Synthesizing fault-tolerant distributed algorithms

Danny Dolev  
Hebrew University of Jerusalem

Keijo Heljanko

Joel Rybicki

Jukka Suomela

Siert Wieringa  
Aalto University & HIIT

Christoph Lenzen  
MPI Saarbrücken

Janne H. Korhonen  
Matti Järvisalo  
University of Helsinki & HIIT

Ulrich Schmid  
TU Wien

July 24, 2014  
FRIDA 2014, Vienna, Austria
What is this talk about?

Developing fault-tolerant distributed algorithms for consensus-like problems using computational techniques.
Verification vs synthesis

Verification:
“Check that given A satisfies the specification $S$.”

Synthesis:
“Construct an A that satisfies a specification $S$.”
The model problem

The *synchronous counting problem*:

- Closely related to consensus
- Self-stabilization
- Byzantine fault tolerance
- Hard to come up with correct algorithms
Our work

Prior work: Are there **efficient** and **compact** deterministic algorithms? Dolev et al. *(SSS 2013)*

Recent work: Developing and evaluating different **synthesis techniques**
Synchronous counting
The model

- $n$ processors
- $s$ states per node
- arbitrary initial state
The model

• $n$ processors
• $s$ states per node
• arbitrary initial state

Synchronous step:
1. send state to all neighbours
2. update state
The model

- $n$ processors
- $s$ states per node
- arbitrary initial state

Synchronous step:
1. send state to all neighbors
2. update state

algorithm = transition function
Self-stabilizing counting

1. Stabilization
2. Stabilization
3. Stabilization
4. Counting
Self-stabilizing counting

A simple algorithm solves the problem
Self-stabilizing counting

Solution: Follow the leader.
Self-stabilizing counting

Solution: Follow the leader.
Self-stabilizing counting

Solution: Follow the leader.
Self-stabilizing counting

Solution: Follow the leader.
Tolerating Byzantine failures

Assume that at most $f$ nodes may be Byzantine.
Tolerating Byzantine failures

Assume that at most $f$ nodes may be Byzantine.
Tolerating Byzantine failures

Assume that at most $f$ nodes may be Byzantine.
Tolerating Byzantine failures

Assume that at most $f$ nodes may be Byzantine.
Tolerating Byzantine failures can send different messages to non-faulty nodes!
Tolerating Byzantine failures can send different messages to non-faulty nodes!

Note: Easy if self-stabilization is not required!
Fault-tolerant counting

1. Stabilization
2. Stabilization
3. Stabilization
4. Counting
The model with failures

- $n$ processors
- $s$ states
- arbitrary initial state
- at most $f$ Byzantine nodes
Some basic facts

• How many states (per node) do we need?
  - \( s \geq 2 \)

• How many faults can we tolerate?
  - \( f < \frac{n}{3} \)

• How fast can we stabilize?
  - \( t > f \)

Pease et al., 1980
Fischer & Lynch, 1982
Solving synchronous counting

*Deterministic* solutions with large $s$ known for similar problems (e.g. D. Dolev & Hoch, 2007)

*Randomized* solutions for counting with small $s$ and large $t$ in expectation (e.g. S. Dolev: Self-stabilization)

We have synthesized *deterministic* algorithms with small $s$ and $t$ for the case $f = 1$ (*SSS '13*)
Finding an algorithm

The size of the search space is $s^b$ where $b = ns^n$.

<table>
<thead>
<tr>
<th>parameters</th>
<th>search space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 4$</td>
<td>$2^{64} \approx 10^{19}$</td>
</tr>
<tr>
<td>$s = 2$</td>
<td></td>
</tr>
</tbody>
</table>
Finding an algorithm

The size of the search space is $s^b$ where $b = ns^n$.

<table>
<thead>
<tr>
<th>parameters</th>
<th>search space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 4$</td>
<td>$2^{64} \approx 10^{19}$</td>
</tr>
<tr>
<td>$s = 2$</td>
<td></td>
</tr>
<tr>
<td>$n = 4$</td>
<td>$3^{324} \approx 10^{154}$</td>
</tr>
<tr>
<td>$s = 3$</td>
<td></td>
</tr>
</tbody>
</table>

We need a clever way to do the search!
Main results, $f = 1$

If $4 \leq n \leq 5$:

- **lower bound**: no 2-state algorithm
- **upper bound**: 3 states suffice

If $n \geq 6$:

- 2 states always suffice
Synthesis techniques
Our initial approach

• Fix $n$, $s$ and $f$

• The existence of an algorithm is a finite combinatorial decision problem

• Apply **SAT solvers** to a base case that implies a general solution
Generalizing from a base case

For any fixed $s, f$ and $t$:

- There is an algorithm $A$ for $n$ nodes

\[ \Downarrow \]

- There is an algorithm $B$ for $n+1$ nodes with same $s, f$ and $t$
Verification is easy

• Let $F$ be a set of faulty nodes, $|F| \leq f$

• Construct a *state graph* $G_F$ from $A$:
  
  **Nodes** = actual states
  
  **Edges** = possible state transitions
execution = walk
deadlock = loop
livelock = cycle
Verification is easy

- **A is correct** $\iff$ **Every $G_F$ is good**
- no deadlocks $\iff$ **$G_F$ is loopless**
- stabilization $\iff$ **All nodes have a path to 0**
- counting $\iff$ **\{0,1\} is the only cycle**
From verification to synthesis

The encoding uses the following variables:

\[ x_{i,u,s} \iff A_i(u) = s \]
\[ e_{q,r} \iff \text{edge } (q, r) \text{ exists} \]
\[ p_{q,r} \iff \text{path } q \rightsquigarrow r \text{ exists} \]

\[ x_{i,u,s} \xrightarrow{\text{}} e_{q,r} \xrightarrow{\text{}} p_{q,r} \]
The SAT approach

- Solver is a black box: no domain-knowledge
- Relatively easy to setup
- Size of instances blows up:
The SAT approach

- Solver is a black box: no domain-knowledge
- Relatively easy to setup
- Size of instances **blows up**:

<table>
<thead>
<tr>
<th>instance: n, s, t</th>
<th>variables</th>
<th>clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 3 10</td>
<td>6k</td>
<td>31k</td>
</tr>
<tr>
<td>5 3 10</td>
<td>45k</td>
<td>36k</td>
</tr>
<tr>
<td>6 3 10</td>
<td>403k</td>
<td>4M</td>
</tr>
</tbody>
</table>
Counter-example guided search

• A problem-specific synthesis algorithm
• CEGAR-inspired search
• Uses SAT solver to find *counter-examples*
• Learn constraints on-the-fly
A high-level overview

While algorithm candidates exist:

• Guess an algorithm $A$
• Use a SAT solver to check if $A$ is correct
• If not, solver gives a counter-example. Learn new constraints that forbid bad algorithms

How to learn *useful* constraints from counter-examples?
Some experiments
Experiment setup

- SAT encoding: MiniSAT and lingeling solvers
- ‘symsync’: the guided search algorithm
- same instance on 100 processors in parallel, different random seeds
s = 2, n = 7, f = 1, cyclic — positive results

fast algorithm

cnf-minisat

cnf-lingeling

sym-sync
some algorithm
orange = fast algorithm
blue = some algorithm
**SAT**: finds best solutions faster

**guided search**: finds some solution faster
orange = fast algorithm
blue = some algorithm

s = 3, n = 5, f = 1, cyclic — positive results

cnf-minisat

cnf-lingeling

symsync

10^{-1}  10^{0}  10^{1}  10^{2}  10^{3}  10^{4}

6  7  8  9  10  11  31
blue = some algorithm

orange = fast algorithm

SAT: finds best solutions faster

guided search: finds some solution faster

s = 3, n = 5, f = 1, cyclic _ positive results
Summary

• Synthesis a tool for *theory of distributed computing*
• Results: optimal fault-tolerant algorithms
• Complementary approaches for fast synthesis
Summary

- Synthesis a tool for *theory of distributed computing*
- Results: optimal fault-tolerant algorithms
- Complementary approaches for fast synthesis

Synthesis in our other work:
- local graph coloring
- finding large cuts  
  \texttt{arXiv:1402.2543}
Summary

• Synthesis a tool for *theory of distributed computing*
• Results: optimal fault-tolerant algorithms
• Complementary approaches for fast synthesis

Synthesis in our other work:

• local graph coloring
• finding large cuts [arXiv:1402.2543]

Thanks!