PADEC
A Framework for Certified Self-Stabilization

Karine Altisen, Pierre Corbineau, Stéphane Devismes,

Univ. Grenoble Alpes, CNRS, Grenoble INP[1], VERIMAG, 38000 Grenoble, France

October, 2017

[1] Institute of Engineering Univ. Grenoble Alpes

Proving Self-stabilization

From [Lamport, 2012],

"proofs are still written in prose pretty much the way they were in the 17th *century. [...]"*
"proofs are unnecessarily hard to understand, and they encourage sloppiness that leads to errors."

More complex

- Algorithms
- Topologies,
- Scheduling assumptions
- ...

$\Longrightarrow$ **Transition to automated proof-checking**

The Coq Proof Assistant

- Functional language for definitions
- Interactive proof-editing
- Automated proof-checking

Coq has received the **ACM Software System 2013 award.**

Example Applications:

- System proofs
  - CompCert certified C compiler [X.Leroy *et al.*]
- Mathematical proofs
  - Four-color theorem [G. Gonthier *et al.*]

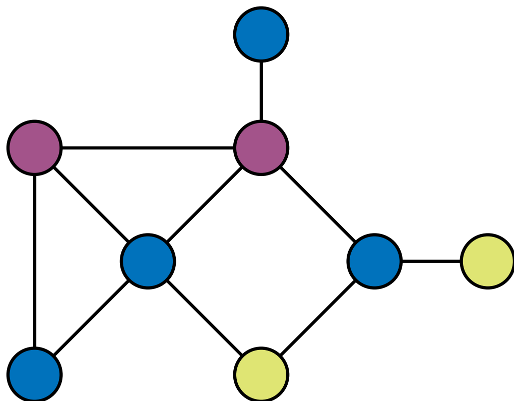
PADEC Project

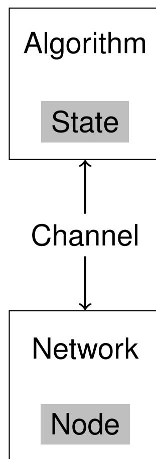
«Preuves d'Algorithmes Distribués En Coq»
"Proofs of Distributed Algorithms with Coq"

- **Goal:** Formal proofs for distributed *self-stabilizing* algorithms.
- **Formalism:** Coq and its libraries as a foundation
- **PADEC provides a Coq library including:**
  - Computational Model
  - Lemmas corresponding to common proof patterns.
  - Case-studies.

Distributed System

Distributed System = *Network* + *Algorithm*
both communicate via *Channel*

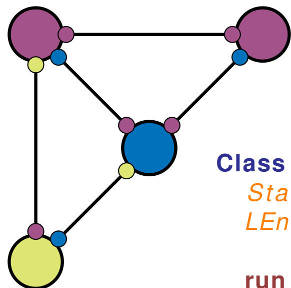




Network and Topology

```
Class Network (Channel: Type): Type := mkNet {
  Node: Type;
  peer: Node → Channel → Node ∪ { ⊥ };
  is_channel n1 c12 n2 := (peer n1 c12) = (Some n2);

  peers: Node → list Channel;
  peers_spec: ∀ n1 c12,
    (c12 ∈ peers n1) ⟺ ∃ n2, (is_channel n1 c12 n2);

  ρ: Node → Channel → Channel;
  ρ_spec: ∀ n1 n2 c12 c21,
    (is_channel n1 c12 n2 ∧ is_channel n2 c21 n1)
    → (ρ n1 c12) = c21;

  all_nodes: list Node;
  all_nodes_prop: ∀ n, n ∈ all_nodes
}.
```

Locally Shared Memory Model



```
Class Algorithm (Channel:Type) := mkAlgo {
  State: Type;
  LEnv := Channel → State ∪ { ⊥ };

  run: list Channel → (Channel → Channel)
       → State → LEnv → State ∪ { ⊥ };

  (* use: (run peers ρ state neigh_states) *)

  ROState: Type; RO_part: State → ROState;
  RO_stable: (* ROState cannot be overwritten *)
      ...
}.
```

Functional Representation of Algorithm

Operational Representation

Variables:

$n \in \mathbb{N}$

...

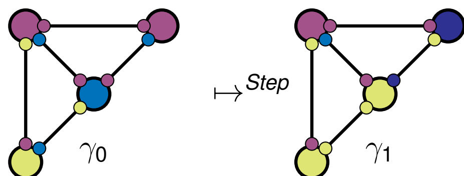
Actions:

$\text{Guard}_1 \to \text{Assign}_1$
$\text{Guard}_2 \to \text{Assign}_2$
...

Functional Representation

```
Record state := mkState {
  n: nat;
  ...
}.

run peers ρ s ℓ :=

       Assign_1 s  if (Guard_1 s ℓ)
else   Assign_2 s  if (Guard_2 s ℓ)

else   ...

else   ⊥
```

Relational Semantics



Configuration (the state of every node):

γ_0 : *Env* *Env* := *Node* $\rightarrow$ *State*

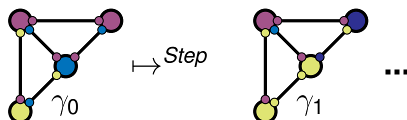
Step of execution `Step` γ_1 γ_0 `Step`: *Env* $\rightarrow$ *Env* $\rightarrow$ **Prop**

For all node n,

- $\gamma_1(n) = \gamma_0(n)$ OR
- `run` returns a state s' and $\gamma_1(n) = s'$

$\gamma_0 <> \gamma_1$

Relational Semantics (2)



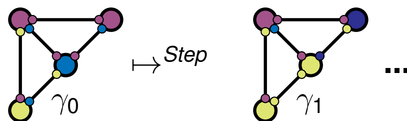
Execution: $e = \gamma_0 \mapsto^{Step} \gamma_1 \mapsto \ldots$

$e = \gamma_0 \mapsto^{Step} \gamma_1 \mapsto^{Step} \ldots \gamma_T$ (γ_T is *terminal*)

`is_exec e` `is_exec: Exec → Prop`

```
CoInductive Exec: Type :=
| e_one:  Env → Exec | e_cons: Env → Exec → Exec.

CoInductive is_exec: Exec → Prop :=
| i_one: ∀ (g: Env), terminal g → is_exec (e_one g)
| i_cons: ∀ (e: Exec) (g: Env),
    is_exec e → Step g (Fst e) → is_exec (e_cons g e).
```

Relational Semantics (3)



Daemon

- No more constraint → **Unfair Daemon**
- **Weakly Fair Daemon:** every enabled node is eventually executed (or neutralized)

  `weakly_fair e`   `weakly_fair:` *Exec* → **Prop**

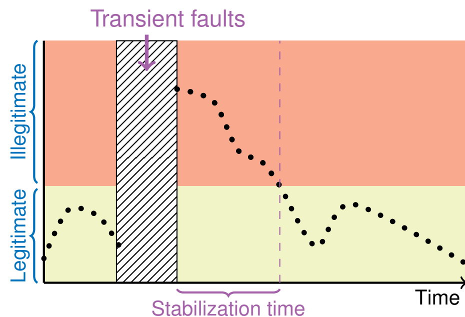
```
weakly_fair e := ∀ (n: Node),
  Always (fun e' => enabled n e'
                  → Eventually (act_neut n) e') e.
```

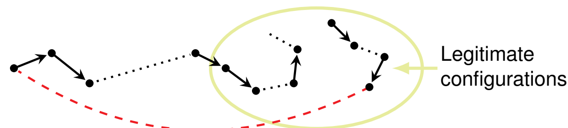
Eventually / Always Operators

```
Inductive Eventually (P: Exec → Prop): Exec → Prop :=
| eventually_now: ∀ e, P e → Eventually P e
| eventually_later: ∀ g e, Eventually P e →
                              Eventually P (e_cons g e).
```

```
CoInductive Always (P: Exec → Prop): Exec → Prop :=
| always_one: ∀ g, P (e_one g) → Always P (e_one g)
| always_cons: ∀ g e, P (e_cons g e) →
                         Always P e → Always P (s_cons g e).
```

Self-Stabilization





- Convergence
- Closure
- Spec. ok

Specification (2)

closure $\mathcal{L}$:= $\forall\ \gamma\ \gamma'$,
  Assume_RO $\gamma \rightarrow \gamma \in \mathcal{L} \rightarrow$ Step $\gamma'\ \gamma \rightarrow \gamma' \in \mathcal{L}$.

convergence $\mathcal{L}$:= $\forall$ e,
  Assume_RO (Fst e) $\rightarrow$ is_exec e $\rightarrow$
  Eventually (fun e => (Fst e) $\in \mathcal{L}$) e.

spec_ok $\mathcal{L}$ *SP* := $\forall$ e,
  Assume_RO (Fst e) $\rightarrow$ is_exec e $\rightarrow$ (Fst e) $\in \mathcal{L} \rightarrow$ *SP* e.

self_stab *SP* := $\exists\ \mathcal{L}$,
  closure $\mathcal{L}$ $\wedge$ **convergence** $\mathcal{L}$ $\wedge$ **spec_ok** $\mathcal{L}$ *SP*.

k-Clustering Algorithm

Competitive self-stabilizing k-clustering
Datta, A.K., Larmore, L.L., Devismes, S., Heurtefeux, K., Rivierre, Y., TCS (2016)

Self-stabilizing algorithm for *k*-clustering, from rooted spanning tree

- 3-rule algorithm
- Proof of convergence + closure + spec. ok
- + Quantitative guarantee: bound on the number of clusters

Tools for Convergence

$\rightarrow$ Use a potential function *Pot* on configurations
and a well-founded order $<$ st:

$$\forall\ \gamma_1\ \gamma_2,\ \texttt{Step}\ \gamma_2\ \gamma_1\ \rightarrow\ Pot\,\gamma_2 < Pot\,\gamma_1$$

Usually: aggregating local potential values at all nodes

- Sum of potentials at all nodes (integer values)
- Multiset of potentials at all nodes (arbitrary ordered values)

Tools for Convergence (2)

Finite Multiset ordering: To obtain M_1 smaller than M_2

- remove some copies of big values from M_2
- replace them with any number of smaller values in M_1

This finite multiset ordering is *well-founded,*
(provided that the value ordering relation is well-founded)
[Dershowitz,Manna 1979]
Coq proof: [CoLoR Library, 2011]

Simplified criteria: during a step,

- potential must change at some node and
- when a node increases its potential,
  there must be another node with higher potential whose potential decreases
  (alibi/scapegoat node)

Quantitative Properties

- Comparison of arbitrary set cardinalities
  - Witnessed by an injective functional relation between elements
- Counting of elements by comparison to $\{0, \ldots, n-1\}$
- Effect of set-theoretic operators on cardinality:
  - intersection, union, product,
  - set comprehension, inclusion
  - singleton, empty set
  - logical operators on comprehension predicates

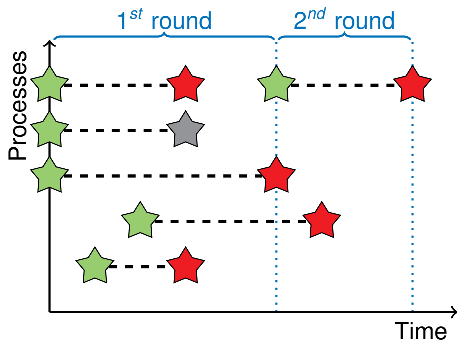
Work in progress

Non silent algorithms

- Express and prove fairness properties,
- Token circulation

Complexity

- Steps
- Rounds



Thank you!

Any Question?

PADEC website:

`http://www-verimag.imag.fr/˜altisen/PADEC/`

A Framework for Certified Self-Stabilization.
Karine Altisen, Pierre Corbineau, Stéphane Devismes
Logical Methods in Computer Science
(special issue of FORTE 2016)
(To appear)