Model checking of distributed algorithms using synchronous programs

Jahier, Erwan; Altisen, Karine; Devismes, Stéphane; Sant'Anna, Gabriel B.

How to cite

JAHIER, Erwan et al. Model checking of distributed algorithms using synchronous programs. In: Theoretical computer science, 2025, vol. 1045, p. 115292. doi: 10.1016/j.tcs.2025.115292

This publication URL: https://archive-ouverte.unige.ch/unige:185085
Publication DOI: 10.1016/j.tcs.2025.115292

Theoretical Computer Science 1045 (2025) 115292

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Model checking of distributed algorithms using synchronous programs ☆

Erwan Jahier [a,*], Karine Altisen [a], Stéphane Devismes [b], Gabriel B. Sant'Anna [c]

[a] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France
[b] Universite de Picardie Jules Verne, MIS, Amiens, France
[c] Canonical, Brazil

ARTICLE INFO

Section Editor: Paul G. Spirakis

Handling Editor: Shlomi Dolev

Keywords:
Self-stabilization
Synchronous programming
Model-checking

ABSTRACT

The development of trustworthy distributed algorithms requires the verification of some key properties with respect to the formal specification of the expected system executions. The atomic-state model (ASM) is the most commonly used computational model to reason on self-stabilizing algorithms. In this work, we propose methods and tools to automatically verify the self-stabilization of distributed algorithms defined in that model. To that goal, we exploit the similarities between the ASM and computational models issued from the synchronous programming area to reuse their associated verification tools, and in particular their model checkers. This allows the automatic verification of all safety properties (including bounded liveness) of any algorithm under various asynchrony assumptions (from fully asynchronous to fully synchronous) and regardless of the hypotheses on the network (*e.g.*, on its topology, its edge and node labeling).

1. Introduction

Designing a distributed algorithm, checking its validity, and analyzing its performance is often difficult. Indeed, locality of information and asynchrony of communications imply numerous possible interleavings in executions of such algorithms. This is even more exacerbated in the context of fault-tolerant distributed computing, where failures, occurring at unpredictable times, have a drastic impact on the system behavior. Yet, in this research area, correctness and complexity analyses are usually made by pencil-and-paper proofs. As advances are made in distributed fault-tolerant computing, systems become more complex and require stronger correctness properties. As a consequence, the combinatorics in the proofs establishing functional and complexity properties of these distributed systems constantly increases and requires ever more subtle arguments. In this context, computer-aided tools such as simulators, proof assistants, and model checkers are appropriate, and sometimes even mandatory, to help the design of a solution and to increase confidence in its soundness.

Simulation tools [4,6] are interesting to test and find flaws early in the design process. However, simulators only partially cover the set of possible executions. In contrast, proof assistants [30] offer strong formal guarantees. However, they are semi-automatic in the sense that the user must write the proof in a formal language specific to the software, which then mechanically checks it. Usually,

☆ This work has been partially funded by the ANR project SkyData (ANR-22-CE25-0008-01).
* Corresponding author.
E-mail address: erwan.jahier@univ-grenoble-alpes.fr (E. Jahier).

https://doi.org/10.1016/j.tcs.2025.115292
Received 1 April 2024; Received in revised form 27 January 2025; Accepted 28 April 2025

Available online 6 May 2025

proof assistants require a considerable amount of effort since they often necessitate a full reengineering of the initial pencil-and-paper proof. Finally, and contrary to the two previous methods, model checking [11] allows a complete and fully automatic verification of the soundness of a distributed system for a given topology.

We consider model checking for distributed algorithms, focusing on *self-stabilization*, a versatile lightweight fault-tolerant paradigm [3,13]. Starting from an arbitrary configuration, a self-stabilizing algorithm makes a distributed system eventually reach a so-called *legitimate* configuration from which every possible execution suffix satisfies the intended specification. Since an arbitrary configuration may be the result of transient faults,[1] self-stabilization is commonly considered as a general approach for tolerating such faults in a distributed system. Our goal is to automatically verify the self-stabilization of distributed algorithms written in the atomic-state model (ASM), the most commonly used model in the area. To that end, we exploit the similarities between ASM and computational models issued from formal methods based on synchronous programming languages [17], such as LUSTRE [18], to reuse their associated verification tools, in particular model checkers such as KIND2 [7]. This allows the automatic verification of all safety properties (including bounded liveness) of any deterministic algorithm, regardless the assumptions made on network topologies and for various levels of asynchrony (synchronous and unfair daemons).

Contribution We propose a language-based framework, named SALUT, to verify the self-stabilization of deterministic distributed algorithms written in ASM. In particular, we implement a translation from the network topology to a LUSTRE program, based on API designed to encode the algorithms. The verification process then comes down to a state-space exploration problem performed by the model checker KIND2 [7]. Our framework is modular and flexible thanks to a clear separation between the description of algorithms, daemons, topologies, and properties to check. As a result, our framework is versatile and induces more simplicity by maximizing the code reuse. For example, using classical daemons (*e.g.*, synchronous, distributed, central) and standard network topologies (*e.g.*, rings, trees, random graphs) provided in the framework, the user just has to encode the algorithm and the properties to verify.

We demonstrate the versatility and study the scalability of our method by verifying many different self-stabilizing algorithms of the literature, solving both static and dynamic tasks in various contexts in terms of topologies and daemons. In particular, we include the common benchmarks (namely, Dijkstra's K-state algorithm [13], Ghosh's mutual exclusion [16], Hoepman's ring-orientation [20]) studied by the state-of-the-art, yet ad hoc, approaches [8,9,32] for comparison purposes. Our results show that the versatility of our solution does not come at the price of sacrificing too much efficiency in terms of verification time.

Related work Pioneer works on verification of distributed self-stabilizing algorithms have been led by Lakhnech and Siegel [28,31]. They propose formal frameworks to open the possibility of computer-aided-verification machinery. However, these two preliminary works do not propose any toolbox to apply and validate their approach.

Tsuchiya et al. [32] proposed to use the NuSMV [10] symbolic model checker. They validate their approach by verifying several self-stabilizing algorithms defined in ASM under the central and distributed daemon assumptions. These case studies are representative since they cover various settings in terms of topologies and problem specifications. Yet, their approach is not generic since it entangles in the same user-written SMV file the description of the algorithm, the expected property, the topology, and the daemon. Chen et al. [8] proposes a variant of the NuSMV [10] encoding of [32], that does not use any fairness assumption.

Chen and Kulkarni [9] use SMT solvers [14] to verify stabilizing algorithms. They apply bounded model-checking techniques to determine whether a given algorithm is stabilizing. They highlight trade-offs between verification with SMT solvers and the previously mentioned works on symbolic model checking [8,32]. In particular, the latter is more efficient, but limited to finite memory algorithms. Approaches in [8,9] are limited in terms of versatility and code reuse since, by construction, the verification is restricted to the central daemon, and again the whole system modeling is ad hoc and stored in to a single user-written file.

Whittlesey-Harris and Nesterenko [33] modeled in the SPIN [21] model-checker a specific yet practical self-stabilizing application, namely the fluids and combustion facility of the international space station, to automatically verify it. Few experimental results are given, but no analysis nor comparison with [32] is given.

SASA [4] is an open-source framework dedicated to the simulation of self-stabilizing algorithms in ASM. It provides all features needed to test, debug and evaluate self-stabilizing algorithms (such as an interactive debugger with graphical support, predefined daemons and custom test oracles). The SASA simulation facilities can actually be used with SALUT. The main difference is that algorithms should be written in OCAML rather than in LUSTRE – which is more convenient as LUSTRE is a more constrained language (it targets critical systems) and has a less rich programming environment. On the other hand, with SASA, one can only perform simulations.

Roadmap The rest of the paper is organized as follows. Sections 2 and 3 respectively present ASM and the synchronous programming paradigm. Section 4 proposes a general way of embedding ASM into a synchronous programming model. Section 5 shows how to take advantage of this embedding to formulate ASM algorithm verification problems. Section 6 describes a possible implementation of this general framework using the LUSTRE language and Section 7 explains how to use LUSTRE and KIND2 to perform automatic verifications in practice. Section 8 presents some experimentation results. In Section 9, we detail the model checking of two use cases using SALUT. In Section 10, we describe how SALUT integrates with the SASA toolchain. We make concluding remarks in Section 11.

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

Inputs:
K : a positive integer satisfying $K \geq n$
$q.Pred$: the predecessor of q in the ring

Variables:
$v \in \{0, ..., K-1\}$

Action for the root:
T_{root} :: $q.v = q.Pred.v \hookrightarrow q.v \leftarrow (q.v + 1) \ mod \ K$

Action for non-root processes:
T_p :: $q.v \neq q.Pred.v \hookrightarrow q.v \leftarrow q.Pred.v$

Fig. 1. The Dijkstra's K-state algorithm for n-size rooted unidirectional rings [13].

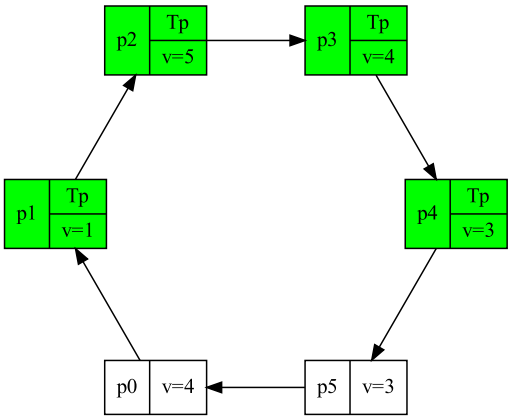

Fig. 2. Unidirectional ring of six processes rooted at *p0*.

2. The atomic-state model

A distributed system is a finite set of processes, each equipped with a *local algorithm* working on a finite set of local variables. Processes can communicate with other processes through communication links that define the network topology. In ASM [13], communications are abstracted away as follows: each process can read its variables and those of its neighbors or predecessors (depending on whether or not communication links are bidirectional) and can only write to its own variables. The local algorithm of each process is given as a collection of guarded actions of the following form: $\langle label \rangle :: \langle guard \rangle \hookrightarrow \langle statement \rangle$. The label is only used to identify the action. The guard is a Boolean predicate involving variables that the process can read. The statement describes modifications of the process variables. An action is *enabled* if its guard evaluates to true. A process can execute an action (precisely, its statement) only if the action is enabled. By extension, a process is said to be enabled when at least one of its action is enabled. An example of distributed algorithm is given in Fig. 1.

The semantics of a distributed system in ASM is defined as follows. A *configuration* consists of the set of values of all process states, the state of each process being defined by the values of its variables. An *execution* is a sequence of configurations, two consecutive configurations being linked by a *step*. The system *atomically* steps into a different configuration when at least one process is enabled. In this case, a non-empty set of enabled nodes is activated by an adversary, called *daemon*, which models the asynchronism of the system. Each activated process executes the statement of one of its enabled actions, producing the next configuration of the execution. Many assumptions can be made on the daemon. Daemons are usually defined as the conjunction of their *spreading* and *fairness* properties [3]. In this paper, we consider four classical spreading properties: central, locally central, synchronous, and distributed. A central daemon activates only one process per step. A locally central daemon never activates two neighbors simultaneously. At each step, the synchronous daemon activates all enabled processes. A distributed daemon activates at least one process, maybe more, at each step. Concerning the fairness, *every daemon considered in this paper is assumed to be either synchronous or unfair*. Unfairness is the most general fairness assumption: an unfair daemon might never select an enabled process unless it is the only remaining one.

Fig. 2 displays an example of distributed system where the algorithm of Fig. 1 runs. This algorithm is executed on a rooted unidirectional ring. By rooted, we mean that all processes except one, the root (here, *p0*), execute the same local algorithm. In the figure, each enabled process (colored) is decorated by the enabled action label (top-right). In the current configuration, processes from *p1* to *p4* are enabled because their *v*-variable is different from that of their predecessor; see Action T_p. The root process, *p0*, is disabled since its *v*-variable is different from that of its predecessor; see Action T_{root}. So, the daemon has to choose any non-empty subset of $\{p1, p2, p3, p4\}$ to be activated. In the present case, each activated process will copy its predecessor value during the step; see Action T_p.

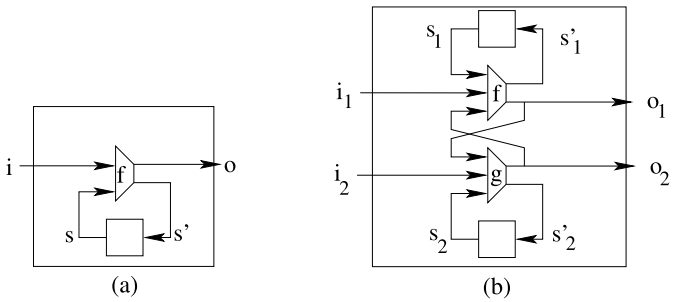

Fig. 3. General scheme of (a) a synchronous node and (b) synchronous composition.

A distributed system is meant to execute under a set of assumptions, which are in particular related to the topology (in the above example, a rooted unidirectional ring) and the daemon and to achieve a given specification (in the above example, the token circulation). Under a given set of assumptions, a distributed system is said to be *self-stabilizing* w.r.t. a specification if it reaches a set of configurations, called the *legitimate* configurations, satisfying the following three properties [3]:

- *Convergence:* every execution satisfying the assumptions eventually reaches a legitimate configuration.
- *Closure:* every execution satisfying the assumptions and starting from a legitimate configuration only contains legitimate configurations.
- *Correctness:* every execution satisfying the assumptions and starting from a legitimate configuration satisfies the specification.

3. The synchronous programming model

We now briefly recall the main concepts grounding the *synchronous programming paradigm* [17] used in the sequel. At top level, a synchronous program can be activated periodically (time-triggered) or sporadically (event-triggered). A program execution is therefore made of a sequence of *steps*. To perform such a step, the environment has to provide inputs. The step itself consists in (1) computing outputs, as a function of the inputs and the internal state of the program, and (2) updating the program internal state.

The specific feature of synchronous programs is the way internal components interact when composed: one step of the whole composition consists of a "simultaneous" step of all the components, which communicate atomically with each other. Moreover, programs have a formal *deterministic* semantics: this enables to validate the program using testing and formal verification.

Following the presentation in [17], a *synchronous node*[2] is a straightforward generalization of synchronous circuits (Mealy machines) that work with arbitrary datatypes: such a machine has a memory (a state) and a combinational part, and that computes the output and the next state as a function of the current input and the current state. The general dataflow scheme of a synchronous node is depicted in Fig. 3.a: it has a vector of inputs, i, and a vector of outputs, o; its internal state variable is denoted by s. A step of the node is defined by a function made of two parts, $f = (f_o, f_s)$: f_o (resp. f_s) computes the output (resp. the next state, s') from the current input and the current state:

$$o = f_o(i, s) \qquad s' = f_s(i, s)$$

The behavior of the node is the following: it starts in some initial state s_0. In a given state s, it deterministically reacts to an input valuation i by returning the output $o = f_o(i, s)$ and by updating its state by $s' = f_s(i, s)$ for the next reaction. Those nodes can be composed, by plugging one's outputs to the other's inputs, as long as those wires do not introduce any combinational loop. The general scheme of the (synchronous) composition between two nodes is shown in Fig. 3.b, where the step is computed by

$$o_1 = f_o(i_1, o_2, s_1) \qquad o_2 = g_o(i_2, o_1, s_2)$$
$$s'_1 = f_s(i_1, o_2, s_1) \qquad s'_2 = g_s(i_2, o_1, s_2)$$

and either the result of $f_o(i_1, o_2, s_1)$ should not depend on o_2 or the result of $g_o(i_2, o_1, s_2)$ should not depend on o_1 (this check is done by the compiler).

We now introduce two simple synchronous nodes that are used in the sequel. The first one is a single delay node, noted δ (Fig. 4.a): it receives an input i of some generic type τ and returns its input delayed by one step; it has a state variable s of type τ. A step of δ is computed by:

$$f^{\delta}_o(i, s) = s \qquad f^{\delta}_s(i, s) = i$$

[2] Here, what we name a (ASM) process is also often called a node in the literature; we have chosen to call it a process to avoid confusion with synchronous nodes.

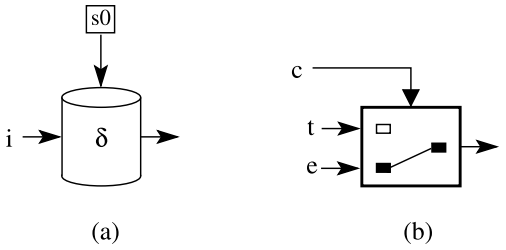

Fig. 4. (a) Delay synchronous node and (b) if-then-else synchronous node.

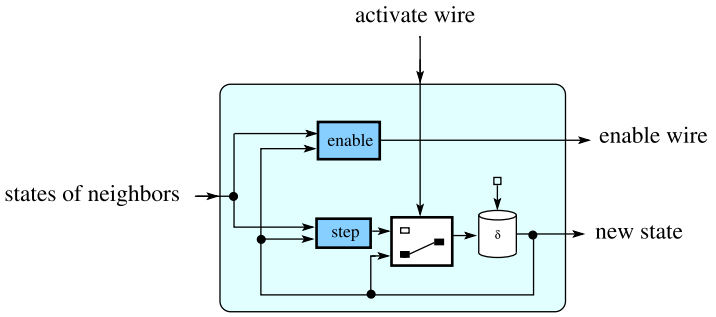

Fig. 5. Formalizing an ASM process as a synchronous node.

The second node (see Fig. 4.b) is a memoryless if-then-else operator: it returns its second input when its first input is true, and its third input otherwise:

$$f_o^{ite}(c,t,e) = \text{if } c \text{ then } t \text{ else } e \qquad f_s^{ite}(c,t,e) = \_$$

Since this node is stateless, $f_s^{ite}(c,t,e)$ returns nothing.

4. From ASM processes to synchronous nodes

ASM and synchronous programming models have a lot in common, in particular with respect to the atomicity of steps: all nodes of the program (resp. all processes of the network) react at the same logical instant, using the same global configuration; moreover, at the end of a global step, all nodes (resp. processes) outputs are broadcast away instantaneously to define the new configuration. Another important similarity is the way the non-determinism is handled. As a synchronous program is deterministic, non-determinism is handled by adding external inputs, that may be arbitrary.

On the other hand, in ASM, non-determinism due to asynchronism is modeled by daemons. For those reasons, using synchronous programs (and their associate toolboxes) is very natural to simulate and formally verify ASM algorithms.

We now explain how to encode ASM processes into synchronous nodes. In a network of n processes, each process is mapped to a synchronous node. This node contains two inner nodes encoding ASM guarded actions of the process (see Fig. 5): (1) `enable`, whose inputs are the states the process can read (*i.e.*, the predecessors in the graph); this node has a single output, a Boolean array, which elements are true if and only if the corresponding process guards are enabled; (2) `step`, with the same inputs as `enable`, and that outputs a new state (as computed by the statement of the enabled action[3]); this state is used as the new value of the corresponding process state if the daemon chooses to activate the process (see the activate wire in Fig. 5); the previous value is used otherwise.

The communication links in the network topology are data wires in the synchronous model. For each process, the new state output wire of its node instance is plugged onto some other node instances, corresponding to its neighbors, as defined by the network topology; see the leftmost outer node in Fig. 6.

5. ASM algorithms verification via synchronous observers

Once we have a formal model (made of synchronous nodes) of the processes local algorithms and the network, it is possible to automatically verify properties using so-called *synchronous observers* [19]. A synchronous observer encodes a property of a system using a synchronous node that observes the behavior of its outputs and returns a Boolean value that indicates whether the sequence of those outputs is correct. This technique is therefore limited to *safety properties*. It can be used during simulations to implement test

[3] If several actions are enabled, the daemon selects at least one of them.

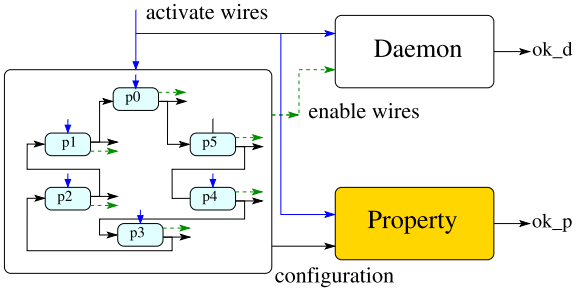

Fig. 6. Verifying a property using synchronous observers.

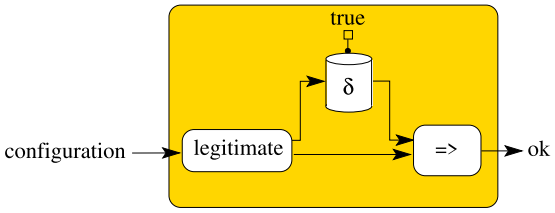

Fig. 7. The closure property Observer.

oracles that provide verdicts on whether the tests pass or not [25], since observers can be executed; we use it here for verification purposes.

First, the assumptions on the environment of the system under verification are formalized by a synchronous observer; here, those assumptions are handled by the daemon that decides which processes should be activated among the enabled ones. Therefore, the assumption observer, named *daemon*, has $2 \times n$ input wires: n activate wires and n enable wires, one each per process; it outputs a Boolean whose value states the fact that the assumption made on the daemon is satisfied. This explains why our whole framework is limited to unfair or synchronous daemons, since other fairness assumptions are not safety properties. The classical daemon assumptions (synchronous, distributed, central, ...), encoded as synchronous nodes, are provided as a library [1].

Second, the (safety) *property* to be verified is also encoded as a synchronous observer. For example, we provide how to define the closure property of the self-stabilization definition (see Section 2). This property states that an algorithm never steps from a legitimate to an illegitimate configuration. The corresponding synchronous node is provided in Fig. 7 where the fact of being legitimate for a configuration is also encoded in a synchronous node. Now, the observer checks that if the previous configuration (computed by the δ node) was legitimate, then so is the current one.

Once defined, both synchronous observers (*i.e.*, the assumptions and the property) are composed with the synchronous nodes encoding the topology and local algorithms. This synchronous composition is illustrated in Fig. 6, where a property is checked against ASM algorithms running on the network of Fig. 2. For the sake of clarity, we have omitted some wires: the processes output wires from left-to-right holding the state values are plugged into the configuration wire of the property node; the processes output wires from left-to-right holding the enable values are plugged into the enable wire of the daemon node; the processes input wires from up-to-down holding the activation values, that are also used as inputs for the daemon observer, are plugged into the corresponding processes.

Finally, the verification of the property consists in checking that the whole composition never causes the synchronous observer of the property to return false while the daemon observer has always returned true. This boils down to a state-space exploration problem performed by some model-checker.

6. SALUT: self-stabilizing algorithms in LUSTRE

In this section, we describe SALUT, a framework that implements the ideas presented so far. In order to implement such a framework, one has to (1) choose a format to describe the network, (2) choose a language to implement synchronous nodes, (3) propose an API for that language to define `enable` and `step` functions, and (4) implement a translator from the format chosen in (1) to the language chosen in (2).

6.1. Network description

We have chosen to base the network description on DOT [15]: the rationale for choosing DOT was that many visualization tools and graph editors support the DOT format and many bridges from one and to another graph syntax exist. DOT *graphs* are defined as sets of *nodes* and *edges*. Graphs, nodes, and edges can have *attributes* specified by name-value pairs, so that we can take advantage of

DOT attributes to (1) associate nodes with their algorithms, (2) optionally associate nodes with their initial states, and (3) associate graphs with parameters.

```
digraph diring6 {
graph [ links_number=6 ]
  p0 [algo="root.lus"]  p1 [algo="p.lus"]  p2 [algo="p.lus"]
  p3 [algo="p.lus"]     p4 [algo="p.lus"]  p5 [algo="p.lus"]
  p0 -> p1 -> p2 -> p3 -> p4 -> p5 -> p0
}
```

Listing 1: The dot file used to generate Fig. 2.

6.2. LUSTRE, a language to implement synchronous nodes

LUSTRE is a dataflow synchronous programming language designed for the development and verification of critical reactive systems [18]. It combines the synchronous model and the dataflow paradigm – which is based on block diagrams, where blocks are parallel operators concurrently computing their own output and possibly maintaining some states. Moreover, two LUSTRE model checkers are freely available to perform formal verifications (see Section 7).

6.3. A LUSTRE API to define ASM algorithms

We now present how to use the SALUT LUSTRE API to implement a local algorithm on a process in the ASM, following the formalization in Section 4. For each algorithm, one needs to define the process state datatype. Then, for each local algorithm, one needs to define a LUSTRE version of the enable and step nodes – the 2 leftmost inner nodes of Fig. 5.

For Dijkstra's K-state algorithm of Fig. 1, the state of each process is an integer and there is one action for the root process and one action for non-root processes:

```
type state = int;
const actions_number = 1;
```

The `root_enable` and `root_step` are implemented in Listing 2 and 3 for the root process. Their interfaces (lines 1-2) are the same for all nodes and all algorithms.

```
1  function root_enable <<const d: int>>(st:state; ngbrs:neigh^d)
2  returns (enabled: bool^actions_number);
3  let
4    enabled = [ st = state (ngbrs[0]) ];
5  tel ;
```

Listing 2: The LUSTRE `enable` node for the root process

The `enable` nodes take as inputs the state of the process (of type `state`) and an array containing the states the process has access to – namely, its predecessor in the ring. Such states are provided as an array of size `d`, where `d` is the degree of the process. As in LUSTRE, array sizes should be compile-time constants, the `d` parameter is provided as a static parameter (within «»). Type `neigh` contains information about every process the node can directly access (depending on the context, a neighbor or a predecessor), in particular its state, accessed using the `state` getter (see line 4 of Listing 2). Enable nodes return an array of Booleans of size `actions_number`, stating for each action whether it is enabled or not.

The K-state algorithm of the root process is enabled when the process state value is equal to that of its predecessor (line 4 of Listing 2), as stated in the guard of the root action in Fig. 1. In LUSTRE, stateless nodes are declared as `function` (line 1 of Listing 2 and 3) and square brackets (`[index]`) gives access to the content of the array at a particular `index`.

```
1  function root_step <<const d: int>>(st: state; ngbrs: neigh^d; a:action)
2  returns (new: state);
3  let
4     new = (st + 1) mod k;
5  tel;
```

Listing 3: The LUSTRE `step` node for the root process.

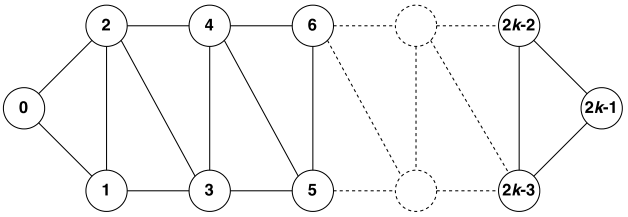

Fig. 8. Example of topology for the Ghosh's algorithm.

Action for the top process $2k-1$**:**
$A_{top} :: s[2k-1] = s[2k-2] \hookrightarrow s[2k-1] \leftarrow \neg s[2k-1]$

Action for the bottom process 0**:**
$A_{bottom} :: s[0] \neq s[1] \hookrightarrow s[0] \leftarrow \neg s[0]$

Action for the x**-process** $2i$ **with** $i \in [1..k-1]$**:**
$A_x :: s[2i-2] = s[2i-1] = s[2i+1] \neq s[2i] \hookrightarrow s[2i] \leftarrow \neg s[2i]$

Action for the y**-process** $2i-1$ **with** $i \in [1..k-1]$**:**
$A_y :: s[2i-2] = s[2i-1] = s[2i] \neq s[2i+1] \hookrightarrow s[2i-1] \leftarrow \neg s[2i-1]$

Fig. 9. Actions of the Ghosh's algorithm [16].

The `step` nodes have the same input parameters as `enable` ones, plus the active action label (see `a` in Listing 3, line 1). It returns the new value of the process state (line 2). The node body (line 4) is a direct encoding of the statement of the root action given in Fig. 1. The predefined node `mod` computes the modulo operation. For this algorithm, there is only one possible action, so the argument `a` is not used.

The `enable` and `step` nodes are similarly implemented for the non-root processes (see [22] for the complete implementation).

6.4. Another example: the Ghosh's algorithm LUSTRE encoding

In [16], Ghosh proposes to exhibit nontrivial network topologies where self-stabilizing mutual exclusion can be solved using only one bit of memory per process. Legitimate configurations of this algorithm are those where exactly one process is enabled.

The considered topologies contain $2k$ processes with $k > 2$. Processes (named *machines* in [16]) are labeled from 0 to $2k-1$ and split into four types: the *bottom process* labeled 0, the *top process* labeled $2k-1$, the *x-processes* labeled $2i$ with $i \in [1..k-1]$, and the *y-processes* labeled $2i-1$ with $i \in [1..k-1]$. An example of topology is given in Fig. 8.

The state $s[i]$ of each process $i \in [0..2k-1]$ is a single boolean value. Each process has a single action allowing it to flip the value of its state, as described in Fig. 9.

Consider a configuration γ where all bits are equal. γ is legitimate since only one process is enabled, the top one. A first bit flip occurs at the top process (using Action A_{top}). Next, bit flips are propagated along the path $2k-3, 2k-5, \ldots, 1$ (using Action A_y). Then, the bottom process flips its bit (using Action A_{bottom}) and bit flips are propagated back toward the top process along the path $2, 4, \ldots, 2k-2$ (using Action A_x) until reaching γ again.

According to the actions described in Fig. 9, we have one action for each type of process.

```
type state = bool;
type action = enum { T };
const actions_number = 1;
```

The step function is the same for all processes: when a process is activated, its new state is made of the negation of its old state. We give the one of the top process as an illustrative example.

```
function p_top_step<<const d:int>>(st:state;neighbors:neigh^d;a:action)
returns (new_st : state);
let
  new_st = not st;
tel;
```

Then, we have the following guards implemented as one enable node per kind of processes.

```
function p_top_enable<<const d:int>>(s2:state;neighbors:neigh^d)
returns (enabled:bool^actions_number);
let
  enabled = [ state(neighbors[1]) = s2 ];
tel;
function p0_enable<<const d:int>>(st:state; neighbors:neigh^d)
returns (enabled : bool^actions_number);
let
  enabled = [ st <> state(neighbors[0]) ];
tel;
function p_y_enable<<const d:int>>(st:state; neighbors:neigh^d)
returns (enabled : bool^actions_number);
let
  enabled = [ state(neighbors[0]) = state(neighbors[1]) and
              state(neighbors[2]) = state(neighbors[1]) and
              st = not state(neighbors[0])           ];
tel;
function p_x_enable<<const d:int>>(st:state;neighbors:neigh^d)
returns (enabled : bool^actions_number);
let
  enabled = [ state(neighbors[0+if d=3 then 0 else 1]) = st and
              st = state(neighbors[1+if d=3 then 0 else 1]) and
              state(neighbors[2+if d=3 then 0 else 1]) =
                   not state(neighbors[1+if d=3 then 0 else 1]) ];
tel;
```

The encoding of Fig. 9 in LUSTRE is direct, except for the use of the degree in `p_x_enable`. Indeed, all y-processes have 4 neighbors, expect Process 1 that has 3. Beside, y-processes with 4 neighbors only need the values of 3 of them; and since SALUT sorts processes is the neighbors array in alphabetical order, the useless neighbor is the first one in the array, hence the shift of 1 in the LUSTRE encoding when $\texttt{d} \neq 3$.

6.5. *The SALUT translator*

Most of the nodes required to describe ASM algorithms are generated automatically from the network topology using a DOT to LUSTRE translator. Only the `enable` and `step` nodes need to be provided. SALUT generates from the DOT file a node, called `topology` (see the leftmost outer node in Fig. 6), that reproduces in LUSTRE the network topology. In particular, SALUT takes care of wiring the `enable` and the `step` node instances to the appropriate processes and the appropriate values of parameter `d` (the node degree, that can vary from one node instance to another).

7. Automatic formal verification

We have seen in Section 5 and Fig. 6 that safety properties on ASM algorithms can be encoded using synchronous observers. By defining such observers in LUSTRE, we can perform the verification of these properties automatically using existing verification tools for LUSTRE such as KIND2 [7]. Technically, to use such a tool, one just needs to point out a Boolean variable in the node. Then, the tool will try to prove that the designated variable is always `true` for all possible sequences of the node inputs, by performing a symbolic state space exploration. Hence, one just needs to encode the desired properties into a Boolean variable, as done in the `verify` node given in Listing 4. This section is devoted now on how to express useful properties, using Listing 4 as a guideline.

Remember that synchronous observers can only encode safety properties. Therefore, this framework will be able to prove that a self-stabilizing algorithm is correct, up to this restriction.

An algorithm is self-stabilizing w.r.t. a given specification (see Definition, page 4) if the three following conditions hold:

- *Convergence:* every execution should reach a legitimate configuration in finite time. This is a typical liveness property in general; but most of the time, convergence can be expressed as a safety property using a synchronous observer; see Section 7.2.
- *Closure:* the set of legitimate configurations is closed. This is a typical safety property and its observer has already been presented as an example in Section 5, see Fig. 7.
- *Correctness:* every execution started in a legitimate configuration should satisfy the specification. Depending on the nature of the specification property, it can be encoded as a synchronous observer or not. For instance, the K-state algorithm solves a token circulation problem. The fact that there exists a unique token is a safety property but the circulation property is, in general, a liveness property which cannot be encoded as a synchronous observer. However, very often, any liveness properties can be reformulated as a safety ones. For example, in the K-state algorithm, every process holds the token every n steps during an execution started in a legitimate configuration.

The `verify` node in Listing 4 encodes the aforementioned properties for the K-state algorithm. The node output `ok` corresponds to the wire called ok_p in Fig. 6. This node has two input variables (line 4): `init_config`, that holds the initial configuration of

```
1  const n = card;  -- processes number extracted from the dot file
2  const worst_case = n*(n-1) + (n-4)*(n+1) div 2 + 1; -- in steps
3
4  node verify(active: bool^1^n; init_config: state^n)
5  returns (ok: bool);
6  var
7    config: state^n;
8    enabled: bool^1^n; -- 1 as the algorithm has only 1 rule per process
9    enabled1: bool^n;  -- enabled projection
10   legitimate, round: bool;
11   closure, converge_cost, converge_wc, converge: bool;
12   steps, cost, round_nb: int;
13 let
14   assert(true -> init_config = pre(init_config));
15   assert(true -> daemon_is_central<<1,n>>(active, pre(enabled)));
16   config, enabled, round, round_nb = topology(active, init_config);
17   --
18   enabled1 = map<<nary_or<<1>>,n>> (enabled); -- projection
19   legitimate = nary_xor<<n>>(enabled1);
20   closure = true -> (pre(legitimate) => legitimate);
21   --
22   cost = cost(enabled, config);
23   converge_cost = (true -> legitimate or pre(cost)>cost);
24   steps = 0 -> (pre(steps) + 1); -- 0, 1, 2, ...
25   converge_wc = (steps >= worst_case) => legitimate;
26   converge = true -> ((init_config = config) => legitimate);
27   --
28   ok = closure and (converge_cost or converge_wc or converge);
29 tel;
```

Listing 4: LUSTRE formalization of some properties of the K-state algorithm.

the system, and `active` which is a matrix of Booleans of size `1` × `n` indicating which processes are activated. The first dimension is used to deal with algorithms that are made of several guarded actions, here only one is used; and `n` is the number of processes.

The model checker will verify the following property: *for every initial configuration and every activation sequence, variable* `ok` *remains true forever*. The verification is done on every execution where the `assert` properties (lines 14-15) hold forever. The first assert ensures that the variable `init_config` remains constant (while keeping its initial value). The second one encodes the daemon assumptions based on the processes which are activated (variable `activate`) among the enabled ones (see variable `enabled`, line 8, which has the same type as variable `activate`).

Here a central daemon is used (the node `daemon_is_central` not shown in the listing): it checks that exactly one enabled node is actually activated. Note that such a property is not enforced at the first instant (`true->...`[4]) since `pre(enabled)` (and `pre(init_config)`), which returns the previous value of `enabled`, is undefined at that instant.

The `topology` node (line 16) computes a new configuration; see variable `config`. The `topology` node also outputs elements relative to round computation, which are not used here. The configuration returned by the `topology` node at the first step is the initial one, given by `init_config`; for all other steps, the configuration is computed by `topology` from the previous configuration (which is stored as an internal memory in `topology`; see Fig. 5) and the process activations. At every step, the set of enabled processes is computed according to the configuration.

In the following, we describe the rest of the `verify` node (from line 17) which encodes several properties to be verified. The `ok` variable (line 28) gathers those properties.

7.1. Closure

The `verify` node contains the encoding of the closure property (line 20): once the system has reached a legitimate configuration, it remains in legitimate configurations forever. The definition of a legitimate configuration is done with the variable `legitimate` in line 19: in the example, a configuration is legitimate when it holds a unique token, *i.e.*, exactly one process is enabled. This is checked using a XOR operator (*n.b.*, `nary_xor` is a node that returns `true` if and only if exactly one element of its input Boolean array is `true`). Then, the definition of closure is given line 20 and is a direct implementation of Fig. 7. Again, this property is not checked at the first instant when `pre(legitimate)` is undefined.

7.2. Convergence

The convergence of some algorithms can be expressed as a safety property. We present three methods to prove convergence using our framework in the following.

[4] The `->` infix binary operator returns the value of its left-hand-side argument at the first instant, and the one of its right-hand-side argument for all the remaining instants.

7.2.1. Cycle detection

The method we propose here is limited to unfair and synchronous daemons, and to deterministic algorithms with finite memory. Under those assumptions, the convergence can be proved by verifying that no cycle exists in the set of illegitimate configurations. Formally, the property is expressed in line 26 as follows: *for every initial configuration, at every step, no configuration cycle has been reached unless the configuration is legitimate*.

Indeed, when the domain of each variable is finite, the number of possible configurations in a given network is finite too. In such a case, an infinite execution contains at least one cycle. If the model-checker is able to verify the property in line 26 and the closure property line 20, this means that no execution contains a cycle of illegitimate configurations, hence every execution reaches a legitimate configuration in a finite number of steps.

Conversely, the model-checker may find a counter-example, namely a prefix of execution made of illegitimate configurations that is cyclic from its initial configuration. For unfair and synchronous daemons, such a cyclic execution prefix can be repeated forever as their choices are not constrained by previous steps, and this violates the convergence property.

Notice that the previous arguments do not hold if the daemon includes fairness requirements. Indeed, a fairness property of the daemon may force it to activate processes whose action will make the execution leave the cycle.

7.2.2. Using a known worst-case

Alternatively, we can take advantage of known upper bounds for the convergence time, being tight or not. In the example, we use the fact that the stabilization time of the algorithm is upper bounded by a worst_case of $n \times (n-1) + (n-4) \times (n+1)/2 + 1$ (Theorem 6.30, [3]). We can check this bound, and so the convergence, by stating that once the upper bound is reached, the configuration should be legitimate. The `steps` variable counts the number of steps elapsed since the beginning of the execution (line 24) and the `convergence_wc` Boolean variable checks the property (lines 2 and 25).

7.2.3. Using a potential function

For algorithms with an available *potential function*, which quantifies how far a configuration is from the set of legitimate configurations, we can check whether this function is decreasing. In the example, the `convergence_cost` Boolean variable (line 23) encodes this property. The `cost` node is not shown here but implements the potential function defined in [2]. The existence of a decreasing function guarantees the convergence. Note that once a legitimate configuration is reached, the potential function may no longer necessarily decreases. This is the kind of subtlety that a verification tool can spot (and has actually spotted) easily.

8. Model-checking experimentations

8.1. Size of topologies that can be model-checked

One of the key advantages of our approach is that topologies, daemons, algorithms, and properties to check are described separately, contrary to the related work [8,9,32] where they are entangled into a single user-written SMV [10] or Yices [14] file. More precisely:

1. SALUT automatically translates into LUSTRE any topology described in the DOT language (for which many graph generators exist).
2. Classical daemons, *i.e.*, synchronous, distributed, central, and locally central, are generically modeled in LUSTRE so that they can be used for any number of nodes and actions (using 2-dimensional arrays). Other daemons can be modeled similarly.
3. Thus, to model-check an algorithm, one just needs to model its guarded actions, using the API described in Section 6.3. Actually, we have done it for several different algorithms: the Dijkstra's K-state algorithm [13], the Ghosh's mutual exclusion [16], a Bread-First Search spanning tree construction [3], a synchronous unison [3], a k-clustering (with k = 2) algorithm [12], a vertex-coloring algorithm [3], and the Hoepman's ring orientation [20].[5]
4. For all those algorithms, we have encoded the closure property and a convergence property based on a known upper bound. We also have encoded a convergence property based on a potential function, when available.

Once an algorithm and the properties to verify are written in LUSTRE, we can model-check them using any daemon and any topology. Of course, not all combinations make sense, *e.g.*, the Dijkstra's K-state algorithm only works on rooted unidirectional rings and the synchronous unison only works under a synchronous daemon. Still, a lot of combinations are possible; Table 1 presents experimental results for a small subset of them.

Table 1 summarizes experiments made with the KIND2 [7] model checker[6] to prove some properties on the corresponding algorithms. Column 1 contains the algorithm name and column 2 the size of its LUSTRE encoding. Columns 3 and 4 respectively contain the topology and the daemon used for the experiment. Column 5 contains the number of lines of LUSTRE code used to encode the convergence properties, apart from the main node declarations; greater ones (K-state and k-clustering) are due to the potential func-

[5] The LUSTRE code of those examples is given in the SALUT git repository [1].

[6] We use KIND2 v2.1.1 with the following command line option: `-smt_solver Bitwuzla -enable BMC -enable IND -timeout 3600` and `uint8` for representing integers, except for K-state/synchronous where `uint16` is necessary (indeed, for n > 13, the worst case in steps is greater than 255).

Table 1
The maximum topology size that can be handled within an hour.

Algorithm	Program size in loc	Topology	Daemon	Properties size in loc	Max topo size WC (POT) in processes nb
K-state	38	rooted directed ring	synchr. central distrib.	100	48 (15) 6 (8) 6 (8)
Ghosh	50	"ring-like"	central distrib.	10	18 16
Coloring	30	ring random[a]	central	16	10 (55) 11 (26)
Sync unison	40	random ring	synchr.	8	40 17
BFS sp. tree	80	tree	distrib.	9	8
k-clustering	130	rooted tree	distrib.	50	9 (10)
Hoepman	110	odd-size ring	central	8	7

[a] Random graphs were generated according to the Erdős–Rényi model.

Table 2
Maximum topology sizes handled within an hour using various approaches.

Algo/demon \ Method	SALUT+KIND2	SALUT-like+NUSMV	[32]+NUSMV	[8]+NUSMV
K-state/central	6 (8)	9	10	10
K-state/distributed	6 (8)	8	9	9
Ghosh/central	18	18	30	46
Ghosh/distributed	16	14	32	14

tion encoding. Column 6 contains the maximal number of processes for which we get a (positive) result in less than 1 hour[7] (and ditto for the potential-based convergence, when available, in parentheses).

The topology sizes we can handle are small, but large enough to spot faults in algorithms, expected properties, or their LUSTRE encoding. Potential functions, when available, sometimes allow to check bigger topologies; in fact, we are able to check in less than one hour a ring of 55 processes using the potential function of the coloring algorithm, whereas using the worst-case-based convergence, we are only able to check the algorithm convergence on rings of size 10. The closure property is often much cheaper to model-check. For instance, in less than an hour, we are able to check the Dijkstra's K-state algorithm on a unidirectional rooted ring made of 45 processes under a distributed daemon.

8.2. The cost of genericity

In order to evaluate our approach, we have performed some experiments to compare it with the encodings proposed in [8] and [32] – those two latter are similar; [32] uses modules, whereas in [8] module definitions are inlined.

Table 2 summarizes the result of those experiments performed on the K-state and Ghosh's algorithms, using central and distributed daemons, where we searched for the biggest topologies that can be model-checked in less than an hour. The second column shows the result obtained using SALUT (that can also be found in Table 1; the numbers in parentheses are the ones obtained using a potential function). The fourth and fifth columns show the results obtained with NUSMV[8] and the encodings of [32] and [8], respectively – using an ad hoc NUSMVprogram generator parameterized by the topology size. The third column will be explained later. In Column 4, the fact that we get better results for Ghosh with a distributed daemon has already been noticed and explained in [32]. In Column 5, modules inlining creates exponentially big files (2^n) for distributed daemons which makes NUSMV fail from n > 14 (segmentation fault).

Table 2 shows that the direct encoding gives better performances than the one of SALUT; see Columns 2, 4, and 5. In order to understand why, let us take an abstract view on the [8] direct encoding for central daemons. For each process $i \in \{1, \ldots, n\}$, let us call n_i its set of neighbors states ($n_i = \text{neigh}(i)$), s_i its state value, e_i a Boolean that is true when i is enabled ($\text{enab}(s_i, n_i)$), and a_i a Boolean that is true when i is activated. Then, *next*, the transition system function to be model-checked can be defined by the following formula:

[7] We used a multi-core server, where each core is made of Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz. Note that KIND2 is able to use several cores in parallel.
[8] We used NUSMV version 2.6.0.

$$\bigvee_{i=1}^{n}(e_i \wedge next(s_i) = \text{step}(s_i,n_i) \wedge \bigwedge_{j\neq i} next(s_j) = s_j) \tag{1}$$

Note that in these settings, the algorithm is encoded in the enab and step functions, whereas the topology is encoded in the neigh one.

SALUT uses KIND2, which is based on SMT (Sat Modulo Theory) solvers. In order to figure out to which extent the performance gain is due to the use of NUSMV, we have performed the following encoding, that mimics what SALUT/KIND2 provides to SMT solvers:

$$(\bigwedge_{i=1}^{n} e_i \Rightarrow a_i \wedge next(s_i) = \text{if } a_i \text{ then step}(s_i,n_i) \text{ else } s_i) \wedge \oplus_{i=1}^{n} a_i \tag{2}$$

Notice that in this encoding, contrary to formula (1), the constraints of the daemon are completely separated from the other aspects, like in the ASM semantics encoding. In particular, for distributed daemons, one just needs to replace the final exclusive disjunction ($\oplus_i a_i$) by an disjunction ($\bigvee_i a_i$) in formula (2), to state that several processes can be activated.

Comparing the results in Columns 2 and 3 suggests that a direct encoding in NUSMV can increase the performance for the K-state algorithm, but not for the Ghosh's one. This can be due to the modularity of the approach, where everything is described separately, and then automatically generated. But the difference between Column 3 on one side and Columns 4 and 5 on the other side suggests another source of performance loss. Indeed, the main difference is that Formula (1) (used for Columns 4 and 5) is in Disjunctive Normal Form (DNF), whereas Formula (2) is not (used for Column 3). This comes from the fact that, in our framework, the definition of the daemon is not entangled with the definition of the ASM semantics. This is good for separation of concerns, but the experiment suggests that it is bad (but not catastrophic) for performance.

As far as SMT-based verification is concerned, the proposed framework used with the KIND2 model checker does not seem to pay the price of genericity, as we get performances that have the same order of magnitude with those of Chen et al. [9]. Indeed, for the K-state and the Ghosh's algorithms convergence under a central daemon, and using a timeout of one hour as we did in Table 1, Chen et al. [9] report to model-check topologies of size 5 and 14, respectively. We are able to handle slightly bigger topologies (6 and 18, respectively). But the difference is not significant and our numbers were obtained using a more recent computer.

9. Use cases

Below, we illustrate the utility of model checking for distributed algorithms with two use cases: proving self-stabilization in small topologies and searching for counter-examples.

9.1. The K-state algorithm for rings of size 3

Usually, proofs of distributed algorithms follow a general pattern starting from a given value of some global network parameter. To prove the correctness or complexity of a distributed algorithm for smaller parameter values, we often have to deal with several edge cases that make the proof more combinatorial and therefore error-prone. In this case, model checking is a valuable alternative to obtain sounder results. Below, we illustrate this claim in the K-state algorithm.

The stabilization time in steps of the K-state algorithm can be expressed as an analytical function which depends on the number of processes n, as far as n is greater than 4. The bound has been proven tight in [2] using the Coq proof assistant. Now to obtain the values and tightness (*i.e.*, an execution that matches the bound) for the peculiar values $n \leq 3$, we have first used the model-checking approach (about 10 lines of code in this framework; see Listing 3). As a matter of comparison, the mechanized proof with those cases requires up to 1400 lines of ad hoc Coq code! [2]

9.2. Synchronous unison

The *synchronous unison* is a clock synchronization problem which requires the system to be synchronous. In this problem, each process holds a local clock, and at each step, all clocks have to synchronously increment modulo a so-called period $K \geq 2$ so that all clocks of the network are always equal.

A self-stabilizing synchronous unison algorithm written in ASM is studied in [3]. It assumes a synchronous daemon and works in every anonymous arbitrarily connected bidirectional network. Actually, it is a straightforward adaptation of an algorithm originally proposed for a synchronous link-register model [5]. The correctness proof given in [3] establishes the self-stabilization of the algorithm as soon as the period K satisfies $K \geq \max(2, 2\mathcal{D}-1)$, where $\mathcal{D}$ is the network diameter. The tightness of this bound is also investigated. However, obtained results are partial: the bound is proven to be tight in infinitely many networks but for *even* values of K only. More precisely, it is shown that for every bidirectional line L of diameter $\mathcal{D} > 1$, and every *even* value K in $[2..2\mathcal{D}-1)$, there exists an execution of the algorithm that does not stabilize. Hence, the tightness of the bound for odd values of K remains an open question. Intuitively, the stabilization of the synchronous unison is more difficult when the diameter of the network is large. In fact, the bidirectional line of n nodes is the connected graph of n nodes where the diameter is maximum ($\mathcal{D} = n-1$). Unfortunately, we could not extend the results on bidirectional lines L of diameter $\mathcal{D} > 1$ to every *odd* value K in $[2..2\mathcal{D}-1)$. The fact that a line is not a regular graph seems to help the stabilization in case of odd period. Thus, a natural candidate for showing the tightness of the bound for odd values of K is the class of bidirectional rings since rings are regular graphs where the diameter is maximum.

Table 3
Searching for cycles in unison: the experiment results.

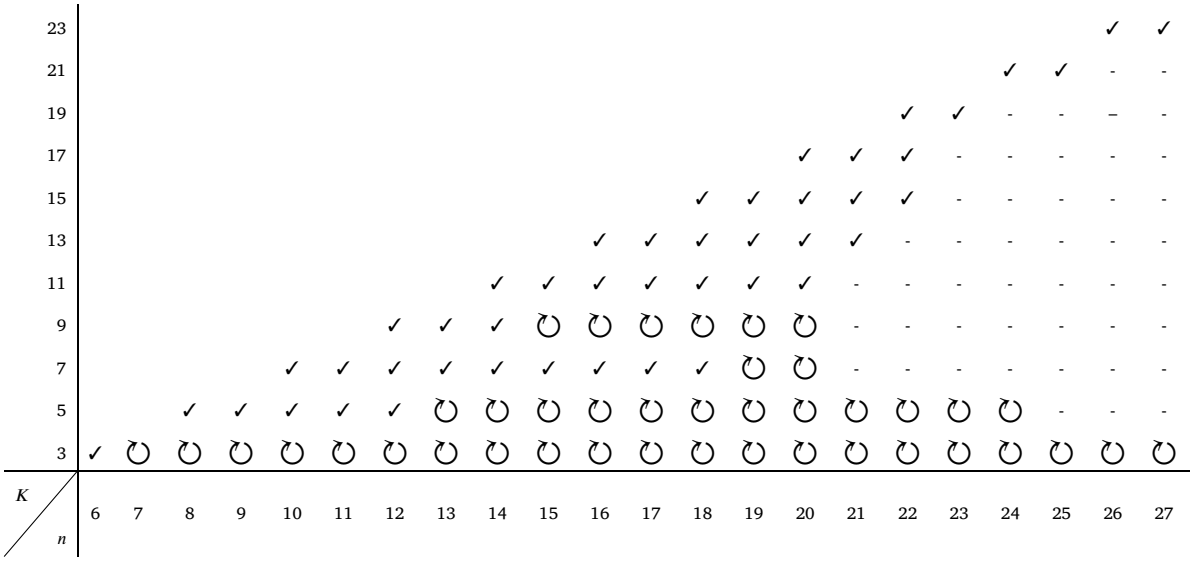

Table 3 shows the result of the experiment of executions in bidirectional rings. The topology size (n) is in abscissa, and the period (K) in ordinate. The diameter D of a bidirectional ring is $D = \lfloor \frac{n}{2} \rfloor$. Hence for rings of size smaller than 5, there is odd value for K in the considered interval, $[2..2D-1)$. For each pair of values (K, n), a loop sign (↻) indicates that a cycle has been found; a tick sign (✓) indicates that KIND2 has proven the convergence; and a minus sign (-) indicates that the verification process has been running for 2 weeks without returning any verdict.

The initial question that has motivated this experiment is the following: is there a n for which executions are non-stabilizing for every odd values of K in $[2..2D-1)$. The only case we were able to exhibit is for $n = 7$, but this case is particular since $[2..5)$ contains only one odd value: 3. For greater rings (up to 27 nodes), we could not find a size for which it was the case.

Furthermore, the results of this experiment can lead to another conjecture (which has not be proven yet): when a cycle is detected of a given pair (K, n), then for all $n' > n$, there is also a cycle at (K, n').

10. Integration into the SASA framework

10.1. The SASA framework

SASA is a tool dedicated to the simulation of distributed algorithms defined in the atomic state model [4]. Users specify the topology with DOT [15], and the guarded actions algorithm definitions (enable and step) in OCAML [29]. In addition to the basic simulation engine, several features are provided in the framework that rely on the Synchrone Reactive Toolbox [26]. The most important ones, summarized in Fig. 10, are the following.

- The usual daemons are provided: synchronous, central, locally central, and distributed. The SASA framework also provides a way for users to play the role of the daemon, where one can choose which of the enabled nodes one wants to activate, either via a graphical or via a command line interface. An API lets one program its own daemon. It is also possible to use a worst-case daemon which uses classical optimization heuristics (such as local search, branch and bound, Tabu list) to search for the longest stabilization time [24].
- Interactive simulation facilities are provided, where one can perform simulations step-by-step or round-by-round, moving forward or backward, while visualizing the configuration values on a graphical representation of the topology [22]. New navigation commands can be programmed [23].
- Generic scripts ease the execution of test campaigns that allow to study the asymptotic complexity of algorithms. Automated testing can be performed, where the test decision is done via (Lustre) oracles that formalize the algorithms expected properties, e.g., by providing stabilization-time upper bounds in terms of steps, rounds, or moves.
- DOT files containing classical graphs (rings, grids, trees, stars, cliques, and random using various heuristics) of arbitrary size can be generated.

10.2. SASA and SALUT

From the user point of view, the only difference between SASA and SALUT is that the former is based on OCAML, while the latter is based on LUSTRE. The main advantage of using LUSTRE is to enable Model-checking.

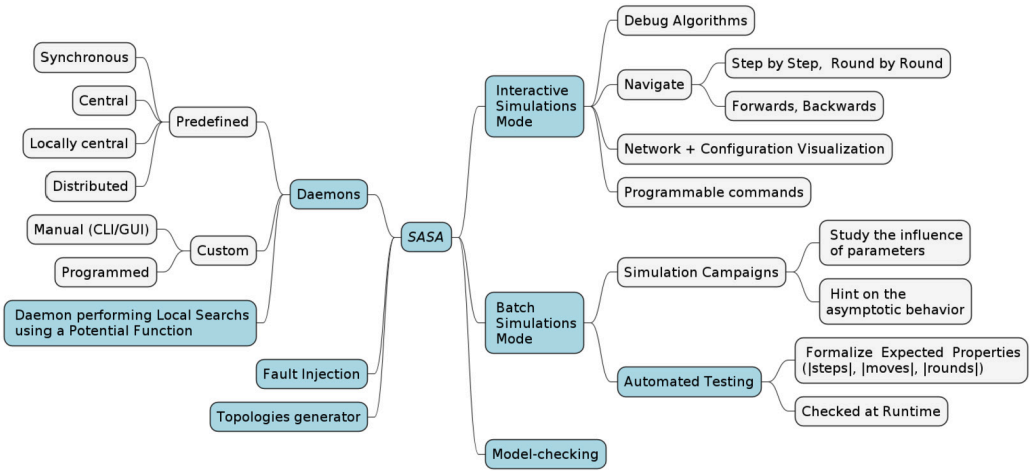

Fig. 10. The SASA framework.

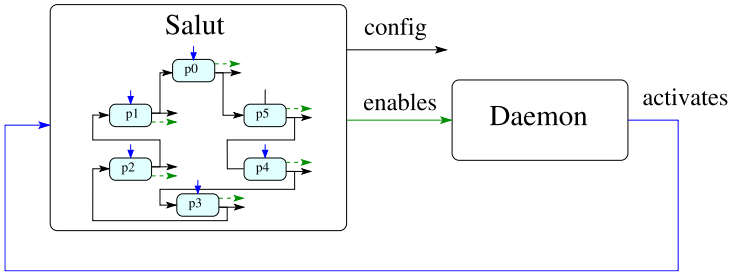

Fig. 11. Simulations dataflow with SALUT. The `enables` and `activates` wires hold arrays of Booleans; the `config` wire holds an array of (node state) values.

The SASA framework, which is made of tools to generate topologies, perform interactive simulations, or perform test campaigns, can be used in both cases.

10.3. Simulation with SALUT

Once the dot topology has been translated into LUSTRE as described in Section 6.5, we get a LUSTRE program that can be compiled and executed [26]. The resulting program outputs, at each step, the values for the current state values of processes, as well as Booleans stating for each process whether it is enabled or not. This information is used by the daemon to choose which processes should be active, as represented in the dataflow diagram of Fig. 11.

This is actually the exact same behavior and interface as the basic SASA simulation engine. For this reason, the plugging of this LUSTRE-based simulation engine onto the SASA framework was straightforward.

10.4. Simulation-based validation of SALUT

The SASA framework relies on the Synchrone Reactive toolbox, that contains in particular LURETTE [25], a tool dedicated to the functional testing of reactive programs. The program under test is executed in closed-loop (feedback) with its environment, under the supervision of an observer, that plays the role of the test oracle. The program expected properties are formalized, typically in LUSTRE, so that the test decision can be automated. At each step, the program under test inputs and outputs are provided to the oracle observer, which should always return true – otherwise, the test is considered failed.

The LURETTE infrastructure can be used to experimentally validate the SALUT translator, and make sure that it behaves as its SASA counterpart. This is described Fig. 12. The program under test is the SASA simulation engine, and its environment is the daemon. The test oracle is defined using the SALUT resulting LUSTRE program, to check that the configurations and the set of enabled processes are always equal, for every step.

An oracle violation means either that the LUSTRE and the OCAML encoding of ASM algorithm differs, or that the SASA simulation engine differs from the SALUT one. In theory, both could differ at the same time and compensate, but it is very unlikely.

We have run extensive tests using the 4 predefined daemons, the 7 algorithms of Table 1, and various topologies. The oracles were made of the same properties used for model-checking in Sections 7 and 8. This was particularly helpful during the algorithms LUSTRE encoding phase.

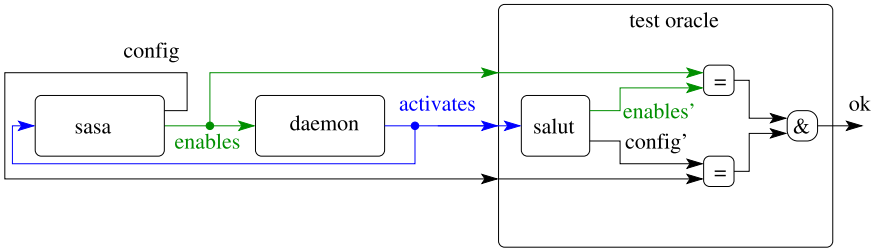

Fig. 12. Comparing SASA and SALUT simulations.

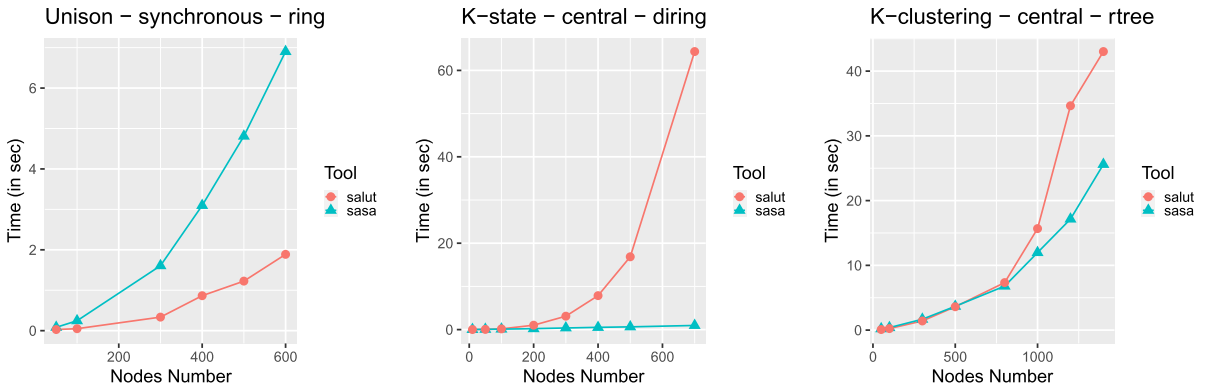

Fig. 13. Comparing SASA and SALUT simulation performances.

10.5. Simulation experimentation

We have performed some experiments to compare simulation times with SASA and SALUT. The conclusion we draw is that it depends on algorithms: SASA is faster on the majority of them, but not all. This is illustrated in Fig. 13 which contains a visualization of some experiments performed on three different algorithms: unison, K-state, and k-clustering. In the figure, the number of nodes is given in abscissa, and the number of seconds necessary to perform 1000 steps of simulation is given in ordinate. When a legitimate configuration is reached, a fault is injected to make sure that all the steps are performed in illegitimate configurations.

Note that the compilation time is not included in the time measurements of Fig. 13. For SASA, only the OCAML algorithm implementation needs to be compiled, which require, on the examples of Fig. 13, a few milliseconds. Everything else is done at simulation time.

For SALUT on the other hand, the DOT topology is translated first into a LUSTRE program, that is then compiled into a C program, finally used to generate an executable. For example, for the rightmost point of Fig. 13, that corresponds the k-clustering algorithm on a rooted-tree of 1400 nodes, the whole compilation process takes 6 minutes: in more detail, it took 1 minute for generating the 21122 lines LUSTRE file from the 2811 lines of the DOT file, 3 minutes for generating the 1 million lines C file, and 2 minutes for generating the 24 MB executable. This is actually almost 4 times longer than the corresponding 1000-steps simulation time.

In any case, this generated code size blow-up is definitely a limitation of SALUT for simulation – but it was not designed for that purpose.

10.6. Toolset outline

Fig. 14 summarizes the toolset presented in this article. SASA and SALUT allow to implement, in OCAML and in LUSTRE respectively, distributed algorithms defined in ASM. They use the same input format, DOT, for describing networks. They both permit to perform algorithms simulations. The advantage of SASA is to allow (generally faster) simulations on much bigger topologies, while the advantage of SALUT is that it allows the formal verification of algorithms by model-checking.

11. Conclusion

The main achievement of this article is to provide a way

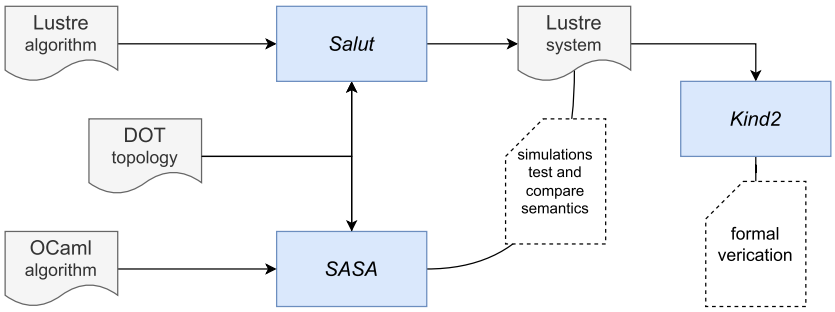

Fig. 14. The toolset Recap.

- to formalize ASM algorithms in a straightforward[9] way, where each aspect of an ASM algorithm (guard, action, expected properties) has a direct correspondence in the LUSTRE formalization;
- to perform automatic verification using model-checking out of this formalization.

This comes with an open-source framework that takes advantage of existing synchronous languages and model-checking tools. The encoding of the topology is automatically generated. Generic daemons are provided and cover the most commonly used cases (synchronous, distributed, central, and locally central). The article and its companion open-source repository contain many algorithms encoding examples, as well as examples of checkable properties including upper bounds on stabilization time that can be expressed in steps, moves, and rounds.

It is worth noting that SALUT has been developed as a natural extension of SASA [4], an open-source simulator dedicated to self-stabilizing algorithms defined in ASM. The integration of verification tools in the SASA suite is interesting from a technical and methodological point of view as it offers a unified interface for both simulating and verifying algorithms. In future works, it would be interesting to complete this suite with bridges to proof assistants to obtain, in the spirit of TLA+ [27], an exhaustive toolbox for computed-aided validation of self-stabilizing algorithms.

CRediT authorship contribution statement

Erwan Jahier: Writing – original draft, Software. **Karine Altisen:** Conceptualization. **Stéphane Devismes:** Writing – review & editing. **Gabriel B. Sant'Anna:** Software.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] The Sasa source code repository, https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/synchrone/sasa.
[2] K. Altisen, P. Corbineau, S. Devismes, Certification of an exact worst-case self-stabilization time, in: ICDCN, 2021, pp. 46–55.
[3] K. Altisen, S. Devismes, S. Dubois, F. Petit, Introduction to Distributed Self-Stabilizing Algorithms, Synthesis Lectures on Distributed Computing Theory, vol. 8, 2019.
[4] K. Altisen, S. Devismes, E. Jahier, Sasa: a simulator of self-stabilizing algorithms, Comput. J. 66 (4) (2023) 796–814.
[5] A. Arora, S. Dolev, M.G. Gouda, Maintaining digital clocks in step, Parallel Process. Lett. 1 (1991) 11–18.
[6] A. Casteigts, Jbotsim: a tool for fast prototyping of distributed algorithms in dynamic networks, in: SIMUtools, 2015, pp. 290–292.
[7] A. Champion, A. Mebsout, C. Sticksel, C. Tinelli, The kind 2 model checker, in: CAV, 2016.
[8] J. Chen, F. Abujarad, S.S. Kulkarni, Towards scalable model checking of self-stabilizing programs, J. Parallel Distrib. Comput. 73 (4) (2013) 400–410.
[9] J. Chen, S.S. Kulkarni, Smt-based model checking for stabilizing programs, in: ICDCN, 2013, pp. 393–407.
[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV version 2: an OpenSource tool for symbolic model checking, in: CAV, 2002.
[11] E.M. Clarke, E.A. Emerson, J. Sifakis, Model checking: algorithmic verification and debugging, Commun. ACM 52 (11) (2009) 74–84.
[12] A.K. Datta, S. Devismes, K. Heurtefeux, L.L. Larmore, Y. Rivierre, Competitive self-stabilizing k-clustering, Theor. Comput. Sci. 626 (2016) 110–133.
[13] E.W. Dijkstra, Self-stabilizing systems in spite of distributed control, Commun. ACM (1974).
[14] B. Dutertre, Yices 2.2, in: CAV, 2014.
[15] E.R. Gansner, S.C. North, An open graph visualization system and its applications to software engineering, Softw. Pract. Exp. 30 (11) (2000) 1203–1233.
[16] S. Ghosh, Binary self-stabilization in distributed systems, Inf. Process. Lett. 40 (3) (1991) 153–159.
[17] N. Halbwachs, S. Baghdadi, Synchronous modeling of asynchronous systems, in: EMSOFT'02, 2002.
[18] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, The synchronous data flow programming language Lustre, Proc. IEEE 79 (9) (1991) 1305–1320.

[9] During a 4-weeks internship, a first-year student has been able to learn LUSTRE, the SALUT framework, as well as encode and verify 3 of the 7 algorithms presented in this article.

[19] N. Halbwachs, F. Lagnier, P. Raymond, Synchronous observers and the verification of reactive systems, in: AMAST, 1993.
[20] J. Hoepman, Uniform deterministic self-stabilizing ring-orientation on odd-length rings, in: WDAG, 1994, pp. 265–279.
[21] G.J. Holzmann, D.A. Peled, An improvement in formal verification, in: IFIP, 1994.
[22] E. Jahier, Verimag tools tutorials: tutorials related to Sasa, https://www-verimag.imag.fr/vtt/tags/sasa/.
[23] E. Jahier, RDBG: a reactive programs extensible debugger, in: SCOPES, 2016.
[24] E. Jahier, K. Altisen, S. Devismes, Exploring worst cases of self-stabilizing algorithms using simulations, in: SSS, Jersey City, New Jersey, USA, October 2023.
[25] E. Jahier, S. Djoko-Djoko, C. Maiza, E. Lafont, Environment-model based testing of control systems: case studies, in: TACAS, 2014, pp. 636–650.
[26] E. Jahier, P. Raymond, The synchrone reactive toolbox, http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox.
[27] M.A. Kuppe, L. Lamport, D. Ricketts, The TLA+ toolbox, in: F-IDE@FM, 2019.
[28] Y. Lakhnech, M. Siegel, Deductive verification of stabilizing systems, in: WSS, 1997.
[29] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, J. Vouillon, The objective caml system, documentation and user's manual, https://caml.inria.fr/pub/docs/manual-ocaml/.
[30] L.C. Paulson, Natural deduction as higher-order resolution, J. Log. Program. 3 (1986) 237–258.
[31] M. Siegel, Formal verification of stabilizing systems, in: FTRTFT, 1998.
[32] T. Tsuchiya, S. Nagano, R.B. Paidi, T. Kikuno, Symbolic model checking for self-stabilizing algorithms, IEEE Trans. Parallel Distrib. Syst. 12 (1) (2001) 81–95.
[33] R.S. Whittlesey-Harris, M. Nesterenko, Fault-tolerance verification of the fluids and combustion facility of the international space station, in: ICDCS, 2006, p. 5.