Model Checking of Fault-Tolerant Distributed Algorithms

Part II: Modeling Fault-tolerant Distributed Algorithms

Annu Gmeiner Igor Konnov Ulrich Schmid
Helmut Veith Josef Widder

Rigorous Systems Engineering

Uni Salzburg, June 2015

Why Model Checking?

- an alternative proof approach
- useful counter-examples
- ability to define and vary assumptions about the system
- and see why it breaks
- closer to code level
- good degree of automation

Distributed Algorithms: Model Checking Challenges

- unbounded data types

  unbounded number of rounds (round numbers part of messages)

- parameterization in multiple parameters

  among n processes $f \leq t$ are faulty with $n > 3t$

- contrast to concurrent programs

  diverse fault models (adverse environments)

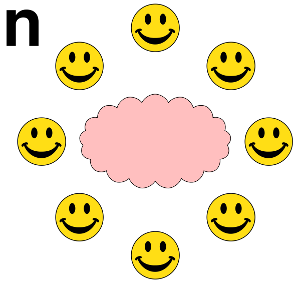
- continuous time

  fault-tolerant clock synchronization

- degrees of concurrency: synchronous, asynchronous partially synchronous

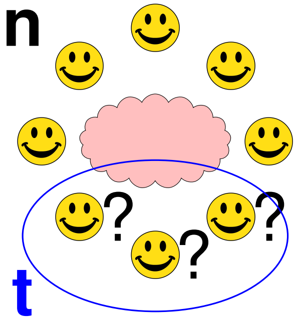
  a process makes at most 5 steps between 2 steps of any other process

Fault-tolerant distributed algorithms



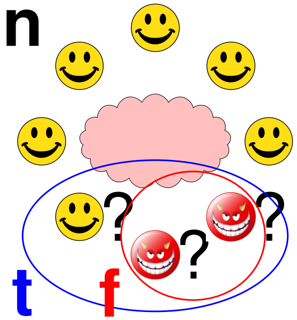
- *n* processes communicate by messages
- all processes know that at most *t* of them might be faulty
- *f* are actually faulty

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

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

Challenge #1: fault models

- clean crashes: least severe
  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: most severe
  faulty processes can do anything
  encompass all behaviors of above models

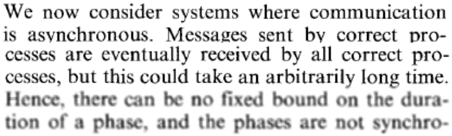
Challenges #2 & #3: Pseudo-code and Communication

Translate pseudo-code to a formal description

- that allows us to verify the algorithm
- and does not oversimplify the original algorithm.

Assumptions about the communication medium

- are usually written in plain English,
- spread across research papers,
- constitute folklore knowledge.



We now consider systems where communication is asynchronous. Messages sent by correct processes are eventually received by all correct processes, but this could take an arbitrarily long time. Hence, there can be no fixed bound on the duration of a phase, and the phases are not synchro-

Asynchronous Reliable Broadcast (Srikanth & Toueg, 87)

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} initially 0 or 1
  $accept_i$: {0, 1} initially 0

An atomic 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;
```

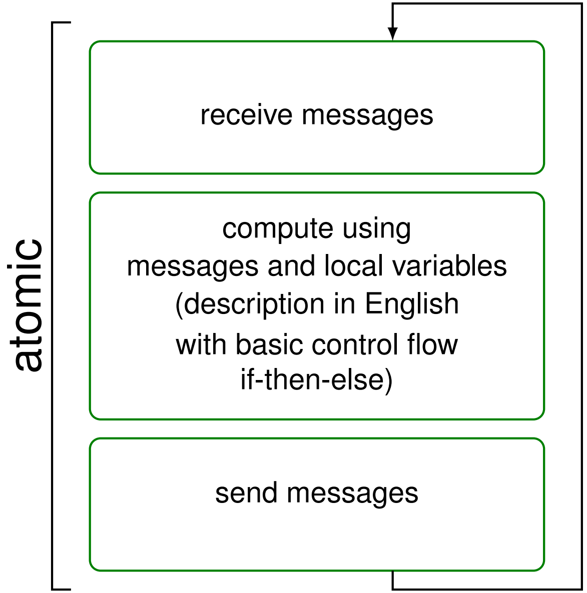
Asynchronous Reliable Broadcast (Srikanth & Toueg, 87)

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

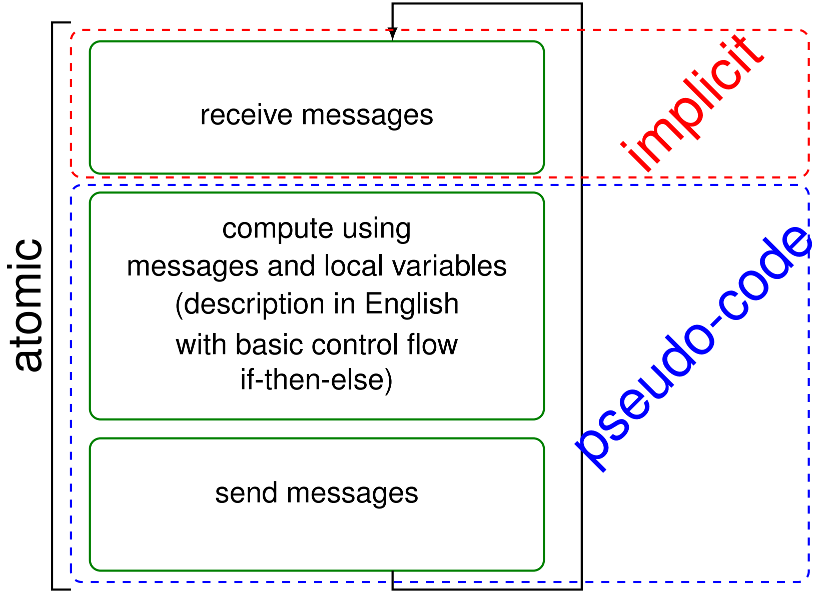
```
Variables of process i
  v_i: {0, 1} initially 0 or 1
  accept_i: {0, 1} initially 0

An atomic 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;
```

- asynchronous
- t Byzantine faults
- correct if $n > 3t$
- the code is parameterized in n and t
  $\Rightarrow$ process template $P(n, t, f)$

Typical Structure of a Computation Step



Typical Structure of a Computation Step



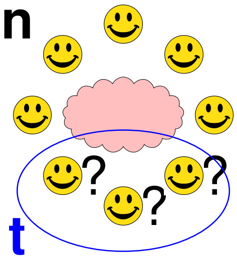
Challenge #4: Parameterized Model Checking

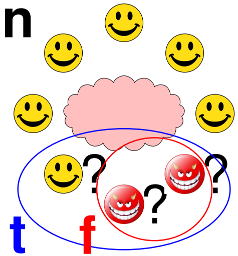
Parameterized model checking problem:

- given a process template $P(n,t,f)$,
- resilience condition $RC : n > 3t \wedge t \geq f \geq 0$,
- fairness constraints Φ, e.g., "all messages will be delivered"
- and an LTL$_{\text{-X}}$ formula φ

- show for all n, t, and f satisfying RC

$$(P(n,t,f))^{n-f} + f \text{ faults} \models (\Phi \rightarrow \varphi)$$





Challenge #5: Liveness in Distributed Algorithms

Interplay of safety and liveness is a central challenge in DAs

- achieving safety and liveness is non-trivial
- asynchrony and faults lead to impossibility results
  (recall first part of lecture (Fischer *et al.*, 1985))

Challenge #5: Liveness in Distributed Algorithms

Interplay of safety and liveness is a central challenge in DAs

- achieving safety and liveness is non-trivial
- asynchrony and faults lead to impossibility results
  (recall first part of lecture (Fischer *et al.*, 1985))

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”

Verification efforts often have to simplify assumptions

Summary

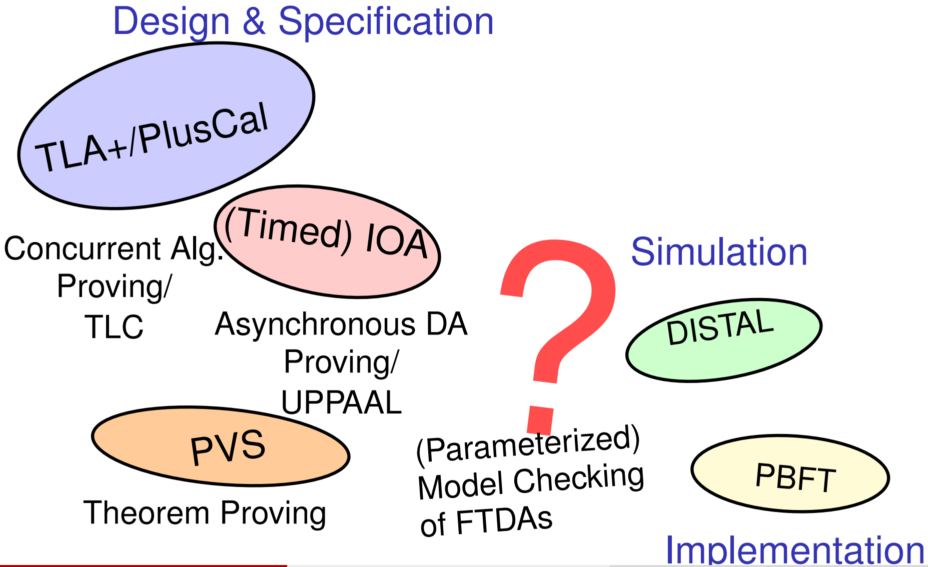
We have to model:

- faults,
- communication medium captured in English,
- algorithms written in pseudo-code.

and check:

- safety and liveness
- of parameterized systems
- with unbounded integers,
- non-standard fairness constraints,

Existing formalization frameworks


Design & Specification
TLA+/PlusCal
(Timed) IOA
Concurrent Alg. Proving/ TLC
Asynchronous DA Proving/ UPPAAL
?
Simulation
DISTAL
PVS
Theorem Proving
(Parameterized) Model Checking of FTDAs
PBFT
Implementation

Alternative frameworks

TLA (temporal logic of actions):

- used to design (distributed) algorithms by refinement of the spec
- verification with proof assistants (low degree of automation)

Encodings of DA in proof assistant PVS (e.g., by Rushby):

- ad-hoc encoding
- found a bug in a published synchronous Byzantine Agreement algorithm (Lincoln & Rushby, 1993)

I/O-Automata:

- originally designed to write clearer hand-written proofs
- limited tool support, e.g., Veromodo toolset is still in beta
- suitable only for asynchronous distributed algorithms

Alternative frameworks

TLA (temporal logic of actions):

- used to design (distributed) algorithms by refinement of the spec
- verification with proof assistants (low degree of automation)

Encodings of DA in proof assistant PVS (e.g., by Rushby):

- ad-hoc encoding
- found a bug in a published synchronous Byzantine Agreement algorithm (Lincoln & Rushby, 1993)

I/O-Automata:

- originally designed to write clearer hand-written proofs
- limited tool support, e.g., Veromodo toolset is still in beta
- suitable only for asynchronous distributed algorithms

proof assistants are very general, but with low automation degree
"everything is possible, but nothing is easy"

Simulation and Implementation

Distal:

- Domain-specific language (Biely *et al.*, 2013)
- Simulation and evaluate performance of fault-tolerant algorithms

Practical Byzantine Fault-Tolerance (Castro *et al.*, 1999)
and other practical algorithms:

- Implementation with optimizations
- Precise semantics is unclear
- The system is partially synchronous:
  non-divergent message delays are assumed

In this part

We introduce efficient encoding in PROMELA.

Verify safety and liveness of fault-tolerant algorithms (fixed parameters).

Find counterexamples for parameters known from the literature.

This proves adequacy of our modeling.

Preliminaries: Promela

Promela

PROMELA ≡ PROcess MEta LAnguage

SPIN ≡ Simple Promela INterpreter
(not that simple any more)

Here we give a short introduction and cover only the features important to our work.

Detailed documentation, tutorials, and books on:
http://spinroot.com

Gerard Holzmann

Top-level: global variables and processes

```
/* global declarations visible to all processes */
  int x; /* a global integer (as in C) */

  mtype = { X, Y }; /* constant message types */
/* a FIFO channel with at most 2 messages of type mtype */
  chan c = [2] of { mtype };

  active[2] proctype ProcA() {
    ...
  }

  proctype ProcB() {
    ...
  }

  init {
    run ProcB(); run ProcB();
  }
```

Two processes are created at the initial state

Processes can be created later using: run ProcB()

A special process, use to create other processes

One process: Basics

```
int x, y;

active proctype ProcA() {

  int z;

  z = x;

  x > y;

  true;

  z++;

  skip;
}
```

Declare a local variable

Assignment

Block until the expression is evaluated to true

one step to execute, no effect

same as true

One process: Control flow

```
int x, y;

active proctype P() {
main:
  if
    :: x == 0 -> x = 1;
    :: y == 0 -> y = 1;

    :: x == 1 && y == 1
      -> x = 0; y = 0;
  fi;
  goto main;
}
```

A guarded command

non-deterministically selects an option whose first expression is not blocked.

continues executing the rest of the option step-by-step.

One process: Control flow (cont.)

```
int x = 0, y = 0;

active
proctype P() {
main:
 if
  :: x == 0 -> x = 1;
  :: y == 0 -> y = 1;

  :: x == 1 && y == 1
   -> x = 0; y = 0;
 fi;
 goto main;
}
```

Run 1	Run 2	Run 3
x=0,y=0	x=0,y=0	x=0,y=0
x=1,y=0	x=0,y=1	x=1,y=0
x=1,y=1	x=1,y=1	x=1,y=1
x=0,y=0	x=0,y=0	x=0,y=0
x=0,y=1	x=1,y=0	x=1,y=1
x=1,y=1	x=1,y=1	x=1,y=1
x=0,y=0	x=0,y=0	x=0,y=0

One process: Loops

```
int x;

active proctype P() {
  do
    :: x == 10 -> x = 0;
    :: x == 10 -> break;
    :: x < 10  -> x++;
  od;

A:
  if
    :: x == 10 -> x = 0;
    :: x == 10 -> goto B;
    :: x < 10  -> x++;
  fi;
  goto A;
B:
}
```

a do..od loop

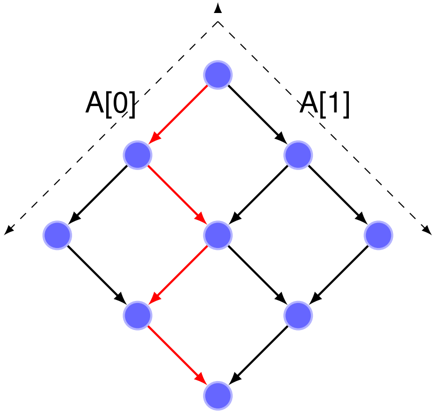
basically the same. goto A introduces one more step

Many Processes: Interleavings

Pure interleaving semantics

Every statement is executed atomically

```
int x = 0, y = 1;

active[2] proctype A() {
  x = 1 - x;
  y = 1 - y;
}
```



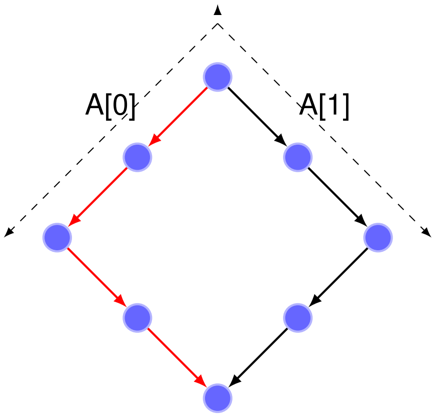
The red path is an example execution where the steps of processes 0 and 1 alternate.

Many Processes: Atomics

use `atomic { ... }` to make execution of a sequence indivisible.

non-deterministic choice with `if..fi` is still allowed!

```
int x = 0, y = 1;

active[2] proctype A() {
  atomic {
    x = 1 - x;
    y = 1 - y;
  }
}
```

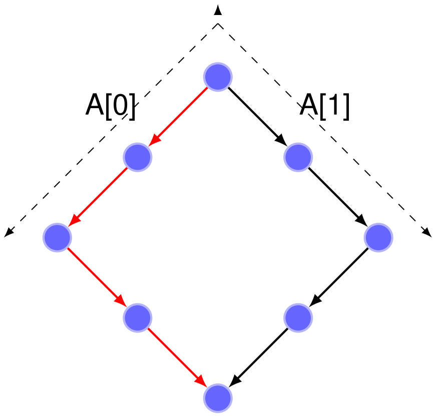


Many Processes: Atomics

use **atomic { ... }** to make execution of a sequence indivisible.

non-deterministic choice with `if..fi` is still allowed!

```
int x = 0, y = 1;

active[2] proctype A() {
  atomic {
    x = 1 - x;
    y = 1 - y;
  }
}
```



Larger atomic steps lead to less possible paths and states.
Note: different atomicity degrees may lead to different verification results

(Asynchronous) message passing

```
mtype = { A, B };
chan chan1 = [1] of { mtype };
chan chan2 = [1] of { mtype };

active proctype Ping() {
  chan1!A;
  do
    :: chan2?B -> chan1!A;
  od;
}

active proctype Pong() {
  do
    :: chan1?A -> chan2!B;
  od;
}
```

queue of size 1

insert A to “chan1”

when B is on the top of “chan2”, remove it and insert A to “chan1”

Blocking receive

```
mtype = { A, B };
chan chan1 = [1] of { mtype };
chan chan2 = [1] of { mtype };

active proctype Ping() {
  chan1!A;
  do :: chan2?B ->  ⟵ deadlock!
        chan1!A;
  od;
}

active proctype Pong() {
  do :: chan1?A ->
        chan1?A; ⟵ deadlock!
        chan2!B;
  od;
}
```

Ping sends A, Pong receives A, chan1?A is blocked

Blocking send

```
mtype = { A, B };
chan chan1 = [1] of { mtype };
chan chan2 = [1] of { mtype };
active proctype Ping() {
  chan1!A;
  do :: chan2?B ->
       chan1!A;
       chan1!A;
       chan1!A; ⟵ deadlock!
       chan1!A;
  od;
}

active proctype Pong() {
  do :: chan1?A ->
       chan2!B; ⟵ deadlock!
  od;
}
```

When chan1=[A] and chan2=[B], the system deadlocks

The shortest counter-example has 10 steps

Use Spin to find it

Promela vs. C

PROMELA looks like C

But it is not!

Non-determinism in the if statements (internal non-determinism)

Non-determinstic scheduler (external non-determinism)

Atomic statements

Message passing

PROMELA is a modeling language

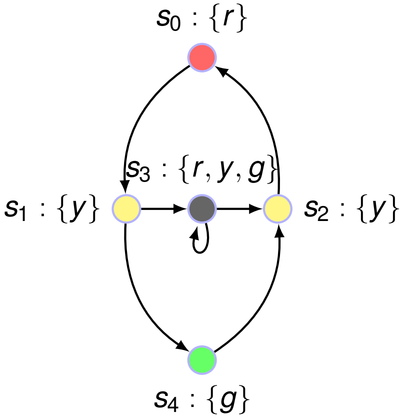
Preliminaries:

Kripke Structures
Linear Temporal Logic (LTL)
Control Flow Automata (CFA)

Kripke structures

A Kripke structure is a $M = (S, S_0, R, AP, L)$, where:

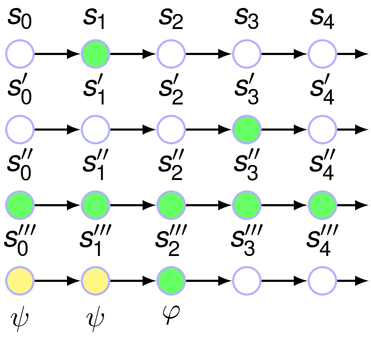
- S is a set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is a transition relation,
- AP is a set of atomic propositions,
- $L : S \to 2^{AP}$ is a state-labeling function.



Linear Temporal Logic

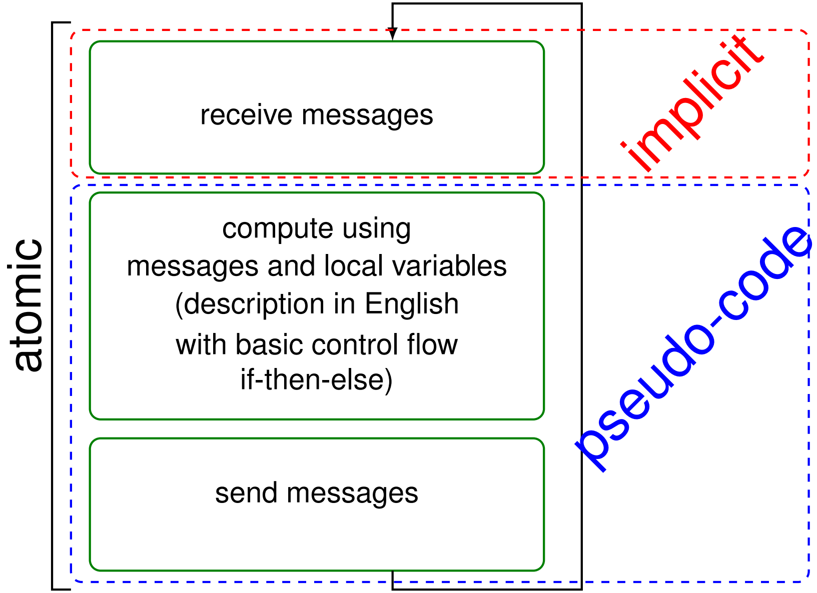
An LTL formula is defined inductively w.r.t. atomic propositions *AP*:

- (*base*) $p \in AP$ is an LTL formula,
- if φ and ψ are LTL formulas, then the following expressions are LTL formulas:
  - Nexttime: $\mathbf{X}\,\varphi$,
  - Eventually: $\mathbf{F}\,\varphi$,
  - Globally: $\mathbf{G}\,\varphi$,
  - Until: $\psi\,\mathbf{U}\,\varphi$.
  - Boolean combinations: $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$.



s_0 s_1 s_2 s_3 s_4
s'_0 s'_1 s'_2 s'_3 s'_4
s''_0 s''_1 s''_2 s''_3 s''_4
s'''_0 s'''_1 s'''_2 s'''_3 s'''_4
ψ ψ φ

Recall: Typical Structure of a Computation Step

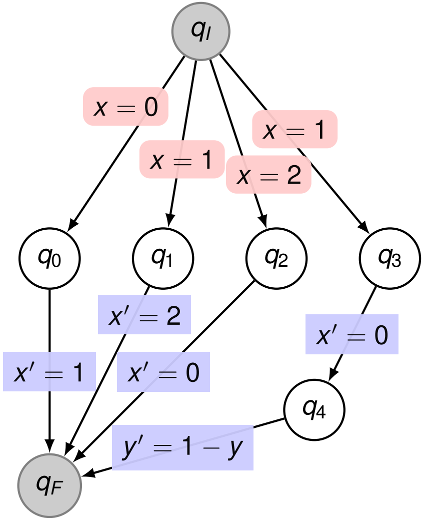


CFA: Intermediate representation

Intermediate representation of a loop body: a path from q_I to q_F encodes one iteration.

Every variable is assigned at most once (SSA).

```
active proctype P() {
 int x, y;

 do
  :: x == 0 -> x = 1;
  :: x == 1 -> x = 2;
  :: x == 2 -> x = 0;
  :: x == 1
     -> x = 0; y = 1 - y;
 od;
}
```



Example: from a CFA to a Kripke structure

Kripke structure $M(n,t,f) = (S, S_0, R, AP, L)$ of $N(n,t,f)$ processes

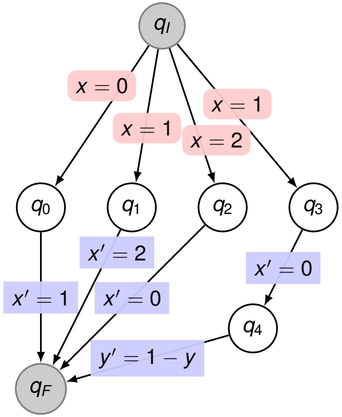
For a path π from q_I to q_F construct a formula $\phi_\pi(x, y, x', y')$

A state is a pair of $\overline{x} = (x_1, \dots, x_N) \in \mathbb{N}^N$ and $\overline{y} = (y_1, \dots, y_N) \in \mathbb{N}^N$ and the initial states are $S_0 = \{(0, \dots, 0)\}$

$((\overline{x}, \overline{y}), (\overline{x}', \overline{y}')) \in R$ iff there are process index k. $1 \le k \le N$ and path π:

- [k moves]: $\phi_\pi(x_k, y_k, x'_k, y'_k)$ holds
- [others do not]: $\forall i \in \{1, \dots, N\} \setminus \{k\}$. $x'_i = x_i$, $y'_i = y_i$.

Propositions
$AP = \{[\exists i.\ y_i \neq 0], [\forall i.\ y_i \neq 0]\}$
and a state $(\overline{x}, \overline{y})$ is labeled as:
$p \in L((\overline{x}, \overline{y}))$ iff $(\overline{x}, \overline{y}) \models p$.



Example: Properties of ST87 in LTL

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

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

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

$$\mathbf{G}\left((\bigwedge_{i=1}^{n-f} v_i = 1) \rightarrow \mathbf{F}\, (\bigvee_{j=1}^{n-f} accept_j = 1) \right)$$ Liveness

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

$$\mathbf{G}\left((\bigvee_{i=1}^{n-f} accept_i = 1) \rightarrow \mathbf{F}\, (\bigwedge_{j=1}^{n-f} accept_j = 1) \right)$$ Liveness

Model Checking Problems

Finite-state MC

Input:

- a process template P,
- an LTL formula φ (including fairness),
- values of parameters n, t, and f.

Problem: check, whether $M(n, t, f) \models \varphi$.

Parameterized MC

Input:

- a process template P,
- an LTL formula ϕ (including fairness)
  with atomic propositions of the form $[\exists i. x_i < y]$ and $[\forall i. x_i < y]$

Problem: check, whether $\forall n, t, f : n > 3t \wedge t \geq f \wedge f \geq 0.\ M(n, t, f) \models \phi$.

Parameterized modeling & Non-parameterized model checking

as in SPIN'13: (John *et al.*, 2013)

Modeling of threshold-based algorithms in Promela. . .

We introduce efficient encoding of threshold-based
fault-tolerant algorithms in PROMELA
(with parametrization!)

Verify safety and liveness of fault-tolerant algorithms (fixed parameters).

Find counterexamples for parameters known from the literature.

This proves adequacy of our modeling.

Modeling of threshold-based algorithms in Promela...

We introduce efficient encoding of threshold-based fault-tolerant algorithms in PROMELA (with parametrization!)

Verify safety and liveness of fault-tolerant algorithms (fixed parameters).

Find counterexamples for parameters known from the literature.

This proves adequacy of our modeling.

For our method, we exploit specifics of FTDAs:

1. central feature of the algorithms (message counting);
2. specific message passing (we do not need to know who sent but how many of them sent messages);
3. the way faults affect messages (again, counting messages).

Case Studies

We consider a number of threshold-based algorithms.

Our running example **ST87** for

1. Byzantine faults (BYZ)
2. omission faults (OMIT)
3. symmetric faults (SYMM)
4. clean crashes (CLEAN).

5. Forklore reliable broadcast for clean crashes
[Chandra & Toueg 96, **CT96**]

(to be continued)

Characteristics of the FTDA by Srikanth & Toueg, 87

Variables of process i

```
$v_i$: {0, 1} initially 0 or 1
$accept_i$: {0, 1} initially 0
```

An atomic 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;
```

Characteristics of the FTDA by Srikanth & Toueg, 87

```
Variables of process $i$
 $v_i$: {0, 1} initially 0 or 1
 $accept_i$: {0, 1} initially 0

An atomic 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;
```

- the algorithm consists of threshold-guarded commands, only
- thresholds $t+1$ and $n-t$
- communication is by “send to all”
- how processes distinguish distinct senders is not part of the algorithm (i.e., algorithm description is high level)

Case Studies (cont.): Larger Algorithms

more involved algorithms in the purely asynchronous setting:

6. Asynchronous Byzantine Agreement (Bracha & Toueg 85, **BT85**)
    - Byzantine faults
    - two phases and two message types
    - five status values
    - properties: unforgeability, correctness (liveness), agreement (liveness)
7. Condition-based Consensus (Mostéfaoui et al. 01, **MRRR01**)
    - crash faults
    - two phases and four message types
    - nine status variables
    - properties: validity, agreement, termination (liveness)
8. Fast Byzantine Consensus: common case (Martin, Alvisi 06, **MA06**)
    - Byzantine faults
    - the core part of the algorithm
    - no cryptography

Experimental Results at Glance

Algorithm	Fault	Parameters	Resilience	Properties	Time
1. ST87	BYZ	$n=7, t=2, f=2$	$n>3t$	**U**, **C**, **R**	6 sec.
1. ST87	BYZ	$n=7, t=3, f=2$	$n>3t$	**U**, **C**, **R**	5 sec.
1. ST87	BYZ	$n=7, t=1, f=2$	$n>3t$	**U**, **C**, **R**	1 sec.
2. ST87	OMIT	$n=5, t=2, f=2$	$n>2t$	**U**, **C**, **R**	4 sec.
2. ST87	OMIT	$n=5, t=2, f=3$	$n>2t$	**U**, **C**, **R**	5 sec.
3. ST87	SYMM	$n=5, t=1, f_p=1, f_s=0$	$n>2t$	**U**, **C**, **R**	1 sec.
3. ST87	SYMM	$n=5, t=2, f_p=3, f_s=1$	$n>2t$	**U**, **C**, **R**	1 sec.
4. ST87	CLEAN	$n=3, t=2, f_c=2, f_{nc}=0$	$n>t$	**U**, **C**, **R**	1 sec.
5. CT96	CRASH	$n=2$	—	**U**, **C**, **R**	1 sec.
6. BT85	BYZ	$n=5, t=1, f=1$	$n>3t$	**R**	131 sec.
6. BT85	BYZ	$n=5, t=1, f=2$	$n>3t$	**R**	1 sec.
6. BT85	BYZ	$n=5, t=2, f=2$	$n>3t$	**R**	1 sec.
7. MRRR01	CRASH	$n=3, t=1, f=1$	$n>2t$	**V0**, **V1**, **A**, **T**	1 sec.
7. MRRR01	CRASH	$n=3, t=1, f=2$	$n>2t$	**V0**, **V1**, **A**, **T**	1 sec.
8. MA06	BYZ	$p=4, a=6, l=4,$ $t=1, f=1$	$p>3t$, $a>5t$, $l>3t$	**CS1**, **CS3**, **CL1**, **CL2**	3 hrs.
8. MA06	BYZ	$p=4, a=5, l=4,$ $t=1, f=1$	$p>3t$, $a>5t$, $l>3t$	**CS1**, **CS3**, **CL1**, **CL2**	14 min.
8. MA06	BYZ	$p=4, a=6, l=4,$ $t=1, f=2$	$p>3t$, $a>5t$, $l>3t$	**CS1**, **CS3**, **CL1**, **CL2**	2 sec.

Experimental Results at Glance

Algorithm	Fault	Parameters	Resilience	Properties	Time
1. ST87	BYZ	$n = 7, t = 2, f = 2$	$n > 3t$	**U**, **C**, **R**	6 sec.
1. ST87	BYZ	$n = 7, t = 3, f = 2$	$n > 3t$	**U**, **C**, **R**	5 sec.
1. ST87	BYZ	$n = 7, t = 1, f = 2$	$n > 3t$	**U**, **C**, **R**	1 sec.
2. ST87	OMIT	$n = 5, t = 2, f = 2$	$n > 2t$	**U**, **C**, **R**	4 sec.
2. ST87	OMIT	$n = 5, t = 2, f = 3$	$n > 2t$	**U**, **C**, **R**	5 sec.
3. ST87	SYMM	$n = 5, t = 1, f_p = 1, f_s = 0$	$n > 2t$	**U**, **C**, **R**	1 sec.
3. ST87	SYMM	$n = 5, t = 2, f_p = 3, f_s = 1$	$n > 2t$	**U**, **C**, **R**	1 sec.
4. ST87	CLEAN	$n = 3, t = 2, f_c = 2, f_{nc} = 0$	$n > t$	**U**, **C**, **R**	1 sec.
5. CT96	CRASH	$n = 2$	—	**U**, **C**, **R**	1 sec.
6. BT85	BYZ	$n = 5, t = 1, f = 1$	$n > 3t$	**R**	131 sec.
6. BT85	BYZ	$n = 5, t = 1, f = 2$	$n > 3t$	**R**	1 sec.
6. BT85	BYZ	$n = 5, t = 2, f = 2$	$n > 3t$	**R**	1 sec.
7. MRRR01	CRASH	$n = 3, t = 1, f = 1$	$n > 2t$	**V0**, **V1**, **A**, **T**	1 sec.
7. MRRR01	CRASH	$n = 3, t = 1, f = 2$	$n > 2t$	**V0**, **V1**, **A**, **T**	1 sec.
8. MA06	BYZ	$p = 4, a = 6, l = 4, t = 1, f = 1$	$p > 3t, a > 5t, l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**	3 hrs.
8. MA06	BYZ	$p = 4, a = 5, l = 4, t = 1, f = 1$	$p > 3t, a > 5t, l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**	14 min.
8. MA06	BYZ	$p = 4, a = 6, l = 4, t = 1, f = 2$	$p > 3t, a > 5t, l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**	2 sec.

Experimental Results at Glance

Algorithm	Fault	Parameters	Resilience	Properties
1. **ST87**	BYZ	$n = 7, t = 2, f = 2$	$n > 3t$	**U**, **C**, **R**
1. **ST87**	BYZ	$n = 7, t = 3, f = 2$	$n > 3t$	**U**, **C**, **R**
1. **ST87**	BYZ	$n = 7, t = 1, f = 2$	$n > 3t$	**U**, **C**, **R**
2. **ST87**	OMIT	$n = 5, t = 2, f = 2$	$n > 2t$	**U**, **C**, **R**
2. **ST87**	OMIT	$n = 5, t = 2, f = 3$	$n > 2t$	**U**, **C**, **R**
3. **ST87**	SYMM	$n = 5, t = 1, f_p = 1, f_s = 0$	$n > 2t$	**U**, **C**, **R**
3. **ST87**	SYMM	$n = 5, t = 2, f_p = 3, f_s = 1$	$n > 2t$	**U**, **C**, **R**
4. **ST87**	CLEAN	$n = 3, t = 2, f_c = 2, f_{nc} = 0$	$n > t$	**U**, **C**, **R**
5. **CT96**	CRASH	$n = 2$	—	**U**, **C**, **R**

Experimental Results at Glance

Algorithm	Fault	Parameters	Resilience	Properties	Time
1. **ST87**	BYZ	$n = 7, t = 2, f = 2$	$n > 3t$	**U**, **C**, **R**	6 sec.
1. **ST87**	BYZ	$n = 7, t = 3, f = 2$	$n > 3t$	**U**, **C**, **R**	5 sec.
1. **ST87**	BYZ	$n = 7, t = 1, f = 2$	$n > 3t$	**U**, **C**, **R**	1 sec.
2. **ST87**	OMIT	$n = 5, t = 2, f = 2$	$n > 2t$	**U**, **C**, **R**	4 sec.
2. **ST87**	OMIT	$n = 5, t = 2, f = 3$	$n > 2t$	**U**, **C**, **R**	5 sec.
3. **ST87**	SYMM	$n = 5, t = 1, f_p = 1, f_s = 0$	$n > 2t$	**U**, **C**, **R**	1 sec.
3. **ST87**	SYMM	$n = 5, t = 2, f_p = 3, f_s = 1$	$n > 2t$	**U**, **C**, **R**	1 sec.
4. **ST87**	CLEAN	$n = 3, t = 2, f_c = 2, f_{nc} = 0$	$n > t$	**U**, **C**, **R**	1 sec.
5. **CT96**	CRASH	$n = 2$	—	**U**, **C**, **R**	1 sec.
6. **BT85**	BYZ	$n = 5, t = 1, f = 1$	$n > 3t$	**R**	131 sec.
6. **BT85**	BYZ	$n = 5, t = 1, f = 2$	$n > 3t$	**R**	1 sec.
6. **BT85**	BYZ	$n = 5, t = 2, f = 2$	$n > 3t$	**R**	1 sec.
7. **MRRR01**	CRASH	$n = 3, t = 1, f = 1$	$n > 2t$	**V0**, **V1**, **A**, **T**	1 sec.
7. **MRRR01**	CRASH	$n = 3, t = 1, f = 2$	$n > 2t$	**V0**, **V1**, **A**, **T**	1 sec.
8. **MA06**	BYZ	$p = 4, a = 6, l = 4,$ $t = 1, f = 1$	$p > 3t$, $a > 5t$, $l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**	3 hrs.
8. **MA06**	BYZ	$p = 4, a = 5, l = 4,$ $t = 1, f = 1$	$p > 3t$, $a > 5t$, $l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**	14 min.
8. **MA06**	BYZ	$p = 4, a = 6, l = 4,$ $t = 1, f = 2$	$p > 3t$, $a > 5t$, $l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**	2 sec.

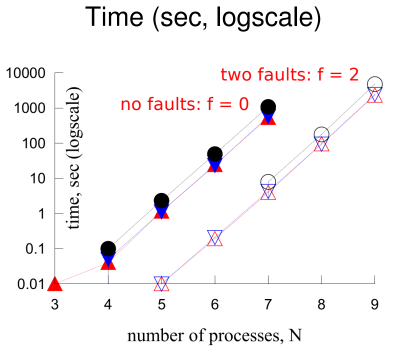
Experimental Results at Glance

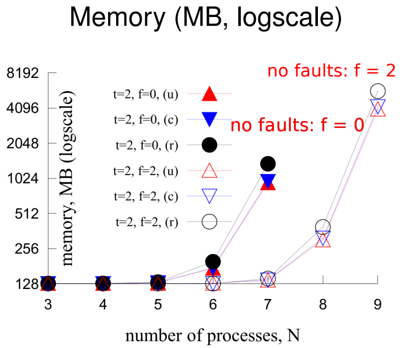
6. BT85	Byz	$n = 5, t = 1, f = 1$	$n > 3t$	**R**
6. BT85	Byz	$n = 5, t = 1, f = 2$	$n > 3t$	**R**
6. BT85	Byz	$n = 5, t = 2, f = 2$	$n > 3t$	**R**
7. MRRR01	Crash	$n = 3, t = 1, f = 1$	$n > 2t$	**V0**, **V1**, **A**, **T**
7. MRRR01	Crash	$n = 3, t = 1, f = 2$	$n > 2t$	**V0**, **V1**, **A**, **T**
8. MA06	Byz	$p = 4, a = 6, l = 4,$ $t = 1, f = 1$	$p > 3t$, $a > 5t$, $l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**
8. MA06	Byz	$p = 4, a = 5, l = 4,$ $t = 1, f = 1$	$p > 3t$, $a > 5t$, $l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**
8. MA06	Byz	$p = 4, a = 6, l = 4,$ $t = 1, f = 2$	$p > 3t$, $a > 5t$, $l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**

Experimental Results at Glance

Algorithm	Fault	Parameters	Resilience	Properties	Time
1. ST87	BYZ	$n=7, t=2, f=2$	$n>3t$	**U**, **C**, **R**	6 sec.
1. ST87	BYZ	$n=7, t=3, f=2$	$n>3t$	**U**, **C**, **R**	5 sec.
1. ST87	BYZ	$n=7, t=1, f=2$	$n>3t$	**U**, **C**, **R**	1 sec.
2. ST87	OMIT	$n=5, t=2, f=2$	$n>2t$	**U**, **C**, **R**	4 sec.
2. ST87	OMIT	$n=5, t=2, f=3$	$n>2t$	**U**, **C**, **R**	5 sec.
3. ST87	SYMM	$n=5, t=1, f_p=1, f_s=0$	$n>2t$	**U**, **C**, **R**	1 sec.
3. ST87	SYMM	$n=5, t=2, f_p=3, f_s=1$	$n>2t$	**U**, **C**, **R**	1 sec.
4. ST87	CLEAN	$n=3, t=2, f_c=2, f_{nc}=0$	$n>t$	**U**, **C**, **R**	1 sec.
5. CT96	CRASH	$n=2$	—	**U**, **C**, **R**	1 sec.
6. BT85	BYZ	$n=5, t=1, f=1$	$n>3t$	**R**	131 sec.
6. BT85	BYZ	$n=5, t=1, f=2$	$n>3t$	**R**	1 sec.
6. BT85	BYZ	$n=5, t=2, f=2$	$n>3t$	**R**	1 sec.
7. MRRR01	CRASH	$n=3, t=1, f=1$	$n>2t$	**V0**, **V1**, **A**, **T**	1 sec.
7. MRRR01	CRASH	$n=3, t=1, f=2$	$n>2t$	**V0**, **V1**, **A**, **T**	1 sec.
8. MA06	BYZ	$p=4, a=6, l=4,$ $t=1, f=1$	$p>3t$, $a>5t$, $l>3t$	**CS1**, **CS3**, **CL1**, **CL2**	3 hrs.
8. MA06	BYZ	$p=4, a=5, l=4,$ $t=1, f=1$	$p>3t$, $a>5t$, $l>3t$	**CS1**, **CS3**, **CL1**, **CL2**	14 min.
8. MA06	BYZ	$p=4, a=6, l=4,$ $t=1, f=2$	$p>3t$, $a>5t$, $l>3t$	**CS1**, **CS3**, **CL1**, **CL2**	2 sec.

Experimental Results: on ST87, the Byzantine Case





- The more faults we have, the easier the problem is:
  - Two faults: we can check the systems of up to nine processes
  - No faults: we can check the systems of up to seven processes
- Precision of modeling: we found counter-examples for the corner cases $n = 3t$ and $f > t$, where the resilience condition is violated.

(June 2013: somebody wrote on Wikipedia that $n = 3t$ should work :-)

Discussion of the specifications

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

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

The specification of Byzantine FTDAs have the following features:

- Only the states of correct processes are evaluated.

  Faulty processes may be Byzantine. (no assumption on behavior)
- Specifications do not talk about individual processes.

  Only global safety and progress are important.
- Indexed temporal logic is not required!

  Quantification over processes is on the level of atomic propositions.

Threshold-Guarded Distributed Algorithms

Standard construct: quantified guards (t=f=0)

- Existential Guard

  `if received` m `from` *`some`* `process then ...`
- Universal Guard

  `if received` m `from` *`all`* `processes then ...`

Threshold-Guarded Distributed Algorithms

Standard construct: quantified guards (t=f=0)

- Existential Guard

  ```
  if received m from some process then ...
  ```

- Universal Guard

  ```
  if received m from all processes then ...
  ```

what if faults might occur?

Threshold-Guarded Distributed Algorithms

Standard construct: quantified guards (t=f=0)

- Existential Guard

```
if received $m$ from some process then ...
```

- Universal Guard

```
if received $m$ from all processes then ...
```

what if faults might occur?

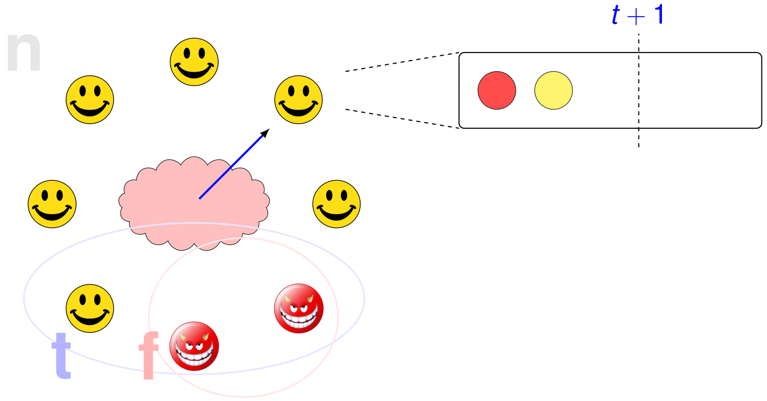
Fault-Tolerant Algorithms: *n* processes, at most *t* are Byzantine

- Threshold Guard

```
if received $m$ from $n-t$ processes then ...
```

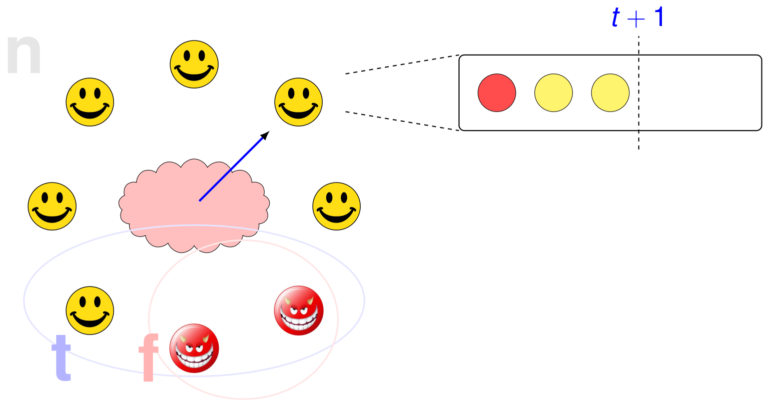
- (the processes cannot refer to **f**!)

Counting Argument in Threshold-Guarded Algorithms



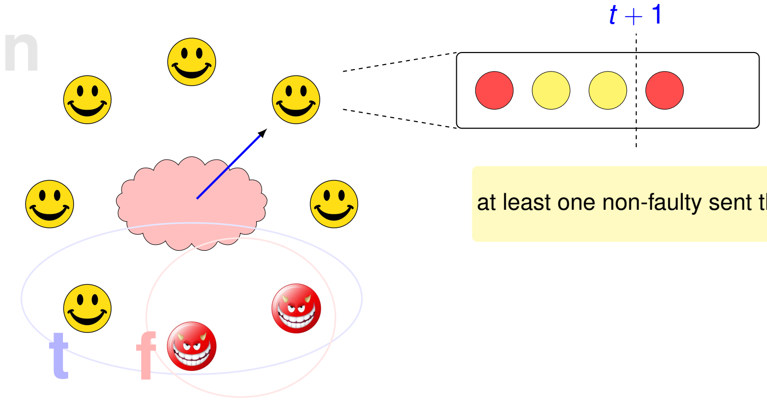
Correct processes count incoming messages from distinct processes

Counting Argument in Threshold-Guarded Algorithms



Correct processes count incoming messages from distinct processes

Counting Argument in Threshold-Guarded Algorithms



at least one non-faulty sent the message

Correct processes count incoming messages from distinct processes

Modeling threshold-based algorithms in Promela

As the distributed algorithms are given in pseudo-code,

we have to decide on how to encode in PROMELA:

- send to all and receive
- counting expressions "received <m> from $n - t$ distinct processes"
- faults

Modeling threshold-based algorithms in Promela

As the distributed algorithms are given in pseudo-code,

we have to decide on how to encode in PROMELA:

- send to all and receive
- counting expressions "received <m> from $n - t$ distinct processes"
- faults

In what follows, we compare side-by-side two solutions:

- A straightforward encoding with PROMELA channels and explicit representation of faulty processes. [Solution 1]
- An advanced encoding with shared variables and fault injection. [Solution 2]

Modeling threshold-based algorithms in Promela

As the distributed algorithms are given in pseudo-code,

we have to decide on how to encode in PROMELA:

- send to all and receive
- counting expressions "received $<$m$>$ from $n - t$ distinct processes"
- faults

In what follows, we compare side-by-side two solutions:

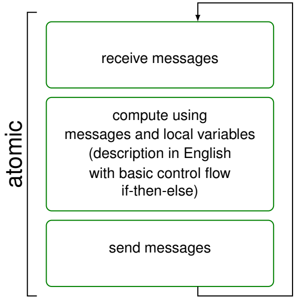
- A straightforward encoding with PROMELA channels and explicit representation of faulty processes. [Solution 1]
- An advanced encoding with shared variables and fault injection. [Solution 2]

To decouple encoding of reliable message passing and of faults,

- we first consider message passing without faults,
- and then show how to encode faults.

Template in Promela

We implement the following loop on the right.



```
/* shared state:
   a variable or a channel */
active proctype[N(n,t,f)] P(){
 /* local variable to count
    messages from distinct
    processes               */
 int nrcvd;
 /* initialization */
loop: atomic {
/*
1. receive and count messages
2. compute using nrcvd
3. send messages */
 }
 goto loop;
}
```

Modeling Message Passing

All our case studies are designed with the assumption of classic reliable asynchronous message passing as in (Fischer *et al.*, 1985):

- non-blocking communication,

  operations “receive” and “send” are executed immediately.

- if a message can be received now, it may be also received later,

  a process does not have to receive a message as soon as it is able to.

- every sent message is eventually received,

  but there are no bounds on the delays.

Solution 1: Message Passing using Promela channels

A straightforward encoding using message channels:

```
/* message type */
mtype = { ECHO };
/* point-to-point channels */
chan p2p[N][N] = [1] of { mtype };
/* tag received messages */
bit  rx[N][N];
```

Sending a message to all processes:

```
for (i : 1 .. N) { p2p[_pid][i]!ECHO; }
```

Note: `_pid` denotes the process identifier in PROMELA

(we use it solely to encode message passing).

Solution 1: Message Passing (cont.)

Receiving and counting messages from distinct processes

(no faults yet):

```
/* local*/ int nrcvd = 0; /* initially, no messages*/
...
i = 0;
do  /* is there a message from process i?*/
   :: (i < N) && nempty(p2p[i][_pid]) ->
     p2p[i][_pid]?ECHO; /* remove it */
     if
       :: !rx[i][_pid] -> /* 1. the first time: */
          rx[i][_pid] = 1; /* a. mark as received  */
          nrcvd++; break;  /* b. increase local counter*/
          :: rx[i][_pid];  /* 2. ignore a duplicate */
     fi; i++; /* next process */
   :: (i < N) -> i++; /* receive nothing from i */
   :: i == N -> break;
od
```

Solution 2: Simulating message passing with variables

Keeping the number of send-to-all's by (correct) processes:

```
int nsnt; /* shared variable */
 /* number of send-to-all's sent by correct processes */
```

Sending a message to all:

```
nsnt++;
```

Receiving and counting messages from distinct processes (no faults):

```
if /* pick a larger value ≤ nsnt */
  :: ((nrcvd + 1) < nsnt) ->
    nrcvd++; /* one more message */
  :: skip;    /* or nothing */
fi;
```

Reliable communication as a fairness property:

$$\mathbf{F}\,\mathbf{G}\,[\forall i.\mathit{nrcvd}_i \geq \mathit{nsnt}]$$

Solution 2: Some questions you might ask

Q1: instead of

```
if
  :: ((nrcvd + 1) < nsnt) ->
    nrcvd++; /* one more message */
  :: skip;    /* or nothing */
fi;
```

why cannot we just write:

```
nrcvd = nsnt;
```

A1: You can, but that will be another model, not [FLP85]!

[FLP85] only guarantees that every message is eventually received.

Solution 2: Some questions you might ask (cont.)

Reliable communication:

every sent message is eventually received.

Q2: Why do we write

$$\mathbf{F}\,\mathbf{G}\,[\forall i . nrcvd_i \geq nsnt] \tag{1}$$

instead of:

$$\forall i.\ \mathbf{G}\,\mathbf{F}\,[nrcvd_i \geq nsnt] \tag{2}$$

A2: We like to write (2), but it will require us to use another logic called indexed LTL, which will cause problems in the parameterized case.

For threshold-based algorithms, the value of `nsnt` is changes at most *n* times.

Under this assumption, (2) is equivalent (1).

Solution 1 (cont.): Explicit Modeling of Faults

(Lamport *et al.*, 1982) introduce Byzantine processes that can virtually do anything.

In our case, Byzantine behavior boils down to sending `ECHO` to some of the correct processes and not sending `ECHO` to the others:

```
active[F] proctype Byz() {
step:
  atomic {
    i = 0;
    do /* send ECHO to process i */
      :: i < N -> p2p[_pid][i]!ECHO; i++;
       /* or not */
      :: i < N -> i++;
      :: i == N -> break;
    od
  };
  goto step;
}
```

Solution 2 (cont.): Injecting Faults into Message Counters

We instantiate $n - f$ correct processes and no faulty processes.

Instead, we say that the correct processes may receive up to f additional messages due to faults:

```
if :: ((nrcvd + 1) < nsnt + f) ->
       nrcvd++;     /* receive one more message */
   :: skip;  /* or nothing */
fi;
```

Solution 2 (cont.): Injecting Faults into Message Counters

We instantiate $n - f$ correct processes and no faulty processes.

Instead, we say that the correct processes may receive up to f additional messages due to faults:

```
if :: ((nrcvd + 1) < nsnt + f) ->
      nrcvd++;      /* receive one more message */
   :: skip;   /* or nothing */
fi;
```

The fairness still forces the processes to receive all the messages sent by the correct processes:

$$\mathbf{F}\,\mathbf{G}\,[\forall i.nrcvd_i \geq nsnt]$$

Note: each correct process sends at most one ECHO message.

Solution 2 (cont.): Modeling different kinds of faults

Byzantine faults (previous slide):

- create only correct processes, i.e., $n - f$ processes
  only they have to satisfy spec
- extra messages from Byzantine: `((nrcvd + 1) < nsnt + f)`
- fairness (reliable communication): $\mathbf{F}\,\mathbf{G}\,[\forall i.nrcvd_i \geq nsnt]$

Solution 2 (cont.): Modeling different kinds of faults

Byzantine faults (previous slide):

- create only correct processes, i.e., $n - f$ processes
  only they have to satisfy spec
- extra messages from Byzantine: `((nrcvd + 1) < nsnt + f)`
- fairness (reliable communication): $\mathbf{F}\,\mathbf{G}\,[\forall i.\mathit{nrcvd}_i \geq \mathit{nsnt}]$

Omission faults (processes fail to send messages):

- create all processes, i.e., n processes
  all of them are mentioned in the specification
- no additional messages: `((nrcvd + 1) < nsnt)`
- fairness (with possible message loss due to faults)
  $\mathbf{F}\,\mathbf{G}\,[\forall i.\mathit{nrcvd}_i \geq \mathit{nsnt} - f]$

Solution 2 (cont.): Modeling different kinds of faults

Byzantine faults (previous slide):

- create only correct processes, i.e., $n - f$ processes
  only they have to satisfy spec
- extra messages from Byzantine: `((nrcvd + 1) < nsnt + f)`
- fairness (reliable communication): $\mathbf{F}\,\mathbf{G}\,[\forall i.nrcvd_i \geq nsnt]$

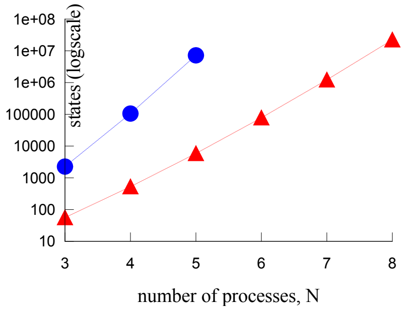
Omission faults (processes fail to send messages):

- create all processes, i.e., n processes
  all of them are mentioned in the specification
- no additional messages: `((nrcvd + 1) < nsnt)`
- fairness (with possible message loss due to faults)
  $\mathbf{F}\,\mathbf{G}\,[\forall i.nrcvd_i \geq nsnt - f]$

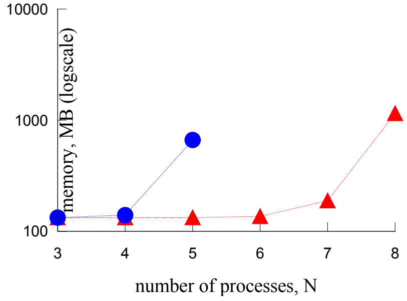
Crash faults: similar to omissions with crash control state added

Experiments: Solution 1 vs. Solution 2

States (logscale)



Memory (MB, logscale, $\leq$ 12 GB)



- Solution 1: Channels + explicit Byzantine processes (blue)
- Solution 2: shared variables + fault injection (red)

- in the presence of one Byzantine faulty process ($f = 1$)
  (case $f = 2$ runs out of memory too fast)

Summary

We show how to model threshold-based fault-tolerant algorithms starting with an imprecise description

We create PROMELA models using expert advice.

The tool demonstrates that the model behaves as predicted by theory
(for concrete values of parameters)

This reference implementation allows us to optimize the encoding

... and to make the model amenable to parameterized verification

End of Part II

References I

Biely, M., Delgado, P., Milosevic, Z., & Schiper, A. 2013 (June).
Distal: A framework for implementing fault-tolerant distributed algorithms.
Pages 1–8 of: Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on.

Castro, Miguel, Liskov, Barbara, *et al.* 1999.
Practical Byzantine fault tolerance.
Pages 173–186 of: OSDI, vol. 99.

Fischer, Michael J., Lynch, Nancy A., & Paterson, M. S. 1985.
Impossibility of Distributed Consensus with one Faulty Process.
J. ACM, **32**(2), 374–382.
http://doi.acm.org/10.1145/3149.214121.

John, Annu, Konnov, Igor, Schmid, Ulrich, Veith, Helmut, & Widder, Josef. 2013.
Towards Modeling and Model Checking Fault-Tolerant Distributed Algorithms.
Pages 209–226 of: SPIN.
LNCS, vol. 7976.

Lamport, Leslie, Shostak, Robert E., & Pease, Marshall C. 1982.
The Byzantine Generals Problem.
ACM Trans. Program. Lang. Syst., **4**(3), 382–401.

References II

Lincoln, P., & Rushby, J. 1993.
A formally verified algorithm for interactive consistency under a hybrid fault model.
Pages 402–411 of: FTCS-23.
http://dx.doi.org/10.1109/FTCS.1993.627343.

Folklore Reliable Broadcast (e.g., Chandra & Toueg, 96)

Correct processes agree on value v_i in the presence of crash faults.

Variables of process i

```
$v_i$: {0, 1} initially 0 or 1
$accept_i$: {0, 1} initially 0
```

An atomic step:

```
if ($v_i = 1$ or received <echo> from some process)
   and $accept_i = 0$
then begin
   send <echo> to all;

   $accept_i$ := 1;
end
```

Folklore Reliable Broadcast (e.g., Chandra & Toueg, 96)

Correct processes agree on value v_i in the presence of crash faults.

Variables of process i

```
$v_i$: {0, 1} initially 0 or 1
$accept_i$: {0, 1} initially 0
```

An atomic step:

```
if ($v_i = 1$ or received <echo> from some process)
    and $accept_i = 0$
then begin
    send <echo> to all;
        /* when crashing it sends to a subset of processes */
    $accept_i$ := 1;  /* it can also crash here */
end
```

Verification Problem as in Distributed Computing

Given a distributed algorithm $\mathcal{A}$ and specifications φ_U, φ_C, φ_R,

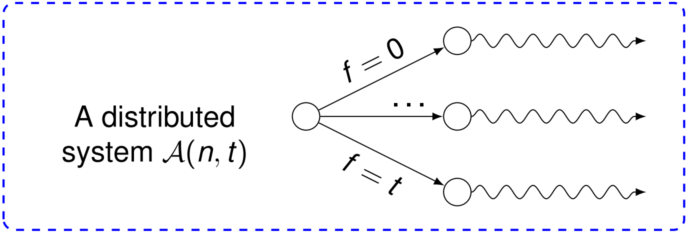
- Fix n and t with $n > 3t$,
- show that every execution of $\mathcal{A}(n, t)$ satisfies φ_U, φ_C, φ_R.

Verification Problem as in Distributed Computing

Given a distributed algorithm $\mathcal{A}$ and specifications φ_U, φ_C, φ_R,

- Fix n and t with $n > 3t$,
- show that every execution of $\mathcal{A}(n,t)$ satisfies φ_U, φ_C, φ_R.

- In every execution:
  - the number of faulty processes is restricted, i.e., $f \leq t$;
  - processes can use n and t in the code, but not f;
  - f is constant
    (if a process fails late, its “correct” behavior was a Byzantine trick).

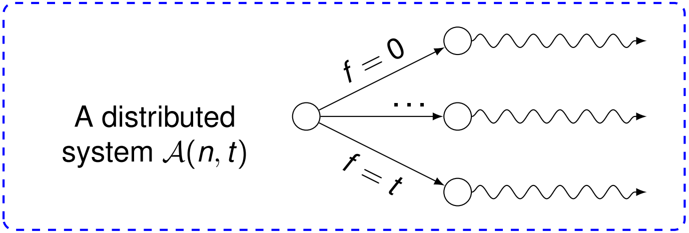


Verification Problem as in Distributed Computing

Given a distributed algorithm $\mathcal{A}$ and specifications φ_U, φ_C, φ_R,

- Fix n and t with $n > 3t$,
- show that every execution of $\mathcal{A}(n, t)$ satisfies φ_U, φ_C, φ_R.

- In every execution:
  - the number of faulty processes is restricted, i.e., $f \leq t$;
  - processes can use n and t in the code, but not f;
  - f is constant
    (if a process fails late, its “correct” behavior was a Byzantine trick).

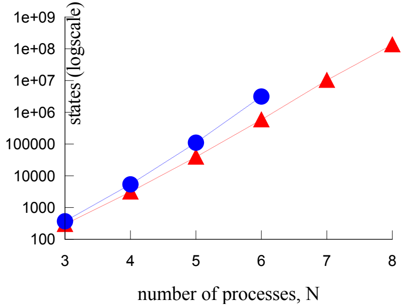


- Counterexamples when $f > t$?

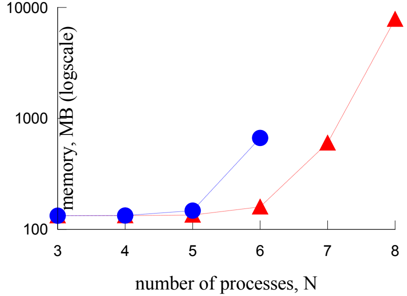
Experiments: Channels vs. Shared Variables

enumerating reachable states in SPIN with POR and state compression

States (logscale)



Memory (MB, logscale, limit of 12 GB)


memory, MB (logscale)
number of processes, N