Model Checking of Distributed Algorithms using Synchronous Programs

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

▶ **To cite this version:**

Erwan Jahier, Karine Altisen, Stéphane Devismes, Gabriel B. Sant'Anna. Model Checking of Distributed Algorithms using Synchronous Programs. 25th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2023), Shlomi Dolev, Baruch Schieber, Oct 2023, Jersey City, NJ, United States. hal-04172396

HAL Id: hal-04172396
https://cnrs.hal.science/hal-04172396

Submitted on 27 Jul 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Model Checking of Distributed Algorithms using Synchronous Programs*

Erwan Jahier[1], Karine Altisen[1], Stéphane Devismes[2], and Gabriel B. Sant'Anna[3]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG
[2] Université de Picardie Jules Verne, MIS, 80039 Amiens Cedex 1, France
[3] BRy Tecnologia, Brazil

Abstract. The development of trustworthy self-stabilizing 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 (and bounded liveness) properties of any algorithm, regardless of any assumptions on network topologies and execution scheduling.

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 progress is 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 the confidence in its soundness.

Simulation tools [3,4] are interesting to test and find flaws early in the design process. However, simulators only partially cover the set of possible executions and so are not suited to formally establish properties. In contrast, proof assistants [24] 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, proof assistants require a considerable amount of effort since they

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

often necessitate a full reengineering of the initial pencil-and-paper proof. Finally, and contrary to the two previous methods, model checking [9] allows a complete and fully automatic verification of the soundness of a distributed system for a given topology.

We consider model checking for self-stabilization, a versatile lightweight fault-tolerant paradigm [2,11]. A self-stabilizing algorithm makes the system eventually reach a so-called *legitimate* configuration from which every possible execution satisfies the intended specification, regardless of its configuration – the initial one, or any configuration resulting from a finite number of transient faults. 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 the ASM and computational models issued from formal methods based on synchronous programming languages [15], such as LUSTRE [16], to reuse their associated verification tools, in particular model checkers such as KIND2 [5]. This allows the automatic verification of all safety (and bounded liveness) properties of any algorithm, regardless the assumptions made on network topologies and the asynchrony of the execution model (daemons).

Contribution. We propose a language-based framework, named SALUT, to verify the self-stabilization of distributed algorithms written in ASM. In particular, we implement a translation from the network topology to a LUSTRE program, this latter calling upon an API designed to encode the algorithm. The verification then comes down to a state-space exploration problem performed by the model checker KIND2 [5]. Our proposal is modular and flexible thanks to a clear separation between the description of algorithms, daemons (which are also programmable), 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 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 [11], Ghosh's mutual exclusion [14], Hoepman's ring-orientation [18]) studied by the state-of-the-art, yet ad hoc, approaches [6,7,26] 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 scalability and verification time.

Related Work. Pioneer works on verification of distributed self-stabilizing algorithm have been led by Lakhnech and Siegel [23,25]. 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.

In 2001, Tsuchiya *et al.* [26] proposed to use the symbolic model checker NuSMV [8]. 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 specifi-

cations. Yet, their approach is not generic since it mixes in the same user-written SMV file the description of the algorithm, the expected property, and the the topology.

In 2006, Whittlesey-Harris and Nesterenko [27] modeled in SPIN [19] a specific yet practical self-stabilizing application, namely the fluids and combustion facility of the international space station, to automatically verify it. A few experimental results are given, but no analysis or comparison with [26] is given.

Chen *et al.* [6] focus on the bottlenecks, in particular related to fairness issues, involved by the verification of self-stabilizing algorithms. They also use the NuSMV [8] model checker. Chen and Kulkarni [7] use SMT solvers to verify stabilizing algorithms [12]. 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 [6,26]. Approaches in [6,7] 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.

SASA [3] is an open-source tool dedicated to the simulation of self-stabilizing algorithms in the 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 constrainted language (it targets critical systems) and has a less rich programming environnement. 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 the ASM and a theoretical model that grounds the synchronous programming paradigm. Section 4 proposes a general way of embedding the 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 the LUSTRE toolbox to perform automatic verifications in practice. Section 8 presents some experimentation results. We make concluding remarks in Section 9.

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 the ASM model [11], communications are abstracted away as follows: each process can read its variables and those of its neighbors or predecessors (depending on

Inputs: K, a positive integer $\geq n$; and $q.Pred$, the predecessor in the ring
Variables: $v \in \{0, ..., K-1\}$
Actions for non-root processes:
$T_p :: q.v \neq q.Pred.v \hookrightarrow q.v \leftarrow q.Pred.v$
Actions for the root:
$T_{root} :: q.v = q.Pred.v \hookrightarrow q.v \leftarrow (q.v+1)\ mod\ K$

Fig. 1: The Dijkstra's K-state algorithm for n-size rooted unidirectional rings [11].

whether or not communication links are bidirectionnal) and can only write to its own variables. The local algorithm of a 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 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 (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 the 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 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 such a daemon. Daemons are usually defined as the conjunction of their spreading and fairness properties [2]. 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. Every daemon we deal with in this paper is considered to be unfair, meaning that it might never select an enabled process unless it is the only remaining one.

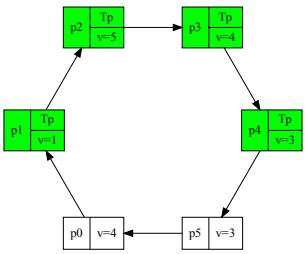

Fig. 2: Unidirectional ring of six processes rooted at *p0*.

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*), executes the same local algorithm. In the figure, each enabled process, given in color, 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 chose 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.

A distributed system is meant to execute under a set of assumptions, which are in particular related by the topology (in the above example, a rooted unidirectional ring) and the daemon (in the above example, the distributed 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: Correctness: every execution satisfying the assumptions and starting from a legitimate configuration satisfies the specification; Closure: every execution satisfying the assumptions and starting from a legitimate configuration only contains legitimate configurations; Convergence: every execution satisfying the assumptions eventually reaches a legitimate configuration.

3 The Synchronous Programming Model

We now briefly present the main concepts grounding the *synchronous programming paradigm* [15] that are 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 [15], a *synchronous node*[4] 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

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

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.

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

[4] 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.

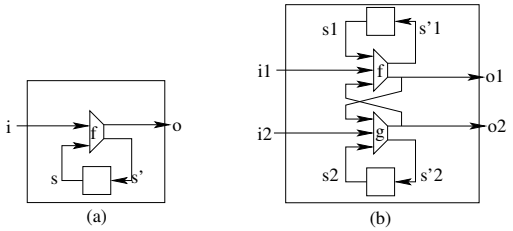

Fig. 3: General scheme of a synchronous node (a) and synchronous composition (b).

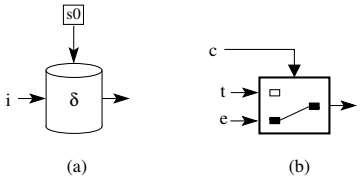

Fig. 4: Delay (a) and if-then-else (b) synchronous nodes.

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:

$$f_o^{\delta}(i,s) = s \qquad f_s^{\delta}(i,s) = i$$

The second node (see Fig. 4.b) is a stateless 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

The 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 broadcasted 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 – often called *oracles* in the programming language community. On the other hand, in the 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 the ASM guarded actions of the process (see Fig. 5): (1) `enable`, whose inputs are the states the process can read (the predecessors in the graph); this node has a single output, a Boolean array, which elements are true if and only if the corresponding processes guards are enabled; (2) `step`, with the same inputs as `enable`, and

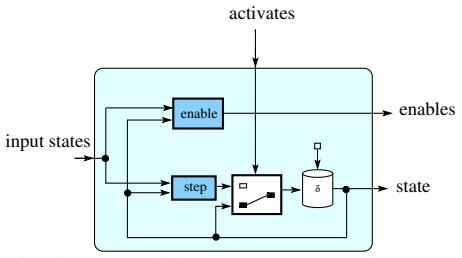

Fig. 5: Formalizing an ASM process as a synchronous node.

that outputs a new state (as computed by the statement of the enabled action); this state is used as the new value of the corresponding process state if the daemon chooses to activate the process; the previous value is used otherwise.

The communication links in the network topology are translated into data wires in the synchronous model. For each process, the `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 left-most node in Fig. 6.

5 ASM Algorithms Verification via Synchronous Observers

Once we have a formal model (made of synchronous nodes) of the process local algorithms and the network, it is possible to automatically verify some properties using so-called *synchronous observers* [17]: the desired properties can be expressed by the means of another synchronous node that observes the behavior of the outputs and returns a Boolean that states whether configurations sequences are correct.

Classically, the assumptions of the environment of the system under verification is also formalized by a synchronous observer; here, those assumptions are handled by the daemon, which decides which processes should be activated among the enabled ones. Therefore, the assumption observer is named daemon; it has $2 \times n$ input wires: n `activate` wires and n `enable` wires, one each per process; it outputs a Boolean whose value states whether the assumption made on the daemon (*e.g.*, synchronous, distributed, central) is satisfied. Those classical daemon assumptions, encoded as synchronous nodes, are provided as a library [1].

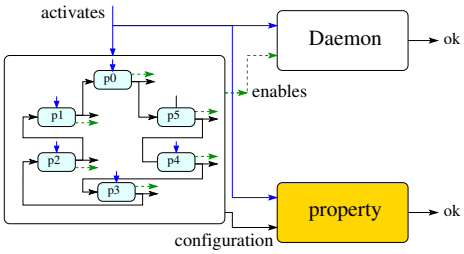

Fig. 6: Verifying a property using synchronous observers.

The verification of a given *property* then consists in checking that the synchronous composition of the synchronous nodes encoding the processes topology, the daemon observer, and the property observer never causes the latter to return false while the daemon observer has always returned true; this boils down to a state-space exploration problem. The 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 `enables` values are plugged into the `enables` 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.

In order to prove the closure property of the self-stabilization definition (Section. 2), which states that an algorithm never steps from a legitimate to an illegitimate config-

uration, one can use the observer of Fig. 7. It checks that if the previous configuration (computed by the δ node) was legitimate, then so is the current one.

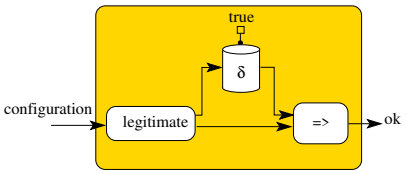

Fig. 7: The closure property Observer.

Similarly to the closure property, one can formalize classical convergence theorems, such as, "if $K \geq n$ and the daemon is distributed, then the stabilization time of the algorithm of Fig. 1 is at most $2n-3$ rounds" [5] (Theorem 6.37 of [2]). For some ASM algorithms, called *silent*, the legitimate configurations are the terminal ones, where no process is enabled. But for other algorithms, such as the one presented in Section 2, a definition of legitimacy needs to be provided.

Using synchronous observers, one can just specify *safety properties*, which state that nothing bad will happen. Liveness properties, such as "the algorithm will eventually converge", cannot be expressed. But stronger (and equally interesting) properties such as "the algorithm will converge in at most $f(n)$ steps" can. Moreover, observers can be executed and used during simulations to implement test oracles [21]: this allows to test the whole model at first hand, before verification.

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 (i) chose a format to describe the network, (ii) chose a language to implement synchronous nodes, (iii) propose an API for that language to define `enable` and `step` functions, and (iv) implement a translator from the format chosen in (i) to the language chosen in (ii).

Network description. We have chosen to base the network description on DOT [13]: 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.

LUSTRE, a language to implement synchronous nodes. LUSTRE is a dataflow synchronous programming language designed for the development and verification of critical reactive systems [16]. It combines the synchronous model – where the system reacts instantaneously to a flow of input events at a precise discrete time scale – with the dataflow paradigm – which is based on block diagrams, where blocks are parallel operators concurrently computing their own output and possibly maintaining some states. Choosing LUSTRE to implement the synchronous nodes of Section 3 is natural as they

[5] A *round* is a time unit that captures the execution time according to the speed of the slowest processes; see [2] for a formal definition.

were designed to model LUSTRE programs in the first place [17]. Moreover, two LUSTRE model checkers are freely available to perform formal verifications (*cf.* Section 7).

A LUSTRE API to define ASM algorithms. The LUSTRE API for SALUT follows the formalization of 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 left-most inner nodes of Fig. 5.

For Dijkstra's 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 1 and 2 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 1: The Lustre `enable` for the root process

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, the ones of its neighbors. 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 neighbors, in particular its state, accessed using the `state` getter (see Line 4 of Listing 1). 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 1), as stated in the guard of the root action in Fig. 1. In LUSTRE, stateless nodes are declared as `function` (Line 1 of Listing 1 and 2) 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 2: The Lustre `step` node for the root process

Step nodes have the same input parameters as enable ones, plus the active action label (see `a` in Listing 2, 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 [20] for the complete implementation).

The SALUT translator. All the nodes required to describe the ASM algorithms are then generated automatically from the network topology using a DOT to LUSTRE translator. The two nodes, `enable` and `step`, are the only LUSTRE programs that need to be provided. SALUT generates from the DOT file a node, called `topology` (the leftmost node in Fig. 6). In particular, SALUT takes care of wiring the `enable` and the `step` node instances to the right processed and the right values of the degree `d` parameter, that can vary from one node instance to another.

7 Automatic Formal Verification

We have seen that properties on ASM algorithms can be proven using synchronous observers (*cf.* Fig. 6). By defining such observers in LUSTRE, we can perform the verification of these properties automatically using existing verification tools for LUSTRE such as KIND2 [5]. Technically, to use such a tool, one just needs to point out a node Boolean variable. 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, as done in the `verify` node given in Listing 3. This section is devoted to the explanation of this listing.

```
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 since the algorithm has only 1 rule per process
9    enabled1: bool^n;   -- enabled projection
10   legitimate, round: bool;
11   closure, converge_cost, converge_wc: bool;
12   steps, cost, round_nb: int;
13 let
14   config, enabled, round, round_nb = topology(active, init_config);
15   assert(true -> daemon_is_central<<1,n>>(active, pre enabled));
16   enabled1 = map<<nary_or<<1>>,n>> (enabled); -- projection
17   legitimate = nary_xor<<n>>(enabled1);
18   closure = true -> (pre(legitimate) => legitimate);
19   cost = cost(enabled, config);
20   converge_cost = (true -> legitimate or pre(cost)>cost);
21   steps = 0 -> (pre(steps) + 1); -- 0, 1, 2, ...
22   converge_wc = (steps >= worst_case) => legitimate;
23   ok = closure and converge_cost and converge_wc;
24 tel;
```

Listing 3: LUSTRE formalization of some properties of the K-state algorithm.

This node is a particular instance of Fig. 6. The information related to process enabling (`enabled` variable in Listing 3) and activation (`active`) are contained into 2-dimensional Boolean arrays. The first dimension is used to deal with algorithms that are made of several guarded actions (here only one is used). The second dimension is used to deal with topologies that have more than one process.

According to the current configuration and an array of Booleans `active` indicating which processes have been activated by the daemon, the `topology` node computes a

new configuration `config`, which is an integer array of size n, and a matrix of Booleans `enabled` of size $1 \times n$ to indicate which processes are enabled (Line 14). The `topology` node also outputs elements relative to round computation, which are not used here. At the first step, the current configuration returned by Node `topology` is the initial one, i.e, its argument `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`, cf. Fig. 5) and the process activations. At every step, the set of enabled processes is computed according to the configuration. The verification tools will try to prove that, *e.g.*, `ok` is always true for all possible values of its inputs, namely for every initial configuration, and all process activation scheduling.

Daemon assumptions. As already mentioned, in order to fully encode the ASM algorithms, we need to express assumptions about the daemon; In LUSTRE, this can be done through the `assert` directive (Line 15). Here, a central daemon is used (the node `daemon_is_central` not shown here): it checks that, at each step, only enabled nodes can be activated, and that exactly one can be activated. Note that such a property is not checked at the first instant (`true->...`[6]) since `pre(enabled)`, which returns the previous value of `enabled`, is undefined at that instant.

Closure. The node `verify` should also define the properties involved in the definition of self-stabilization; see Section 2. The first expressed property is the closure: 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 17, which checks that exactly one process is enabled 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 in Line 18 and is a direct implementation of Fig. 7. Again, this property is not checked at the first instant when `pre(legitimate)` is undefined.

Convergence. We now focus on the convergence part. For algorithms with an available *potential function*, that quantifies how far a configuration is from the set of legitimate configurations, we can check whether this function is decreasing; see the Boolean `convergence_cost` and Lines 19-20 (the `cost` node is not shown here). The existence of a decreasing function guarantees the convergence. Note that once a legitimate configuration is reached, the potential function does not necessarily decrease anymore. This is the kind of subtleties that a verification tool can spot (and have actually spotted) easily.

Alternatively, we can take advantage of known upper bounds for the convergence time, being tight or not. In our case, we use Theorem 6.30 of [2] (Line 2). Then, we can check this bound, and so the convergence, by stating that once the upper bound is reached, the configuration should be legitimate; see the variable `steps` that counts the number of steps elapsed since the beginning of the execution and the Boolean `convergence_wc` that checks the property (Lines 21-22).

[6] 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.

8 Experimentations

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 [6,7,26] where they are mixed into a single user-written SMV [8] or Yices [12] 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-dimension arrays). Other daemons can be modeled similarly.
3. To model-check an algorithm, one thus just need to model its guarded actions, using the API described in Section 6. Actually, we have done it for several different algorithms: the Dijkstra's K-state algorithm [11] whose LUSTRE encoding is made of 37 lines of code (loc), the Ghosh's mutual exclusion [14] (50 loc), a Bread-First Search spanning tree construction [2] (80 loc), a synchronous unison [2] (40 loc); a k-clustering (with k=2) algorithm [10] (130 loc), a vertex-coloring algorithm [2] (30 loc), and the Hoepman's ring orientation [18] (110 loc).
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,[7] 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 results for a small subset of them.

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

Algorithm	LUSTRE prog size for WC-conv (ϕ-conv) in loc	Topology	Daemon	Max topo size for WC-conv (ϕ-conv) in processes nb
K-state	7 (86)	rooted unidirectional ring	synchronous central distributed	48 (15) 6 (8) 6 (8)
Ghosh	8	"ring-like"	central distributed	18 16
Coloring	6 (8)	ring random	central	10 (55) 11 (26)
Sync unison	6	random ring	synchronous	40 17
BFS sp. tree	7	tree	distributed	8
k-clustering	30 (20)	rooted tree	distributed	9 (10)
Hoepman	6	odd-size ring	central	7

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

Table 1 summarizes experiments made with the KIND2 [5] model checker[8] to prove some properties against the corresponding algorithm encoding. Columns 1, 3, and 4 respectively contain the algorithm name, the topology,[9] and the daemon used for the experiment. Column 2 contains the number of lines of LUSTRE code used to encode the worst-case-based convergence property, apart from the main node declarations (and in parentheses the number of lines to encode the potential function property, when available). Column 5 contains the maximal number of processes for which we get a (positive) result in less than 1 hour[10] (and ditto for the potential-based convergence in parentheses).

The topology sizes we can handle are quite 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; indeed, 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 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.

Performance comparison. Table 2 reports the time (in seconds) necessary to check the convergence of the K-state algorithm under a central daemon, using different topology sizes, different solvers, and different problem encodings. We note "-" when the timeout of one hour is reached. All experiments were conducted on the same computer, except for the second column. Indeed, the exact encoding was not provided in the article, so we simply report the number from Table 3 of [7]. Column 3 shows the result of the proposed framework. Column 4 of Table 2 shows the result of a NuSMV program that was not automatically generated by SALUT, but that mimics the corresponding generated Lustre code (discussed below). Columns 5 and 6 show the result of a direct encoding of the problem in NuSMV as described in [26] and [6], respectively.

On this algorithm (and on the Ghosh's algorithm), the encoding performed using the BDD-based solver NuSMV give better performances. Therefore, this allows to handle topologies with a little bit more nodes. However:

- it seems unlikely that a problem occuring on topologies of size 10 can never occur on ones of size 6;
- nothing guarantees that it would be the case for all algorithms; and
- the BDD-encoding is limited to finite domains.

Moreover, using our proposal to target (for example) NuSMV should be not too difficult. Indeed, once completely expanded (using the `-ec` of the Lustre V6 compiler), the Lustre program provided to KIND2 is actually very close to a NuSMV program. In order

[8] We use kind2 v1.9.0 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 $n > 13$, $WC > 256$).

[9] Random graphs were generated using the Erdős–Rényi model.

[10] 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.

Table 2: Measuring the time to prove the convergence of the K-state algorithm.

topology size	Yices (SMT) encoding of [7]	SALUT+KIND2 (SMT)	"SALUT"+NuSMV (BDD)	NuSMV (BDD) encoding of [6]	NuSMV (BDD) encoding of [26]
5	200 (from [7])	6	0	0	0
6	-	190	0	0	0
7		-	2	0	1
8			10	3	4
9			60	80	10
10			600	20	20
11			-	650	40
12				2800	180
13				-	2500

to try that idea out, we wrote a NuSMV program that mimics the Lustre code coming from the Dijkstra's K-state (and measured its performance in Column 4 of Table 2).

As far as SMT-based verification is concerned, the proposed framework used with the KIND2 model checker (which delegates the solving part to external SMT solvers) does not seem to pay the price of genericity, as we get performances that have the same order of magnitude. Indeed, for the K-state and the Ghosh algorithms convergence under a central daemon, and using a timeout of one hour as we did in Table 1, Chen *et al.* [7] 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 Conclusion

This work presents an open-source framework that takes advantage of synchronous languages and model-checking tools to formally verify self-stabilizing algorithms. 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). One just needs to formalize (in Lustre) the ASM actions and the properties to verify.[11] The article and its companion open-source repository contain many algorithm encoding examples, as well as examples of checkable properties including stabilization time upper bounds that can be expressed using steps, moves, or rounds.

It is worth noting that SALUT has been developed as a natural extension of SASA [3], an open-source simulator dedicated to self-stabilizing algorithms defined in the 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+ [22], an exhaustive toolbox for computed-aided validation of self-stabilizing algorithms.

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

References

1. The SASA source code repository. https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/synchrone/sasa.
2. K. Altisen, S. Devismes, S. Dubois, and F. Petit. *Introduction to Distributed Self-Stabilizing Algorithms*, volume 8 of *Synthesis Lectures on Distributed Computing Theory*. 2019.
3. K. Altisen, S. Devismes, and E. Jahier. sasa: a simulator of self-stabilizing algorithms. *Comput. J.*, 66(4):796–814, 2023.
4. A. Casteigts. Jbotsim: a tool for fast prototyping of distributed algorithms in dynamic networks. In *SIMUtools*, pages 290–292, 2015.
5. A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli. The kind 2 model checker. In *CAV*, 2016.
6. J. Chen, F. Abujarad, and S. S. Kulkarni. Towards scalable model checking of self-stabilizing programs. *J. Parallel Distributed Comput.*, 73(4):400–410, 2013.
7. J. Chen and S. S. Kulkarni. Smt-based model checking for stabilizing programs' . In *ICDCN*, pages 393–407, 2013.
8. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, 2002.
9. E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.
10. A. K. Datta, S. Devismes, K. Heurtefeux, L. L. Larmore, and Y. Rivierre. Competitive self-stabilizing k-clustering. *Theor. Comput. Sci.*, 626:110–133, 2016.
11. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 1974.
12. B. Dutertre. Yices 2.2. In *CAV*, 2014.
13. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30(11):1203–1233, 2000.
14. S. Ghosh. Binary self-stabilization in distributed systems. *IPL*, 40(3):153–159, 1991.
15. N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*, 2002.
16. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
17. N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *AMAST*, 1993.
18. J. Hoepman. Uniform deterministic self-stabilizing ring-orientation on odd-length rings. In *WDAG*, pages 265–279, 1994.
19. G. J. Holzmann and D. A. Peled. An improvement in formal verification. In *IFIP*, 1994.
20. E. Jahier. Verimag Tools Tutorials: Tutorials related to SASA. https://www-verimag.imag.fr/vtt/tags/sasa/.
21. E. Jahier, S. Djoko-Djoko, C. Maiza, and E. Lafont. Environment-model based testing of control systems: Case studies. In *TACAS*, pages 636–650, 2014.
22. M. A. Kuppe, L. Lamport, and D. Ricketts. The TLA+ toolbox. In *F-IDE@FM*, 2019.
23. Y. Lakhnech and M. Siegel. Deductive verification of stabilizing systems. In *WSS*, 1997.
24. L. C. Paulson. Natural deduction as higher-order resolution. *J. Log. Prog.*, 3:237–258, 1986.
25. M. Siegel. Formal verification of stabilizing systems. In *FTRTFT*, 1998.
26. T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE TPDS*, 12(1):81–95, 2001.
27. R. S. Whittlesey-Harris and M. Nesterenko. Fault-tolerance verification of the fluids and combustion facility of the international space station. In *ICDCS*, page 5, 2006.