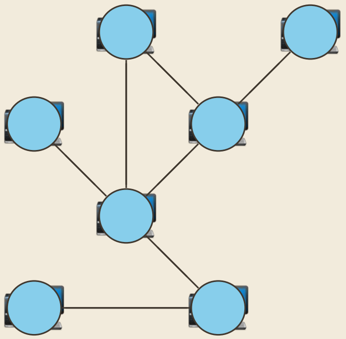
Outline

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Plan

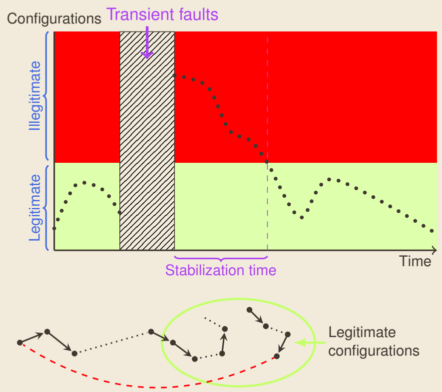
1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Distributed Systems Algorithms



- Process
  - Autonomous
  - Interconnected
- Hypotheses
  - Connected
  - Bidirectional
  - Identified
- Expected Property
  - Fault-tolerance

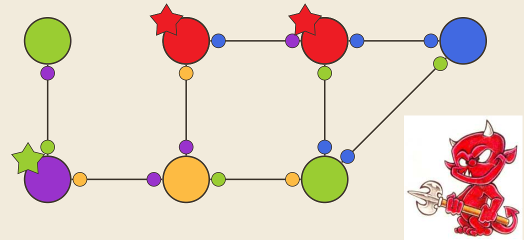
Self-Stabilizing Algorithms



Atomic (Synchronous?) State Model

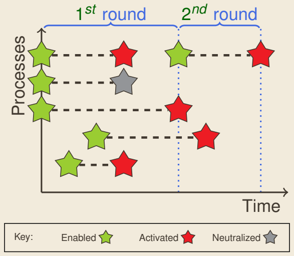
Performing an Atomic Step consists in:

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



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):
  - Used in the Self-Stabilizing Algorithms community
  - Higher-level: atomic instantaneous communications
  - General Algorithms transformations into MPM methods exist

---

Some Classical Examples

- Dijkstra's Token Ring
- Coloring Algo
- Synchronous Unison
- A-Synchronous Unison
- BFS spanning tree
- DFS spanning tree [Collin-Dolex-94]

Introduction to Distributed Self-Stabilizing Algorithms

Karine Altisen
Stéphane Devismes
Swan Dubois
Franck Petit

"Introduction to Distributed Self-Stabilizing Algorithms" Altisen, Devismes, Dubois, Petit 2019.

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 :: $p.v = p.Pred.v \hookrightarrow p.v \leftarrow (p.v+1) mod\ K$

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
- Local Variable:
  - $p.v \in \{0, ..., K-1\}$
- Action:
  - T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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

Coloring Algo

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) $\hookrightarrow p.c \leftarrow min(Free(p))$

```
cd test/coloring; rdbg -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 $m \geq max(2, 2 \times \mathscr{D} - 1)$
- Local Variable:
  - $p.c \in \{0, ..., m-1\}$ holds the clock of p
- Macro:
  - $NewClockValue(p) = (min(\{q.c : q \in p.N\} \vee \{p.c\}) + 1 mod\ m$
- Action:
  - Incr :: $p.c \neq NewClockValue(p) \hookrightarrow p.c \leftarrow NewClockvalue(p)$

```
cd test/unison; rdbg -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:
  - $behind(a,b) = ((b.c - a.c) \mod K) \leq n$
- Actions:
  - I :: $\forall q \in p.N, behind(p,q) \hookrightarrow p.c \leftarrow (p.c+1) \mod K$
  - R :: $p.c \neq 0 \wedge (\exists q \in p.N, \neg behind(p,q) \wedge \neg behind(q,p)) \hookrightarrow p.c \leftarrow 0$

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

BFS Spanning tree (1/2)

For the Root process

- Parameters:
  - $root.N$: the set of root's neighbors
  - D : an integer such that $D \geq \mathscr{D}$
- Local Variable:
  - $root.d \in \{0, ..., D\}$ holds the distance to the root
- Action:
  - CD :: $root.d \neq 0 \hookrightarrow root.d \leftarrow 0$

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 $D \geq \mathscr{D}$
- 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 :: $p.d \neq Dist(p) \hookrightarrow p.d \leftarrow Dist(p)$
  - CP :: $DistOK(p) \vee p.par.d \neq p.d - 1 \hookrightarrow p.par \leftarrow q \in p : Ns.t.q(d) = p(d) - 1$

```
cd test/bfs; rdbg -sut "sasa fig51.dot -distributed-demon"
```

DFS Spanning Tree (1/2)

For the Root process

- Parameters:
  - $p.N$: the set of root's neighbors
  - δ: a integer $\geq n$
- Local Variable:
  - $p.path$: an array integers of size δ
- Action:
  - Path :: $p.path \neq [] \hookrightarrow p.pathgets[]$

DFS Spanning Tree (2/2)

For each Non-Root process

- Parameters:
  - $p.N$: the set of process's neighbors
  - δ: 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 δ
- Macros:
  - $ComputePar(p.N) = [...]$
  - $ComputePath(p.N) = [...]$
- Actions:
  - Par :: $p.par \neq ComputePar(p.N) \hookrightarrow p.pargetsComputePar(p.N)$
  - Path :: $p.path \neq ComputePath(p.N) \hookrightarrow p.pathgetsComputePath(p.N)$

```
cd test/dfs; rdbg -sut "sasa g.dot"
```

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

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

Simulators Dedicated to Self-Stabilization

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)

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!

Plan

SASA: main features

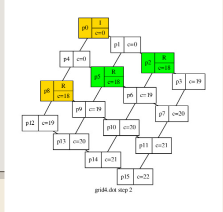
- Batch Simulations
  - Debug Algorithms
  - Perform simulation campaigns,
    - Study the influence of some parameters
    - Evaluate the (average-case) complexity Lower bounds
- Test oracles to formalize expected properties
  - involve the number of steps, moves, or rounds to reach a legitimate configuration (differs from algorithms).
- Daemon can be configured
  - Predefined: synchronous, central, locally central, or distributed
  - Custom daemons: manual or programmed
- Interactive Simulations
  - step by step, or round by round, forward or backward
  - while visualizing the network, the enabled, the activated actions
  - New commands can also be programmed

Defining The Network Topology

- Take advantage of the GraphViz `dot` language
  - Simple syntax
  - Open-source
  - Plenty of visualizers, editors, parsers, exporters
- `dot` attributes
  - name-value pairs that can be ignored (pragmas)
  - node attributes: `algo`, `init`
  - graph attributes: global simulation parameters

A Topology Example: a 4x4 grid

```
graph g {
 graph [n=24]
  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

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

- Users need to define 4 things:
  1. a list of action labels
  2. an enable function, which encodes the guards of the algorithm
  3. a step function, that triggers enabled actions
  4. a state initialization function (used if not provided in the DOT file)

```
type action  = string
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 -> 's
```

Algorithm Programming Interface (2/4)

Each node can get (or not) information on its neighbors:

```
exception Not_available

val state : 's neighbor -> 's
val pid   : 's neighbor -> string
val spid  : 's neighbor -> string
val reply : 's neighbor -> int
val weight: 's neighbor -> int
```

Algorithm Programming Interface (3/4)

Some of the topological information can be accessed:

```
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
```

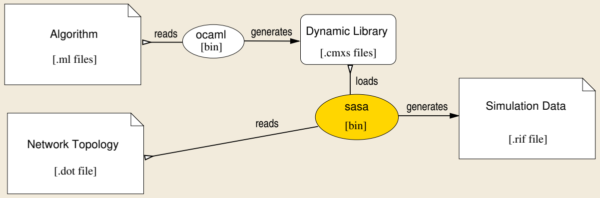
37 straightforward loc

Algorithm Programming Interface (3/4)

Registration

```
type 's algo_to_register = {
  algo_id    : string;
  init_state: int -> 's;
  enab       : 's enable_fun;
  step       : 's step_fun;
  actions    : action list option  }
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  state_of_string: (string -> 's) option;
  copy_state: 's -> 's  }
val register : 's to_register -> unit
```

The SASA Core Simulator Architecture



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 :: $p.v = p.Pred.v \hookrightarrow p.v \leftarrow (p.v+1) mod\ K$

```
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 ["T"] 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:
  $p.v \in \{0,...,K-1\}$
- Action:
  T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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

```
cd test/dijksra; rdbg -sut "sasa
ring.dot -distributed-demon"
```

Coloring Algo

- 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)
  $\hookrightarrow p.c \leftarrow min(Free(p))$

```
open Algo
let k=3
let init_state _ = Random.int k
let neigbhors_vals nl = List.map (fun n -> state n) nl
let confl v nl = List.mem v (neigbhors_vals nl)
let free nl =
 let confl1 = List.sort_uniq compare (neigbhors_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 -locally-central-demon"
```

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

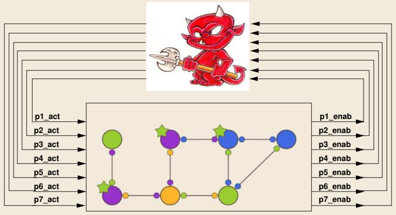
Algorithms in the ASM viewed as Reactive programs

loop:

1. Reads neighbors vars
2. Computes `pi_enab`
3. Chooses `pi_act` (Daemon)
4. Computes states (`pi_act`)

loop:

- 4. Init -> Computes states (`pi_act`)
- 1. Reads neighbors vars
- 2. Computes `pi_enab`
- 3. Chooses `pi_act` (Daemon)



The LURETTE dataflow

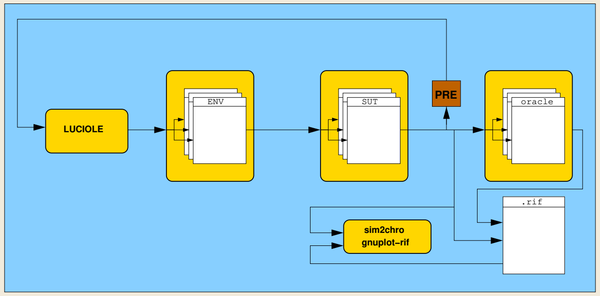


Figure: The LURETTE dataflow schema

RDBG

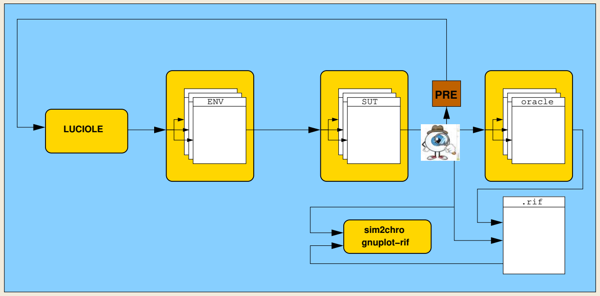


Figure: The RDBG dataflow schema

RDBG

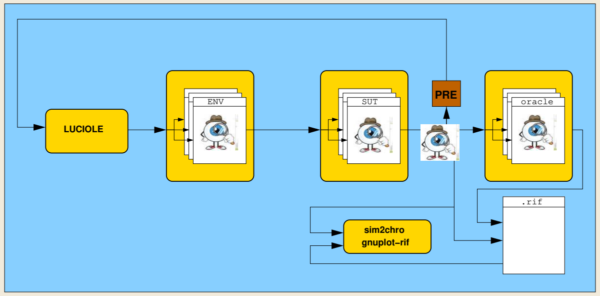


Figure: The RDBG dataflow schema

Lurette and Test Oracles

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

```
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

```
cd test/dijkstra; rdbg -env "sasa ring.dot -custom-demon"
-sut-nd "lutin ring.lut -n distributed"
```

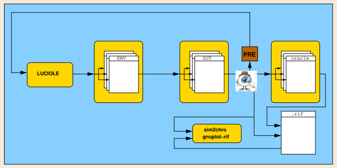
RDBG

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`

RDBG and SASA



- 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

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

Plan

Performance Evaluation: Benchmarks Algorithms

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 d istributed 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.

	`grid.dot`		`ER.dot`		`biggrid.dot`		`bigER.dot`	
	Time/step	Mem	Time/step	Mem	Time/step	Mem	Time/step	Mem
BFS	0.2 ms	13 MB	10.6 ms	49 MB	2.04 s	83 MB	3.03 s	1062 MB
DFS-l	1 ms	44 MB	144.7 ms	63 MB	2.57 s	92 MB	15.83 s	1062 MB
DFS-a	0.5 ms	39 MB	94.3 ms	170 MB	7.64 s	6642 MB	86.93 s	29945 MB
COL	0 ms	7 MB	35.8 ms	63 MB	27.93 s	75 MB	16.81 s	1083 MB
SYN	0.3 ms	38 MB	10.9 ms	63 MB	887.05 s	874 MB	13.58 s	1099 MB
ASY	0.1 ms	38 MB	4.5 ms	63 MB	0.03 s	83 MB	2.82 s	1115 MB

- 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. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Polymorphic versus Variant Type

- An alternative to polymorphism to hold processes local state:

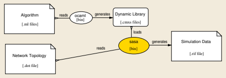
```
type value = I of int | F of float | B of Bool | A of state array | ...
type env = string -> value
```

But:

- What if one need a type that is not in this variant list?
- Variable values need to be set/get in/from the env^t all the time.

```
let step_f c nl a =                let step_f env nl a =
   match a with                       match a with
    | "I" -> modulo (c + 1) k          | "I" ->
    | "R" -> 0                            let c_val = match env_get env "c" with
                                                | I i -> i
                                                | _ -> assert false
                                          in
                                          set_env env "c" (I(modulo ((c_val)+1) k))
                                       | "R" -> set_env env "c" (I 0)
```

Dynamic versus Static Linking



- Dynamic Linking: Pros
  - Easier to use
  - Save Disk space
  - Separation of concerns: user code only depends on a simple API
- Dynamic Linking: Cons
  - Can not be combined gently with Polymorphic values!

Dynamic Type Checking of Polymorphic Nodes

- Dynamic linking in OCAML needs to be done via imperative tables
  - The code to be linked registers functions into tables
  - The main executable reads the tables of functions
- But storing polymorphic values into a mutable data-type is not possible in ML-like languages; one can only store so-called weakly polymorphic values!
- Weak variables can’t escape the scope of a compilation unit

https://ocamlverse.github.io/content/weak_type_variables.html

---

Dynamic Type Checking of Polymorphic Nodes

- Solution: use the (evil) `Obj` module:
  - `Obj.obj:  'a -> t`: to register polymorphic functions into tables
  - `Obj.repr:  t -> 'a`: to retrieve them from the simulation engine
- Using `Obj` breaks type safety: how to prevent users to register functions of different type?

By forcing all functions to be registrated at the same time:

```
type ’s algo_to_register = {
  algo_id : string;
  init_state: int -> ’s;
  enab : ’s enable_fun;
  step : ’s step_fun;
  actions : action list option }
type ’s to_register = {
  algo : ’s algo_to_register list; (* <==== ALL AlGO HAVE THE SAME TYPE! *)
  state_to_string: ’s -> string;
  state_of_string: (string -> ’s) option;
  copy_state: ’s -> ’s }
val register : ’s to_register -> unit
```

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Conclusion

- An open-source SimulAtor of Self-stabilizing Algorithms
- writen 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
- Installation via
  - `docker`
  - `opam`
  - `git`

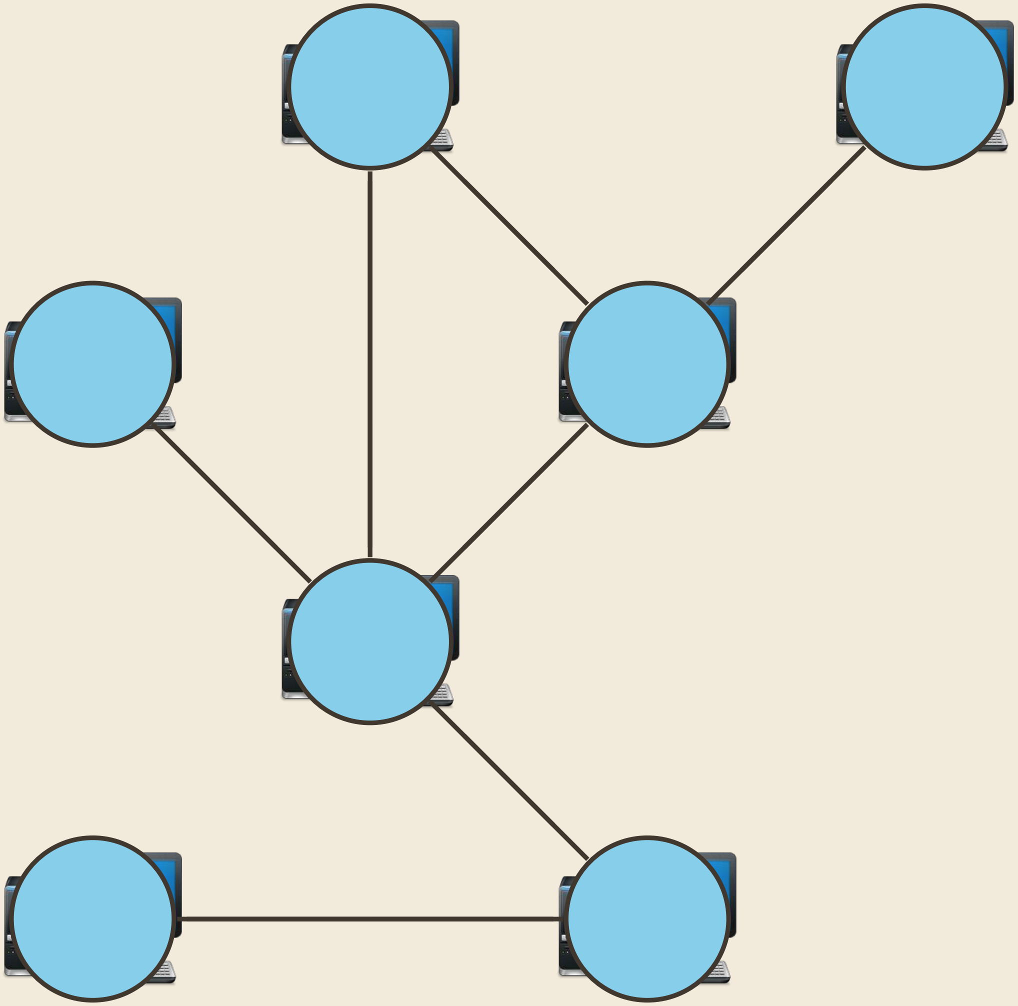
https://verimag.gricad-pages.univ-grenoble-alpes.fr/synchrone/sasa

Outline

Plan

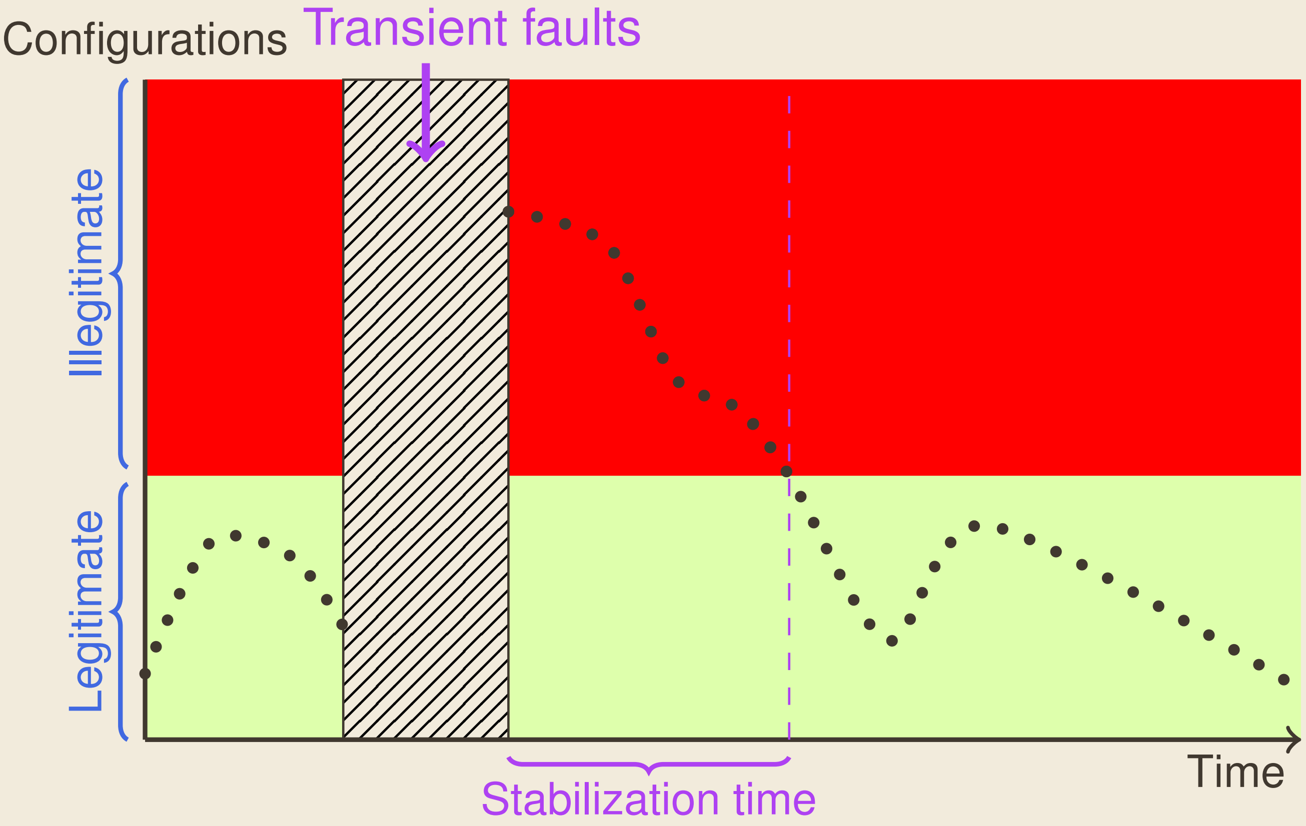
1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

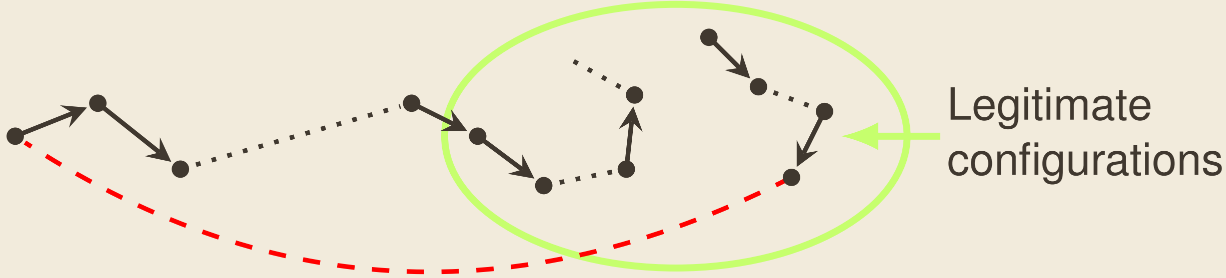
Distributed Systems Algorithms



- Process
  - Autonomous
  - Interconnected
- Hypotheses
  - Connected
  - Bidirectional
  - Identified
- Expected Property
  - Fault-tolerance

Self-Stabilizing Algorithms

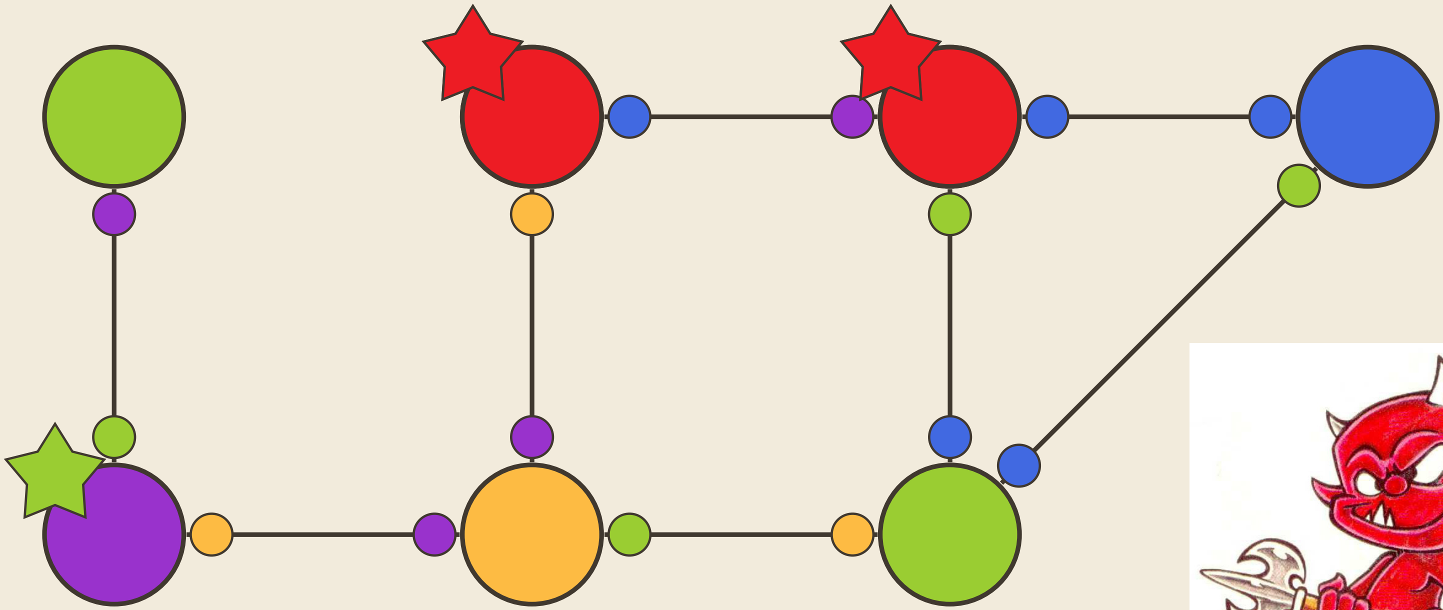




Atomic (Synchronous?) State Model

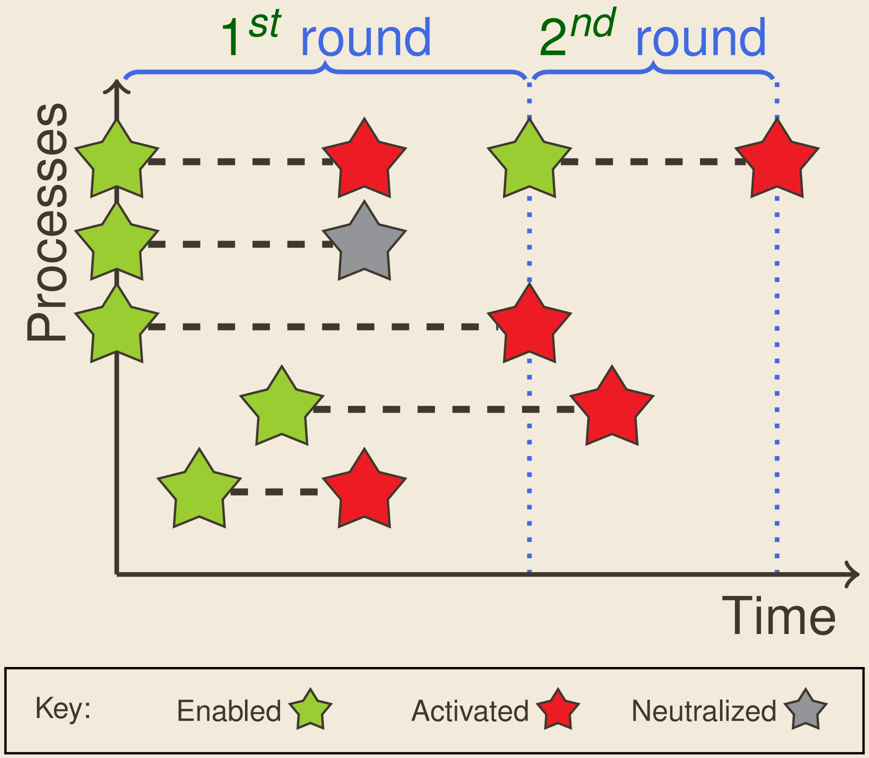
Performing an Atomic Step consists in:

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

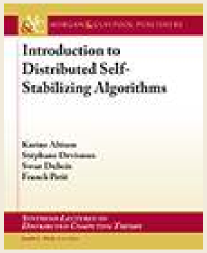


Message Passing Versus Atomic State Models

- Message Passing Model (MPM)
  - Used in the Distributed Algorithms community
  - Lower-level: queues of events
- Atomic State Model (ASM):
  - Used in the Self-Stabilizing Algorithms community
  - Higher-level: atomic instantaneous communications
  - General Algorithms transformations into MPM methods exist

Some Classical Examples

- Dijkstra’s Token Ring
- Coloring Algo
- Synchronous Unison
- A-Synchronous Unison
- BFS spanning tree
- DFS spanning tree [Collin-Dolex-94]


Introduction to Distributed Self-Stabilizing Algorithms

"Introduction to Distributed Self-Stabilizing Algorithms" Altisen, Devismes, Dubois, Petit 2019.

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 $:: p.v = p.Pred.v \hookrightarrow p.v \leftarrow (p.v+1) mod\ K$

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
- Local Variable:
  - $p.v \in \{0, ..., K-1\}$
- Action:
  - T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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

Coloring Algo

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) $\hookrightarrow p.c \leftarrow min(Free(p))$

```
cd test/coloring; rdbg -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 $m \geq max(2, 2 \times \mathscr{D} - 1)$
- Local Variable:
  - $p.c \in \{0, ..., m-1\}$ holds the clock of p
- Macro:
  - $NewClockValue(p) = (min(\{q.c : q \in p.N\} \vee \{p.c\}) + 1 mod\ m$
- Action:
  - Incr :: $p.c \neq NewClockValue(p) \hookrightarrow p.c \leftarrow NewClockvalue(p)$

```
cd test/unison; rdbg -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:
  - $behind(a,b) = ((b.c - a.c) \mod K) \leq n$
- Actions:
  - I :: $\forall q \in p.N, behind(p,q) \hookrightarrow p.c \leftarrow (p.c+1) \mod K$
  - R :: $p.c \neq 0 \wedge (\exists q \in p.N, \neg behind(p,q) \wedge \neg behind(q,p)) \hookrightarrow p.c \leftarrow 0$

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

BFS Spanning tree (1/2)

For the Root process

- Parameters:
    - $root.N$: the set of root's neighbors
    - D : an integer such that $D \geq \mathscr{D}$
- Local Variable:
    - $root.d \in \{0, ..., D\}$ holds the distance to the root
- Action:
    - CD :: $root.d \neq 0 \hookrightarrow root.d \leftarrow 0$

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 $D \geq \mathscr{D}$
- 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 :: $p.d \neq Dist(p) \hookrightarrow p.d \leftarrow Dist(p)$
  - CP :: $DistOK(p) \vee p.par.d \neq p.d - 1 \hookrightarrow p.par \leftarrow q \in p : Ns.t.q(d) = p(d) - 1$

```
cd test/bfs; rdbg -sut "sasa fig51.dot -distributed-demon"
```

DFS Spanning Tree (1/2)

For the Root process

- Parameters:
  - $p.N$: the set of root's neighbors
  - δ: a integer $\geq n$
- Local Variable:
  - $p.path$: an array integers of size δ
- Action:
  - Path :: $p.path \neq [] \hookrightarrow p.pathgets[]$

DFS Spanning Tree (2/2)

For each Non-Root process

- Parameters:
  - $p.N$: the set of process's neighbors
  - δ: 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 δ
- Macros:
  - $ComputePar(p.N) = [...]$
  - $ComputePath(p.N) = [...]$
- Actions:
  - Par :: $p.par \neq ComputePar(p.N) \hookrightarrow p.pargetsComputePar(p.N)$
  - Path :: $p.path \neq ComputePath(p.N) \hookrightarrow p.pathgetsComputePath(p.N)$

```
cd test/dfs; rdbg -sut "sasa g.dot"
```

Plan

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

Simulators Dedicated to Self-Stabilization

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)

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!

Plan

SASA: main features

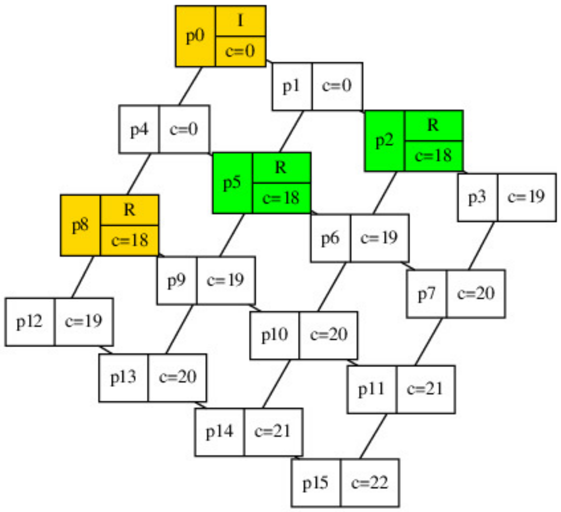
- Batch Simulations
    - Debug Algorithms
    - Perform simulation campaigns,
        - Study the influence of some parameters
        - Evaluate the (average-case) complexity Lower bounds
- Test oracles to formalize expected properties
    - involve the number of steps, moves, or rounds to reach a legitimate configuration (differs from algorithms).
- Daemon can be configured
    - Predefined: synchronous, central, locally central, or distributed
    - Custom daemons: manual or programmed
- Interactive Simulations
    - step by step, or round by round, forward or backward
    - while visualizing the network, the enabled, the activated actions
    - New commands can also be programmed

Defining The Network Topology

- Take advantage of the GraphViz `dot` language
  - Simple syntax
  - Open-source
  - Plenty of visualizers, editors, parsers, exporters
- `dot` attributes
  - name-value pairs that can be ignored (pragmas)
  - node attributes: `algo`, `init`
  - graph attributes: global simulation parameters

A Topology Example: a 4x4 grid

```
graph g {
 graph [n=24]
  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"]
```



grid4.dot step 2

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

- Users need to define 4 things:
  1. a list of action labels
  2. an enable function, which encodes the guards of the algorithm
  3. a step function, that triggers enabled actions
  4. a state initialization function (used if not provided in the DOT file)

```
type action   = string
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 -> 's
```

Algorithm Programming Interface (2/4)

Each node can get (or not) information on its neighbors:

```
exception Not_available

val state : 's neighbor -> 's
val pid   : 's neighbor -> string
val spid  : 's neighbor -> string
val reply : 's neighbor -> int
val weight: 's neighbor -> int
```

Algorithm Programming Interface (3/4)

Some of the topological information can be accessed:

```
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
```

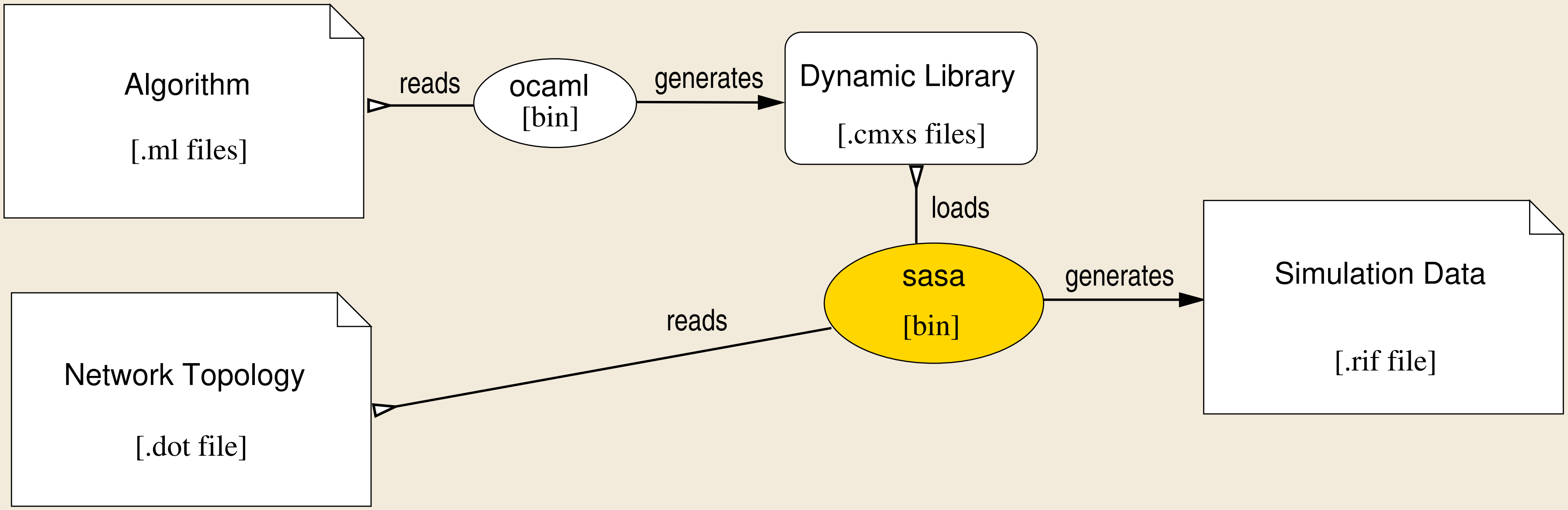
37 straightforward loc

Algorithm Programming Interface (3/4)

Registration

```
type 's algo_to_register = {
  algo_id    : string;
  init_state: int -> 's;
  enab       : 's enable_fun;
  step       : 's step_fun;
  actions    : action list option   }
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  state_of_string: (string -> 's) option;
  copy_state: 's -> 's   }
val register : 's to_register -> unit
```

The SASA Core Simulator Architecture



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 :: $p.v = p.Pred.v \hookrightarrow p.v \leftarrow (p.v + 1) mod\ K$

```
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 ["T"] 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:
  $p.v \in \{0, ..., K-1\}$
- Action:
  T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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

```
cd test/dijksra; rdbg -sut "sasa
ring.dot -distributed-demon"
```

Coloring Algo

- 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)
  $\hookrightarrow p.c \leftarrow min(Free(p))$

```
open Algo
let k=3
let init_state _ = Random.int k
let neigbhors_vals nl = List.map (fun n -> state n) nl
let confl v nl = List.mem v (neigbhors_vals nl)
let free nl =
 let confl1 = List.sort_uniq compare (neigbhors_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 -locally-central-demon"
```

Plan

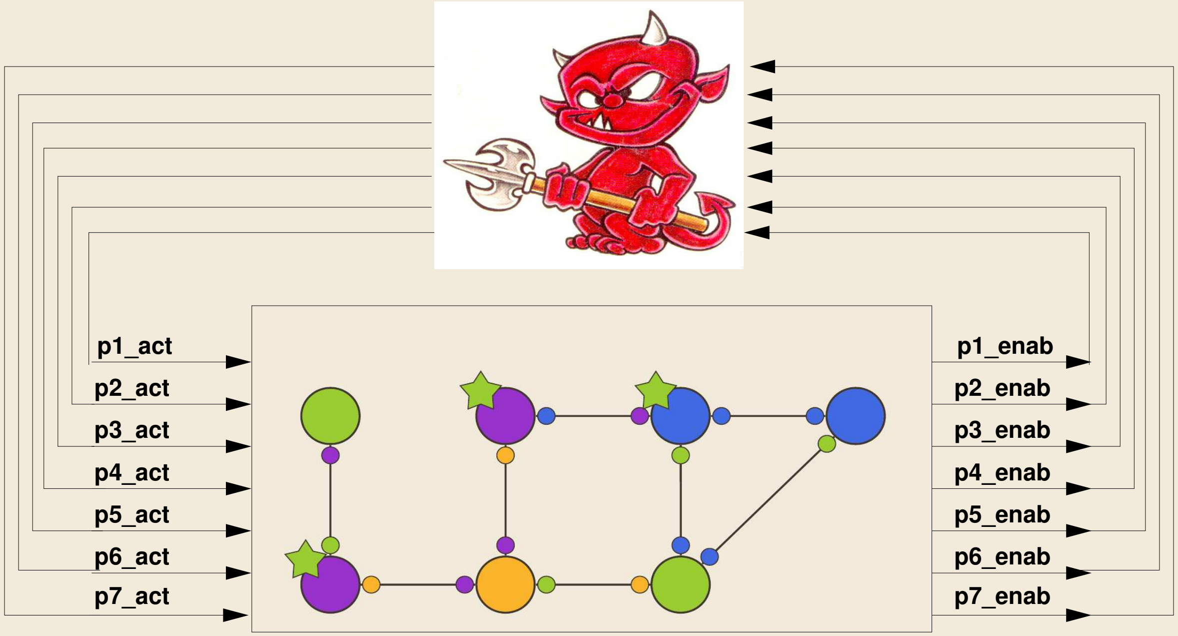
Algorithms in the ASM viewed as Reactive programs

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

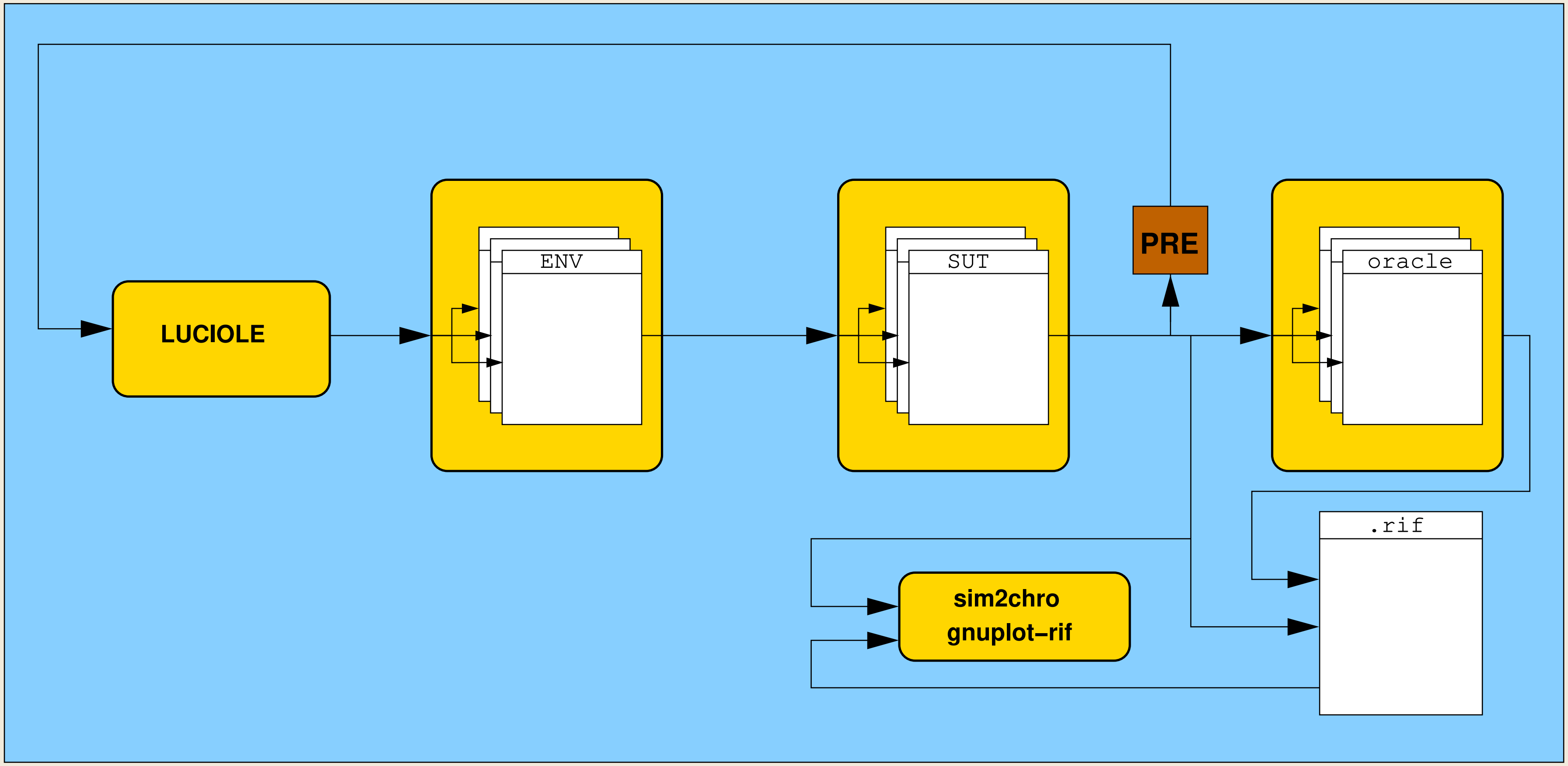


Figure: The LURETTE dataflow schema

RDBG

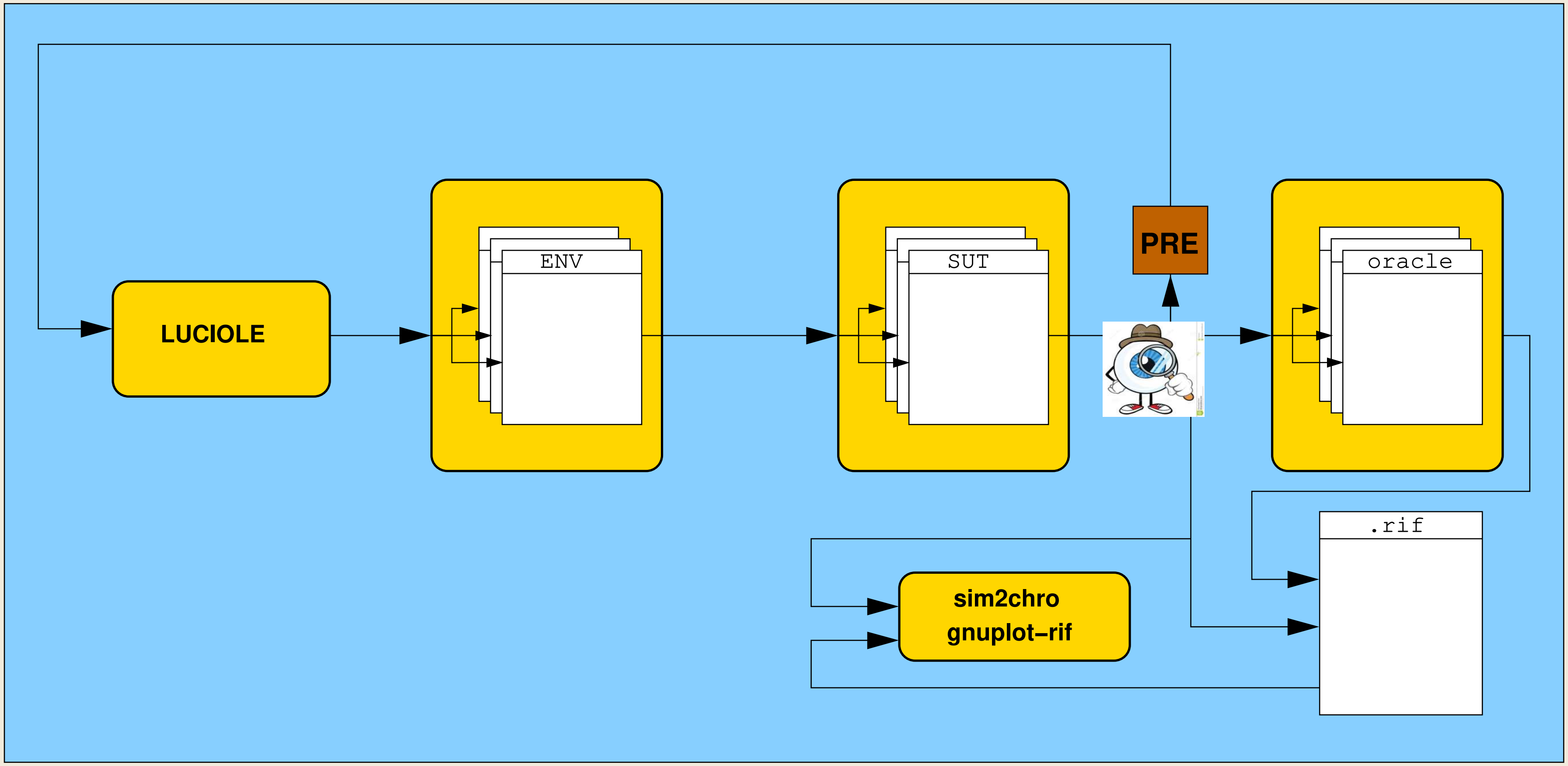


Figure: The RDBG dataflow schema

RDBG

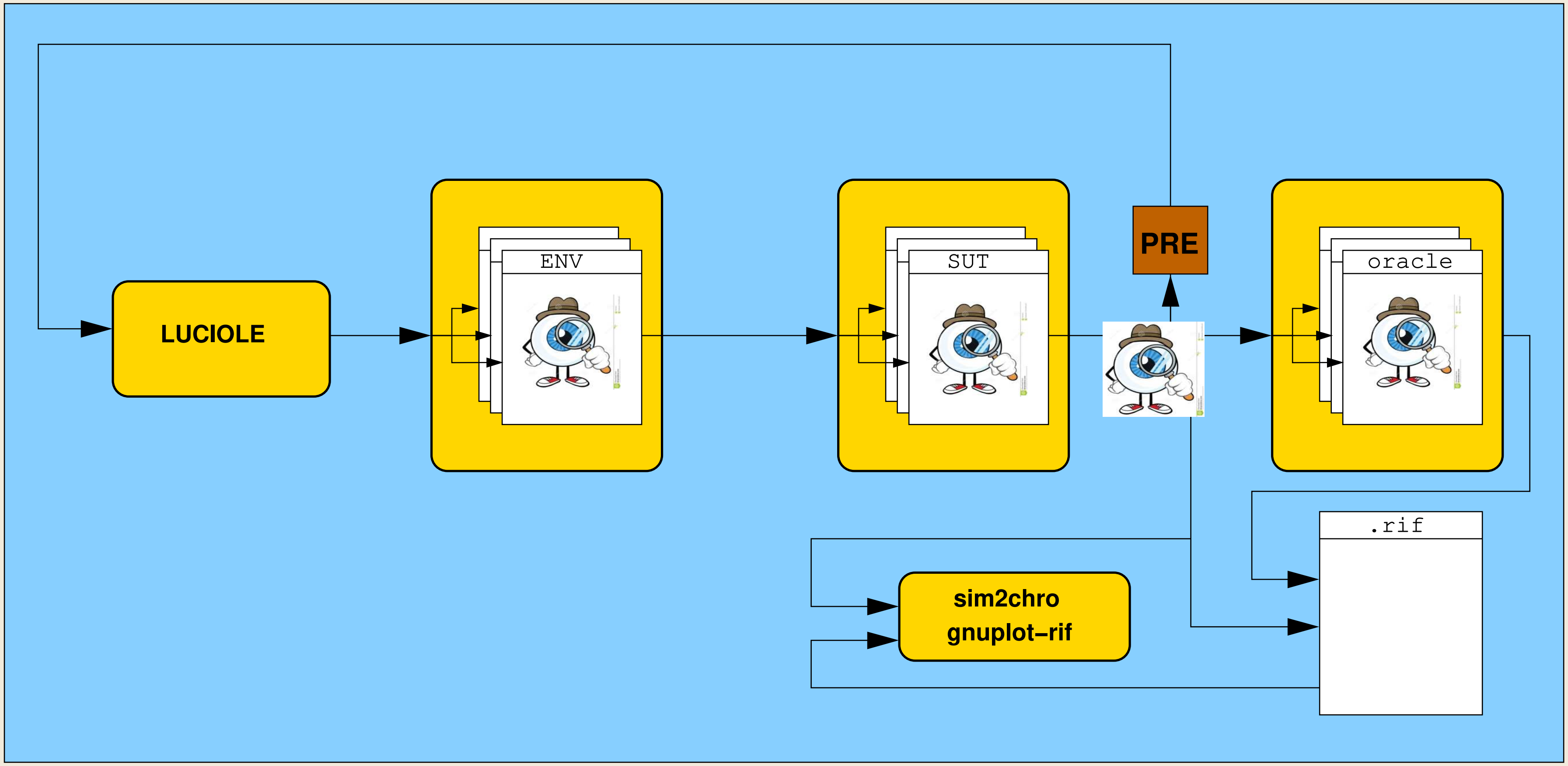


Figure: The RDBG dataflow schema

Lurette and Test Oracles

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

```
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

```
cd test/dijkstra; rdbg -env "sasa ring.dot -custom-demon"
-sut-nd "lutin ring.lut -n distributed"
```

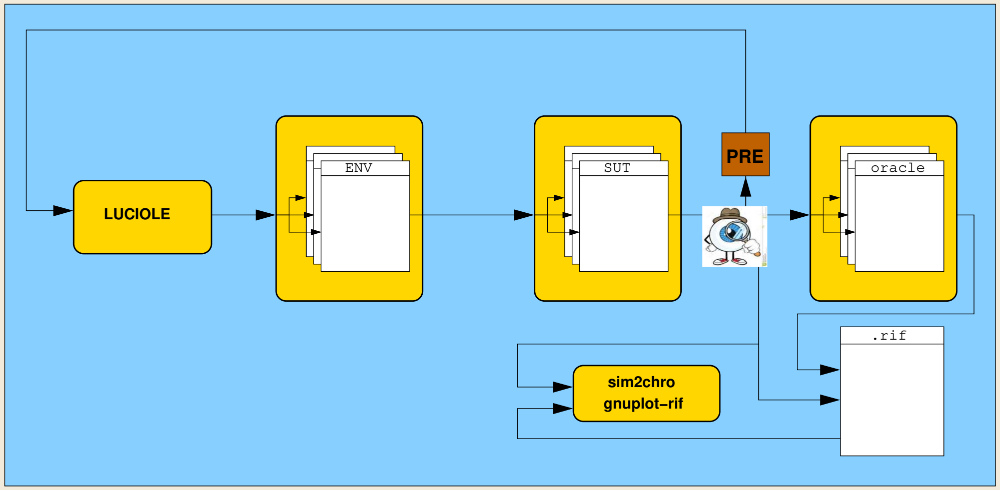
RDBG

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`

RDBG and SASA



- 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

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

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Performance Evaluation: Benchmarks Algorithms

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.

	`grid.dot`		`ER.dot`		`biggrid.dot`		`bigER.dot`	
	Time/step	Mem	Time/step	Mem	Time/step	Mem	Time/step	Mem
BFS	0.2 ms	13 MB	10.6 ms	49 MB	2.04 s	83 MB	3.03 s	1062 MB
DFS-l	1 ms	44 MB	144.7 ms	63 MB	2.57 s	92 MB	15.83 s	1062 MB
DFS-a	0.5 ms	39 MB	94.3 ms	170 MB	7.64 s	6642 MB	86.93 s	29 945 MB
COL	0 ms	7 MB	35.8 ms	63 MB	27.93 s	75 MB	16.81 s	1083 MB
SYN	0.3 ms	38 MB	10.9 ms	63 MB	887.05 s	874 MB	13.58 s	1099 MB
ASY	0.1 ms	38 MB	4.5 ms	63 MB	0.03 s	83 MB	2.82 s	1115 MB

- 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. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Polymorphic versus Variant Type

- An alternative to polymorphism to hold processes local state:

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

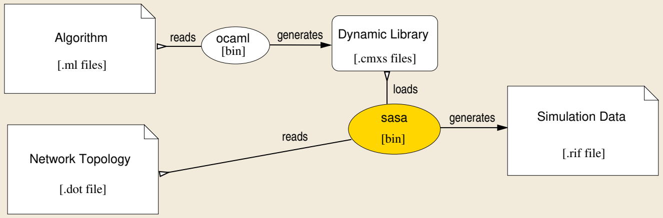
But:

- What if one need a type that is not in this variant list?
- Variable values need to be set/get in/from the env^t all the time.

```
let step_f c nl a =
   match a with
    | "I" -> modulo (c + 1) k
    | "R" -> 0
```

```
let step_f env nl a =
   match a with
    | "I" ->
       let c_val = match env_get env "c" with
           | I i -> i
           | _ -> assert false
       in
       set_env env "c" (I(modulo ((c_val)+1) k))
    | "R" -> set_env env "c" (I 0)
```

Dynamic versus Static Linking



- Dynamic Linking: Pros
  - Easier to use
  - Save Disk space
  - Separation of concerns: user code only depends on a simple API
- Dynamic Linking: Cons
  - Can not be combined gently with Polymorphic values!

Dynamic Type Checking of Polymorphic Nodes

- Dynamic linking in OCAML needs to be done via imperative tables
  - The code to be linked registers functions into tables
  - The main executable reads the tables of functions
- But storing polymorphic values into a mutable data-type is not possible in ML-like languages; one can only store so-called weakly polymorphic values!
- Weak variables can't escape the scope of a compilation unit

`https://ocamlverse.github.io/content/weak_type_variables.html`

Dynamic Type Checking of Polymorphic Nodes

- Solution: use the (evil) `Obj` module:
  - `Obj.obj: 'a -> t`: to register polymorphic functions into tables
  - `Obj.repr: t -> 'a`: to retrieve them from the simulation engine
- Using `Obj` breaks type safety: how to prevent users to register functions of different type?

By forcing all functions to be registrated at the same time:

```
type 's algo_to_register = {
  algo_id : string;
  init_state: int -> 's;
  enab : 's enable_fun;
  step : 's step_fun;
  actions : action list option }
type 's to_register = {
  algo : 's algo_to_register list; (* <==== ALL AlGO HAVE THE SAME TYPE! *)
  state_to_string: 's -> string;
  state_of_string: (string -> 's) option;
  copy_state: 's -> 's }
val register : 's to_register -> unit
```

Plan

Conclusion

- An open-source SimulAtor of Self-stabilizing Algorithms
- writen 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
- Installation via
  - `docker`
  - `opam`
  - `git`

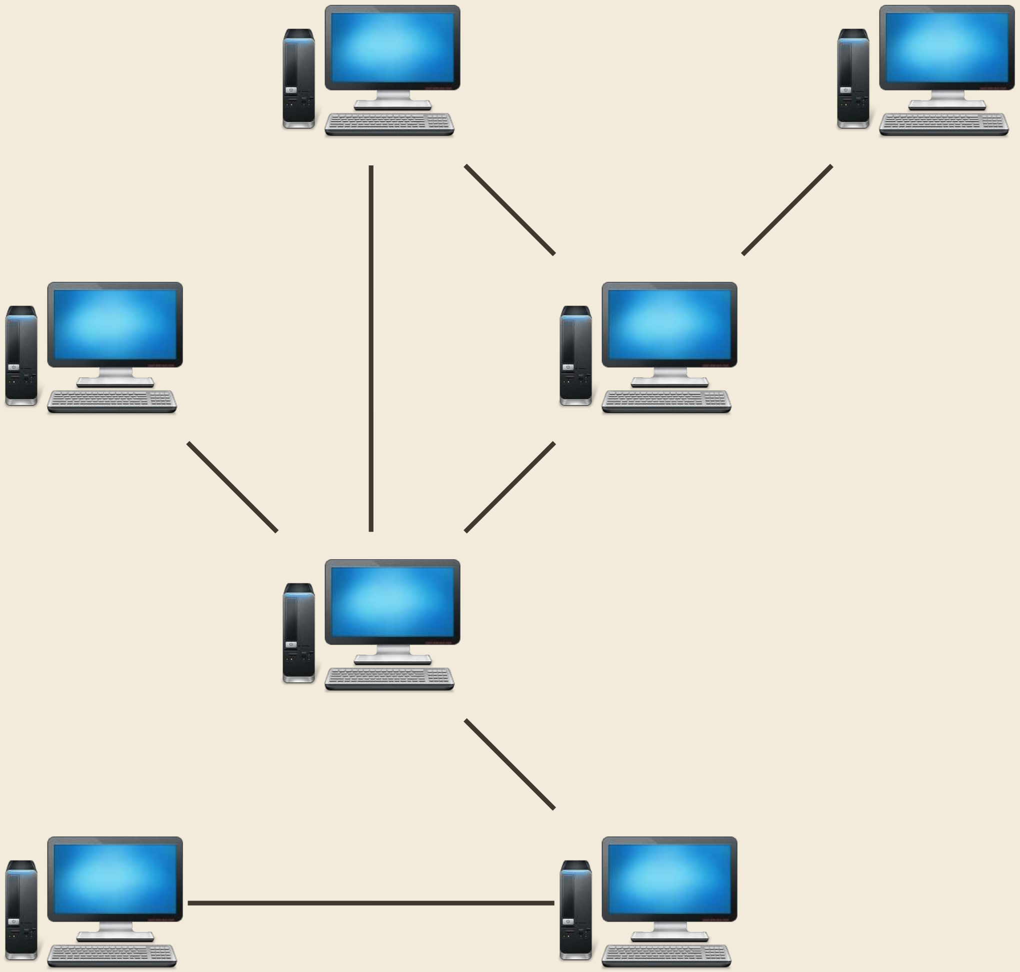
`https://verimag.gricad-pages.univ-grenoble-alpes.fr/synchrone/sasa`

Outline

Plan

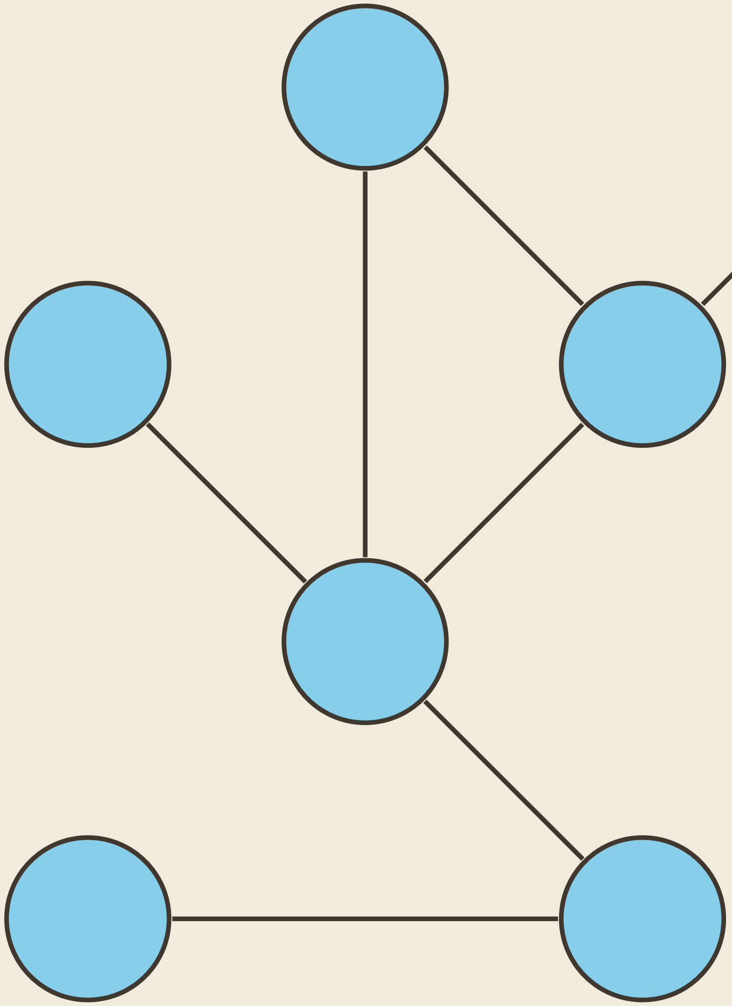
1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

Distributed Systems Algorithms



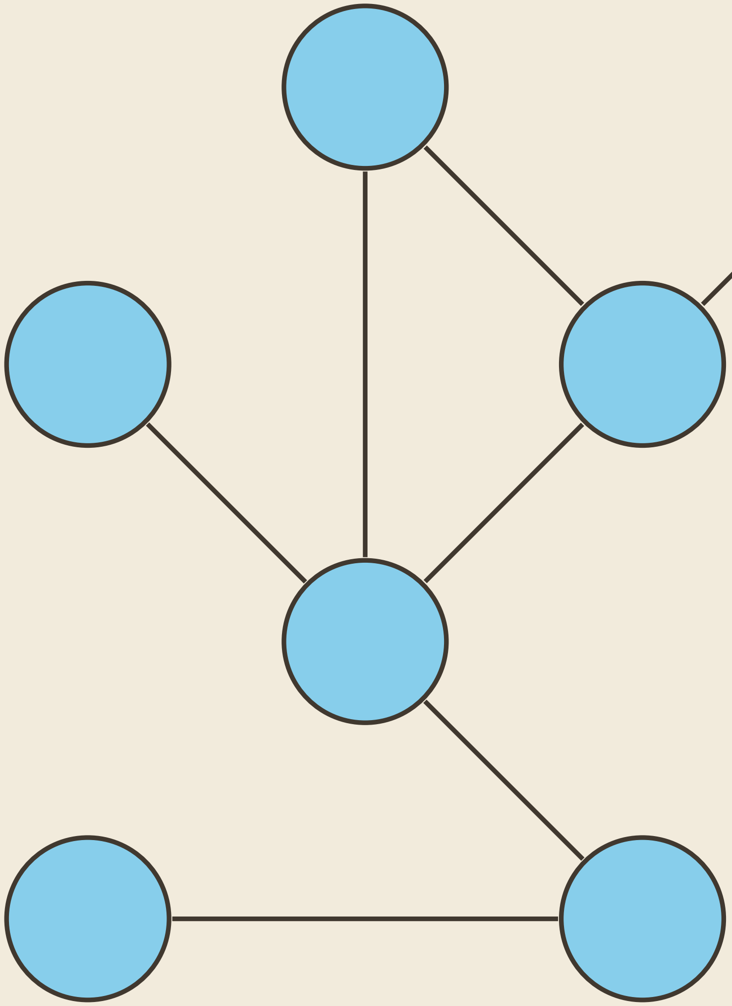
- Process
  - Autonomous
  - Interconnected

Distributed Systems Algorithms



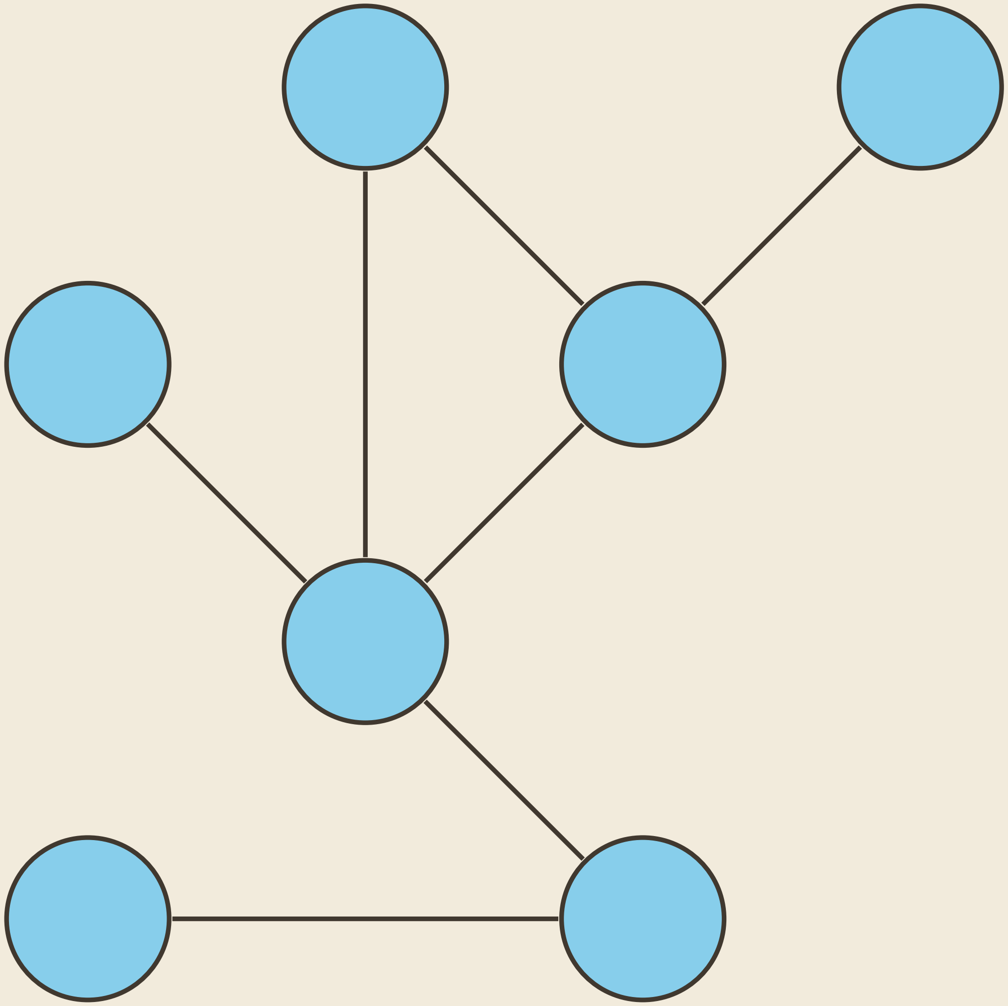
- Process
  - Autonomous
  - Interconnected

Distributed Systems Algorithms



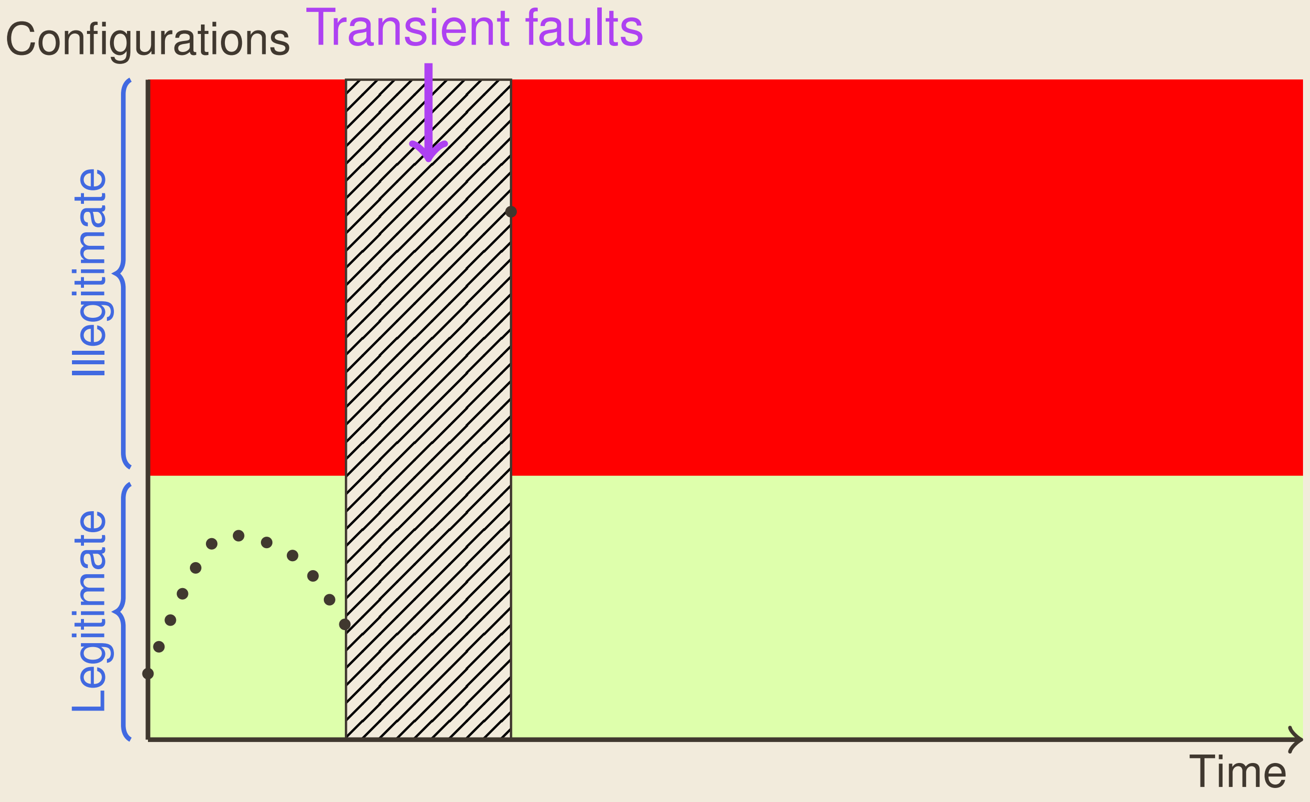
- Process
  - Autonomous
  - Interconnected
- Hypotheses
  - Connected
  - Bidirectional
  - Identified

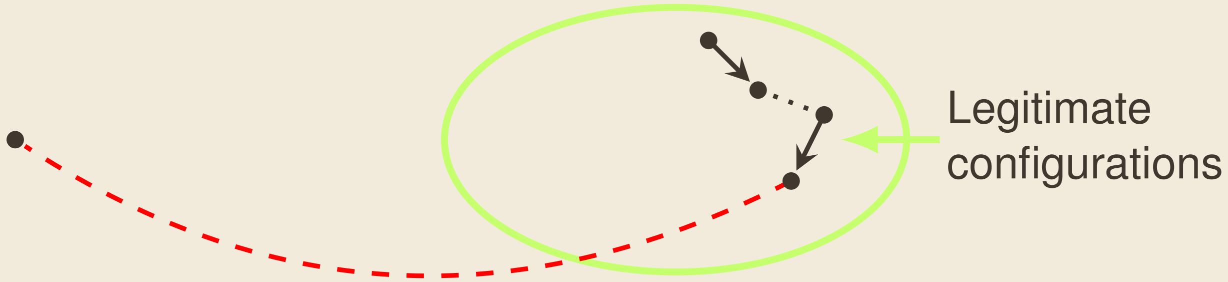
Distributed Systems Algorithms



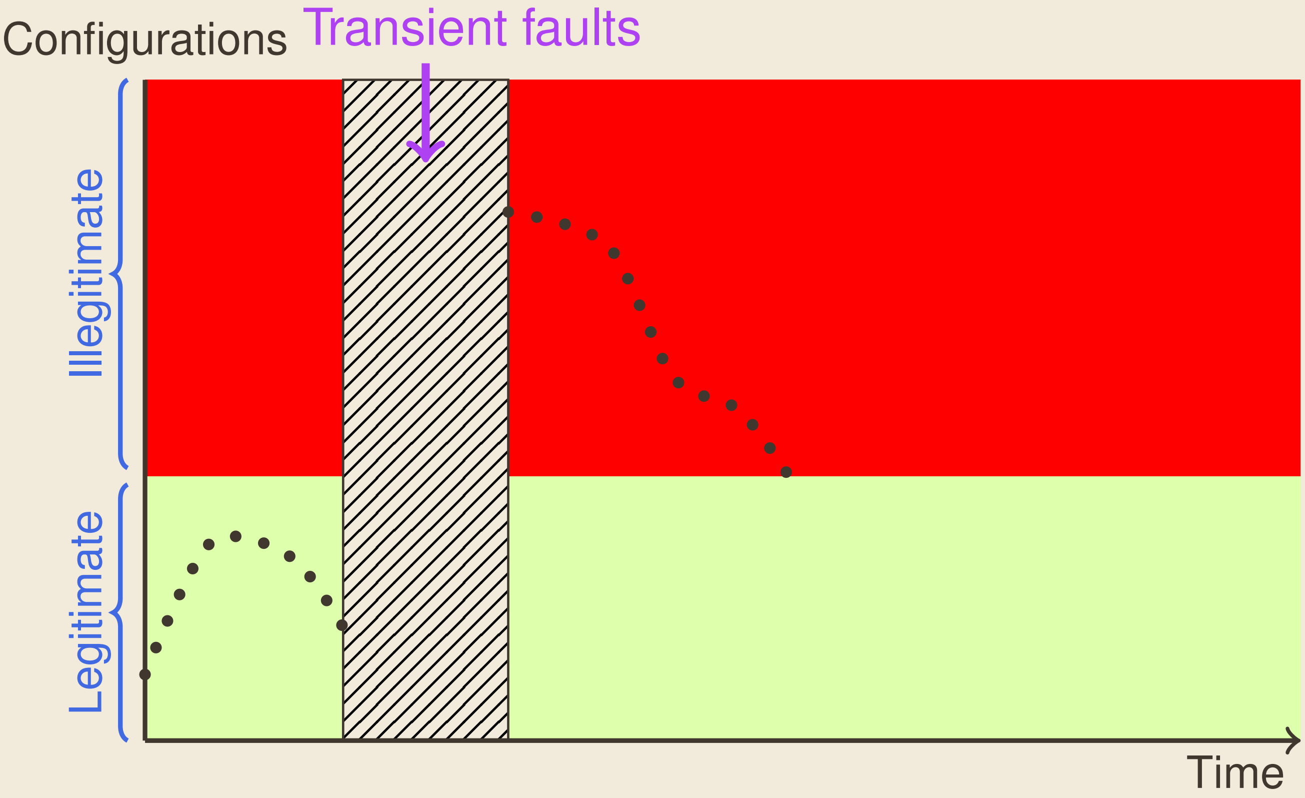
- Process
  - Autonomous
  - Interconnected
- Hypotheses
  - Connected
  - Bidirectional
  - Identified
- Expected Property
  - Fault-tolerance

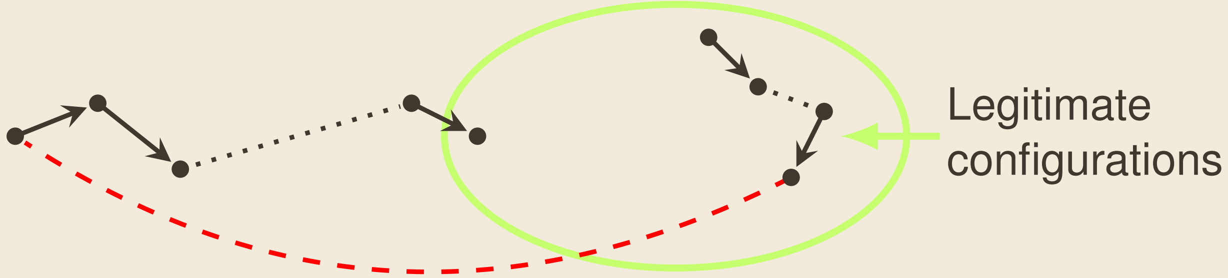
Self-Stabilizing Algorithms



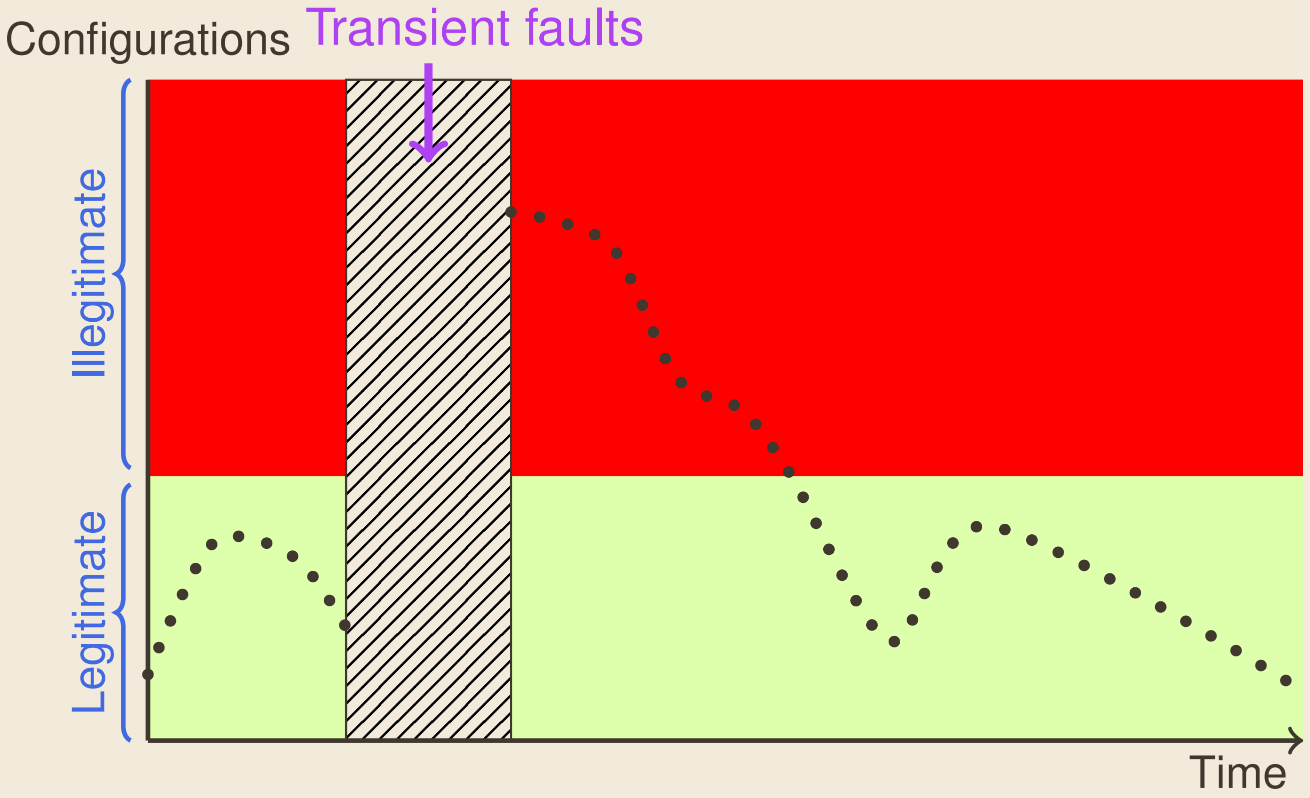


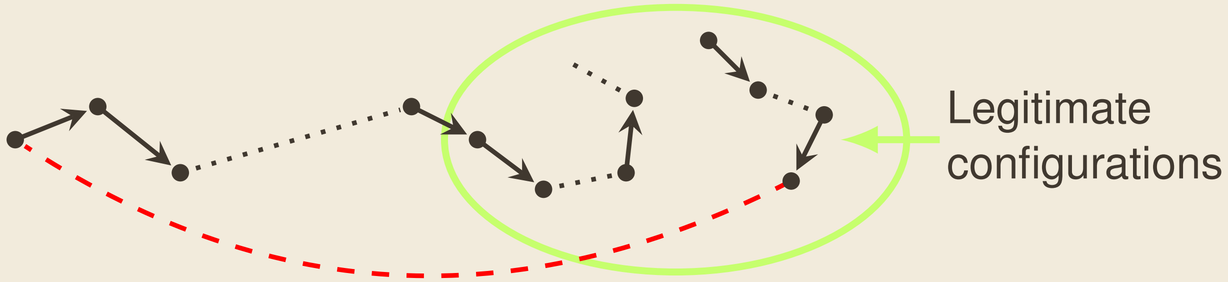
Self-Stabilizing Algorithms



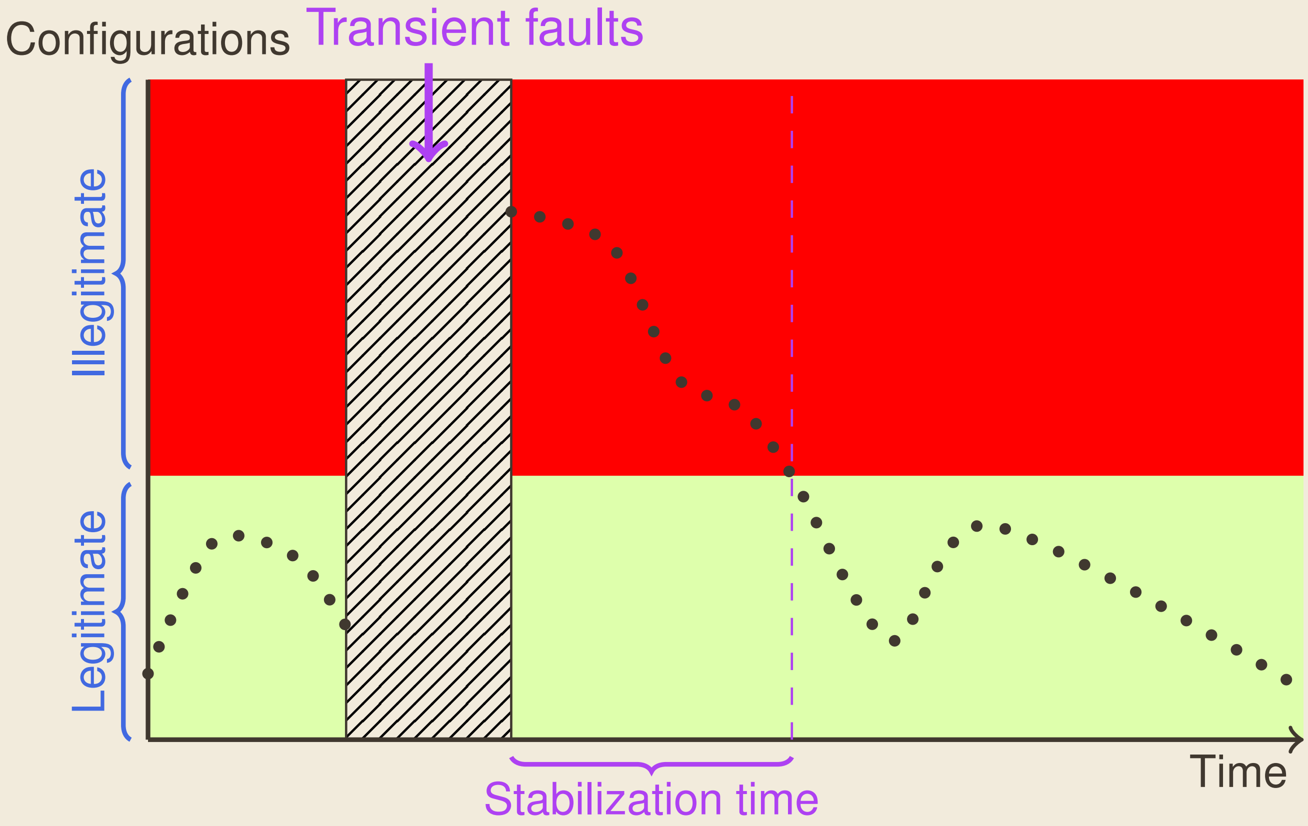


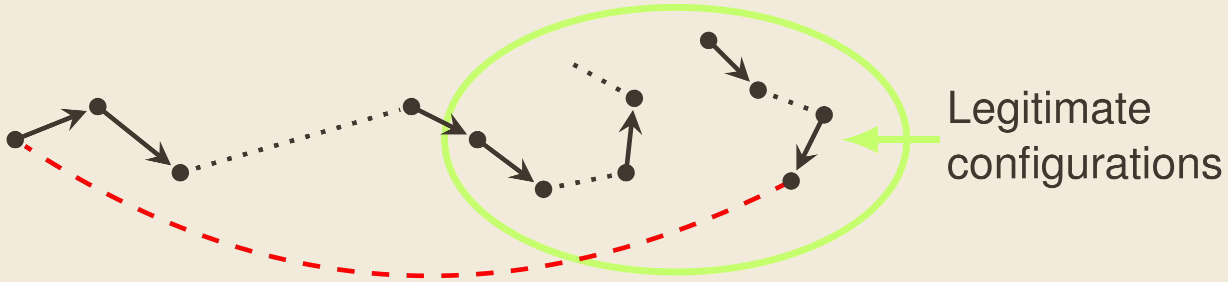
Self-Stabilizing Algorithms





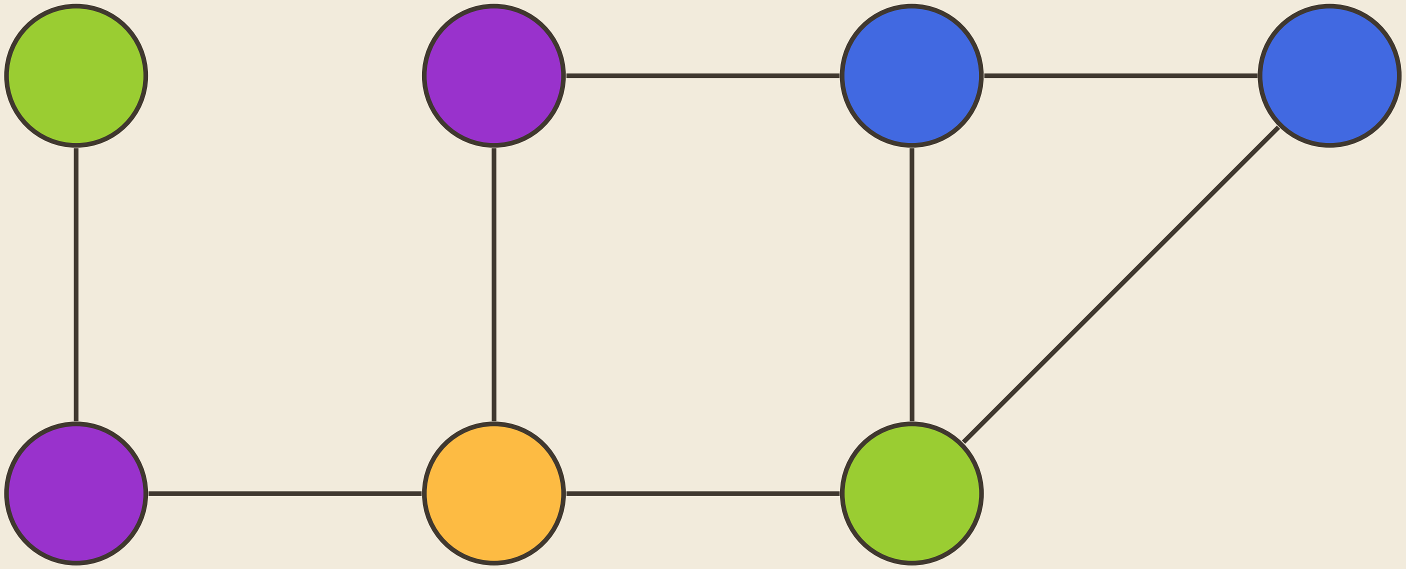
Self-Stabilizing Algorithms





Atomic (Synchronous?) State Model

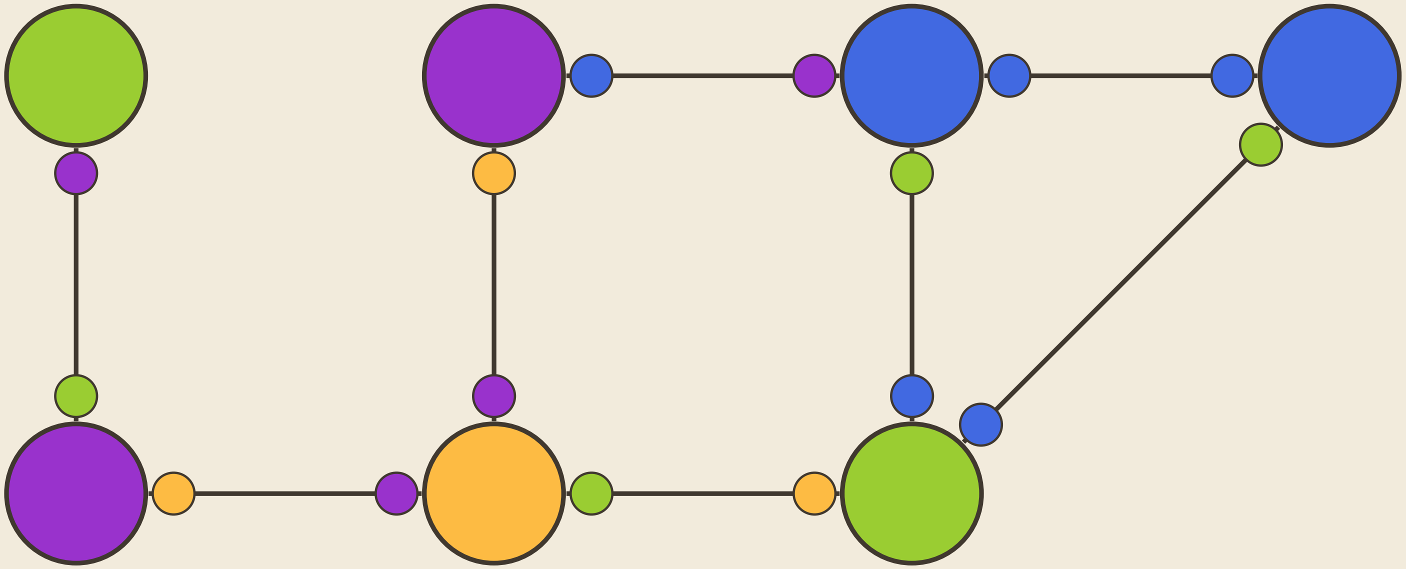
From a particular Configuration of local Memories



Atomic (Synchronous?) State Model

Performing an Atomic Step consists in:

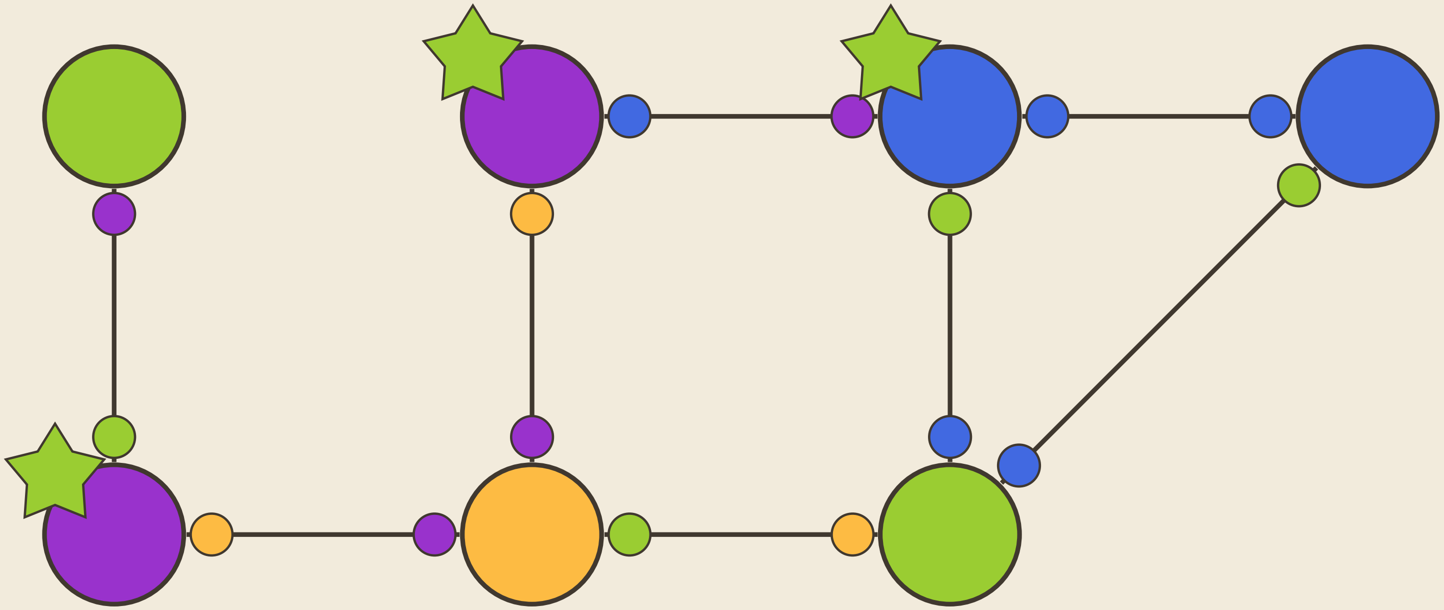
1. Reading neighbors variables



Atomic (Synchronous?) State Model

Performing an Atomic Step consists in:

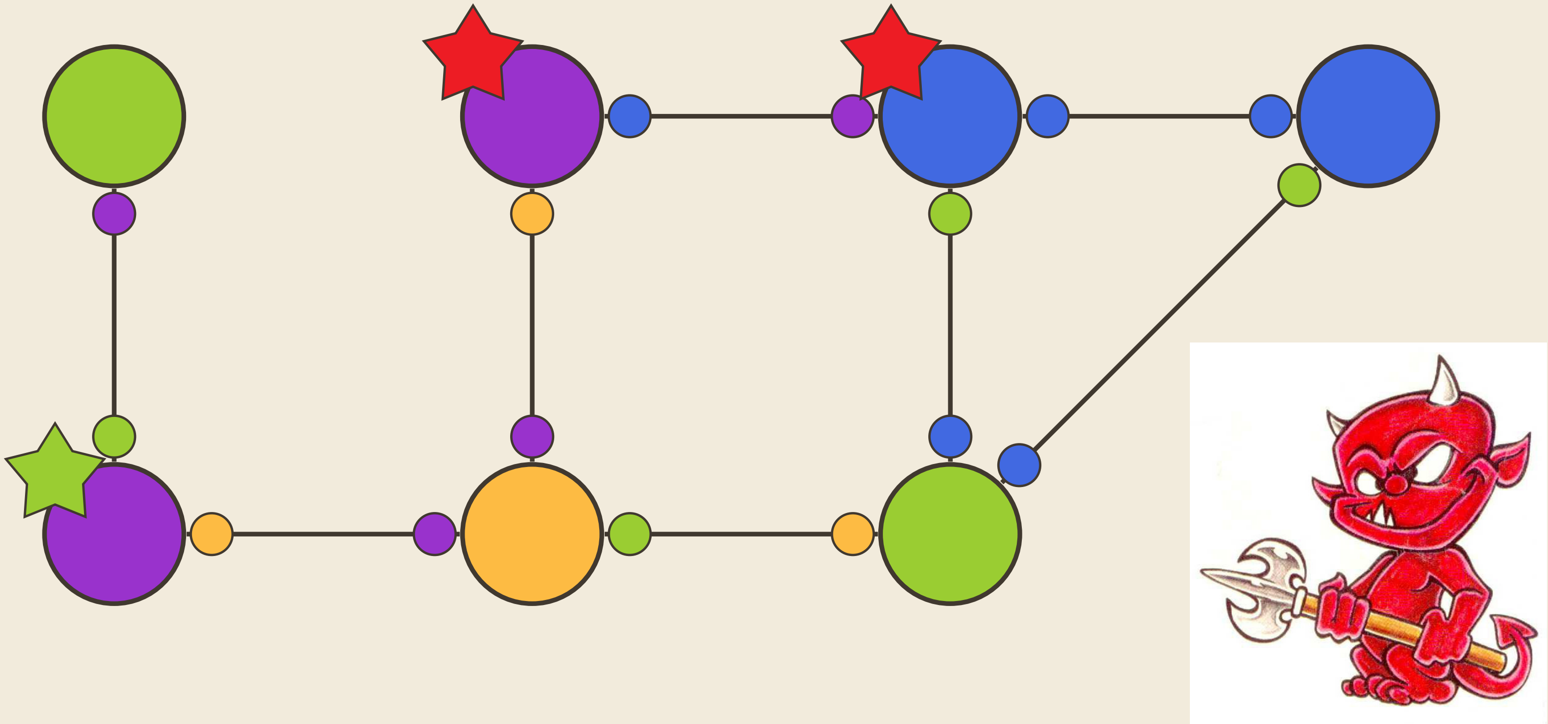
1. Reading neighbors variables
2. Computing enabled nodes



Atomic (Synchronous?) State Model

Performing an Atomic Step consists in:

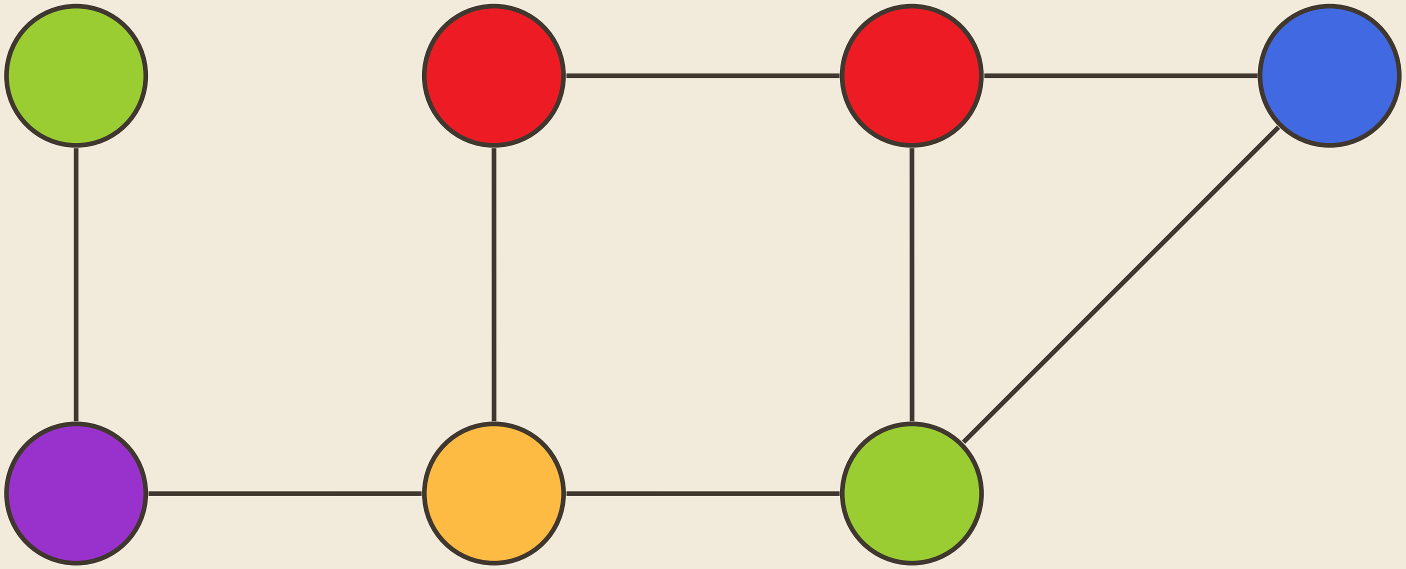
1. Reading neighbors variables
2. Computing enabled nodes
3. Choosing nodes to activate: a Daemon models the asynchronism



Atomic (Synchronous?) State Model

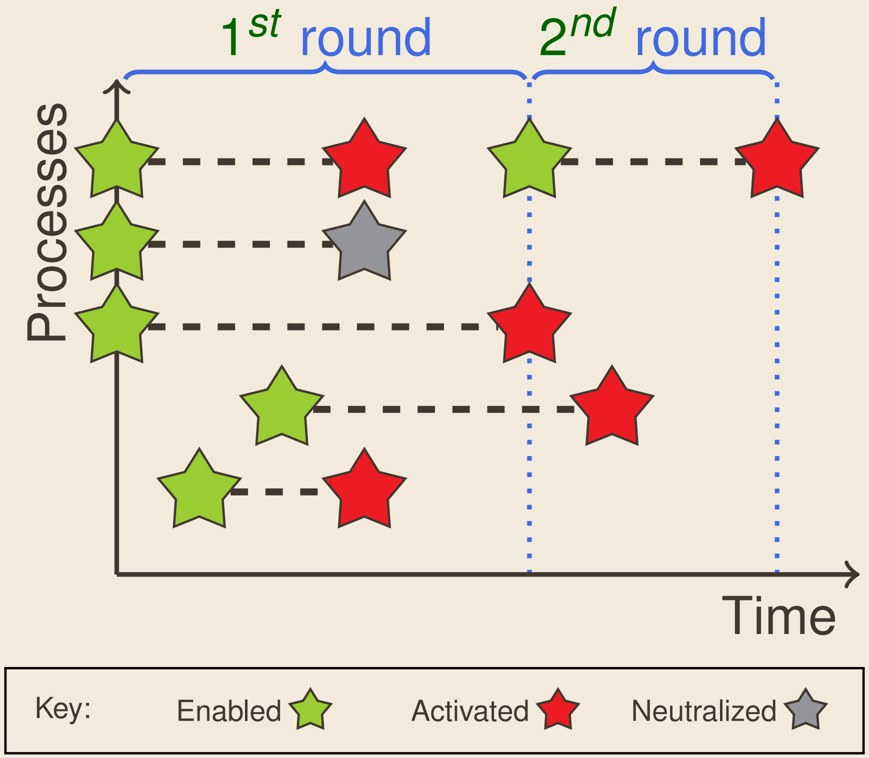
Performing an Atomic Step consists in:

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

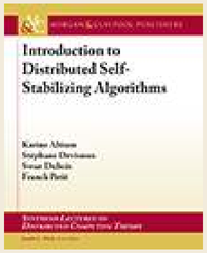


Message Passing Versus Atomic State Models

- Message Passing Model (MPM)
  - Used in the Distributed Algorithms community
  - Lower-level: queues of events
- Atomic State Model (ASM):
  - Used in the Self-Stabilizing Algorithms community
  - Higher-level: atomic instantaneous communications
  - General Algorithms transformations into MPM methods exist

Some Classical Examples

- Dijkstra's Token Ring
- Coloring Algo
- Synchronous Unison
- A-Synchronous Unison
- BFS spanning tree
- DFS spanning tree [Collin-Dolex-94]


Introduction to
Distributed Self-
Stabilizing Algorithms

"Introduction to Distributed Self-Stabilizing Algorithms" Altisen, Devismes, Dubois, Petit 2019.

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

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 :: $p.v = p.Pred.v \hookrightarrow p.v \leftarrow (p.v+1) mod\ K$

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

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

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:
    - $p.v \in \{0, ..., K-1\}$
- Action:
    - T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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:
  - $p.v \in \{0, ..., K-1\}$
- Action:
  - T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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

Coloring Algo

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

Coloring Algo

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) $\hookrightarrow p.c \leftarrow min(Free(p))$

Coloring Algo

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) $\hookrightarrow p.c \leftarrow min(Free(p))$

Coloring Algo

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) $\hookrightarrow p.c \leftarrow min(Free(p))$

```
cd test/coloring; rdbg -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 $m \geq max(2, 2 \times \mathscr{D} - 1)$
- Local Variable:
  - $p.c \in \{0, ..., m-1\}$ holds the clock of p
- Macro:
  - $NewClockValue(p) = (min(\{q.c : q \in p.N\} \vee \{p.c\}) + 1 mod\ m$
- Action:
  - Incr :: $p.c \neq NewClockValue(p) \hookrightarrow p.c \leftarrow NewClockvalue(p)$

```
cd test/unison; rdbg -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:
  - $behind(a,b) = ((b.c - a.c) \mod K) \leq n$
- Actions:
  - I :: $\forall q \in p.N, behind(p,q) \hookrightarrow p.c \leftarrow (p.c+1) \mod K$
  - R :: $p.c \neq 0 \wedge (\exists q \in p.N, \neg behind(p,q) \wedge \neg behind(q,p)) \hookrightarrow p.c \leftarrow 0$

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

BFS Spanning tree (1/2)

For the Root process

- Parameters:
  - $root.N$: the set of root's neighbors
  - D : an integer such that $D \geq \mathscr{D}$

BFS Spanning tree (1/2)

For the Root process

- Parameters:
  - $root.N$: the set of root's neighbors
  - D : an integer such that $D \geq \mathscr{D}$
- 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 $D \geq \mathscr{D}$

- Local Variable:
  - $root.d \in \{0, ..., D\}$ holds the distance to the root

- Action:
  - CD :: $root.d \neq 0 \hookrightarrow root.d \leftarrow 0$

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 $D \geq \mathscr{D}$
- 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 :: $p.d \neq Dist(p) \hookrightarrow p.d \leftarrow Dist(p)$
  - CP :: $DistOK(p) \vee p.par.d \neq p.d - 1 \hookrightarrow p.par \leftarrow q \in p : Ns.t.q(d) = p(d) - 1$

```
cd test/bfs; rdbg -sut "sasa fig51.dot -distributed-demon"
```

DFS Spanning Tree (1/2)

For the Root process

- Parameters:
  - $p.N$: the set of root's neighbors
  - δ: a integer $\geq n$
- Local Variable:
  - $p.path$: an array integers of size δ
- Action:
  - Path :: $p.path \neq [] \hookrightarrow p.pathgets[]$

DFS Spanning Tree (2/2)

For each Non-Root process

- Parameters:
  - $p.N$: the set of process's neighbors
  - δ: 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 δ
- Macros:
  - $ComputePar(p.N) = [...]$
  - $ComputePath(p.N) = [...]$
- Actions:
  - Par :: $p.par \neq ComputePar(p.N) \hookrightarrow p.pargetsComputePar(p.N)$
  - Path :: $p.path \neq ComputePath(p.N) \hookrightarrow p.pathgetsComputePath(p.N)$

```
cd test/dfs; rdbg -sut "sasa g.dot"
```

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

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

Simulators Dedicated to Self-Stabilization

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)

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!

Plan

1. Self-stabilizing Algorithms in the Atomic-State Model
2. Simulation of Self-stabilizing Algorithms
3. **SASA**
4. Integration with Synchronous tools
5. Performance Evaluation
6. Some Design Choices
7. Conclusion

SASA: main features

- Batch Simulations
  - Debug Algorithms
  - Perform simulation campaigns,
    - Study the influence of some parameters
    - Evaluate the (average-case) complexity Lower bounds

SASA: main features

- Batch Simulations
  - Debug Algorithms
  - Perform simulation campaigns,
    - Study the influence of some parameters
    - Evaluate the (average-case) complexity Lower bounds
- Test oracles to formalize expected properties
  - involve the number of steps, moves, or rounds to reach a legitimate configuration (differs from algorithms).

SASA: main features

- Batch Simulations
  - Debug Algorithms
  - Perform simulation campaigns,
    - Study the influence of some parameters
    - Evaluate the (average-case) complexity Lower bounds
- Test oracles to formalize expected properties
  - involve the number of steps, moves, or rounds to reach a legitimate configuration (differs from algorithms).
- Daemon can be configured
  - Predefined: synchronous, central, locally central, or distributed
  - Custom daemons: manual or programmed

SASA: main features

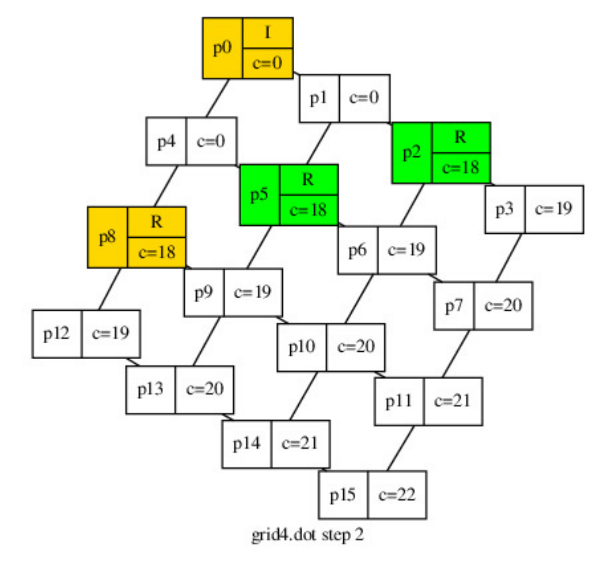
- Batch Simulations
  - Debug Algorithms
  - Perform simulation campaigns,
    - Study the influence of some parameters
    - Evaluate the (average-case) complexity Lower bounds
- Test oracles to formalize expected properties
  - involve the number of steps, moves, or rounds to reach a legitimate configuration (differs from algorithms).
- Daemon can be configured
  - Predefined: synchronous, central, locally central, or distributed
  - Custom daemons: manual or programmed
- Interactive Simulations
  - step by step, or round by round, forward or backward
  - while visualizing the network, the enabled, the activated actions
  - New commands can also be programmed

Defining The Network Topology

- Take advantage of the GraphViz `dot` language
  - Simple syntax
  - Open-source
  - Plenty of visualizers, editors, parsers, exporters
- `dot` attributes
  - name-value pairs that can be ignored (pragmas)
  - node attributes: `algo`, `init`
  - graph attributes: global simulation parameters

A Topology Example: a 4x4 grid

```
graph g {
 graph [n=24]
  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"]
```



grid4.dot step 2

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

1. a list of action labels

```
type action  = string
```

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

- Users need to define 4 things:
  1. a list of action labels
  2. an enable function, which encodes the guards of the algorithm

```
type action  = string
type 's enable_fun = 's -> 's neighbor list -> action list
```

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

- Users need to define 4 things:
  1. a list of action labels
  2. an enable function, which encodes the guards of the algorithm
  3. a step function, that triggers enabled actions

```
type action   = string
type 's enable_fun = 's -> 's neighbor list -> action list
type 's step_fun   = 's -> 's neighbor list -> action -> 's
```

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

- Users need to define 4 things:
  1. a list of action labels
  2. an enable function, which encodes the guards of the algorithm
  3. a step function, that triggers enabled actions
  4. a state initialization function (used if not provided in the DOT file)

```
type action  = string
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 -> 's
```

Algorithm Programming Interface

- 37 straightforward loc of Ocaml Interface (`mli`) file (162 with comments)
- Local states are polymorphic

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

- Users need to define 4 things:
  1. a list of action labels
  2. an enable function, which encodes the guards of the algorithm
  3. a step function, that triggers enabled actions
  4. a state initialization function (used if not provided in the DOT file)

```
type action   = string
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 -> 's
```

Algorithm Programming Interface (2/4)

Each node can get (or not) information on its neighbors:

```
exception Not_available

val state : 's neighbor -> 's
val pid   : 's neighbor -> string
val spid  : 's neighbor -> string
val reply : 's neighbor -> int
val weight: 's neighbor -> int
```

Algorithm Programming Interface (3/4)

Some of the topological information can be accessed:

```
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 (3/4)

Some of the topological information can be accessed:

```
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
```

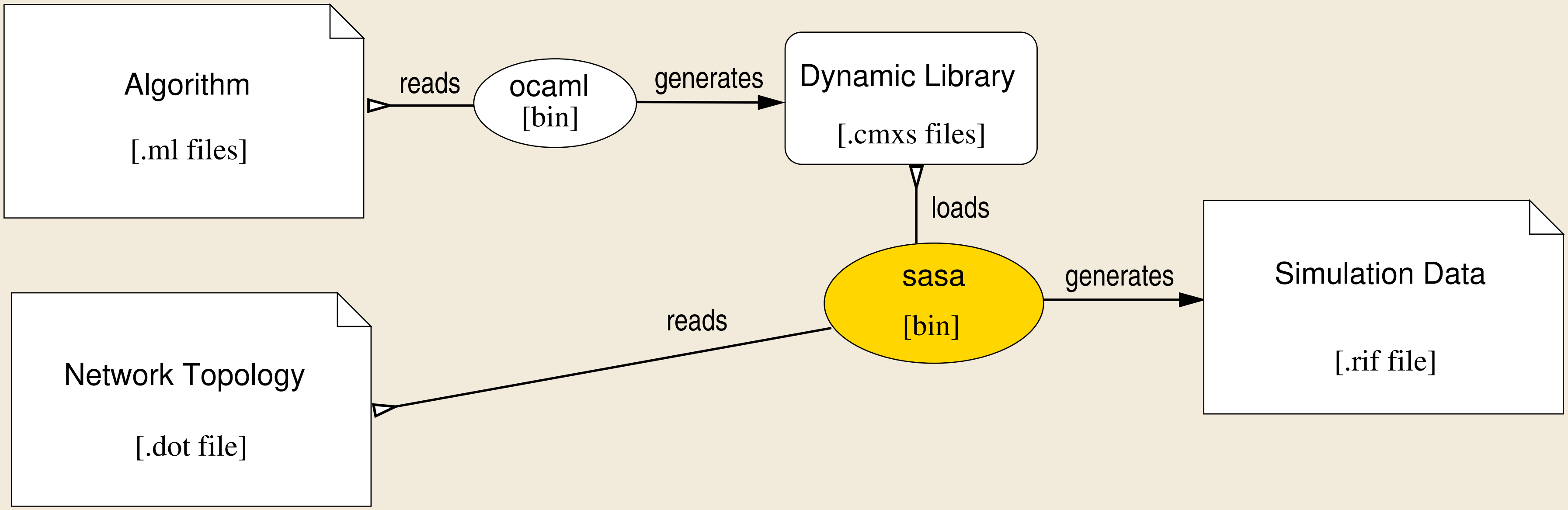
37 straightforward loc

Algorithm Programming Interface (3/4)

Registration

```
type 's algo_to_register = {
  algo_id    : string;
  init_state: int -> 's;
  enab       : 's enable_fun;
  step       : 's step_fun;
  actions    : action list option  }
type 's to_register = {
  algo : 's algo_to_register list;
  state_to_string: 's -> string;
  state_of_string: (string -> 's) option;
  copy_state: 's -> 's  }
val register : 's to_register -> unit
```

The SASA Core Simulator Architecture



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 :: $p.v = p.Pred.v \hookrightarrow p.v \leftarrow (p.v + 1) mod\ K$

```
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 ["T"] 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:
  $p.v \in \{0, ..., K-1\}$
- Action:
  T :: $p.v \neq p.Pred.v \hookrightarrow p.v \leftarrow p.Pred.v$

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

```
cd test/dijksra; rdbg -sut "sasa
ring.dot -distributed-demon"
```

Coloring Algo

- 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)
  $\hookrightarrow p.c \leftarrow min(Free(p))$

```
open Algo
let k=3
let init_state _ = Random.int k
let neigbhors_vals nl = List.map (fun n -> state n) nl
let confl v nl = List.mem v (neigbhors_vals nl)
let free nl =
 let confll = List.sort_uniq compare (neigbhors_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 [] confll 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"]
```

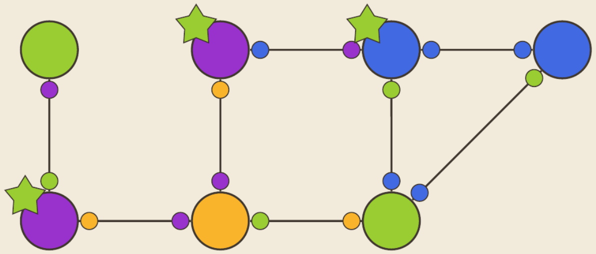
```
cd test/coloring; rdbg -sut "sasa
grid4.dot -locally-central-demon"
```

Plan

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`)



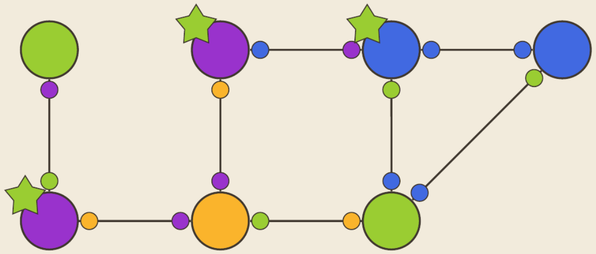
Algorithms in the ASM viewed as Reactive programs

loop:

1. Reads neighbors vars
2. Computes `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)



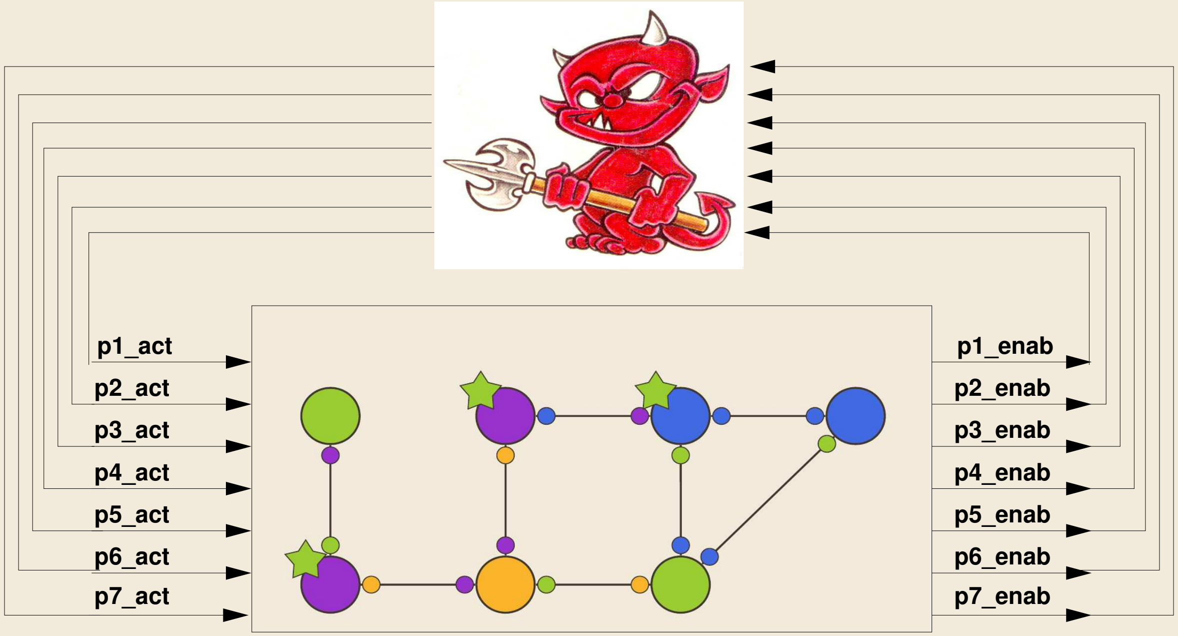
Algorithms in the ASM viewed as Reactive programs

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

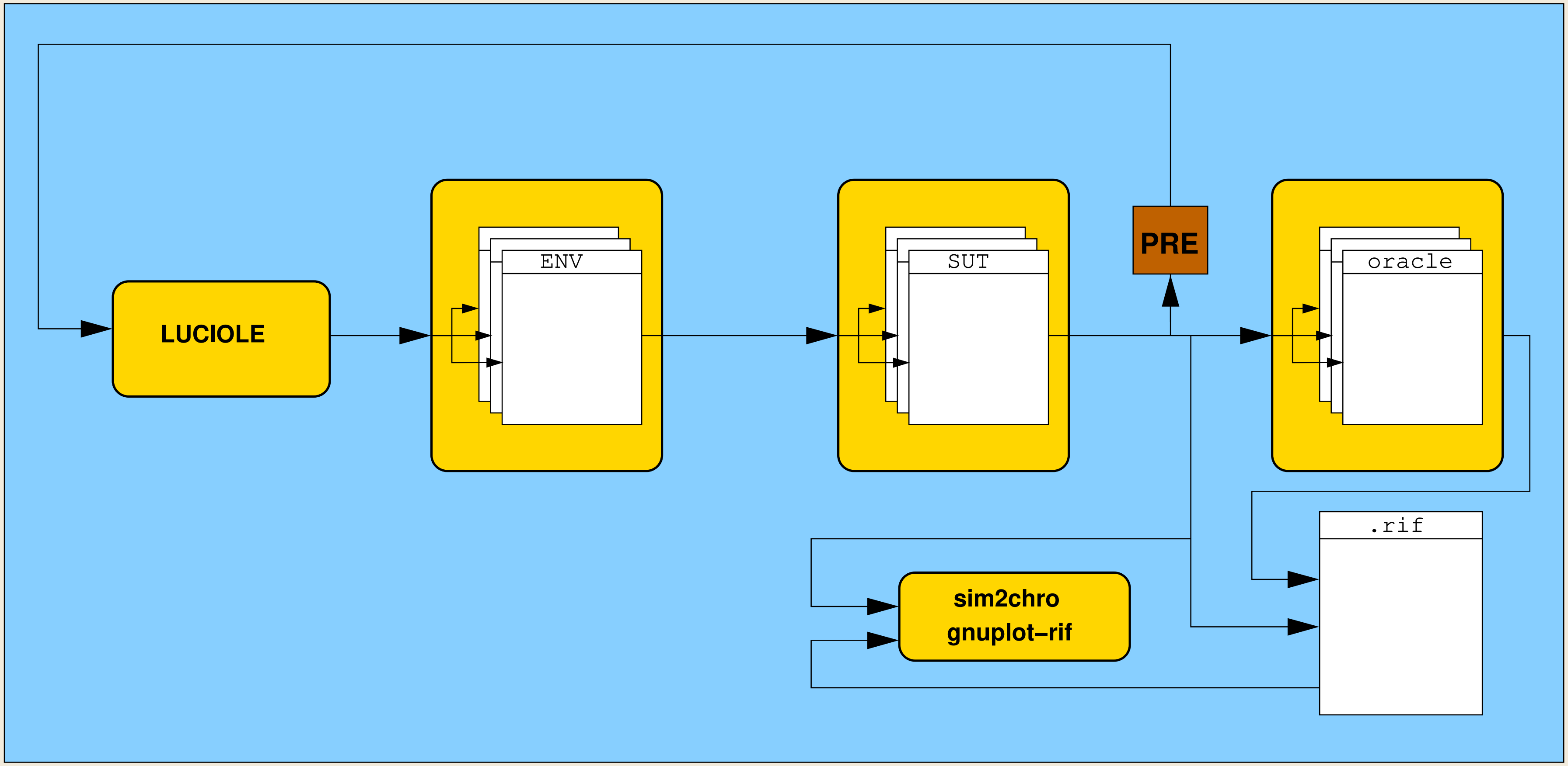


Figure: The LURETTE dataflow schema

RDBG

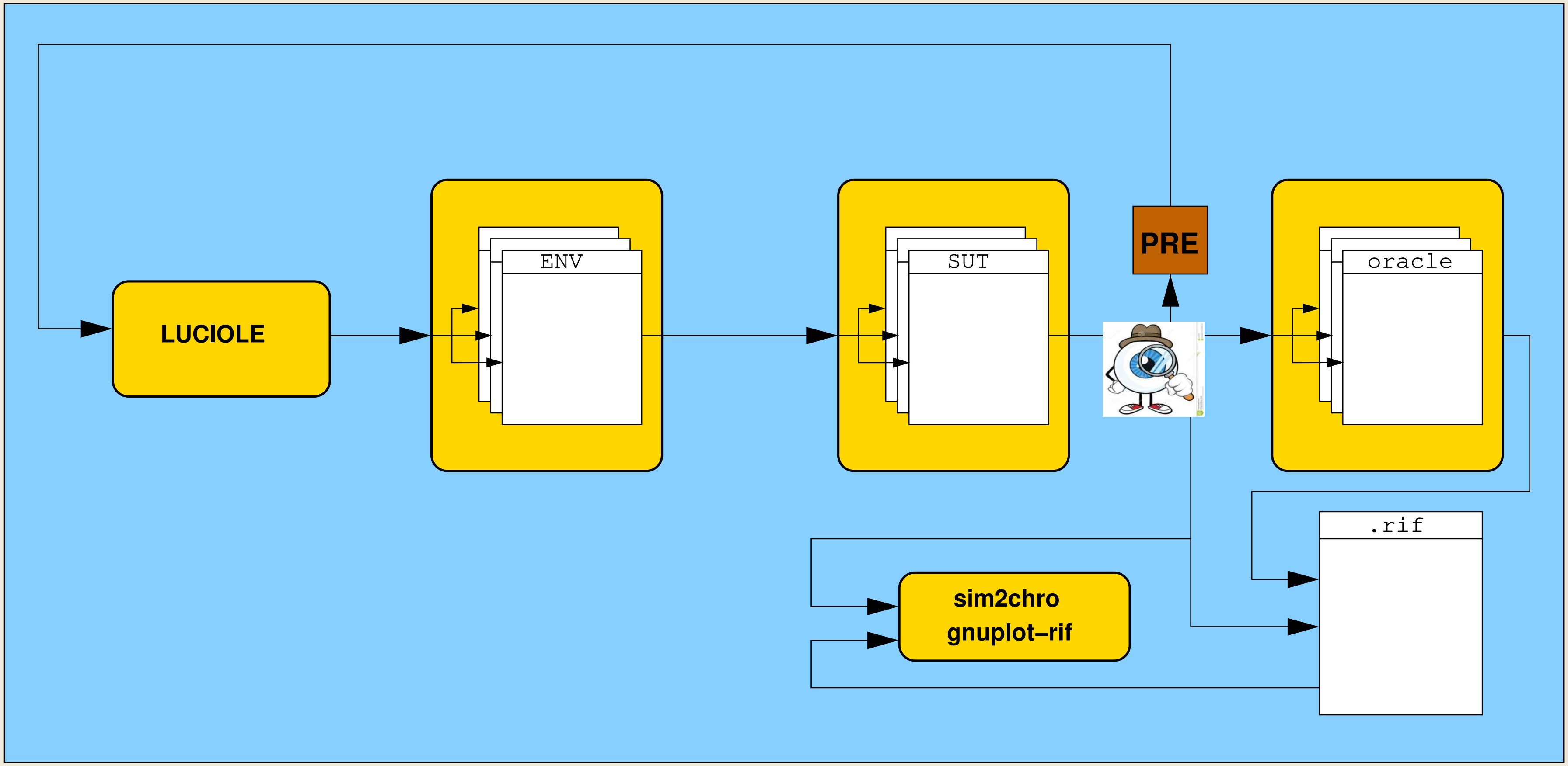


Figure: The RDBG dataflow schema

RDBG

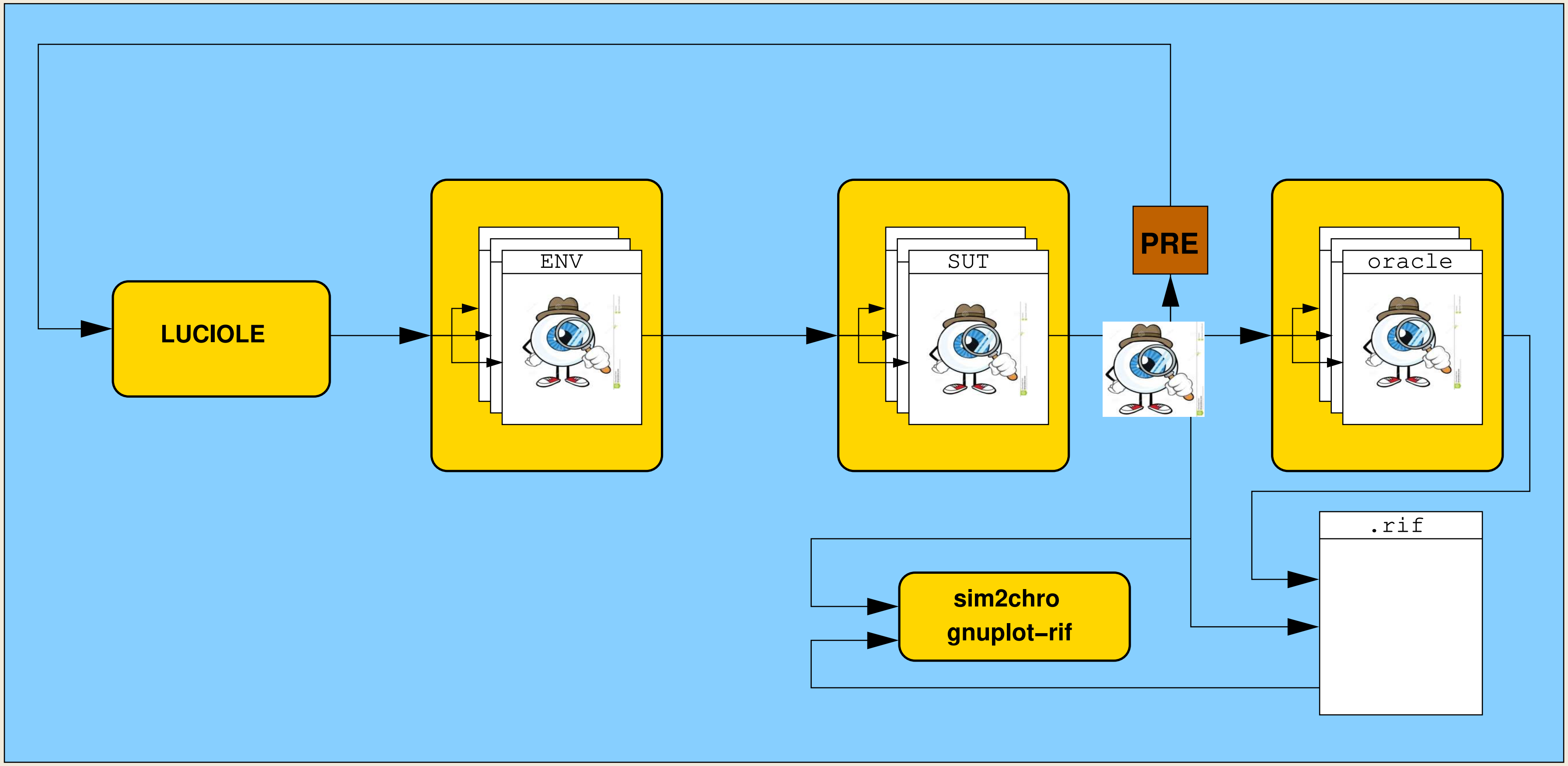


Figure: The RDBG dataflow schema

Lurette and Test Oracles

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

```
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"
```

RDBG

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`

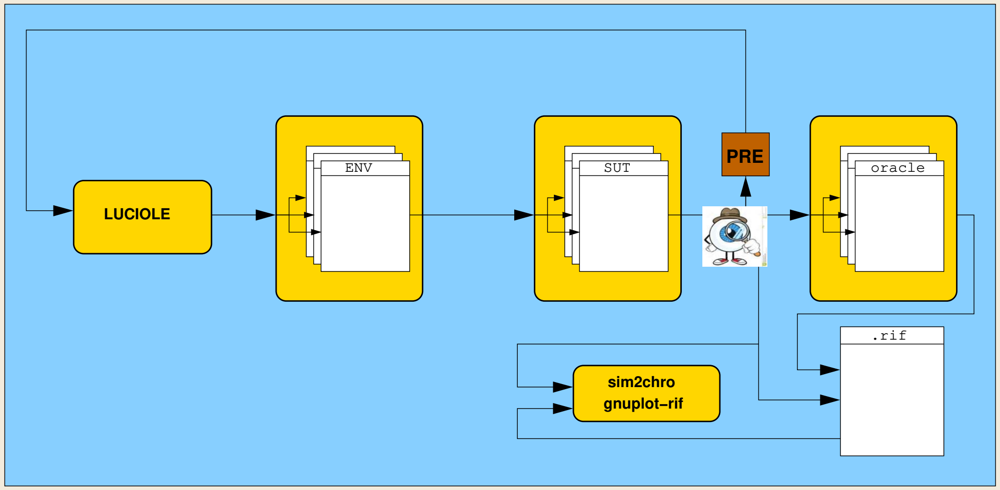
RDBG

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.

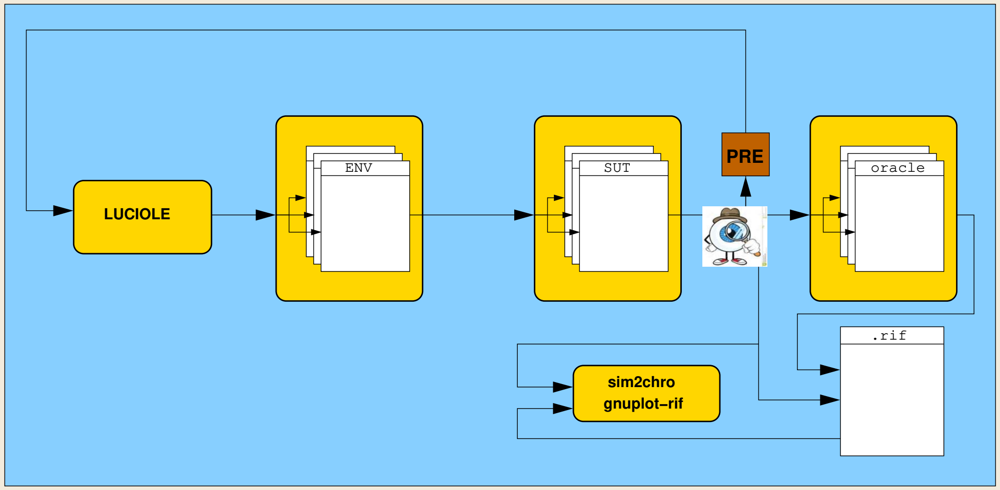
`http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/rdbg/README.html`

RDBG and SASA



- 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, 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

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

Plan

Performance Evaluation: Benchmarks Algorithms

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 d istributed 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.

	`grid.dot`		`ER.dot`		`biggrid.dot`		`bigER.dot`	
	Time/step	Mem	Time/step	Mem	Time/step	Mem	Time/step	Mem
BFS	0.2 ms	13 MB	10.6 ms	49 MB	2.04 s	83 MB	3.03 s	1062 MB
DFS-l	1 ms	44 MB	144.7 ms	63 MB	2.57 s	92 MB	15.83 s	1062 MB
DFS-a	0.5 ms	39 MB	94.3 ms	170 MB	7.64 s	6642 MB	86.93 s	29 945 MB
COL	0 ms	7 MB	35.8 ms	63 MB	27.93 s	75 MB	16.81 s	1083 MB
SYN	0.3 ms	38 MB	10.9 ms	63 MB	887.05 s	874 MB	13.58 s	1099 MB
ASY	0.1 ms	38 MB	4.5 ms	63 MB	0.03 s	83 MB	2.82 s	1115 MB

- 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. Simulation of Self-stabilizing Algorithms
3. SASA
4. Integration with Synchronous tools
5. Performance Evaluation
6. **Some Design Choices**
7. Conclusion

Polymorphic versus Variant Type

- An alternative to polymorphism to hold processes local state:

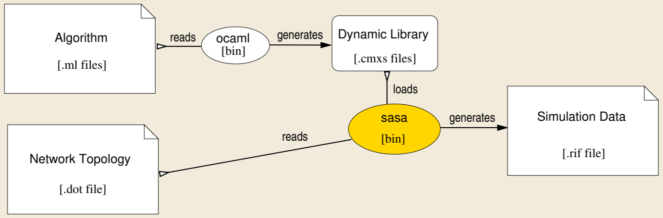
```
type value = I of int | F of float | B of Bool | A of state array | ...
type env = string -> value
```

But:

- What if one need a type that is not in this variant list?
- Variable values need to be set/get in/from the env^t all the time.

```
let step_f c nl a =             let step_f env nl a =
   match a with                    match a with
    | "I" -> modulo (c + 1) k       | "I" ->
    | "R" -> 0                             let c_val = match env_get env "c" with
                                           | I i -> i
                                           | _ -> assert false
                                       in
                                       set_env env "c" (I(modulo ((c_val)+1) k))
                                    | "R" -> set_env env "c" (I 0)
```

Dynamic versus Static Linking



- Dynamic Linking: Pros
  - Easier to use
  - Save Disk space
  - Separation of concerns: user code only depends on a simple API
- Dynamic Linking: Cons
  - Can not be combined gently with Polymorphic values!

Dynamic Type Checking of Polymorphic Nodes

- Dynamic linking in OCAML needs to be done via imperative tables
    - The code to be linked registers functions into tables
    - The main executable reads the tables of functions
- But storing polymorphic values into a mutable data-type is not possible in ML-like languages; one can only store so-called weakly polymorphic values!
- Weak variables can't escape the scope of a compilation unit

`https://ocamlverse.github.io/content/weak_type_variables.html`

Dynamic Type Checking of Polymorphic Nodes

- Solution: use the (evil) `Obj` module:
  - `Obj.obj: 'a -> t`: to register polymorphic functions into tables
  - `Obj.repr: t -> 'a`: to retrieve them from the simulation engine
- Using `Obj` breaks type safety: how to prevent users to register functions of different type?

Dynamic Type Checking of Polymorphic Nodes

- Solution: use the (evil) `Obj` module:
  - `Obj.obj: 'a -> t`: to register polymorphic functions into tables
  - `Obj.repr: t -> 'a`: to retrieve them from the simulation engine
- Using `Obj` breaks type safety: how to prevent users to register functions of different type?

By forcing all functions to be registrated at the same time:

```
type 's algo_to_register = {
  algo_id : string;
  init_state: int -> 's;
  enab : 's enable_fun;
  step : 's step_fun;
  actions : action list option }
```

Dynamic Type Checking of Polymorphic Nodes

- Solution: use the (evil) `Obj` module:
  - `Obj.obj: 'a -> t`: to register polymorphic functions into tables
  - `Obj.repr: t -> 'a`: to retrieve them from the simulation engine
- Using `Obj` breaks type safety: how to prevent users to register functions of different type?

By forcing all functions to be registrated at the same time:

```
type 's algo_to_register = {
  algo_id : string;
  init_state: int -> 's;
  enab : 's enable_fun;
  step : 's step_fun;
  actions : action list option }
type 's to_register = {
  algo : 's algo_to_register list; (* <==== ALL AlGO HAVE THE SAME TYPE! *)
  state_to_string: 's -> string;
  state_of_string: (string -> 's) option;
  copy_state: 's -> 's }
val register : 's to_register -> unit
```

Plan

Conclusion

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

Conclusion

- An open-source SimulAtor of Self-stabilizing Algorithms
- writen 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
- writen 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
- Installation via
  - `docker`
  - `opam`
  - `git`

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