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

Myself

- 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

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.

Sone (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. the are extensions: refinement, composition
4. less suitable for model checking, but in this case we're dealing with algorithms with infinite state space.

Example

$$\textbf{prog}\ \textit{example} \;=\; P_1 \,[\!]\, P_2$$

prog 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.

Reasoning over Temporal Properties in UNITY

- Safety

$$_{P}\vdash p \textbf{ unless } q \quad = \quad (\forall a \in \mathbf{a}P :: \{p \wedge \neg q\}\ a\ \{p \vee q\})$$

- Progress-1

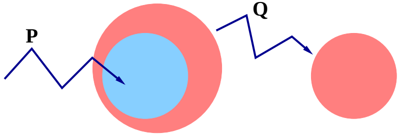
$$_{P}\vdash p \textbf{ ensures } q \quad = \quad \begin{array}{l} _{P}\vdash p \textbf{ unless } q \\ \text{and } \ (\exists a \in \mathbf{a}P :: \{p \wedge \neg q\}\ a\ \{q\}) \end{array}$$

- Progress-general

$$_{P}\vdash p \mapsto q \quad = \quad \text{transitive, left-disjunctive closure of } \textbf{ensures}$$

Expressing Self-stabilization in UNITY

1. Weaken it to 'convergence': $p \leadsto q \;=\; p \mapsto q$ and $\circlearrowleft q$



2. Take this def. instead:

$$p \leadsto q \;=\; (\exists q_0 :: p \mapsto q_0 \wedge q \text{ and } \circlearrowleft (q_0 \wedge q))$$

Property of Convergence

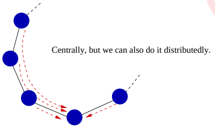
- Bonus: convergence is conjunctive!

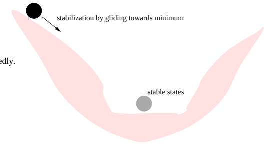
$$p_1 \leadsto q_1 \text{ and } p_2 \leadsto q_2 \text{ implies } p_1 \wedge p_2 \leadsto q_1 \wedge q_2$$

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

$$\frac{(\forall m : m \rightarrow n : q.m) \;\leadsto\; q.n}{\textbf{true} \leadsto (\forall n \in A :: q.n)}$$

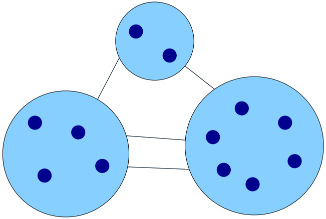
where $\rightarrow$ is a well-founded relation over a finite domain A of rounds (altitudes).





Lentfert's FSA Algorithm

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

$$
\begin{array}{lll}
d & : & \textbf{array } V \times V \textbf{ of } Value \quad \text{— data} \\
r & : & \textbf{array } V \times V \times V \textbf{of } Value \quad \text{— copies}
\end{array}
$$

prog $FSA_{V,N} \;=\; ([\![\; a \in V :: node_a)$

where

prog $node_a \;=\; ([\![\; b \in V :: process_{a,b})$

prog $process_{a,b}$

read ...**write** ...

init **true**

assign $d[a][b] := \Phi(a, b, r[a][-][b])$

$[\![\quad ([\![a' \in N(a) :: r[a'][a][b] := d[a][b])$ — send to a'

Lentfert's FSA Theorem

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

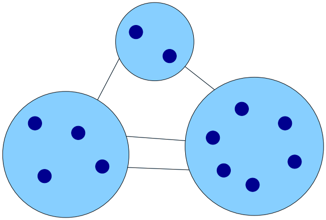
$$\textbf{true} \;\leadsto\; (\forall a, b \in V :: OK(a, b, d[a][b]))$$

if:

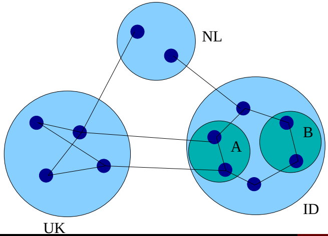
1. Find a finite set A of altitudes, with WF order $\rightarrow$
2. Split *OK* over A: $\quad OK(a, b, val) \;=\; (\forall n \in A :: ok(n, a, b, val))$
3. Φ should push progress down the hill:

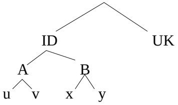
$$(\forall m, a' : m \rightarrow n \wedge a \in N(a') : ok(m, a', b, x[a']))$$
$$\Rightarrow$$
$$ok(n, a, b, \Phi(a, b, x))$$

Generalizing FSA to networks with domains



and with hierarchy:





u 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:

$$J \;{}_P\!\vdash p \,\textbf{unless}\, q \;\; = \;\; \circlearrowright J \;\text{ and }\; {}_P\!\vdash J \wedge p \,\textbf{unless}\, q$$

- Access patterns as a context:

$$J, V \;{}_P\!\vdash p \,\textbf{unless}\, q \;\; = \;\; p, q \in \textbf{Pred}(V) \;\text{ and }\; J \;{}_P\!\vdash p \,\textbf{unless}\, q$$

  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:

$$\frac{P \div Q \;\text{ and }\; {}_{Q}\vdash\ \circlearrowleft J \;\text{ and }\; J, \mathbf{w}(P)\ {}_{P}\vdash\ p \leadsto q}{J, \mathbf{w}(P \| Q)\ {}_{P\|Q}\vdash\ p \leadsto q}$$

- More general:

$$\frac{J, V\ {}_{P}\vdash\ p \mapsto q \;\text{ and }\; J\ {}_{Q}\vdash\ V = \mathit{val}\ \textbf{unless}\ q}{J, V\ {}_{P\|Q}\vdash\ p \mapsto q}$$

Domain-level FSA Algorithm

$$
\begin{array}{lll}
d & : & \textbf{array } V \times Dom \times Dom \textbf{ of } Value \qquad \text{— data} \\
r & : & \textbf{array } V \times Dom \times Dom \textbf{ of } Value \qquad \text{— copies}
\end{array}
$$

$$
\textbf{prog } DFSA_{Dom,\mathcal{N}} \quad = \quad (\| \; A, a : A \in Dom \wedge a \in A : node_{a,A})
$$

where

$$
\begin{array}{lll}
\textbf{prog } node_{a,A} & = & (\| \; B \in Dom :: process_{a,A,B}) \\
 & & \| \\
 & & (\| \; A' \in \mathcal{N}(A) :: broadcast_{A,A'})
\end{array}
$$

prog $process_{a,b}$

...

$$
\textbf{assign} \quad d[a][A][B] := \Phi(A, B, r[a][-][B])
$$

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:

1. Theory: Prasetya and Swierstra, Formal Design of Self-stabilizing Programs, in Journal of Highspeed Network, special issue on self-stabilizing systems, 14, 2005.
2. Mechanical verification: Prasetya, Mechanically Supported Design of Self-stabilizing Algorithms, Phd thesis, 1995.