Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction

Annu John Igor Konnov Ulrich Schmid Helmut Veith Josef Widder

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

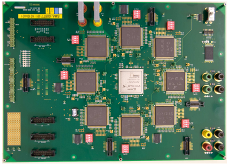
Why fault-tolerant (FT) distributed algorithms

faults not in the control of system designer

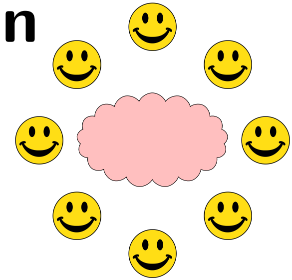
- bit-flips in memory
- power outage
- disconnection from the network
- intruders take control over some computers

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"

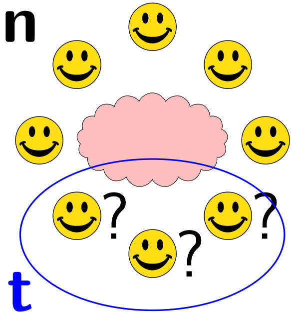


Fault-tolerant distributed algorithms



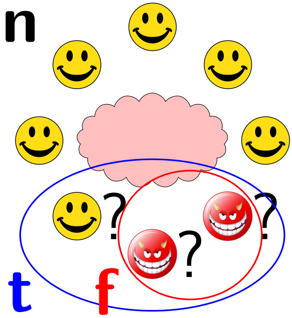
- n processes communicate by messages

Fault-tolerant distributed algorithms



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

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., $n > 3t \wedge t \geq f \geq 0$
- 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)

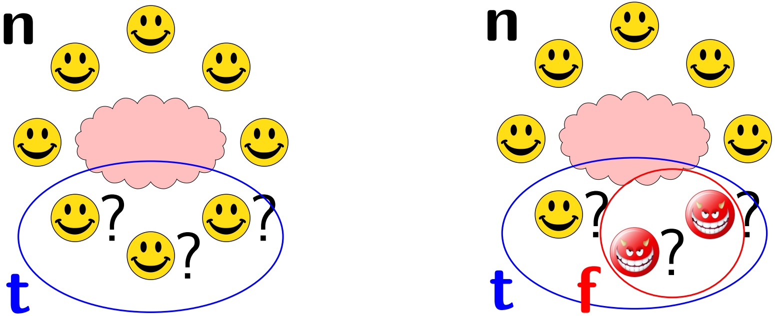
Distributed algorithms perspective:

- "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:

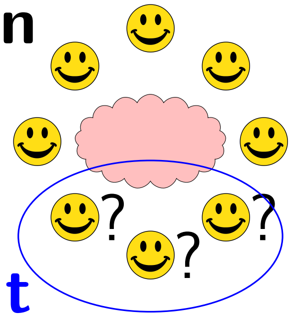
- given a distributed algorithm and spec. φ
- show for all n, t, and f satisfying $n > 3t \wedge t \geq f \geq 0$
  $M(n, t, f) \models \varphi$
- every $M(n, t, f)$ is a system of $n - f$ correct processes

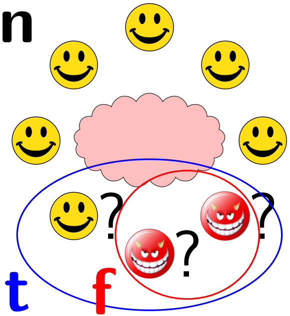


Model checking problem for fault-tolerant DA algorithms

Parameterized model checking problem:

- given a distributed algorithm and spec. φ
- 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, accept_j remains 0 forever.

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

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.

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

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

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

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.

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

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.

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

Liveness

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

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

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 ...`

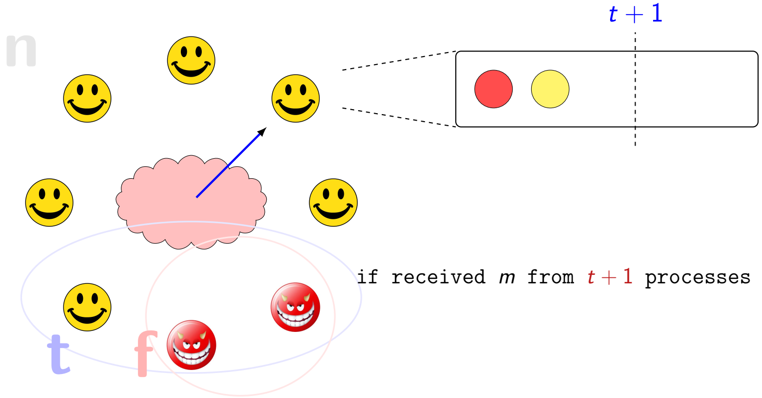
These guards allow one to treat the processes in a parameterized way

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**!)

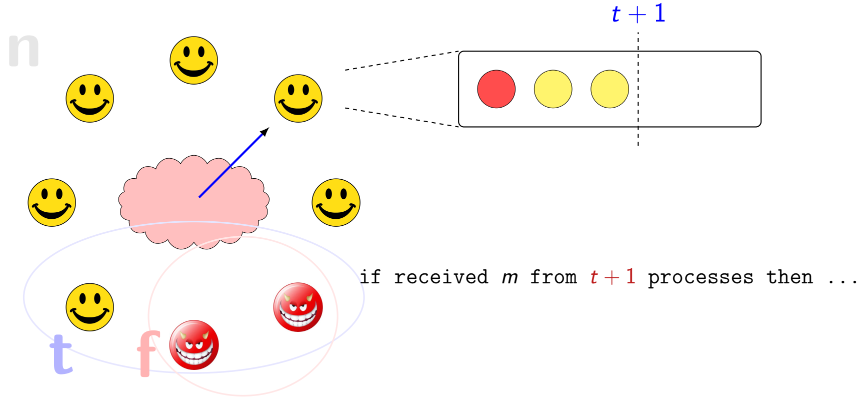
Counting argument in threshold-guarded algorithms



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

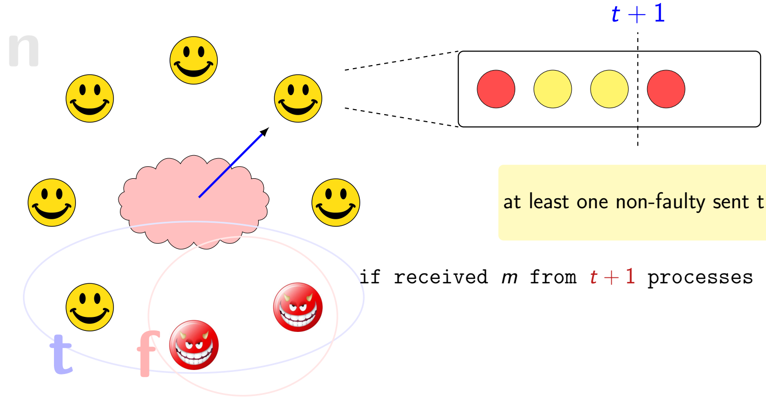
Correct processes count distinct incoming messages

Counting argument in threshold-guarded algorithms



Correct processes count distinct incoming messages

Counting argument in threshold-guarded algorithms



at least one non-faulty sent the message

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

Correct processes count distinct incoming messages

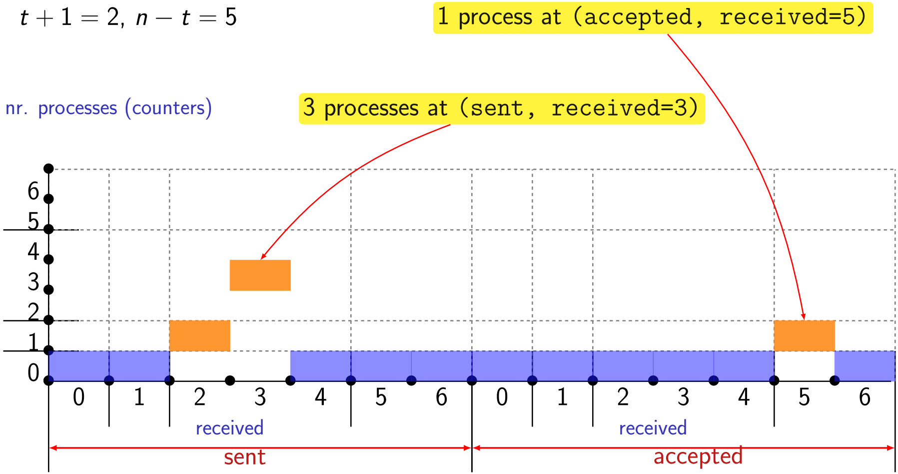
our abstraction

at a glance

Data + counter abstraction over parametric intervals

$n = 6,\ t = 1,\ f = 1$

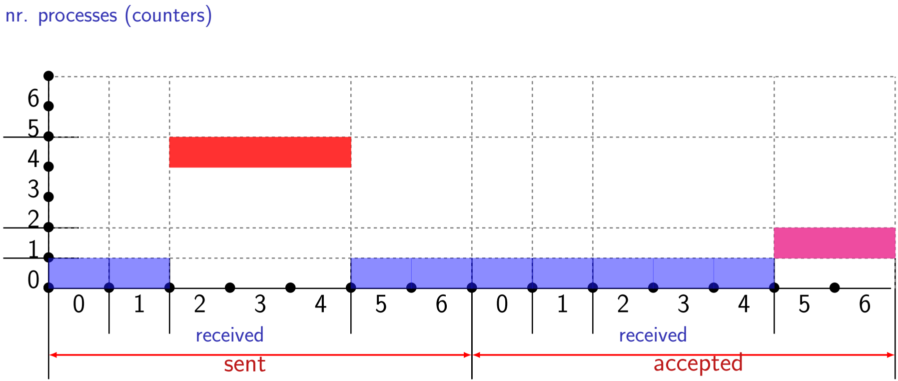
$t + 1 = 2,\ n - t = 5$



Data + counter abstraction over parametric intervals

$n = 6,\ t = 1,\ f = 1$

$t + 1 = 2,\ n - t = 5$

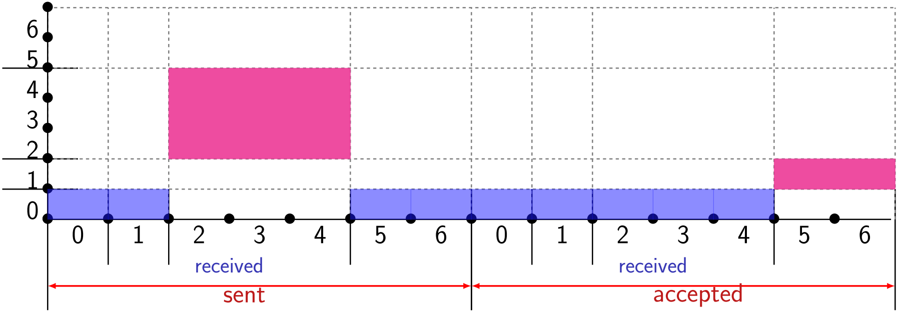


Data + counter abstraction over parametric intervals

$n = 6,\ t = 1,\ f = 1$

$t + 1 = 2,\ n - t = 5$

nr. processes (counters)



Data + counter abstraction over parametric intervals

~~$n = 6$, $t = 1$, $f = 1$~~

$n > 3 \cdot t \wedge t \geq f$

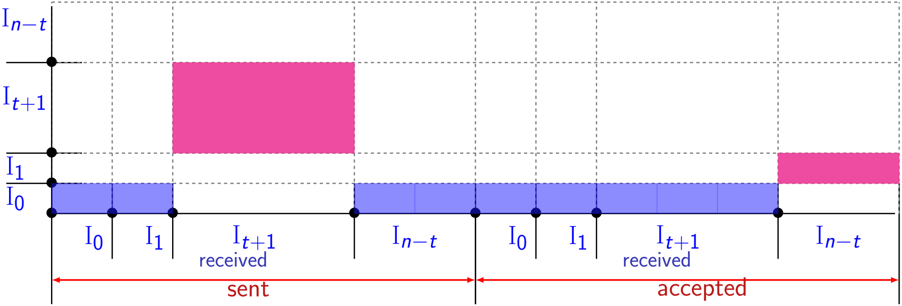
Parametric intervals:

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

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

$I_{n-t} = [n-t, \infty)$

nr. processes (counters)



Data + counter abstraction over parametric intervals

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

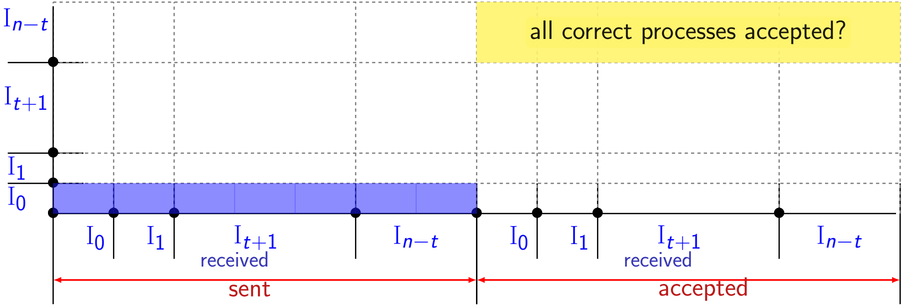
Parametric intervals:

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

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

$I_{n-t} = [n-t, \infty)$

nr. processes (counters)



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., $K: \quad C \mapsto \mathbf{0}, \quad T \mapsto \mathbf{1}, \quad N \mapsto$ **"many"**.
- perfectly reflects mutual exclusion properties
  e.g., $\mathbf{G}\,(K(C) = \mathbf{0} \vee K(C) = \mathbf{1})$.

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., $K: \quad C \mapsto \mathbf{0}, \quad T \mapsto \mathbf{1}, \quad N \mapsto$ **"many"**.
- perfectly reflects mutual exclusion properties
  e.g., $\mathbf{G}\,(K(C) = \mathbf{0} \vee K(C) = \mathbf{1})$.

Our parametric data + counter abstraction:

- unboundendly 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

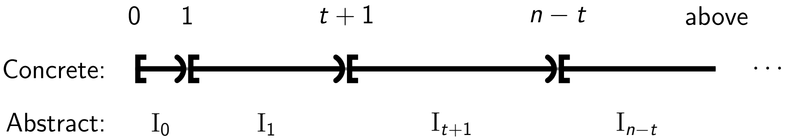
Technical challenges

How to do data abstraction?

How to do counter abstraction?

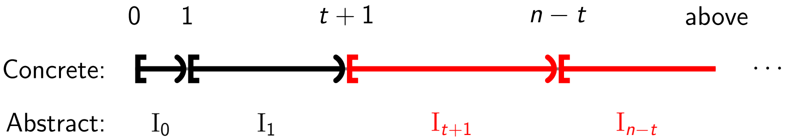
How to refine spurious counter-examples introduced by the abstraction?

Abstract operations



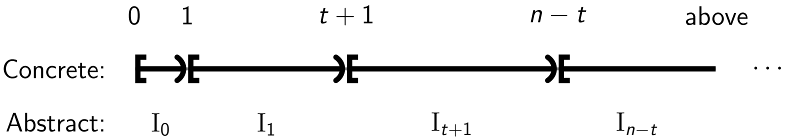
Concrete $t + 1 \leq x$

Abstract operations



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

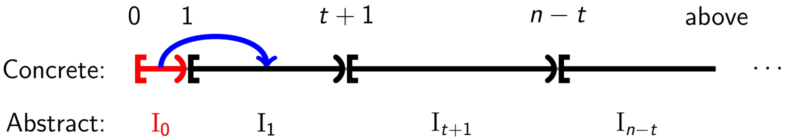
Abstract operations



Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

Concrete $x' = x + 1$,

Abstract operations

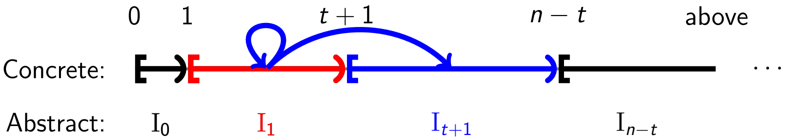


Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \vee x = I_{n-t}$.

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

$$x = I_0 \quad \wedge \; x' = I_1 \; \dots$$

Abstract operations

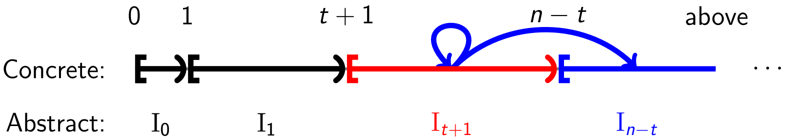


Concrete $t + 1 \leq x$ is abstracted as $x = I_{t+1} \vee x = I_{n-t}$.

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

$$\begin{aligned} & x = I_0 \quad \wedge \; x' = I_1 \\ \vee & x = I_1 \quad \wedge (x' = I_1 \quad \vee x' = I_{t+1}) \; \ldots \end{aligned}$$

Abstract operations

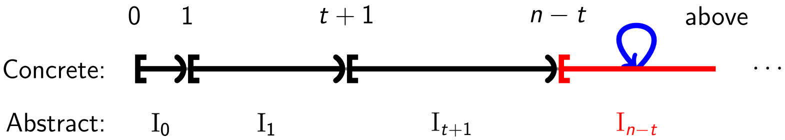


Concrete $t + 1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

$$\begin{aligned} & x = \mathrm{I}_0 \quad \wedge \ x' = \mathrm{I}_1 \\ \vee & x = \mathrm{I}_1 \quad \wedge (x' = \mathrm{I}_1 \quad \vee x' = \mathrm{I}_{t+1}) \\ \vee & x = \mathrm{I}_{t+1} \wedge (x' = \mathrm{I}_{t+1} \vee x' = \mathrm{I}_{n-t}) \ \ldots \end{aligned}$$

Abstract operations



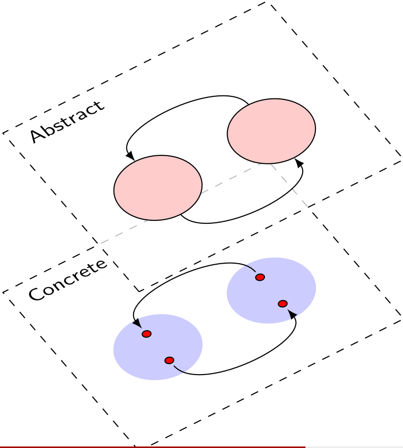
Concrete $t + 1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

$$
\begin{aligned}
& x = \mathrm{I}_0 \quad \wedge\ x' = \mathrm{I}_1 \\
\vee & x = \mathrm{I}_1 \quad \wedge (x' = \mathrm{I}_1 \quad \vee x' = \mathrm{I}_{t+1}) \\
\vee & x = \mathrm{I}_{t+1} \wedge (x' = \mathrm{I}_{t+1} \vee x' = \mathrm{I}_{n-t}) \\
\vee & x = \mathrm{I}_{n-t} \wedge\ x' = \mathrm{I}_{n-t}
\end{aligned}
$$

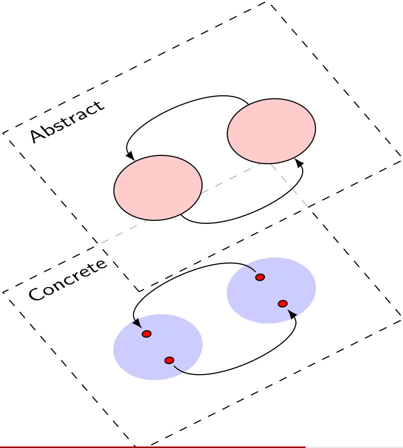
Parametric abst. refinement — uniformly spurious paths

Classical CEGAR:

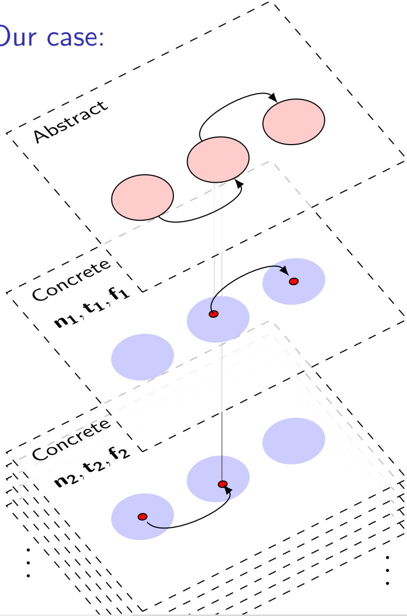


Parametric abst. refinement — uniformly spurious paths

Classical CEGAR:

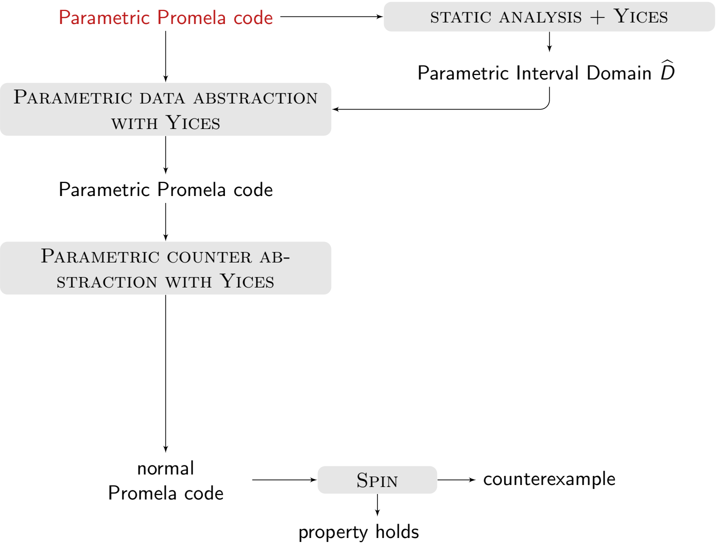


Our case:

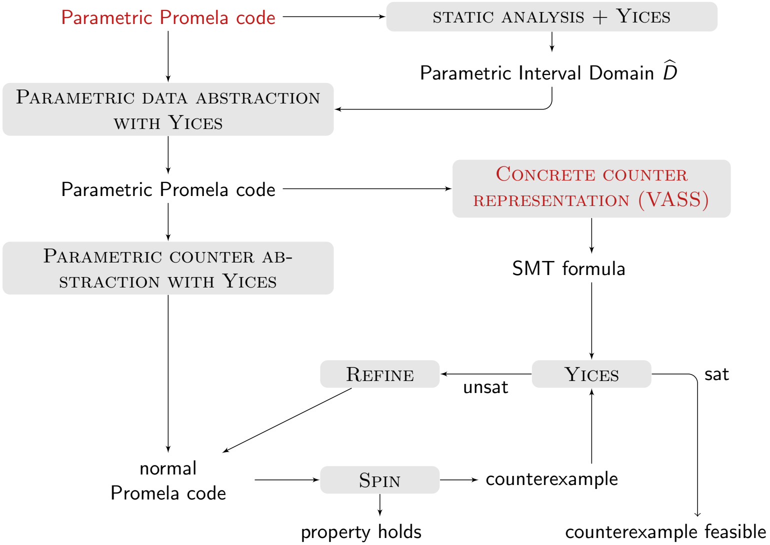


the implementation

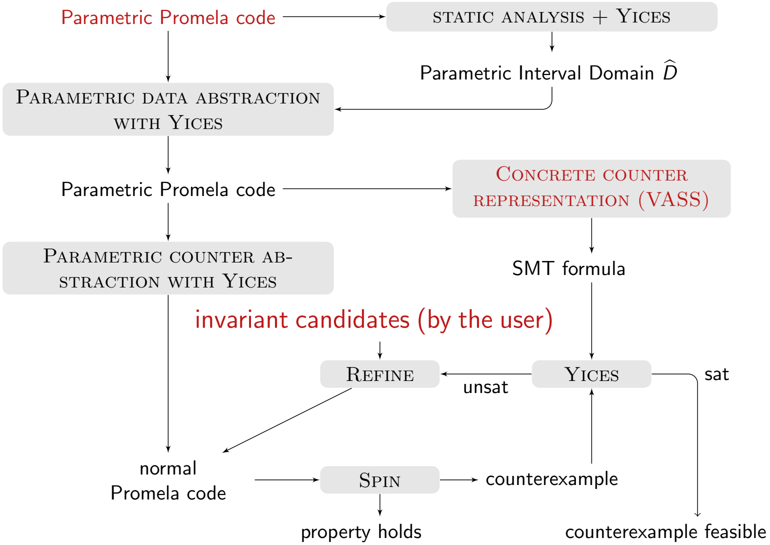
Tool Chain: ByMC



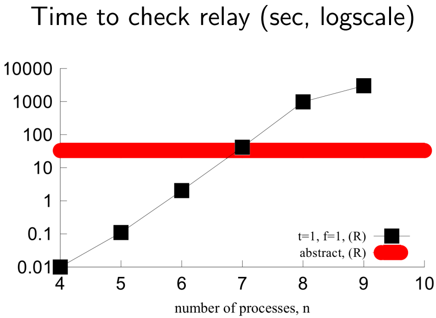
Tool Chain: ByMC

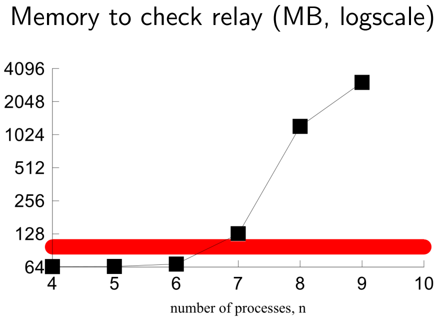


Tool Chain: ByMC



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.

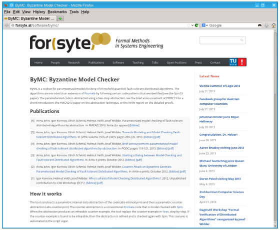
Experimental results at a glance

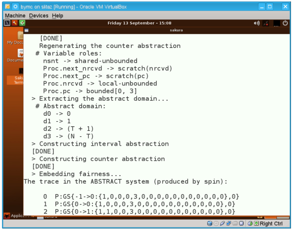
Algorithm	Fault	Resilience	Property	Valid?	#Refinements	Time
ST87	Byz	$n > 3t$	U	✓	0	4 sec.
ST87	Byz	$n > 3t$	C	✓	10	32 sec.
ST87	Byz	$n > 3t$	R	✓	10	24 sec.
ST87	Symm	$n > 2t$	U	✓	0	1 sec.
ST87	Symm	$n > 2t$	C	✓	2	3 sec.
ST87	Symm	$n > 2t$	R	✓	12	16 sec.
ST87	Omit	$n > 2t$	U	✓	0	1 sec.
ST87	Omit	$n > 2t$	C	✓	5	6 sec.
ST87	Omit	$n > 2t$	R	✓	5	10 sec.
ST87	Clean	$n > t$	U	✓	0	2 sec.
ST87	Clean	$n > t$	C	✓	4	8 sec.
ST87	Clean	$n > t$	R	✓	13	31 sec.
CT96	Clean	$n > t$	U	✓	0	1 sec.
CT96	Clean	$n > t$	A	✓	0	1 sec.
CT96	Clean	$n > t$	R	✓	0	1 sec.
CT96	Clean	$n > t$	C	✗	0	1 sec.

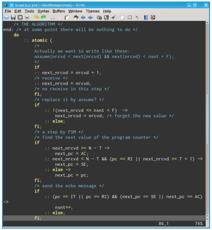
When resilience condition is wrong...

Algorithm	Fault	Resilience	Property	Valid?	#Refinements	Time
ST87	Byz	$n > 3t \wedge f \leq t{+}1$	**U**	✗	9	56 sec.
ST87	Byz	$n > 3t \wedge f \leq t{+}1$	**C**	✗	11	52 sec.
ST87	Byz	$n > 3t \wedge f \leq t{+}1$	**R**	✗	10	17 sec.
ST87	Byz	$n \geq 3t \wedge f \leq t$	**U**	✓	0	5 sec.
ST87	Byz	$n \geq 3t \wedge f \leq t$	**C**	✓	9	32 sec.
ST87	Byz	$n \geq 3t \wedge f \leq t$	**R**	✗	30	78 sec.
ST87	Symm	$n > 2t \wedge f \leq t{+}1$	**U**	✗	0	2 sec.
ST87	Symm	$n > 2t \wedge f \leq t{+}1$	**C**	✗	2	4 sec.
ST87	Symm	$n > 2t \wedge f \leq t{+}1$	**R**	✓	8	12 sec.
ST87	Omit	$n > 2t \wedge f \leq t$	**U**	✓	0	1 sec.
ST87	Omit	$n > 2t \wedge f \leq t$	**C**	✗	0	2 sec.
ST87	Omit	$n > 2t \wedge f \leq t$	**R**	✗	0	2 sec.

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]

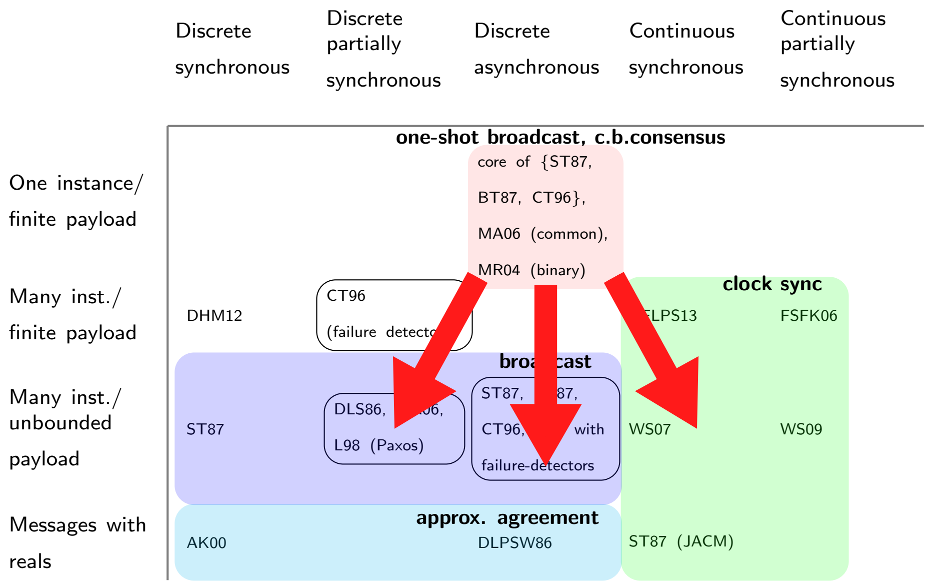
- Implementation.
- Experiments on Chandra-Toueg 1990.
- No resilience conditions like $n > 3t$.
- Safety only.

Our current work

one-shot broadcast, c.b.consensus

	Discrete synchronous	Discrete partially synchronous	Discrete asynchronous	Continuous synchronous	Continuous partially synchronous
One instance/ finite payload			core of {ST87, BT87, CT96}, MA06 (common), MR04 (binary)		
Many inst./ finite payload					
Many inst./ unbounded payload					
Messages with reals					

Future work: threshold guards + orthogonal features



Thank you!

[http://forsyte.at/software/bymc]

Fairness, Refinement, and Invariants

- In the Byzantine case we have $\mathit{in\_transit} : \forall i. (\mathit{recv}_i \geq \mathit{sent})$ and $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$.
- In this case communication fairness implies computation fairness.
- But in the abstract version *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., $\mathit{sent} = K([\mathit{sv} = \mathit{SE} \vee \mathit{sv} = \mathit{AC}])$ (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.

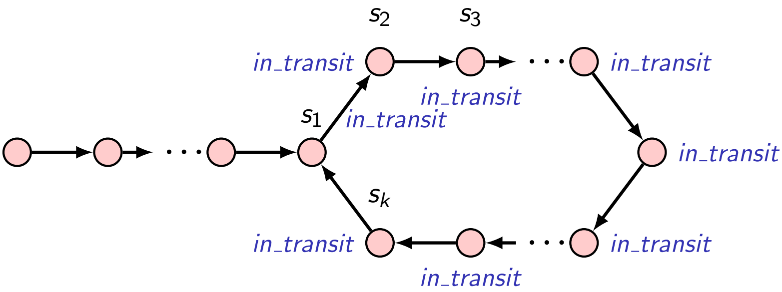
Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$ necessary to verify liveness

Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg in\_transit$ necessary to verify liveness

counter example:

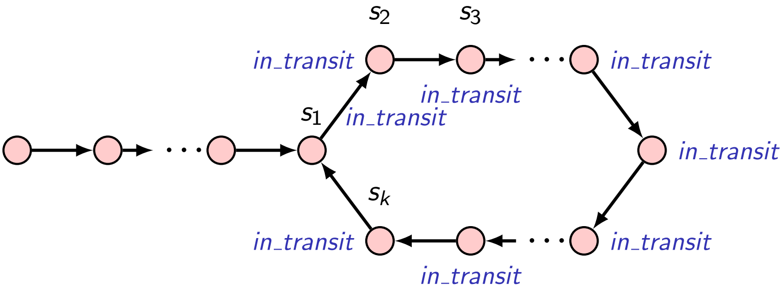


if $\forall j$ all concretizations of s_j violate $\neg in\_transit$, then CE is spurious.

Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$ necessary to verify liveness

counter example:



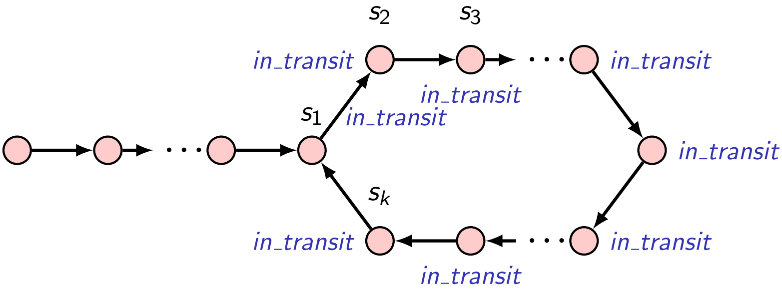
if $\forall j$ all concretizations of s_j violate $\neg \mathit{in\_transit}$, then CE is spurious.

refine justice to $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit} \ \wedge\ \mathbf{G}\,\mathbf{F}\left(\bigvee_{1 \le j \le k} \neg at(s_j)\right)$

Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$ necessary to verify liveness

counter example:



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

refine justice to $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit} \ \wedge\ \mathbf{G}\,\mathbf{F}\left(\bigvee_{1 \le j \le k} \neg at(s_j)\right)$

. . . we use unsat cores to refine several loops at once

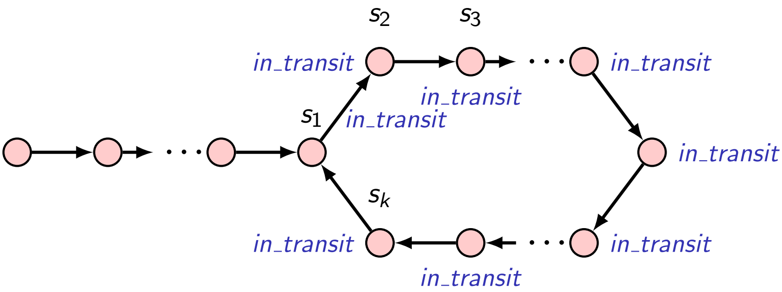
Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$ necessary to verify liveness

Parametric abstraction refinement — justice suppression

justice **G F** $\neg$*in_transit* necessary to verify liveness

counter example:

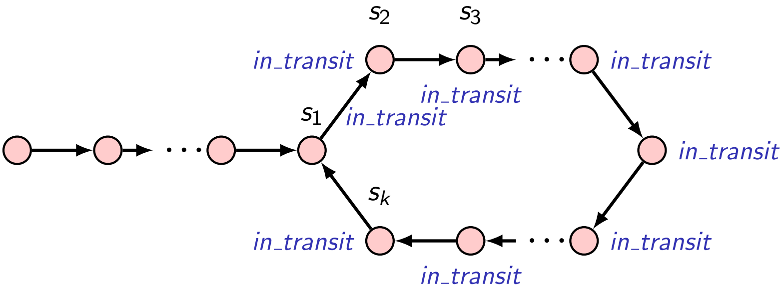


if $\forall j$ all concretizations of s_j violate $\neg$*in_transit*, then CE is spurious.

Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$ necessary to verify liveness

counter example:



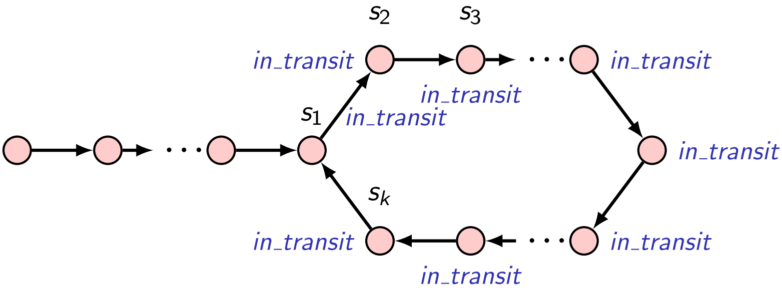
if $\forall j$ all concretizations of s_j violate $\neg \mathit{in\_transit}$, then CE is spurious.

refine justice to $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit} \ \wedge\ \mathbf{G}\,\mathbf{F} \left(\bigvee_{1 \le j \le k} \neg at(s_j) \right)$

Parametric abstraction refinement — justice suppression

justice $\mathbf{G}\,\mathbf{F}\,\neg in\_transit$ necessary to verify liveness

counter example:



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

refine justice to $\mathbf{G}\,\mathbf{F}\,\neg in\_transit \;\wedge\; \mathbf{G}\,\mathbf{F}\left(\bigvee_{1 \le j \le k} \neg at(s_j)\right)$

…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;
```

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;
```

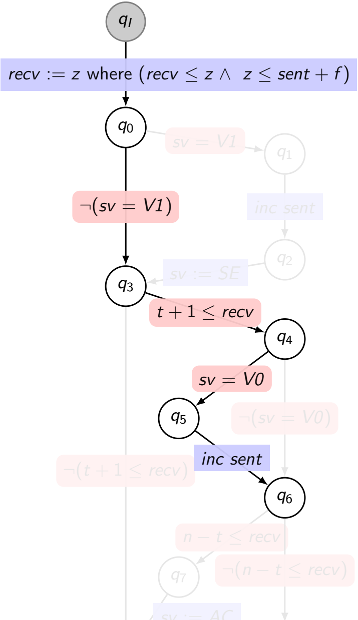
asynchronous

t byzantine faults

correct if $n > 3t$
resilience condition rc

parameterized process skeleton $p(n, t)$

Abstract CFA



Abstract CFA