Formal Approach to self-stabilizing algorithms

Wishnu Prasetya

Dept. of Information and Computing Sciences
Utrecht University

A multi-disciplinary workshop on dependability of all-IP networks
Espoo, Finland, May 18-19, 2006
Areas:

1. Mechanical verification of distributed algorithms
2. Composition and refinement based approach in distributed systems
3. Architecture and implementation of software verification tools

Software technology group, Utrecht University, Netherlands

Focusing in technologies related to programming languages: compilers, program analysis, program verification.
Outline

Introduction

Formalism

Lentfert’s FSA

Lifting to Hierarchical FSA

Closing
Self-stabilization (Dijkstra, 1974)

Let $Q$ specifies a set of *legitimate* (stable) states of a system $S$; $S$ is self-stabilizing to $Q$ if:

1. *(Progress)* From any state $S$ can progress to some state in $Q$

2. *(Stability)* $Q$ is closed under the execution of $S$. 
UNITY (Chandy & Misra, 1989)-based formalism

We use UNITY programs as models.

Some (non-) features of UNITY logic:

1. can express temporal properties.
2. less expressive than LTL, but operates at a more abstract level (e.g. fairness is built-in).
3. there are extensions: refinement, composition
4. less suitable for model checking, but in this case we’re dealing with algorithms with infinite state space.
Example

\[ \text{prog} \text{ } \text{example} \text{ } = \text{ } P_1 \parallel P_2 \]

\[ \text{prog} \text{ } P_1 \]

read \[ d_1, d_2 \]
write \[ d_1 \]
init \[ true \]
assign \[ d_2 < d_1 \rightarrow d_1 := d_2 \]

Execution model: reactive; each action is atomic; execution is weakly fair.
Formalism

Reasoning over Temporal Properties in UNITY

▶ Safety

\[ p \vdash p \text{ unless } q = (\forall a \in aP :: \{ p \land \neg q \} a \{ p \lor q \}) \]

▶ Progress-1

\[ p \vdash p \text{ ensures } q = p \vdash p \text{ unless } q \]

\[ \text{and } (\exists a \in aP :: \{ p \land \neg q \} a \{ q \}) \]

▶ Progress-general

\[ p \vdash p \rightarrow q = \text{transitive, left-disjunctive closure of } p \text{ ensures } q \]
Expressing Self-stabilization in UNITY

1. Weaken it to 'convergence': \( p \leadsto q = p \rightarrow q \) and \( \circ q \)

2. Take this def. instead:

\[
p \leadsto q = (\exists q_0 :: p \rightarrow q_0 \land q \land \circ (q_0 \land q))
\]
Property of Convergence

▶ Bonus: convergence is conjunctive!

\[ p_1 \leadsto q_1 \text{ and } p_2 \leadsto q_2 \text{ implies } p_1 \land p_2 \leadsto q_1 \land q_2 \]

▶ Give us this rolling-down-the-hill stabilization strategy:

\[
(\forall m : m \to n : q.m) \leadsto q.n
\]

\[
\text{true} \leadsto (\forall n \in A :: q.n)
\]

where \( \to \) is a well-founded relation over a finite domain \( A \) of rounds (altitudes).

Centrally, but we can also do it distributedly.
Lentfert originally worked on an algorithm to distributedly compute minimum distance between any two nodes in a network:

Idea-1: maintain in every node $a$ the variable $d[a][b]$, which eventually should contain what $a$ thinks to be its distance to $b$.

Idea-2: maintain for every $b$ a separate process $P_{a,b}$ in node $a$ to manage $d[a][b]$. 
Lentfert’s FSA Algorithm

\[ d : \text{array } V \times V \text{ of Value} \quad \text{—— data} \]
\[ r : \text{array } V \times V \times V \text{ of Value} \quad \text{—— copies} \]

\[
\text{prog } FSA_{V,N} = (\forall a \in V :: \text{node}_a)
\]

where
\[
\text{prog } \text{node}_a = (\forall b \in V :: \text{process}_{a,b})
\]

\[
\text{prog } \text{process}_{a,b}
\]
\[
\text{read} \quad \ldots \text{write} \quad \ldots
\]
\[
\text{init} \quad \text{true}
\]
\[
\text{assign} \quad d[a][b] := \Phi(a, b, r[a][a][b])
\]

\[
\forall (a' \in N(a) :: r[a'][a][b] := d[a][b]) \quad \text{—— send to } a'
\]
Lentfert’s FSA Theorem

Let $OK$ be a predicate over $V \times V \times Value$. FSA self-stabilizes:

$$\text{true} \implies (\forall a, b \in V :: OK(a, b, d[a][b]))$$

if:

1. Find a finite set $A$ of altitudes, with WF order $\rightarrow$
2. Split $OK$ over $A$: $OK(a, b, val) = (\forall n \in A :: ok(n, a, b, val))$
3. $\Phi$ should push progress down the hill:

$$\forall m, a' : m \rightarrow n \land a \in N(a') : ok(m, a', b, x[a'])$$

$$\Rightarrow$$

$$ok(n, a, b, \Phi(a, b, x))$$
Generalizing FSA to networks with domains

and with hierarchy:

u cannot see x,y ... or any domain under UK
u can see y,B,UK

u can see y,B,UK
u cannot see x,y ... or any domain under
Adding context to UNITY Properties

- Adding a bounding region as a context:

  \[ J \models p \text{ unless } q = \bigcirc J \text{ and } p \models J \land p \text{ unless } q \]

- Access patterns as a context:

  \[ J, V \models p \text{ unless } q = p, q \in \text{Pred}(V) \text{ and } J \models p \text{ unless } q \]

  This specified behavior is insensitive to what \( P \) or its environment does on \( V^C \)!

- Note that context is not static. The environment can cause \( P \) to change context!

- Analogously extend the definition of other operators.
We get compositionality

- For example, to split tasks over write-disjoint components:

\[
P \div Q \quad \text{and} \quad Q \not\triangleright J \quad \text{and} \quad J, w(P) \not\triangleright p \leadsto q
\]

\[
J, w(P \parallel Q) \not\triangleright p \leadsto q
\]

- More general:

\[
J, V \not\triangleright p \leftrightarrow q \quad \text{and} \quad J \not\triangleright V = \text{val unless } q
\]

\[
J, V \not\triangleright p \leftrightarrow q
\]
Lifting to Hierarchical FSA

Domain-level FSA Algorithm

\[ d : \text{array } V \times \text{Dom} \times \text{Dom} \text{ of Value} \quad \text{— data} \]
\[ r : \text{array } V \times \text{Dom} \times \text{Dom} \text{ of Value} \quad \text{— copies} \]

prog \( DFSA_{\text{Dom},\mathcal{N}} \) = (\( \langle A, a : A \in \text{Dom} \land a \in A : \text{node}_{a,A} \rangle \))

where

prog \( \text{node}_{a,A} \) = (\( \langle B \in \text{Dom} :: \text{process}_{a,A,B} \rangle \))
\[ \langle \langle A' \in \mathcal{N}(A) :: \text{broadcast}_{A,A'} \rangle \) \]

prog \( \text{process}_{a,b} \)

... assign \( d[a][A][B] := \Phi(A, B, r[a][-][B]) \)
Closing words

- We have used this formalism to mechanically verify FSA, DFSA, and some instances.
- Lots of technical details in the actual mechanization; but that’s of course not a problem for a machine.
- It’s not a cheap project, but you only need to do it once. More economic when targeting (generic) algorithms.
- Practical value: generate programs from proven algorithms.

- Further reading: