < l$. A *D-path* is a decreasing path P ending at a root in error. Note that if a node p is in error, then either it is a root or it has a neighbor with a smaller height which is also in error. In both cases, p is the first node of a D-path.

Consider a step $\gamma^a \mapsto \gamma^b$ and a node p which is the first node of a D-path P in γ^a. Let r be the root of P. If P contains no nodes in error in γ^b, then r is no longer a root in γ^b and $\gamma^a \mapsto \gamma^b$ is thus the last step of a segment. Otherwise, let q be the first node of P which is in error in γ^b (*n.b.*, q may have executed R_P in $\gamma^a \mapsto \gamma^b$, thus making P locally increasing after q). Since q is in error, it is the first node of a D-path, and if we remove the part of P after q and replace it with this D-path, we obtain a D-path from p. This implies that any node p which belongs to a D-path remains in a D-path until the last step of the segment. Since no nodes of a D-path can apply the rule R_U, once a node applies an error rule (and thus belongs to a D-path), it can no longer apply the rule R_U until the end of the segment.

Thus, in any segment, the respective numbers of R_U and R_P rules that a node p applies is the number of values that $p.h$ can reach after having applied the respective rules. This allows us to easily bound the numbers of applications of the rules R_U and R_P in the error recovery phase by B per node and per segment, leading to an overall $O(n^2B)$ bound on the number of moves in this phase.

Since B can be very large and even $+\infty$, we now give a bound which does not depend on B. It is quite easy to see, by induction on l, that if a node p consecutively applies the rule R_U $2l+1$ times, then all the nodes at distance at most l from p must increase their height by at least one in the meantime. Since a root cannot increase its height inside a segment, it implies that, in the error recovery phase, the rule R_U is applied at most $2D$ times per node and per segment.

Let us focus on the rule R_P, and let s be a segment. If a node p_0 applies the rule R_P in $\gamma^{j_1} \mapsto \gamma^{j_1+1} \in s$, then $p_0.h^{j_1+1} = p_1.h^{j_1} + 1$ for some neighbor p_1 of p_0. Now, p_1 may have also applied the rule R_P in s and $p_1.h^{j_1} = p_1.h^{j_2+1} = p_2.h^{j_2} + 1$, with p_2 a neighbor of p_1 in error, and so on. This defines a *causality chain* $p_0 \cdots p_l$, and $p_0.h^{j_1+1} = p_l.h^{j_l} + l$. Now, because rules R_P and R_U do not alternate in s, a given node cannot appear twice in the causality chain, thus $l < n$. Moreover, $p_l.h^{j_l}$ is either the value of $p_l.h$ at the beginning of s, or 0 if p_l has applied the rule R_R. The height $p_0.h$ can thus take at most $n(n+1)$ distinct values in s, which implies that the rule R_P is applied $O(n^2)$ times per node and per segment. Note that a more careful analysis gives an overall bound of $O(n^3)$ instead of the $O(n^4)$ bound sketched here. Combining this result with our first bound, we have at most $O(\min(n^3, n^2B))$ moves in the error recovery phase. Note that this bound is valid regardless of the mode in which the algorithm works.

Now, in the simulation phase, only the rule R_U can be applied. Each move increases the height of the node by one. Thus, we have at most B such applications for each node, which gives a total number of moves of $O(\min(n^3+nB, n^2B))$. In greedy mode, this is all that we can do. In lazy mode, if, at the beginning of the simulation phase, all heights are less than T, we have seen that the common final height is T in which case the number of moves is at most nT. Otherwise, since the nodes with maximum height never apply the rule R_U anymore, and since the difference of height between two nodes is at most D in this phase, the number of moves is at most $nD \leq n^3$. In the end, in lazy mode, we have a total number of moves which is $O(\min(n^3+nT, n^2B))$.

4.3 Round Complexity

Let r be a root in $\gamma^{r_{D+1}}$, the first configuration of Round $D+1$. By the end of the first round (using the rule R_R if needed), $r.h = 0$ and $r.s = E$. Now, since r is still a root in $\gamma^{r_{D+1}}$ (and thus cannot make a move in the meantime), this is still true in $\gamma^{r_{D+1}}$. Furthermore, every neighbor p of r such that $p.h > 1$ can apply Rule R_P. So, by the end of Round 2, $p.h \leq 1$ and, since r cannot increase its height, so does p. By induction on the distance $d(p,r)$ between p and r, in $\gamma^{r_{D+1}}$, $p.h \leq d(p,r)$, for every node p.

Let a *cliff* be a pair (p,q) of neighboring nodes such that $p.h \geq q.h+2$. We claim that $\gamma^{r_{D+1}}$ contains no cliff (p,q). Since a root in $\gamma^{r_{D+1}}$ cannot apply the rule R_R, q must be in error. Now q is the first node of a D-path ending at a root r in error, and thus $q.h \geq d(q,r)$. Since $p.h \geq q.h+2$, we have $p.h > d(p,r)$. But this is impossible because r is a root in $\gamma^{r_{D+1}}$, and we have proven that $p.h \leq d(p,r)$.

Since there are no cliffs in $\gamma^{r_{D+1}}$ and the rule R_R cannot be applied, we both have that the difference between heights of any two nodes is at most D, and that no nodes can apply an error rule anymore. But then, any node in error of maximum height can apply the rule R_C, which implies that by the end of Round $2D+2$, there are no longer roots.

If $B < D$, a similar analysis gives a bound of $2B+2$ rounds. This may seem counterintuitive as the procedure described above is an error broadcast/error feedback mechanism, and it is natural to expect that such a reset procedure takes $\Omega(D)$ rounds. However, the main purpose is to delete erroneous cells and all the cells which depend on them. Now $p.L[i]$ depends on $q.L[j]$ if and only if $j > i$ and $i - j \geq dist(p,q)$. Thus, if $B < D$, issues in a root r cannot impact the nodes at distance further than B. In this regard, a reset procedure in $O(\min(B,D))$ rounds is no longer that strange. Note that, similarly as for the move complexity, this bound for the error recovery phase is valid regardless of the mode.

Let us focus on the simulation phase. In greedy mode, any node of lowest height can apply the rule R_U, thus the simulation phase takes at most B rounds, which gives an overall $O(B)$ round complexity. In lazy mode, suppose that, initially, the maximum height $h_{\max}$ is at most T (the case $h_{\max} > T$ is similar). Note that if all $q \in N[p]$ are such that $\mathtt{st}_q^i = \mathtt{st}_q^{i+1}$, then $\mathtt{st}_p^{i+1} = \mathtt{st}_p^{i+2}$. Thus any p such that $\mathtt{st}_p^{i+1} \neq \mathtt{st}_p^{i+2}$ has some $q \in N[p]$ such that $\mathtt{st}_q^i \neq \mathtt{st}_q^{i+1}$. Therefore, there exists a sequence $p_0, \ldots, p_{T-1}$ such that $p_{i+1} \in N[p_i]$ and $\mathtt{st}_{p_i}^i \neq \mathtt{st}_{p_i}^{i+1}$.

At the end of Round 1, $p_0.h \geq 1$. Any neighbor q of p_0 such that $q.h = 0$ at the beginning of Round 2 can apply the rule R_U and thus $q.h \geq 1$ at the end of Round 2. More generally, any q at distance d from p_0 is such that $q.h \geq 1$ at the end of Round $d+1$. Note that p_1 has all the dependencies for $p_1.L[2]$ at the end of Round 3. Thus, at the end of Round 4, $p_1.h \geq 2$. Furthermore, at that time, any neighbor q of p_1 also has its dependencies for $q.L[2]$. More generally, by induction on i, we show that p_i has height at least i at the end of Round $3i+1$, and that any q at distance d from p_i is such that $q.h \geq i$ at the end of Round $3i+1+d$. This implies that after at most $3T+D$ rounds, the algorithm terminates. Hence, in lazy mode, we obtain an overall bound of $O(T+D)$ rounds.

5 Instances

To illustrate the versatility and efficiency of our approach, we now sketch examples solving three benchmark distributed computing problems: leader election, breadth-first spanning tree, and 3-coloring in rings. The two first instances answer two open questions left in [19].

5.1 Leader Election

The Problem. Recall that leader election requires all nodes to eventually permanently designate a single node of the network as the leader; the network being assumed to be connected. To (deterministically) achieve this task, we should augment our model with unique node identifiers: the state of each node p now includes its own identifier $p.ID$, a non-modifiable integer. Then, nodes will have to compute the identifier of the leader.

The Algorithm. For each node p, the algorithm simply consists in computing into an integer variable $p.Best$ the minimum identifier of the network. First, each node p initializes $p.Best$ with its own identifier $p.ID$. Then, at each synchronous round, p evaluates the minimum value *min* among the *Best* variables in its closed neighborhood and moves by updating $p.Best$ with *min* if necessary. Thus, after each round, each node knows the minimum identifier of nodes one hop further. After at most D rounds, the computation terminates and the *Best* variable of each node is forever equal to the minimum identifier of the network.

Contribution and Related Work. Using our transformer in the lazy mode, we obtain a fully-polynomial silent self-stabilizing leader election algorithm that stabilizes in $O(D)$ rounds and $O(n^3)$ moves. Moreover, by giving an upper bound B on D as input of the transformer, we obtain a bounded-memory solution achieving similar time complexities. Precisely, if we made the usual assumption that identifiers are stored in $O(\log n)$ bits, we obtain a memory requirement in $O(B.\log n)$ bits per node.

To our knowledge, our solution is the first fully-polynomial asynchronous silent self-stabilizing solution of the literature. Indeed, several asynchronous self-stabilizing leader election algorithms [3, 22, 23] written in the atomic-state model have been proposed for arbitrary connected and identified networks. However, none of them is proven fully-polynomial. Actually, they all achieve a stabilization time in $\Theta(n)$ rounds. Note that the algorithm in [3] has a stabilization time in moves that is polynomial in n, while [22, 23] have been proven to stabilize in a number of moves that is at least exponential in n; see [3]. Notice also that the algorithm proposed in [39] actually achieves a leader election in $O(D)$ rounds, yet under synchronous settings.

5.2 Breadth-First Search Spanning Tree Construction

The Problem. We now consider the problem of distributedly computing a breadth-first search (BFS) spanning tree in a connected network rooted at some node r. By "distributedly", we mean that every non-root node will eventually designate a neighbor as its parent in the computed spanning tree. Being BFS, the length of the unique path in the tree from any node p to the root r should be equal to the distance from p to r in the network.

To distinguish the root, we use the non-modifiable boolean variable at each node p. This latter indicates whether or not the node is the root. Moreover, as we could seen, processes should be able to distinguish their neighbors in this problem. Yet, this time, nodes are not assumed to be identified. Instead, we augment the model with a local labeling at each node that distinguishes the channel incoming from their neighbors (one can think about port numbers, for example).

The Algorithm. Each node p holds a variable so-called *parent pointer* which is used to designate its parent in the BFS spanning tree. Initially, each parent pointer is set to the special value $NULL$ meaning that no node has parent yet. Since the root has no parent at all, its pointer is already at the right value and so never take any step. Then, at each synchronous round, every non-root node p whose parent pointer is $NULL$ checks whether a neighbor is the root or has a non-$NULL$ parent pointer; in this case p definitely designates the channel to such a neighbor with its pointer. If several neighbors satisfy the condition, p breaks ties using channel labels. After at most D synchronous rounds, all non-root nodes have a parent, *i.e.*, the BFS spanning tree is (definitely) defined and the execution stops.

Contribution and Related Work. Similarly to the leader election instance, using our transformer in the lazy mode, we obtain a fully-polynomial silent self-stabilizing BFS algorithm that stabilizes in $O(D)$ rounds and $O(n^3)$ moves. Moreover, by giving an upper bound B on D as input of the transformer, we obtain a bounded-memory solution achieving similar time complexities. Precisely, its memory requirement is $O(B.\log \Delta)$ bits per node, where Δ is the maximum node degree in the network.

To our knowledge, our solution is the first fully-polynomial asynchronous silent self-stabilizing solution of the literature that achieves a stabilization time in rounds that is asymptotically linear on the network diameter. Indeed, several self-stabilizing algorithms that construct BFS spanning trees in arbitrary connected and rooted networks have been proposed in the atomic-state model [18, 19, 21, 35]. In [26], the BFS spanning tree construction of Huang and Chen [35] is shown to be exponential in moves. The algorithm in [18] is not silent and computes a BFS spanning tree in $O(\Delta \cdot n^3)$ moves and $O(D^2+n)$ rounds. The silent algorithm given in [19] is fully-polynomial: it has a stabilization time in $O(D^2)$ rounds and $O(n^6)$ moves. The algorithm given in [36] is not silent and is shown to stabilize in $O(D \cdot n^2)$ rounds in [21], however notice that its memory requirement is in $O(\log \Delta)$ bit per node.

Another self-stabilizing algorithm, implemented in the link-register model, is given in [31]. It uses unbounded node local memories. However, it is shown in [26] that a straightforward bounded-memory variant of this algorithm, working in the atomic-state model, achieves an asymptotically optimal stabilization time in rounds, *i.e.*, $O(D)$ rounds where D is the network diameter; however, in the same paper, its move complexity is also shown to be at least exponential in n.

5.3 Vertex 3-Coloring in Oriented Rings

The Problem. The coloring problem consists in assigning a color (a natural integer) to every node in such a way that no two neighbors have the same color. To (deterministically) achieve this task, we should augment our model with unique node identifiers chosen in $[0..n^c-1]$, with $c \in \mathbb{N}^*$. The orientation of the ring is given by the channel labels. A node should distinguish the state of its clockwise neighbor from its counterclockwise one. For instance, we can assume the channel number of the clockwise neighbor is smaller than the one of counterclockwise neighbor.

The Algorithm. The algorithm of Cole and Vishkin [17] can be adapted for the *LOCAL* model [41] to compute a vertex 3-coloring of any oriented ring of n identified nodes in $log^*(n^c)+7$ rounds. Being

written in the *LOCAL* model, it is then a synchronous terminating algorithm that is suited for our model.

Contribution and Related Work. Using our transformer in the greedy mode, we obtain a silent self-stabilizing 3-coloring algorithm on oriented rings that stabilizes in $O(B)$ rounds and $O(n^2B)$ moves. Moreover, its memory requirement is in $O(B.\log n)$ bits per node. If we carefully choose B to be in $O(log^*(n))$, then we obtain a solution that stabilizes in $O(\log^* n)$ rounds and $O(\log^*(n).n^2)$ moves using $O(\log^* n.\log n)$ bits per node.

To our knowledge, our solution is the first self-stabilizing 3-coloring ring algorithm achieving such small complexities. Indeed, self-stabilizing node coloring has been almost exclusively investigated in the context of anonymous networks. Vertex coloring cannot be deterministically solved in asynchronous and anonymous settings [5]. This impossibility has been circumvented by considering central schedulers or probabilistic settings [11, 12, 34]. In [40], a self-stabilizing version of the Cole and Vishkin algorithm is proposed, but the solution (based on the rollback of Awerbuch and Varghese [10]) does not achieve a move complexity polynomial in n.

6 Move Complexity and Energy-saving

As stated in the introduction, messages are a major, if not the, source of energy consumption in a distributed algorithm. Actually, since our algorithm assumes the atomic-state model, which does not consider messages, this section is a bit shaky. But we can still implement such an algorithm in practical (message passing) systems. As explained in [4, 30], to do so, all nodes should permanently check whether or not they should change their local state due to the modification of some neighbors' local states, which means that they have to regularly send information to all their neighbors and to store the last known state from each neighbor.

There are two kinds of sent messages: the messages needed by the algorithm and the messages used to check for transient faults. A first remark is that the second ones need not be sent too often in most applications.

Now, both kinds of messages involve messages containing whole states, which may be heavy. To avoid this problem, one can adopt the lightweight approach proposed in [30]: since neighbors store the state of their neighbors, a solution is not to send the state but only a proof of it. Such proof can be much smaller: for example, a hash of the state salted with a nonce, together with that nonce. If the proof does not correspond to the locally stored copy, a node can request a whole (heavy) copy of the state. This should only happen during or just after the transient faults. With this additional strategy, the self-stabilizing messages can be both rare and lightweight, while the number of heavy messages for the algorithm directly depends on the number of moves.

Together with the previous state-proof mechanism, we can use the features of our transformer to further reduce the volume of exchanged information. We can just encode the difference between the new and the previous state of the sending node. This leads to a reduction of these messages from $O(B \times S)$ (recall that S is the space complexity of *AlgI*) to only $O(S+\log B)$. Indeed, we can encode the name of a rule (R_R, R_C, R_P or R_U) with 2 bits. The rule R_p then only requires an integer value stored using $O(\log B)$ bits. The most expensive rule is R_U which requires $O(S)$ bits, to send the value of the newly computed *AlgI*'s state.

7 The Energy Consumption of the Rollback Compiler

We now show that the rollback compiler of Awerbuch and Varghese [10] can have an exponential number of moves. Apart from the algorithm it simulates, this compiler takes a constant B as input: the rollback simulates B rounds of the input algorithm. The lists $p.L$ are thus all set to have length B. A node p just corrects its faulty cells $p.L[i]$ each time it is needed.

Consider that each node p has an input $p.I \in \{0,1\}$, and let *AlgI* be an algorithm which computes the minimum of these values (in the variable $p.S$ of each node). The algorithm simply consists for each node in repeatedly evaluating $min = \min(q.S \mid q \in N[p])$, and updates $p.S$ to min whenever $p.S \neq min$.

We now define the graph G_k that will be used in our exponential execution. The graph G_1 is the path b_1,a_1,c_1,d_1,e_1, and G_k is recursively defined as the disjoint union of G_{k-1} and a new path b_k, a_k, c_k, d_k, e_k, together with the edges b_kc_{k-1} and e_kc_{k-1}. The *bottom path of* G_k is the simple path from c_k to e_1 which goes through the nodes c_i, d_i and e_i. Given any positive number $i \leq B$, we denote by $\bar{i}$ the list L of size B such that $L[j]=1$ for $0 \leq j < i$ and $L[j]=0$ otherwise. The *index* of a node p such that $p.L=\bar{i}$ is i.

We now consider the following initial configuration of *AlgI* on G_k. For every node p, $p.I=1$. Every node a_i satisfies $a_i.L=\overline{d(a_i,c_k)}$, and for all other nodes, $p.L=\overline{d(p,c_k)+1}$ (see Fig 1). We then recursively define the partial execution Γ_i which starts from the initial configuration of G_i and whose global effect is just to increase the indices of the a-nodes of G_i by one:

- Γ_1 activates a_1.
- Γ_{i+1} applies Γ_i, then activates b_{i+1}, followed by the nodes of the bottom path of G_i, and all the nodes a_j for $j \leq i$. Next, Γ_{i+1} activates a_{i+1}, b_{i+1}, and the nodes of the bottom path of G_i. Finally, Γ_{i+1} applies Γ_i again.

By construction, the length of Γ_{i+1} is more than twice the length of Γ_i. Thus, in the graph family G_k, the move complexity is exponential in k, and thus in n (since $n=5k$).

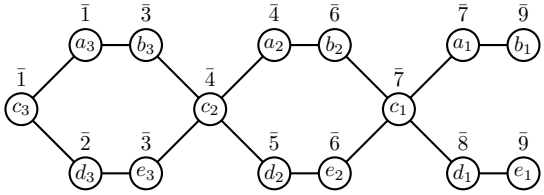

Figure 1: The initial configuration of G_3

8 Conclusion

We have proposed a versatile transformer that builds efficient silent self-stabilizing solutions. Precisely, our transformer allows for a good trade-off between time and energy consumption. In fact, we

can apply it to obtain fully polynomial solutions to various problems with round complexities asymptotically linear in D or even better.

Our transformer can be seen as a powerful tool to simplify the design of asynchronous self-stabilizing algorithms since it reduces the initial problem to implementing an algorithm just working in synchronous settings. More precisely, all tasks, even non-self-stabilizing ones, that terminate in a synchronous setting in various models, including the *LOCAL* model, can be made self-stabilizing using our transformer. Interestingly, the *LOCAL* model was initially introduced as an unrealistically strong model to prove lower bounds valid in weaker models. Using our transformer, it becomes a tool to provide upper bounds and even time-optimal self-stabilizing algorithms with low energy usage. Another interesting application of our transformer is the weakening of fairness assumptions of silent self-stabilizing algorithms (*e.g.*, asynchronous algorithms assuming a weakly fair or a synchronous daemon) without compromising efficiency (indeed, such algorithms can be provided as input of the transformer).

The perspectives of this work concern the space overhead and the move complexity. The space overhead of our solution depends on the synchronous execution time of the input algorithm. In the spirit of the resynchronizer proposed by Awerbuch and Varghese [10], we may build another transformer that would be space-efficient, possibly assuming more constraints on input algorithms. Concerning the move complexity, for many problems, the trivial lower bound in moves for the asynchronous (silent) self-stabilization is $\Omega(nD)$; while we usually obtain upper bounds in $O(n^3)$ moves with our transformer. Reducing the gap between those two bounds is another challenging perspective of our work.

Acknowledgments

This study has been partially supported by the French ANR projects SkyData (ANR-22-CE25-0008) and TEMPOGRAL (ANR-22-CE48-0001).

The authors would like to deeply thank Laurent Lyaudet for his help in the proofreading of earlier versions of this paper and for his helpful comments.

References

[1] Y. Afek and S. Dolev. 2002. Local Stabilizer. *J. Parallel and Distrib. Comput.* 62, 5 (2002), 745–765. https://doi.org/10.1006/jpdc.2001.1823

[2] Y. Afek, S. Kutten, and M. Yung. 1997. The Local Detection Paradigm and Its Application to Self-Stabilization. *Theoretical Computer Science* 186, 1-2 (1997), 199–229. https://doi.org/10.1016/S0304-3975(96)00286-1

[3] K. Altisen, A. Cournier, S. Devismes, A. Durand, and F. Petit. 2017. Self-Stabilizing Leader Election in Polynomial Steps. *Information and Computation* 254, 3 (2017), 330 – 366. https://doi.org/10.1016/j.ic.2016.09.002

[4] K. Altisen, S. Devismes, S. Dubois, and F. Petit. 2019. *Introduction to Distributed Self-Stabilizing Algorithms.* Morgan & Claypool. https://doi.org/10.2200/S00908ED1V01Y201903DCT015

[5] D. Angluin. 1980. Local and Global Properties in Networks of Processors. In *12th Annual Symposium on Theory of Computing (STOC'80).* 82–93. https://doi.org/10.1145/800141.804655

[6] B. Awerbuch, A. Bar-Noy, and M. Gopal. 1994. Approximate distributed Bellman-Ford algorithms. *IEEE Transactions on Communications* 42, 8 (1994), 2515–2517. https://doi.org/10.1109/26.310604

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

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

[9] B Awerbuch and M. Sipser. 1988. Dynamic Networks Are as Fast as Static Networks (Preliminary Version). In *29th Annual Symposium of Foundations of Computer Science (FOCS'88).* 206–220. https://doi.org/10.1109/SFCS.1988.21938

[10] B. Awerbuch and G. Varghese. 1991. Distributed Program Checking: a Paradigm for Building Self-stabilizing Distributed Protocols. In *32nd Annual Symposium on Foundations of Computer Science (FOCS'91).* 258–267. https://doi.org/10.1109/SFCS.1991.185377

[11] S. Bernard, S. Devismes, M. Gradinariu Potop-Butucaru, and S. Tixeuil. 2009. Optimal deterministic self-stabilizing vertex coloring in unidirectional anonymous networks. In *23rd International Symposium on Parallel&Distributed Processing (IPDPS'09).* 1–8. https://doi.org/10.1109/IPDPS.2009.5161053

[12] S. Bernard, S. Devismes, K. Paroux, M. Potop-Butucaru, and S. Tixeuil. 2010. Probabilistic Self-Stabilizing Vertex Coloring in Unidirectional Anonymous Networks. In *11th International Conference on Distributed Computing and Networking (ICDCN'10).* 167–177. https://doi.org/10.1007/978-3-642-11322-2_19

[13] L. Blin, P. Fraigniaud, and B. Patt-Shamir. 2014. On Proof-Labeling Schemes versus Silent Self-stabilizing Algorithms. In *16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'14).* 18–32. https://doi.org/10.1007/978-3-319-11764-5_2

[14] P. Boldi and S. Vigna. 2002. Universal dynamic synchronous self-stabilization. *Distributed Computing* 15 (2002), 137–153. https://doi.org/10.1007/s004460100062

[15] J. Burman and S. Kutten. 2007. Time Optimal Asynchronous Self-stabilizing Spanning Tree. In *21st International Symposium on Distributed Computing (DISC'07).* 92–107. https://doi.org/10.1007/978-3-540-75142-7_10

[16] B. Chen, H. Yu, Y. Zhao, and P. B. Gibbons. 2014. The Cost of Fault Tolerance in Multi-Party Communication Complexity. *J. ACM* 61, 3 (2014), 1–64. https://doi.org/10.1145/2597633

[17] R. Cole and U. Vishkin. 1986. Approximate and Exact Parallel Scheduling with Applications to List, Tree and Graph Problems. In *27th Annual Symposium on Foundations of Computer Science (FOCS'86).* 478–491. https://doi.org/10.1109/SFCS.1986.10

[18] A. Cournier, S. Devismes, and V. Villain. 2009. Light enabling snap-stabilization of fundamental protocols. *ACM Transactions on Autonomous and Adaptive Systems* 4, 1 (2009), 1–27. https://doi.org/10.1145/1462187.1462193

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

[20] Dolev D. and R. Reischuk. 1982. Bounds on Information Exchange for Byzantine Agreement. In *1st Annual Symposium on Principles of Distributed Computing (PODC'82).* 132–140. https://doi.org/10.1145/800220.806690

[21] A. K. Datta, S. Devismes, C. Johnen, and L. L. Larmore. 2019. Brief Announcement: Analysis of a Memory-Efficient Self-stabilizing BFS Spanning Tree Construction. In *21th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'19).* 99–104. https://doi.org/10.1007/978-3-030-34992-9_8

[22] A. K. Datta, L. L. Larmore, and P. Vemula. 2011. An $O(n)$-time self-stabilizing leader election algorithm. *J. Parallel and Distrib. Comput.* 71, 11 (2011), 1532–1544. https://doi.org/10.1016/j.jpdc.2011.05.008

[23] A. K. Datta, L. L. Larmore, and P. Vemula. 2011. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *Theoretical Computer Science* 412, 40 (2011), 5541–5561. https://doi.org/10.1016/j.tcs.2010.05.001

[24] S. Devismes, D. Ilcinkas, and C. Johnen. 2022. Optimized Silent Self-Stabilizing Scheme for Tree-Based Constructions. *Algorithmica* 84, 1 (2022), 85–123. https://doi.org/10.1007/s00453-021-00878-9

[25] Stéphane Devismes, David Ilcinkas, Colette Johnen, and Frédéric Mazoit. 2023. *Making local algorithms efficiently self-stabilizing in arbitrary asynchronous environments.* Technical Report 2307.06635. arXiv. arXiv:2307.06635 [cs.DC]

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

[27] E. W. Dijkstra. 1974. Self-stabilization in Spite of Distributed Control. *Commun. ACM* 17, 11 (1974), 643–644. https://doi.org/10.1145/361179.361202

[28] S. Dolev. 1993. Optimal time self stabilization in dynamic systems. In *7th International Workshop on Distributed Algorithms (WDAG'93).* 160–173. https://doi.org/10.1007/3-540-57271-6_34

[29] S. Dolev. 2000. *Self-Stabilization.* MIT Press.

[30] S. Dolev, M. G. Gouda, and M. Schneider. 1999. Memory Requirements for Silent Stabilization. *Acta Informatica* 36, 6 (1999), 447–462. https://doi.org/10.1007/s002360050180

[31] S. Dolev, A. Israeli, and S. Moran. 1993. Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Computing* 7, 1 (1993), 3–16. https://doi.org/10.1007/BF02278851

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

[33] M. J. Fischer, N. A. Lynch, and M. S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

[34] M. Gradinariu and S. Tixeuil. 2000. Self-stabilizing Vertex Coloration and Arbitrary Graphs. In *4th International Conference on Principles of Distributed Systems (OPODIS'00).* 55–70.

[35] S. Huang and N. Chen. 1992. A Self-Stabilizing Algorithm for Constructing Breadth-First Trees. *Inform. Process. Lett.* 41, 2 (1992), 109–117. https://doi.org/10.1016/0020-0190(92)90264-V

[36] C. Johnen. 1997. Memory Efficient, Self-Stabilizing Algorithm to Construct BFS Spanning Trees. In *16th Annual Symposium on Principles of Distributed Computing (PODC'97)*. 288. https://doi.org/10.1145/259380.259508

[37] S. Katz and K. J. Perry. 1993. Self-Stabilizing Extensions for Message-Passing Systems. *Distributed Computing* 7, 1 (1993), 17–26. https://doi.org/10.1007/BF02278852

[38] A. Kosowski and L. Kuszner. 2005. A Self-stabilizing Algorithm for Finding a Spanning Tree in a Polynomial Number of Moves. In *6th International Conference on Parallel Processing and Applied Mathematics (PPAM'05)*. 75–82. https://doi.org/10.1007/11752578_10

[39] A. Kravchik and S. Kutten. 2013. Time Optimal Synchronous Self Stabilizing Spanning Tree. In *27th International Symposium on Distributed Computing (DISC'13)*. 91–105. https://doi.org/10.1007/978-3-642-41527-2_7

[40] C. Lenzen, J. Suomela, and R. Wattenhofer. 2009. Local Algorithms: Self-stabilization on Speed. In *11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*. 17–34. https://doi.org/10.1007/978-3-642-05118-0_2

[41] N. Linial. 1992. Locality in Distributed Graph Algorithms. *SIAM J. Comput.* 21, 1 (1992), 193–201. https://doi.org/10.1137/0221015