# SASA: a SimulAtor of Self-stabilizing Algorithms

Karine Altisen, Stéphane Devismes, Erwan Jahier

# SASA: a SimulAtor of Self-stabilizing Algorithms

Karine Altisen Stéphane Devismes Erwan Jahier

Univ. Grenoble Alpes, CNRS, Grenoble INP
VERIMAG, 38000 Grenoble, France
March 23, 2022

## Abstract

In this paper, we present SASA, an open-source SimulAtor of Self-stabilizing Algorithms. Self-stabilization defines the ability of a distributed algorithm to recover after transient failures. SASA is implemented as a faithful representation of the atomic-state model (also called the locally shared memory model with composite atomicity). This model is the most commonly used one in the self-stabilizing area to prove both the correct operation of self-stabilizing algorithms and complexity bounds on them. SASA encompasses all features necessary to debug, test, and analyze self-stabilizing algorithms. All these facilities are programmable to enable users to accommodate to their particular needs. For example, asynchrony is modeled by programmable stochastic daemons playing the role of input sequence generators. Properties of algorithms can be checked using formal test oracles. The SASA distribution also provides several facilities to easily achieve (batch-mode) simulation campaigns. We show that the lightweight design of SASA allows to efficiently perform huge such campaigns. Following a modular approach, we have aimed at relying as much as possible the design of SASA on existing tools, including OCAML, DOT, and several tools developed in the *Synchrone Group* of the VERIMAG laboratory.

## 1 Introduction

Starting from an arbitrary configuration, a self-stabilizing algorithm [1] makes a distributed system eventually reach a so-called *legitimate* configuration from which every possible execution suffix satisfies the intended specification. Self-stabilization is defined in the reference book by Dolev [2] as a conjunction of two properties: *convergence*, which requires every execution of the algorithm to eventually reach a legitimate configuration; and *correctness*, which requires every execution starting from a legitimate configuration to satisfy the 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.

[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.

Remark that the definition of self-stabilization does not directly refer to the possibility of (transient) faults. Actually, this is mainly due to the fact that, in contrast to most of existing fault tolerance (*a.k.a.* robust) proposals, self-stabilization is a *non-masking* approach: it does not try to hide effects of faults, but rather aims at repairing the system after faults [3]. So, proving or simulating a self-stabilizing system does not involve any failure pattern: only the consequences of faults, modeled by the arbitrary initial configuration, are treated. In other words, the actual convergence of the system is guaranteed only if there is a sufficiently large time window without any fault, which is indeed the case when faults are transient.

Self-stabilizing algorithms are mainly compared according to their *stabilization time*, *i.e.*, the maximum time, starting from an arbitrary configuration, before reaching a legitimate configuration. By definition, the stabilization time is impacted by worst case scenarios which are often unlikely in practice. So, in many cases, the average-case time complexity may be a more accurate measure of performance assuming a probabilistic model. However, the arbitrary initialization, the asynchronism, the arbitrary network topology, and the algorithm design itself often make the probabilistic analysis intractable. In contrast, another popular approach consists in empirically evaluating the average-case time complexity via simulations. A simulation tool is also of prime interest to test and find flaws early in the design process. Indeed, in the distributed computing area, correctness of distributed algorithms is often subtle. Asynchronous distributed systems often involve numerous autonomous loosely interconnected processes (referred to as nodes in the following); and interleaving between their executions is decided by a nondeterministic adversary, which models the unpredictable timing behavior of the network on which the system is deployed. In such a context, a good software engineering practice consists in conducting an extensive simulation campaign, including corner cases (generated using well-chosen scheduling and particular topologies) to increase the confidence on the correctness of the distributed algorithm before starting to formally prove it.

We are interested here in simulating the *atomic-state model* (ASM), because it is the most commonly used computational model in the self-stabilizing area. The ASM is a locally shared memory model which abstracts away the communication between nodes: in this model, each node can directly read the local states of its neighbors in the network. As a consequence, the ASM is quite simple to represent since there is no communication channel and configurations of the system are merely vectors of node states. Moreover, execution steps are atomic transitions between configurations.

**Contribution.** We provide to the self-stabilizing community an open-source, versatile, lightweight (in terms of memory footprint), and efficient (in terms of simulation time) simulator, called SASA, to help the design and evaluate the average performances of self-stabilizing distributed algorithms written in the *atomic-state model* (ASM). SASA is a straightforward implementation of the ASM and so obviously takes advantage of the inherent simplicity of this model (*e.g.*, SASA does not require a queue of events) to be efficient both in terms of simulation time and memory footprint.

The SASA programming interface is simple, yet rich enough to allow a direct encoding of any distributed algorithm described in the ASM. All important concepts used in this model are available: simulations can be run and evaluated in moves, atomic steps, and rounds; the three main time units used in the ASM. Classical execution schedulers, *a.k.a.* daemons, are available: central, locally central, distributed, and synchronous daemons. All levels of anonymity are available, such as fully anonymous, rooted, or identified. Finally, distributed algorithms can be either uniform (all nodes execute the same local algorithm), or non-uniform.

SASA can perform batch simulations which can use test oracles to check expected properties. For example, one can check that the stabilization time in rounds is upper bounded by a given function. The distribution provides several facilities to achieve batch-mode simulation campaigns. Simulations can also be run interactively, step by step, for debugging purposes.

During the simulator development, a constant guideline has been to take, as much as possible, advantage of existing tools. SASA relies on OCAML to program the self-stabilizing algorithms, DOT [4] to define topologies, and the Synchrone Reactive Toolbox [5] for testing using formal oracles and debugging. Another guideline has been to make all SASA's facilities easily configurable and programmable so that users can define specific features tailored for their particular needs.

To validate our approach, we have encoded various benchmark algorithms into SASA, *e.g.*, the Dijkstra token ring algorithm [1], a Breadth-First Search (BFS) [3] and a Depth-First Search (DFS) [6] spanning tree construction, vertex-coloring algorithms [3, 7, 8], a synchronous [9] and an asynchronous [10] unison. For example, we can execute 1 000 steps of the heavy DFS spanning tree construction of Collin and Dolev [6], whose memory requirement is $\Theta(N \log(\Delta))$ bits per node,[2] on a grid of 10 000 nodes in 47 minutes and using 381 megabytes of memory only. Moreover, we show that the lightweight design of SASA allows to efficiently perform huge simulation campaigns by comparing the average performances of three vertex-coloring algorithms as a case study.

[2] $N$ is an upper bound on the number of nodes and $\Delta$ is the maximal degree of the network.

**Related work.** Until now, only a few studies deal with the empirical evaluation of the average performances of self-stabilizing algorithms, *e.g.*, [11, 12]. Moreover, these experiments are done, most of the time, using homemade engines that propose only few features and whose design is tailored to specific needs.

Networking simulators, such as WSNET [13], OMNeT [14], or NS2 [15], are not suited for self-stabilizing algorithms. They usually measure performances in terms of simulation time which does not correspond to any unit of time (such as rounds) used by the self-stabilizing community. Consequently, simulation times cannot be fairly compared to analytical worst case upper bounds. Moreover, they are usually dedicated to particular architectures, such as IP networks or wireless networks, that are far away from the architecture-independent models in which self-stabilizing algorithms are written. Consequently, proposed algorithms should be adapted to fit the architecture targeted by the simulator [16]. Again, such modifications do not allow fair comparison with analytical bounds. Finally, networking simulators are usually implemented using queues of events: at each step of simulation, an event is dequeued and induces some computations in the system which may lead to the generation of new events to be enqueued. SASA is not based on such a heavy mechanism: its semantics is defined as a sequence of global steps, from one configuration to another. This implementation choice makes SASA lightweight in terms of memory footprint and computation time, allowing then to simulate algorithms on large networks.

Only few simulators dedicated to self-stabilization in locally shared memory models, such as the ASM, have been proposed [17, 18, 19, 20]. Overall, they all have limited capabilities and features, and are not extensible since not programmable. Using these simulators, only few pre-defined properties, such as convergence, can be checked on the fly.

In more detail, Flatebo and Datta [17] propose a simulator of the ASM to evaluate leader election, mutual exclusion, and $\ell$-exclusion algorithms on restricted topologies, mainly rings. This simulator is not available anymore. It proposes limited facilities including classical daemons and evaluation of stabilization time in moves only.

Müllner *et al.* [19] present a simulator (written in Erlang) of the register model, a computational model which is close to the ASM. This simulator does not allow to evaluate stabilization time. Actually, it focuses on three fault tolerance measures initially devoted to masking fault-tolerant systems (namely, reliability, instantaneous availability, and limiting availability [21]) to evaluate them on self-stabilizing systems. However, these measures are still uncommon today in analyses of self-stabilizing algorithms. Memory footprints given in [19] underline the lightweight nature of SASA. As an illustrative example, Müllner *et al.* simulate the same spanning tree constructions as we do: while they need up to 1 gigabyte of memory for simulating a 256-node random network, SASA only need up to 235 megabytes for executing the same algorithms in the same settings.

The simulator (written in Java) proposed by Har-Tal [18] allows to run self-stabilizing algorithms in the register model on small networks (around 10 nodes). It only proposes a small amount of facilities, *i.e.*, the execution scheduling is either synchronous, or controlled step by step by the user. Only the legitimacy of the current configuration can be tested. Finally, it provides neither batch mode, nor debugging tools.

Evcimen *et al.* describe in [20] a simulation engine (written in C#) for self-stabilizing algorithms in message passing. Their simulator uses heavy mechanisms to implement this model, such as queue of events, threads, and fault injection. By contrast, the fact that SASA targets the ASM allows its implementation to be lighter. Then, in the Evcimen *et al.*'s simulator, the execution scheduler can be only fully asynchronous. Now, being corner cases, central and synchronous executions are very useful to find bugs or to exhibit a worst-case scenario. Moreover, using this simulator, performances can be only measured in terms of simulation time, which does not correspond to the time units used by the self-stabilizing community (such as rounds). Finally, the simulator cannot be executed in batch mode and can only detect whether a legitimate configuration is reached.

A preliminary version of this paper has been published in a conference [22]. Since then, the SASA distribution has been further refined and enriched with many new facilities, including, in particular, tools for automating simulation campaigns.

**Roadmap.** Section 2 is a digest of the ASM, illustrated with a running example. Section 3 presents the SASA simulator. Section 4 explains the connection between SASA and tools from the *Synchrone Reactive Toolbox*. Experimental results obtained with SASA are presented in Section 5. In Section 6, we propose an example of simulation campaign whose aim is to compare the average stabilization time of three vertex-coloring algorithms. Finally, we conclude in Section 7 with future work.

**Online material.** The tool is fully documented online [23] with a technical report, how-tos, videos, and tutorials. We also provide a link to an open-access git repository which contains the necessary material to reproduce the results given in this article; see [24]. In particular, it contains instructions to install the necessary tools, to replay the interactive session described in Section 2, to generate the data in Table 1 of Section 5, and to generate Figures 4, 5, and 6 of Section 6.

## 2 An Example: Asynchronous Unison in the Atomic-State Model

We present the atomic-state model (ASM) using the *asynchronous unison* algorithm given in [10] as a running example. The formal code of this algorithm is presented in Algorithm 1. The asynchronous unison is a clock synchronization problem which requires the difference between clocks of every two neighbors to be at most one increment at each instant.

**Distributed algorithms.** A distributed algorithm consists of a collection of local algorithms, one per node. The local algorithm of each node $p$ (see Algorithm 1) is made of a finite set of *variables* and a finite set of *actions* (written as guarded commands) to update them.

Some of the variables, like $\mathcal{N}_p$ in Algorithm 1, may be constant inputs in which case their values are predefined. Actually, here, $\mathcal{N}_p$ represents the local view of the topology at each node $p$: $\mathcal{N}_p$ is the set of $p$'s neighbors in the network. Algorithm 1 assumes the network is connected and undirected, so $q \in \mathcal{N}_p$ if and only if $p \in \mathcal{N}_q$. Then, each node holds a single writable variable, noted $p.c$ and called its local *clock*. Each clock $p.c$ is an integer with range 0 to $K-1$, where $K > n^2$ is a parameter common to all nodes, and $n$ denotes the number of nodes. Communication is carried out by read and write operations on variables: a node can read its variables and those of its neighbors, but can write only to its own variables. For example, in Algorithm 1, each node $p$ can read the value of $p.c$ and that of $q.c$, for every $q \in \mathcal{N}_p$, but can only write to $p.c$. The *state* of a node is defined by the values of its variables. A *configuration* is a vector consisting of the states of each node.

Each action is of the following form: $\langle label \rangle :: \langle guard \rangle \hookrightarrow \langle statement \rangle$. *Labels* are only used to identify actions. A *guard* is a Boolean predicate involving the variables of the node and those of its neighbors. The *statement* is a sequence of assignments on the node's variables. An action can be executed only if its guard evaluates to *true*, in which case the action is said to be *enabled*. More generally, a node is enabled if at least one of its actions is enabled. In Algorithm 1, we have two (locally mutually exclusive) actions per node $p$, $I(p)$ and $R(p)$.

**Steps and executions.** Nodes run their local algorithm by *atomically* executing actions. Asynchronism is modeled by a *nondeterministic* adversary called *daemon*. An execution is a sequence of configurations, where the system moves from a configuration to another as follows. Assume the current configuration is $\gamma$. If no node is enabled in $\gamma$, the execution is done. Otherwise, the daemon *activates* a non-empty subset $S$ of nodes that are enabled in $\gamma$; then every node in $S$ *atomically* executes one of its action enabled in $\gamma$, leading the system to a new configuration $\gamma'$, and so on. The transition from $\gamma$ to $\gamma'$ is called a *step*. Usual daemons include:

1. the *synchronous* daemon which activates every enabled node at each step,
2. the *central* daemon which activates exactly one enabled node at each step,
3. the *locally central* daemon which never activates two neighbors at the same step, and
4. the *distributed* daemon which activates at least one node at each step.

**Self-stabilization of Algorithm 1.** Recall that Algorithm 1 is an asynchronous unison algorithm where each node has a

**Algorithm 1** Asynchronous unison: local algorithm for each node $p$

```
Constant Input: $\mathcal{N}_p$, the set of $p$'s neighbors

Variable: $p.c \in \{0,...,K-1\}$, where $K > n^2$, and $n$ is the number of nodes
Predicate: $behind(a,b) = ((b.c - a.c) \bmod K) \leq n$
Actions: $I(p)$ :: $\forall q \in \mathcal{N}_p, behind(p,q)$                                   $\hookrightarrow$ $p.c \leftarrow (p.c+1) \bmod K$
         $R(p)$ :: $p.c \neq 0 \wedge (\exists q \in \mathcal{N}_p, \neg behind(p,q) \wedge \neg behind(q,p))$   $\hookrightarrow$ $p.c \leftarrow 0$
```

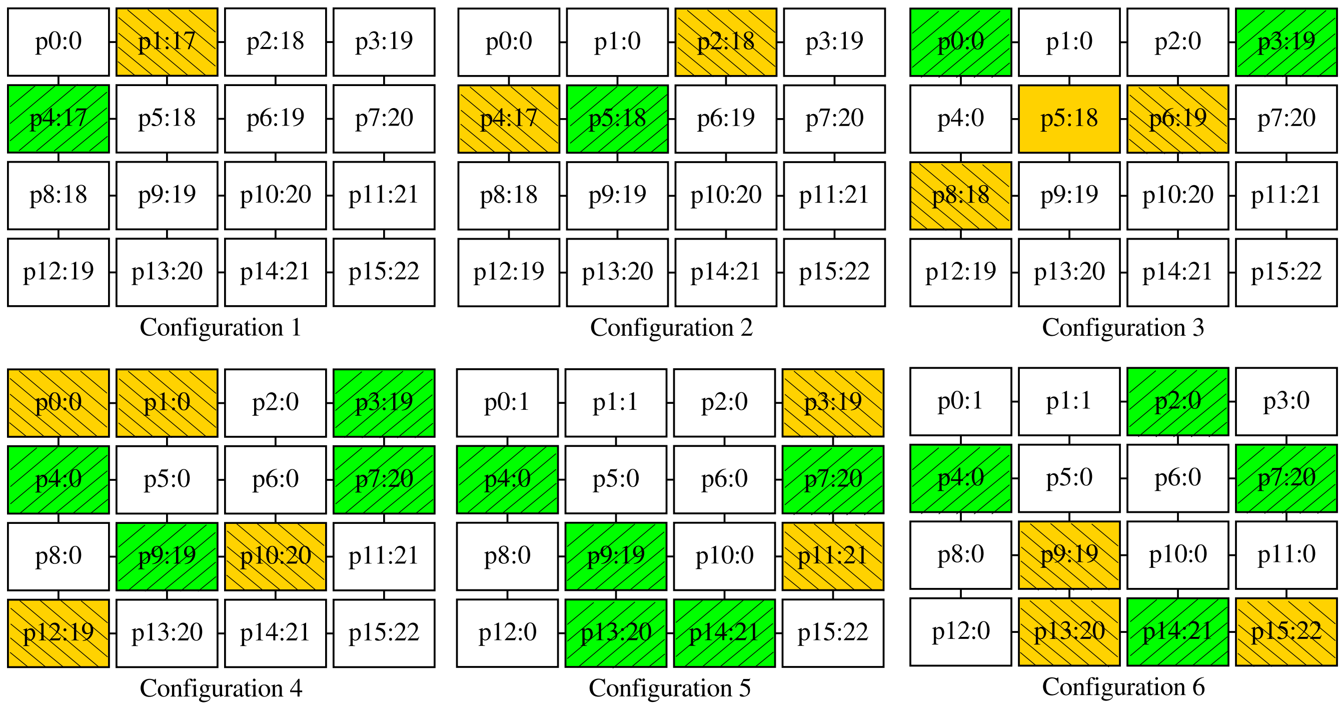

Figure 1: Five steps of Algorithm 1 for $K = 257$ on a 16-node grid (extracted from SASA). "$pi : j$" means that $pi.c = j$. Enabled nodes are in orange and green. Moreover, orange nodes are activated within the next step.

*clock* $p.c$ taking values in the range 0 to $K-1$, $K$ being a global parameter satisfying $K > n^2$. The asynchronous unison specification requires that

1. in each configuration, the difference between clocks of every two neighbors is at most one modulo $K$ (*Safety*); and

2. each clock is incremented infinitely often (*Liveness*).

In [10], Algorithm 1 is proven to be self-stabilizing under the distributed daemon in the sense that starting from an arbitrary configuration, every asynchronous execution of Algorithm 1 eventually reaches a legitimate configuration from which every possible execution suffix satisfies the above asynchronous unison specification. The legitimate configurations of Algorithm 1 are merely those where for any two neighbors $p$ and $q$, we have $p.c \in \{(q.c-1) \bmod K, q.c, (q.c+1) \bmod K\}$.

In Algorithm 1, each node $p$ maintains its clock $p.c$ using two actions, $I(p)$ and $R(p)$. $I(p)$ is the normal incrementation action. From a legitimate configuration, only $I$ actions are executed. $R(p)$ is a reset action whose goal is to correct local strong inconsistencies, *i.e*, a clock is reset when the gap between its value and that of a neighboring clock is too high.

We now describe these two actions using the sample of execution given in Figure 1.

When the clock of $p$ is at most $n$ increments behind the clock values of all its neighbors, $p$ is enabled to (normally) increment its clock modulo $K$ by Action $I(p)$. For example, in Configuration 5 of Figure 1, $I(p4)$ is enabled because $p4.c = 0$ and the clock values of its neighbors ($p0$, $p5$, and $p8$) belong to $\{0, 1\}$.

In contrast, if the clock of $p$ is not equal to 0 and $p$ has a neighbor $q$ such that $p.c$ is more than $n$ increments behind $q.c$ and $q.c$ more than $n$ increments behind $p.c$, then $p$ should reset its clock to 0 using Action $R(p)$. For example, in Configuration 1 of Figure 1, $p1.c = 17$ and $p0.c = 0$. Now, $p1.c$ (resp., $p0.c$) should be incremented 240 times (resp., 17 times) to reach 0 (resp., 17) since $K = 257$. As $n = 16$, $\neg behind(p1, p0) \wedge \neg behind(p0, p1)$ holds in Configuration 1. Moreover, $p1.c \neq 0$, so $R(p1)$ is enabled. After this reset, the next time $p$ will move, it will normally increment by $I(p)$, *i.e.*, its clock will be at most $n$ increments behind the clock values of all its neighbors. To ensure this eventually happens, resets might be propagated from neighbors to neighbors. Actually, thanks to resets, the system eventually reaches a configuration from which the difference between clocks of every two neighbors is at most $n$ increments modulo $K$. From such a configuration, no clock is further reset and the system inherently converges (using $I$ actions) to a legitimate configuration from which the asynchronous unison specification holds.

**Time complexity units.** Three main units are used for counting time: *steps, moves*, and *rounds*. Steps simply refer to atomic steps of the execution. A node *moves* when it executes an action in a step. Hence, several moves may occur in a step. Rounds capture the execution time according to the speed of the slowest node. The first round of an execution $e$ is the minimal prefix $e'$ of $e$ during which every node that is enabled in the first configuration of $e$ either executes an action or becomes disabled (due to some neighbor actions) at least once. The second round of $e$ starts from the last configuration of $e'$, and so on.

We now illustrate these notions using the sample of execution given in Figure 1. The second step (from Configuration 2 to 3) contains two moves (by $p2$ and $p4$). The first round ends in Configuration 3. Indeed, there are two enabled nodes, $p1$ and $p4$ in Configuration 1. Node $p1$ moves in the first step, however in Configuration 2, the round is not done since $p4$ has neither moved nor become disabled. The first round terminates after $p4$ has moved in the second step. Consequently, the second round starts in Configuration 3. This latter terminates in Configuration 6.

The stabilization time of an execution is the number of time units (steps, moves, or rounds) before reaching the first legitimate configuration. By extension, the stabilization time of an algorithm is the maximum stabilization time over all possible executions. As a matter of fact, the stabilization time of Algorithm 1 has been shown to be in $O(n\mathscr{D})$ rounds, where $n$ is the number of nodes and $\mathscr{D}$ is the diameter of the network [25].

# 3 The SASA Simulator

An important goal of this section is to convince the future end-users of SASA, mainly researchers from the self-stabilizing community, that (1) SASA offers all features they need (including classical measures of the model, interactive graphical step-by-step simulation, batch simulation), and (2) encoding their algorithms into SASA is straightforward since very close to usual way they write algorithms in the ASM. Therefore, in the following, we present the tool from an end-user point of view.

## 3.1 Main Features

This section outlines SASA main features. More information is available online as SASA tutorials [26].

**Batch simulations.** They are useful to perform simulation campaigns, to evaluate time complexity of algorithms on wide families of networks. They can also be used to study the influence of some parameters.

**Interactive graphical simulations.** It is possible to run a simulation step by step, or round by round, forward or backward, while visualizing the network as well as the enabled and activated nodes; see snapshots in Figure 1. New commands can be programmed so that users can navigate through the simulation according to their needs.

**Predefined and customized daemons.** The daemon, which parameterizes the simulation, can be configured. First, SASA provides several *predefined daemons*, including the synchronous, central, locally central, or distributed daemon; for such daemons, non-determinism is resolved uniformly at random. The user can also build its own *customized daemon*: this is useful to experiment new activation heuristics, or explore worst cases. The daemon can be interactively controlled using a graphical widget: at each step, the user selects the nodes to be activated among the enabled ones. The daemon can also be programmed; such a program can take advantage of the simulation history to guide the simulation into particular directions.

**Test oracles.** Expected safety properties of algorithms can be formalized and used as test oracles. Typically, they involve the number of steps, moves, or rounds that are necessary to reach a (user-defined) legitimate configuration. In order to define such properties, the user program is given access to node state values and activation status [26]. Properties are checked on the fly at every simulation step.

## 3.2 The Core of SASA

The core of SASA is a stand-alone simulator; see Figure 2. The user has to define both a network and a self-stabilizing algorithm following the API given in Section 3.3. The algorithm is written as an OCAML program: the interface has been designed in such a way that the OCAML program implementing the algorithm is as close as possible to guarded commands, the usual way to write algorithms in the ASM. The network topology is specified using the DOT language [4]. The OCAML algorithm is compiled into a dynamic library which is used, together with the DOT network file, by SASA to generate simulation data. A simulation data file contains an execution trace made of the sequence of configurations. This trace also contains the enabled and activated action history. Such traces can be visualized using chronogram viewers.

## 3.3 The Application Programming Interface

The SASA programming interface allows to define the network using the DOT language and the algorithms using a 48-lines OCAML interface Module (called the `Algo` API). Both are presented below.

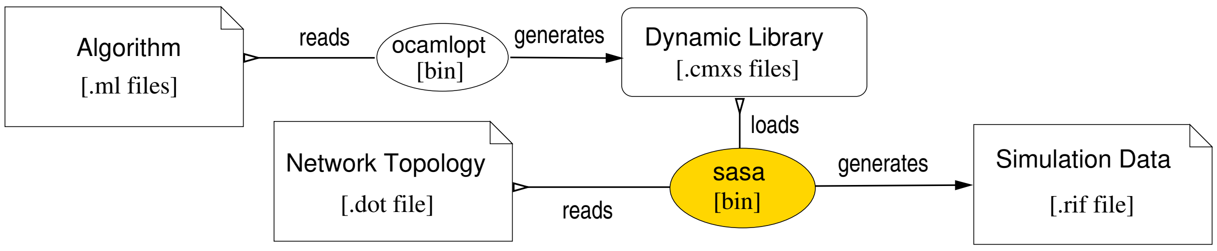

Figure 2: The SASA core simulator architecture

**Networks in DOT.** *Graphviz* [4] is an open-source tool suite for drawing graphs specified using the DOT language. Many visualization tools and graph editors support the DOT format and many bridges from/to other graph tools exist. DOT allows to define *graphs* as sets of *nodes* and *edges*. Graphs, nodes, and edges can have *attributes* specified by name-value pairs. SASA takes advantage of DOT attributes to:

1. associate nodes with their algorithms;
2. optionally associate nodes with their initial states; and
3. associate graphs with simulation parameters.

**Local states.** To define an algorithm at each node of the network, SASA first needs to instantiate the type of its local state. Node states are defined using the *polymorphic type* `'st` which can represent any data the designer needs, *e.g.*, integers, arrays, or structures. Nodes can access their neighbor states using the abstract type `'st neighbor` (the "`'st`" part means that the type `neighbor` is parameterized by the type `'st`). The access to neighboring states is made by function `state` which takes a neighbor as input and returns its state; see Listing 1.

```
type 'st neighbor
val state: 'st neighbor -> 'st
```

Listing 1: Access to neighbors' states

**Algorithms.** To define an instance of the local algorithm of each node, SASA requires (see Listing 2):

1. a list of action names (made of arbitrary strings);
2. an enable function that encodes the guards of the algorithm;
3. a step function that triggers an enabled action;
4. a state initialization function used if no initial configuration is provided in the DOT file. Indeed, even if self-stabilization does not require it, initialization is mandatory to begin the simulation. For example, pseudo-random functions can be used to obtain an arbitrary initial configuration.

```
type action  = string
type 'st enable_fun =
  'st -> 'st neighbor list -> action list
type 'st step_fun =
  'st -> 'st neighbor list -> action -> 'st
type 'st state_init_fun =
  int -> string -> 'st
```

Listing 2: The step, enable, and initialization function types

A function of type `enable_fun` takes the current state of the node and the list of its neighbors as arguments. It returns a list of enabled actions. A function of type `step_fun` takes the same arguments plus the action to activate, and returns an updated state. The initial configuration can be set using an initialization function, of type `state_init_fun`, that takes as argument the number of neighbors of the node, and a string that holds, if needed, the node identifier. Indeed, by default, SASA algorithms are anonymous. But, to implement identified algorithms, the user has to explicitly store the identifier of a node into its state; this can be done during initialization using the second parameter of the initialization function.

**Topological information.** Algorithms usually depend on parameters relative to the network topology. For example, Algorithm 1 requires to know the number of nodes. SASA gives access to those parameters through various functions. Listing 3 presents a few of them: `card` returns the number of nodes in the network (*card* stands for cardinal and `unit` is the OCAML equivalent of `void`); `diameter` returns the diameter of the network, (*i.e.*, the maximum distance between every pair of nodes), `min_degree` (resp. `max_degree`) returns the minimum (resp. maximum) degree of nodes, and `is_connected` returns a Boolean indicating whether the network is connected.

```
val card: unit -> int
val diameter: unit -> int
val min_degree: unit -> int
val max_degree: unit -> int
val is_connected: unit -> bool
```

Listing 3: Some of the topological parameters provided by the API

**Example.** Listing 4 shows the implementation of Algorithm 1 in SASA: notice that we obtain a faithful straightforward translation of Algorithm 1. In particular, according to the `Algo`

API, function `enable_f` has type `enabled_fun` and implements the guards of the algorithm; first the guard of Action $I(p)$ and then the guard of Action $R(p)$. Similarly, function `step_f` has type `step_fun` and implements the statement of Action $I(p)$ and $R(p)$ depending on the name of the action (parameter `a`).

```
open Algo
let n = card ()
let k = n * n + 1
let (init_state: int state_init_fun) =
  fun _ _ -> (Random.int k)
let modulo x n =
  if x < 0 then n+x mod n else x mod n
let behind pc qc = (modulo (qc-pc) k) <= n
let (enable_f: int enable_fun) = fun c nl ->
  if List.for_all
      (fun q -> behind c (state q)) nl
  then ["I(p)"]
  else if
    List.exists
       (fun q -> not (behind c (state q))
              && not (behind (state q) c))
       nl
     && c <> 0
  then ["R(p)"] else []
let (step_f: int step_fun) = fun c nl a ->
  match a with
   | "I(p)" -> modulo (c + 1) k
   | "R(p)" -> 0
   | _ -> assert false
let actions = Some ["I(p)"; "R(p)"]
```

Listing 4: Implementation of Algorithm 1

## 3.4 Design Choices

To implement SASA, we have chosen to rely as much as possible on existing tools and languages. In the following, we justify some of the choices done during the implementation.

**Network description using DOT.** In order to describe the network topology, introducing yet another a format was not an option. Apart from DOT, several other formats exists to describe and draw graphs (*e.g.*, GML, VISML). Advantages of DOT include the fact that it is open-source, standard, easy to edit, and versatile enough thanks to its attribute mechanism. Moreover, there exists plenty of visualization tools and graph editors that support the DOT format. Anyway, bridges from and to most of the other popular alternatives exist.

**Implementing algorithms in OCAML.** A more difficult choice was the one for the language used to define the algorithms themselves. An option would have been to design a new dedicated ad hoc language, *i.e.*, a simple language tailored to the writing usages in the self-stabilizing community. It would have avoided the need to learn a full-fledged programming language, with concepts that are not necessarily useful for the kind of programs ones usually write to solve and study self-stabilizing problems. It would also have allowed to control its abstract syntax tree and to generate programs in other languages (*e.g.*, LATEX code for article inclusion).

On the other hand, using an existing language saves a lot of time: no need to implement and maintain compilers, rich set of libraries, programming environments, *etc*.

The main reason for choosing OCAML is that ml-languages are very popular in several academic communities (the main target of SASA), in particular thanks to their clean semantics and type-safety. The OCAML compiler produce efficient binary code which is important for batch simulations. Moreover, OCAML is freely and widely available on a lot of architectures, has an active community, lots of tools and libraries. Its foreign language interface makes possible the use of other languages such as C, JAVA, JAVASCRIPT, PYTHON, and more. Supporting other languages as input of SASA is thus doable and might be done in the future.

**Dynamic versus static linking.** We have chosen to have a plugin-based architecture (see Figure 2), where user algorithms are (compiled and then) dynamically linked to the SASA simulator. Another possibility would have been to provide a library that, once statically linked to the user programs, would have produced a simulation executable. The major reason for this choice is that this architecture offers a better separation of concerns. Indeed, the user program cannot access code that has not been explicitly exposed by the API, *e.g.*, the SASA engine internals. As a side effect, this approach saves disk space and eases the installation of the tool since there is no need to install SASA dependencies.

```
type value =
    I of int | F of float | B of Bool
  | A of state array | S of struct | [...]
type env = string -> value
```

Listing 5: Representing local states with a variant Type: an alternative to polymo«rphic states

**Polymorphic types to represent local states.** In SASA, node local states are represented by a polymorphic type; see type `'st` in Listings 1, 2, and 3. An alternative would have been to provide a *variant type* (also called *sum*, or *disjoint union* type [3]), such as the type `value` in Listing 5, for enumerating all possible types of local variables required in the algorithm implementation, and to set up an environment (type `env`) that maps any node variable name to its value.

We chose a polymorphic type to represent local states because it is a more general approach. Indeed, with a variant type, possible local variable types would have been restricted to be one element in the `value` list. Furthermore, using variant type lengthens the code with unnecessarily heavy construct, since every occurrence of a local variable in the algorithm requires a pattern-matching construction to access the value. Nevertheless, the use of polymorphic functions together with dynamic linking raises some issues, as explained in the next paragraph.

[3]https://en.wikipedia.org/wiki/Algebraic_data_type

**Dynamic type checking of polymorphic types.** Distributed algorithms are linked to the SASA simulation engine at *runtime*. The dynamic linking in OCAML does not provide facilities to access value names defined by the component to be linked (as in C). Thus, the component should register its entry points with the main program by modifying tables of functions. In SASA, the API functions are polymorphic (in the sense of functional languages parametric polymorphism[4]; *cf.* `'st` in the algorithm local state type). But storing polymorphic values into a mutable table of functions is not possible in ML-languages such as OCAML; one can only store so-called *weakly polymorphic* values, which are useless in an interface file (*cf.* Chapter 5 of [27]). One solution is to use the `Obj` module, and in particular the functions:

- `Obj.obj: 'a -> t` to be able to register polymorphic functions into tables; and
- `Obj.repr: t -> 'a` to retrieve them from tables in the simulation engine.

Those type erasure and type reification functions do nothing but breaking the type system and should therefore be used carefully. Here, the API presented in Section 3.3 imposes that, for each graph, all nodes agree on the local state type. But using the `Obj` library, nothing prevents users to define and compile two nodes working on different types without any compiler complaint, which would lead to runtime errors. In order to let the OCAML type checker make sure that this situation never occurs, we simply require that all local algorithms are registered by a single function call, that takes as argument a list of algorithms; see Listing 6. The algorithms to register have to be provided as a list, via the `algo` field. Note that the polymorphic type `'st` is common to all algorithms; hence the `register` function, called once, enforces one and the same type for all local states.

Notice that the FRAMA-C program analyzer uses the same trick based on the `Obj` module to deal with plugins that implement a polymorphic interface [28, 29]. They dynamically check the plugins correctness using phantom types to make sure that all monomorphic instances of the polymorphic type are compatible. In our context, this is not necessary as we only deal with one type instance at a time.

```
type 'st algo_to_register = {
  algo_id    : string;
  init_state: 'st state_init_fun;
  enab       : 'st enable_fun;
  step       : 'st step_fun;
}
type 'st to_register = {
    algo: 'st algo_to_register list;
    state_to_string:  'st  -> string;
    state_of_string: (string -> 'st) option;
    copy_state: 'st -> 'st;
    actions: action list
  }
(** To be called once *)
val register: 'st to_register -> unit
```

Listing 6: Registration function

[4]https://en.wikipedia.org/wiki/Parametric_polymorphism

## 4 Connection to the Synchrone Reactive Toolbox

The SASA simulator produces an execution trace which is a sequence of output vectors; see the architecture, Figure 2. Furthermore, running the simulator may require some additional inputs at each step in two cases. First, the algorithm itself may require external inputs to model, *e.g.*, requests in case of a resource allocation algorithm. Second, the user may not want to use one of the predefined daemons and may instead wish to control the daemon behavior. In either cases, SASA behaves as a *reactive system*, *i.e.*, it consists of an infinite loop where it first reads its input, then performs a step, and finally produces outputs. Hence, it was easy and fruitful to connect SASA to the *Synchrone Reactive Toolbox* [5], which targets the development and validation of *reactive systems*. Before explaining how we perform this connection, we briefly present the relevant part of this toolbox.

### 4.1 A Toolbox Dedicated to the Design of Correct Reactive Systems

The aforementioned toolbox has been developed over the last two decades around the LUSTRE synchronous programming language [30], which targets the design of *reactive programs*. A reactive program continuously interacts with its environment, typically through sensors and actuators. It is often embedded, with limited memory, and critical. This has motivated the design of languages that forbid programs from using an unbounded amount of memory and execution time, and where time is a first class concept. A lot of work has been dedicated to verification of temporal properties using model checking on such reactive programs. To formally verify a LUSTRE program, model checkers [31, 32] use a formal description of the program environment and its expected properties in order to prove, by exploring all reachable states, that no incorrect behavior ever occurs. When such an exhaustive program verification is not tractable, simulation tools can take advantage of the formalization to automate tests. The formal description of the environment is used to generate random (yet realistic) inputs, while the formalization of expected properties are used as test oracles.

We now briefly present three tools from the *Synchrone Reactive Toolbox* [5], namely LURETTE, LUCIOLE, and RDBG, which are used to run SASA simulations. Their involvement into SASA will be detailed in Subsection 4.2.

**LURETTE** is a black-box (a.k.a. functional) testing tool dedicated to reactive programs [33]. At each discrete logical instant, a reactive program (1) reads inputs, (2) performs a step, and (3) produces outputs. In order to test such a program (called SUT, for System Under Test), LURETTE runs the SUT together (in coroutine) with an environment model; the environment is also a reactive program whose inputs (resp. outputs) are the SUT outputs (resp. inputs). LURETTE also runs the test oracle, which is yet another reactive program that formalizes the algorithm expected properties. It reads the environment and SUT outputs, and returns a Boolean that states whether the test succeeds or not.

**LUCIOLE** is a simple graphical user interface that allows the user to manually choose the input values of reactive programs, step by step. LURETTE automatically calls LUCIOLE when an input of the SUT or its environment is missing.

Figure 3 sums up the dataflow between the various LURETTE components described so far. If necessary, LUCIOLE performs one step and transmits its values to the environment (ENV) and/or the SUT; the environment performs one step and transmits its values to the SUT and the oracle; the SUT performs one step and transmits its values to the oracle; then the oracle performs one step to decide if the test succeeds. The SUT output is stored (in the PRE box, see Figure 3) to be used by the environment for the next simulation step. Afterwards, this whole process is repeated at will. The simulation data are stored in a RIF file (RIF stands for Reactive Input Format). RIF simulation data can be displayed into chronograms using the tools SIM2CHRO or GNUPLOT-RIF.

**RDBG** is a programmable Reactive Program DeBugGer [34]. RDBG uses the same infrastructure as LURETTE, has exactly the same input files and option set, and therefore can be used to track bugs when some oracle is violated. RDBG runs a coroutine between the simulator (the debuggee) and a Real-Eval-Print-loop (REPL) command interpreter (the debugger) that let users (1) inspect simulation values, and (2) navigate from a *watch-point* to another – watch-points are pre-defined observation points, where the user can stop to inspect runtime data such as step counters, or the list of variable values. The RDBG REPL is actually based on an OCAML REPL interpreter where all LURETTE modules are loaded; new debugging commands can therefore be programmed in OCAML by end-users [34], which facilitated the SASA integration.

## 4.2 Taking Advantage of the Reactive Toolbox in SASA

The SASA simulator actually behaves as a reactive program: it reads its input, performs an atomic step, and produces its outputs. The LURETTE and RDBG frameworks can therefore be used to run SASA simulations, and provide new features to the SASA core simulator, as explained below.

**Algorithm properties and test oracles.** LURETTE computes whether the simulation is running correctly using test oracles. A LURETTE oracle can be any reactive program whose inputs are the SUT inputs and outputs, and whose output is a single Boolean value monitoring the execution correctness.

SASA outputs the node state values and two Boolean values per action to indicate whether the action is enabled, and if so whether it is activated. For example, the simulation of the algorithm described in Section 2 outputs five values for each node $p$: an integer for its clock, and two Booleans for each of its two actions $I(p)$ and $R(p)$. Using the algorithm state values at each step, it is possible to compute whether the current configuration is legitimate. Similarly, using the history of enabled and activated actions, we can count the number of moves and rounds. More generally, from those data, the theorems used in the self-stabilizing literature can be formalized and used as oracles to detect *on-the-fly* flaws in theorems or algorithms. The formalization of upper bounds of several classical self-stabilizing algorithms [3] has been done [26]. Nevertheless, this method can only check safety properties, including bounded liveness, since these are the only ones that can be checked by simulations. Conversely, theorems stating, for example, that a bound exists (with no precision about its value) cannot be checked by running simulations.

**Programmable daemons.** The second main characteristic of LURETTE is its ability to provide simulation inputs that can depend on the SUT outputs (feedback loop). First, this is necessary for algorithms with external inputs that may depend on the algorithm behavior. For instance, in a resource allocation problem, the algorithm has to be informed that the allocated resource has been released so that it can allocate it one more time.

Second, we use this feature to program customized daemons. Indeed, SASA has an option to execute in the *customized-daemon mode*; this mode delegates the choice of actions to activate to its environment. Therefore, in this mode, the environment plays the role of the daemon: at each step, SASA (1) reads from the daemon-environment Boolean values indicating for each action whether it has to be activated, (2) performs an atomic step, and (3) outputs Boolean values indicating for each action whether it is enabled for the next step. This allows the daemon-environment to choose the actions to activate at next step among the enabled ones.

If SASA is launched *via* LURETTE in this customized-daemon mode with no environment program, LURETTE will automatically call the LUCIOLE GUI: this allows to execute an interactively controlled daemon, where the user can conveniently choose at each step which actions to activate *via* graphical widgets. Otherwise, the environment-daemon can also be programmed as a reactive component. Programmed daemons can be used to explore worst-case scenarios. Indeed, in many cases (see, for example, worst-case analyzes given in [3]), worst-case scenarios are obtained with a specific initial configuration from which activations are made in a particular order.

For example, in our framework we can easily program a central daemon where nondeterminism is resolved using priorities on node (*e.g.*, using node identifiers) instead of randomization. This kind of customized daemon does not fairly activate processes compared to the standard ones. With well-chosen priorities, we could use such a daemon to exhibit a quadratic scenario in steps (*i.e.*, with same order of magnitude as the known worst case) for the Dijkstra token ring algorithm [1]. Such scenario is very unlikely to occur when using standard daemons.

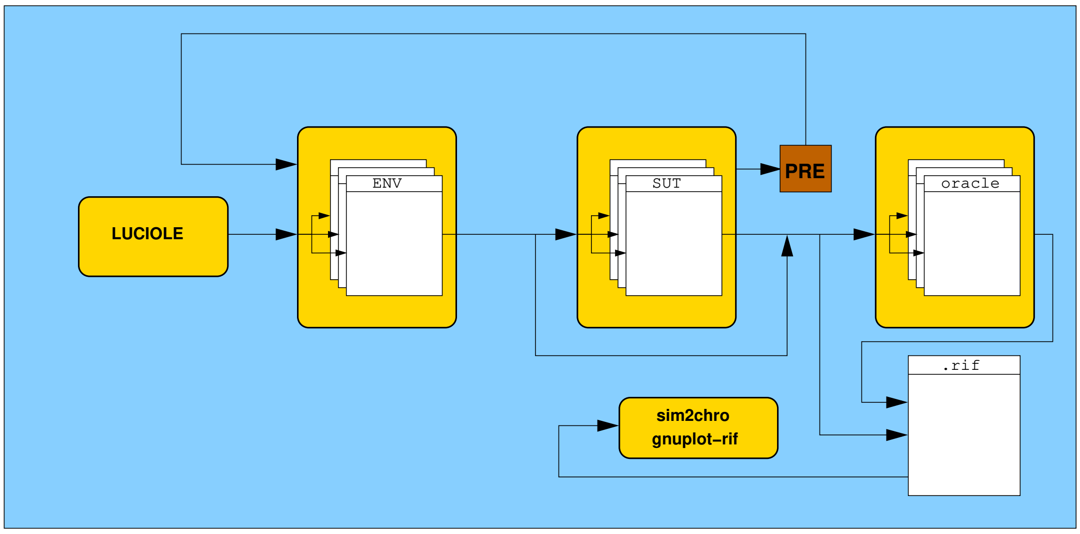

Figure 3: The LURETTE and RDBG dataflow schema. The SUT inputs is made of the environment outputs. The environment inputs is made of the SUT outputs. Missing inputs are provided by the LUCIOLE GUI. SUT and environment outputs are provided to the oracle to decide if the test succeeds. The SIM2CHRO and GNUPLOT-RIF chronogram generators can be used to visualize simulations data. Simulations data are saved into a `.rif` file.

**Debugging and interactive simulations.** Another fruitful use of the synchrone reactive toolbox consists in running SASA simulations from RDBG. First, because it allows to perform simulations step-by-step. Moreover, the programmable capability of RDBG lets us implement easily a set of debugging commands dedicated to SASA. For example, to program an RDBG command that moves forward until the next round, one just needs to define a OCAML function that (1) gets from the current watch-point the list of enabled actions and stores it into a list called `list_ref`; (2) moves to the next watch-point; (3) gets the list of activated and neutralized actions at this step, and uses it to update `list_ref`; (4) returns to (2) if `list_ref` is not empty; otherwise, concludes that a round has elapsed.

Notice that for SASA, no specific instrumentation has been performed, and thus RDBG can only observe the SASA interface variables, namely the algorithm local states as well as the enabled and activated actions. But this coarse-grained information is enough to program RDBG commands useful for SASA simulation purpose, such as:

- moving backward and forward from step to step, or from round to round (using the round function explained above);
- moving forward until a particular (*e.g.*, legitimate) configuration is reached;
- counting the number of steps, moves, or rounds;
- generating a dot file decorated with state values, using different colors for enabled and activated nodes. This file can be generated and output to a visualization tool, so that the user can follow the simulation graphically. That is actually how Figure 1 was generated.

Those commands, together with a few others, are provided in the SASA distribution. More user-defined commands can be added similarly.

## 5 Validation of the Simulator

To validate the tool, we have implemented and tested classical algorithms from the literature using various assumptions on the daemon and topology. The goal of those experiments is threefold. First, the sample of algorithms and assumptions is rich enough to guarantee that the SASA API can express a large set of designer requirements. Second, those algorithms are used to validate the semantics of the model implemented in SASA using various test oracles. Third, we evaluate the performances of the simulator by measuring execution time and memory consumption on large networks.

**Algorithms under test.** We have implemented many existing self-stabilizing algorithms. We give a non-exhaustive list of typical such algorithms below; these latter have been used as benchmarks to evaluate the performances of SASA.

**DTR:** A token circulation for rooted unidirectional rings assuming a distributed daemon [1].

**BFS:** A breadth first search spanning tree construction for connected rooted networks assuming a distributed daemon [3].

**DFS:** A depth first search spanning tree construction for connected rooted networks assuming a distributed daemon [6].

This algorithm has been implemented using two different data structures to encode the local states, namely lists and arrays. This leads to different performances; see **DFS-l** for list implementation and **DFS-a** for array implementation.

**COL:** A vertex-coloring algorithm for anonymous networks assuming a locally central daemon [8].

**SYN:** A synchronous unison for anonymous networks assuming a synchronous daemon [9];

**ASY:** Algorithm 1, used as the running example all along this paper [10].

All these algorithms can be found in the SASA gitlab repository; see [23].

**Methodology.** For each algorithm of the list above, we have written a direct implementation of the original guarded command algorithm. Such implementations include the running assumptions, *e.g.*, the topology and daemon. Then, we have used the interactive graphical feature of SASA through the debugger RDBG to test and debug them on well-chosen small corner-case topologies. Finally, we have implemented test oracles to check known properties on these algorithms, including correctness from (resp. convergence to) a legitimate configuration, as well as bounds on their stabilization time in moves, steps, and rounds, when available. Testing all those properties is a way to check the implementation of the algorithms. But again, as these properties are well-known results, this is, above all, a mean to check whether the implementation of SASA fits the computational model and its semantics.

**Performances.** Some performance results are given in Table 1. They have been obtained on an Intel(R) Core(TM) i7 7 600 CPU at 2.80GHz with 4 gigabytes of RAM. We are interested in comparing the performances of the simulator on the algorithms under test, according to different topologies. Note that every algorithm assumes an arbitrary topology, except **DTR** which requires a ring network. Hence we only present measurements on other algorithms in Table 1: **BFS**, **DFS-l**, **DFS-a**, **COL**, **SYN**, **ASYN**. For each of them, we have run simulations on several kinds of topologies: two square grids, named `grid.dot` and `biggrid.dot`, of 100 nodes (180 links) and 10 000 nodes (19 800 links), respectively; as well as two random graphs, named `ER.dot` and `bigER.dot`, built using the Erdős–Rényi model [35] with 256 nodes (9 811 links, average degree 76) and 2 000 nodes (600 253 links, average degree 600), respectively; see the results in Table 1.

Every simulation, launched automatically, has been run for 10 000 steps, except for the two big graphs (`biggrid.dot` and `bigER.dot`). For these latter, we have only performed 10 steps. Note that some of the algorithms are silent, *i.e.*, they aims at reaching a *terminal* legitimate configuration, where all nodes are disabled. They may stop their execution because they have stabilized before achieving such a number of steps. Therefore, for fair evaluation, we provide the execution time elapsed per step (Time/Step).

As it only works on rings, DTR does not appear in Table 1. For the ring of size 1 000, we measured 5 ms per step, and consumed 14 megabytes. For the ring of size 10 000, we measured less than 1 second per step, and consumed 42 megabytes.

Note that every simulation has been performed without data file generation. Indeed, for large networks, this would produce huge files and the simulator would use most of its time writing the data file. For example, the simulation of **DFS-a** on `biggrid.dot` would generate 0.8 gigabyte of data per step (100 millions integer values); a 10 000 steps simulation would therefore need to write several thousands of TB, which would be disk- and time-consuming. For such examples, being able to generate inputs and check oracles on the fly is a real advantage.

## 6 Simulation Campaigns

This section illustrates how to take advantage of SASA in batch mode to study the time complexity of self-stabilizing algorithms. As an illustrative example, we show how various vertex-coloring algorithms can be compared and how SASA can be used to empirically study the influence of topologies and daemons on their stabilization time in moves, steps, and rounds.

### 6.1 Three Vertex-Coloring Algorithms

We consider three randomized vertex-coloring algorithms [3, 7, 8] designed for arbitrary anonymous networks. Those algorithms are randomized variants of Algorithm **COL** that withstand the distributed daemon. Actually, randomness is mandatory under the distributed daemon since self-stabilizing vertex-coloring cannot be solved deterministically in regular anonymous networks [3] for example.

In all these variants, the local state of each node $p$ is made of a single integer variable $p.c$ whose domain is $\{0,...,K\}$, where $K$ is greater than or equal to the maximum degree of the network. The variable $p.c$ is called *the color of* $p$. In a given configuration, a color $x \in \{0,...,K\}$ is said to be *free* at node $p$ if no neighbor $q$ of $p$ satisfies $q.c = x$. Each of the considered algorithms aims at reaching a configuration from which $p.c$ contains a constant free color, for every node $p$. To that goal, each algorithm has a single action at each node, described below.

**COL-a1:** "Uniform When Activated" [7].

- A node $p$ is enabled if its color is not free.
- If $p$ is activated, $p.c$ is set to a value selected uniformly at random in the set made of $p.c$ and all free colors.

**COL-a2:** "Smallest When Activated" [3].

- A node $p$ is enabled if its color is not free.

[6]Remark that, memory consumptions presented here are quite different from, and sometimes significantly better than, those given the conference paper [22]. Actually, overtime we have changed the machine used to make our experiments and also slightly modified the code of SASA.

| | `grid.dot` 100 nodes, 180 links density: 3.6 % | | `ER.dot` 256 nodes, 9 811 links density: 0.04 % | | `biggrid.dot` 10 000 nodes, 19 800 links density: 30 % | | `bigER.dot` 2 000 nodes, 600 253 links density: 30 % | |
|---|---|---|---|---|---|---|---|---|
| | Time/Step | Memory | Time/Step | Memory | Time/Step | Memory | Time/Step | Memory |
| **BFS** | 0.4 ms | 10 MB | 15 ms | 30 MB | 3 s | 74 MB | 4 s | 1218 MB |
| **DFS-l** | 0.8 ms | 12 MB | 173 ms | 37 MB | 3 s | 74 MB | 93 s | 1218 MB |
| **DFS-a** | 0.5 ms | 11 MB | 129 ms | 34 MB | 7 s | 3630 MB | 147 s | 1397 MB |
| **COL** | 0 ms | 10 MB | 13 ms | 34 MB | 21 s | 74 MB | 10 s | 1218 MB |
| **SYN** | 0.3 ms | 11 MB | 8 ms | 34 MB | 2 s | 74 MB | 5 s | 1220 MB |
| **ASY** | 0.1 ms | 11 MB | 6 ms | 37 MB | 53 ms | 74 MB | 4 s | 1218 MB |

Table 1: Performance evaluation of SASA on the benchmark algorithms. Time elapsing is measured in user+system time in seconds or milliseconds, and has been divided by the number of simulation steps. Memory consumption is given in megabytes (MB), and has been obtained using the "maximum resident set size" given by the GNU `time` utility.[6]

**Algorithm 2** The "smallest when activated" coloring algorithm (**COL-a2**, [3]): local algorithm for each node $p$

**Constant Input:** $\mathcal{N}_p$, the set of $p$'s neighbors

**Variable:** $p.c \in \{0,...,K\}$, where $K \geq \Delta$, and $\Delta$ is the maximum degree of the network.

**Predicate:** $free(x) = \forall q \in \mathcal{N}_p, q.c \neq x$

**Function:** $Random()$: returns a Boolean value generated uniformly at random.
$min\{S\}$: returns the minimum value of the non-empty set $S$.

**Actions:**

```
$Conflict(p)$  ::  $\neg free(p.c)$  $\hookrightarrow$  if $Random()$ then $p.c \leftarrow min\{x, x \in \{0,...,K\} \wedge free(x)\}$
```

- If $p$ is activated, it tosses a coin (uniformly at random) to decide between leaving $p.c$ unchanged and setting $p.c$ to the smallest free color.

**COL-a3:** "Always the Biggest" [8].

- A node $p$ is enabled if its color is not the biggest free color.
- If $p$ is activated, it tosses a coin to decide (uniformly at random) between leaving $p.c$ unchanged and setting $p.c$ to the biggest free color.

The second and third algorithms almost only differ by their guards, *i.e.*, the condition under which a node is enabled. Indeed, choosing the biggest or the smallest free colors has no impact, yet this is the way they were defined in their corresponding papers. However, the third algorithm is more specific for the choice of colors: a node may be enabled even when its color is free. As a consequence, the third algorithm has less legitimate configurations, and thus is expected to stabilize more slowly.

As an illustrative example, the "Smallest When Activated" coloring algorithm (**COL-a2**) is presented in Algorithm 2 and its implementation in SASA is provided in Listing 7. Other algorithms (**COL-a1** and **COL-a3**) are similar.

## 6.2 Automating Simulation Campaigns

We provide in the SASA distribution a set of tools that automate the execution of simulation campaigns. Those programs are made of a few hundred lines of OCAML and R code; actually, they could have been written in any language, as they only depend on command-line tools. The distribution contains:

- tools to generate classical topologies (rings, cliques, grids, trees, etc.) and random (*e.g.*, Erdős–Rényi) graphs in the dot format;
- tools to parameterize and launch simulation campaigns, (*i.e.*, launch the simulator as many times as required on specified contexts) and log the results; and
- tools to visualize those results as plots.

A *simulation campaign* automatically runs SASA on several algorithms, using various daemons, on various graphs. A *simulation* estimates, for a particular algorithm-daemon-graph triplet, the average stabilization time according to three complexity mesure units: moves, steps and rounds. To compute estimations for the three complexity measures, new simulations (at least 10) are computed until a satisfactory precision is reached. More precisely, new simulations are launched as long as the size of the confidence interval, with a confidence level of 95%, is higher than a particular percentage (called *precision percentage*, typically 1%) of the estimations.

The simulation campaign is fully described using a simple and easy to parameterize script (which is available in the SASA toolbox – see Listing 8 in Appendix for the full script). To obtain the result shown in this paper about the three randomized coloring algorithms (see Figure 4 to 6), we set the precision percentage to 1% and chose to perform the simulations using three daemons (the synchronous, locally central and distributed daemons) over three families of graphs (rings, cliques and random graphs). Precisely, we used 10 rings of size 500 to 5 000, 10 cliques of size 30 to 300, and 10 Erdős–Rényi random graphs [35] (generated with a probability of 0.4 for creating an edge) of size 30 to 300.

In order to compute the estimations for the 270 triplets (about rounds, steps, moves) with a precision of 1%, 180 818

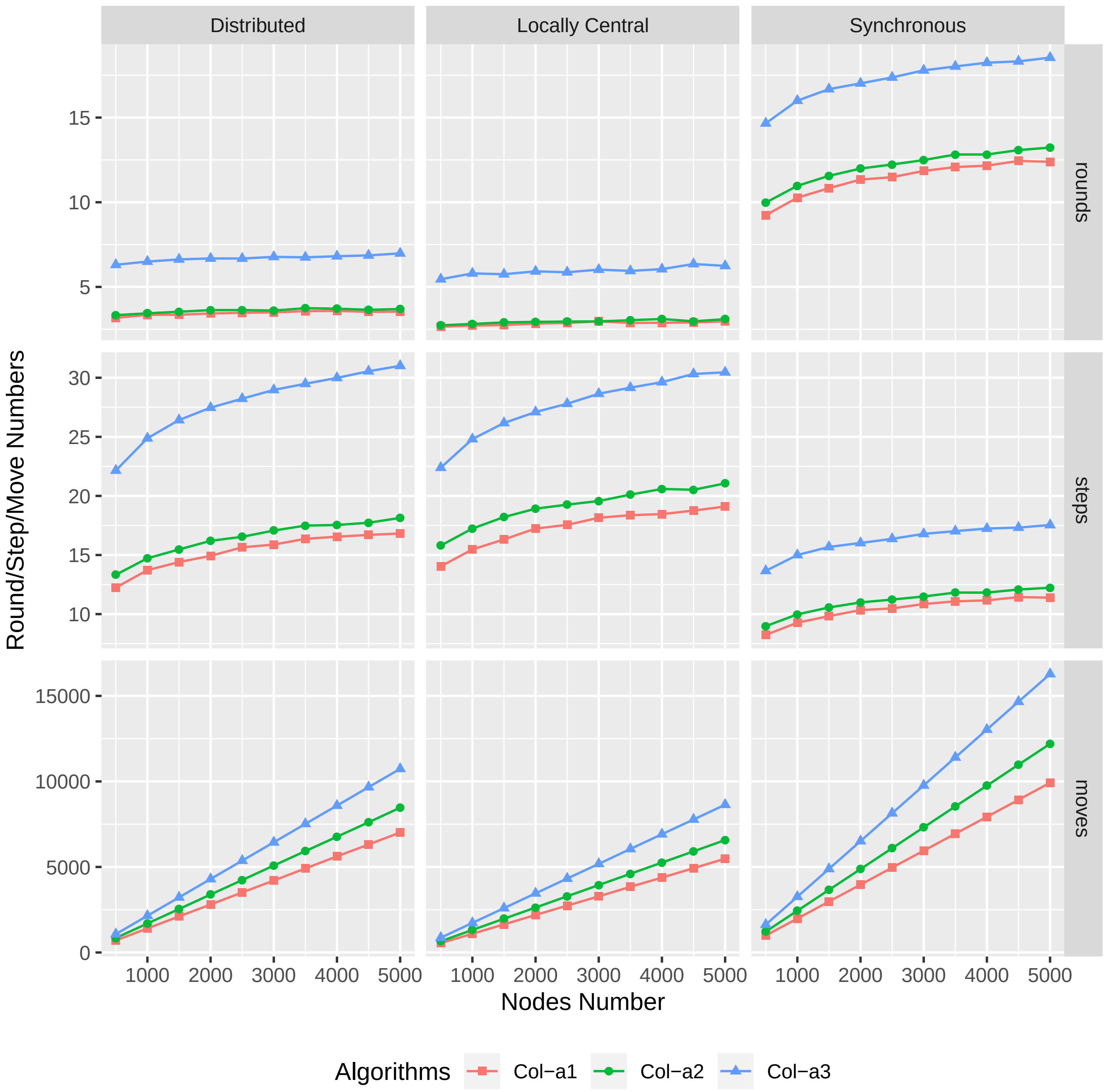

Figure 4: A simulation campaign performed on rings of size 500 to 5 000 (estimated means)

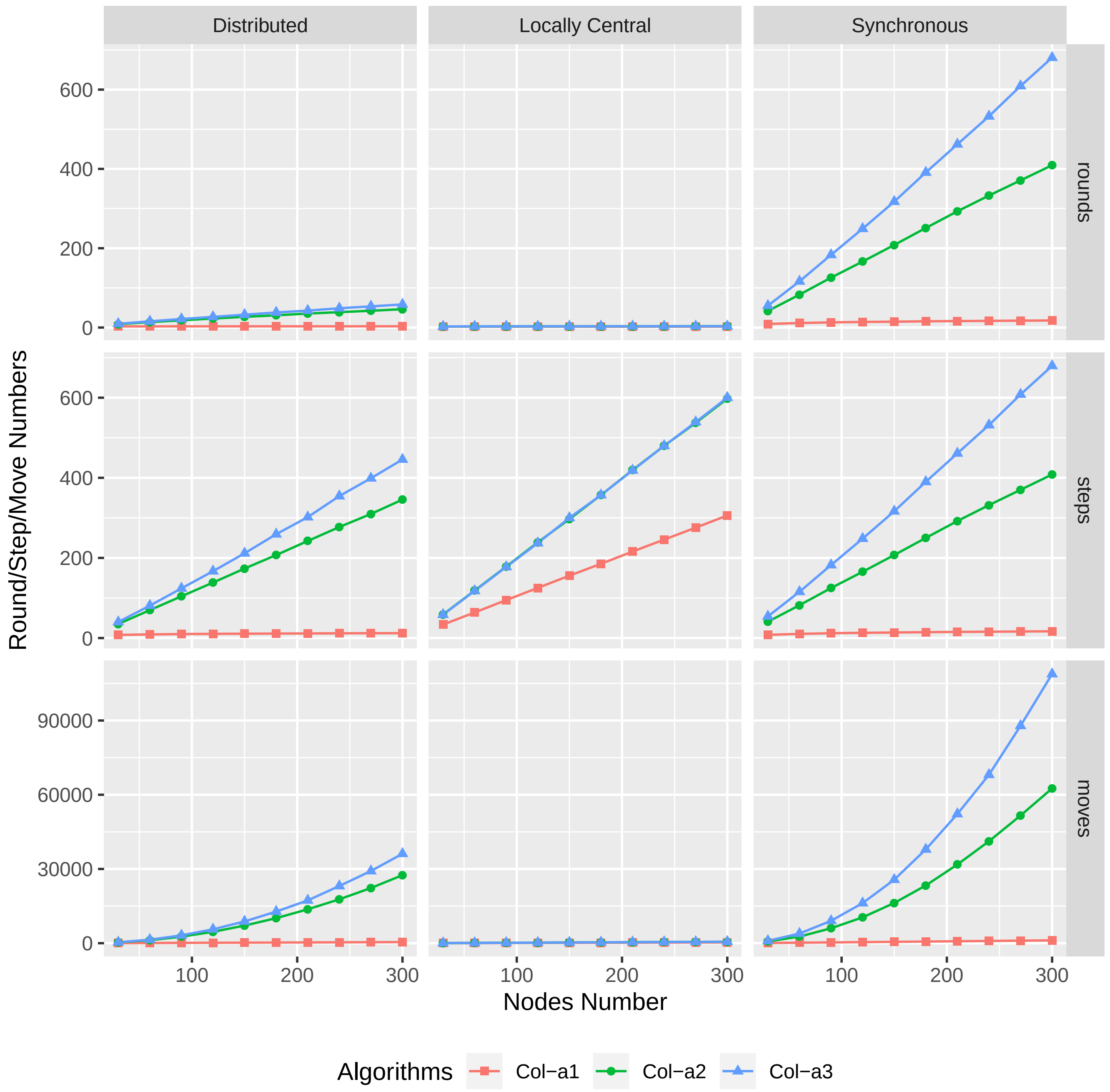

Figure 5: A simulation campaign performed on cliques of size 30 to 300 (estimated means)

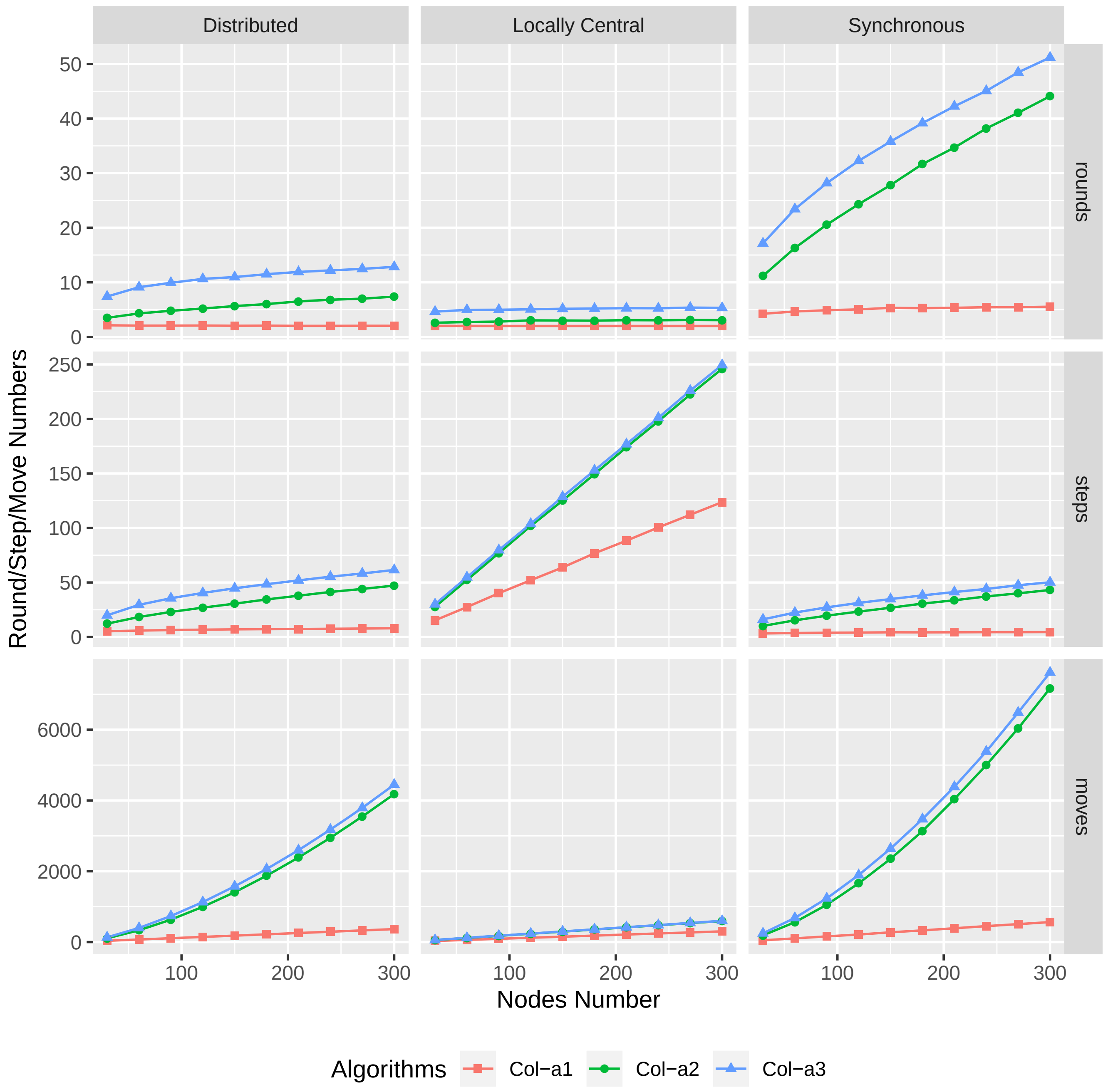

Figure 6: A simulation campaign performed on random graphs of size 30 to 300 (estimated means)

```
open Algo
let k = max_degree () + 1
let (init_state: int state_init_fun) = fun _ _ -> 0
let get_free nl = (* From a neighbor list, returns the free colors is asc. order: O(n.log(n)) *)
  let comp_neg x y = - (compare x y) in (* to get the result in descending order *)
  let n_colors = List.map (fun n -> state n) nl in
  let n_colors = List.sort_uniq comp_neg n_colors in (* neighbor colors, no dupl, desc order *)
  let rec aux free n_colors i = (* for i=k-1 to 0, put i in free if not in n_colors *)
    if i < 0 then free else
      (match n_colors with
       | x::tail -> if x = i then aux free tail (i-1) else aux (i::free) n_colors (i-1)
       | [] -> aux (i::free) n_colors (i-1)
      )
  in
  aux [] n_colors (k-1)
let (enable_f: int enable_fun) = fun c nl ->
  if List.exists (fun n -> state n=c) nl then ["conflict"] else []
let (step_f: int step_fun) = fun e nl a -> match a with
  | ["conflict"] -> if (Random.bool ()) then e else (List.hd (get_free nl))
  | _ -> assert false
let actions = ["conflict"]
```

Listing 7: Implementation of the "smallest when activated" coloring algorithm **COL-a2**

simulations were necessary (669 simulations per triplet on average, and 3 116 at worst). By launching 30 jobs in parallel, we get the result in less than four hours on a machine with 64 Intel Xeon Gold 6138 CPU @ 2.00GHz; those four hours represents 94 hours of wall-clock cumulative time. Notice that, in order to take a first glance at the results, one can speed-up the simulation by relaxing the precision requirements, at the cost of degrading the curves smoothness. By using a precision of 10% on this campaign, less than 15 minutes were necessary on the same machine (6 hours of wall-clock cumulative time).

The simulation campaign generates raw data files, where lines are made of the graph name, the algorithm, the complexity measure, as well as their minimal, maximal, and mean values. From such data files, the script produces several kinds of visualization (as pdf files), such as the ones provided in Figures 4, 5 and 6. In those figures, we present the estimated means; the script also generates similar curves with minimum and maximum values. More details about this simulation campaign are provided in [36].

Each figure stands for a family of graphs: Figure 4 for rings, Figure 5 for cliques, and Figure 6 for Erdős–Rényi graphs. Sub-pictures in a given raw (resp. line) use the same daemon (resp. measure). For example, in Figure 4, the bottom-right sub-picture presents the simulation results obtained for Algorithms **COL-a1** (in red), **COL-a2** (in green) and **COL-a3** (in blue) executed on ring networks using a *synchronous* daemon; the curves represent the number of *moves* (vertical axis), which ranges from a few hundred to more than 10 000, depending on the size of the ring (horizontal axis), which ranges from 500 to 5 000 nodes.

Therefore, for each of the nine sub-pictures of those figures, one can compare the asymptotic behavior of each algorithm when the size of the graphs increases, for a particular daemon and a particular measure. Here, the curves evolve with respect to the graphs size, since this quantity is interesting for the algorithms under study. Note that for other algorithms, it may be more interesting to increase the diameter of the graph, or its maximal degree.

Studying accurately the asymptotic behavior of those algorithms is not the purpose of this paper; nevertheless, observe that the curves obtained by such kind of simulations may provide valuable insights before starting any theoretical study. This can also complete an analytical complexity study. For example, when an empirical average complexity is drastically smaller than the known upper bound (like we will see here for the coloring problem), this suggests that either (1) the worst-case scenario is infrequent, or (2) the known upper bound is far from being tight. Follow some remarks raised by those simulations.

- Algorithm **COL-a1** according to [7] is expected to stabilize in at most $(\Delta+1)\cdot n$ moves, where $n$ is the number of nodes in the network and $\Delta$ its maximum degree. Actually, the sub-pictures in Figure 4 - bottom-line, show linear curves which may suggest that, for rings, the mean case is linear as is the analytical worst case.
- As far as the number of steps is concerned, [8] states that **COL-a3** (see blue curves, middle line in every figure) should stabilize in $O(n\ln(n))$ on average, while our simulations suggest that it is far less (it might be linear and even logarithmic in the ring case).
- To the best of our knowledge, the asymptotic number of rounds has not been studied for those algorithms. Our simulations suggest an $O(n)$ average stabilization time in rounds on cliques (see upper line in Figure 5), while it seems to be lower - maybe $O(\ln(n))$ - for other families, ring and random graphs, see upper lines in Figures 4 and 6.

Those remarks directly come from the shape of the curves; they can be strengthened using statistical analysis (see script and results from the SASA toolbox in [37]).

## 7 Conclusion and Future Work

This paper presents an open-source SimulAtor of Self-stabilizing Algorithms, called SASA. Its programming interface is simple, yet rich enough to allow a direct encoding of any distributed algorithm written in the atomic-state model, the most commonly used model in the self-stabilizing area.

In order to limit the engineering effort, SASA relies on existing tools such as the OCAML programming environment to define the algorithms, DOT to define the networks, and the Synchrone Reactive Toolbox [5] to carry out formal testing and interactive simulations.

We will continue to enrich SASA, in particular to handle more adversarial environments (*e.g.*, to handle topological changes) and to be able to more tightly evaluate the average performances of self-stabilizing algorithms. For example, we have included a fault injection mechanism. We plan to use this mechanism to inject a few faults after the stabilization of an given algorithm and then evaluate whether it is efficient (on the average) w.r.t. fault-containing-related metrics such as the containment radius, the contamination number, and the fault gap [38]. Another possible extension would be to develop tools to construct worst-case executions.

In the spirit of TLA+ [39], an interesting future work consists in connecting SASA to tools enabling formal verification of self-stabilizing algorithms. By connecting SASA to model-checkers [32, 31], the expected properties specified as LUSTRE oracles could be verified on some particular networks.

Furthermore, SASA could be connected to the PADEC framework [40], which provides libraries to develop mechanically checked proofs of self-stabilizing algorithms using the Coq proof assistant [41]. Since Coq is able to perform automatic OCAML program extraction, we should be able to simulate the certified algorithms using the same source. During the certification process, it could be useful to perform simulations in order to guide the formalization into Coq theorems, or find flaws (*e.g.*, in technical lemmas) early in the proof elaboration.

## Acknowledgements

This study has been partially supported by the French ANR projects ANR-16-CE40-0023 (DESCARTES) and ANR-16-CE25-0009-03 (ESTATE).

## References

[1] Dijkstra, E. W. (1974) Self-stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, **17**, 643–644.

[2] Dolev, S. (2000) *Self-Stabilization*. MIT Press, Cambridge, Massachusetts, États-Unis.

[3] Altisen, K., Devismes, S., Dubois, S., and Petit, F. (2019) *Introduction to Distributed Self-Stabilizing Algorithms*, Synthesis Lectures on Distributed Computing Theory, **8**. Morgan & Claypool Publishers, London.

[4] Gansner, E. R. and North, S. C. (2000) An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, **30**, 1203–1233.

[5] Jahier, E. and Raymond, P. The synchrone reactive toolbox. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/reactive-toolbox.

[6] Collin, Z. and Dolev, S. (1994) Self-stabilizing depth-first search. *Inf. Process. Lett.*, **49**, 297–301.

[7] Bernard, S., Devismes, S., Paroux, K., Gradinariu Potop-Butucaru, M., and Tixeuil, S. (2010) Probabilistic Self-stabilizing Vertex Coloring in Unidirectional Anonymous Networks. *11th International Conference on Distributed Computing and Networking, ICDCN 2010*, Kolkata, India, January, Lecture Notes in Computer Science, **5935**, pp. 167–177. Springer.

[8] Gradinariu, M. and Tixeuil, S. (2000) Self-stabilizing vertex coloration and arbitrary graphs. In Butelle, F. (ed.), *Procedings of the 4th International Conference on Principles of Distributed Systems (OPODIS)*, Paris, France, December 20-22 Studia Informatica Universalis, pp. 55–70. Suger, Saint-Denis, rue Catulienne, France.

[9] Arora, A., Dolev, S., and Gouda, M. G. (1991) Maintaining digital clocks in step. *Parallel Processing Letters*, **1**, 11–18.

[10] Couvreur, J., Francez, N., and Gouda, M. G. (1992) Asynchronous unison (extended abstract). *Proceedings of the 12th International Conference on Distributed Computing Systems*, Yokohama, Japan, June 9-12, pp. 486–493. IEEE Computer Society.

[11] Adamek, J., Farina, G., Nesterenko, M., and Tixeuil, S. (2017) Evaluating and optimizing stabilizing dining philosophers. *J. Parallel Distrib. Comput.*, **109**, 63–74.

[12] Adamek, J., Nesterenko, M., and Tixeuil, S. (2012) Evaluating practical tolerance properties of stabilizing programs through simulation: The case of propagation of information with feedback. In Richa, A. W. and Scheideler, C. (eds.), *Stabilization, Safety, and Security of Distributed Systems - 14th International Symposium*, Toronto, Canada, October 1-4, Lecture Notes in Computer Science, **7596**, pp. 126–132. Springer.

[13] Fraboulet, A., Chelius, G., and Fleury, E. (2007) Worldsens: Development and prototyping tools for application specific wireless sensors networks. *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 25-27 IPSN '07, pp. 176–185. ACM.

[14] Varga, A. and Hornig, R. (2008) An overview of the omnet++ simulation environment. *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Marseille, France, March Simutools '08,

pp. 60:1–60:10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[15] Issariyakul, T. and Hossain, E. (2010) *Introduction to Network Simulator NS2*, 1st edition. Springer Publishing Company, Incorporated, Berlin, Germany.

[16] Datta, A. K., Devismes, S., Heurtefeux, K., Larmore, L. L., and Rivierre, Y. (2016) Competitive self-stabilizing k-clustering. *Theoretical Computer Science (TCS)*, **626**, 110–133.

[17] Flatebo, M. and Datta, A. K. (1992) Simulation of self-stabilizing algorithms in distributed systems. *Proceedings 25th Annual Simulation Symposium*, Orlando, Florida, USA, April 6-9, pp. 32–41. IEEE Computer Society.

[18] Har-Tal, O. (2000). A simulator for self-stabilizing distributed algorithms. https://www.cs.bgu.ac.il/~projects/projects/odedha/html/. Distributed Computing Group at ETH Zurich.

[19] Müllner, N., Dhama, A., and Theel, O. E. (2008) Derivation of fault tolerance measures of self-stabilizing algorithms by simulation. *Proceedings of the 41st Annual Simulation Symposium*, Ottawa, Canada, April 14-16, pp. 183–192. IEEE Computer Society.

[20] Evcimen, H. T., Arapoglu, O., and Dagdeviren, O. (2018) Selfsim: A discrete-event simulator for distributed self-stabilizing algorithms. *International Conference on Artificial Intelligence and Data Processing (IDAP)*, Malatya, Sep 28, 2018 - Sep 30, pp. 1–6. IEEE.

[21] Trivedi, K. S. (2002) *Probability and Statistics with Reliability, Queuing and Computer Science Applications, Second Edition*. Wiley, Hoboken (New Jersey).

[22] Altisen, K., Devismes, S., and Jahier, E. (2020) sasa: A simulator of self-stabilizing algorithms. In Ahrendt, W. and Wehrheim, H. (eds.), *Tests and Proofs - 14th International Conference, TAP@STAF 2020*, Bergen, Norway, June 22-23, Lecture Notes in Computer Science, **12165**, pp. 143–154. Springer.

[23] Jahier, E. SASA: a SimulAtor of Self-stabilizing Algorithms. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/sasa.

[24] Jahier, E. SASA artifact: How to reproduce the results. https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/reproducible-research/sasartifact.

[25] Boulinier, C. (2007) L'unisson. Thesis (in french) Université de Picardie Jules Verne.

[26] Jahier, E. Verimag Tools Tutorials: Tutorials related to SASA. https://www-verimag.imag.fr/vtt/tags/sasa/.

[27] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., and Vouillon, J. The objective caml system, documentation and user's manual. https://caml.inria.fr/pub/docs/manual-ocaml/.

[28] Cuoq, P., Signoles, J., Baudin, P., Bonichon, R., Canet, G., Correnson, L., Monate, B., Prevosto, V., and Puccetti, A. (2009) Experience report: Ocaml for an industrial-strength static analysis framework. *SIGPLAN Not.*, **44**, 281–286.

[29] Signoles, J. (2011) Une bibliothèque de typage dynamique en OCaml. *22e journées francophones des langages applicatifs (JFLA'11)*, La Bresse, January Studia Informatica Universalis, pp. 1–17. Hermann.

[30] Jahier, E., Raymond, P., and Halbwachs, N. (2018) *The Lustre V6 Reference Manual*. VERIMAG, Grenoble.

[31] Ratel, C., Nicolas, and Raymond, P. (1991) Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 112–119. ACM.

[32] Champion, A., Mebsout, A., Sticksel, C., and Tinelli, C. (2016) The kind 2 model checker. In Chaudhuri, S. and Farzan, A. (eds.), *Computer Aided Verification - 28th International Conference, CAV 2016*, Toronto, ON, Canada, July 17-23, Lecture Notes in Computer Science, **9780**, pp. 510–517. Springer.

[33] Jahier, E., Halbwachs, N., and Raymond, P. (2013) Engineering Functional Requirements of Reactive Systems using Synchronous Languages. *International Symposium on Industrial Embedded Systems*, Porto, Portugal, 06, pp. 140–149. IEEE.

[34] Jahier, E. (2016) RDBG: a reactive programs extensible debugger. In Stuijk, S. (ed.), *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems, SCOPES 2016*, Sankt Goar, Germany, May 23-25, pp. 116–125. ACM.

[35] Erdös, P. and Rényi, A. (1959) On random graphs. *Publicationes Mathematicae Debrecen*, **6**, 290–297.

[36] Jahier, E. Comparing randomized coloring algorithms. https://www-verimag.imag.fr/vtt/articles/comparing-randomized-coloring/.

[37] Jahier, E. Statistical analysis of data obtained by simulation campaigns. http://www-verimag.imag.fr/vtt/articles/stat-analysis.

[38] Herman, T. and Pemmaraju, S. V. (2000) Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, **73**, 41–46.

[39] Lamport, L. (2002) *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., USA.

[40] Altisen, K., Corbineau, P., and Devismes, S. (2017) A framework for certified self-stabilization. *Logical Methods in Computer Science*, **13**.

[41] Bertot, Y. and Castéran, P. (2004) *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions* Texts in Theoretical Computer Science. An EATCS Series. Springer, Berlin, Germany.

## A An example of Simulation campaign script

Listing 8 contains an OCaml program, included in the SASA distribution, see `coloring_campaign.ml`, that computes the simulation campaign explained in the article about the three randomized coloring algorithms.

The precision is set to 1% in Line 2 and and the algorithms implementation directories of Algorithm **COL-a1**, **COL-a2** and **COL-a3** are supplied in Lines 3 and 4. We have set the three daemons under which the algorithms will be executed, see Line 5: the synchronous daemon (`-sd`), the locally central daemon (`-lcd`), and the distributed daemon (`-dd`).

The networks are described in the `networks` variable (see Line 9) which contains the 30 graphs that will be used in the simulation campaign. Precisely, Line 6 defines 10 rings from size 500 to 5 000, Line 7 10 cliques of size 30 to 300, and Line 8 10 Erdõs–Rényi random graphs [35] (generated with a probability of 0.4 for creating an edge) of size 30 to 300.

This program uses the `genExpeMakefiles.ml` (see Line 1) and `parseLog.ml` (see Line 14) OCAML programs, provided in the SASA distribution, and generates a Makefile (called `Makefile.expe-rules`, see Line 12). This Makefile is used to launch the simulation campaign (Line 22). Note that simulations can be run in parallel using the `-jobs` option of `make`; see Line 22.

```
 1 #use "genExpeMakefiles.ml";;
 2 precision := 0.01;;
 3 let dir = "../../test/"
 4 let algos = [dir^"alea-coloring-alt"; dir^"alea-coloring-unif"; dir^"alea-coloring"]
 5 let daemons = ["-sd"; "-lcd"; "-dd"]
 6 let rings   = List.init 10 (fun n->Ring (500*(n+1)))    (*Rings of size 500, 1000, ..., 5000*)
 7 let cliques = List.init 10 (fun n->Clique (30*(n+1)))  (*Cliques of size 30, 60, ..., 300*)
 8 let er      = List.init 10 (fun n->ER (30*(n+1), 0.4)) (*ER of size 30, 60, ..., 300*)
 9 let networks = rings @ cliques @ er
10
11 let gen_make_rules () = (* Generate a Makefile that can launch all the simulations *)
12   gen_makefile "Makefile.expe-rules" daemons algos networks;;
13
14 #use "parseLog.ml";;
15 let gen_pdf () =
16   let gl = ["clique"; "ring"; "er"] in
17   (* Parse simulation log files to extract data and generate graphics via an R script *)
18   parse_log ["Col-a1", "alea-coloring-unif"] gl daemons;
19   parse_log ["Col-a2", "alea-coloring"]      gl daemons;
20   parse_log ["Col-a3", "alea-coloring-alt"]  gl daemons;
21   List.iter (fun n -> sh ("./gen_pdf.r "^n^".data coloring")) gl
22 let _ = gen_make_rules (); sh "make; make cmxs; make --jobs 30 log"; gen_pdf ()
```

Listing 8: How to parameterize a simulation campaign and generate Figures 4, 5, and 6.