Model Checking of Fault-Tolerant Distributed Algorithms

Part II: Modeling Fault-tolerant Distributed Algorithms

Annu Gmeiner Igor Konnov Ulrich Schmid Helmut Veith
Josef Widder

SFM-14:ESM. Bertinoro, Italy, EU

Why Model Checking?

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

Distributed Algorithms: Model Checking Challenges

- degrees of concurrency

  many degrees of partial synchrony

- 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 $n > 3t$

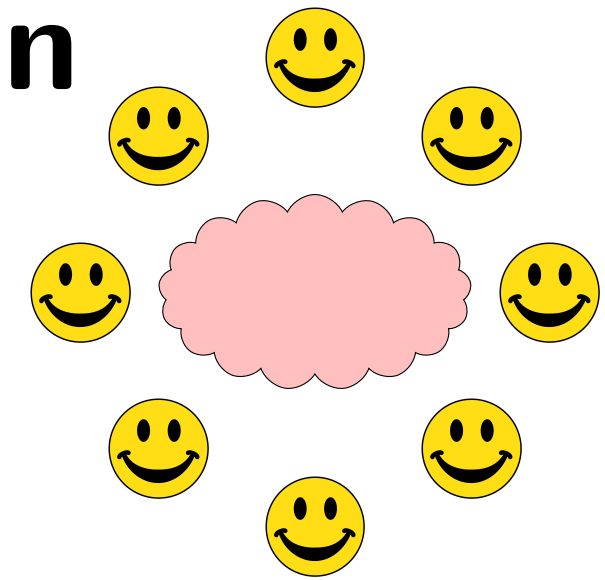
- contrast to concurrent programs

  diverse fault models (adverse environments)

- continuous time

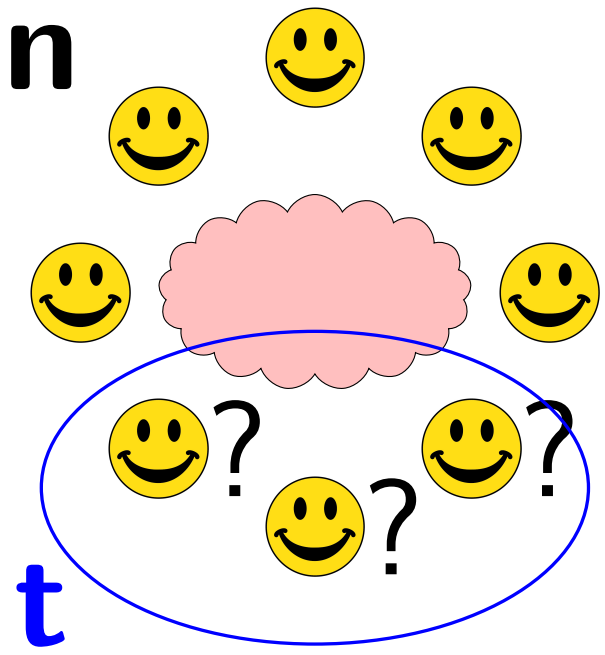
  fault-tolerant clock synchronization

Fault-tolerant distributed algorithms



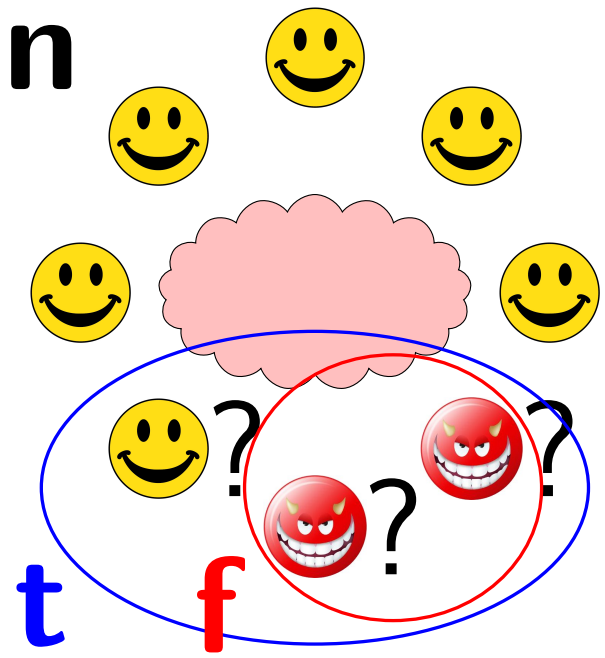
- n processes communicate by messages
- all processes know that at most t of them might be faulty
- f are actually faulty

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



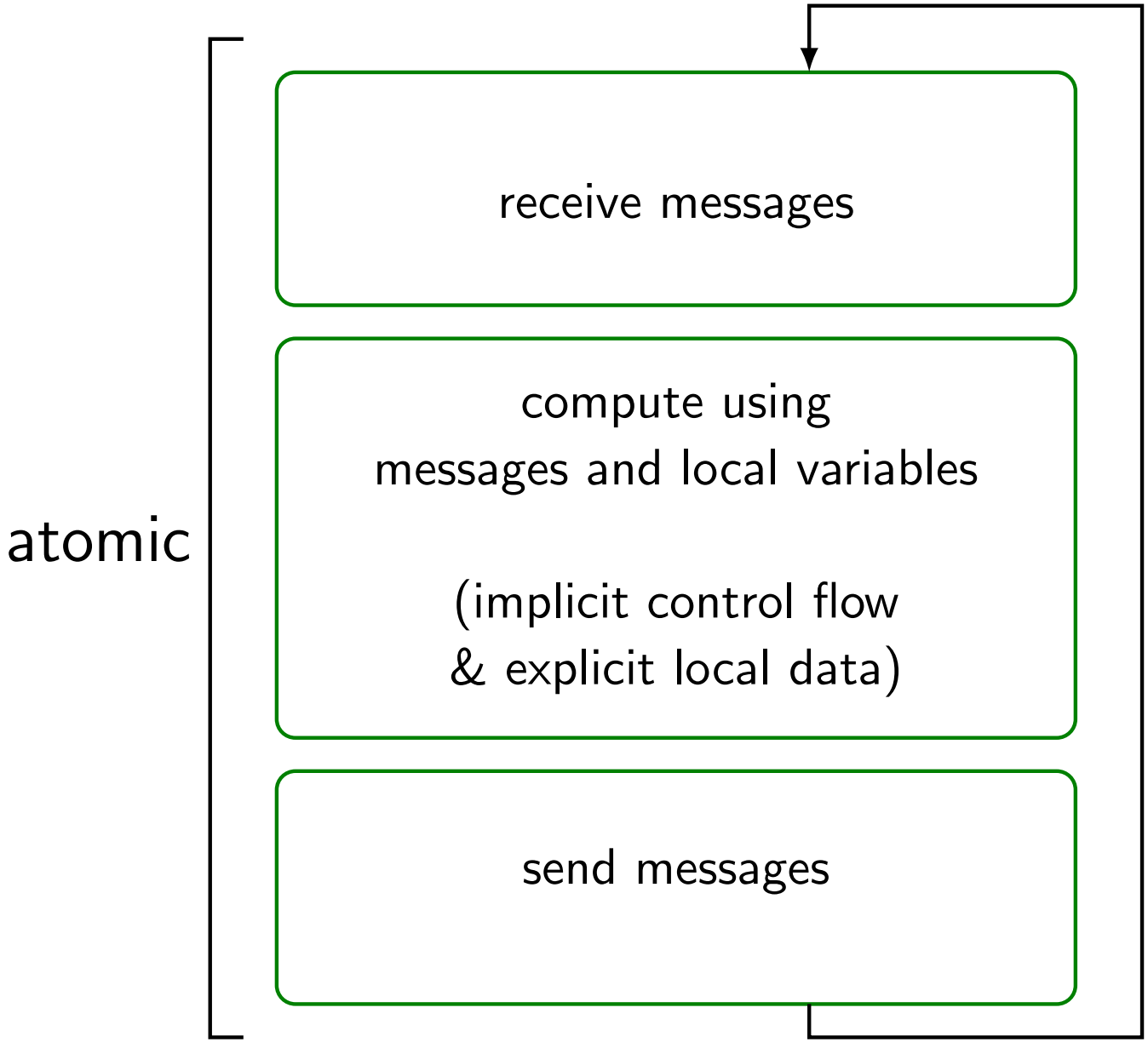
- n processes communicate by messages
- all processes know that at most t of them might be faulty
- f are actually faulty

Challenge #1: fault models

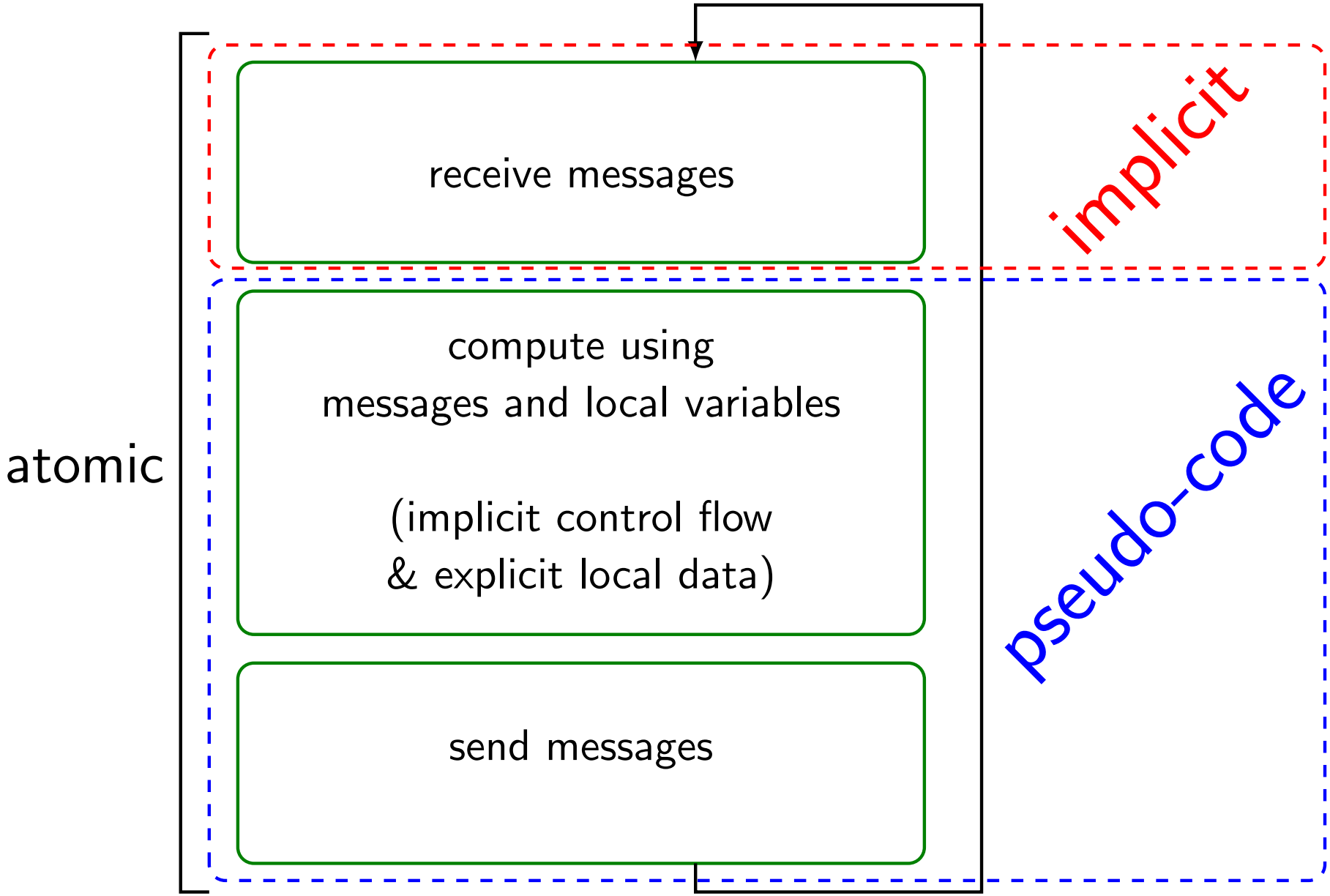
- clean crashes:
  faulty processes prematurely halt after/before "send to all"
- crash faults:
  faulty processes prematurely halt (also) in the middle of "send to all"
- omission faults:
  faulty processes follow the algorithm, but some messages sent by them might be lost
- symmetric faults:
  faulty processes send arbitrarily to all or nobody
- Byzantine faults:
  faulty processes can do anything

- hybrid models:
  combinations of the above

Typical Structure of a Computation Step



Typical Structure of a Computation Step



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 $n > 3t$
resilience condition RC

parameterized process
skeleton $P(n, t)$

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-

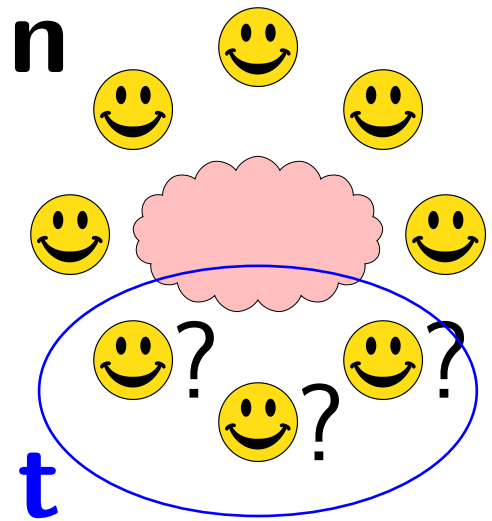
Challenge #4: Parameterized Model Checking

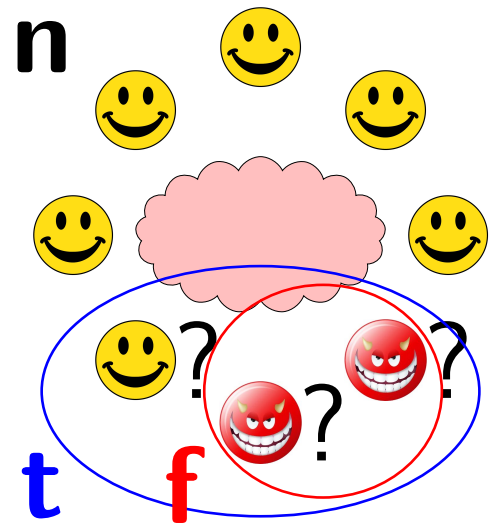
Parameterized model checking problem:

- given a parameterized family $\{M(n,t,f)\}$,
- resilience condition $RC : n > 3t \wedge t \geq f \geq 0$,
- justice constraints Φ,
- and an LTL_{-X} formula φ

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

$$M(n,t,f) \models (\Phi \rightarrow \varphi)$$





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

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

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

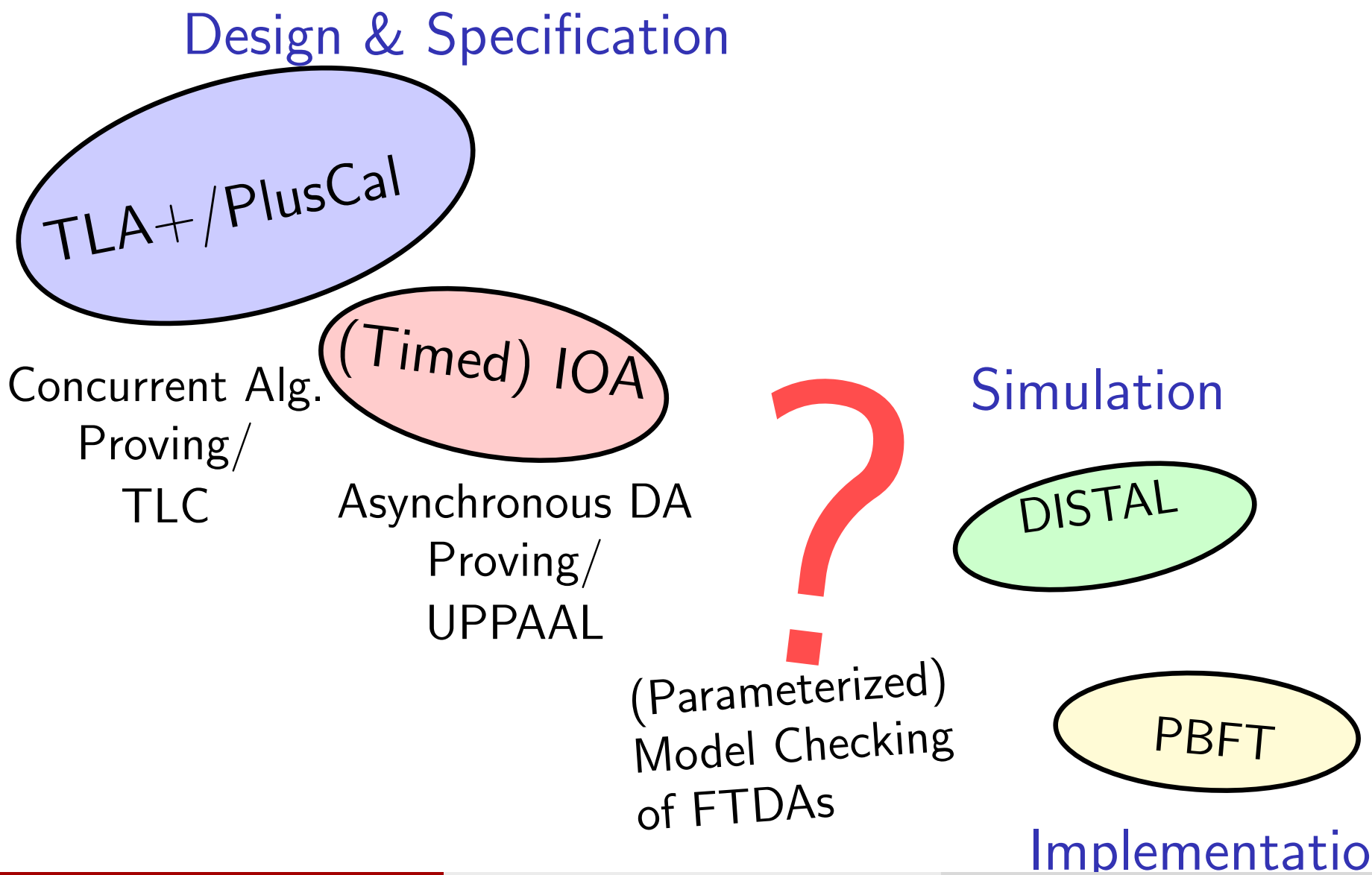
We have to model:

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

and check:

- safety and liveness
- of parameterized systems
- with unbounded types,
- non-standard fairness constraints,
- and maybe real-time.

Existing formalization frameworks



Properties of verification 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 algorithm

I/O-Automata:

- originally design to write clearer hand-written proofs
- several implementations in proof assistants
- suitable for asynchronous distributed algorithms

Properties of verification 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 algorithm

I/O-Automata:

- originally design to write clearer hand-written proofs
- several implementations in proof assistants
- suitable for asynchronous distributed algorithms

proof assistants are very general, but with low automation degree

"everything is possible, but nothing is easy"

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 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

What is it doing?

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

is it equivalent to do..od?

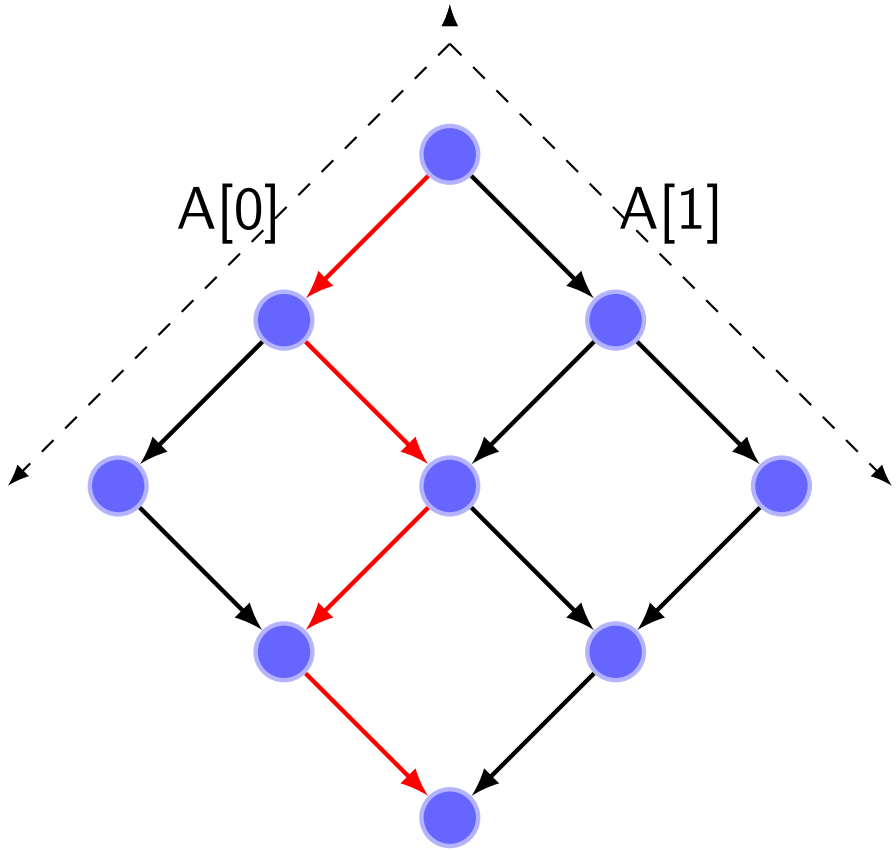
Many Processes: Interleavings

Pure interleaving semantics

Every statement is executed atomically

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

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



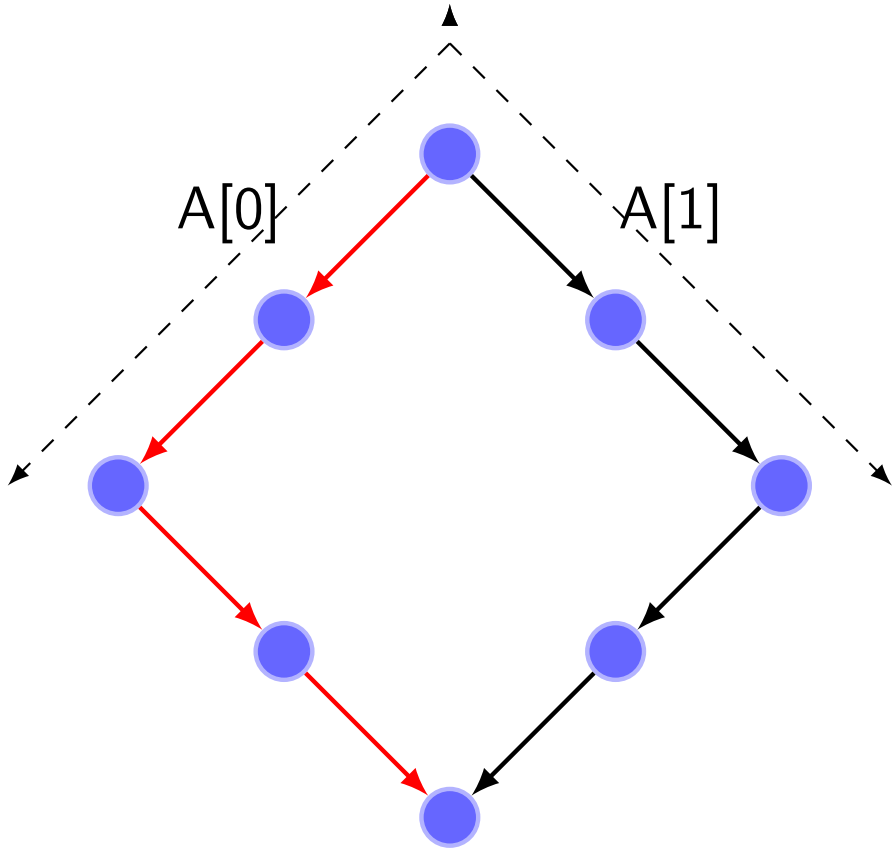
Many Processes: Atomics

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;
  }
}
```



Example: Semaphore

```
#define N 0
#define T 1
#define C 2
int x = 1;

active proctype P() {
  int sv = N;
  do
    :: sv == N -> sv = N;
    :: sv == N -> sv = T;
    :: sv == C -> /* do critical stuff */;
    :: sv == C -> sv = N; x++;
    :: atomic {
         sv == T && x < 1
              -> x--; sv = C;
       }
  od;
}
```

(Asynchronous) message passing

```
mtype = { A, B };
chan to = [1] of { mtype };
chan from = [1] of { mtype };

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

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

insert A to "to"

when B is on the top of "to", receive it and insert A to "from"

Blocking send

```
mtype = { A, B };
chan to = [1] of { mtype };
chan from = [1] of { mtype };

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

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

what happens here?

Blocking receive

```
mtype = { A, B };
chan to = [1] of { mtype };
chan from = [1] of { mtype };

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

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

what happens here?

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

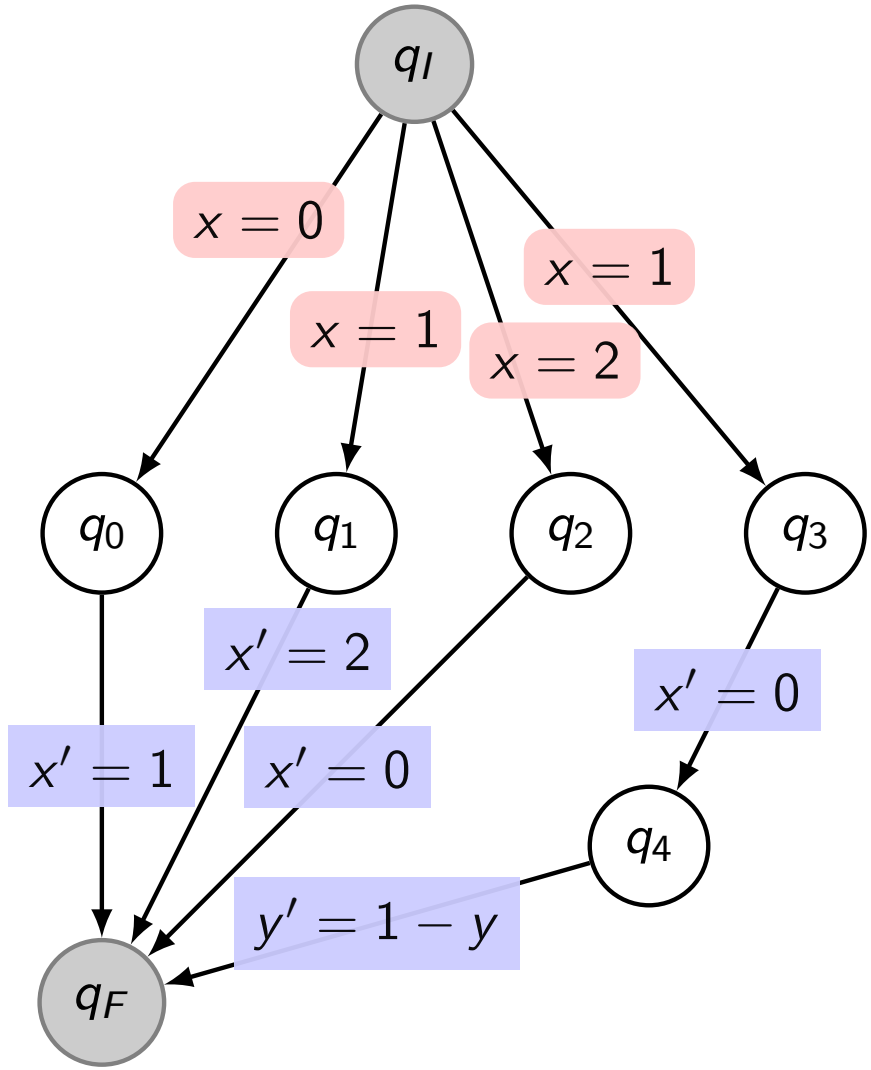
CFA: Intermediate representation

Intermediate representation of a loop iteration: 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;
}
```

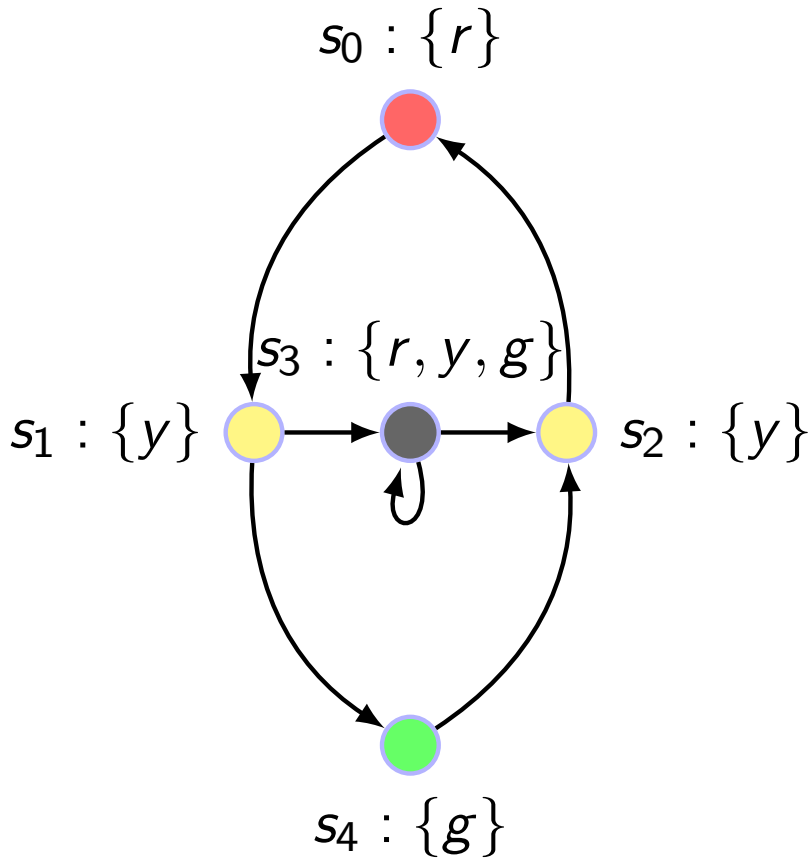


Kripke structures

The way to express semantics

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

- S is a *finite* 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.

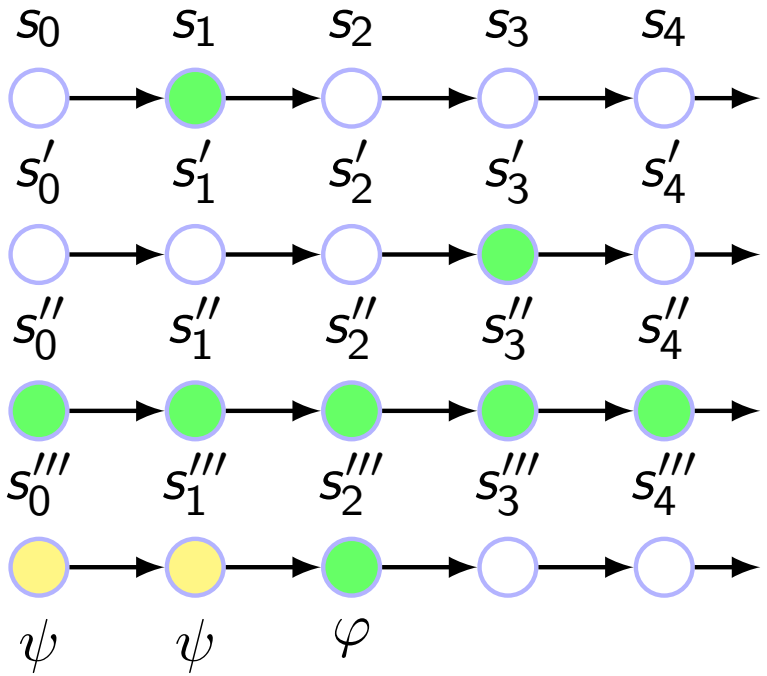


Linear Temporal Logic

The way to write specifications:

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 $\neg\varphi$.



Model Checking Problems

Consider P^N as interleaving of N processes of type P.

Finite-state MC

Input:

- a process template P (a Kripke structure),
- an LTL formula φ,
- the number of processes $N \geq 1$.

Problem: check, whether $P^N \models \varphi$.

Parameterized MC

Input:

- a process template P (a Kripke structure),
- an (indexed) LTL formula ϕ,

Problem: check, whether $\forall N \geq 1.\ P^N \models \phi(N)$.

Model Checking

Specification of ST87

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

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

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

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

Relay. If a correct process i sets accept_i to 1, then eventually all correct processes j set accept_j to 1.

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

Case Studies

Our running example **ST87** for

- Byzantine faults (Byz)
- omission faults (Omit)
- symmetric faults (Symm)
- clean crashes (Clean).

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

Case Studies (cont.): Larger Algorithms

more involved algorithms in the purely asynchronous setting:

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

Experimental Results at Glance

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

Experimental Results at Glance

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

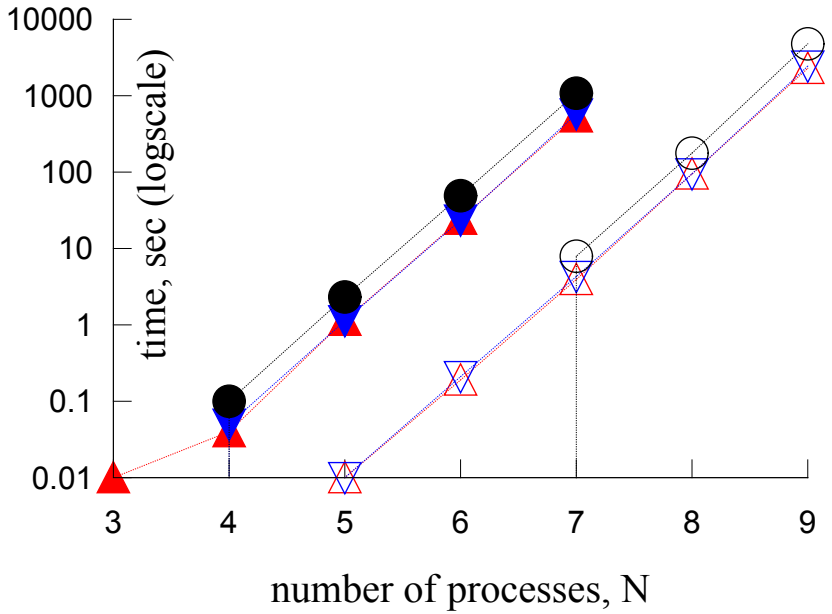
BT85	BYZ	$n = 5, t = 1, f = 1$	$n > 3t$	**R**
BT85	BYZ	$n = 5, t = 1, f = 2$	$n > 3t$	**R**
BT85	BYZ	$n = 5, t = 2, f = 2$	$n > 3t$	**R**
MRRR01	CRASH	$n = 3, t = 1, f = 1$	$n > 2t$	**V0**, **V1**, **A**, **T**
MRRR01	CRASH	$n = 3, t = 1, f = 2$	$n > 2t$	**V0**, **V1**, **A**, **T**
MA06	BYZ	$p = 4, a = 6, l = 4,$ $t = 1, f = 1$	$p > 3t,\ a > 5t,\ l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**
MA06	BYZ	$p = 4, a = 5, l = 4,$ $t = 1,\ f = 1$	$p > 3t,\ a > 5t,\ l > 3t$	**CS1**, **CS3**, **CL1**, **CL2**
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

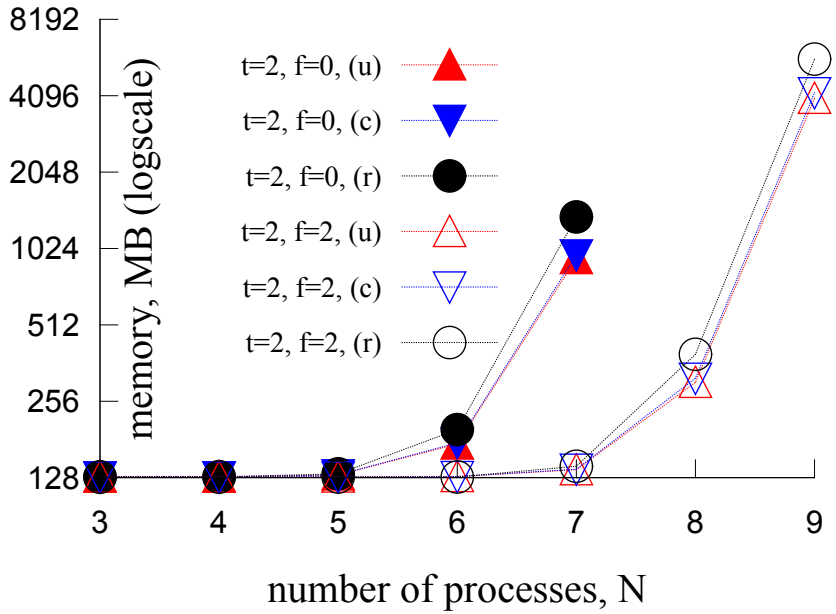
Algorithm	Fault	Parameters	Resilience	Properties	Time
ST87	Byz	$n = 7, t = 2, f = 2$	$n > 3t$	**U**, **C**, **R**	6 sec.
ST87	Byz	$n = 7, t = 3, f = 2$	$n > 3t$	**U**, **C**, **R**	5 sec.
ST87	Byz	$n = 7, t = 1, f = 2$	$n > 3t$	**U**, **C**, **R**	1 sec.
ST87	Omit	$n = 5, t = 2, f = 2$	$n > 2t$	**U**,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

Time (sec, logscale)



Memory (MB, logscale)



- We can check the properties up to nine processes ($f = 2$) and up to seven processes (no faults)
- We found counter-examples for the cases $n = 3t$ and $f > t$, where the resilience condition is violated.

(June 12, 2013: somebody wrote on Wikipedia that $n = 3t$ should work :-)

To achieve verification...

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.

To achieve verification...

We introduce efficient encoding in PROMELA

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

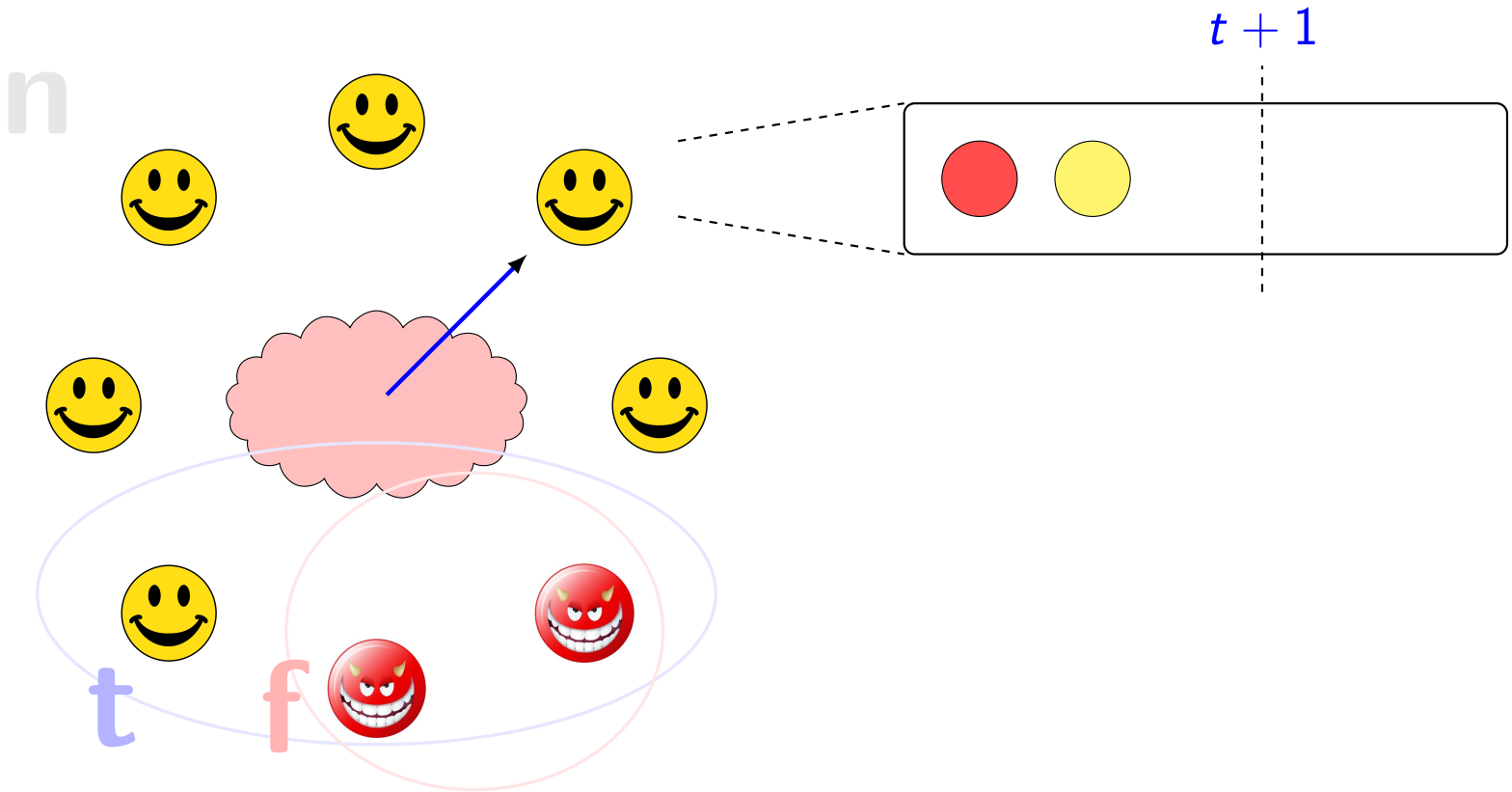
Find counterexamples for parameters known from the literature.

This proves adequacy of our modeling.

We can do that, because we look at:

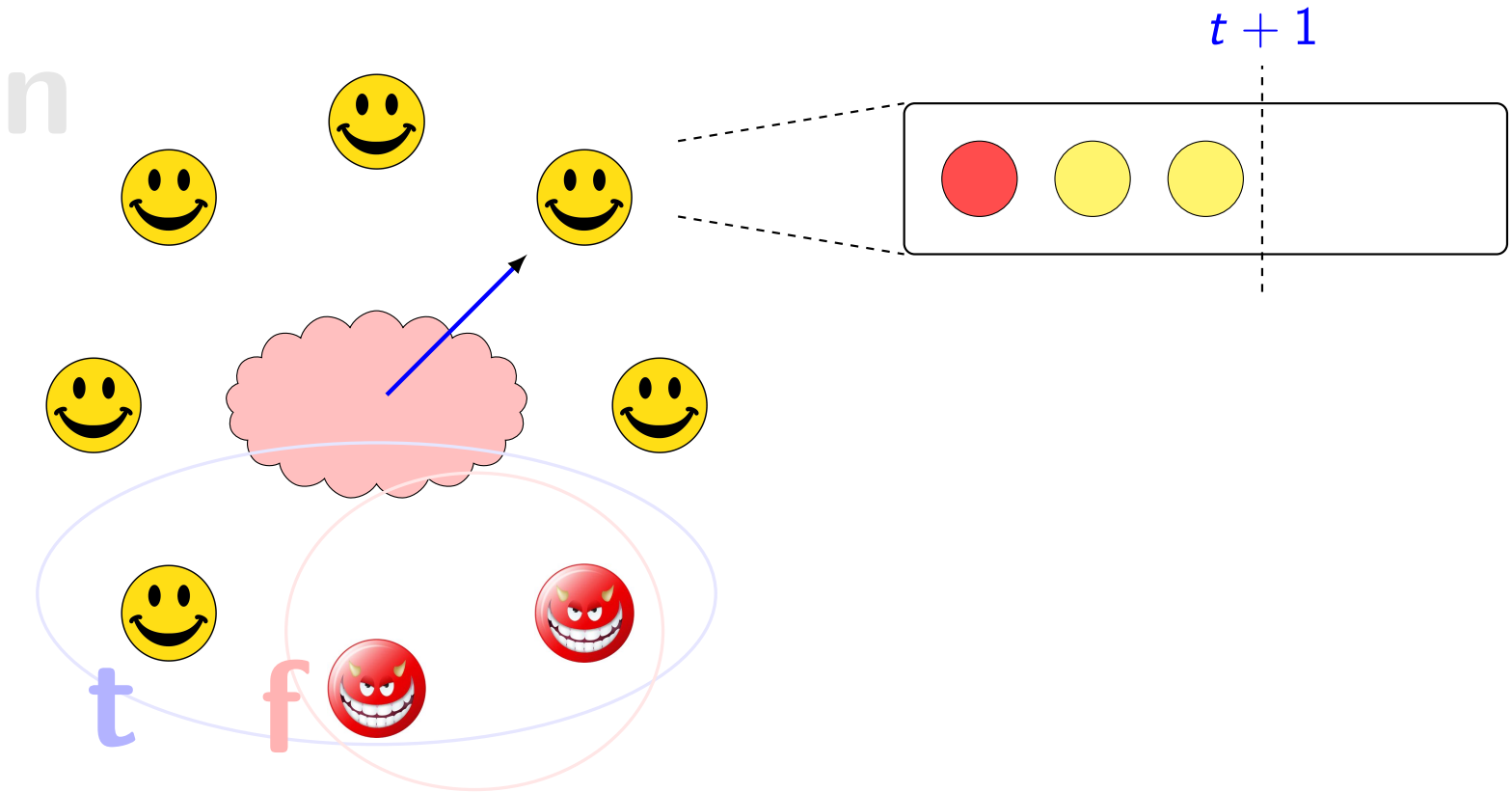
1. central feature of the algorithms
   (message counting);
2. specific message passing
   (we do not need to know who sent but how many of them sent messages);
3. the way faults affect messages
   (again, counting messages).

Counting Argument in Threshold-Guarded Algorithms



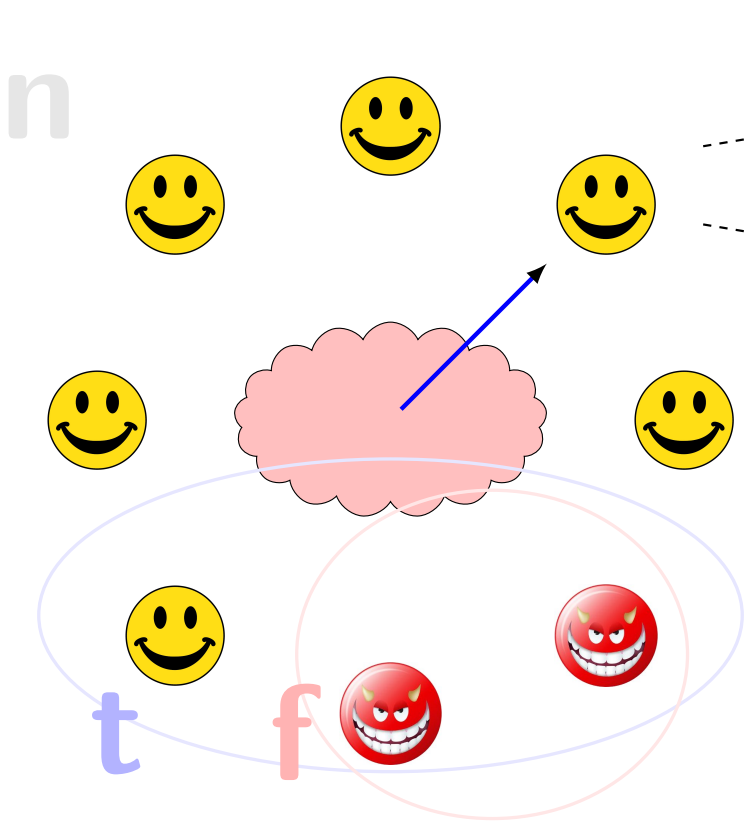
Correct processes count distinct incoming messages

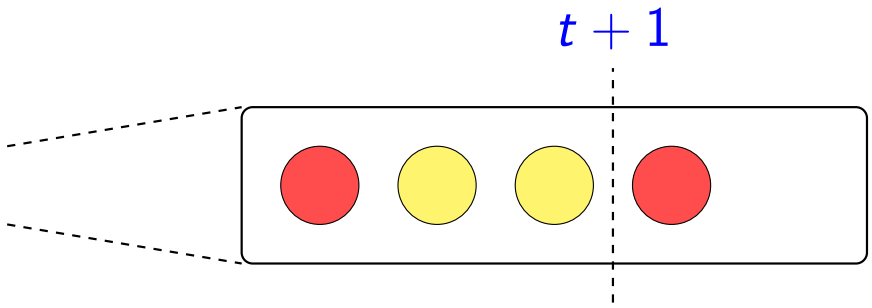
Counting Argument in Threshold-Guarded Algorithms



Correct processes count distinct incoming messages

Counting Argument in Threshold-Guarded Algorithms





at least one non-faulty sent the message

Correct processes count distinct incoming messages

What shall we model?

As we have pseudo-code on input,

we have to decide on how to encode:

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

Message Passing

Classic reliable asynchronous message passing as in [FLP85] has the characteristics:

- non-blocking communication,
- if a message is in the buffer, it may be received later,
- every sent message is eventually received

Message Passing using Promela channels

A straightforward encoding using message channels:

```
mtype = { ECHO };
chan p2p[NxN] = [1] of { mtype };
bit  rx[NxN];
```

Sending a message to all processes:

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

Message Passing (cont.)

Receiving and counting messages from distinct processes

(no faults yet):

```
int nrcvd = 0;
...
i = 0;
do :: (i < N) && nempty(p2p[i * N + _pid]) ->
      p2p[i * N + _pid]?ECHO;
      if
        :: !rx[i * N + _pid] ->
           rx[i * N + _pid] = 1;
           nrcvd++; break;
        :: rx[i * N + _pid];
      fi; i++;
   :: (i < N) -> i++;
   :: i == N -> break;
od
```

Shared Variables

Keeping the number of messages sent by (correct) processes:

```
int nsnt;
```

Sending a message to all:

```
nsnt++;
```

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

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

Reliable communication as a fairness property:

$$\mathbf{G}\,\mathbf{F}\,\neg[\exists i.nrcvd_i < nsnt]$$

Byzantine Processes

Pease, Shostak, Lamport 1980: "...a bad processor might report one value to a given processor and another value to some other processors"

Explicit modeling:

```
active[F] proctype Byz() {
step:
  atomic {
    i = 0;
    do
      :: i < N -> sendTo(i); i++;
      :: i < N -> i++; /* some */
      :: i == N -> break;
    od
  };
  goto step;
}
```

Injecting Faults into Message Counters

Creating only $n - f$ correct processes.

But the correct processes may receive up to f messages from the f faulty processes:

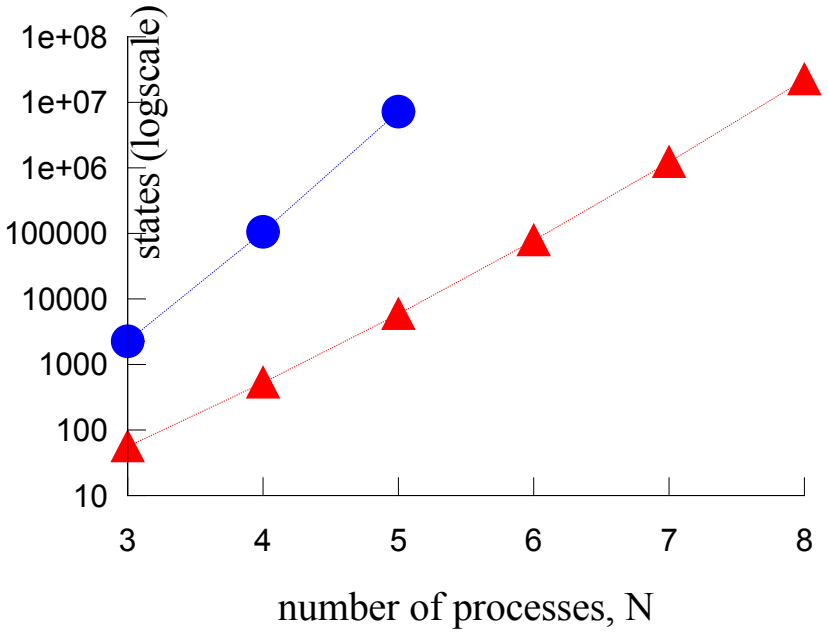
```
if
  :: (next_nrcvd < nsnt + f) ->
     next_nrcvd = nrcvd + 1; /* one more message */
  :: next_nrcvd = nrcvd;     /* or nothing */
fi;
```

Still, we guarantee to receive up to *nsnt* messages:

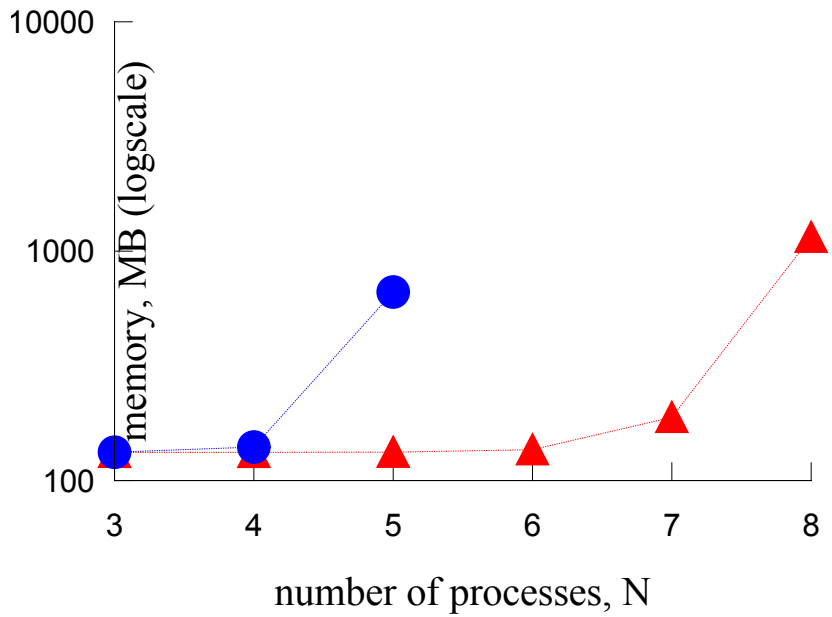
$$\mathbf{G}\,\mathbf{F}\,\neg[\exists i.nrcvd_i < nsnt]$$

Experiments: Channels + Explicit Faults vs. Shared Memory + Fault Injection

States (logscale)



Memory (MB, logscale, limit of 12 GB)



- Channels + explicit Byzantine processes (blue)
- vs. shared variables + fault injection (red)
- in the presence of one Byzantine faulty process

Our Lessons about Modeling Fault-Tolerant DAs

They have their own specific features and assumptions.

It takes some efforts to adequately formalize them.

> *It is essential to have the experts in these algorithms in the loop.*

Standard model checkers are not well tuned up to FTDAs.

Straightforward modeling does not work well.

> *Thinking in terms of the parameterized model checking problem gives interesting insights.*

Thank you!

decidability?

Decidability and Undecidability in Retrospect

- Apt & Kozen 1986

  one process $P(n)$ with a parameterized loop bound n,
  $P(n)$ simulates n steps of a TM, non-halting is undecidable,
  so does PMC for n non-communicating processes $P^n(n)$.

Decidability and Undecidability in Retrospect

- Apt & Kozen 1986

  one process $P(n)$ with a parameterized loop bound n,
  $P(n)$ simulates n steps of a TM, non-halting is undecidable,
  so does PMC for n non-communicating processes $P^n(n)$.

- Suzuki 1988

  one fixed process P independent of n,
  P^n is a ring, processes communicate by rendezvous,
  P^n simulates n steps of a TM.

Decidability and Undecidability in Retrospect

- Apt & Kozen 1986

  one process $P(n)$ with a parameterized loop bound n,
  $P(n)$ simulates n steps of a TM, non-halting is undecidable,
  so does PMC for n non-communicating processes $P^n(n)$.

- Suzuki 1988

  one fixed process P independent of n,
  P^n is a ring, processes communicate by rendezvous,
  P^n simulates n steps of a TM.

- German & Sistla 1992

  stars $C \parallel U^n$,
  C and U communicate by rendezvous,
  $\forall n \geq 1.\ (C \parallel U^n) \models f$ simulates a 2CM.

Decidability and Undecidability in Retrospect

- Apt & Kozen 1986
  one process $P(n)$ with a parameterized loop bound n,
  $P(n)$ simulates n steps of a TM, non-halting is undecidable,
  so does PMC for n non-communicating processes $P^n(n)$.
- Suzuki 1988
  one fixed process P independent of n,
  P^n is a ring, processes communicate by rendezvous,
  P^n simulates n steps of a TM.
- German & Sistla 1992
  stars $C \parallel U^n$,
  C and U communicate by rendezvous,
  $\forall n \geq 1.\ (C \parallel U^n) \models f$ simulates a 2CM.
- Results for Petri Nets, e.g., Esparza 1997
  checking a linear μ-calculus formula against a Petri net simulates a 2CM.

Two Counter Machines

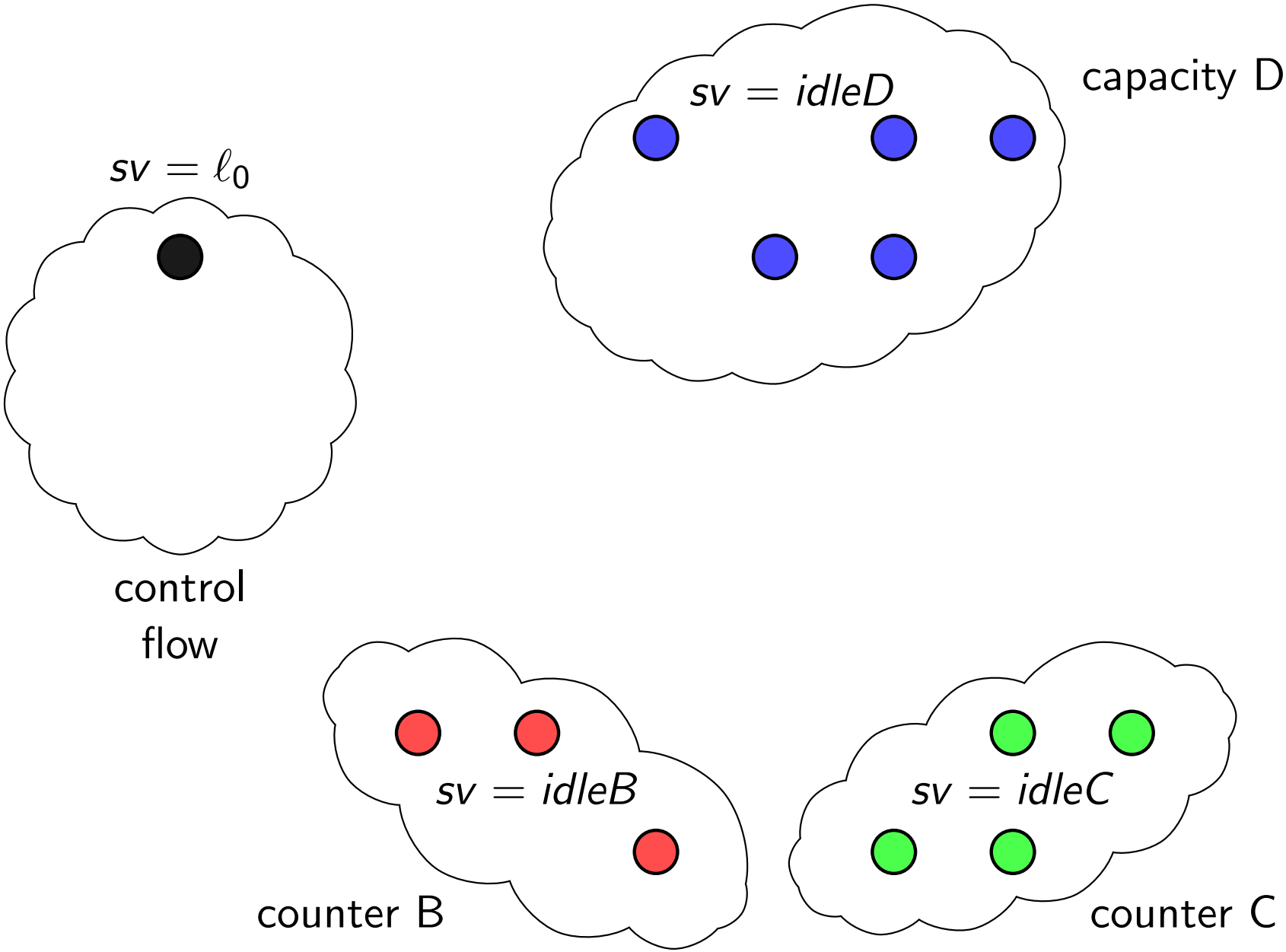
A machine composed of the following commands

$$\begin{aligned} \ell_i &: \textbf{inc } \mathcal{C}(\ell_i);\ \textbf{goto } \ell_j \\ \ell_i &: \textbf{if } \mathcal{C}(\ell_i) = 0 \textbf{ then goto } \ell_j \\ &\quad\ \textbf{else dec } \mathcal{C}(\ell_i);\ \textbf{goto } \ell_k \\ \ell_m &: \textbf{halt} \end{aligned}$$

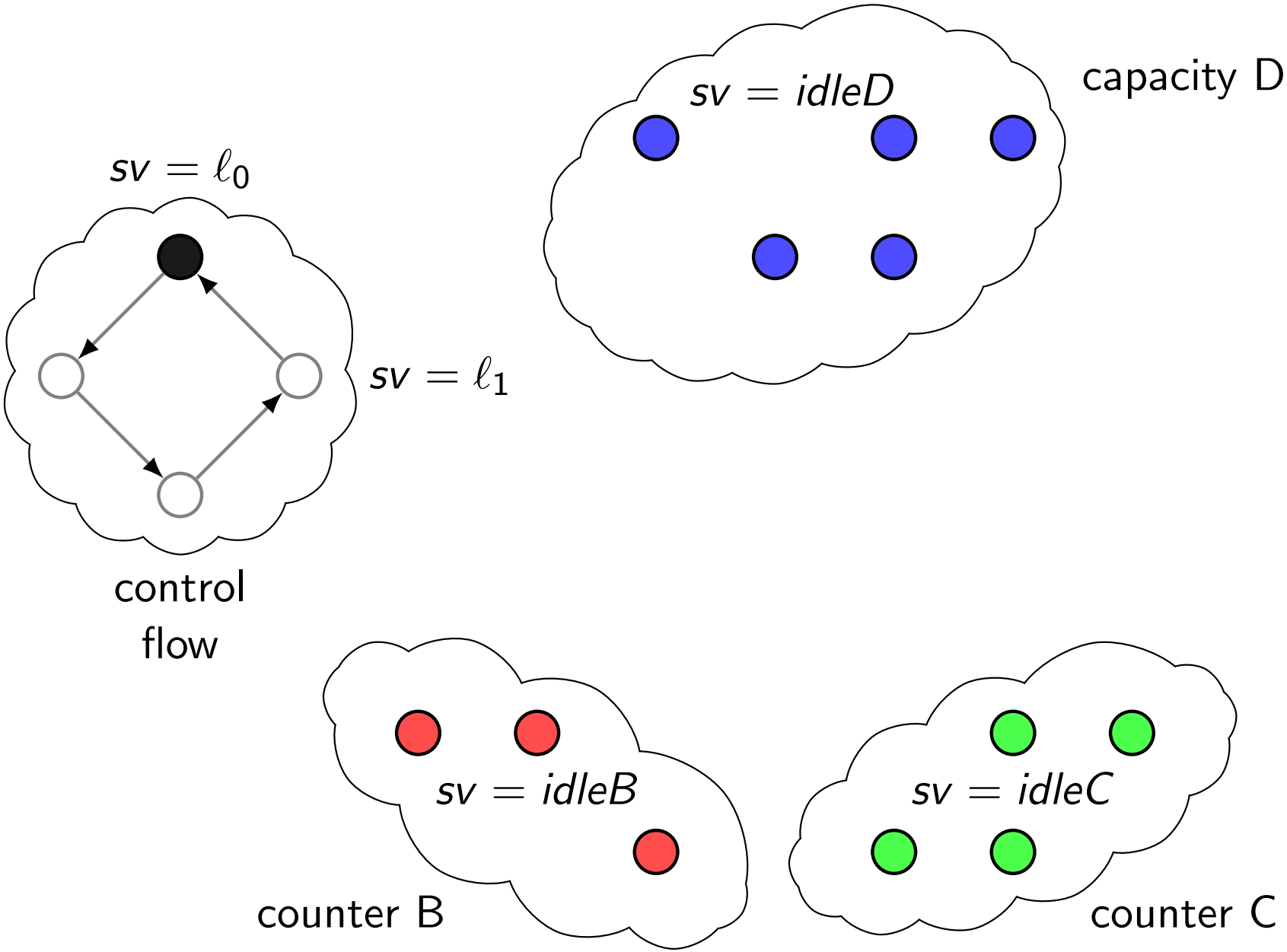
over two counters $\mathcal{C}(\ell_i) \in \{B, C\}$.

The halting (as well as non-halting) problem is well-known to be undecidable [Minsky1967].

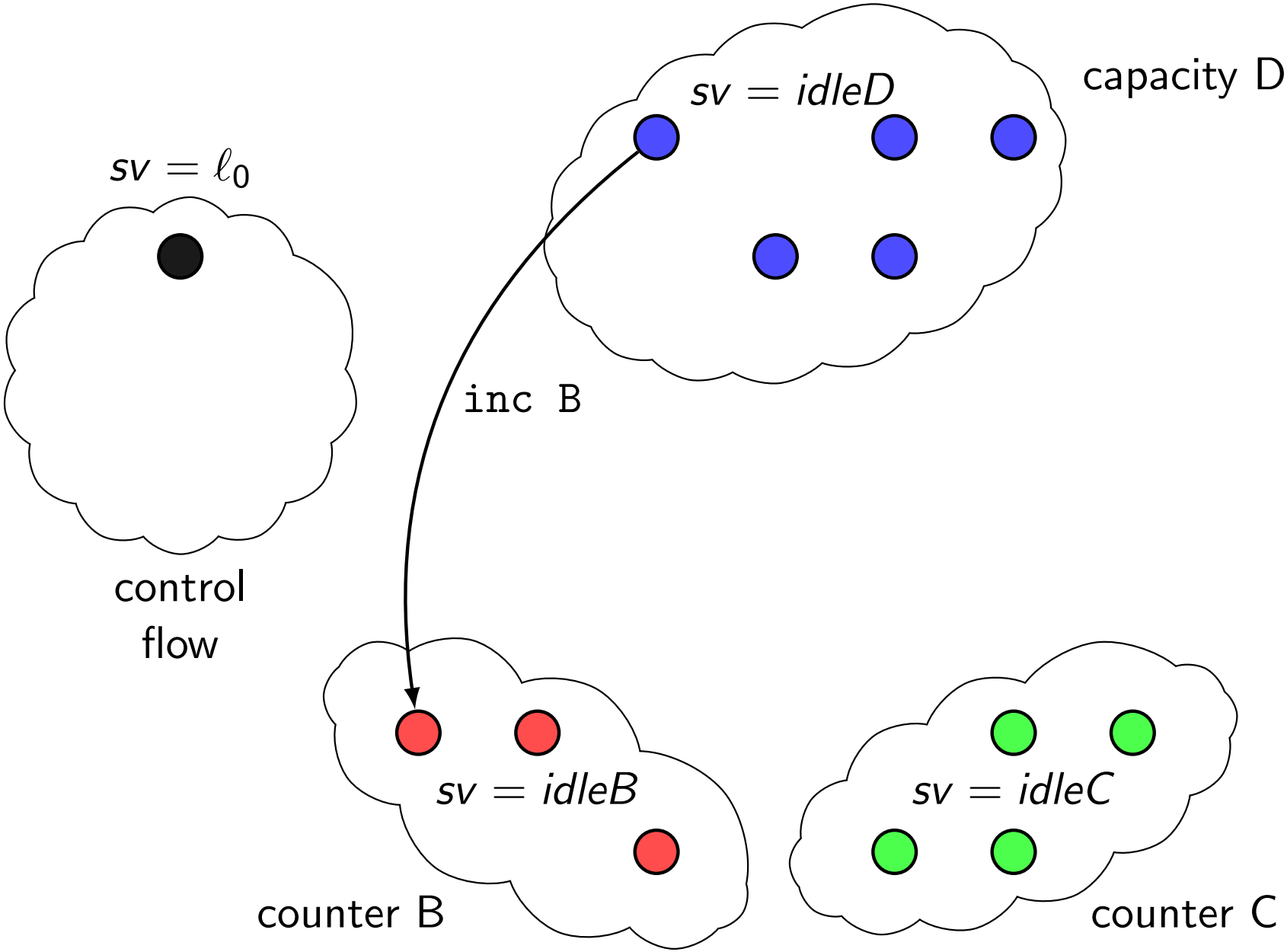
Simulating 2CM



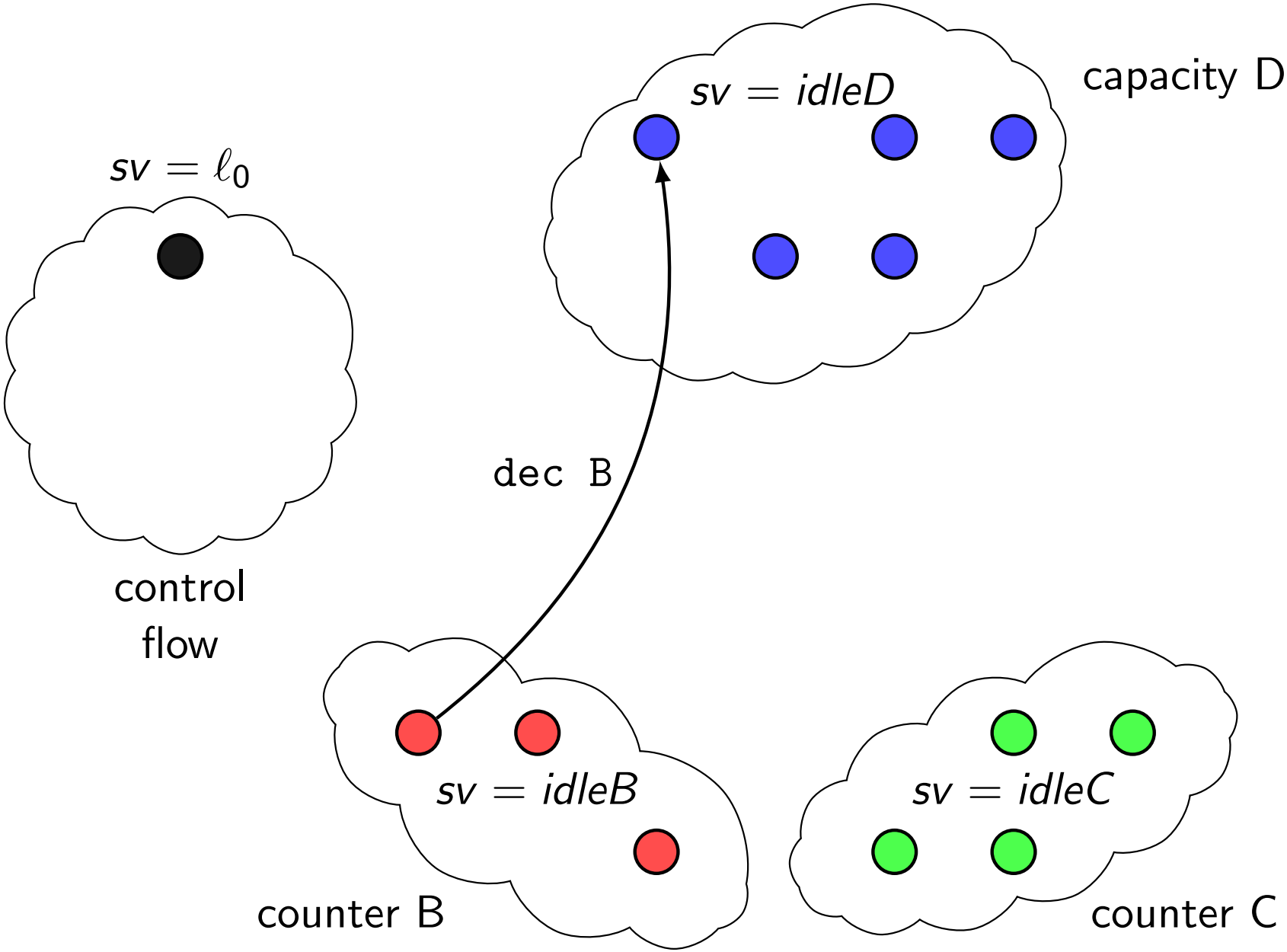
Simulating 2CM



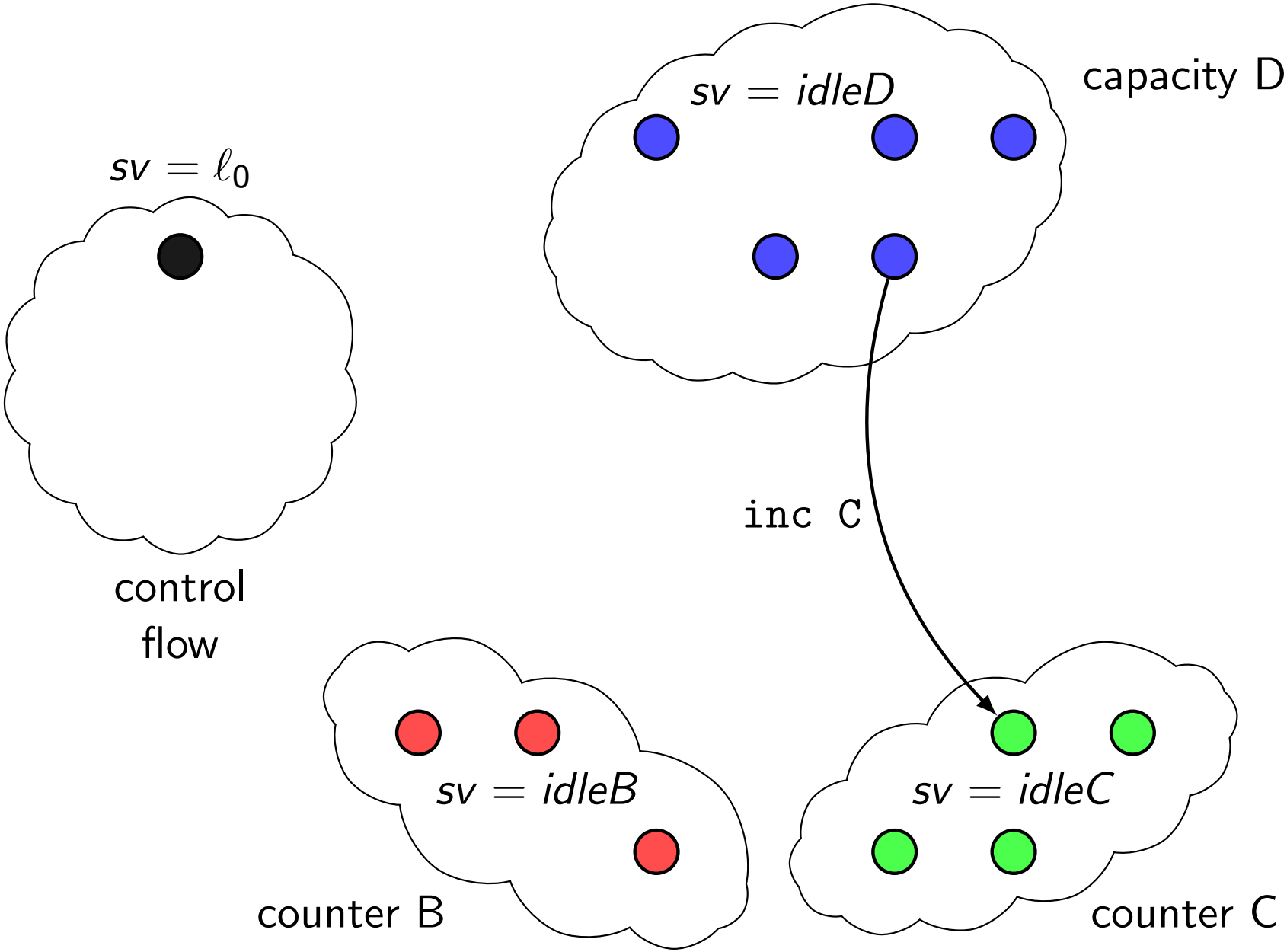
Simulating 2CM



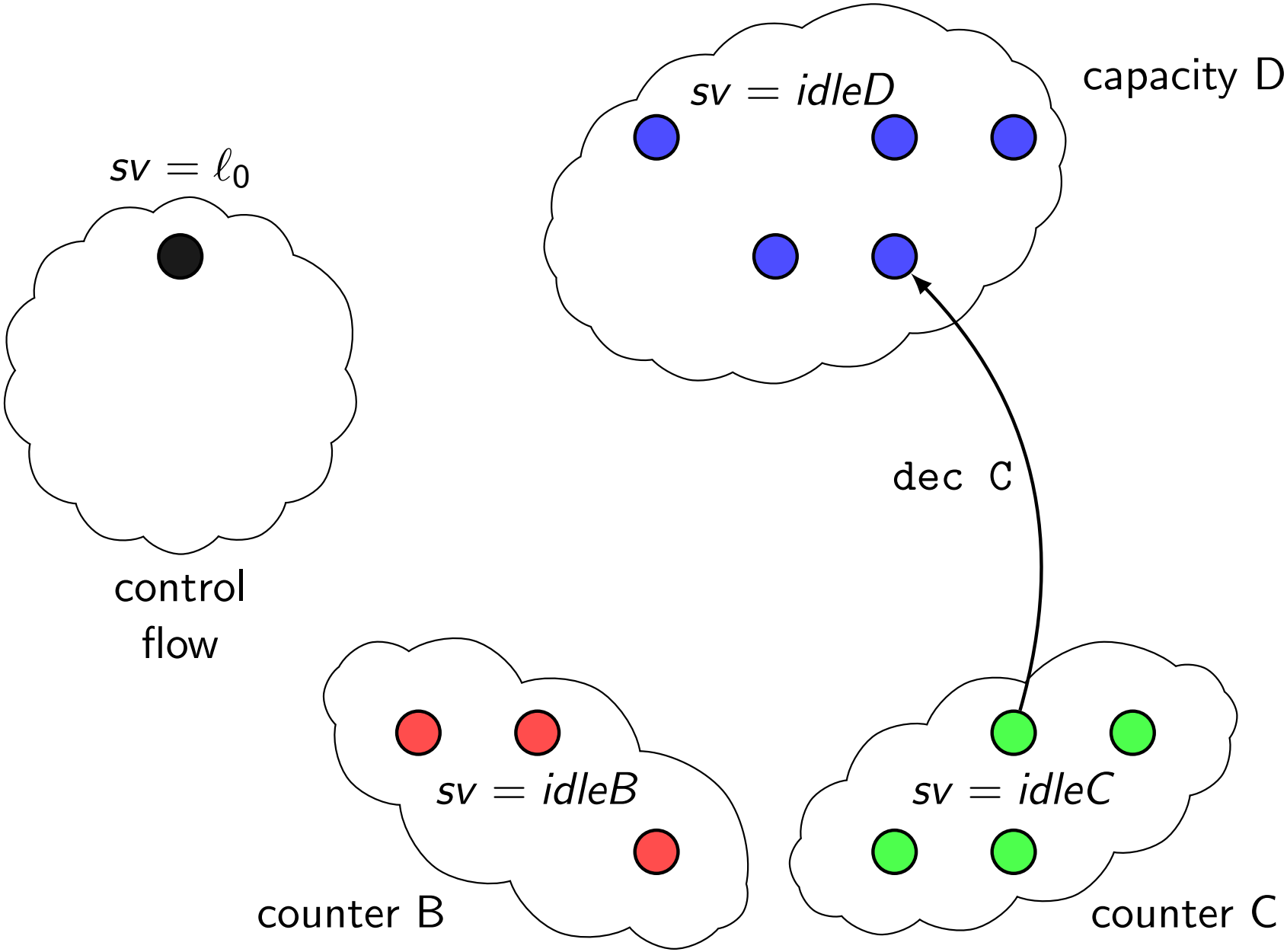
Simulating 2CM



Simulating 2CM



Simulating 2CM



Enforcing Test for Zero & Handshake via Specifications

Note that our atomic propositions have the form $[\exists i.\ f(i)]$ and $[\forall i.\ f(i)]$.

Thus, we can enforce test for zero as follows:

$$[\exists k.sv_k = (\ell_i, \ell_j, SynC)] \rightarrow \neg[\exists k.sv_k = (\mathcal{C}(\ell_i), \mathcal{C}(\ell_i), IdID)]$$

Enforcing Test for Zero & Handshake via Specifications

Note that our atomic propositions have the form $[\exists i.\ f(i)]$ and $[\forall i.\ f(i)]$.

Thus, we can enforce test for zero as follows:

$$[\exists k.sv_k = (\ell_i, \ell_j, SynC)] \rightarrow \neg[\exists k.sv_k = (\mathcal{C}(\ell_i), \mathcal{C}(\ell_i), IdlD)]$$

Handshake is more complicated (it encodes 7 steps):

$$\begin{aligned}
[\exists k.sv_k = (x, y, SynD)] \rightarrow & [\exists k.sv_k = (v, w, SynC)] \wedge \\
[\exists k.sv_k = (x, y, AckD)] \rightarrow & \neg[\exists k.sv_k = (x, y, SynD)] \wedge \\
[\exists k.sv_k = (x, y, AckD)] \rightarrow & [\exists k.sv_k = (v, w, AckC)] \wedge \\
[\exists k.sv_k = (w, w, IdlC] \rightarrow & (\neg[\exists k.sv_k = (x, y, SynD)] \wedge \\
& \neg[\exists k.sv_k = (x, y, AckD)])
\end{aligned}$$

Undecidability for Threshold-based FTDAs

Theorem

Let $\mathcal{M}$ be a two counter machine, and $(RC, size)$ be a natural pair of resilience condition and system size function. One can efficiently construct a non-communicating CFA $A(\mathcal{M})$ and an LTL$_{-X}$ property $\varphi_{\text{nonhalt}}(\mathcal{M})$ such that the following two statements are equivalent:

- *$\mathcal{M}$ does not halt.*
- $\forall \mathbf{p} \in \mathbf{P}_{RC}$, $M(\mathbf{p}) \models_{\emptyset} \varphi_{\text{nonhalt}}(\mathcal{M})$.

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,

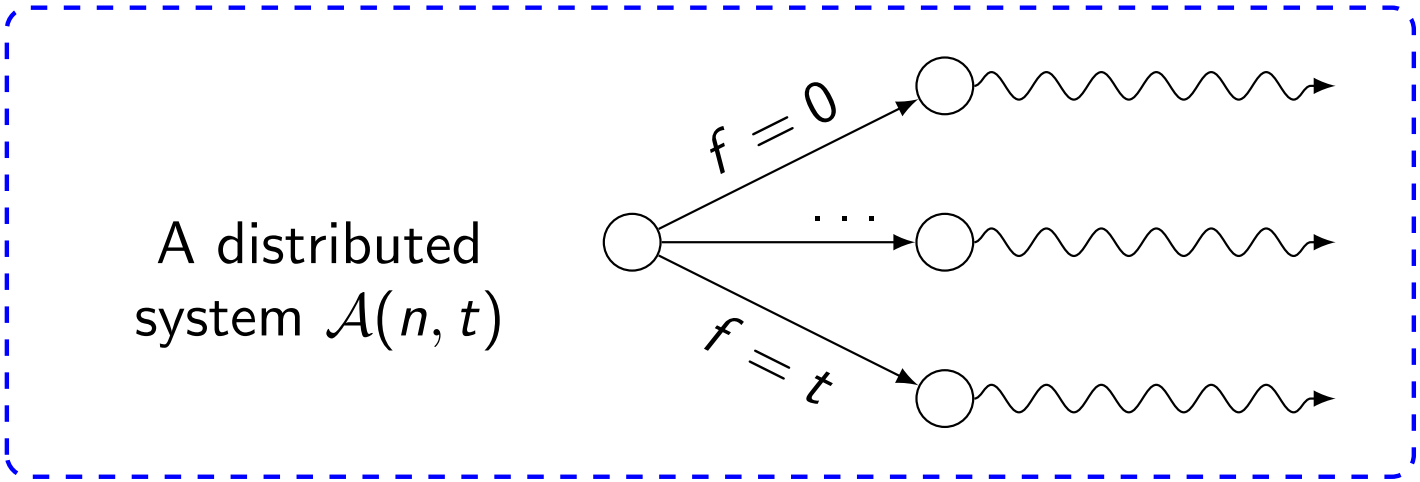
- Fix n and t with $n > 3t$,
- show that every execution of $\mathcal{A}(n, t)$ satisfies φ_U, φ_C, φ_R.

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).

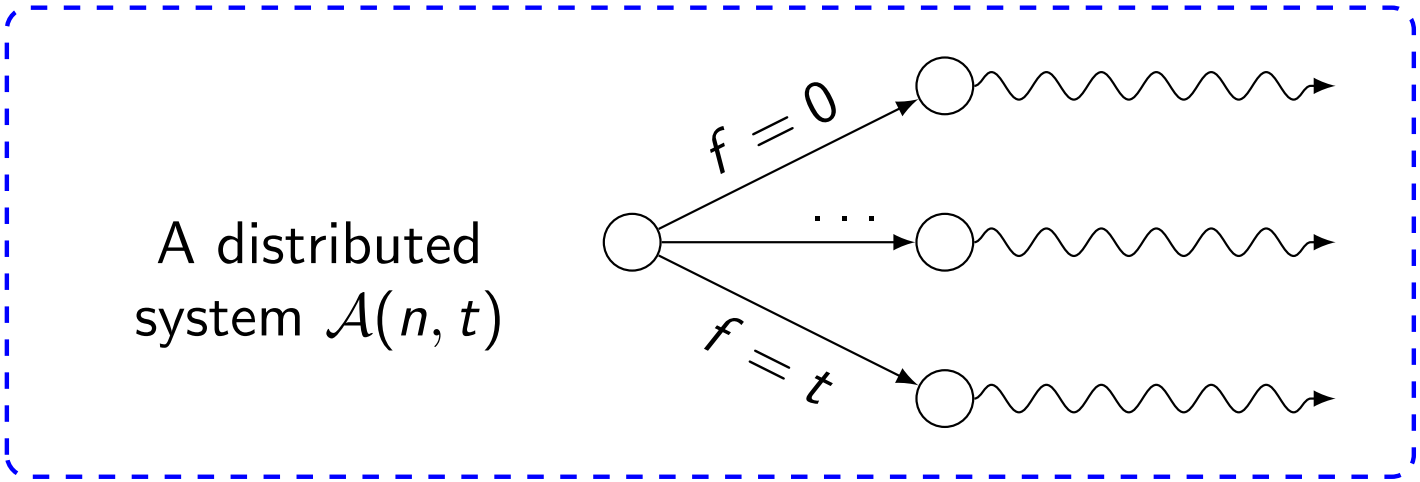


Verification Problem as in Distributed Computing

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 $f > t$?

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? 

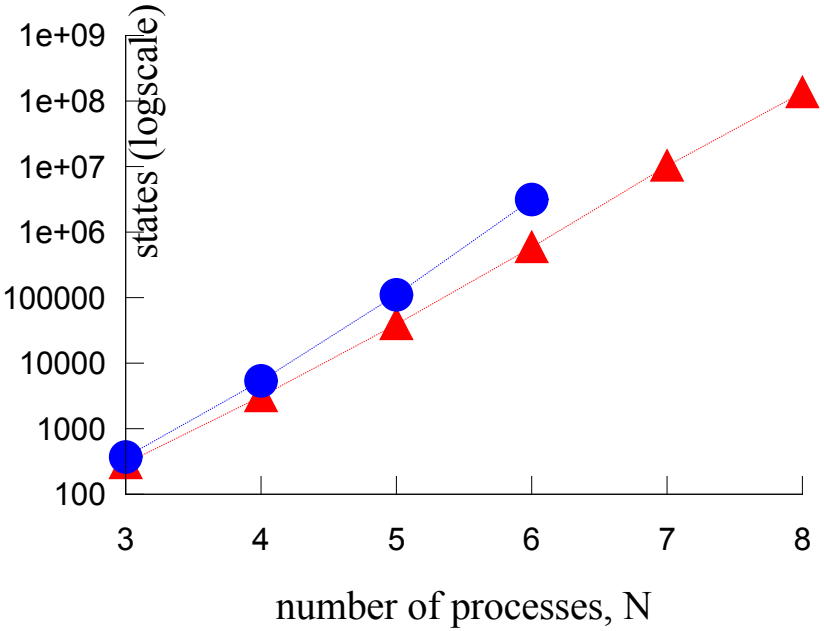
Fault-Tolerant Algorithms: n processes, at most t are Byzantine

- Threshold Guard
  `if received` m `from` $n - t$ `processes then ...`
- (the processes cannot refer to **f**!)

Experiments: Channels vs. Shared Variables

enumerating reachable states in SPIN with POR and state compression

States (logscale)



Memory (MB, logscale, limit of 12 GB)