Squeezing Streams and Composition of Self-stabilizing Algorithms

Karine Altisen, Pierre Corbineau, Stéphane Devismes

Squeezing Streams and Composition of Self-Stabilizing Algorithms*

Karine Altisen, Pierre Corbineau, and Stéphane Devismes

Univ. Grenoble Alpes, CNRS, Grenoble INP**, VERIMAG, 38000 Grenoble, France

Abstract. Composition is a fundamental tool when dealing with complex systems. We study the hierarchical collateral composition which is used to combine self-stabilizing distributed algorithms. The PADEC library is a framework developed with the Coq proof assistant and dedicated to the certification of self-stabilizing algorithms. We enrich PADEC with the composition operator and a sufficient condition to show its correctness. The formal proof of the condition leads us to develop new tools and methods on potentially infinite streams, these latter ones being used to model the algorithms' executions. The cornerstone has been the definition of the function `Squeeze` which removes duplicates from streams.

Keywords: Coq Proof Assistant · Streams · Coinduction · Composition · Distributed Algorithm · Self-Stabilization.

1 Introduction

In computer science, *separation of concerns* is a standard design principle which consists of decomposing a complex problem into several simpler ones. These subproblems are then solved independently, and finally, glued together to obtain a global solution to the initial problem. With this in mind, *composition* is a natural tool that simplifies both the design and proof of complex algorithms. For example, the sequential composition of two algorithms "$\mathcal{A}_1$; $\mathcal{A}_2$" enforces $\mathcal{A}_1$ and $\mathcal{A}_2$ to be executed in sequence, *i.e.*, $\mathcal{A}_2$ is initiated only after $\mathcal{A}_1$'s completion. Composition methods are widely used in distributed systems [10,25,4].

Self-stabilization [21] is a versatile fault-tolerant paradigm of distributed computing. Indeed, a self-stabilizing distributed algorithm resumes a correct behavior within finite time, regardless the initial state of the system, and therefore also after a finite number of transient faults hit the system and place it in some arbitrary global state. The ability to implement sequential composition in a distributed system mainly relies on the ability to locally detect the termination. Now, termination detection is inherently impossible for self-stabilizing algorithms [34]. Indeed, since the system may suffer from faults such as memory corruption, the nodes cannot trust their local memory. To circumvent such an

* This study has been partially supported by the ANR projects DESCARTES (ANR-16-CE40-0023) and ESTATE (ANR-16-CE25-0009).

** Institute of Engineering Univ. Grenoble Alpes

issue, several other composition operators devoted to self-stabilizing algorithms have been proposed, *e.g.*, the fair [23] and cross-over [6] compositions. We are more particularly interested in the *hierarchical collateral composition* [20], a simple and widely used variant of the collateral composition [27]. This composition actually emulates the sequential composition "$\mathcal{A}_1\,;\ \mathcal{A}_2$" by providing the same output despite $\mathcal{A}_1$ and $\mathcal{A}_2$ being executed (more or less) concurrently.

The PADEC framework [3,2] consists in a library for *certifying* self-stabilizing algorithms. The certification of an algorithm means proving its correctness *formally* using a proof assistant, here Coq [35,9], *i.e.*, a tool which allows to develop formal proofs interactively and *mechanically check* them. The framework includes tools to model self-stabilizing algorithms, certified general statements that can be used to build certified correctness proofs of such algorithms, and case studies that validate them. In PADEC, the semantics of self-stabilizing algorithms' executions is defined as potentially infinite streams and properties, such as algorithm specifications, are defined using temporal logic on those streams. Hence, the definitions and proofs presented in PADEC as well as this paper, make an intensive use of streams and thus of coinductive definitions and proofs.

Overview of the contributions. The first contribution of this paper consists of new general tools for streams, in particular a squeezing operator. This latter is actually a productive filter on streams that uses both inductive and coinductive mechanisms. Our second contribution is a case study: we apply the squeezing operator to certify the hierarchical collateral composition of self-stabilizing algorithms. To our knowledge, our proposal is the first work on the certification of a composition operator for self-stabilization.

Detailed contributions. We develop many tools for streams. Our streams are potentially infinite sequences of at least one element and require to be defined over a partial setoid, *i.e.*, over a type endowed with a partial equivalence relation that models equality; thus justifying this new development. Apart from usual tools required by developments on streams, such as temporal logic operators, we also provide tools specific for PADEC. In particular, the *squeezing toolbox* provides a filter to remove any duplicated value from a given stream that may contain an infinite suffix of duplicates. We study the conditions under which such a squeezed stream can be computed and provide a function that actually builds it. This filter can be viewed as an extension of a work by Bertot [8]. Indeed, although Bertot's filter relies on a general predicate (ours simply uses the equality between two consecutive elements), the squeezing operator is designed for more complex streams (that can be finite or infinite) and allows to remove an infinite suffix. In his paper, Bertot clearly explains the difficulty to formally define such a filter since this latter mixes both coinduction and induction mechanisms. The definition of squeezing is even more difficult since it requires to decide at each step whether the filtering of new elements should continue or be given up because a constant, potentially infinite, suffix has been reached.

As an application, we use these tools to enrich the PADEC library with a formalization of the hierarchical collateral composition operator $\oplus$ and a sufficient

condition to show its correctness. By correctness, we mean that if $\mathcal{A}_1$ and $\mathcal{A}_2$ are self-stabilizing *w.r.t.* their specification, then $\mathcal{A}_1 \oplus \mathcal{A}_2$ is also self-stabilizing *w.r.t.* both specifications. Executions of self-stabilizing algorithms and their compositions are modeled as streams, and the squeezing toolbox has been the cornerstone to solve the major locks in the correctness proof of the composition operator.

Related work. Previous work dealing with PADEC [3] only considered *terminating* algorithms that did not require any scheduling assumption, consequently their proofs were only induction-based. Here, $\mathcal{A}_2$ may be a *non-terminating* algorithm (*e.g.*, a token circulation). Moreover, the sufficient condition to show the correctness of the composition assumes a weakly fair scheduling, which requires a coinductive definition. Coinductive objects and proofs allow to reason on potentially infinite objects. They are supported by major proof assistants such as Coq [26], Isabelle [32], and Agda [1]. Coinductive constructions are commonly used to represent potentially infinite behaviors of programs and systems (see, for example, [31] for sequential programs and [17] for distributed systems) but also for modeling lazy programs such as the prime number sieve [8].

Proofs in distributed algorithms are commonly written by hand, based on informal reasoning. This potentially leads to errors when arguments are not perfectly clear, as explained by Lamport in his position paper [30]. As a matter of facts, several studies [7,22] reveal, using formal methods, some bugs in existing literature. Hence, certification of distributed algorithms is a powerful tool to prevent bugs in their proofs, and so, to increase confidence in their correctness. Certification of non fault-tolerant distributed algorithms is addressed in [13,14,17]; and certification in the context of fault-tolerant, yet non self-stabilizing, distributed computing is addressed in [28,5]. Up to now, only few *simple* self-stabilizing algorithms have been certified, *e.g.*, [29] (in PVS) and [16,3] (in Coq). By simple, we mean non-composed algorithms working on particular topologies (*i.e.*, rings, lines, or trees) and/or assuming restrictions on possible interleaving (*e.g.*, in [29], only sequential executions are considered). Now, progress in self-stabilization has led to consider more and more complex distributed systems running in increasingly more adversarial environments. As an illustrative example, the three first algorithms proposed by Dijkstra in 1974 [21] were designed for oriented ring topologies and assuming sequential executions only, while nowadays most self-stabilizing algorithms are designed for fully asynchronous arbitrary connected networks, *e.g.*, [19,12], and even for networks, such as peer-to-peer systems, where the topology (frequently) varies over the time, *e.g.*, [11]. Consequently, the design of self-stabilizing algorithms becomes more and more intricate, and accordingly, their proofs of correctness and complexity. To handle such difficulties, designers must adopt a modular approach, *e.g.*, using composition operators. Consequently, a preliminary necessary step to certify present-day self-stabilizing algorithms is the certification of a composition operator.

Roadmap. Section 2 introduces streams and self-stabilization as defined in PADEC. Section 3 presents the composition. Section 4 details the squeezing toolbox and shows its application into the proof of correctness of the composition.

The composition and the stream toolboxes contain about 1500 lines of Coq specifications and 4800 lines of Coq proofs.[1] This represents about 25% of the whole PADEC library. We advocate the reader to visit the following webpage for a deeper understanding of our work.

`http://www-verimag.imag.fr/~altisen/PADEC/`

All documentation and source codes are available at this address.

2 Streams and Self-Stabilization in the PADEC Library

PADEC is a Coq library designed to model and prove results on self-stabilizing algorithms. The framework makes an intensive use of (partial-)setoids, *i.e.*, types for which the equality is represented by a (partial-)equivalence relation. This choice is justified in [3] and has some consequences on the design of the framework. Nevertheless, we omit here the technical issues due to the use of such (partial-)setoids, since it is out of the scope of this paper.

We now present self-stabilizing algorithms as they are defined in distributed computing and the PADEC library. Beforehand, we introduce streams as they are used to model executions of self-stabilizing algorithms in PADEC.

2.1 Streams

We implement a stream as a potentially infinite sequence of at least one element. Each element belongs to some given type `A`. A stream is then defined as a value of the following type.

```
CoInductive Stream: Type := | O: A → Stream
                            | C: A → Stream → Stream.
```

Remark that such a stream cannot be empty since each constructor (`O`, `C`) enforces the existence of a first element. Moreover, it may be *finite or infinite* since the keyword **CoInductive** generates the greatest fixed point capturing potentially infinite constructions.[2] For instance, the finite stream of naturals `1 2 3 4` is given by `s4 = C 1 (C 2 (C 3 (O 4)))` and the infinite stream of naturals, made of an infinite number of 1, is defined by **CoFixpoint** `ones: Stream (A := nat) := C 1 ones`. Therefore, the above definition allows to construct both finite and infinite streams thanks to the two constructors. In contrast, streams from the standard Coq API [35] and those proposed by Bertot [8] are made of only one constructor, which enforces the stream to be necessarily infinite.

We define the function `(H: Stream → A)` which returns the first element of the stream, *e.g.*, `(H s4)` returns 1. For any function `(f: A → B)` and any type

[1] As evaluated by the *ad hoc* tool `coqwc`.

[2] In contrast, the keyword **Inductive** generates the smallest fixed point and only captures finite constructions.

`B`, we note `(f • H: Stream → B)` the function defining the composition of `H` and `f` as follows: `((f • H) s)` returns `(f (H s))`, for any stream `s`.

We now briefly introduce tools on streams that will be used in the sequel. The following predicates are usual temporal logic operators [33,15]. They are defined *w.r.t.* a given predicate `P` over streams. The first one checks that there is a suffix of the stream in which `P` is satisfied. The second one checks that `P` is satisfied in every suffix of the stream.

```
Inductive Eventually (P: Stream → Prop): Stream → Prop :=
| ev_now: ∀ s, P s → Eventually P s
| ev_later: ∀ a s, Eventually P s → Eventually P (C a s).

CoInductive Always (P: Stream → Prop): Stream → Prop :=
| al_one: ∀ a, P (O a) → Always P (O a)
| al_cons: ∀ a s, P (C a s) → Always P s → Always P (C a s).
```

Note the difference between the two definitions: `Eventually` is defined using the keyword **Inductive** since a proof of `(Eventually P s)`, for some stream `s` and predicate `P`, should only contain a finite number of `ev_later`. In contrast, `Always` uses **CoInductive**: a proof of `(Always P s)` would potentially contain an infinite number of `al_cons` and so, should be lazily constructed. We defined many other properties and technical tools that ease the use of those predicates (see [2] for details), *e.g.*, we use `Eventually` to check that a stream is finite:

```
finite: Stream → Prop := Eventually P_finite.
```

where `(P_finite s)` holds if and only if the stream `s` is made of a single element `a`, *i.e.*, is equal to `(O a)`.

2.2 Self-Stabilization: Model and Semantics

Most of self-stabilizing algorithms are designed in the *atomic-state* model, a computational model introduced by Dijkstra [21], which abstracts away the communications between nodes of the network. The PADEC framework has been developed using this model (see [3]). However, we do not detail the model here, since this is not the heart of the contribution. Instead, we summarize features that are mandatory to present and understand our contributions.

A distributed algorithm is executed over a *network*, made of a *finite* number of *nodes* (we introduce the Coq type `Node` to represent nodes). Each node p is endowed with a *local state* (of type `State`) defined by the value of its local variables. Node p updates its local state by executing its local algorithm in *atomic moves*, where it first reads its own local state and that of its neighbors, and then only writes its own variables. Notice that some variables owned by p, usually system inputs, should never be written by its local algorithm. Such variables are declared *read-only*. A node is said to be *enabled* if its next move will actually modify its local state. Otherwise, the node is said to be *disabled*.

We call *environment* the global state of the network. In PADEC, environments are functions from `Node` to `State`: `Env := Node → State`, namely for an

environment `(g: Env)` and a node `(n: Node)`, `(g n)` is the local state of `n` in `g`. If no node is enabled in `g`, then `g` said to be *terminal*. This property is defined by the predicate `terminal: Env` $\rightarrow$ **Prop**. Each node can locally evaluate whether or not it is enabled. So, since the number of nodes is finite, the `terminal` property is *decidable*, *i.e.*, the evaluation of `(terminal g)` is computable:[3]

Lemma `terminal_dec: ∀ g, { terminal g }+{ ¬terminal g }.`

Let `g` be the current environment. If `g` is not terminal, then a *step* of the distributed algorithm is performed as follows: every node `n` that is enabled in `(g n)` is candidate to be executed; some candidates (at least one) are nondeterministically *activated*, meaning that they atomically update their local state using their local algorithm, leading the system to a new environment `g'`. This nondeterminism actually materializes the asynchronism of the system.

Notice that two environments linked by a step are necessarily different. This point is fundamental in asynchronous deterministic algorithms: the system progress can only be observed when the environment changes. In PADEC, we use the relation `Step: Env` $\rightarrow$ `Env` $\rightarrow$ **Prop** to encode all possible steps.

A *maximal run* in the network is defined as a stream of environments, using type `Exec:` **Type** `:= Stream (A := Env)`, where every pair of consecutive environments in the stream is a step, and if the stream is finite then its last environment is necessarily terminal. This notion is captured by the predicate

```
is_max_run (e: Exec): Prop := Always P_run e.
```

where `(P_run e)` checks that the stream `e` matches one of the two following patterns. Either `e` is `(O g)` and the environment `g` is terminal, *i.e.*, `(terminal g)` holds; or `e` is `(C g e')` (with `g` an environment and `e'` a stream) and there is a step from `g` to `(H e')`, *i.e.*, `(Step (H e) g')` holds.

We model the nondeterminism of the system using an artifact called the *daemon*. In this paper, we focus on the so-called *weakly fair daemon* [24]: a maximal run is executed under the weakly fair daemon if and only if every node that is continuously enabled is eventually activated by the daemon. To encode the weakly fair daemon, we define the following predicate:

```
weakly_fair (e: Exec): Prop := ∀ (n: Node),
  Always (fun e' => EN n e' → Eventually (AN n) e') e.
```

Namely, all along a run `e`, whenever some node `n` is enabled (predicate `(EN n)`), it is eventually either activated or neutralized (predicate `(AN n)`), *i.e.*, either it is eventually chosen by the daemon to execute in a step, or it eventually becomes disabled, due to the move of some of its neighbors. Note that this definition involves both inductive and coinductive predicates.

A self-stabilizing algorithm is designed to fulfill a given specification under some assumptions, often related to the system. In the literature, those assumptions are directly encoded in the configurations using constants whose values achieve some conditions. For example, an identified network is modeled using a

[3] The notation `{ A }+{ B }` (so-called `sumbool` in Coq) means there exists an algorithm able to choose between Conditions `A` and `B`.

constant variable, called identifier, for each process and assuming that every two distinct processes have different identifiers. Following the literature, we express those assumptions (predicate `Assume: Env` $\rightarrow$ **Prop**) on the read-only variables of the nodes. Such assumptions need only to be checked on the initial environment of a run. Indeed, they are then inherently satisfied all along the run since they only rely on read-only variables.

To sum up, we define an *execution* of the algorithm to be a stream of environments which is a maximal run satisfying the daemon constraints and where the read-only assumptions are satisfied in its first environment. Hence, executions are encoded by the following predicate.

```
is_exec (e: Exec): Prop :=
  (Assume • H) e ∧ is_max_run e ∧ weakly_fair e.
```

It is important to note that a self-stabilizing algorithm can be initiated from any environment where the read-only assumptions are satisfied. This, in particular, means that *every suffix of an execution is also an execution.*

The specification of an algorithm is given as a predicate $\mathcal{S}$`: Exec` $\rightarrow$ **Prop**. Then, an algorithm $\mathcal{A}$ is *self-stabilizing* (predicate `self_stabilization` $\mathcal{A}$`:` **Prop**) *w.r.t.* a specification $\mathcal{S}$ under the weakly fair daemon if there exists a set of environments called *legitimate* and detected using the predicate `LEG: Env` $\rightarrow$ **Prop**, such that for every execution `e` (implicitly `e: Exec` and `is_exec e`),

- if its initial environment is legitimate, then each of its environments is legitimate, *i.e.*, `(LEG • H) e` $\rightarrow$ `Always (LEG • H) e` *(Closure)*;
- it converges to a legitimate environment, *i.e.*, `Eventually (LEG • H) e` *(Convergence)*; and
- if it is initiated in a legitimate environment, then it satisfies the specification, *i.e.*, `(LEG • H) e` $\rightarrow$ $\mathcal{S}$ `e` *(Specification).*

In this paper, we also consider the class of *silent* self-stabilizing algorithms. In the atomic-state model, an algorithm $\mathcal{A}$ is *silent* if all executions are finite: `silent` $\mathcal{A}$`:` **Prop** `:=` $\forall$ `(e: Exec), is_exec e` $\rightarrow$ `finite e.` A silent algorithm is designed to converge to terminal environments satisfying some properties. So, the specification of such an algorithm is rather formulated as a predicate over environments $\mathcal{S}^g$`: Env` $\rightarrow$ **Prop**, henceforth called *environment specification.*

3 Composition

The hierarchical collateral composition has been introduced in [20] together with a simple sufficient condition to show its correctness. We now describe the operator, its modeling, and the certification of the sufficient condition in PADEC. Beyond the higher confidence in the accuracy of the result, certification, by enforcing proofs to be more rigorous, leads to a deeper understanding of the result.

The goal of the hierarchical collateral composition operator is to mimic the sequential composition "$\mathcal{A}_1$`;` $\mathcal{A}_2$". $\mathcal{A}_1$ and $\mathcal{A}_2$ run concurrently modulo some priorities (see details below) and collaborate together using common variables. The goal of $\mathcal{A}_1$ is to self-stabilizingly output correct inputs to $\mathcal{A}_2$. $\mathcal{A}_2$ is self-stabilizing provided that its inputs, in particular those computed by $\mathcal{A}_1$, are

correct. Hence, the actual convergence of $\mathcal{A}_2$ is ensured only after $\mathcal{A}_1$ has stabilized. For example, the clustering algorithm for general networks given in [18] is a hierarchical collateral composition $\mathcal{A}_1 \oplus \mathcal{A}_2$, where $\mathcal{A}_1$ is a spanning tree construction and $\mathcal{A}_2$ a clustering algorithm dedicated to tree topologies.

$\mathcal{A}_1$ should converge so that its output variables permanently fulfill the input assumptions of $\mathcal{A}_2$ to ensure that $\mathcal{A}_2$ stabilizes in nominal conditions. To that goal, we assume that $\mathcal{A}_1$ is silent, *e.g.*, in [18], once the spanning tree construction has stabilized, all its variables, in particular those defining the tree, are constant.

For each node, the *local variables* of the composite algorithm $\mathcal{A}_1 \oplus \mathcal{A}_2$ are made of variables specific to $\mathcal{A}_1$ and $\mathcal{A}_2$ respectively, but also of variables common to $\mathcal{A}_1$ and $\mathcal{A}_2$. Those variables store, in particular, the output of $\mathcal{A}_1$ used as input by $\mathcal{A}_2$. They should be read-only in $\mathcal{A}_2$ since $\mathcal{A}_2$ should not prevent $\mathcal{A}_1$ from stabilizing by overwriting these variables.

In the previous collateral composition [27] of Gouda and Herman, the choice for an activated node to execute either $\mathcal{A}_1$ or $\mathcal{A}_2$, when both are enabled, was nondeterministic. In contrast, in the hierarchical collateral composition, the *composite algorithm* gives priority to $\mathcal{A}_1$ over $\mathcal{A}_2$ *locally at each node*. Let p be a node enabled *w.r.t.* $\mathcal{A}_1 \oplus \mathcal{A}_2$ in some environment and assume that p is activated by the daemon in the next step.

- If $\mathcal{A}_1$ is enabled at p (*n.b.*, $\mathcal{A}_2$ may be enabled at p too), then p makes a move of $\mathcal{A}_1$ only.
- Otherwise, p is disabled *w.r.t.* $\mathcal{A}_1$, but enabled *w.r.t.* $\mathcal{A}_2$, and so makes a move of $\mathcal{A}_2$ (only).

Hence, when p moves in $\mathcal{A}_1 \oplus \mathcal{A}_2$, it either executes $\mathcal{A}_1$ or $\mathcal{A}_2$, but not both. We should underline that this priority mechanism is only local: globally, a step of $\mathcal{A}_1 \oplus \mathcal{A}_2$ may contain moves of $\mathcal{A}_1$ only, moves of $\mathcal{A}_2$ only, but also a mix of them, yet executed at different nodes.

3.1 The Composite Algorithm in Coq

We model the composite algorithm $\mathcal{A}_1 \oplus \mathcal{A}_2$ in Coq as follows. We define the local states `S3` of $\mathcal{A}_1 \oplus \mathcal{A}_2$ assuming that the local states of $\mathcal{A}_1$, noted `S1`, can be handled using the following getter and setter:

- `read1: S3 → S1` is a projection from `S3` to `S1`,
- `write1: S1 → S3 → S3` modifies the `S1`-part of a composite state.

Functions `read2` and `write2` are defined similarly for the local states `S2` of $\mathcal{A}_2$.[4] Those functions follow the properties given by the commutative diagram of Figure 1. For example, to update the `S1`-part of the composite local state `(x3: S3)` with `(x1: S1)`, we use `write1(x1,x3)`: this produces a new `S3` local state with `S1`-part `x1`, namely, `read1(write1(x1,x3))` returns `x1`. Additionally, we encode the fact that any writing in the `S2`-part (by $\mathcal{A}_2$), that respects the read-only condition, actually does not modify the `S1`-part of an `S3` state. Indeed, the common part between `S1` and `S2` is necessarily read-only for `S2` (see x in Figure 2).

[4] `S1`, `S2`, `S3` stand for type `State` dedicated to Algorithms $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{A}_1 \oplus \mathcal{A}_2$, respectively.

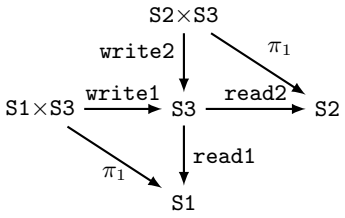

Fig. 1. Commutative diagram for `read` and `write`. π_1 gives access to the first element of the pair.

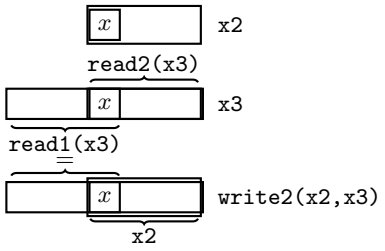

Fig. 2. `write2` cannot modify `S1`-part. x is the common part between `S1` and `S2`, read-only for `S2`.

We generalize the projections `read1` and `read2` to environments and streams. The projection `envread1` to $\mathcal{A}_1$ of an environment `g` of $\mathcal{A}_1 \oplus \mathcal{A}_2$ is an environment for $\mathcal{A}_1$ defined as `read1 • g`. Namely, for a node `n`, the projection of `(g n)` on $\mathcal{A}_1$, is obtained by `(read1(g n))`. The projection on $\mathcal{A}_1$ of a stream `s` of $\mathcal{A}_1 \oplus \mathcal{A}_2$ is called `execread1` and is obtained using a cofixed point that applies `envread1` to every element of the stream (*i.e.*, a map on a stream). In particular, `((H • execread1) s)` and `((envread1 • H) s)` represent one and the same environment. The projections `envread2` and `execread2` on $\mathcal{A}_2$ are defined similarly.

3.2 Correctness of the Composition

The composition operator is proven correct under the following hypotheses:
$\mathcal{H}_1$: The daemon is weakly fair.
$\mathcal{H}_2$: $\mathcal{A}_1$ is silent and self-stabilizing *w.r.t.* the environment specification $\mathcal{S}_1^g$: given the read-only assumption `Assume1`, each of its executions is finite and terminates in an environment satisfying $\mathcal{S}_1^g$.
$\mathcal{H}_3$: $\mathcal{A}_2$ is self-stabilizing *w.r.t.* specification $\mathcal{S}_2$: given the read-only assumption `Assume2`, each of its executions eventually reaches a legitimate environment (predicate `LEG2`) from which $\mathcal{S}_2$ is satisfied.
$\mathcal{H}_4$: The read-only assumption of $\mathcal{A}_1 \oplus \mathcal{A}_2$ is `Assume1` on $\mathcal{A}_1$-projections.
$\mathcal{H}_5$: $\mathcal{S}_1^g$ implies `Assume2`, *i.e.*, $\forall$ `g,` $\mathcal{S}_1^g$ `(envread1 g)` $\rightarrow$ `Assume2 (envread2 g).`
Under those hypotheses, we have proven the theorem below for the specification $\mathcal{S}$ `:= fun e => Always (`$\mathcal{S}_1^g$ `• H • execread1) e` $\wedge$ `(`$\mathcal{S}_2$ `• execread2) e.`

Theorem `Composition_Correctness: self_stabilization` $\mathcal{A}_1 \oplus \mathcal{A}_2$`.`

The above theorem states that $\mathcal{A}_1 \oplus \mathcal{A}_2$ eventually reaches an environment from which $\mathcal{S}_2$ holds and $\mathcal{S}_1^g$ is satisfied in all environments.

We now outline the proof of the theorem. We first have to exhibit a predicate that defines the legitimate environments `LEG3` of $\mathcal{A}_1 \oplus \mathcal{A}_2$. This predicate holds in each environment that is terminal for $\mathcal{A}_1$ and legitimate for $\mathcal{A}_2$:

```
LEG3 := fun g3 => terminal (envread1 g3) ∧ LEG2 (envread2 g3).
```

Then, we prove the following intermediate result:

Lemma `t1_e2: ∀ e, is_exec e → (terminal • H) (execread1 e) → Always (terminal • H) (execread1 e) ∧ is_exec (execread2 e).`

Namely, any execution `e` of $\mathcal{A}_1 \oplus \mathcal{A}_2$ that starts in an $\mathcal{A}_1$-terminal environment remains in $\mathcal{A}_1$-terminal environments and is actually an execution for $\mathcal{A}_2$. First, from an environment which is terminal for $\mathcal{A}_1$, there is no way to update variables of $\mathcal{A}_1$. So, `e` remains in environments that are $\mathcal{A}_1$-terminal. This claim also implies that each step of `e` is actually a step of $\mathcal{A}_2$ and, consequently, `(execread2 e)` is a maximal run of $\mathcal{A}_2$ satisfying the weakly fair condition. Finally, `((H • execread2) e)` satisfies `Assume2`. Indeed, $\mathcal{A}_1$ being silent and self-stabilizing, this implies that if $\mathcal{A}_1$ starts in a terminal environment, then, this environment satisfies $\mathcal{S}_1^g$. Thus, we can use hypothesis $\mathcal{H}_5$ on the first environment of `e`. Hence, we can conclude that `(execread2 e)` is an execution of $\mathcal{A}_2$.

In the rest of the explanation, we consider an arbitrary execution `e` of $\mathcal{A}_1 \oplus \mathcal{A}_2$.

Closure. To show the closure, we have to prove that if `e` starts in a legitimate environment of $\mathcal{A}_1 \oplus \mathcal{A}_2$ (*i.e.*, an environment satisfying `LEG3`), it always remains in such environments. This is straightforward using Lemma `t1_e2`. Indeed, first, `e` remains in environments that are $\mathcal{A}_1$-terminal. Second, as `(execread2 e)` is an execution for $\mathcal{A}_2$, we can use the closure property of $\mathcal{A}_2$ on `(execread2 e)` (since $\mathcal{A}_2$ is self-stabilizing) and prove that legitimate environments for $\mathcal{A}_2$ are maintained forever in `(execread2 e)`.

Specification. We have to prove that if `e` is initiated in `LEG3`, then `(`$\mathcal{S}$ `e)` holds. We use Lemma `t1_e2` again. First, every environment of `e` is $\mathcal{A}_1$-terminal, and so satisfies $\mathcal{S}_1^g$. Second, `(execread2 e)` is an execution of $\mathcal{A}_2$ on which we can apply the specification property of $\mathcal{A}_2$ (since $\mathcal{A}_2$ is self-stabilizing), hence satisfies $\mathcal{S}_2$.

Convergence. We should prove that `e` eventually reaches an environment that is legitimate for $\mathcal{A}_1 \oplus \mathcal{A}_2$. This goal is split into three subgoals:

`(1) Eventually (terminal • H) (execread1 e)`

i.e., `e` eventually reaches an environment which is terminal for $\mathcal{A}_1$. This part of the proof is postponed to Section 4. Claim `(1)` ensures that `e` contains a suffix σ that starts in a terminal environment for $\mathcal{A}_1$: we have `((terminal • H) (execread1 `σ`))`. The second subgoal is then:

`(2) Always (terminal • H) (execread1 `σ`)`

i.e., σ remains $\mathcal{A}_1$-terminal. As any suffix of an execution is also an execution, so is σ. Hence, Claim `(2)` is immediate from Lemma `t1_e2`. The third subgoal is:

`(3) Eventually (LEG2 • H) (execread2 `σ`)`

After `e` has reached an environment that is terminal for $\mathcal{A}_1$, it eventually reaches an environment that is legitimate for $\mathcal{A}_2$. Indeed, its suffix σ eventually reaches an environment that is legitimate for $\mathcal{A}_2$: to prove this, we use the convergence of $\mathcal{A}_2$ (as $\mathcal{A}_2$ is self-stabilizing) since, by Lemma `t1_e2`, `(execread2 `σ`)` is an execution of $\mathcal{A}_2$. Now, as σ eventually reaches `LEG2`, so does `e`.

4 Squeezing Streams and Convergence of Composition

The main part of the proof consists in proving that any execution of the composite algorithm eventually reaches an environment which is terminal for $\mathcal{A}_1$ (Claim `(1)`). This requires to use the assumption that $\mathcal{A}_1$ is silent, *i.e.*, Hypothesis $\mathcal{H}_2$. To that goal, we consider an execution `e` of $\mathcal{A}_1 \oplus \mathcal{A}_2$ and we focus on its projection on $\mathcal{A}_1$, `(execread1 e)`. Now, this latter stream is usually not an execution of $\mathcal{A}_1$ and $\mathcal{H}_2$ only applies to executions of $\mathcal{A}_1$. Actually, each step of `e` matches one of the two following cases: either at least one node executes $\mathcal{A}_1$ in the step, or all activated nodes only execute $\mathcal{A}_2$. In the former case, the projection of the step on $\mathcal{A}_1$ is a step of $\mathcal{A}_1$. In the latter case, the projection gives two identical environments of $\mathcal{A}_1$. Hence, `(execread1 e)` is made of steps of $\mathcal{A}_1$, separated by duplicates. So, to apply $\mathcal{H}_2$, it is mandatory to construct an execution of $\mathcal{A}_1$ by computing the *squeezing* of `(execread1 e)`, *i.e.*, the stream obtained by removing all duplicates from `(execread1 e)`.

In Subsection 4.1, we describe how to compute the squeezing of a general stream. Again, it is a *filter* in the sense of Bertot [8] since it removes elements from the stream. Yet, its filtering predicate is particular, as it forbids any two consecutive elements to be equal. But, in contrast with Bertot [8], the squeezing applies to streams that can be finite or infinite and allows to remove an infinite suffix of duplicates. Therefore, we have an additional issue: the squeezing needs to decide at each step whether to continue or give up because a constant, potentially infinite, suffix has been reached.

The object resulting from squeezing – so-called *squeezed stream* – is complex since it is defined as an explicit construction. Consequently, dealing with it directly in proofs requires heavy Coq developments. To avoid such implementation details, we rather work on an abstraction stream relation, called *simulation*, which encompasses the useful properties of the squeezed stream.

4.1 Squeezing

We now explain how to build the squeezing of an arbitrary stream whose elements are of type `A`. Let `s` be such a stream. The squeezed version of `s` contains exactly the same elements as `s`, in the same order, yet without any duplicate.

For example, if `s = 1 2 2 3 3 3 3 3 4 5 6 6 7 8 8 8 ...` (`s` ends with an *infinite* suffix of `8`), then the squeezing of `s` is the *finite* sequence `s' = 1 2 3 4 5 6 7 8`. Every element in `s` is still present in `s'`, following the same increasing order, yet every duplicate has been removed from `s`, including the infinite sequence of `8`. Note that a squeezing may not be finite, *e.g.*, if `s` is the infinite repetition of the pattern `1 2 3`, then the squeezing of `s` is `s` itself.

We want to *build* the squeezed stream, *i.e.,* to define a function `Squeeze` which *computes* the squeezed version of an input stream `s`. This computation will be carried out by a coinductive function. To compute `(Squeeze s)`, it is necessary to *test* whether all elements in the stream `s` are identical to `(H s)`. If so, `(Squeeze s)` will be `(O (H s))`; otherwise it will be `(C (H s) (Squeeze s*))`, where $s^\star$ is the maximal suffix of `s` starting with an element distinct from `(H s)`. Now, a stream

may be infinite, and so the aforementioned test is undecidable in general. Thus, we need extra information in order to make the decision and, since we will base the result of the squeezing algorithm on this decision, it has to be *constructive*. In Coq, constructive objects are in sort **Type** and carry computational content.[5] We assume `(Skippable (H s) s)` where:

```
Skippable (a: A) (s: Stream): Type :=
  { is_constant a s }+{ reach_diff a s }.
```

Any value of `(Skippable (H s) s)` carries either a proof of `(is_constant (H s) s)` or a proof of `(reach_diff (H s) s)`.

For an element `a` and a stream `s`, `(is_constant a s)` means `(Always (fun s => H s ≅ a) s)`,[6] *i.e.*, the stream `s` contains nothing but the element `a`, albeit finite or not. Then, `(reach_diff a s)` means that `s` begins with a finite number of `a` followed by some element different from `a`. To be able to compute `Squeeze`, `(reach_diff a s)` should also provide a way to compute the suffix $s^\star$ of `s` where all the instances of `a` at the beginning of `s` have been removed. Actually, we implement `(reach_diff a s)` as `(Acc (Rskip a) s)`. For an element `a` and two streams `s1`, `s2`, `(Rskip a s1 s2)` holds when either `s1` and `s2` are both reduced to the single element `a` or `s2` is equal to `(C a s1)`, *i.e.*, `C a s1 ≅ s2 ∨ s1 ≅ O a ∧ O a ≅ s2`. The inductive proposition `Acc` is taken from the `Coq.Init.Wf` standard library which provides tools on well-founded inductions. Predicate `(Acc (Rskip a) s)` means that any descending chain from `s`, using relation `(Rskip a)`, is finite. Using a well-founded induction on a value of `(reach_diff a s)`, we are able to define a recursive function `Skip` with dependent arguments `(a: A)`, `(s: Stream)`, and `(rd: reach_diff a s)` that computes the maximal suffix of `s` starting with an element distinct from `a`. Thus, whenever we obtain a proof of `(reach_diff (H s) s)`, we can compute $s^\star$ using `Skip`.

However, to be able to compute the corecursive call `(Squeeze` $s^\star$`)`, we need to exhibit a value of `(Skippable (H` $s^\star$`)` $s^\star$`)`. This means that we need an algorithm that may compute, repeatedly and lazily, along the stream, a value of `(Skippable (H` σ`)` σ, where σ is any suffix of `s`. This is performed by `Always_`, the counterpart in **Type** of `Always`. So, we obtain the following definition:

```
Squeezable (s: Stream): Type :=
  Always_ (fun s => Skippable (H s) s) s.
```

The construction of `Squeeze` can now be completed as a cofixed point with dependent arguments `(s: Stream A)` and `(sq: Squeezable s)` (*n.b.*, we omit parameter `sq` when it is clear from the context).

As a direct consequence of the definition, we can show that a squeezed stream contains no duplicate:

```
Lemma Squeeze_Always_moves:
  ∀ (s: Stream) (sq: Squeezable s), Always moves (Squeeze s).
```

[5] Conversely, objects in sort **Prop** are only proofs of logical statements.

[6] ≅ is a generic notation that represents equality; when it applies on streams, it implements pointwise equality.

In the lemma, the predicate `moves` checks that a stream differs on its two first elements if they exist. The lemma is proven using a coinductive proof that follows the definition of `Squeeze`. It essentially relies on the fact that for every element `a` and stream `s` on which `(Skip a s)` can be evaluated (*i.e.*, which begins with a finite number of `a`), the first element of `(Skip a s)` is different from `a`.

4.2 Preserving Properties by Simulation

We now define the *simulation relation*. As usual, our simulation defines an abstract view, yet adapted to our context. Given two streams `X` and `Y`, `(Y` $\leq_{sim}$ `X)` means that `Y` is obtained from `X` by removing some of its duplicates, namely `Y` and `X` contains exactly the same elements, in the same order, yet each element is at most as duplicated in `Y` as in `X`. For instance, with `s = 1 2 2 3 3 3 3 4 5 6 6 7 8 8...` (ending with an infinite sequence of 8) and `s' = 1 2 2 3 3 4 5 6 7 8...`, we have `(s'` $\leq_{sim}$ `s)` since every value, from 1 to 8, appears in both sequences and the number of 1 (resp. 2, 3, 4, 5, 6, 7, 8) is smaller or equal in `s'`. The relation $\leq_{sim}$ is based on inductive and coinductive mechanisms, defined as follows:

```
CoInductive ≤sim : Stream → Stream → Prop :=
| sim_constant: ∀ a s, is_constant a s → O a ≤sim s
| sim_cons: ∀ a s1 s2 s3, s1 ≤sim s2 → C_plus a s2 s3 →
                          C a s1 ≤sim s3.
```

The first constructor `sim_constant` means that every stream made of one element `a` is smaller than any stream constantly made of `a` (albeit finite or infinite). The second constructor `sim_cons` means that given an element `a` and two streams `s1` and `s2` such that `s1` is smaller than `s2`, if the stream `s3` is obtained from `s2` by adding a positive number of `a` (namely, `(C_plus a s2 s3)` holds), then `(C a s1)` is smaller than `s3`. The predicate `C_plus` is inductively defined and `(C_plus a s1 (C a s2))` checks that either `s1` and `s2` are equal or `(C_plus a s1 s2)`.

We can show that $\leq_{sim}$ is a *partial order*, and as squeezing means removing all duplicates of a stream, we can prove that for a given squeezable stream, its squeezing is minimal *w.r.t.* $\leq_{sim}$ (see [2] for details):

```
Lemma Squeeze_is_min: ∀ (s: Stream) (sq: Squeezable s),
  Squeeze s ≤sim s ∧ ∀ x, x ≤sim s → Squeeze s ≤sim x.
```

We show that some properties can be transferred between $\leq_{sim}$-related streams. Precisely, a property `P` is defined to be (decreasing) *monotonic* (resp. *comonotonic*) *w.r.t.* $\leq_{sim}$ as follows:

```
monotonic P   := ∀ x y, x ≤sim y → P y → P x
comonotonic P := ∀ x y, x ≤sim y → P x → P y
```

The proof of Claim `(1)` requires to prove the *preservation* of the following properties. First, we prove a result related to the implication: for two predicates `P` and `Q`, such that `P` is comonotonic and `Q` is monotonic, we easily obtain that `(fun s => P s → Q s)` is monotonic. For some property `P` which is

monotonic (resp. comonotonic), `(Eventually P)` is monotonic (resp. comonotonic): indeed if `P` is reached by a given stream `y`, then it is also reached by any stream `x` that contains less (resp. more) duplicates. Similarly, for some monotonic property `P`, `(Always P)` is monotonic. Some other *ad-hoc* properties are proven (co)monotonic, if necessary, using straightforward coinductions.

4.3 Proof of Claim `(1)`

The core of the proof is to use `Squeeze` on `(execread1 e)` and to show that the result is actually an execution of $\mathcal{A}_1$.

To allow the use of `Squeeze (execread1 e)`*, we need to show that* `(execread1 e)` *is squeezable,* meaning that from any environment of `(execread1 e)`, we can decide whether the remaining sequence of environments is constant. This proof uses the fact that `e` is weakly fair and that the predicate `terminal` is decidable. First, if initially `e` is terminal for $\mathcal{A}_1$, then it remains so forever, and `(execread1 e)` is a constant sequence made of the environment `(H (execread1 e))` only. Second, we show that if initially, `e` is not terminal for $\mathcal{A}_1$, then necessarily, we have `reach_diff (H (execread1 e)) (execread1 e)` which means that `(execread1 e)` begins by a finite number of duplicates of `(H (execread1 e))`. Indeed, as `(H e)` is not terminal for $\mathcal{A}_1$, there exists a node which is enabled to execute its local algorithm $\mathcal{A}_1$ in e. It will remain continuously enabled until being activated or neutralized, meaning that the node or one of its neighbors has made a move of $\mathcal{A}_1$. This activation or neutralization eventually occurs due to the weakly fair assumption and the fact that $\mathcal{A}_1$ has priority over $\mathcal{A}_2$. Following this remark, the proof is done by induction on the weakly fair assumption. Third, whether or not `e` is initially terminal for $\mathcal{A}_1$ is decidable (Lemma `terminal_dec`). Hence, the proof that `(execread1 e)` is squeezable is performed coinductively and each step of the coinduction decides whether the current environment is terminal for $\mathcal{A}_1$.

So we can build `Se = Squeeze (execread1 e)` *and show it is an execution of* $\mathcal{A}_1$.
a) `Se` *is initiated under* `Assume1` since a stream and its squeezing have the same initial environment.
b) `Se` *is weakly fair.* We have `Se` $\leq_{sim}$ `(execread1 e)` by Lemma `Squeeze_is_min`. We show that `(execread1 e)` is weakly fair by induction and coinduction on the definition of `weakly_fair`. Now, we can prove that `weakly_fair` is monotonic directly using preservation properties. So, we conclude that `Se` is weakly fair.
c) `Se` *is a maximal run.* We first prove the following intermediate claim:

```
(c1) ∀ e1, Always moves e1 → e1 ≤sim execread1 e →
           is_max_run e1.
```

The proof is split into two subgoals: `(Always s_Step e1)` and `(Always s_terminal e1)`. Let `s` be any stream. `(s_Step s)` means that when `s` is made of at least two elements (*i.e.*, `s` is equal to some `(C a ss)`), then `(Step (H ss) a)` holds. `(s_terminal s)` means that when `s` is made of a single element `a` (*i.e.*, `s` is equal to `(O a)`)), then `(terminal a)` holds.

Subgoal 1 follows from the fact that `Always (fun e => moves e → s_Step e)` is monotonic. This latter can be shown by a direct coinduction, which mainly relies on the fact that as `s_Step` applies on the first two elements of the stream and can hold only when they are different.

For Subgoal 2, we use the assumption `(weakly_fair e)` to show the property

```
(c2) ∀ g1, is_constant g1 (execread1 e) → terminal g1.
```

namely, if `(execread1 e)` is constantly made of an environment `g1`, then `g1` is terminal. Actually, we proceed by contradiction and prove that if `g1` is not terminal, then there exists a node `n` which is enabled in `g1`. Therefore, due to the weakly fair assumption, `n` will be eventually activated or neutralized in $\mathcal{A}_1 \oplus \mathcal{A}_2$, hence in $\mathcal{A}_1$, since $\mathcal{A}_1$ has priority over $\mathcal{A}_2$ in $\mathcal{A}_1 \oplus \mathcal{A}_2$. Then, by induction on `(Eventually (AN n) e)`, we obtain that `e` cannot be constantly made of `g1`. Subgoal 2 is then obtained with a direct coinductive proof using Claim `(c2)`.

This concludes the proof of Claim `(c1)` which can be applied on `Se` since, again, `(Se` $\leq_{sim}$ `execread1 e)` and `(Always moves)` hold on `Se` (see Lemma `Squeeze_Always_moves`). Hence, `Se` is a maximal run. Actually, `e1` and `Se` represent one and the same stream, but working on `e1` has allowed to get rid of the (complex) construction of `Se = Squeeze (execread1 e)` in the proof.

As a conclusion, we deduce (by a), b), and c)) that the squeezing `Se` of the stream `(execread1 e)` is an execution of $\mathcal{A}_1$ and use it on $\mathcal{H}_2$ ($\mathcal{A}_1$ is silent). Hence, `Se` is finite, *i.e.*, it eventually reaches a terminal environment. As `Eventually` is comonotonic, `(execread1 e)` eventually reaches a terminal environment too, hence Claim `(1)` holds. This concludes the proof of convergence.

5 Conclusion

The *composition theorem* proves that hierarchical collateral composition preserves self-stabilization when applied under convenient assumptions, in particular assuming weakly fair executions. It comes with a *toolbox for squeezing streams* that was mandatory to achieve the proof of the theorem. As an example, we instantiated the theorem with the two first layers of the algorithm proposed in [20]. The first layer builds a rooted spanning tree on an identified connected network; the second layer assumes such a tree exists and computes a k-dominating set of the network ($k \in \mathbb{N}$) using this tree. Both algorithms are self-stabilizing under a weakly fair daemon, and our result certifies that their composition is also self-stabilizing and so builds a k-dominating set of an arbitrary connected identified network.

Composition techniques, in particular the hierarchical collateral composition, are widely used in the self-stabilizing area [27,23,18] because adopting a modular approach is unavoidable to design and prove complex present-day algorithms. Certification of such techniques is a step beyond traditional handmade proofs that offers hugely more confidence in the correctness of the result; and also a step towards the certification of complex multi-layered algorithms.

References

1. Abel, A., Pientka, B., Thibodeau, D., Setzer, A.: Copatterns: programming infinite structures by observations. In: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013 (2013)
2. Altisen, K., Corbineau, P., Devismes, S.: PADEC: A Framework for Certified Self-Stabilization. `http://www-verimag.imag.fr/~altisen/PADEC/`
3. Altisen, K., Corbineau, P., Devismes, S.: A Framework for Certified Self-Stabilization. Logical Methods in Computer Science (special issue of FORTE 2016) **13**(4) (2017)
4. Altisen, K., Devismes, S., Durand, A.: Concurrency in snap-stabilizing local resource allocation. J. Parallel Distrib. Comput. **102**, 42–56 (2017)
5. Auger, C., Bouzid, Z., Courtieu, P., Tixeuil, S., Urbain, X.: Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In: Stabilization, Safety, and Security of Distributed Systems - 15th International Symposium, SSS 2013, Osaka, Japan, November 13-16, 2013. Proceedings (2013)
6. Beauquier, J., Gradinariu, M., Johnen, C.: Cross-Over Composition - Enforcement of Fairness under Unfair Adversary. In: 5th International Workshop on Self-Stabilizing Systems, (WSS 2001), Springer LNCS 2194. pp. 19–34 (2001)
7. Bérard, B., Lafourcade, P., Millet, L., Potop-Butucaru, M., Thierry-Mieg, Y., Tixeuil, S.: Formal verification of mobile robot protocols. Distributed Computing **29**(6), 459–487 (2016). https://doi.org/10.1007/s00446-016-0271-1, `https://doi.org/10.1007/s00446-016-0271-1`
8. Bertot, Y.: Filters on CoInductive Streams, an Application to Eratosthenes' Sieve. In: Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings (2005)
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
10. Blin, L., Fraigniaud, P., Patt-Shamir, B.: On Proof-Labeling Schemes versus Silent Self-stabilizing Algorithms. In: 16th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2014), Springer LNCS 8756 (2014)
11. Caron, E., Chuffart, F., Tedeschi, C.: When self-stabilization meets real platforms: An experimental study of a peer-to-peer service discovery system. Future Generation Computer Systems **29**(6), 1533 – 1543 (2013)
12. Caron, E., Datta, A.K., Depardon, B., Larmore, L.L.: A self-stabilizing k-clustering algorithm for weighted graphs. J. Parallel Distrib. Comput. **70**(11), 1159–1173 (2010)
13. Castéran, P., Filou, V., Mosbah, M.: Certifying Distributed Algorithms by Embedding Local Computation Systems in the Coq Proof Assistant. In: Symbolic Computation in Software Science (SCSS'09) (2009)
14. Chen, M., Monin, J.F.: Formal Verification of Netlog Protocols. In: Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China (2012)
15. Coupet-Grimal, S.: An Axiomatization of Linear Temporal Logic in the Calculus of Inductive Constructions. J. Log. Comput. **13**(6), 801–813 (2003)
16. Courtieu, P.: 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 (2002)

17. Courtieu, P., Rieg, L., Tixeuil, S., Urbain, X.: Certified Universal Gathering in R^2 for Oblivious Mobile Robots. In: Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings (2016)
18. Datta, A.K., Devismes, S., Heurtefeux, K., Larmore, L.L., Rivierre, Y.: Competitive self-stabilizing k-clustering. Theor. Comput. Sci. **626**, 110–133 (2016)
19. Datta, A.K., Gurumurthy, S., Petit, F., Villain, V.: Self-Stabilizing Network Orientation Algorithms In Arbitrary Rooted Networks. Stud. Inform. Univ. **1**(1), 1–22 (2001)
20. Datta, A.K., Larmore, L.L., Devismes, S., Heurtefeux, K., Rivierre, Y.: Self-Stabilizing Small k-Dominating Sets. IJNC **3**(1), 116–136 (2013)
21. Dijkstra, E.W.: Self-stabilizing Systems in Spite of Distributed Control. Commun. ACM **17**(11), 643–644 (1974)
22. Doan, H.T.T., Bonnet, F., Ogata, K.: Model Checking of a Mobile Robots Perpetual Exploration Algorithm. In: Liu, S., Duan, Z., Tian, C., Nagoya, F. (eds.) Structured Object-Oriented Formal Language and Method - 6th International Workshop, SOFL+MSVL 2016, Tokyo, Japan, November 15, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10189, pp. 201–219 (2017)
23. Dolev, S.: Self-stabilization. MIT Press, Cambridge, MA, USA (2000)
24. Dubois, S., Tixeuil, S.: A Taxonomy of Daemons in Self-stabilization. CoRR **abs/1110.0334** (2011), `http://arxiv.org/abs/1110.0334`
25. Fei, L., Yong, S., Hong, D., Yizhi, R.: Self Stabilizing Distributed Transactional Memory Model and Algorithms. Journal of Computer Research and Development **51**(9), 2046 (2014)
26. Giménez, E.: An Application of Co-inductive Types in Coq: Verification of the Alternating Bit Protocol. In: Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers (1995)
27. Gouda, M., Herman, T.: Adaptive Programming. IEEE Transactions on Software Engineering **17**, 911–921 (1991)
28. Küfner, P., Nestmann, U., Rickmann, C.: Formal Verification of Distributed Algorithms. In: Theoretical Computer Science, vol. 7604, pp. 209–224. Springer Berlin Heidelberg (2012)
29. Kulkarni, S.S., Rushby, J.M., Shankar, N.: A case-study in component-based mechanical verification of fault-tolerant programs. In: 1999 ICDCS Workshop on Self-stabilizing Systems, Austin, Texas, June 5, 1999, Proceedings (1999)
30. Lamport, L.: How to write a 21st century proof. Journal of Fixed Point Theory and Applications **11**(1), 43–63 (2012)
31. Leroy, X., Grall, H.: Coinductive big-step operational semantics. Inf. Comput. **207**(2), 284–304 (2009)
32. Paulson, L.C.: Mechanizing Coinduction and Corecursion in Higher-Order Logic. J. Log. Comput. **7**(2), 175–204 (1997)
33. Pnueli, A.: The Temporal Logic of Programs. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 46–57. IEEE Computer Society (1977)
34. Tel, G.: Introduction to Distributed Algorithms. Cambridge University Press, 2nd edn. (2001)
35. The Coq Development Team: The Coq Proof Assistant Documentation (Jun 2012), `http://coq.inria.fr/refman/`