PADEC
Interactive Proof for Self-Stabilizing Algorithms

Karine Altisen, Pierre Corbineau, Stéphane Devismes

How to Gain Confidence into Distributed Algorithms?

Why? Complex statements:
Algorithms, Topologies, Scheduling assumptions...

Pen&paper Proof (usual practice)

Proof = artifact to *convince of the validity* of an assertion

From **[Lamport, How to Write a 21st Century Proof, 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."

How to Gain Confidence into Distributed Algorithms?

Pen&paper Proof (usual practice) → prone to error?

Test, Simulation → few pattern cases

Verification, e.g. Model-Checking → scaling

Machine-checked Proof (proof assistant) → heavy development

Challenges:

→ correctness, few convergence

→ very few quantitative properties

→ no complexity

➔ **PADEC**

A Coq Framework to Prove *Self-stabilizing* Algorithms in the *Atomic State Model (ASM)*

The PADEC Project

«Preuves d’Algorithmes Distribués En Coq»

"Proofs of Distributed Algorithms using Coq"

- **Goal:** Formal proofs for *self-stabilizing* distributed algorithms in the *Atomic State Model (ASM)*
- **Formalism:** *Coq* and its libraries as a foundation

PADEC provides a Coq library including:

- General tools
- Computational model and specifications
- Lemmas corresponding to common proof patterns
- Case-studies

The Coq Proof Assistant: Functional Programming and Formal Proofs

- Functional and formal language with mathematically defined semantics
- For definitions and proofs
- Interactive proof-editing
- Automated checking of formal proofs

Success Stories:

- System proofs: CompCert, certified C compiler
- Mathematical proofs: Feit-Thompson theorem

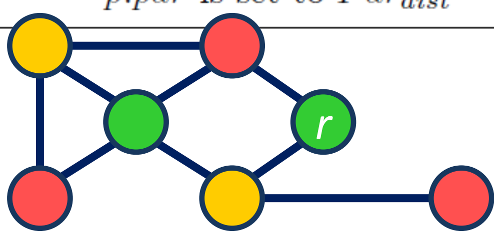
Coq uses the same formal language for *programs* and for *proofs*

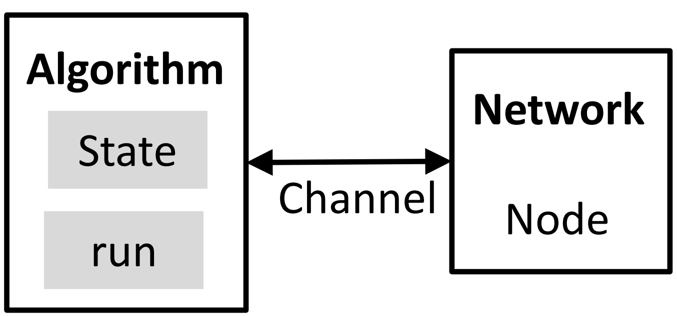
program	proof
type	logical statement
type checking	proof checking
programming	proving

PADEC – Short How To

Algorithm 1 Algorithm BFS, code for each node p.

Constant Local Input: $p.neigh \subseteq Node$; $p.root \in \{t, f\}$
Local Variables: $p.d \in \mathbb{N}$; $p.par \in Node$
Macros:
$Dist_p = \min\{q.d + 1, q \in p.neigh\}$
$Par_{dist} = \text{fst } \{q \in p.neigh, q.d + 1 = p.d\}$
Algorithm for the root(`p.root = true`)
`Root` Action: if $p.d \neq 0$ then
$p.d$ is set to 0
Algorithm for any non-root node(`p.root = false`)
CD Action: if $p.d \neq Dist_p$ then
$p.d$ is set to $Dist_p$
CP Action: if $p.d = Dist_p$ and $p.par.d + 1 \neq p.d$ then
$p.par$ is set to Par_{dist}





Instantiate ***Algorithm***:
- State = a record of local var.
- run = a faithful translation

Express ***Assumption***:
- **Daemon** *e.g., weakly fair*
- **Network**, *e.g. rooted, bidir, connected*

Express ***Specification***:
- **Self-stabilizing** w.r.t. a **problem** *e.g., BFS spanning tree*
- **Complexity**, *e.g. convergence requires at most (Diameter+2)* ***Rounds***

Prove it!!

Definitions

Examples

PADEC – Big Picture

Common proof patterns & results

Case studies

***Libraries**:*
Setoid support
Streams, LTL,
Counting

Computational Model
ASM Semantics

Assumptions
- **Daemons**
- **Networks**

Unfair, weakly fair, synchronous

Connected, ring, tree
Identified, (semi-)anonymous
Measures (distance, diameter)

Relational <-> Functional

Specification
- **Self-Stabilization**
- **Problem**
- **Complexity: Steps, Rounds**

BFS spanning tree
Token circulation
Dominating set, Clustering

Induction Schema

BFS spanning tree (rounds)
Dijkstra Token Ring (steps)

Composition
- Hierarchical Collateral

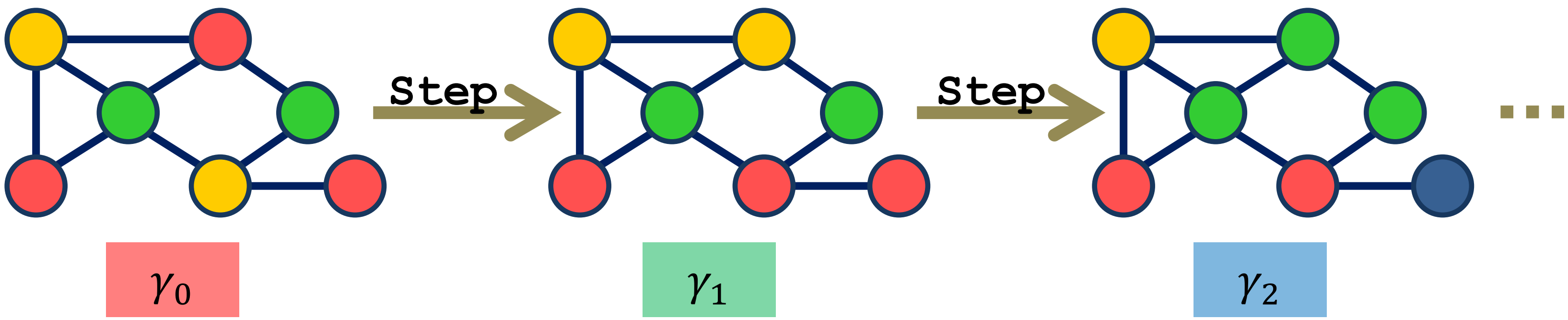
Proof of
- Specification
- Complexity

BFS + KClustering

Tools for convergence
Lexico, Well-founded,
Potential & multisets

KDomSet, KClustering

Computational Model – ASM Semantics



Configuration γ_i **:** `Env` (state of all nodes `Env := Node -> State`)

Atomic step - read local & neighbor variables → enabled?
- daemon selection
- node computation → update local variables

relation $\xrightarrow{\texttt{Step}}$ `:= Env -> Env -> Prop`

Streams

Execution `Exec := Stream Env`
such that (*predicate* `is_exec: Exec -> Prop`)
- Each two consecutive configurations are linked by $\xrightarrow{\texttt{Step}}$
- if the stream is finite, the last configuration is *terminal*

Relational semantics <-> Functional semantics

Relational semantics: an execution is defined by any `e: Stream` **`Env`** such that `is_exec e`

Functional Semantics: defines an execution by

-> an initial configuration γ and

-> a daemon: selects the set of nodes to execute at each step, defined as an (infinite) stream of selections

`build_exec (γ: Env) (daemon: Daemon): Exec`

Equivalence between both semantics:

- *Soundness:*

  `∀daemon γ, is_exec (build_exec γ daemon).`

- *Completeness:*

  `∀e, is_exec e →`
  `∃daemon, e =~= build_exec (Head_of e) daemon.`

Setoid support

Configurations are functions: γ`: Env` and `Env := Node -> State`

In former implementations, configurations were lists of states.

=> Proof depends on the order of elements and repeats

=> *heavy* additional developments

Need: equality on functions

to be able to express γ_0 is the same configuration as γ_1

γ_0 `=~=` γ_1 `<-> ∀x,` γ_0 `x ==` γ_1 `x`

Difficulty: Default Coq equality = *Leibniz Equality*

= proof (program) equality = *intentional* equality

not satisfactory for functions!!

Example:

`(fun x => x)` is not "Leibniz-equal to" `(fun x => x + 0)`

Reduced form for `(fun x => x + 0)`: `(fun x => match x with 0 => 0 | S x' => S (x' + 0) end)`

Reduced form for `(fun x => x)` is itself.

Not synctactically equal!! (upto renaming)

Setoid Support

⇒ **In PADEC, every type has a *user-defined equality*.**

- **Base-type:** *equivalence relation on* `Node` *and* `State`
- **Function type:** e.g. `Env := Node -> State`

  γ_0 `=~=` γ_1 `<-> ∀n n′: Node,` γ_0 `n =` γ_1 `n′`

  → NOT reflexive in general!! *Partial equivalence relation*

=> Proofs are restricted to proper objects, e.g. such that γ `=~=` γ

=> Explicitely defined functions have to be proved compatible

`∀x y, x=~=y -> f x =~= f y`

Other types that are incompatible with Leibniz equality:

- Coinductive types: Executions `Exec`
- Comprehensions: e.g. natural numbers such that… `{n: nat | … }`
- …

Setoid Support:

Type classes mechanism → automation

Library for relations on datatypes

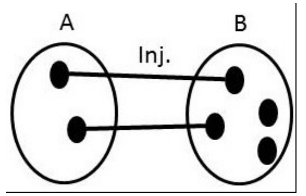
Assumptions about Networks

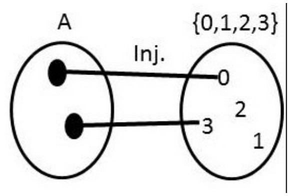
Networks

- Basic properties (bidirectional, connected, rooted)
- Topologies (ring, tree)
- Measures (number of nodes, distance, diameter)

Counting

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





=> Let *n* be the number of nodes …

=> Diameter is smaller that *n*

…

Express and prove results about Quantitative Properties and Complexities

Assumptions about Daemons

Daemon – models the asynchronism in the ASM model

In PADEC: a *predicate* over executions `Exec -> Prop`

Classical daemons are available in PADEC:

unfair, weakly fair, synchronous…

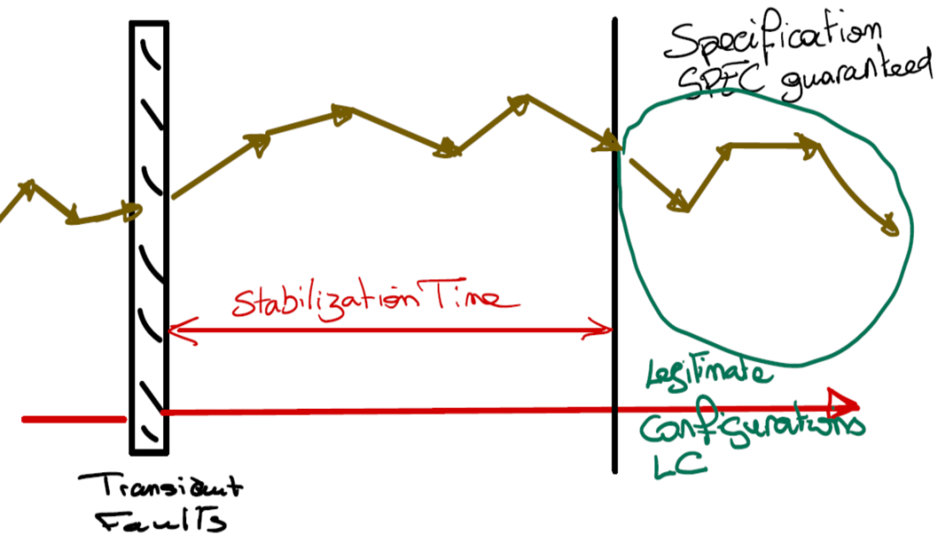
```
unfair e := True (* no constraint *)

weakly_fair e := (* a node which is enabled is eventually activated or
                   neutralized, and this forever *)
   ∀p, Always (fun e => EN p e -> Eventually (AN p) e) e
```

LTL Library

- *Linear Temporal Logic*
- *Defines classical LTL Operators*
- *On Type* **`Exec`**

Specification – Self-Stabilization



Defined w.r.t. a problem specification

`SPEC: Exec -> Prop`

`self_stab SPEC :=`
`∃LC: Env -> Prop,`
`∀e,`

Closure: if `e` starts in `LC` then `Always` `e` remains in `LC`

Convergence: `Eventually` `e` reaches `LC`

Specification: if `e` starts in `LC` then `SPEC e`

Tools for *Convergence :*

- Lexicographic ordering,
- Well-foundedness,
- Potential & Multiset ordering

Tools for Convergence

Well-foundedness

prove that relation $\xrightarrow{\texttt{Step}}$

outside `LC` and restricted to **Assumptions**

is Well-Founded

(every decreasing sequence is finite)

Potential

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

$\forall\, \gamma_0\, \gamma_1,\ \gamma_0 \xrightarrow{\texttt{Step}} \gamma_1 \texttt{ -> } \textbf{\textit{Pot}}\ \gamma_1 < \textbf{\textit{Pot}}\ \gamma_0$

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: Local Potentials

Multiset of potentials at all nodes

Finite Multiset ordering: [Dershowitz,Manna 1979]

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)

Coq Support: [CoLoR Library]

Local Potential (at each node)

Simplified criteria: during a step, $\xrightarrow{\texttt{Step}}$

- 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)*

Specification – Problem - Complexity

Problems

- *BFS spanning tree*
- *Token circulation*
- *K-Clustering* and quantitative property on the number of clusters

Expressed in `SPEC: Exec -> Prop`

Complexity measures

- **Steps** (number of atomic steps in executions) *Dijkstra Token Ring (steps)*
- **Rounds** *BFS spanning tree (rounds)*

Induction Schema – *e.g. (simplified):*

`P(n): Exec -> Prop` `e: Exec`

If ∀`e`, ∀`n` ≤ `B`, `P(n) e -> e` reaches `P(n+1)` in at most one Steps/Rounds
If `P(0) e` holds
Then `e` reaches `P(B)` in at most `B` Steps/Rounds

Hierarchical Collateral Composition

A1;A2

A1 assumes **H1**
is self-stabilizing w.r.t. **SPEC1** and terminates (silent)

A2 shares variables with **A1** but cannot overwrite them
assumes **SPEC1**
is self-stabilizing w.r.t. **SPEC2**

weakly fair daemon (so that **A1** can converge)

Proof of specification: A1;A2 is self-stabilizing, w.r.t. SPEC2 assuming **H1**
(convergence is quite tricky)

Proof of complexity: *(WIP)*

Comments and Lessons

PADEC: a Coq Framework to prove Self-Stabilizing Algorithms

General Model: (not dedicated to a particular case)
Atomic State Model, Daemons, …
→ Close to designer

Reasoning on formal proof: as close as possible of the pen&paper proof
→ Get rid of generality using simplifying tools!

Generic powerful tools: counting, slices, graph properties…

Formal proofs: strengthen assumptions; develop new proofs
and sometimes bring new results!

PADEC

http://www-verimag.imag.fr/~altisen/PADEC/

#loc = 96k (spec); 33k (proof); 7k (comments)

PADEC
Coq Library

TOC

- Model
- Token Ring
- K-Dominating Set
- K-Clustering
- BFS

TOOLS

- PADEC Index
- Coq Reference
- Back to Main

PADEC - Coq Library

Model and General Results about the Model

- **Algorithm:** network and algorithm definitions
- **RelModel:** semantics of the model (relational version)
- **FunModel:** semantics of the model (functional version and equivalence wrt relational semantics)
- **Exec:** execution of the system (type and support)
- **Self_Stabilization:** definition of the properties
- **Fairness:** definition of scheduling assumptions (daemon)

Tool for Termination or Convergence

- **P_Q_Termination:** tools for proving convergence of an algorithm. Relies on the Dershowitz-Manna order on finite multi-sets to define sufficient conditions on local potentials. In those tools, we use **CoLoR Libray**.

Tools for Composition

- **Composition:** collateral composition - definition, proof of correctness under weakly fair assumptions
- **Compo_ex:** example on how to use the composition operator, based on **"Self-Stabilizing Small k-Dominating Sets"**

Tools for Complexity

- **Steps:** step complexity. Tools to measure stabilization times (and other performances) in steps. Relies on **Stream_Length**