Certified Round Complexity of Self-Stabilizing Algorithms

Karine Altisen
Université Grenoble Alpes, CNRS, Grenoble INP,[1] VERIMAG, 38000 Grenoble, France

Pierre Corbineau
Université Grenoble Alpes, CNRS, Grenoble INP,[1] VERIMAG, 38000 Grenoble, France

Stéphane Devismes
Université de Picardie Jules Verne, MIS, 80039 Amiens, France

Abstract

A proof assistant is an appropriate tool to write sound proofs. The need of such tools in distributed computing grows over the years due to the scientific progress that leads algorithmic designers to consider always more difficult problems. In that spirit, the *PADEC* Coq library has been developed to certify self-stabilizing algorithms. Efficiency of self-stabilizing algorithms is mainly evaluated by comparing their stabilization times in *rounds*, the time unit that is primarily used in the self-stabilizing area. In this paper, we introduce the notion of rounds in the PADEC library together with several formal tools to help the certification of the complexity analysis of self-stabilizing algorithms. We validate our approach by certifying the stabilization time in rounds of the classical Dolev *et al*'s self-stabilizing Breadth-first Search spanning tree construction.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Logic and verification

Keywords and phrases Certification, proof assistant, Coq, self-stabilization, round complexity

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.2

Funding This work has been partially funded by the ANR project SkyData (ANR-22-CE25-0008-01).

1 Introduction

Proving the correctness and analyzing the time complexity of distributed algorithms, especially fault-tolerant ones, is usually complex and subtle due to the many uncertainties we have to face, *e.g.*, locality of information, asynchrony of communications, faults, topological changes, just to quote a few. In this context, *certification* is an appropriate method to increase confidence of algorithmic designers in the functional and non-functional properties of their solutions. Indeed, the certification consists in formally writing proofs using a proof assistant, a software solution such as *Coq* [40, 7] or *Isabel/HOL* [34] that allows to develop formal proofs interactively and *mechanically check* them.

It is important to note that to guarantee the soundness of proofs, a proof assistant requires a level of detail that is drastically higher than in paper-and-pencil proofs and often necessitates a full reengineering of the initial proof. As a consequence, importing a paper-and-pencil proof into a proof assistant is usually an intricate task. However, to circumvent this difficulty, many libraries have been developed to facilitate the work of proof designers, *e.g.*, [8, 5, 1]. Such libraries mainly tackle two orthogonal goals: (1) they help to write formal proofs to prevent bugs while (2) keeping them readable and understandable for a non-expert in certification.

[1] Institute of Engineering Univ. Grenoble Alpes

© Karine Altisen, Pierre Corbineau, and Stéphane Devismes;
licensed under Creative Commons License CC-BY 4.0
37th International Symposium on Distributed Computing (DISC 2023).
Editor: Rotem Oshman; Article No. 2; pp. 2:1–2:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

PADEC [1] is a library for the certification of distributed self-stabilizing algorithms written in the *atomic-state model* [21], the most commonly used model in the self-stabilizing area. This library is based on the proof assistant Coq. It contains formal definitions and tools whose suitability has been demonstrated through several relevant use cases from the literature.

Self-stabilization is a versatile and lightweight fault-tolerant paradigm of distributed computing [21, 4]. A self-stabilizing algorithm enables a distributed system to resume a correct behavior within finite time, regardless its initial configuration; and therefore also after a finite number of transient faults place it in an arbitrary configuration. It is worth noting that self-stabilization makes no hypotheses on the nature (*e.g.*, memory corruption or topological changes) or extent of transient faults that could hit the system, and self-stabilizing systems recover from the effects of those faults in a unified manner. Such versatility comes at a price, *e.g.*, after transient faults cease, there is a finite period of time, called the *stabilization phase*, during which the safety properties of the system are violated. Hence, self-stabilizing algorithms are mainly compared according to their *stabilization time*, the worst-case duration of the stabilization phase.

To evaluate (stabilization) time, three main units are used in the atomic-state model: moves, (atomic) steps, and rounds. A move corresponds to a local state update at some process. Actually, it is rather a unit of work since it captures the amount of computations an algorithm needs. Steps essentially captures the same information as moves: a step is a global transition in an execution. *Rounds* [22, 13] evaluate the execution time according to the speed of the slowest processes. It is a non-atomic unit contrary to the two previous ones. Essentially, it is the adaptation to the atomic-state model of the notion of *time units* used in the message-passing model [39]. Roughly speaking, from a given configuration, a round is over as soon as all processes get a chance to move at least once.

The concept of steps have been already imported in PADEC [2]. Yet, as for moves, complexities in steps somehow neglect the parallel aspects of the distributed algorithm they evaluates. As a matter of fact, worst-case executions in steps are most of the time sequential; see, *e.g.*, [3, 2]. Perhaps, this is why the rounds are the most commonly used time units in the self-stabilizing area.

Contribution. In this paper, we enrich the PADEC library with the concept of rounds. Certifying complexities, especially stabilization times, in rounds is a major concern since it allows to increase confidence in the soundness of claimed bounds. As a matter of fact, paper-and-pencil proven complexity bounds are sometimes inaccurate due to implicit assumptions and a lack of details. For example, the stabilization time of Huang and Chen's Breadth-first Search (BFS) spanning tree construction [27] was conjectured to stabilize in $O(\mathcal{D})$ rounds, where $\mathcal{D}$ is the network diameter. Now, this algorithm is actually made of two non-mutually exclusive rules and the absence of priority on those rules leads to a possible execution that stabilizes in $\Omega(n)$ rounds, where $n > \mathcal{D}$ is the number of processes [19].

We add the formal definition of rounds in PADEC together with several companion formal tools whose aim is to ease the certification by making it as close as possible to the paper-and-pencil round complexity analyses one can find in the self-stabilizing area. To achieve this, we provide several certified meta-theorems consisting in general proof patterns allowing the users to mimic the usual way round complexities are proven. Thus, they can focus on the true difficulty of the result instead of drowning the proof in tedious details

requested by the proof assistant.

We validate our approach and illustrate the usefulness of our general formal tools by certifying the stabilization time in rounds of the straightforward translation into the atomic-state model of Dolev *et al*'s algorithm [22], which was initially written in the Read/Write atomicity model. This latter constructs a BFS spanning tree in a rooted bidirectional connected network. This task is fundamental in the self-stabilizing area since it is widely used as a basic building block of more complex self-stabilizing solutions; see, *e.g.*, [24, 17]. Notice that we also certify the self-stabilization of our use case assuming a weakly fair daemon.

Beyond the certification in Coq, our work leads to a better understanding of the intrinsic nature of non-atomic time units such as rounds.

Related Work. Many formal approaches have been used in the context of distributed computing. There exist exhaustive tool suites to validate a given distributed algorithm, such as the TLA+ toolbox [32]. Synthesis [9, 23] aims at automatically constructing algorithms based on a given specification, a fixed topology, and sometimes a restricted scheduling (*e.g.*, synchronous execution); this technique is now often based on SMT-solvers. Verification using model-checking [41, 31] is also fully automated and requires to fix settings similarly to synthesis. Both synthesis and model-checking only succeed with small topologies, due to computation limits. Notice also that model checking has been also successfully used to prove impossibility results applying on small-scale distributed systems [20]. In contrast, a proof assistant allows to validate a given algorithm for arbitrary-sized topologies, but is only semi-automated and may require heavy development for each algorithm, justifying then the development of helpful libraries.

The correctness of several non fault-tolerant distributed algorithms have been certified; *e.g.*, Castéran and Filou [10] consider distributed algorithms written in the local model, and a certified proof of Lamport's Bakery algorithm is given in [26]. Certification of fault-tolerant, yet non self-stabilizing, distributed systems has been addressed using various proof assistants, *e.g*, in Isabel/HOL [28, 12, 11, 29], TLA+ [16, 18], Coq [38], and Nuprl [36, 37]. This so-called *robust* fault tolerance approach aims at masking the effect of faults, whereas self-stabilization is non-masking by essence. Hence, the techniques used for these two approaches are widely different. In the robust context, many certification results are related to agreement problems, such as consensus or state-machine replication, in fully connected networks. Overall, most of these aforementioned works only certify the safety property of the considered problem [16, 18, 36, 37, 28, 38]. However, both liveness and safety properties are certified in [12, 11, 29]. To the best of our knowledge, the certification of time complexity of robust fault-tolerant algorithms has never been addressed. Finally, robust fault tolerance has been also considered in the context of mobile robot computing: using the PACTOLE Coq framework, impossibility results for swarm robotics that are subjected to Byzantine faults have been certified [6, 15]. Once again, to the best of our knowledge, certification of time complexity has never been addressed in the robot context.

Several frameworks to certify self-stabilizing algorithms using the Coq proof assistant have been proposed, *e.g.*, [14, 1]. In particular, the PADEC framework has already been used to certify the exact stabilization time in *steps* of the first Dijkstra's self-stabilizing token ring algorithm [2]. Certification of the correctness (safety and liveness) of the first Dijkstra's token ring algorithm has been previously achieved using various proof assistants, *i.e.*, PVS [35, 25, 30] and Isabel/HOL [33]. Interestingly, Fokkink *et al.* [25] have certified a quantitative property; precisely they show that the minimum number of states per node the algorithm needs to converge in any sequential execution is $N-1$, where N is the number of nodes. However, overall among these works, only PADEC addresses time complexity issues.

Coq Development. The development for this contribution represents about 11,000 lines of Coq code (loc, as measured by `coqwc`), precisely `#loc: spec = 2,698; proof = 7,892; comments = 484`. The Coq development related to the paper is available as an online browsing documentation at `http://www-verimag.imag.fr/~altisen/PADEC`. We encourage the reader to visit this webpage for a deeper understanding of our work.

Roadmap. The rest of the paper is organized as follows. In Section 2, we present our use case and the PADEC framework. Section 3 is devoted to the formalization of rounds in PADEC. In Section 4, we illustrate how to use the general tools given in the previous section to certify the round complexity of our use case. We make concluding remarks in Section 5.

2 A BFS Spanning Tree Algorithm and its Certification

In this section, we present an algorithm, denoted by $\mathcal{BFS}$, which will be used as the common use case all along the paper. Algorithm $\mathcal{BFS}$ allows us to define self-stabilization, the atomic-state model, and its semantics. We also use this algorithm as an illustrative example to introduce the PADEC framework and the method to certify a self-stabilizing algorithm.

2.1 Algorithm Definition and Informal Model

$\mathcal{BFS}$ is a self-stabilizing distributed algorithm that computes a BFS spanning tree in an arbitrary rooted, connected, and bidirectional network. By "bidirectional", we mean that each node can both transmit and acquire information from its adjacent nodes in the network topology, *i.e.*, its neighbors. The algorithm being distributed, these are the only possible direct communications. "Rooted" indicates that a particular node, called the root and denoted by r, is distinguished in the network. As in the present case, algorithms for rooted networks are (usually) semi-anonymous: all nodes have the same code except the root.

Algorithm 1 Algorithm $\mathcal{BFS}$, code for each node p.

Constant Local Input:

$p.neighbors \subseteq Channels$; $p.root \in \{true, false\}$

/* *p.neighbors as well as other sets below are implemented as lists* */

Local Variables:

$p.d \in \mathbb{N}$; $p.par \in Channels$

Macros:

$Dist_p = \min\{q.d + 1, q \in p.neighbors\}$

Par_{dist} returns the first channel in the list $\{q \in p.neighbors, q.d + 1 = p.d\}$

Action for the root, *i.e.*, for p such that $p.root = true$

Action $Root$: **if** $p.d \neq 0$ **then** $p.d := 0$

Actions for any non-root node, *i.e.*, for p such that $p.root = false$

Action CD: **if** $p.d \neq Dist_p$ **then** $p.d := Dist_p$

Action CP: **if** $p.d = Dist_p$ and $p.par.d + 1 \neq p.d$ **then** $p.par := Par_{dist}$

Algorithm $\mathcal{BFS}$ is written in the *atomic-state model*, where nodes communicate through locally shared variables: a node can read its variables and those of its neighbors, but can only write to its own variables. Every node can access its neighbors (to read its variables) through (local) channels.

The network is locally defined at each node p using constant inputs. The fact that the network is rooted is implemented using a constant Boolean input called $p.root$ which is false for every node except r. The input $p.neighbors$ is the set of channels linking p to its neighbors. When it is clear from the context, we do not distinguish a neighbor from the channels to that neighbor.

$\mathcal{BFS}$ is the straightforward translation into the atomic-state model of Dolev *et al*'s algorithm [22], which was initially written in the Read/Write atomicity model. Its code is given in Algorithm 1 as a set of three locally-mutually-exclusive actions. Each action is of the form: **if** *condition* **then** *statement*. In the following, we say that an action is *enabled* when its condition is true. By extension, a node is said to be enabled when at least one of its actions is enabled.

The semantics of the system is defined as follows. The current system *configuration* is given by the current value of all variables at each node. If no node is enabled in the current configuration, then the configuration is said to be *terminal* and the execution is over. Otherwise, a *step* is performed: a *daemon* (an oracle that models the asynchronism of the system) *activates* a non-empty set of enabled nodes. Each activated node then *atomically executes* the statement of its enabled action, leading the system to a new configuration, and so on and so forth.

Assumptions can be made about the daemon. Here, we assume that the daemon is *weakly fair* meaning that every continuously enabled nodes is eventually chosen by the daemon. More precisely, this means that every enabled node is eventually either activated or *neutralized*. A node p is neutralized in the step from configuration γ to configuration γ' if p is enabled in γ but not in γ' while being not activated during that step. Such situation occurs when a node is made disabled by the activation of some of its neighbors.

In Algorithm $\mathcal{BFS}$, each node p maintains two variables. First, each node p evaluates in $p.d$ its distance to the root. Then, each non-root node p maintains the pointer $p.par$ to designate as *parent* a neighbor that is closest to the root (*n.b.*, $r.par$ is meaningless). Algorithm $\mathcal{BFS}$ is a self-stabilizing BFS spanning tree construction in the sense that, regardless the initial configuration, it makes the system converge to a terminal configuration where par-variables describe a BFS spanning tree rooted at r. To that goal, nodes first compute into their d-variable their distance to the root. The root simply forces the value of $r.d$ to be 0; see Action $Root$. Then, the d-variables of other nodes are gradually corrected: every non-root node p maintains $p.d$ to be the minimum value of the d-variables of its neighbors incremented by one; see $Dist_p$ and Action CD. In parallel, each non-root node p chooses as parent a neighbor q such that $q.d = p.d - 1$ when $p.d$ is locally correct (*i.e.*, $p.d = Dist_p$) but $p.par$ is not correctly assigned (*i.e.*, $p.par.d$ is not equal to $p.d - 1$); see Action CP.

2.2 The PADEC Library

PADEC [1] is a general framework for the certification in Coq [7] of self-stabilizing algorithms. It includes the definition of the atomic-state model, tools for the definition of the algorithms and their properties, lemmas for common proof patterns, and case studies. The atomic-state model is carefully defined in PADEC to be as close as possible to the standard usage of the self-stabilizing community. Moreover, it is made general enough to encompass every usual hypothesis (*e.g.*, about topology or scheduling). First, a finite network is described using types `Node` and `Channel`, which respectively represent the nodes and the links between nodes. Then, the distributed algorithm is defined by providing a local algorithm at each node. This latter is defined using a type `State` that represents the local state of a node (*i.e.*, the values of its local variables) and a function `run` that encodes the local algorithm itself. Function `run` computes a new state depending on the current state of the node and that of its neighbors.

The model semantics defines a *configuration* as a function of type `Env := Node → State` that provides the (local) state of each node. An *atomic step* of the distributed algorithm is encoded as a binary relation over configurations that checks the conditions given in the informal model; see Section 2.1. An *execution* `e` is a finite or infinite *stream* of configurations, which models a *maximal* sequence of configurations where any two consecutive configurations are linked by the step relation. "Maximal" means that `e` is finite if and only if its last configuration is terminal. We use the coinductive[1] type `Exec` to represent an execution stream and the coinductive predicate `is_exec: Exec →` **Prop**[2] to check the above condition.

Daemons are also defined as predicates over executions using Linear Time Logic (LTL) operators provided in the PADEC library. For example, the fact that an execution is scheduled according to a weakly fair daemon is expressed by the following property: for every node `n`, it is `Always` (*a.k.a.* **G**lobally) the case that if `n` is enabled, then `Eventually` (*a.k.a.* **F**inally) `n` is activated or neutralized.

The semantics that uses the step relation is referred to as the *relational* semantics. As a way to strengthen the framework, PADEC also defines a *functional* semantics, which produces traces (*i.e.*, finite prefixes) of executions; those two semantics are proven to be equivalent.

Self-stabilization in PADEC is defined according to the usual practice: the property is formalized as a predicate `self_stabilization SPEC` that depends on the predicate `SPEC: Exec →` **Prop**, the specification of the algorithm. An algorithm is *self-stabilizing w.r.t. the specification* `SPEC` if there exists a set of legitimate configurations, encoded by some property `Leg: Env →` **Prop**, that satisfies the following three properties in every execution `e`:

- if `e` starts in a legitimate configuration (*i.e.*, if `Leg (H e)` holds, where `H e` is the first configuration of `e`), then `e` only contains legitimate configurations (*Closure*);
- `e` eventually reaches a legitimate configuration (*Convergence*); and
- if `Leg (H e)` holds, then `e` satisfies the intended specification, *i.e.*, `SPEC e` holds (*Specification*).

An algorithm is *silent* when each of its executions eventually reaches a terminal configuration; in such a case, the set of legitimate configurations is chosen to be the set of terminal configurations. Again, the closure, convergence, and silent properties use the LTL predicates `Always` and `Eventually`.

2.3 The Formal Algorithm

The formal algorithm is encoded in PADEC as a straightforward faithful translation in Coq of Algorithm 1. Together with its formal code, we have developed several technical results to facilitate the formal proof and complexity analysis of Algorithm $\mathcal{BFS}$.

We also had to encode the specification of $\mathcal{BFS}$ into PADEC. To that goal, we have defined the network in PADEC using the PADEC types `Node` and `Channel` as well as the predicate `is_channel: Node → Channel → Node →` **Prop**, where `is_channel n c n'` means that a channel `c` connects a node `n` to another `n'`. This network encodes the following graph relation: `R_Net := fun n n': Node => ∃ c: Channel, is_channel n c n'`. Namely, an edge from Node `n` to Node `n'` exists in the graph if and only if a channel `c` connects `n` to `n'`.

Then, the BFS spanning tree specification states that the algorithm should output a subgraph T of `R_Net` such that T is a locally-defined[3] BFS spanning tree of `R_Net` rooted at a given root node r.

---

[1] Coinduction allows to define and reason about potentially infinite objects.
[2] Predicates in Coq have type **Prop**.
[3] In our context, locally-defined means that each non-root node should be endowed with a pointer designating its parent in T.

To express the rooted BFS spanning tree, we have defined several tools about trees, distances, and diameter. In particular, `dist: Node → Node → nat` is a constructive distance function between nodes and $\mathcal{D}$: `nat` computes the diameter of the graph. We have also introduced a few graph properties; in particular the one expressing that a graph is a subgraph of another one using inclusion of relations. We also needed to introduce the notion of DAG (Directed Acyclic Graph) and rooted trees. A DAG is a directed graph that contains no (directed) cycle or equivalently, its transitive closure is not reflexive. A graph is a (directed) tree rooted at r if it is a DAG such that (1) every node has at most one outgoing edge (the out-neighbor, if it exists, is the parent of the node), and (2) for every non-root node x, there exists a path from x to r. Finally, the relation T is defined as a BFS spanning tree rooted at r of `R_Net` if T is (1) a spanning tree, *i.e.*, a subgraph of `R_Net` containing all nodes and a tree rooted at r, and (2) BFS, *i.e.*, the distance from every node to r is the same in T and `R_Net`.

Proving the self-stabilization and silence of Algorithm $\mathcal{BFS}$ for this specification then consists in proving that all its executions eventually reach a terminal configuration where parent pointers describe a BFS spanning tree T rooted at r, provided that `R_Net` is a connected and bidirectional graph rooted at r and the daemon is weakly fair.

3 Rounds

3.1 Rounds in the Atomic-state Model

In computer science, the time complexity is the computational measure that describes the amount of computer time an algorithm uses to solve a problem. Time complexity is estimated by counting the number of transitions performed by the algorithm, *i.e.*, the number of operations that are considered to be elementary in the computational model where the time complexity is evaluated. Of course, such operations are assumed to take a fixed amount of time to be performed. For example, in sequential algorithmics, it is commonly assumed that basic operations, such as divisions or multiplications, are elementary (despite their actual implementations are often not) and so take a constant amount of time.

Here, we are interested in the complexity measure called "round" which is accurately defined using natural language in the self-stabilizing community but requires mental gymnastics since by essence, rounds (*i*) are not atomic in the computational model they are considered (*i.e.*, the atomic-state model) and (*ii*) may be infinite in certain particular cases which – to some extent – should not be taken into account in the complexity evaluation.

We should underline that there exist other non-atomic complexity measures in the literature. For example, in message-passing systems, time complexity is often evaluated in terms of *time units* [39]. To define this later, it is assumed that the message transmission time is at most one time unit and the node execution time is zero. Now, the local algorithm at each node is made of several instructions and may contain loops. In particular, in case of bug, the node may get stuck in an infinite loop. Overall, this means that in general the correctness and the complexity analysis should be studied independently, following the separation of concerns principle: once the correctness has been established, some assumptions can be made for the purpose of defining the time complexity.

Evaluation of time complexity in rounds requires first to explain how a round is built from an execution in the atomic-state model (see Subsubsection 3.1.1), and then to define what it means to achieve a given property within a given amount of rounds (see Subsubsection 3.1.3).

3.1.1 Natural Language Definition

In the atomic-state model, every execution `e` of a given algorithm is split into rounds as follows. Let `U` be the set of enabled nodes in Configuration `H e`, the first configuration of `e`. The first round of `e` terminates at the first configuration `gr` where every node in `U` has been neutralized or activated. If no such a configuration exists in `e`, then the round is infinite and actually consists in the whole (infinite) execution `e`. Otherwise, the second round of `e` is the first round of the suffix of `e` starting from `gr`; and so on and so forth.

3.1.2 Infinite Rounds

It is worth noting that the existence of an infinite round is due to a starvation generated by fairness issues. For example, imagine a situation where the activation of some node x makes another node y enable and conversely; another node z may stay continuously enabled without being ever activated by the daemon making the current round infinite. Such a situation occurs when the daemon is unfair[4] and the algorithm is actually unable to enforce fairness between enabled nodes. In contrast, when the daemon is weakly fair, fairness is guaranteed by definition. So, in every execution under the weakly fair daemon assumption, every round is finite. Remark also that when an execution contains an infinite round, it is the last one and the execution actually only contains a finite number of rounds. Conversely, if every round of an execution is finite, then the execution contains infinitely many rounds. This is in particular true for finite execution. In this latter case, infinitely many empty rounds are defined from the terminal configuration, by definition.

3.1.3 Amount of Rounds to achieve a Property

A round is a unit of time, *i.e.*, it is used to evaluate how many time is required to achieve a given property. Consider an execution `e` where a property `P: Exec` $\rightarrow$ **Prop** is eventually satisfied, *i.e.*, `e` has a suffix that satisfies `P`. The goal is then to evaluate in how many rounds `P` becomes satisfied. For example, evaluating the stabilization time in rounds of an execution of some self-stabilizing algorithm consists in counting the number of initiated rounds before a legitimate configuration is reached in the execution. If `P` is true at the first configuration of `e`, then `P` is satisfied in 0 round. Otherwise, `P` is satisfied in i rounds, where i is the index of the first round of `e` containing a configuration from which `P` is true.

As in the present paper, time complexity proofs often cope with upper bounds rather than exact ones. So, we need to express the fact that an execution *requires at most j rounds to reach* `P`. Let `e` be an execution *containing at least j rounds.* We say that `e` *requires at most j rounds to reach* `P` if `e` contains a suffix which starts before the end of the j-th round and satisfies `P`.

Remark that the assumption on `e` is necessary to cope with the possible existence of an infinite round, in which case the total number of rounds in the execution is finite and maybe smaller than j. However, contrary to more usual cases, where each unit of time is assumed to be finite, our assumption here is weaker: we only require the existence of at least j rounds, so `e` can contain an infinite round as far as it is preceded by at least $j - 1$ finite rounds.

We naturally extend the definition to all executions by fixing the property to true for each execution containing less than j rounds. Therefore, to falsify this extended property, one needs to exhibit an execution in which `P` is still not satisfied despite j rounds have elapsed.

---

[4] Unfair means that no fairness is imposed to the daemon, except the activation of at least one enabled node at each step.

3.2 Rounds in PADEC

We now formally express the previous definitions so that they could be encoded in PADEC.

3.2.1 Set of Unsatisfied Nodes

To compute a round, from its beginning to its end (which may never occur), we use the set `U`, called the *set of unsatisfied nodes*, which is computed as follows:

- At the beginning of the round (say at Configuration `gr`), `U` is initialized to the set of enabled nodes in `gr`; using Function `UNSAT_init gr`.
- Then, at each step (say from `g` to `g'`), the set is updated by removing the nodes that have been activated or neutralized during the step; using Function `UNSAT_update g' g`.
- The current round ends, say at Configuration `g"`, when `U` becomes empty. In this case, `U` is refilled at configuration `g"` with `UNSAT_init g"` since the next round begins.

Notice that sets of nodes are represented in PADEC using Boolean functions: `Node` $\to$ `bool`. We have defined the `PADEC.BoolSet` library to provide tools that handle sets of elements, in particular set operations (such as union, intersection, set difference, ...). The library also provides decidability results in case the set of elements is finite, which is the assumption we made for Type `Node`.

3.2.2 Predicate `At_most_rounds`

Predicate `at_most_rounds P n e` defines the fact that an execution `e` requires at most n rounds to reach a predicate `P`. It is based on an intermediate predicate, called `at_most_rounds_aux`, that has an additional parameter `U`, a set of unsatisfied nodes. Thus, `at_most_rounds` is defined as follows: `at_most_rounds P` n `e := at_most_rounds_aux P (UNSAT_init (H e))` n `e`, where the set of unsatisfied nodes (the second parameter) is initialized at the beginning of the computation of the rounds by `UNSAT_init (H e)`, *i.e.*, the set of enabled nodes in the first configuration of `e`.

We now give more details about the predicate `at_most_rounds_aux`. Recall the informal definition: "if `e` contains at least n rounds, then `e` contains a suffix which starts before the end of the n-th round and satisfies `P`". Of course, we will perform a single traversal of the execution to check both the "if" and "then" parts of the sentence. Actually, this is the role of `at_most_rounds_aux`. Since rounds may be infinite, this predicate is typically coinductive. The predicate `at_most_rounds_aux` is evaluated thanks to the following three rules. At each step of the execution, one of the rules applies.

Rule 1. The first rule, `rnd_here`, detects that the targeted predicate `P` is reached, *i.e.*, if for some execution `e`, `e` satisfies `P`, then `at_most_rounds_aux P U` n `e` holds for any values of `U` and n (even $n = 0$). Indeed, since `P e` holds, `e` requires at most n rounds to reach `P`, for any $n \geq 0$.

The two other rules achieve a traversal of the execution and update the set of unsatisfied nodes meanwhile, according to the informal definition of a round.

Rule 2. The second rule, `rnd_in`, applies when going through the current round but the round is not over (*i.e.*, right after its update, the set of unsatisfied nodes is still not empty). In this case, we decompose the execution as `g • e`, where `g` is its first configuration and `e` the subsequent suffix. Then, to satisfy `at_most_rounds_aux P U` n `(g • e)` – which claims that, given the current set of unsatisfied nodes `U`, the execution requires at most n rounds to reach `P` –

we need that `at_most_rounds_aux P U' ` n `e` holds, where $n > 0$ and `U' := UNSAT_update (H e) g` is not empty, meaning that, given the updated set of unsatisfied nodes `U'`, the execution `e` requires at most n rounds to reach `P`. Indeed,

- the non-empty set `U'` is obtained by removing the nodes from `U` that have been activated or neutralized during the step from `g` to `H e`;
- and the rule applies during the current round, so the number of rounds n is positive and does not change.

Rule 3. The last rule, `rnd_chge`, applies at the end of a round. So, the number of elapsed rounds increases by one. In this case, to satisfy `at_most_rounds_aux P U` $(n+1)$ `(g • e)` – which claims that, given the current set of unsatisfied nodes `U`, the execution requires at most $n+1$ rounds to reach `P` – we need that `at_most_rounds_aux P U"` n `e` holds, where `U" := UNSAT_init(H e)`. Indeed, this time, the set of unsatisfied nodes, `UNSAT_update (H e) g`, becomes empty during the step from `g` to `H e`, so one round has passed. Consequently, the new set of unsatisfied nodes `U"` should be filled with the enabled nodes of Configuration `H e` and, given `U"`, `P` should be satisfied within at most n rounds in `e` (In particular, if $n = 0$, `e` should satisfy `P`; see Rule 1).

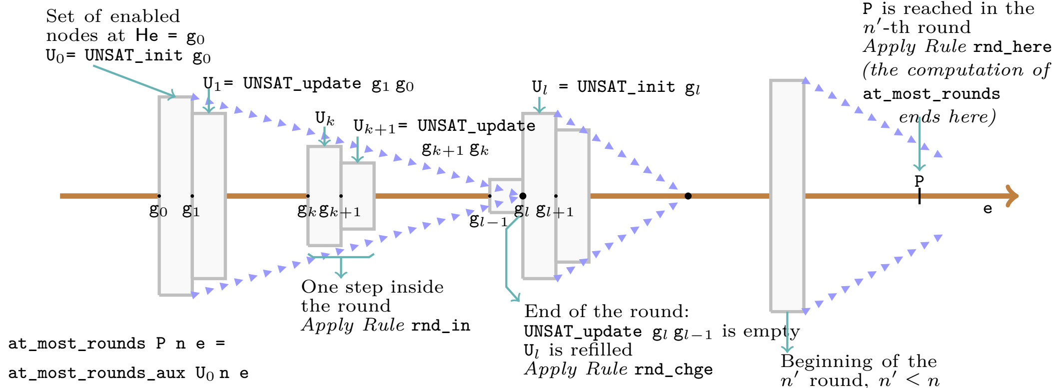


Figure 1 Round principle: evaluation of `at_most_rounds` using the rules of `at_most_rounds_aux`.

Illustration. Figure 1 shows how the rules of `at_most_rounds_aux` apply to evaluate `at_most_rounds`. The horizontal line represents the execution `e` = g_0 g_1 ... starting from g_0 = `H e`. First, to evaluate Predicate `at_most_rounds P n e`, the set of unsatisfied nodes U_0 is initialized to `UNSAT_init` g_0; see the leftmost part of the figure. Then, along the steps inside the round, Rule `rnd_in` is applied and the set of unsatisfied nodes is monotonically nonincreasing. Moreover, sometimes its cardinal decreases; see *e.g.*, the step from g_k to g_{k+1} where U_{k+1} := `UNSAT_update` g_{k+1} g_k. At the end of the round, *i.e.*, at Configuration g_l, Rule `rnd_chge` applies since the update of the set of unsatisfied nodes using `UNSAT_update` g_l g_{l-1} produces an empty set. The set of unsatisfied nodes is then refilled using U_l := `UNSAT_init` g_l. Finally, `P` becomes satisfied during the `n'`-th round with `n'`≤`n`. When it happens, Rule `rnd_here` applies: the evaluation stops and `at_most_rounds P n e` is satisfied.

Remark that if we remove the rightmost part of the figure, *i.e.*, if `P` is never satisfied along `e`, Rule `rnd_here` is never applied. In this case, the evaluation never stops, which is allowed since `at_most_rounds` is coinductive. Note also that the other rules may never be applied. Rule `rnd_chge` may never be applied in case the first (and last) round is infinite. Rule `rnd_in` is never applied when the execution is synchronous, *i.e.*, at each step all enabled nodes are activated.

Straightforward properties of `at_most_rounds`. We can prove that the predicate has several basic, yet interesting and useful, properties, *e.g.*:

- If `at_most_rounds P` n `e` holds, then for every $n' \geq n$, `at_most_rounds P` n' `e`.
- If Predicate P_1 implies Predicate P_2 all along the execution `e`, then `at_most_rounds` P_1 n `e` implies `at_most_rounds` P_2 n `e`.

No need to detail the proofs of these properties: they are simple coinductive proofs that directly use the definition of `at_most_rounds_aux`.

3.2.3 Functional Definition (Computation)

We also provide a functional definition, denoted by `count_rounds P e`, which *returns* in how many rounds of `e` the predicate `P` is reached (for the first time). Obviously, this function requires assumptions which enable its actual computation (precisely, which guarantees the computation eventually stops): it requires the assumption that `e` eventually reaches `P`, this assumption being expressed using a property that actually allows the computation to detect whether `P` is satisfied. This means that the reachability of `P` is encoded by an assumption – denoted by `FP` – that can be used in the function to compute its result. For `FP`, we use a computable inductive predicate which is satisfied whenever the execution actually reaches `P`. Then, `count_rounds` is defined as a fixpoint using a structural induction over the `FP` assumption. In this function, we deal separately with the case where zero is returned: when the assumption `FP` claims that `P` is (immediately) reached, `count_rounds P e` returns 0. Otherwise, it returns the result computed by an auxiliary inductive function that requires one more parameter: the accumulator parameter that encodes the set of unsatisfied nodes. This auxiliary function computes the successive values of the set of unsatisfied nodes (as explained in Subsubsection 3.2.1) until `P` is reached. We have two cases:

- Either the assumption `FP` claims that `P` is reached, hence the auxiliary function returns one round (to count the current round).
- Or the set of unsatisfied nodes is refreshed by removing both activated and neutralized nodes. Moreover, if this new set is empty, the function starts a new round: it adds one to the current result and resets the set of unsatisfied nodes with the enabled nodes of the current configuration.

As a matter of fact, we can prove that the functional and relational definitions are related as expected. Namely, for every execution `e`, every number of rounds n, and every predicate `P`, we have

`at_most_rounds P` n `e` $\longleftrightarrow$ **`exists`** n', $n' \leq n \wedge$ `count_rounds P e =` n'.

Note that this latter property is an equivalence and implies that if `count_rounds P e =` n', then $\forall\ n \geq n'$, `at_most_rounds P` n `e`. Again, there is no need to detail the proof of this property as it is directly obtained by induction on the `FP` assumption.

3.2.4 Induction Scheme

During the development of this round library, we paid a particular attention on facilitating, as much as possible, the use of the round predicate in users' own proofs. To do so, we have developed tools to avoid coinductive proofs which are particularly tricky in Coq. Actually, the fact that an execution requires at most n rounds to reach `P` is usually proven using induction on n. In that spirit, we have developed an induction scheme that follows the classical way inductions on rounds are written in paper-and-pencil proofs.

The particular induction scheme we propose is as follows: assume that we want to prove that an execution requires at most B rounds to reach `P`. Assume also that we have a family of predicate P_n (indexed on natural numbers n) such that P_B implies `P`. We can prove the following lemma.

▶ **Lemma 1** (Lemma `schema_round_induction`). *Assume an execution* `e` *satisfies the following two properties:*

- `e` *satisfies* P_0 *(Base Case).*
- *All along the execution* `e`*, and for every value* $n < B$*, if* `e` *has a suffix* `c` *satisfying* P_n*, then* `c` *requires at most one round to reach* P_{n+1}*,* i.e., `at_most_rounds` P_{n+1} 1 `c` *holds (Induction Step).*

Then, `e` *requires at most* B *rounds to reach* P_B *and so* `P`*.*

The proof of this induction schema is shown by induction on the parameter n. The base case is immediate from the first property on P_0 (*Base Case*). The induction step of the proof uses the second property and is a direct application of the next lemma.

▶ **Lemma 2** (Lemma `schema_round_step`). *Let* `e` *be an execution and* n *be a number of rounds. Let* P_1 *and* P_2 *be two predicates over executions. Assume*

(A) `e` *requires at most* n *rounds to reach* P_1 *and*

(B) *all along the execution* `e`*, if* `e` *has a suffix* `c` *satisfying* P_1*, then* `c` *requires at most one round to reach* P_2*,* i.e., `at_most_rounds` P_2 1 `c` *holds.*

Then, `e` *requires at most* $n + 1$ *rounds to reach* P_2*.*

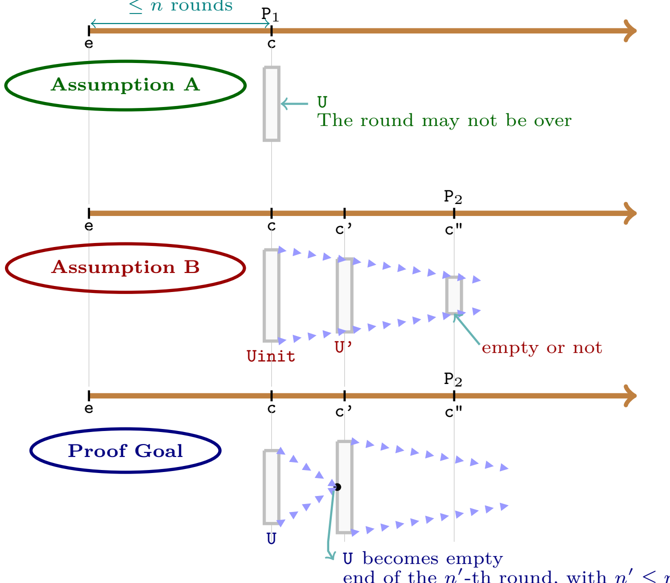


Figure 2 Proof principle of Lemma `schema_round_step`.

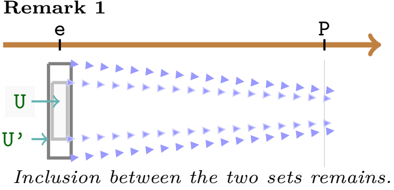
The key point to prove Lemma `schema_round_step` (and the fact that makes the result non-obvious) is that the set of unsatisfied nodes may change from one assumption to the other. Indeed (see Figure 2), using Assumption (A), `e` eventually reaches P_1, *i.e.*, there is some suffix `c` of `e` where P_1 `c` holds. Let `U` be the current set of unsatisfied nodes when it happens (for the first time). Using Assumption (B), we know that P_1 `c` holds. Hence, `c` requires at most 1 round to reach P_2. Now, the computation of this assertion uses `UNSAT_init (H c)` as set of unsatisfied nodes instead of `U`. To obtain that `e` requires at most one more round to reach P_2 from its beginning, we need that `c` requires at most 1 round to reach P_2 using `U` and not `UNSAT_init (H c)`. Obviously, there is no reason for `U` to be equal to `UNSAT_init (H c)`. Nevertheless, we can compare them: since unsatisfied nodes are enabled, `U` is included into `UNSAT_init (H c)`.

Before presenting an overview of the proof of Lemma `schema_round_step`, we have two remarks.

▶ Remark 1. Let `U` and `U'` be two sets of unsatisfied nodes such that `U` ⊆ `U'`. If `e` requires at most one round to reach `P` with the current set `U`, then this is also true with the set `U'`. Namely, if `at_most_rounds_aux P U 1 e`, then `at_most_rounds_aux P U' 1 e`.

Remark 1 is a fairly (easy to prove) intuitive property, since we can prove that the inclusion between the two families of sets – built from `U` and `U'`, respectively – remains; see Figure 3.

▶ Remark 2. Using the same notations as above, if `U` ⊆ `U'` ⊆ `UNSAT_init (H e)` and `at_most_rounds_aux P U' 1 e` holds, then `at_most_rounds_aux P U 2 e` also holds.



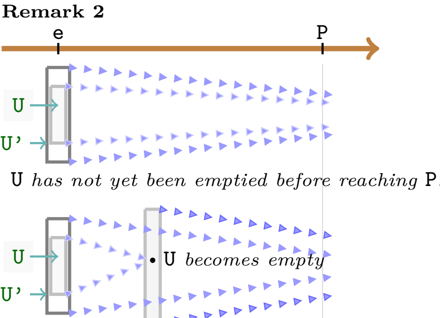


Figure 3 Principles for Remarks 1 and 2.

Proof Overview of Remark 2. (See the sketch given in Figure 3 for an illustration.) Starting the computation of rounds at `e` with `U'`:

- Either `U` does not become empty (and so neither do `U'`) before reaching `P`, hence using `U` as set of unsatisfied nodes, `P` is reached within at most one round of `e`.
- Or `U` becomes empty before reaching `P`, say at Configuration g_c. So, using `U` as set of unsatisfied nodes, one round of `e` is over and `U` is refilled. Let U_{init} be the value of `U` after being refilled at g_c. Let U'_c be the value of `U'` at g_c. As `U'` was included in `UNSAT_init (H e)`, it still contains enabled nodes only. So, U'_c is included into U_{init}. We can then apply Remark 1: since `P` is reached from g_c using U'_c in at most one round, this is also the case from g_c using U_{init}. Overall, this means that using `U` as set of unsatisfied nodes, `P` is reached within at most two rounds in `e`. ◀

We can now conclude with the proof of Lemma `schema_round_step`.

Proof Overview of Lemma `schema_round_step`. The proof of Lemma `schema_round_step` uses coinduction. For the sake of explanation, we summarize the proof using the following two scenarios.

First scenario. if `e` contains less than $n+1$ rounds, then the result immediately holds.

Second scenario. Assume `e` contains at least $n+1$ rounds (*n.b.*, the $(n+1)$-th may be infinite). Then, `e` contains at least n rounds and the first n rounds of `e` are finite. By Assumption (A), `e` actually reaches $\mathtt{P}_1$ within at most n rounds. So, `e` consumes at most n rounds to reach $\mathtt{P}_1$ at the first configuration of some suffix `c`. Let `U` be the set of unsatisfied nodes at `H c`.

Assumption (B) ensures that in `c`, $\mathtt{P}_2$ is actually reached in at most 1 round with set of unsatisfied nodes `UNSAT_init (H c)`. Then, we have two cases. If the nth round terminates at `H c`, `U` $=$ `UNSAT_init (H c)` and we are done. Otherwise, at most $n' < n$ rounds have terminated at Configuration `H c`. Now, since `U` only contains nodes that are enabled in `H c`, we have `U` $\subseteq$ `UNSAT_init (H c)` and we can apply Remark 2 with `U' = UNSAT_init (H c)`. So, with `U` as set of unsatisfied nodes, $\mathtt{P}_2$ is reached within at most two more rounds, and we are done. ◀

4 Round Complexity of the Algorithm

We now illustrate how to use the previous tools by sketching the certification of the stabilization time in rounds of Algorithm $\mathcal{BFS}$. Precisely, we show that $\mathcal{BFS}$ requires at most $\mathcal{D}+2$ rounds to reach a terminal configuration starting from an arbitrary configuration. The full certified proof is detailed in Appendix B. Here, we focus on generic formal tools and show how to apply them. In particular, through out the section, we will introduce two additional useful general tools.

Another goal of this section is to convince the reader that our formalization allows to write certified proofs that are close to the standard usages in the self-stabilizing community. In that spirit, the Coq proof outline given below broadly follows the approach proposed in [4].

The proof is split into the following two main parts. First, we prove (Part A) that $\mathcal{BFS}$ requires at most $\mathcal{D}+1$ rounds to reach a configuration from which the d-variables are correctly assigned forever, *i.e.*, for every node p, $p.d$ is (forever) equal to the distance from p to the root. Then, we prove that once d-variables are correctly assigned forever, $\mathcal{BFS}$ requires at most one more round to reach a terminal configuration (Part B).

To prove Part A (and according to [4]) we use the induction scheme given in **Lemma** `schema_round_induction` with the predicate P_k `e := Always (check_dist` k`) e` and the value B `:=` $\mathcal{D}+1$, where Predicate `check_dist` k `e` holds iff every node p satisfies the condition `CD` k `e` p. This latter condition checks that `(dist` p r `<` k $\wedge$ $p.d$ `= dist` p r`)` $\vee$ `(dist` p r $\geq$ k $\wedge$ $p.d \geq k$`)` holds in `H e`, the first configuration of the execution `e`.

To apply **Lemma** `schema_round_induction`, we have to establish the assumptions of this lemma, namely the *Base Case* and the *Induction Step*. To ease this proof, we have introduced two other generic results in the PADEC Round library. The goal of the first one is to simplify the proof when dealing with local predicates such as `check_dist` and P_k. Actually, P_k checks properties that are local at each node. Precisely, it checks that every node p satisfies the local property `CD` k `e` p all along `e`. For such a local property `Q: Node` $\rightarrow$ `Exec` $\rightarrow$ **Prop**, we can prove that

```
at_most_rounds (Always (fun e' => ∀p, Q p e')) k e ⟷
∀p, at_most_rounds (Always (Q p)) k e
```

Namely, the universal quantifier over the nodes can be shifted to the outer border of the formula, which is easier to handle, since the proof can now be done using a single node. This result, proven by a simple coinduction, is now provided in the PADEC Round library.

The second generic result is the following proof scheme:

▶ **Lemma 3** (Lemma `at_most_rounds_scheme_per_node`). *Let* `P: Node → Exec →` **Prop** *be a predicate and* p *be a node. If*

- *either* `(P` p `e)` *holds when* p *is disabled in* `(H e)`
- *or* `(P` p`)` *becomes true whenever* p *is activated or neutralized during some step of* `e`*,*

then at most one round is required to reach a configuration where `(P` p`)` *holds.*

Indeed, either the node p is disabled at the beginning of the round and then `P` p holds (due to the first assumption); or p is enabled and so belongs to the set of unsatisfied nodes. Now, as this set is empty at the end of the round, the node has been removed because it has been activated or neutralized meanwhile and so `P` p has become true during the round (second assumption). Overall, we obtain that `P` p requires at most one round to be reached.

Using these tools, we have obtained Part A by proving that for every execution `e`,

- the *Base Case* holds, namely, the property P_0 `e := Always (check_dist 0) e` is satisfied;
- and the *Induction Step* also holds, *i.e.*, for every value of $k < \mathcal{D} + 1$, if `e` has a suffix `c` satisfying P_k, then `c` requires at most one round to reach P_{k+1}.

The detailed proof (which is based on a combinatorial study of possible values for the d-variables) is given in Appendix B. Notice however that, by successfully applying our generic proof schemes, we were able to make the certification of the main proof simpler and really close to the one in [4].

Part B and Final Result. Part A proves that Algorithm $\mathcal{BFS}$ requires at most $\mathcal{D} + 1$ rounds to reach a configuration from which `Always (check_dist` $(\mathcal{D}+1)$`) e` is achieved (*i.e.*, the d-variables are correctly assigned forever). After that, at most one more round is required to reach a configuration where the par-variables are correctly set and so to achieve the silence. The proof (Part B) uses the same mechanism as before and mainly relies on the following two simple facts:

- As `check_dist` $(\mathcal{D}+1)$ `e` holds forever, the d-variables no more change.
- Furthermore, when `check_dist` $(\mathcal{D}+1)$ `e` holds, a node can be activated only once (using Action CP): the action, and so the node too, is then disabled forever.

Afterwards, we have merged Parts A and B using Lemma `schema_round_step` to conclude that every execution `e` requires at most $\mathcal{D} + 2$ rounds to reach a terminal configuration, concluding then the proof of the round complexity for Algorithm $\mathcal{BFS}$.

To complete this result, we have also proven that Algorithm $\mathcal{BFS}$ is silent and self-stabilizing w.r.t. its specification. For the convergence property, we can easily show, using the definitions of `weakly_fair` and `at_most_rounds`, that if an execution has been scheduled using a weakly fair daemon and requires at most B rounds to converge to some property `P`, then the execution eventually reaches `P`, *i.e.*, the execution converges to `P` within finite time. Thus, we can deduce that under the weakly fair daemon, Algorithm $\mathcal{BFS}$ converges to a legitimate configuration. The rest of the proof, *i.e.*, Algorithm $\mathcal{BFS}$ satisfies both the specification and closure part of the self-stabilization property, is proven by induction; see Appendix A for details.

Overall, we have certified the following theorem:

▶ **Theorem 4.** *Let G be a connected bidirectional network rooted at some node r. Under a weakly fair daemon, $\mathcal{BFS}$ is a silent self-stabilizing BFS spanning tree construction whose stabilization time is at most $\mathcal{D} + 2$ rounds, where $\mathcal{D}$ is the diameter of G.*

5 Conclusion

Certification is an important tool to increase confidence of algorithmic designers in the correctness of their solutions. This is even more important in fault-tolerant distributed algorithmic, where models, algorithms, and intended specifications are most of the time both complex and subtle. In this context, the PADEC library has been proposed to help the certification (in Coq) of self-stabilizing distributed algorithms written in the atomic-state model. This library encompasses all necessary formal tools to establish the correctness (especially the convergence) and the time complexity of monolithic, as well as composite, self-stabilizing algorithms. The usefulness of all these formal tools has been validated thanks to many non-trivial use cases from the literature. The last contribution, presented here, has been to import in PADEC the most commonly used time complexity measure in the self-stabilizing area: the rounds. The main encountered difficulties were due to the non-atomic nature of the rounds that made them not compositional. The definition of rounds has been provided in PADEC together with many formal companion tools, *e.g.*, `Lemmas schema_round_induction`, `schema_round_step`, `at_most_rounds_scheme_per_node`. The suitability of these general tools has been demonstrated with an appropriate use case from the literature: we have certified the stabilization time of Dolev *et al*'s algorithm [22]. Although the intrinsic nature of rounds implies a coinductive definition, the companion tools provided in the library avoid the user to deal with coinductive proofs, which may be tricky in Coq. Our use case is convincing in this sense since applying the companion tools allows to prevent the use of coinduction in its certified proof. Actually, the only (rather simple) proofs requiring coinduction are due to the `Always (check_dist` *k*`)` property which is itself coinductive.

References

1. Karine Altisen, Pierre Corbineau, and Stéphane Devismes. A framework for certified self-stabilization. *Log. Methods Comput. Sci.*, 13(4), 2017. doi:10.23638/LMCS-13(4:14)2017.
2. Karine Altisen, Pierre Corbineau, and Stéphane Devismes. Certification of an exact worst-case self-stabilization time. In *ICDCN '21: International Conference on Distributed Computing and Networking, Virtual Event, Nara, Japan, January 5-8, 2021*, pages 46–55. ACM, 2021. doi:10.1145/3427796.3427832.
3. Karine Altisen, Alain Cournier, Stéphane Devismes, Anaïs Durand, and Franck Petit. Self-stabilizing leader election in polynomial steps. *Inf. Comput.*, 254:330–366, 2017. doi:10.1016/j.ic.2016.09.002.
4. Karine Altisen, Stéphane Devismes, Swan Dubois, and Franck Petit. *Introduction to Distributed Self-Stabilizing Algorithms*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019. doi:10.2200/S00908ED1V01Y201903DCT015.
5. Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified impossibility results for byzantine-tolerant mobile robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2013. doi:10.1007/978-3-319-03089-0_13.
6. Cédric Auger, Zohir Bouzid, Pierre Courtieu, Sébastien Tixeuil, and Xavier Urbain. Certified impossibility results for byzantine-tolerant mobile robots. In Teruo Higashino, Yoshiaki Katayama, Toshimitsu Masuzawa, Maria Potop-Butucaru, and Masafumi Yamashita, editors, *Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings*, volume 8255 of *Lecture Notes in Computer Science*, pages 178–190. Springer, 2013.

7 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

8 Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011. doi:10.1017/S0960129511000120.

9 Roderick Bloem, Nicolas Braud-Santoni, and Swen Jacobs. Synthesis of self-stabilising and byzantine-resilient distributed systems. In *Computer Aided Verification - 28th International Conference, CAV 2016*, pages 157–176, 2016.

10 Pierre Castéran and Vincent Filou. Tasks, types and tactics for local computation systems. *Stud. Inform. Univ.*, 9(1):39–86, 2011.

11 Bernadette Charron-Bost, Henri Debrat, and Stephan Merz. Formal verification of consensus algorithms tolerating malicious faults. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, pages 120–134, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

12 Bernadette Charron-Bost and Stephan Merz. Formal verification of a consensus algorithm in the heard-of model. *Int. J. Software and Informatics*, 3(2-3):273–303, 2009.

13 Alain Cournier, Ajoy K. Datta, Franck Petit, and Vincent Villain. Snap-Stabilizing PIF Algorithm in Arbitrary Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 199–206, 2002.

14 Pierre Courtieu. Proving self-stabilization with a proof assistant. In *16th International Parallel and Distributed Processing Symposium (IPDPS 2002), 15-19 April 2002, Fort Lauderdale, FL, USA, CD-ROM/Abstracts Proceedings*, volume 1, page 8pp. IEEE Computer Society, 2002.

15 Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of gathering, a certification. *Inf. Process. Lett.*, 115(3):447–452, 2015.

16 Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA + proofs. In *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 147–154, 2012.

17 Ajoy Kumar Datta, Stéphane Devismes, Karel Heurtefeux, Lawrence L. Larmore, and Yvan Rivierre. Competitive self-stabilizing k-clustering. *Theor. Comput. Sci.*, 626:110–133, 2016. doi:10.1016/j.tcs.2016.02.010.

18 Carole Delporte-Gallet, Hugues Fauconnier, Eli Gafni, and Leslie Lamport. Adaptive register allocation with a linear number of registers. In *Distributed Computing - 27th International Symposium, DISC 2013*, 2013.

19 Stéphane Devismes and Colette Johnen. Silent self-stabilizing BFS tree algorithms revisited. *J. Parallel Distributed Comput.*, 97:11–23, 2016. doi:10.1016/j.jpdc.2016.06.003.

20 Stéphane Devismes, Anissa Lamani, Franck Petit, Pascal Raymond, and Sébastien Tixeuil. Optimal grid exploration by asynchronous oblivious robots. In *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium, SSS 2012, Toronto, Canada, October 1-4, 2012. Proceedings*, volume 7596 of *Lecture Notes in Computer Science*, pages 64–76. Springer, 2012.

21 Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974. doi:10.1145/361179.361202.

22 Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only Read/Write atomicity. *Distributed Computing*, 7(1):3–16, 1993.

23 Fathiyeh Faghih, Borzoo Bonakdarpour, Sébastien Tixeuil, and Sandeep S. Kulkarni. Specification-based synthesis of distributed self-stabilizing protocols. In *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016*, 2016.

24 Lin Fei, Sun Yong, Ding Hong, and Ren Yizhi. Self stabilizing distributed transactional memory model and algorithms. *Journal of Computer Research and Development*, 51(9):2046, 2014.

25 Wan Fokkink, Jaap-Henk Hoepman, and Jun Pang. A note on k-state self-stabilization in a ring with k=n. *Nordic Journal of Computing*, 12(1):18–26, 2005.

26 Wim H. Hesselink. Mechanical verification of lamport's bakery algorithm. *Science of Computer Programming*, 78(9):1622–1638, 2013.

27 Shing-Tsaan Huang and Nian-Shing Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41(2):109–117, 1992.

28 Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005, 2005.

29 Philipp Küfner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms - from pseudo code to checked proofs. In Jos C. M. Baeten, Thomas Ball, and Frank S. de Boer, editors, *Theoretical Computer Science - 7th IFIP TC 1/WG 2.2 International Conference, TCS 2012, Amsterdam, The Netherlands, September 26-28, 2012. Proceedings*, volume 7604 of *Lecture Notes in Computer Science*, pages 209–224, 2012.

30 S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In *19th IEEE International Conference on Distributed Computing Systems*, pages 33–40, 1999.

31 Marta Kwiatkowska, Gethin Norman, and David Parker. Probabilistic verification of herman's self-stabilisation algorithm. *Form. Asp. Comput.*, 24(4–6):661–670, July 2012. `doi:10.1007/s00165-012-0227-6`.

32 Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

33 Stephan Merz. On the verification of a self-stabilizing algorithm. Technical report, University of Munich, 1998.

34 Lawrence C. Paulson. Natural deduction as higher-order resolution. *J. Log. Program.*, 3(3):237–258, 1986. `doi:10.1016/0743-1066(86)90015-4`.

35 S. Qadeer and N. Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In *International Conference on Programming Concepts and Methods (PROCOMET '98) 8–12 June 1998, Shelter Island, New York, USA*, pages 424–443, Boston, MA, 1998. Springer US.

36 Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Formal specification, verification, and implementation of fault-tolerant systems using eventml. *ECEASST*, 72, 2015.

37 Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. Eventml: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Sci. Comput. Program.*, 148:26–48, 2017.

38 Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 619–650, 2018.

39 Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2 edition, 2000. `doi:10.1017/CBO9781139168724`.

40 The Coq Development Team. *The Coq Proof Assistant Documentation*, June 2012. URL: `http://coq.inria.fr/refman/`.

41 Tatsuhiro Tsuchiya, Shin'ichi Nagano, Rohayu Bt Paidi, and Tohru Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE TPDS*, 12(1):81–95, 2001. `doi:10.1109/71.899941`.

A Specification and Closure of Algorithm $\mathcal{BFS}$

Using the definitions given in Section 2.2, we now sketch the proof of the specification part of the self-stabilization of Algorithm $\mathcal{BFS}$. Precisely, we have to prove that the specification of the algorithm is satisfied in any terminal configuration, *i.e.*, the *par*-variables of non-root nodes shape a BFS tree rooted at r that spans the graph `R_Net`. The proof actually follows the one given in [4] for a bounded-memory version of $\mathcal{BFS}$.

Consider a terminal configuration `g`. For a node `p`, `g p` provides the local state of `p` in configuration `g`. Furthermore, let `d (g p)` (resp. `par (g p)`) be the value of `p`.d (resp. `p`.par) in `g`. The proof begins by establishing that in `g`, d-variables are not underestimated:

$$\forall \texttt{ p, d (g p)} \geq \texttt{dist p } r$$

The proof is a simple case analysis; see below.

Proof Overview. Let `p` be a node. If we have `d (g p)` ≥ `dist p` r then we are done. Otherwise, `dist p` r `> d (g p)` and pick a node `pmin` satisfying this inequality with the smallest d-value (we can access `pmin` by filtering the list of all nodes with the *ad hoc* criteria). Then, we can prove that every neighbor `q` of `pmin` satisfies `d (g pmin) < 1 + d (g q))`.

- Indeed, if `dist q` r ≤ `d (g q)`, then we are done since `dist pmin` r ≤ `1 + dist q` r (by definition) and `d (g pmin) < dist pmin` r (by hypothesis).
- Otherwise, `dist q` r `> d (g q)`. But, in this case, `d (g pmin)` ≤ `d (g q)` since `pmin` is a node with minimum `d`-value among the nodes satisfying the inequality, and we are done.

As `pmin` is not the root (indeed `dist pmin` r `> d (g pmin)` ≥ `0`), Action CD is enabled at `pmin`: indeed, `pmin.d` is not equal to `q.d + 1` for some neighbor `q`, since for any of them, we have proved that `d (g pmin) < 1 + d (g q)`. Overall, this proves that `pmin` is enabled, which contradicts the fact that `g` is terminal. ◀

We can now show that for every node p, its d-variable in g (*i.e.*, `d(g p)`) is actually equal to its distance to the root (*i.e.*, `dist g p`).

Proof Overview. The proof is done by induction on the distance from nodes to the root.

- *Base case:* Let `p` be a node such that `dist p` r `= 0`. Then, `p` is the root and as r is disabled, we have $r.d$ `= 0`.
- *Step case:* Let `d` ≥ `0`. Assume the property is satisfied by every node at distance `d` from the root. Let `q` be a node at distance `d + 1` from the root. Obviously, `q` is not the root. Then, by definition of `dist`, `q` has a neighbor, say `p`, at distance `d` from the root. By induction assumption, `d (g p) = d`. As `q` is disabled, we just have to prove that $Dist_q$ `= d + 1`, *i.e.*, `d + 1` is the minimum value in the list `{ x.d + 1, x in q.neighbors }`. Now, `p` is a neighbor of `q` that satisfies `d + 1 = d (g p) + 1`. Moreover, for every other neighbor `p'` of `q`, we have `d + 1` ≤ `d (g p') + 1`. Indeed, by definition, `p'` is at distance `d`, `d + 1`, or `d + 2` from the root and we have seen that `d (g p')` is not underestimated, *i.e.*, `d (g p')` ≥ `dist p'` r ≥ `d`. Hence, we obtain that $Dist_q$ `= d + 1 = d (g q)`, and we are done. ◀

Using Action CP and the fact that d-variables are correctly evaluated in the terminal configuration `g`, we now show that the par-variables define a BFS spanning tree rooted at r in `g`. To that goal, we first define `Par_Rel g n n'` as the relation describing the spanning tree in `g`: `Par_Rel g n n'` holds iff the node `n` is not the root and `par (g n)` is the channel that leads to the node `n'`. By definition of the algorithm and since `g` is terminal, for every non-root node `p`, `Par_Rel g p q` holds for some node `q` such that (p,q) is an edge in `R_Net` and `d (g p) = d (g q) + 1` (remember that these values are also the distances from the nodes to r, hence `dist p` r `= dist q` r `+ 1`). Therefore, we have the following properties:

- `Par_Rel g` is a subgraph of `R_Net`.
- r has no link to some other node using `Par_Rel g`.
- `Par_Rel g` does not contain any cycle, hence it is a DAG.
  Indeed, along any path of `Par_Rel g`, the distances from nodes to the root decreases.
- By definition of `Par_Rel g`, every node has a single parent.

- There is a path from any node `p` to the root in `Par_Rel g` and the length of this path is exactly `dist p` r.
  Notice that this latter property requires to explicitly build the witness path. So, we prove that for every node at distance `d` from the root, there exists a path of length `d` from this node to root in `Par_Rel g`. The proof is done by induction on `d`.
  - The base case (for root node) is trivial.
  - Assume that the property holds for some `d` $\geq$ `0` and consider a node `p` at distance `d + 1` from the root. The parent of `p` using `Par_Rel g`, say `q`, is at distance `d` to the root. Hence, we can apply the induction hypothesis to `q` and then add the edge from `p` to `q` to the path to obtain a path from `p` to r which exists in `Par_Rel g`.

Based on the previous properties, `Par_Rel g` is BFS spanning tree rooted at r. In particular, the distances to the root in `Par_Rel g` are exactly those in `R_Net`. To show this latter fact, we use the last property (there exists a path from every node to root in `Par_Rel g` whose length is the distance to the root) and the fact that the path between any two nodes is unique in a tree (`Path_Rel g` is a tree since it is a DAG with single parent links at each non-root nodes).

Hence the specification of the problem holds in any terminal configuration: the relation `Par_Rel g` (built from the variables computed at each node) is a BFS spanning tree of `R_Net` rooted at r. This concludes the *specification* part of the proof of self-stabilization. Indeed, recall that the set of legitimate configurations is actually the set of terminal configurations.

Finally, since terminal configurations are closed by definition, the *closure* part of the proof is trivially satisfied.

B Detailed Proof of the Round Complexity of Algorithm $\mathcal{BFS}$

We detail here the proof that Algorithm $\mathcal{BFS}$ requires at most $\mathcal{D} + 2$ rounds to reach a terminal configuration, starting from an arbitrary configuration. Under the weakly fair daemon, this property implies convergence. Hence, thanks to results of Appendix A, we can conclude that $\mathcal{BFS}$ is self-stabilizing under the weakly fair daemon and its stabilization time is at most $\mathcal{D} + 2$ rounds.

The proof is split into the following two main parts:

Part A: First, we prove that $\mathcal{BFS}$ requires at most $\mathcal{D} + 1$ rounds to reach a configuration from which the d-variables are correctly assigned forever, *i.e.*, for every node p, $p.d$ is forever equal to the distance from p to the root (**Theorem** `BFS_rounds_CD` in the certified proof).

Part B: Then, we prove that once d-variables are correctly assigned forever, $\mathcal{BFS}$ requires at most one more round to reach a terminal configuration (**Lemma** `last_round_action_CP` in the certified proof).

B.1 Part A

First, we recall the definition of Predicate `check_dist` k `e`, where k is a natural number and `e` is an execution. Following [4], this predicate holds iff for every node p, one of the following two conditions is satisfied in `H e`:

(a) either `CD_a` k `e` p `:=` `dist` p r `<` k $\wedge$ $p.d$ `=` `dist` p r,

(b) or `CD_b` k `e` p `:=` `dist` p r $\geq$ k $\wedge$ $p.d$ $\geq$ k.

Then, we have the following straightforward, yet useful, properties:

1. By definition, `check_dist 0 e` holds, for every execution `e`.

2. `check_dist` $(\mathcal{D}+1)$ `e` holds iff the d-variables in `H e` are correctly assigned.
   Indeed, Case (b) of the definition does not apply in this case.
3. We can easily prove, by checking the rules of the algorithm, that for every execution `e` and every `k`, `check_dist` k `e` is suffix-closed, meaning that once it is satisfied, it holds forever in `e`.
   More formally, `check_dist` k `e` implies that `check_dist` k `c` holds for every suffix `c` of `e`.

We now use the induction scheme given in **Lemma** `schema_round_induction` with the predicate P_k `e := Always (check_dist` k`) e` and the value B `:=` $\mathcal{D}+1$ to prove that any execution `e` requires at most $\mathcal{D}$+1 rounds to reach a configuration where P_B `:= Always (check_dist(`$\mathcal{D}+1$`)) e` holds, *i.e.*, a configuration from which the d-variables are forever correctly assigned. To apply **Lemma** `schema_round_induction`, we have to establish the assumptions of this lemma, namely the *Base Case* and the *Induction Step*. We now consider an arbitrary execution `e` and detail the proof of these two goals.

B.1.1 Base Case

The base case, P_0 `:= Always (check_dist 0)`, is trivial: indeed, `check_dist 0 e` holds, by Property (1), and then we can conclude, by Property (3).

B.1.2 Induction Step

We have to prove that all along the execution `e`, and for every value of $k < \mathcal{D}+1$, if `e` has a suffix `c` satisfying P_k, then `c` requires at most one round to reach P_{k+1}.

First, remark that P_k actually checks local properties at each node since it checks that for every node p, p satisfies all along `e` the local property `CD_a` k `e` $\vee$ `CD_b` k `e`. We use the generic tool to transform the predicate and place the universal quantifier over nodes at the outer border of the formula: we obtain that for any execution `e`, the fact that `e` achieves P_k in at most n rounds is equivalent to the fact that for every node p, `e` reaches in at most n rounds a configuration from which the local property `CD_a` k `e` p $\vee$ `CD_b` k `e` p is satisfied forever. We now consider any node p and split the proof depending on whether or not k is null. In turns, each case is subdivided in two subcases that separately prove `CD_a` $(k+1)$ or `CD_b` $(k+1)$ for p.

Case $k = 0$. We have to prove that at most one more round is required so that `check_dist 1` becomes true. Then again, Property (3) allows to conclude.

(a) *Proof for* `CD_a 1`. Here, by definition of `CD_a`, p can be nothing but the root since the only node at distance less than one from the root is the root itself. Hence, we must prove that at most one round is required so that $r.d$ becomes 0. To that goal, we apply **Lemma** `at_most_rounds_scheme_per_node` and conclude that at most one round is required so that the d-variable of the root becomes 0. Indeed, the assumptions of the lemma are satisfied since:
- if r is disabled, then $r.d = 0$, and
- $r.d$ becomes 0 when r is activated or neutralized (*n.b.*, this latter disjunction is equivalent to "r is activated" since r cannot be neutralized).

(b) *Proof of* `CD_b 1`. We must prove that at most one round is required for every non-root node to have a positive d-variable. Indeed, this case concerns nodes at positive distance from the root, by definition of `CD_b 1`. Again, we can apply **Lemma** `at_most_rounds_scheme_per_node`. Indeed,

- if p is disabled in (`H e`), then $p.d = Dist_p > 0$; and
- when p is activated (resp. neutralized), $p.d$ is set (resp. becomes equal) to $Dist_p > 0$.

Case $1 \leq k < \mathcal{D}+1$. We assume that `check_dist` k `e` holds and use the same mechanisms as previously to prove that at most one round is required to reach a configuration where `CD_a` $(k+1)$ or `CD_b` $(k+1)$ holds for p. The conclusion will be that at most one more round is required to reach `Always (check_dist` $(k+1)$`)`.

(a) *Proof of* `CD_a` $(k+1)$. Here we assume that `dist` p r $< k+1$. If `dist` p r $< k$, then by applying the induction hypothesis (`check_dist` k `e` holds), we get that $p.d$ = `dist` p r forever. Otherwise `dist` p r = k and we apply **Lemma** `at_most_rounds_scheme_per_node` again:
  - If p is disabled in (`H e`), then $p.d = Dist_p = p.par.d + 1$. Then, as `dist` p r = k, p has a neighbor, q, at distance $k-1$ from the root. Using the induction hypothesis, we deduce that $q.d = k-1$. Hence $p.d \leq k$, by definition of $Dist_p$. Consider now the node pointed by $p.par$. As it is a neighbor of p, it is at distance $k-1$, k, or $k+1$ from the root. Cases k and $k+1$ are impossible. Indeed, using the induction hypothesis, we would get that $p.par.d \geq k$ and so $p.d \geq k+1$, a contradiction.
    So, `dist` $p.par$ r = $k-1$. In this case, $p.par.d = k-1$, by induction hypothesis, and so $p.d = k$.
  - When p is activated (resp. neutralized), $p.d$ is set to (resp. becomes) $Dist_p$. So, we have to show that $Dist_p$ = k, *i.e.*, $k-1$ is the smallest value in the d-variables of p's neighbors. Now, as the distance from p to r is k, every neighbor q of p is at distance $k-1$, k, or $k+1$ from the root. By applying the induction hypothesis, we obtain that the neighbors at distance greater than $k-1$ from the root have their d-variables greater than $k-1$; moreover, those at distance $k-1$ from the root have their d-variable equal to $k-1$. As p has at least one neighbor at distance $k-1$ from the root, we obtain that $p.d$ = $Dist_p$ = k = `dist` p r.

(b) *Proof of* `CD_b` $(k+1)$. Here we assume that `dist` p r $\geq k+1$. In this case, every neighbor q of p is at distance at least k from r. Hence, by induction hypothesis, we obtain that $d.q \geq k$. Using this property and the definition of $Dist_p$, we can easily show that the two conditions of **Lemma** `at_most_rounds_scheme_per_node` are fulfilled to establish that at most one more round is required to reach a configuration where the property $p.d \geq k+1$ is satisfied.

B.2 Part B

After `check_dist` $(\mathcal{D}+1)$ `e` is achieved (*i.e.*, once the d-variables are correctly assigned forever), at most one more round is required to reach a configuration where the par-variables are correctly set and so to achieve the silence. The proof uses the same mechanism as before and mainly relies on the following two simple facts:

- As `check_dist` $(\mathcal{D}+1)$ `e` holds forever, the d-variables no more change.
- Furthermore, when `check_dist` $(\mathcal{D}+1)$ `e` holds, a node can be activated only once (using Action CP): the action, and so the node too, is then disabled forever.

B.3 Final Result

Afterwards, Parts A and B are merged using Lemma `schema_round_step` to conclude that every execution `e` requires at most $\mathcal{D}+2$ rounds to reach a terminal configuration. This conclude the proof of the round complexity for Algorithm $\mathcal{BFS}$.