PADEC
A Framework for Certified Self-Stabilization

Karine Altisen, Pierre Corbineau, Stéphane Devismes,

Univ. Grenoble Alpes, CNRS, Grenoble INP\textsuperscript{1}, VERIMAG, 38000 Grenoble, France

October, 2017

\textsuperscript{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
- ...

⇒ 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 \[X.\text{Leroy et al.}\]
- Mathematical proofs
  - Four-color theorem \[G. \text{Gonthier et al.}\]
**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 = \textit{Network} + \textit{Algorithm}
both communicate via \textit{Channel}
Class Network (Channel: Type): Type := mkNet {
    Node: Type;
    peer: Node \rightarrow Channel \rightarrow Node \cup \{ \bot \};
    is_channel n1 c12 n2 := (peer n1 c12) = (Some n2);

    peers: Node \rightarrow list Channel;
    peers_spec: \forall n1 c12, 
            (c12 \in \textbf{peers n1}) \iff \exists n2, (is_channel n1 c12 n2);

    \rho: Node \rightarrow Channel \rightarrow Channel;
    \rho_spec: \forall n1 n2 c12 c21, 
            (is_channel n1 c12 n2 \land is_channel n2 c21 n1) 
            \rightarrow (\rho n1 c12) = c21;

    all_nodes: list Node;
    all_nodes_prop: \forall n, n \in \textbf{all_nodes}
}.
Class Algorithm (Channel: Type) := mkAlgo {
  State: Type;
  LEnv := Channel \rightarrow State \cup \{ \bot \};

  run: list Channel \rightarrow (Channel \rightarrow Channel)
    \rightarrow State \rightarrow LEnv \rightarrow State \cup \{ \bot \};

  (*use: (run peers \rho \state \neigh\_states) *)

  ROState: Type; RO_part: State \rightarrow ROState;
  RO_stable: (* ROState cannot be overwritten *)

  ...}

...
Functional Representation of Algorithm

Operational Representation

Variables:
\[ n \in \mathbb{N} \]
\[
\ldots
\]

Actions:
\[ \text{Guard}_1 \rightarrow \text{Assign}_1 \]
\[ \text{Guard}_2 \rightarrow \text{Assign}_2 \]
\[
\ldots
\]

---

Functional Representation

Record \( \text{state} := \text{mkState} \{ n: \text{nat}; \ldots \} \).

run peers \( \rho \ s \ l := \)

Assign\_1 s \text{ if } (\text{Guard}\_1 s \ l) \text{ else Assign\_2 s if } (\text{Guard}\_2 s \ l) \text{ else } \ldots

else \downarrow
Relational Semantics

Configuration (the state of every node):

\[ \gamma_0 : \text{Env} \quad \text{Env} := \text{Node} \rightarrow \text{State} \]

Step of execution \( \text{Step} \gamma_1 \gamma_0 \) \( \text{Step}: \text{Env} \rightarrow \text{Env} \rightarrow \text{Prop} \)

For all node \( n \),

\[ \gamma_1(n) = \gamma_0(n) \text{ OR} \]

\[ \text{run returns a state } s' \text{ and } \gamma_1(n) = s' \]

\[ \gamma_0 \leftrightarrow \gamma_1 \]
Relational Semantics (2)

Execution: \[ e = \gamma_0 \mapsto_{\text{Step}} \gamma_1 \mapsto \ldots \]  
\[ e = \gamma_0 \mapsto_{\text{Step}} \gamma_1 \mapsto_{\text{Step}} \ldots \gamma_T \text{ (} \gamma_T \text{ is terminal)} \]

\[ \text{is\_exec} \ e \\quad \text{is\_exec:} \quad \text{Exec} \rightarrow \text{Prop} \]

CoInductive \text{Exec: Type} :=
| \text{e\_one: } \text{Env} \rightarrow \text{Exec} | \text{e\_cons: } \text{Env} \rightarrow \text{Exec} \rightarrow \text{Exec}. 

CoInductive \text{is\_exec: Exec} \rightarrow \text{Prop} :=
| \text{i\_one: } \forall \ (g: \text{Env}), \text{terminal } g \rightarrow \text{is\_exec (} \text{e\_one } g \text{)} 
| \text{i\_cons: } \forall \ (e: \text{Exec}) (g: \text{Env}), 
 \text{is\_exec } e \rightarrow \text{Step } g (\text{Fst } e) \rightarrow \text{is\_exec (} \text{e\_cons } g \text{ } e \text{)}.
Daemon

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

\[
\text{weakly\_fair} \ e =: \forall \ (n: \ Node), \quad \text{Always} \ (\text{fun} \ e' \Rightarrow \text{enabled} \ n \ e') \rightarrow \text{Eventually} \ (\text{act\_neut} \ n) \ e' \ e.
\]

weakly\_fair: Exec $\rightarrow$ Prop
Eventually / Always Operators

**Inductive** Eventually (P : Exec \rightarrow Prop) : Exec \rightarrow Prop :=
\begin{align*}
| \text{eventually\_now}: & \forall e, P e \rightarrow \text{Eventually} P e \\
| \text{eventually\_later}: & \forall g e, \text{Eventually} P e \rightarrow \\
& \quad \text{Eventually} P (e\_cons g e).
\end{align*}

**ColInductive** Always (P : Exec \rightarrow Prop) : Exec \rightarrow Prop :=
\begin{align*}
| \text{always\_one}: & \forall g, P (e\_one g) \rightarrow \text{Always} P (e\_one g) \\
| \text{always\_cons}: & \forall g e, P (e\_cons g e) \rightarrow \\
& \quad \text{Always} P e \rightarrow \text{Always} P (s\_cons g e).
\end{align*}
Self-Stabilization

- Convergence
- Closure
- Spec. ok

Legitimate configurations
**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 \((\text{Fst } e) \rightarrow \) is_exec \( e \rightarrow \) Eventually \((\text{fun } e \Rightarrow (\text{Fst } e) \in \mathcal{L} )\) \( e. \)

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

**self_stab** \( SP := \exists \mathcal{L}, \) \
closure \( \mathcal{L} \land \) convergence \( \mathcal{L} \land \) spec_ok \( \mathcal{L} \ SP. \)
**k-Clustering Algorithm**

**Competitive self-stabilizing k-clustering**  

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
→ Use a potential function $\text{Pot}$ on configurations and a well-founded order $<_{\text{st}}$:

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

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)
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
Non silent algorithms

- Express and prove fairness properties,
- Token circulation

Complexity

- Steps
- Rounds

PADEC A Framework for Certified Self-Stabilization
K. Altisen, P. Corbineau, S. Devismes— (19)
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)