A Tutorial on SASA
a SimulAtor of Self-stabilizing Algorithms

Erwan Jahier

Joint work with
Karine Altisen and Stéphane Devismes

November 12, 2020
Outline

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Distributed Systems Algorithms

- Process
  - Autonomous
  - Interconnected
Distributed Systems Algorithms

- Process
  - Autonomous
  - Interconnected
Distributed Systems Algorithms

- Process
  - Autonomous
  - Interconnected
- Hypotheses
  - Connected
  - Bidirectional
  - Identified
Distributed Systems Algorithms

- Process
  - Autonomous
  - Interconnected

- Hypotheses
  - Connected
  - Bidirectional
  - Identified

- Expected Property
  - Fault-tolerance
Self-Stabilizing Algorithms

Configurations

Transient faults

Time

Legitimate configurations
Self-Stabilizing Algorithms

Configurations

Legitimate configurations

Time

Transient faults

Legitimate

Illegitimate

Erwan Jahier

A Tutorial on SASA

November 12, 2020
Self-Stabilizing Algorithms

Configurations

Transient faults

Time

Legitimate configurations
Self-Stabilizing Algorithms

Configurations

Transient faults

Legitimate configurations

Stabilization time
The Atomic State Model (ASM)

- From a particular Configuration of local Memories
- Performing an **Atomic Step** consists in:

```plaintext
1. Reading neighbors variables
2. Computing enabled nodes
3. Choosing nodes to activate: a Daemon models the asynchronism
4. Computing a new configuration
```
The Atomic State Model (ASM)

- From a particular Configuration of local Memories
- Performing an **Atomic Step** consists in:
  1. Reading neighbors variables
The Atomic State Model (ASM)

- From a particular Configuration of local Memories
- Performing an **Atomic Step** consists in:
  1. Reading neighbors variables
  2. Computing enabled nodes
The Atomic State Model (ASM)

• From a particular Configuration of local Memories
• Performing an Atomic Step consists in:
  1. Reading neighbors variables
  2. Computing enabled nodes
  3. Choosing nodes to activate: a Daemon models the asynchronism
The Atomic State Model (ASM)

- From a particular Configuration of local Memories
- Performing an **Atomic Step** consists in:
  1. Reading neighbors variables
  2. Computing enabled nodes
  3. Choosing nodes to activate: a Daemon models the asynchronism
  4. Computing a new configuration
Goal: Study the Algorithm Complexity

- Space Complexity: memory requirement in bits
- Time Complexity (mainly stabilization time) in
  - steps, moves
  - rounds: capture the execution time of the slowest processes

Processes

Time

1<sup>st</sup> round  2<sup>nd</sup> round

Key: Enabled Activated Neutralized
Message Passing Versus Atomic State Models

- **Message Passing Model (MPM)**
  - Used in the Distributed Algorithms community
  - Lower-level: queues of events

- **Atomic State Model (ASM):**
  - A.k.a, the locally shared memory model with composite atomicity
  - Used in the Self-Stabilizing Algorithms community
  - Higher-level: atomic instantaneous communications
  - General Algorithms transformations into MPM methods exist
Simulating Self-stabilizing Algorithms: What for?

- **Debugging**
  - Simulate existing algorithms
  - Design new algorithms
- **Get Insights on the Algorithms Complexity**
  - Average case Complexity
  - Check if the theoretical worst case is good/correct
  - etc.
Existing Simulators of Distributed Systems

• Most simulators work with the Message passing Model (MPM)
• Networking Simulators
  ▶ Architecture-dependent
  ▶ Measures Wall-clock simulation time
• Systematic Methods exist to translate ASM into MPM, but
  ▶ not the same level of abstractions: not good for debugging
  ▶ loose relation with the number of steps, moves, or rounds in the ASM
  ▶ being lower-level, simulations can be very slow: restricted to small topology and simple algorithms
A few Simulators Dedicated to Self-Stabilization exist but

- tailored to **specific needs**
  - mutual exclusion
  - leader election

- provides a few features
  - work on **Specific Topologies**
  - can check pre-defined properties only (e.g., convergence)
  - small set of predefined Daemons
  - complexity in **steps** only (no moves, **no rounds**)


\[\text{Erwan Jahier} \quad \text{A Tutorial on SASA} \quad \text{November 12, 2020}\]
What is missing to the Self-Stabilizing community?

A Simulator able to:

- handle any algorithm written in the ASM
  - simulation close to the model
  - light-weight
- check any property, in terms of steps, moves, or rounds
- to define what the Legitimate Configurations are
- be used with any daemon
What is missing to the Self-Stabilizing community?

A Simulator able to:
- handle any algorithm written in the ASM
  - simulation close to the model
  - light-weight
- check any property, in terms of steps, moves, or rounds
- to define what the Legitimate Configurations are
- be used with any daemon

Well... Not anymore!

SASA: main features
Defining The Network Topology

- Take advantage of the GraphViz dot language
  - Simple syntax
  - Open-source
  - Plenty of visualizers, editors, parsers, exporters

```
digraph ring7 {
  root [ algo="root.ml" init="{root=1;v=1}" ]
  p2 [ algo="p.ml" init="{root=0;v=3}" ]
  p3 [ algo="p.ml" init="{root=0;v=3}" ]
  p4 [ algo="p.ml" init="{root=0;v=2}" ]
  p5 [ algo="p.ml" init="{root=0;v=2}" ]
  p6 [ algo="p.ml" init="{root=0;v=1}" ]
  p7 [ algo="p.ml" init="{root=0;v=1}" ]
  p8 [ algo="p.ml" init="{root=0;v=0}" ]
  root -> p2 -> p3 -> p4 -> p5 -> p6 -> p7 -> p8 -> root
}
```

- dot attributes
  - name-value pairs that can be ignored (as C #pragmas)
  - node attributes: algo, init
  - graph attributes: global simulation parameters
graph g {
    graph [n=16]
    p0 [algo="p.ml" init="0"] p0 -- p1 -- p2 -- p3 -- p7
    p1 [algo="p.ml" init="17"] p0 -- p4 -- p5 -- p6
    p2 [algo="p.ml" init="18"] p11 -- p15
    p3 [algo="p.ml" init="19"] p1 -- p5 -- p9
    p4 [algo="p.ml" init="17"] p10 -- p11 -- p7
    p5 [algo="p.ml" init="18"] p10 -- p14 -- p15
    p6 [algo="p.ml" init="19"] p10 -- p6
    p7 [algo="p.ml" init="20"] p10 -- p9
    p8 [algo="p.ml" init="18"] p12 -- p13 -- p14
    p9 [algo="p.ml" init="19"] p12 -- p8 -- p9
    p10 [algo="p.ml" init="20"] p13 -- p9
    p11 [algo="p.ml" init="21"] p2 -- p6 -- p7
    p12 [algo="p.ml" init="19"] p4 -- p8
    p13 [algo="p.ml" init="20"]
    p14 [algo="p.ml" init="21"]
    p15 [algo="p.ml" init="22"]
}
Algorithm Programming Interface (1/3): to provide

- 42 straightforward loc of Ocaml Interface (mli) file
Algorithm Programming Interface (1/3): to provide

- 42 straightforward loc of Ocaml Interface (mli) file
- The Local State Type is **polymorphic**

```plaintext
type 's neighbor (* bool, int, float, array, struct, etc. *)
```
Algorithm Programming Interface (1/3): to provide

- 42 straightforward loc of Ocaml Interface (mli) file
- The Local State Type is polymorphic

```ocaml
type 's neighbor (* bool, int, float, array, struct, etc. *)
```

- To implement an algorithm, one need to define 3 things:
  1. an enable function, which encodes the guards of the algorithm

```ocaml
type 's enable_fun = 's -> 's neighbor list -> action list
```
Algorithm Programming Interface (1/3): to provide

- 42 straightforward loc of Ocaml Interface (mli) file
- The Local State Type is polymorphic

type 's neighbor (* bool, int, float, array, struct, etc. *)

to implement an algorithm, one need to define 3 things:
  1. an enable function, which encodes the guards of the algorithm
  2. a step function, that triggers enabled actions

```ocaml
  type 's enable_fun = 's -> 's neighbor list -> action list
  type 's step_fun = 's -> 's neighbor list -> action -> 's
```
Algorithm Programming Interface (1/3): to provide

- 42 straightforward loc of Ocaml Interface (mli) file
- The Local State Type is polymorphic

```ocaml
type 's neighbor (* bool, int, float, array, struct, etc. *)
```

- To implement an algorithm, one need to define 3 things:
  1. an **enable** function, which encodes the **guards** of the algorithm
  2. a **step** function, that **triggers** enabled actions
  3. a state **initialization** function

  - used if no initial value is provided in the DOT file
  - can do nothing: random values will be chosen

```ocaml
type 's enable_fun = 's -> 's neighbor list -> action list
type 's step_fun = 's -> 's neighbor list -> action -> 's
type 's state_init_fun = int -> string -> 's
```
Algorithm Programming Interface (2/3): can be used

Each node can access to information concerning neighbors...

```haskell
val state : 's neighbor -> 's
val reply : 's neighbor -> int (* canal number *)
```
Algorithm Programming Interface (2/3): can be used

Each node can access to information concerning neighbors...

```ocaml
val state : 's neighbor -> 's
val reply : 's neighbor -> int (* canal number *)
```

... or topology

```ocaml
val card: unit -> int
val links_number : unit -> int
val diameter: unit -> int
val min_degree : unit -> int
val mean_degree : unit -> float
val max_degree: unit -> int
val is_cyclic: unit -> bool
val is_connected : unit -> bool
val is_tree : unit -> bool
...
val get_graph_attribute : string -> string
```
Algorithm Programming Interface (3/3): registration

```ocaml
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}
```

```ocaml
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  copy_state: 's -> 's;
  state_of_string: (string -> 's) option;
  actions : action list; (* mandatory for test oracles *)
  legitimate_function : 's legitimate_fun option;
  potential_function: 's potential_fun option;
  fault_function : 's fault_fun option;
}
```

```ocaml
val register : 's to_register -> unit
```

nb: can be generated by following some naming conventions

sasa -reg a_dot_file.dot
Algorithm Programming Interface (3/3): registration

```ocaml
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  copy_state: 's -> 's;
}```
Algorithm Programming Interface (3/3): registration

```ocaml
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  copy_state: 's -> 's;
  state_of_string: (string -> 's) option;
}
```

register : 's to_register -> unit

nb: can be generated by following some naming conventions

sasa -reg a_dot_file.dot
Algorithm Programming Interface (3/3): registration

```ocaml
type 's algo_to_register = {
algo_id : string;
init_state: 's state_init_fun;
enab : 's enable_fun;
step : 's step_fun;
}

type 's to_register = {
algo : 's algo_to_register list;
state_to_string: 's -> string;
copy_state: 's -> 's;
state_of_string: (string -> 's) option;
actions : action list; (* mandatory for test oracles *)
}
```

val register : 's to_register -> unit

nb: can be generated by following some naming conventions

sasa -reg a_dot_file.dot
Algorithm Programming Interface (3/3): registration

```ml
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}

val register : 's to_register -> unit
```

Erwan Jahier

A Tutorial on SASA

November 12, 2020
Algorithm Programming Interface (3/3): registration

```ocaml
let type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
};

let type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  copy_state: 's -> 's;
  state_of_string: (string -> 's) option;
  actions : action list; (* mandatory for test oracles *)
  legitimate_function : 's legitimate_fun option;
  potential_function: 's potential_fun option;
};

let register : 's to_register -> unit
```
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}

val register : 's to_register -> unit

val sasa -reg a_dot_file.dot

Erwan Jahier
A Tutorial on SASA
November 12, 2020
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  copy_state: 's -> 's;
  state_of_string: (string -> 's) option;
  actions : action list; (* mandatory for test oracles *)
  legitimate_function : 's legitimate_fun option;
  potential_function: 's potential_fun option;
  fault_function : 's fault_fun option
}
val register : 's to_register -> unit
Algorithm Programming Interface (3/3): registration

```ocaml
type 's algo_to_register = {
  algo_id : string;
  init_state: 's state_init_fun;
  enab : 's enable_fun;
  step : 's step_fun;
}

val register : 's to_register -> unit

val sasa -reg a_dot_file.dot
```

nb: can be generated by following some naming conventions
The SASA Core Simulator Architecture

- Algorithm [.ml files]
  - Reads
  - Generates [ocaml]
  - Generates loads

- Dynamic Library [.cmxs files] [.ml files]
  - Loads
  - Generates [sasa]

- Network Topology [.dot file]
  - Reads

- Simulation Data [.rif file]
  - Generates
Dijkstra’s Token Ring For Root (1/2)

- **Parameters:**
  - \( p.Pred \) : the predecessor of \( p \) in the ring
  - \( k \) : a positive integer

- **Local Variable:**
  - \( p.v \in \{0,...,k-1\} \)

- **Action:**
  - \( T :: p.v = p.Pred.v \leftrightarrow\)
  - \( p.v \leftarrow (p.v + 1) \mod k \)

```ocaml
open Algo
let k = 42
let init_state _ _ = Random.int k
let enable_f e nl =
    let pred = List.hd nl in
    if e = state pred then ["a"] else []
let step_f e nl _ = (e + 1) mod k
```
Dijkstra’s Token Ring For each Non-Root (2/2)

- **Parameters:**
  - $p.Pred$: the predecessor of $p$ in the ring
  - $k$: a positive integer

- **Local Variable:**
  - $p.v \in \{0, \ldots, k-1\}$

- **Action:**
  - $T :: p.v \neq p.Pred.v \leftrightarrow p.v \leftarrow p.Pred.v$

---

```ocaml
open Algo
let k = 42
let init_state _ _ = Random.int k
let enable_f e nl =
  if e<>state (List.hd nl) then ["a"]
  else []
let step_f e nl a = state (List.hd nl)
```

---

```
cd test/dijkstra-ring
make ring.cmxs
sasa ring.dot
rdbgui4sasa -sut "sasa ring.dot"
```

---

Erwan Jahier
A Tutorial on SASA
November 12, 2020
Dijkstra’s Token Ring For each Non-Root (2/2)

- **Parameters:**
  - `p.Pred`: the predecessor of `p` in the ring
  - `k`: a positive integer

- **Local Variable:**
  - `p.v ∈ {0, ..., k − 1}`

- **Action:**

```ocaml
open Algo
let k = 42
let init_state _ _ = Random.int k
let enable_f e nl =
  if e<>state (List.hd nl) then ["a"]
  else []
let step_f e nl a = state (List.hd nl)

cd test/dijkstra-ring
make ring.cmxs
sasa ring.dot
```
Dijkstra’s Token Ring For each Non-Root (2/2)

- **Parameters:**
  - \( p.\text{Pred} \) : the predecessor of \( p \) in the ring
  - \( k \) : a positive integer

- **Local Variable:**
  - \( p.v \in \{0, ..., k - 1\} \)

- **Action:**
  - \( T :: p.v \neq p.\text{Pred}.v \leftrightarrow p.v \leftarrow p.\text{Pred}.v \)

```
open Algo
let k = 42
let init_state _ _ = Random.int k
let enable_f e nl =
  if e<>state (List.hd nl) then ["a"]
  else []
let step_f e nl a = state (List.hd nl)
cd test/dijkstra-ring
make ring.cmxs
sasa ring.dot
rdbgui4sasa -sut "sasa ring.dot"
```
Graph Coloring

- Parameters:
  \( p.N \) : the set of p's neighbors ;
  \( k \) : an integer such that \( k \geq \Delta \)

- Local Variable:
  \( p.c \in \{0,...,k\} \) holds the color of p

- Macros:
  \( Used(p) = \{ q.c : q \in p.N \} \)
  \( Free(p) = \{0,...,k\} \setminus Used(p) \)

- Predicate:
  \( Conflict(p) = \exists q \in p.N : q.c = p.c \)

- Action:
  Color \( \leadsto p.c \leftarrow \text{min}(Free(p)) \)

```ocaml
open Algo
let k=3
let init_state _ _ = Random.int k
let neighbors_vals nl = List.map (fun n -> state n) nl
let confl v nl = List.mem v (neighbors_vals nl)
let free nl =
  let confl1 = List.sort_uniq compare (neighbors_vals nl) in
  let rec aux free confl i =
    if i > k then free else
      (match confl with
        | x::tail ->
          if x=i then aux free tail (i+1)
          else aux (i::free) confl (i+1)
        | [] -> aux (i::free) confl (i+1)
      )
  in
  List.rev (aux [] confl1 0)
let enable_f e nl=if (confl e nl) then ["conflict"] else []
let step_f e nl a = if free nl = [] then e else List.hd f
let actions = Some ["conflict"]
```

cd test/coloring
rdbg -sut "sasa grid4.dot -lcd"
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Classical and simple SS algorithms:
1. Dijkstra’s Token Ring
2. Coloring Algo
3. Synchronous Unison
4. A-Synchronous Unison
5. BFS spanning tree
6. DFS spanning tree
Dijkstra’s Token Ring (1/2)

Get a unique Token that Circulates in rooted unidirected ring For Root process

- **Parameters:**
  - $p.Pred$ : the predecessor of $p$ in the ring
  - $k$ : a positive integer
Dijkstra’s Token Ring (1/2)

Get a unique Token that Circulates in rooted unidirected ring For Root process

- **Parameters:**
  - \( p.Pred \) : the predecessor of \( p \) in the ring
  - \( k \) : a positive integer

- **Local Variable:**
  - \( p.v \in \{0, \ldots, k-1\} \)
Dijkstra's Token Ring (1/2)

Get a unique Token that Circulates in **rooted unidirected ring** For Root process

- **Parameters:**
  - \( p.Pred \) : the predecessor of \( p \) in the ring
  - \( k \) : a positive integer

- **Local Variable:**
  - \( p.v \in \{0, ..., k - 1\} \)

- **Action:**
  - \( T :: p.v = p.Pred.v \leftrightarrow p.v \leftarrow (p.v + 1) \pmod k \)
Dijkstra’s Token Ring (2/2)

For each Non-Root process

- **Parameters:**
  - \( p.Pred \): the predecessor of \( p \) in the ring
  - \( k \): a positive integer

```cd test/dijkstra; rdbg -sut "sasa ring.dot –distributed-demon"```
Dijkstra’s Token Ring (2/2)

For each Non-Root process

- **Parameters:**
  - $p.\text{Pred}$: the predecessor of $p$ in the ring
  - $k$: a positive integer

- **Local Variable:**
  - $p.v \in \{0, \ldots, k-1\}$
For each Non-Root process

- **Parameters:**
  - $p.Pred$ : the predecessor of $p$ in the ring
  - $k$ : a positive integer

- **Local Variable:**
  - $p.v \in \{0, \ldots, k - 1\}$

- **Action:**
  - $T :: p.v \neq p.Pred.v \leftrightarrow p.v \leftarrow p.Pred.v$
Dijkstra’s Token Ring (2/2)

For each Non-Root process

- **Parameters:**
  - $p.Pred$ : the predecessor of $p$ in the ring
  - $k$ : a positive integer

- **Local Variable:**
  - $p.v \in \{0, \ldots, k - 1\}$

- **Action:**
  - $T :: p.v \neq p.Pred.v \Rightarrow p.v \leftarrow p.Pred.v$

```
cd test/dijkstra; rdbg -sut "sasa ring.dot -distributed-demon"
```
Graph Coloring

For each process $p$

- **Parameters:**
  - $p.N$ : the set of $p$’s neighbors
  - $k$ : an integer such that $k \geq \Delta$

- **Local Variable:**
  - $p.c \in \{0, \ldots, k\}$ holds the color of $p$
Graph Coloring

For each process \( p \)

- **Parameters:**
  - \( p.N \): the set of \( p \)'s neighbors
  - \( k \): an integer such that \( k \geq \Delta \)

- **Local Variable:**
  - \( p.c \in \{0, \ldots, k\} \) holds the color of \( p \)

- **Macros:**
  - \( Used(p) = \{q.c : q \in p.N\} \)
  - \( Free(p) = \{0, \ldots, k\} \setminus Used(p) \)

- **Predicate:**
  - \( Conflict(p) = \exists q \in p.N : q.c = p.c \)

- **Action:**
  - \( \text{Color} :: Conflict(p) \rightarrow p.c \leftarrow \min(Free(p)) \)
Graph Coloring

For each process $p$

- **Parameters:**
  - $p.N$ : the set of $p$’s neighbors
  - $k$ : an integer such that $k \geq \Delta$

- **Local Variable:**
  - $p.c \in \{0, ..., k\}$ holds the color of $p$

- **Macros:**
  - $Used(p) = \{q.c : q \in p.N\}$
  - $Free(p) = \{0, ..., k\} \setminus Used(p)$

- **Predicate:**
  - $Conflict(p) = \exists q \in p.N : q.c = p.c$

- **Action:**
  - $Color :: Conflict(p) \leftrightarrow p.c \leftarrow min(Free(p))$
Graph Coloring

For each process \( p \)

- **Parameters:**
  - \( p.N \): the set of \( p \)'s neighbors
  - \( k \): an integer such that \( k \geq \Delta \)

- **Local Variable:**
  - \( p.c \in \{0, \ldots, k\} \) holds the color of \( p \)

- **Macros:**
  - \( \text{Used}(p) = \{q.c : q \in p.N\} \)
  - \( \text{Free}(p) = \{0, \ldots, k\} \setminus \text{Used}(p) \)

- **Predicate:**
  - \( \text{Conflict}(p) = \exists q \in p.N : q.c = p.c \)

- **Action:**
  - \( \text{Color} :: \text{Conflict}(p) \hookrightarrow p.c \hookleftarrow \text{min}(\text{Free}(p)) \)

```
cd test/coloring; rdbgui4sasa -sut "sasa grid4.dot -locally-central-demon"
```
Synchronous unison

For each process $p$

- **Parameters:**
  - $p.N$: the set of $p$'s neighbors
  - $m$: an integer such that $m \geq \max(2, 2 \times D - 1)$

- **Local Variable:**
  - $p.c \in \{0, \ldots, m - 1\}$ holds the clock of $p$

- **Macro:**
  - $NewClockValue(p) = (\min(\{q.c : q \in p.N\} \lor \{p.c\}) + 1 \mod m$

- **Action:**
  - $\text{Incr} :: p.c \neq NewClockValue(p) \rightarrow p.c \leftarrow NewClockvalue(p)$

```
cd test/unison; rdbgui4sasa -sut "sasa ring.dot
-synchronous-demon"
```
A-Synchronous Unison

For each process $p$

- **Parameters:**
  - $p.N$: the set of $p$'s neighbors
  - $k$: an integer such that $k \geq n^2$

- **Local Variable:**
  - $p.c \in \{0, ..., k - 1\}$ holds the clock of $p$

- **Predicate:**
  - $behind(a, b) = \left( (b.c - a.c) \mod k \right) \leq n$

- **Actions:**
  - $I :: \forall q \in p.N, behind(p, q) \hookrightarrow p.c \leftarrow (p.c + 1) \mod k$
  - $R :: p.c \neq 0 \land (\exists q \in p.N, \neg behind(p, q) \land \neg behind(q, p)) \hookrightarrow p.c \leftarrow 0$

```
cd test/async-unison; rdbgui4sasa -sut "sasa ring.dot -central-demon"
```
For the Root process

• Parameters:
  ▶ \textit{root}.\textit{N} : the set of root’s neighbors
  ▶ \textit{D} : an integer such that \( D \geq \mathcal{D} \)
For the Root process

- **Parameters:**
  - $root.N$ : the set of root’s neighbors
  - $D$ : an integer such that $D \geq D$

- **Local Variable:**
  - $root.d \in \{0, ..., D\}$ holds the distance to the root
BFS Spanning tree (1/2)

For the **Root** process

- **Parameters:**
  - \( root.N \) : the set of root’s neighbors
  - \( D \) : an integer such that \( D \geq 0 \)

- **Local Variable:**
  - \( root.d \in \{0,\ldots,D\} \) holds the distance to the root

- **Action:**
  - \( CD :: root.d \neq 0 \rightarrow root.d \leftarrow 0 \)
BFS Spanning tree (2/2)

For each non-Root process p

- **Parameters:**
  - $p.N$: the set of p’s neighbors
  - $D$: an integer such that $D \geq \mathcal{D}$

- **Variables:**
  - $p.d \in \{0, ..., D\}$ holds the distance to the root
  - $p.par \in p.N$ holds the parent pointer of p

- **Macros:**
  - $Dist(p) = \min\{q.d : q \in p.N\}$
  - $DistOK(p) = p.d - 1 = \min\{q.d : q \in p.N\}$

- **Actions:**
  - $CD :: p.d \neq Dist(p) \rightarrow p.d \leftarrow Dist(p)$
  - $CP ::$
    - $DistOK(p) \lor p.par.d \neq p.d - 1 \rightarrow p.par \leftarrow q \in p : Ns.t.q(d) = p(d) - 1$

cd test/bfs-spanning-tree; rdbgui4sasa -sut "sasa fig51.dot -distributed-demon"
For the Root process

- **Parameters:**
  - $p.N$ : the set of root’s neighbors
  - $\delta$ : an integer $\geq n$

- **Local Variable:**
  - $p.path$ : an array integers of size $\delta$

- **Action:**
  - Path :: $p.path \neq [] \leftrightarrow p.pathgets[]$
DFS Spanning Tree [Collin-Dolev-94] (2/2)

For each Non-Root process

- **Parameters:**
  - \( p.N \) : the set of process’s neighbors
  - \( \delta \): a integer \( \geq n \)

- **Local Variables:**
  - \( p.par \in \{0,...,|p.N|−1\} \) the parent of the process
  - \( p.path \) : an array integers of size \( \delta \)

- **Macros:**
  - \( \text{ComputePar}(p.N) = [...]\)
  - \( \text{ComputePath}(p.N) = [...]\)

- **Actions:**
  - \( \text{Par} :: p.par \neq \text{ComputePar}(p.N) \leftrightarrow p.pargetsComputePar(p.N) \)
  - \( \text{Path} :: p.path \neq \text{ComputePath}(p.N) \leftrightarrow p.pathgetsComputePath(p.N) \)

cd test/dfs-list; rdbgui4sasa -sut "sasa g.dot"
More Coding Exercises

- **spanning Tree Constructions** published by
  - Chen, Yu, and Huang in 1991
  - Huang and Chen in 1992
  - Kosowski and Kuszner in 2006
More Coding Exercises

- **spanning Tree Constructions** published by
  - Chen, Yu, and Huang in 1991
  - Huang and Chen in 1992
  - Kosowski and Kuszner in 2006

- and also
  - an **ASM version of a Depth First Search** algorithm published by Collin and Dolev in 1994
  - a **k-clustering** algorithm published by Karine Altisen, Pierre Corbineau, and Stéphane Devismes in 2017

---

Erwan Jahier  A Tutorial on SASA  November 12, 2020
More Coding Exercises

• spanning Tree Constructions published by
  ▶ Chen, Yu, and Huang in 1991
  ▶ Huang and Chen in 1992
  ▶ Kosowski and Kuszner in 2006

• and also
  ▶ an ASM version of a Depth First Search algorithm published by Collin and Dolev in 1994
  ▶ a k-clustering algorithm published by Karine Altisen, Pierre Corbineau, and Stéphane Devismes in 2017
  ▶ aDisconnected Components Detection and Rooted Shortest-Path Tree Maintenance algorithm published by Stéphane Devismes, David Ilcinkas, Colette Johnen in 2017
  ▶ several graph coloring algorithms
More Coding Exercises

• spanning Tree Constructions published by
  ▶ Chen, Yu, and Huang in 1991
  ▶ Huang and Chen in 1992
  ▶ Kosowski and Kuszner in 2006

• and also
  ▶ an ASM version of a Depth First Search algorithm published by Collin and Dolev in 1994
  ▶ a k-clustering algorithm published by Karine Altisen, Pierre Corbineau, and Stéphane Devismes in 2017
  ▶ a Disconnected Components Detection and Rooted Shortest-Path Tree Maintenance algorithm published by Stéphane Devismes, David Ilcinkas, Colette Johnen in 2017
  ▶ several graph coloring algorithms

• cf the gitlab repo for the full refs and possible implementations
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. **Performance Evaluation**
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
We have implemented the following self-stabilizing algorithms:

- **[ASY]** solves unison in any network, under any daemon.
- **[SYN]** solves the unison problem in any network, under a synchronous daemon.
- **[DTR]** solves the token circulation problem through a rooted unidirected ring, under any daemon.
- **[BFS]** builds a BFS spanning tree in any network using a distributed daemon.
- **[DFS]** builds a DFS spanning tree in any network using a distributed daemon.
- **[COL]** solves the coloring algorithm in any network, under a locally central daemon.
Performance Evaluation: Measurements

- 2 Square Grids
  - `grid.dot`: 10 × 10 nodes, 180 links;
  - `biggrid.dot`: 100 × 100 nodes, 19800 links;

- 2 Random Graphs built using the Erdös-Rényi model
  - `ER.dot`: 256 nodes, 9811 links, average degree 76;
  - `bigER.dot`: 2000 nodes, 600253 links, average degree 600.

<table>
<thead>
<tr>
<th></th>
<th>grid.dot</th>
<th>ER.dot</th>
<th>biggrid.dot</th>
<th>bigER.dot</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>0.2 ms</td>
<td>10.6 ms</td>
<td>2.04 s</td>
<td>3.03 s</td>
</tr>
<tr>
<td>DFS-l</td>
<td>1 ms</td>
<td>144.7 ms</td>
<td>2.57 s</td>
<td>15.83 s</td>
</tr>
<tr>
<td>DFS-a</td>
<td>0.5 ms</td>
<td>94.3 ms</td>
<td>7.64 s</td>
<td>86.93 s</td>
</tr>
<tr>
<td>COL</td>
<td>0 ms</td>
<td>35.8 ms</td>
<td>27.93 s</td>
<td>16.81 s</td>
</tr>
<tr>
<td>SYN</td>
<td>0.3 ms</td>
<td>10.9 ms</td>
<td>887.05 s</td>
<td>13.58 s</td>
</tr>
<tr>
<td>ASY</td>
<td>0.1 ms</td>
<td>4.5 ms</td>
<td>0.03 s</td>
<td>2.82 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Mem</th>
<th>Mem</th>
<th>Mem</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>13 MB</td>
<td>49 MB</td>
<td>83 MB</td>
<td>1062 MB</td>
</tr>
<tr>
<td>DFS-l</td>
<td>44 MB</td>
<td>63 MB</td>
<td>92 MB</td>
<td>1062 MB</td>
</tr>
<tr>
<td>DFS-a</td>
<td>39 MB</td>
<td>170 MB</td>
<td>6642 MB</td>
<td>29945 MB</td>
</tr>
<tr>
<td>COL</td>
<td>7 MB</td>
<td>63 MB</td>
<td>75 MB</td>
<td>1083 MB</td>
</tr>
<tr>
<td>SYN</td>
<td>38 MB</td>
<td>63 MB</td>
<td>874 MB</td>
<td>1099 MB</td>
</tr>
<tr>
<td>ASY</td>
<td>38 MB</td>
<td>63 MB</td>
<td>83 MB</td>
<td>1115 MB</td>
</tr>
</tbody>
</table>

- Time/step = user+system time / | simulation steps |
- Mem = “Maximum resident set size” of GNU time
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Simulation Campaigns

The distribution contains scripts to support **SIMulation CAMpaigns**:

- the `sasa/tools/simca/` directory of the git repository
  - Ocaml scripts to automate the running of simulations
  - R scripts to produce graphical outlines
- cf also, in the set of sasa tutorials (*), the ones named:
  - “Simulation Campaigns with sasa”
  - “Comparing Spanning Trees Construction”

Simulation Campaigns

Scripts in `sasa/tools/simca/` automates Test Campaigns

```
simca Scripts Produces

- a list of algorithms `al`
- a list of daemons `dl`
- a list of graphs `gl` (can be generated)
- a precision percentage `p` (0.01 by default)
- a positive int `max_simu_nb` (10000 by default)
- a positive int `timeout` (10000 by default)

|al| . |dl| . |gl| estimations for (move, step, round)

1 estimation <- `{simu(a,d,g)}` simulations run as long as
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
Simulation Campaigns

- 3 coloring algorithms
- 3 daemons - synchronous - locally central - distributed
- 30 graphs - 10 rings (500, 1000, ..., 5000) - 10 cliques (30, 60, ..., 300) - 10 ER (30, 60, ..., 300)
- a precision of 1%
- make --jobs 30

simca

- 270 estimations (rounds, steps, moves)
- 180 818 simulations
- 4 hours (94 hours of Wall-clock cumulative time)
- 669 per estimation on average
- 3116 at worst
Simulation Campaigns

- 3 coloring algorithms
- 3 daemons
  - synchronous
  - locally central
  - distributed
- 30 graphs
  - 10 rings (500, 1000, ..., 5000)
  - 10 cliques (30, 60, ..., 300)
  - 10 ER (30, 60, ..., 300)
- a precision of 1%
- make --jobs 30

simca

- 270 estimations (rounds, steps, moves)
- 180 818 simulations
  - 669 per estimation on average
  - 3116 at worst
  - 4 hours (94 hours of Wall-clock cumulative time)
Some of the graphics generated on Rings

Compare Algorithms on various Daemons Numbers on Ring

<table>
<thead>
<tr>
<th>Distributed</th>
<th>Locally Central</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image_url" alt="Graph" /></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithms
- Always the Biggest
- Smallest When Triggered
- Uniform When Triggered
Some of the graphics generated on Rings

Comparison of Daemons on various Algorithms for Numbers on Ring

- Always the Biggest
- Smallest When Triggered
- Uniform When Triggered

Nodes Number

0 2500 5000 7500 10000

Move/Step/Round Numbers

0 5 10 15 20 25 30 35 40

Daemons
- Distributed
- Locally Central
- Synchronous

Erwan Jahier
A Tutorial on SASA
November 12, 2020
Simulation Campaigns: results on ER Graphs

Compare Algorithms on various Daemons Numbers on Er

<table>
<thead>
<tr>
<th>Move/Step/Round Numbers</th>
<th>Distributed</th>
<th>Locally Central</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes Number</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Algorithms
- Always the Biggest
- Smallest When Triggered
- Uniform When Triggered

Erwan Jahier
A Tutorial on SASA
November 12, 2020
Simulation Campaigns: results on ER Graphs

Compare Daemons on various Algorithms Numbers on Er

<table>
<thead>
<tr>
<th>Daemons</th>
<th>Always the Biggest</th>
<th>Smallest When Triggered</th>
<th>Uniform When Triggered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move/Step/Round Numbers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodes Number</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Daemons
- Distributed
- Locally Central
- Synchronous

Erwan Jahier
A Tutorial on SASA
November 12, 2020
Some of the graphics generated on Cliques

Compare Algorithms on various Daemons Numbers on Clique

- Distributed
- Locally Central
- Synchronous

Algorithms:
- Always the Biggest
- Smallest When Triggered
- Uniform When Triggered

Nodes Number vs. Move/Step/Round Numbers
Some of the graphics generated on Cliques

Compare Daemons on various Algorithms Numbers on Clique

<table>
<thead>
<tr>
<th>Daemons</th>
<th>Nodes Number</th>
<th>Move/Step/Round Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always the Biggest</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smallest When Triggered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uniform When Triggered</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Daemons: Distributed, Locally Central, Synchronous

Erwan Jahier  
A Tutorial on SASA  
November 12, 2020
Ditto on 6 **Spanning Trees Construction Algorithms**

Compare Daemons on various Algorithms Numbers on Er

<table>
<thead>
<tr>
<th>BFS book</th>
<th>CYH91</th>
<th>DFS book</th>
<th>HC92</th>
<th>KK06–algo1</th>
<th>KK06–algo2</th>
</tr>
</thead>
<tbody>
<tr>
<td>moves</td>
<td>moves</td>
<td>moves</td>
<td>moves</td>
<td>moves</td>
<td>moves</td>
</tr>
<tr>
<td>rounds</td>
<td>rounds</td>
<td>rounds</td>
<td>rounds</td>
<td>rounds</td>
<td>rounds</td>
</tr>
<tr>
<td>steps</td>
<td>steps</td>
<td>steps</td>
<td>steps</td>
<td>steps</td>
<td>steps</td>
</tr>
</tbody>
</table>

Nodes Number

Move/Step/Round Numbers

Daemons
- Central
- Distributed

Erwan Jahier

A Tutorial on SASA

November 12, 2020
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Algorithms in the ASM viewed as Reactive programs

loop:

1. Reads neighbors vars
2. Computes $\pi_{enab}$
3. Chooses $\pi_{act}$ (Daemon)
4. Computes states ($\pi_{act}$)
Algorithms in the ASM viewed as Reactive programs

loop:

1. Reads neighbors vars
2. Computes \( \text{pi}_\text{enab} \)
3. Chooses \( \text{pi}_\text{act} \) (Daemon)
4. Computes states (\( \text{pi}_\text{act} \))

loop:

- 4. \textbf{Init} -> Computes states (\( \text{pi}_\text{act} \))
- 1. Reads neighbors vars
- 2. Computes \( \text{pi}_\text{enab} \)
- 3. Chooses \( \text{pi}_\text{act} \) (Daemon)
Algorithms in the ASM viewed as Reactive programs

loop:
1. Reads neighbors vars
2. Computes $\pi_{enab}$
3. Chooses $\pi_{act}$ (Daemon)
4. Computes states ($\pi_{act}$)

loop:
• 4. Init -> Computes states ($\pi_{act}$)
• 1. Reads neighbors vars
• 2. Computes $\pi_{enab}$
• 3. Chooses $\pi_{act}$ (Daemon)
The LURETTE dataflow

Figure: The LURETTE dataflow schema
Figure: The RDBG dataflow schema
Figure: The RDBG dataflow schema
Lurette and Test Oracles

- All Book theorems formalized in Lustre
- Heavy use Lustre V6 genericity to write Topology Independant Oracles

```plaintext
include "../lustre/oracle_utils.lus"

node theorem_5_18<<const an : int; const pn: int>> (Enab, Acti: bool^an^pn)
returns (res:bool);
var
  Round:bool;
  RoundNb:int;
  Silent:bool;
let
  Round = round <<an,pn>>(Enab,Acti);
  RoundNb = count(Round);
  Silent = silent<<an,pn>>(Enab);
  res = (RoundNb >= diameter+2) => Silent ; -- from theorem 5.18 page 57

tel

node bfs_spanning_tree_oracle<<const an:int; const pn:int>> (Enab, Acti: bool^an^pn)
returns (ok:bool);
let
  ok = lemma_5_16 <<an,pn>> (Enab, Acti) and theorem_5_18<<an,pn>> (Enab, Acti);

tel
```
Lurette and Lutin Environments

- Stochastic Reactive Language
- Designed to model Reactive Programs Environments
- Could be used to program custom Daemons with feedback
  - To explore worst cases
  - To simulate Algo that deals with Shared Resources
Lurette and Lutin Environments

- Stochastic Reactive Language
- Designed to model Reactive Programs Environments
- Could be used to program custom Daemons with feedback
  - To explore worst cases
  - To simulate Algo that deals with Shared Resources

```
cd test/dijkstra; rdbg -env "sasa ring.dot -custom-demon"
-sut-nd "lutin ring.lut -n distributed"
```
Synchron’16 (scopes’17)

1. Debug Reactive programs
2. Plugin-based (instrumented runtime): Lustre, Lutin
3. Programmable
   - run: unit -> Event.t
   - next: Event.t -> Event.t
Synchron’16 (scopes’17)

1. Debug Reactive programs

2. Plugin-based (instrumented runtime): Lustre, Lutin

3. Programmable

- run: unit -> Event.t
- next: Event.t -> Event.t

  - Move forward and Backwards (1 slide)
  - Conditional breakpoints (1 line)
  - gdb like Breakpoints (1 slide)
  - Profiling, monitoring, e.g. Computing CFG (~100 loc)
  - Opening an emacs at the current line (10 loc)
  - Debugger Customization
  - etc.

http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/rdbg/README.html
• One can only look at what happens at the interface
• Yet, at lot of thing can be done
  ▶ move forward or backward from step to step, or rounds to rounds (40 loc)
  ▶ Display the graph decorated (200 loc)
    • with enabled/activated status
    • local state values
RDBG and SASA

- One can only look at what happens at the interface
- Yet, a lot of things can be done
  - move forward or backward from step to step, or rounds to rounds (40 loc)
  - Display the graph decorated (200 loc)
    - with enabled/activated status
    - local state values

```bash
cd test/async-unison; rdbg -sut "sasa grid4.dot-central-demon"
```
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Install SASA

Several possibilities

- git
- opam
- docker
- Virtual Machine
Plan

1. (Self-stabilizing Algorithms in the Atomic-State Model)
2. Motivations
3. SASA
4. Coding Exercises
5. Performance Evaluation
6. Simulation Campaigns
7. Integration with Synchronous tools
8. Install SASA
9. Conclusion
Conclusion

• An open-source SimulAtor of Self-stabilizing Algorithms
• written using the atomic-state model (the most commonly used in Self-Stab)
Conclusion

- An open-source SimulAtor of **Self-stabilizing Algorithms**
- written using the **atomic-state** model (the most commonly used in Self-Stab)
- Rely on **existing** tools as much as possible
  - `dot` for Graphs
  - `ocaml` for programming local algorithms
  - *Synchrone (Verimag)* Team Tools for simulation
Conclusion

• An open-source SimulAtor of Self-stabilizing Algorithms
• written using the atomic-state model (the most commonly used in Self-Stab)
• Rely on existing tools as much as possible
  ▶ dot for Graphs
  ▶ ocaml for programming local algorithms
  ▶ Synchrone (Verimag) Team Tools for simulation
• Aims at being easy to use and install

https://verimag.gricad-pages.univ-grenoble-alpes.fr/synchrone/sasa