Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction

Annu John  Igor Konnov  Ulrich Schmid  Helmut Veith  Josef Widder

TU W I E N

for(syte)

RiSE
Rigorous Systems Engineering

FMCAD’13
Portland, OR, USA, Oct 20-23, 2013
Why fault-tolerant (FT) distributed algorithms

faults not in the control of system designer

- bit-flips in memory
- power outage
- disconnection from the network
- intruders take control over some computers
Why fault-tolerant (FT) distributed algorithms

faults not in the control of system designer

- bit-flips in memory
- power outage
- disconnection from the network
- intruders take control over some computers

distributed algorithms intended to make systems more reliable even in the presence of faults

- replicate processes
- exchange messages
- do coordinated computation
- goal: keep replicated processes in “good state”
Fault-tolerant distributed algorithms

- $n$ processes communicate by messages
Fault-tolerant distributed algorithms

- $n$ processes communicate by messages
- all processes know that at most $t$ of them might be faulty

$n > 3t \land t \geq f \geq 0$

no masquerading: the processes know the origin of incoming messages

Igor Konnov (www.forsyte.at)

Parameterized Model Checking of FTDAs...
Fault-tolerant distributed algorithms

- $n$ processes communicate by messages
- all processes know that at most $t$ of them might be faulty
- $f$ are actually faulty
- resilience conditions, e.g., $n > 3t \land t \geq f \geq 0$
- no masquerading: the processes know the origin of incoming messages
Fault models from benign to Byzantine

- **clean crashes:**
  faulty processes prematurely halt after/before “send to all”

- **crash faults:**
  faulty processes prematurely halt (also) in the middle of “send to all”

- **omission faults:**
  faulty processes follow the algorithm, but some messages sent by them might be lost

- **symmetric faults:**
  faulty processes send arbitrarily to all or nobody

- **Byzantine faults:**
  faulty processes can do anything

- **hybrid models:**
  combinations of the above
Automated Verification?
Fault-tolerant DAs: Model Checking Challenges

- **unbounded data types**
  - counting how many messages have been received

- **parameterization in multiple parameters**
  - among $n$ processes $f \leq t$ are faulty with $n > 3t$

- **contrast to concurrent programs**
  - fault tolerance against adverse environments

- **degrees of concurrency**
  - many degrees of partial synchrony

- **continuous time**
  - fault-tolerant clock synchronization
Importance of liveness in distributed algorithms

Interplay of safety and liveness is a central challenge in DAs

- interplay of safety and liveness is non-trivial
- asynchrony and faults lead to impossibility results
Importance of liveness in distributed algorithms

Interplay of safety and liveness is a central challenge in DAs

- interplay of safety and liveness is non-trivial
- asynchrony and faults lead to impossibility results

Rich literature to verify safety (e.g. in concurrent systems)

Distributed algorithms perspective:

- “doing nothing is always safe”
- “tools verify algorithms that actually might do nothing”
Model checking problem for fault-tolerant DA algorithms

Parameterized model checking problem:
- given a distributed algorithm and spec. $\varphi$
- show for all $n$, $t$, and $f$ satisfying $n > 3t \land t \geq f \geq 0$
  $M(n, t, f) \models \varphi$
- every $M(n, t, f)$ is a system of $n - f$ correct processes
Model checking problem for fault-tolerant DA algorithms

Parameterized model checking problem:

- given a distributed algorithm and spec. \( \varphi \)
- show for all \( n, t, \) and \( f \) satisfying resilience condition
  \[ M(n, t, f) \models \varphi \]
- every \( M(n, t, f) \) is a system of \( N(n, f) \) correct processes
Properties in Linear Temporal Logic

**Unforgeability (U).** If \( v_i = 0 \) for all correct processes \( i \), then for all correct processes \( j \), \( \text{accept}_j \) remains 0 forever.

\[
\mathbf{G} \left( \left( \bigwedge_{i=1}^{n-f} v_i = 0 \right) \rightarrow \mathbf{G} \left( \bigwedge_{j=1}^{n-f} \text{accept}_j = 0 \right) \right)
\]

**Completeness (C).** If \( v_i = 1 \) for all correct processes \( i \), then there is a correct process \( j \) that eventually sets \( \text{accept}_j \) to 1.

\[
\mathbf{G} \left( \left( \bigwedge_{i=1}^{n-f} v_i = 1 \right) \rightarrow \mathbf{F} \left( \bigvee_{j=1}^{n-f} \text{accept}_j = 1 \right) \right)
\]

**Relay (R).** If a correct process \( i \) sets \( \text{accept}_i \) to 1, then eventually all correct processes \( j \) set \( \text{accept}_j \) to 1.

\[
\mathbf{G} \left( \left( \bigvee_{i=1}^{n-f} \text{accept}_i = 1 \right) \rightarrow \mathbf{F} \left( \bigwedge_{j=1}^{n-f} \text{accept}_j = 1 \right) \right)
\]
Properties in Linear Temporal Logic

Unforgeability (U). If $v_i = 0$ for all correct processes $i$, then for all correct processes $j$, accept$_j$ remains 0 forever.

$$\mathbf{G} \left( \left( \bigwedge_{i=1}^{n-f} v_i = 0 \right) \rightarrow \mathbf{G} \left( \bigwedge_{j=1}^{n-f} \text{accept}_j = 0 \right) \right)$$

Safety

Completeness (C). If $v_i = 1$ for all correct processes $i$, then there is a correct process $j$ that eventually sets accept$_j$ to 1.

$$\mathbf{G} \left( \left( \bigwedge_{i=1}^{n-f} v_i = 1 \right) \rightarrow \mathbf{F} \left( \bigvee_{j=1}^{n-f} \text{accept}_j = 1 \right) \right)$$

Liveness

Relay (R). If a correct process $i$ sets accept$_i$ to 1, then eventually all correct processes $j$ set accept$_j$ to 1.

$$\mathbf{G} \left( \left( \bigvee_{i=1}^{n-f} \text{accept}_i = 1 \right) \rightarrow \mathbf{F} \left( \bigwedge_{j=1}^{n-f} \text{accept}_j = 1 \right) \right)$$

Liveness
Threshold-guarded fault-tolerant distributed algorithms
Threshold-guarded FTDAs

Fault-free construct: quantified guards ($t=f=0$)

- **Existential Guard**
  
  if received $m$ from *some* process then ...

- **Universal Guard**
  
  if received $m$ from *all* processes then ...

These guards allow one to treat the processes in a parameterized way
Threshold-guarded FTDAs

Fault-free construct: quantified guards \((t=f=0)\)

- **Existential Guard**
  
  if received \(m\) from *some* process then ... 

- **Universal Guard**
  
  if received \(m\) from *all* processes then ... 

These guards allow one to treat the processes in a parameterized way

*what if faults might occur?*
Threshold-guarded FTDAs

Fault-free construct: quantified guards \((t=f=0)\)

- **Existential Guard**
  
  if received \(m\) from *some* process then ...

- **Universal Guard**
  
  if received \(m\) from *all* processes then ...

These guards allow one to treat the processes in a parameterized way

*what if faults might occur?*

Fault-Tolerant Algorithms: \(n\) processes, at most \(t\) are Byzantine

- **Threshold Guard**
  
  if received \(m\) from \(n-t\) processes then ...

- (the processes cannot refer to \(f\)!)
Counting argument in threshold-guarded algorithms

if received $m$ from $t+1$ processes then ...

Correct processes count distinct incoming messages
Counting argument in threshold-guarded algorithms

Correct processes count distinct incoming messages

if received $m$ from $t+1$ processes then ...
Counting argument in threshold-guarded algorithms

Correct processes count distinct incoming messages

if received $m$ from $t+1$ processes then ...
our abstraction
at a glance
Data + counter abstraction over parametric intervals

\[ n = 6, \ t = 1, \ f = 1 \]

\[ t + 1 = 2, \ n - t = 5 \]

nr. processes (counters)

1 process at (accepted, received=5)

3 processes at (sent, received=3)
Data + counter abstraction over parametric intervals

\[ n = 6, \ t = 1, \ f = 1 \]
\[ t + 1 = 2, \ n - t = 5 \]

nr. processes (counters)

![Diagram showing nr. processes (counters)](image)
Data + counter abstraction over parametric intervals

\[ n = 6, \ t = 1, \ f = 1 \]

\[ t + 1 = 2, \ n - t = 5 \]
Data + counter abstraction over parametric intervals

\[ n = 6, \ t = 1, \ f = 1 \]

\[ n > 3 \cdot t \land t \geq f \]

nr. processes (counters)

Parametric intervals:

\[ I_0 = [0, 1) \quad I_1 = [1, t + 1) \]

\[ I_{t+1} = [t + 1, n - t) \]

\[ I_{n-t} = [n - t, \infty) \]
Data + counter abstraction over parametric intervals

\[ n > 3 \cdot t \land t \geq f \]

nr. processes (counters)

Parametric intervals:

\[ I_0 = [0, 1) \quad I_1 = [1, t + 1) \]
\[ I_{t+1} = [t + 1, n - t) \]
\[ I_{n-t} = [n - t, \infty) \]

all correct processes accepted?
Related work: \((0, 1, \infty)\)-counter abstraction

Pnueli, Xu, and Zuck (2001) introduced \((0, 1, \infty)\)-counter abstraction:

- finitely many local states,
  
  e.g., \(\{N, T, C\}\).

- abstract the number of processes in every state,
  
  e.g., \(K : C \mapsto 0, \ T \mapsto 1, \ N \mapsto \text{“many”}\).

- perfectly reflects mutual exclusion properties
  
  e.g., \(G \left(K(C) = 0 \lor K(C) = 1\right)\).
Related work: \((0, 1, \infty)\)-counter abstraction

Pnueli, Xu, and Zuck (2001) introduced \((0, 1, \infty)\)-counter abstraction:

- finitely many local states,
  - e.g., \(\{N, T, C\}\).
- abstract the number of processes in every state,
  - e.g., \(K : C \mapsto 0, \quad T \mapsto 1, \quad N \mapsto \text{“many”}\).
- perfectly reflects mutual exclusion properties
  - e.g., \(G(K(C) = 0 \lor K(C) = 1)\).

Our parametric data + counter abstraction:

- unboundedly many local states (nr. of received messages)
- finer counting of processes:
  - \(t + 1\) processes in a specific state can force global progress,
    while \(t\) processes cannot
- mapping \(t, t + 1, \) and \(n - t\) to “many” is too coarse.
Technical details
Technical challenges

How to do data abstraction?

How to do counter abstraction?

How to refine spurious counter-examples introduced by the abstraction?
Abstract operations

Concrete: $t + 1 \leq x$

Concrete $t + 1 \leq x$

Abstract: $I_0 \quad I_1 \quad I_{t+1} \quad I_{n-t}$
Abstract operations

Concrete: 0 1 \( t + 1 \) \( n - t \) above

Abstract: \( I_0 \) \( I_1 \) \( I_{t+1} \) \( I_{n-t} \)

Concrete \( t + 1 \leq x \) is abstracted as \( x = I_{t+1} \lor x = I_{n-t} \).
Abstract operations

Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \lor x = I_{n-t}$.

Concrete $x' = x + 1$, 

Concrete: $0, 1, t + 1, n - t$ above

Abstract: $I_0, I_1, I_{t+1}, I_{n-t}$
Abstract operations

Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \lor x = I_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

$$x = I_0 \land x' = I_1 \ldots$$
Abstract operations

Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \lor x = I_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

$x = I_0 \land x' = I_1$

$\lor x = I_1 \land (x' = I_1 \lor x' = I_{t+1})$
Abstract operations

Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \lor x = I_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

\[
\begin{align*}
 x &= I_0 \land x' = I_1 \\
 \lor x &= I_1 \land (x' = I_1 \lor x' = I_{t+1}) \\
 \lor x &= I_{t+1} \land (x' = I_{t+1} \lor x' = I_{n-t}) \ldots
\end{align*}
\]
Abstract operations

Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \lor x = I_{n-t}$.

Concrete $x' = x + 1$, is abstracted as:

$$x = I_0 \land x' = I_1$$
$$\forall x = I_1 \land (x' = I_1 \lor x' = I_{t+1})$$
$$\forall x = I_{t+1} \land (x' = I_{t+1} \lor x' = I_{n-t})$$
$$\forall x = I_{n-t} \land x' = I_{n-t}$$
Parametric abst. refinement — uniformly spurious paths

Classical CEGAR:
Parametric abst. refinement — uniformly spurious paths

Classical CEGAR:

Concrete

Abstract

Concrete

n_1, t_1, f_1

n_2, t_2, f_2

Our case:
the implementation
Tool Chain: **ByMC**

- **Parametric Promela code**
- **Static analysis + Yices**
- **Parametric Interval Domain $\hat{D}$**
- **Parametric data abstraction with Yices**
- **Parametric counter abstraction with Yices**
- **Normal Promela code**
- **Spin**
- **Counterexample**
- **Property holds**
Tool Chain: 

**ByMC**

- **Parametric Promela code**
  - STATIC ANALYSIS + YICES
  - Parametric Interval Domain $\hat{D}$

- **Parametric data abstraction with YICES**

- **Parametric Promela code**

- **Parametric counter abstraction with YICES**

- **Concrete counter representation (VASS)**
  - SMT formula

- **Refine**
  - unsat
  - YICES
  - sat

- **Spin**
  - normal Promela code
  - property holds
  - counterexample
  - counterexample feasible
Tool Chain: **ByMC**

1. **Parametric Promela code**
2. **STATIC ANALYSIS + YICES**
   - Parametric Interval Domain $\hat{D}$
3. **PARAMETRIC DATA ABSTRACTION WITH YICES**
4. **PARAMETRIC COUNTER ABSTRACTION WITH YICES**
5. **CONCRETE COUNTER REPRESENTATION (VASS)**
   - SMT formula
   - invariant candidates (by the user)
     - unsat
     - sat
     - YICES
6. **REFINE**
   - property holds
   - counterexample
   - counterexample feasible
7. **SPIN**
   - normal Promela code
Concrete vs. parameterized (Byzantine case)

Parameterized model checking performs well (the red line).

Experiments for fixed parameters quickly degrade ($n = 9$ runs out of memory).

We found counter-examples for the cases $n = 3t$ and $f > t$, where the resilience condition is violated.
## Experimental results at a glance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Fault</th>
<th>Resilience</th>
<th>Property</th>
<th>Valid?</th>
<th>#Refinements</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST87</td>
<td>BYZ</td>
<td>$n &gt; 3t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>4 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>BYZ</td>
<td>$n &gt; 3t$</td>
<td>C</td>
<td>✓</td>
<td>10</td>
<td>32 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>BYZ</td>
<td>$n &gt; 3t$</td>
<td>R</td>
<td>✓</td>
<td>10</td>
<td>24 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>SYMM</td>
<td>$n &gt; 2t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>1 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>SYMM</td>
<td>$n &gt; 2t$</td>
<td>C</td>
<td>✓</td>
<td>2</td>
<td>3 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>SYMM</td>
<td>$n &gt; 2t$</td>
<td>R</td>
<td>✓</td>
<td>12</td>
<td>16 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>OMIT</td>
<td>$n &gt; 2t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>1 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>OMIT</td>
<td>$n &gt; 2t$</td>
<td>C</td>
<td>✓</td>
<td>5</td>
<td>6 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>OMIT</td>
<td>$n &gt; 2t$</td>
<td>R</td>
<td>✓</td>
<td>5</td>
<td>10 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>2 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>C</td>
<td>✓</td>
<td>4</td>
<td>8 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>R</td>
<td>✓</td>
<td>13</td>
<td>31 sec.</td>
</tr>
<tr>
<td>CT96</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>1 sec.</td>
</tr>
<tr>
<td>CT96</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>A</td>
<td>✓</td>
<td>0</td>
<td>1 sec.</td>
</tr>
<tr>
<td>CT96</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>R</td>
<td>✓</td>
<td>0</td>
<td>1 sec.</td>
</tr>
<tr>
<td>CT96</td>
<td>CLEAN</td>
<td>$n &gt; t$</td>
<td>C</td>
<td>✗</td>
<td>0</td>
<td>1 sec.</td>
</tr>
</tbody>
</table>
When resilience condition is wrong...

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Fault</th>
<th>Resilience</th>
<th>Property</th>
<th>Valid?</th>
<th>#Refinements</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST87</td>
<td>Byz</td>
<td>$n &gt; 3t \land f \leq t+1$</td>
<td>U</td>
<td>X</td>
<td>9</td>
<td>56 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Byz</td>
<td>$n &gt; 3t \land f \leq t+1$</td>
<td>C</td>
<td>X</td>
<td>11</td>
<td>52 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Byz</td>
<td>$n &gt; 3t \land f \leq t+1$</td>
<td>R</td>
<td>X</td>
<td>10</td>
<td>17 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Byz</td>
<td>$n \geq 3t \land f \leq t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>5 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Byz</td>
<td>$n \geq 3t \land f \leq t$</td>
<td>C</td>
<td>✓</td>
<td>9</td>
<td>32 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Byz</td>
<td>$n \geq 3t \land f \leq t$</td>
<td>R</td>
<td>X</td>
<td>30</td>
<td>78 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Symm</td>
<td>$n &gt; 2t \land f \leq t+1$</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>2 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Symm</td>
<td>$n &gt; 2t \land f \leq t+1$</td>
<td>C</td>
<td>X</td>
<td>2</td>
<td>4 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Symm</td>
<td>$n &gt; 2t \land f \leq t+1$</td>
<td>R</td>
<td>✓</td>
<td>8</td>
<td>12 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Omit</td>
<td>$n &gt; 2t \land f \leq t$</td>
<td>U</td>
<td>✓</td>
<td>0</td>
<td>1 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Omit</td>
<td>$n &gt; 2t \land f \leq t$</td>
<td>C</td>
<td>X</td>
<td>0</td>
<td>2 sec.</td>
</tr>
<tr>
<td>ST87</td>
<td>Omit</td>
<td>$n &gt; 2t \land f \leq t$</td>
<td>R</td>
<td>X</td>
<td>0</td>
<td>2 sec.</td>
</tr>
</tbody>
</table>
Experimental setup

The tool (source code in OCaml), the code of the distributed algorithms in Parametric Promela, and a virtual machine with full setup are available at: http://forsyte.at/software/bymc
Summary of results

- Abstraction tailored for distributed algorithms
  - threshold-based
  - fault-tolerant
  - allows to express different fault assumptions

- Verification of threshold-based fault-tolerant algorithms
  - with threshold guards that are widely used
  - Byzantine faults (and other)
  - for all system sizes
Summary of results

- Abstraction tailored for distributed algorithms
  - threshold-based
  - fault-tolerant
  - allows to express different fault assumptions

- Verification of threshold-based fault-tolerant algorithms
  - with threshold guards that are widely used
  - Byzantine faults (and other)
  - for all system sizes
Related work: non-parameterized

Model checking of the small size instances:

- clock synchronization [Steiner, Rushby, Sorea, Pfeifer 2004]
- consensus [Tsuchiya, Schiper 2011]
- asynchronous agreement, folklore broadcast, condition-based consensus [John, Konnov, Schmid, Veith, Widder 2013]
- and more...
Related work: parameterized case

Regular model checking of fault-tolerant distributed protocols:

[Fisman, Kupferman, Lustig 2008]

- “First-shot” theoretical framework.
- No guards like $x \geq t + 1$, only $x \geq 1$.
- No implementation.
- Manual analysis applied to folklore broadcast (crash faults).
Related work: parameterized case

Regular model checking of fault-tolerant distributed protocols:

[Fisman, Kupferman, Lustig 2008]

- “First-shot” theoretical framework.
- No guards like $x \geq t + 1$, only $x \geq 1$.
- No implementation.
- Manual analysis applied to folklore broadcast (crash faults).

Backward reachability using SMT with arrays:

[Alberti, Ghilardi, Pagani, Ranise, Rossi 2010-2012]

- Implementation.
- Experiments on Chandra-Toueg 1990.
- No resilience conditions like $n > 3t$.
- Safety only.
## Our current work

<table>
<thead>
<tr>
<th>Discrete synchronous</th>
<th>Discrete partially synchronous</th>
<th>Discrete asynchronous</th>
<th>Continuous synchronous</th>
<th>Continuous partially synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>One instance/ finite payload</td>
<td>Many inst./ finite payload</td>
<td>Many inst./ unbounded payload</td>
<td>Messages with reals</td>
<td>—</td>
</tr>
</tbody>
</table>

**one-shot broadcast, c.b.consensus**

- core of \{ST87, BT87, CT96\}, MA06 (common), MR04 (binary)
Future work: threshold guards + orthogonal features

<table>
<thead>
<tr>
<th></th>
<th>Discrete synchronous</th>
<th>Discrete partially synchronous</th>
<th>Discrete asynchronous</th>
<th>Continuous synchronous</th>
<th>Continuous partially synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>One instance/finite payload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many inst./finite payload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Many inst./unbounded payload</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Messages with reals</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**One-shot broadcast, c.b.consensus**
- Core of \{ST87, BT87, CT96\}, MA06 (common), MR04 (binary)

**Clock sync**
- DLPSW86
- FLS13
- FSFK06
- WS07
- WS09

**Broadcast**
- ST87, BT87, CT96, with failure-detectors

**Approx. agreement**
- AK00
- DLS86, MA06, L98 (Paxos)

- DHM12

**Igor Konnov (www.forsyte.at)**
Thank you!

[http://forsyte.at/software/bymc]
In the Byzantine case we have \( \text{in\_transit} : \forall i. (\text{recv}_i \geq \text{sent}) \) and \( \text{GF} \neg \text{in\_transit} \).

In this case communication fairness implies computation fairness.

But in the abstract version \( \text{sent} \) can deviate from the number of processes who sent the echo message.

In this case the user formulates a simple state invariant candidate, e.g., \( \text{sent} = K([\text{sv} = \text{SE} \lor \text{sv} = \text{AC}]) \) (on the level of the original concrete system).

The tool checks automatically, whether the candidate is actually a state invariant.

After the abstraction the abstract version of the invariant restricts the behavior of the abstract transition system.
Parametric abstraction refinement — justice suppression

justice $\mathbf{GF} \neg in\_transit$ necessary to verify liveness
Parametric abstraction refinement — justice suppression

justice $G F \neg in\_transit$ necessary to verify liveness

counter example:

if $\forall j$ all concretizations of $s_j$ violate $\neg in\_transit$, then CE is spurious.
justice $\mathbf{GF} \neg \text{in\_transit}$ necessary to verify liveness

counter example:

\[
\text{if } \forall j \text{ all concretizations of } s_j \text{ violate } \neg \text{in\_transit}, \text{ then CE is spurious.}
\]

refine justice to $\mathbf{GF} \neg \text{in\_transit} \land \mathbf{GF} \left( \bigvee_{1 \leq j \leq k} \neg \text{at}(s_j) \right)$
justice $\mathbf{G} \mathbf{F} \neg in\_transit$ necessary to verify liveness

counter example:

\[\ell_i \rightarrow s_1 \rightarrow \cdots \rightarrow s_k \rightarrow \cdots \rightarrow s_j \rightarrow \cdots \rightarrow s_n \rightarrow \ell_i\]

\[\text{if } \forall j \text{ all concretizations of } s_j \text{ violate } \neg in\_transit, \text{ then CE is spurious.}\]

refine justice to $\mathbf{G} \mathbf{F} \neg in\_transit \land \mathbf{G} \mathbf{F} \left( \bigvee_{1 \leq j \leq k} \neg at(s_j) \right)$

... we use unsat cores to refine several loops at once
Parametric abstraction refinement — justice suppression

justice $\mathcal{G} \mathcal{F} \neg in\_transit$ necessary to verify liveness
Parametric abstraction refinement — justice suppression

justice $\textbf{GF} \neg \text{in\_transit}$ necessary to verify liveness

counter example:

if $\forall j$ all concretizations of $s_j$ violate $\neg \text{in\_transit}$, then CE is spurious.

\[
\begin{align*}
\text{in\_transit} & \quad \text{in\_transit} \\
\text{in\_transit} & \quad \text{in\_transit} \\
\text{in\_transit} & \quad \text{in\_transit}
\end{align*}
\]
Parametric abstraction refinement — justice suppression

justice $\mathbf{GF \neg in\_transit}$ necessary to verify liveness

counter example:

if $\forall j$ all concretizations of $s_j$ violate $\neg in\_transit$, then CE is spurious.

refine justice to $\mathbf{GF \neg in\_transit} \land \mathbf{GF} \left( \bigvee_{1 \leq j \leq k} \neg at(s_j) \right)$
Parametric abstraction refinement — justice suppression

justice $G F \neg \text{in}_\text{transit}$ necessary to verify liveness
counter example:

if $\forall j$ all concretizations of $s_j$ violate $\neg \text{in}_\text{transit}$, then CE is spurious.

refine justice to $G F \neg \text{in}_\text{transit} \land G F \left( \bigvee_{1 \leq j \leq k} \neg at(s_j) \right)$

... we use unsat cores to refine several loops at once
asynchronous reliable broadcast (srikanth & toueg 1987)

the core of the classic broadcast algorithm from the da literature. it solves an agreement problem depending on the inputs $v_i$.

Variables of process $i$

$v_i$: \{0, 1\} init with 0 or 1

$accept_i$: \{0, 1\} init with 0

An indivisible step:

if $v_i = 1$
then send (echo) to all;

if received (echo) from at least $t + 1$ distinct processes and not sent (echo) before
then send (echo) to all;

if received (echo) from at least $n - t$ distinct processes
then $accept_i := 1;$

byzantine faults correct if $n > 3t$
resilience condition rc parameterized process skeleton $p(n, t)$
asynchronous reliable broadcast (Srikanth & Toueg 1987)

The core of the classic broadcast algorithm from the DA literature. It solves an agreement problem depending on the inputs $v_i$.

Variables of process $i$

$v_i$: \{0, 1\} init with 0 or 1

$accept_i$: \{0, 1\} init with 0

An indivisible step:

if $v_i = 1$
then send (echo) to all;

if received (echo) from at least $t + 1$ distinct processes and not sent (echo) before
then send (echo) to all;

if received (echo) from at least $n - t$ distinct processes
then $accept_i := 1$;

dependent $t$ byzantine faults

correct if $n > 3t$

resilience condition $rc$

parameterized process skeleton $p(n, t)$
Abstract CFA

recv := z where (recv ≤ z ∧ z ≤ sent + f)

- (sv = V1)

sv := SE

 recv := z where (recv ≤ z ∧ z ≤ sent + f)

inc sent

- (sv = V0)

 t + 1 ≤ recv

sv := V0

- (sv = V0)

inc sent

- (t + 1 ≤ recv)

sv := V0

- (sv = V0)

inc sent

- (t + 1 ≤ recv)

sv := SE

sv := AC

¬ (sv = V1)

inc sent

¬ (sv = V1)

inc sent

¬ (sv = V1)

inc sent

¬ (sv = V1)

inc sent

¬ (sv = V1)

inc sent

¬ (sv = V1)

inc sent
Abstract CFA

\[ recv := z \text{ where } (recv \leq z \land z \leq sent + f) \]

\[ [recv = I_0 \land sent = I_0 \land (recv' = I_0 \lor recv' = I_1)] \lor \ldots \]

\[ sv := V_1 \]

\[ \neg(sv = V_1) \]

\[ q_1 \]

\[ q_0 \]

\[ q_3 \]

\[ q_5 \]

\[ q_4 \]

\[ q_6 \]

\[ q_7 \]

\[ q_8 \]

\[ sv = V_0 \]

\[ t + 1 \leq recv \]

\[ inc sent \]

\[ \neg(sv = V_0) \]

\[ \neg(t + 1 \leq recv) \]

\[ \neg(n - t \leq recv) \]

\[ sv := SE \]

\[ inc sent \]

\[ sv := SE \]

\[ sv := SE \]

\[ sv := AC \]

\[ sv := AC \]

Igor Konnov (www.forsyte.at) Parameterized Model Checking of FTDAs... FMCAD’13 35 / 30