Model Checking of Fault-Tolerant Distributed Algorithms

Part III: Parameterized Model Checking of Fault-tolerant Distributed Algorithms by Abstraction

Annu Gmeiner Igor Konnov Ulrich Schmid Helmut Veith
Josef Widder

SFM-14:ESM. Bertinoro, Italy, EU

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

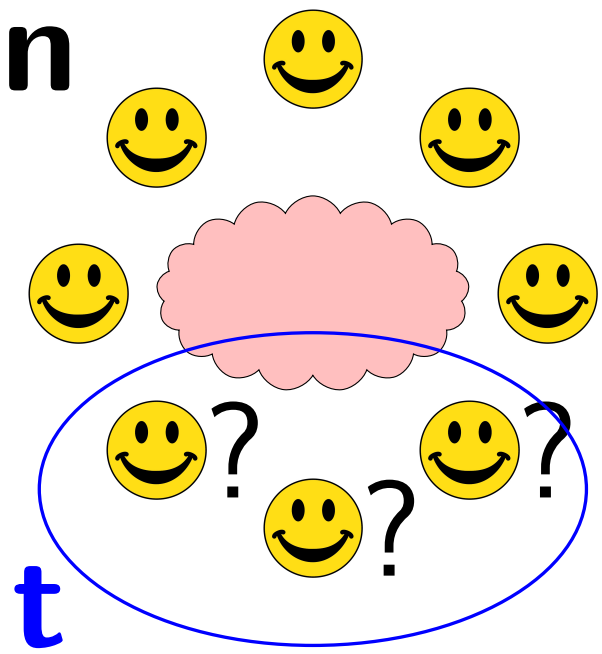
- continuous time

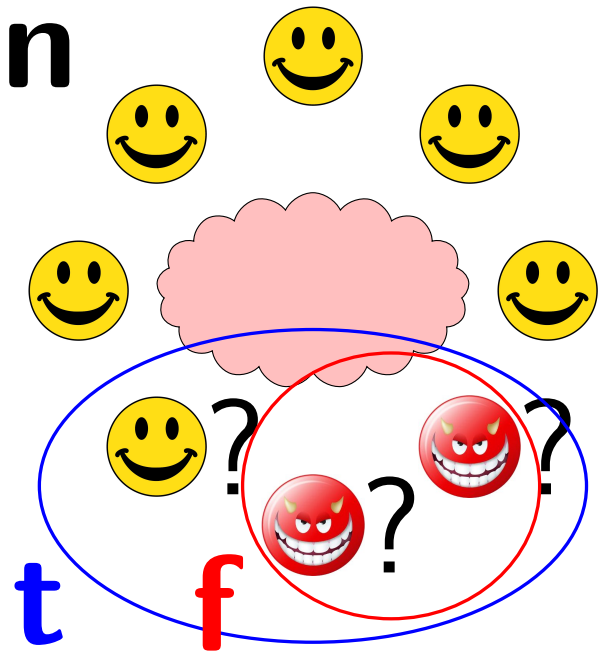
  fault-tolerant clock synchronization

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

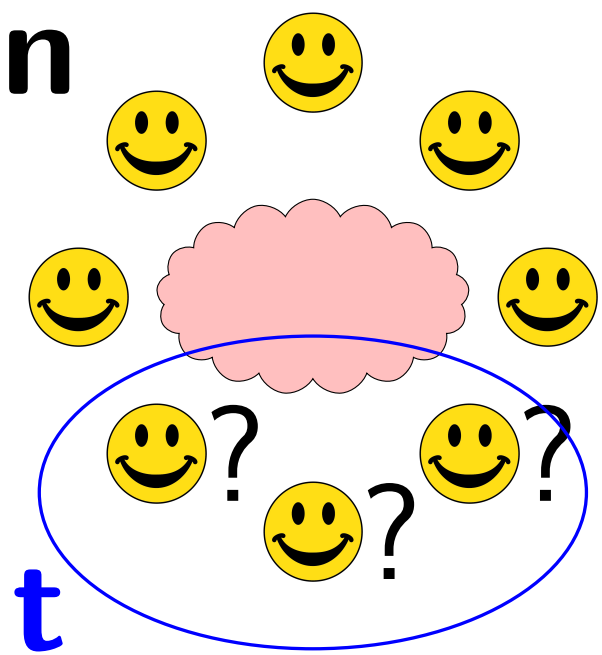


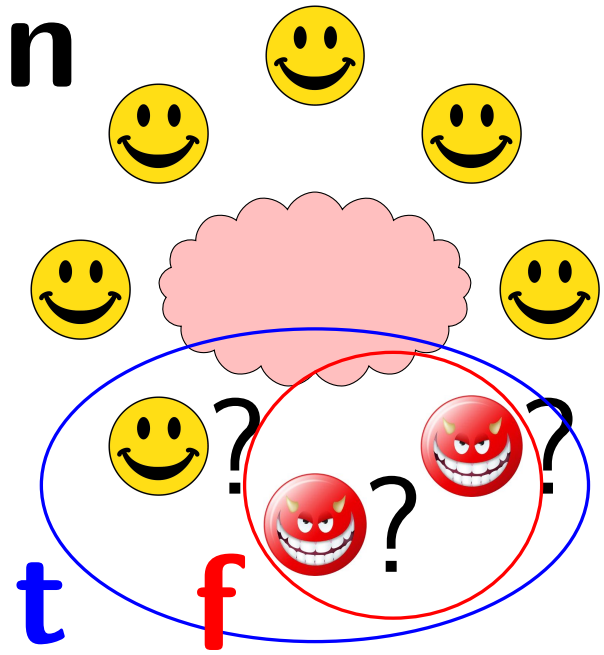


Model checking problem for fault-tolerant DA algorithms

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.

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

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

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

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

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

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**!)

Control Flow Automata

Variables of process i

v_i : {0, 1} **init with** 0 **or** 1

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

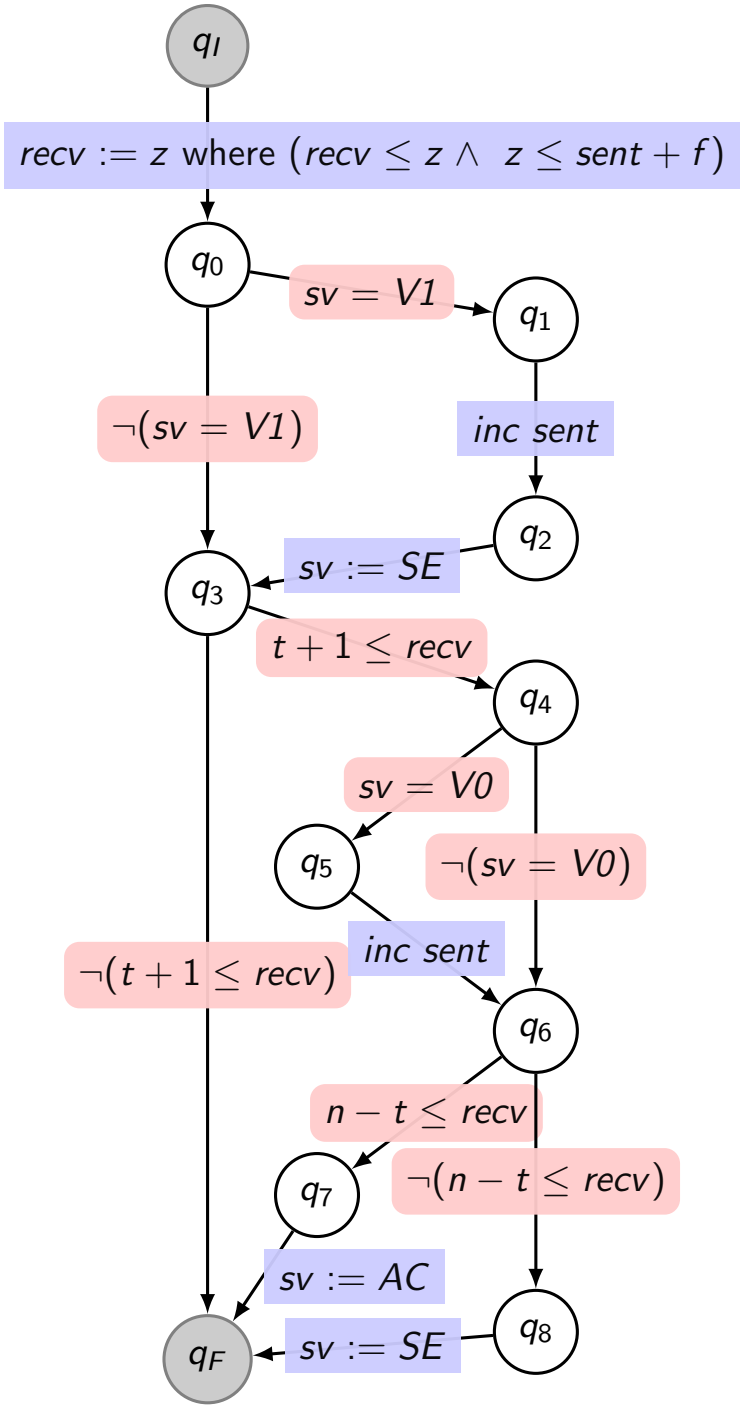
An indivisible step:

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

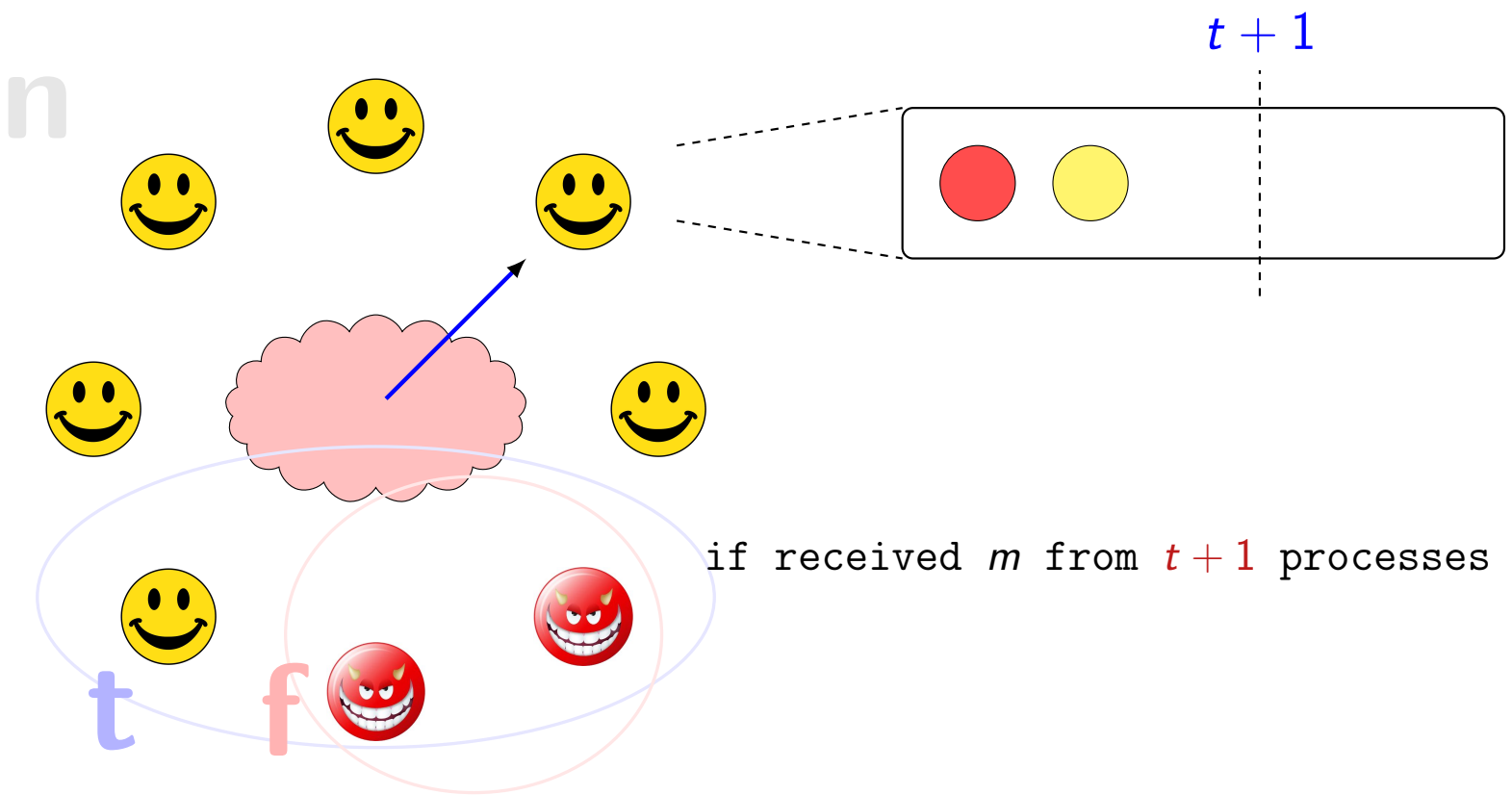
if received (echo) from at least
t + 1 distinct processes
and not sent (echo) before
then send (echo) to all;

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

$n - f$ copies of the process



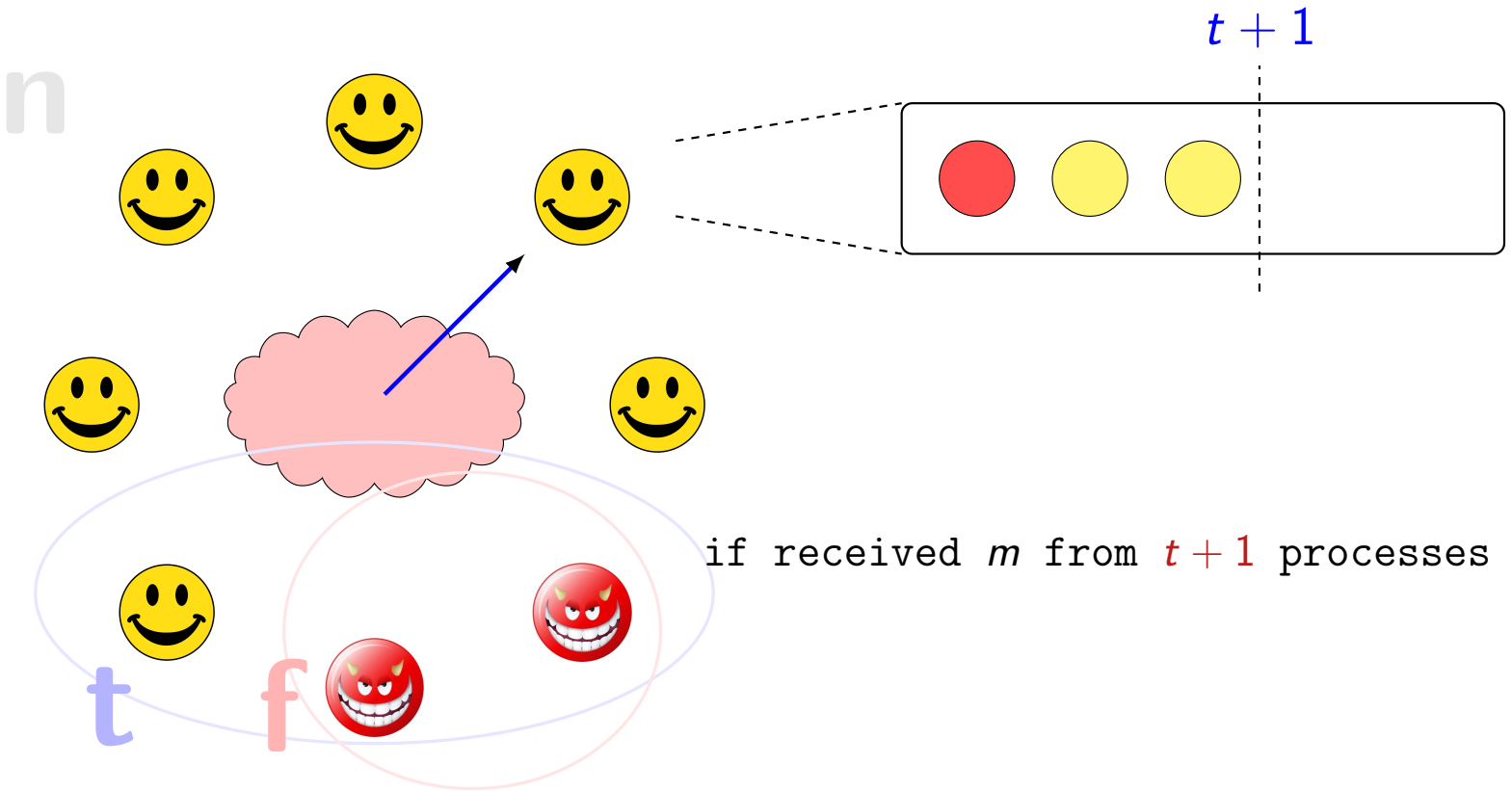
Counting argument in threshold-guarded algorithms



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

Correct processes count distinct incoming messages

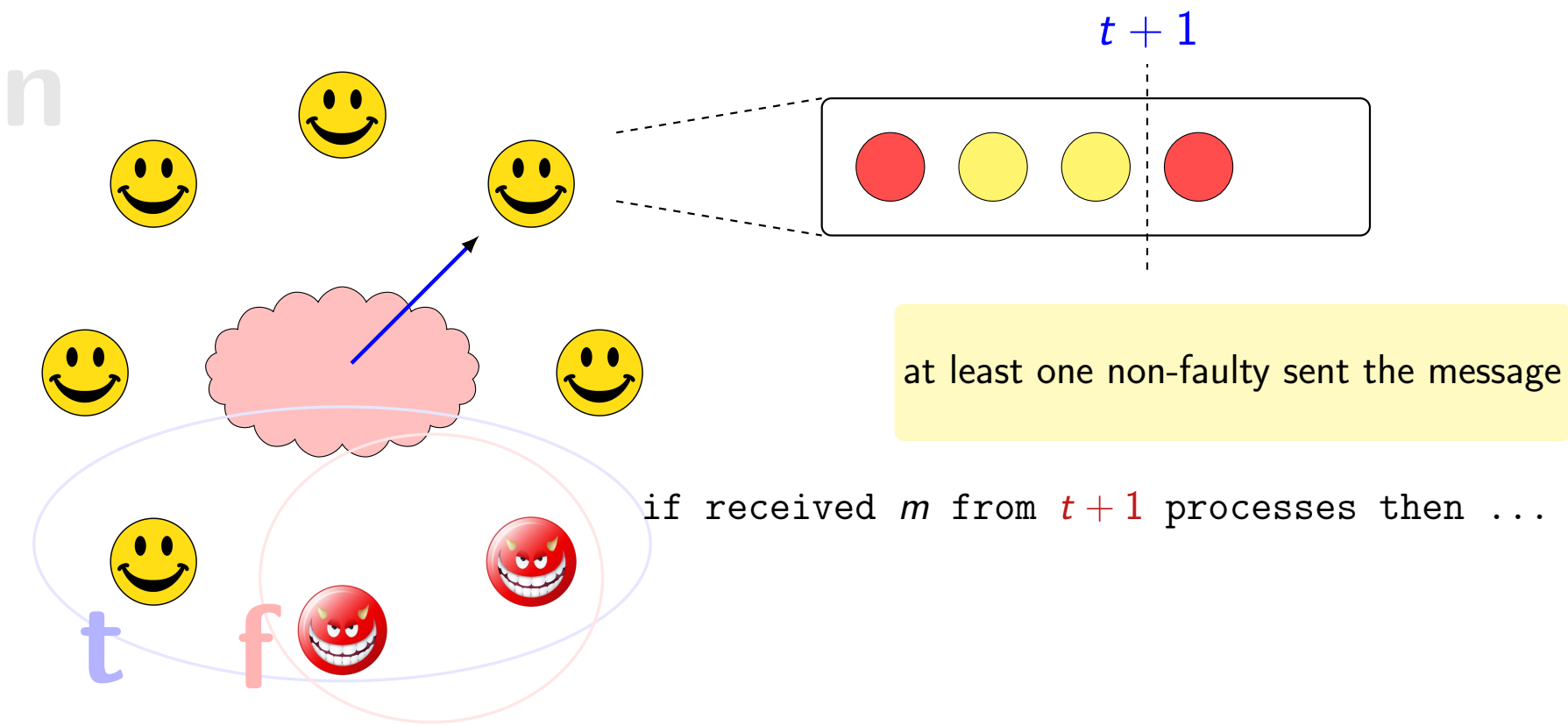
Counting argument in threshold-guarded algorithms



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

Correct processes count distinct incoming messages

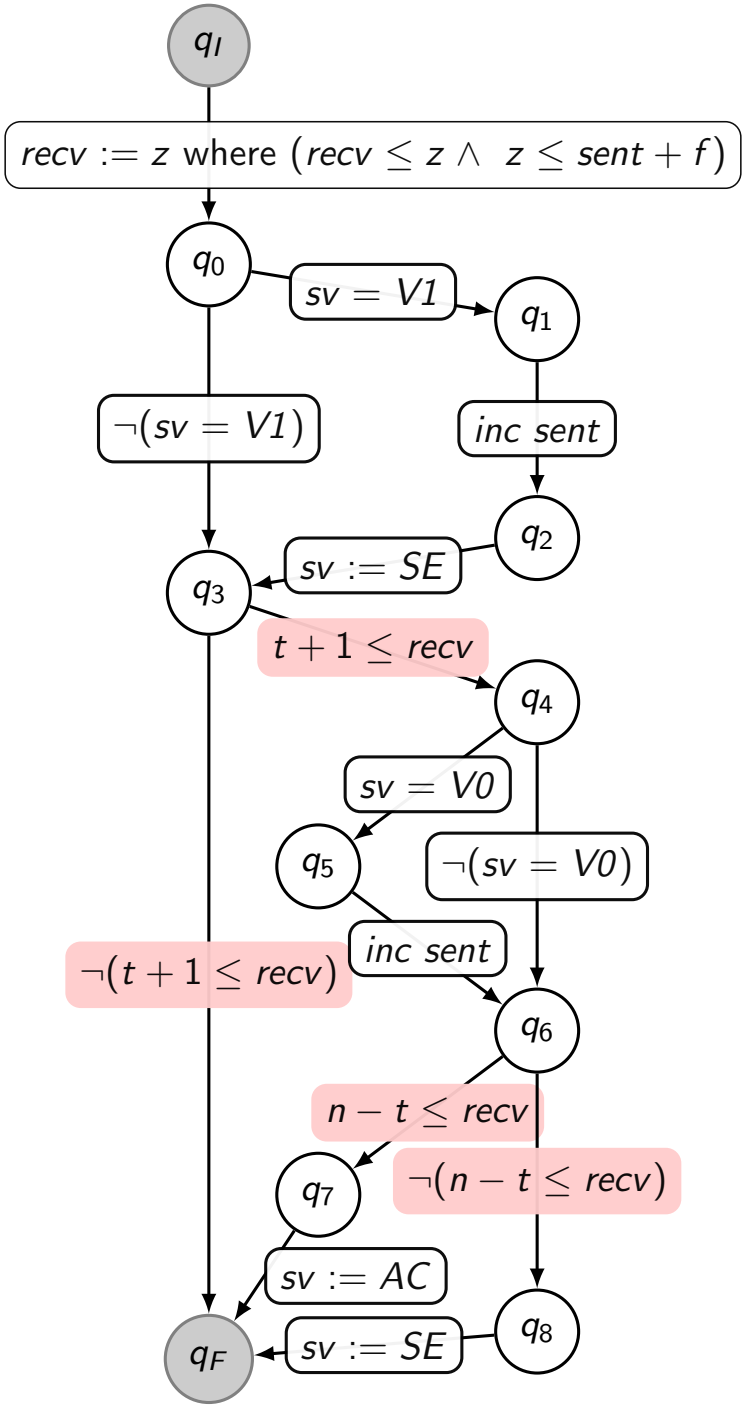
Counting argument in threshold-guarded algorithms



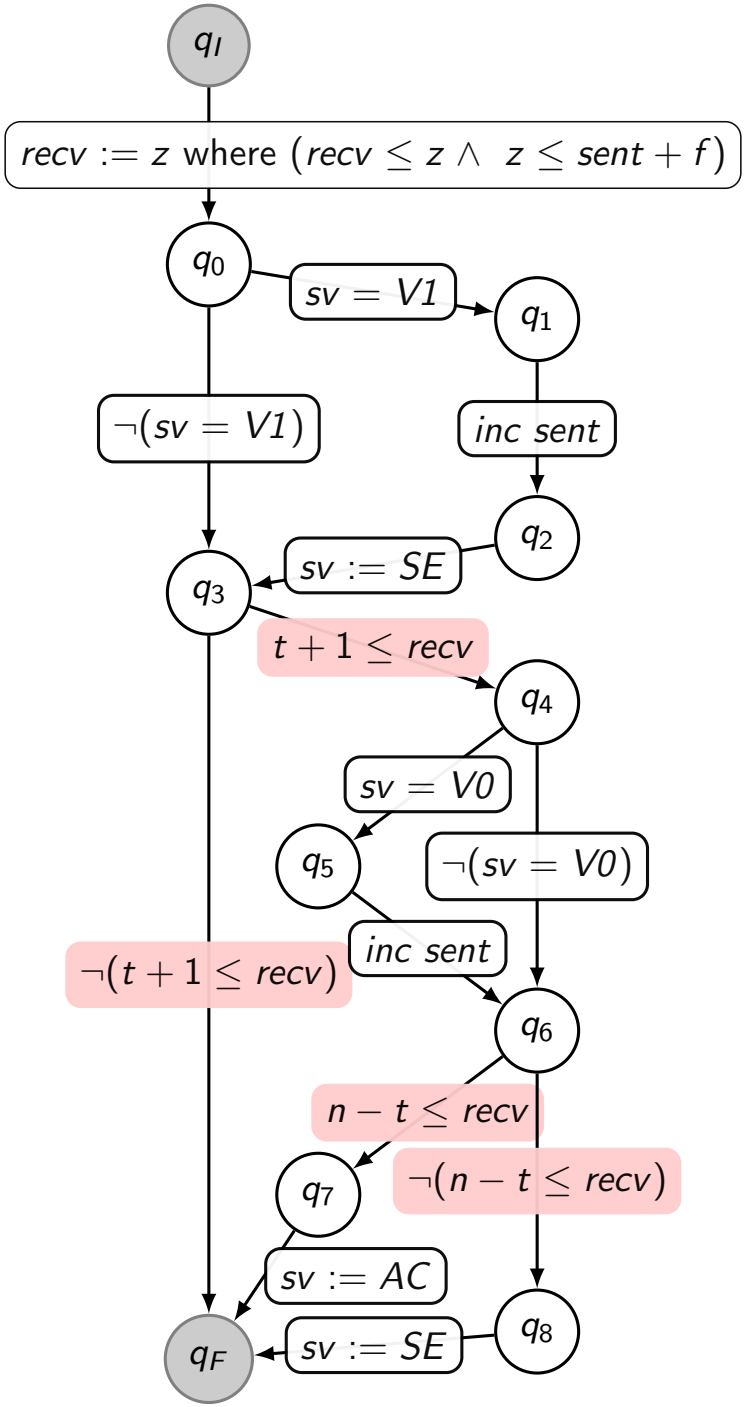
at least one non-faulty sent the message

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

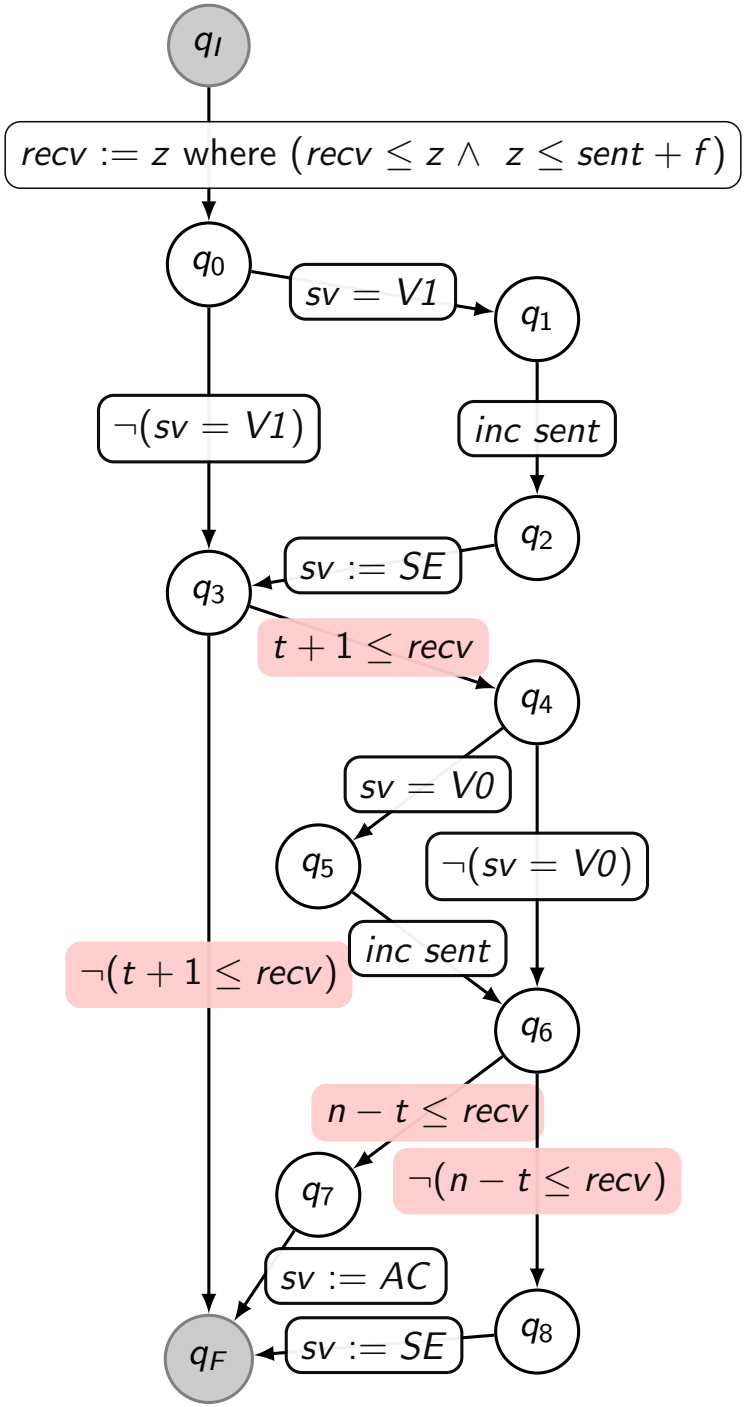
Correct processes count distinct incoming messages



- concrete values are not important
- thresholds are essential:
  $0,\ 1,\ t + 1,\ n - t$



- concrete values are not important
- thresholds are essential:
  $0,\ 1,\ t+1,\ n-t$
- intervals with symbolic boundaries:
  - $\mathrm{I}_0 = [0, 1)$
  - $\mathrm{I}_1 = [1, t+1)$
  - $\mathrm{I}_{t+1} = [t+1, n-t)$
  - $\mathrm{I}_{n-t} = [n-t, \infty)$



- concrete values are not important
- thresholds are essential:
  $0,\ 1,\ t+1,\ n-t$
- intervals with symbolic boundaries:
  - $\mathrm{I}_0 \quad = [0, 1)$
  - $\mathrm{I}_1 \quad = [1, t+1)$
  - $\mathrm{I}_{t+1} = [t+1, n-t)$
  - $\mathrm{I}_{n-t} = [n-t, \infty)$
- Parameteric Interval Abstraction (PIA)
- Similar to interval abstraction:
  $[t+1, n-t)$ rather than $[4, 10)$.
- Total order: $0 < 1 < t+1 < n-t$ for all parameters satisfying RC:
  $n > 3t, t \geq f \geq 0$.

Technical challenges

We have to reduce the verification of an infinite number of instances where

1. the process code is parameterized
2. the number of processes is parameterized

to one finite state model checking instance

Technical challenges

We have to reduce the verification of an infinite number of instances where

1. the process code is parameterized
2. the number of processes is parameterized

to one finite state model checking instance

We do that by:

1. PIA data abstraction
2. PIA counter abstraction

Technical challenges

We have to reduce the verification of an infinite number of instances where

1. the process code is parameterized
2. the number of processes is parameterized

to one finite state model checking instance

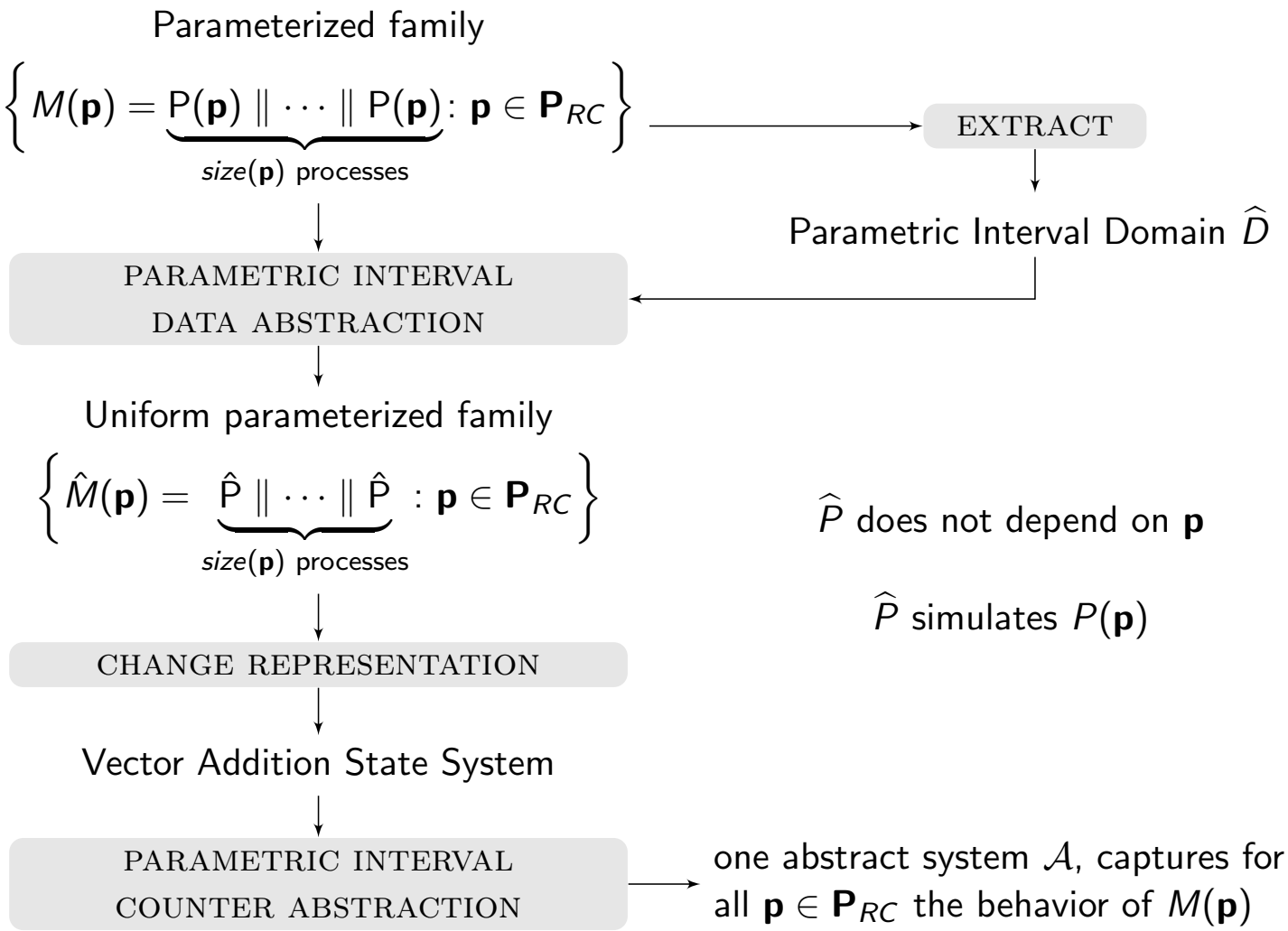
We do that by:

1. PIA data abstraction
2. PIA counter abstraction

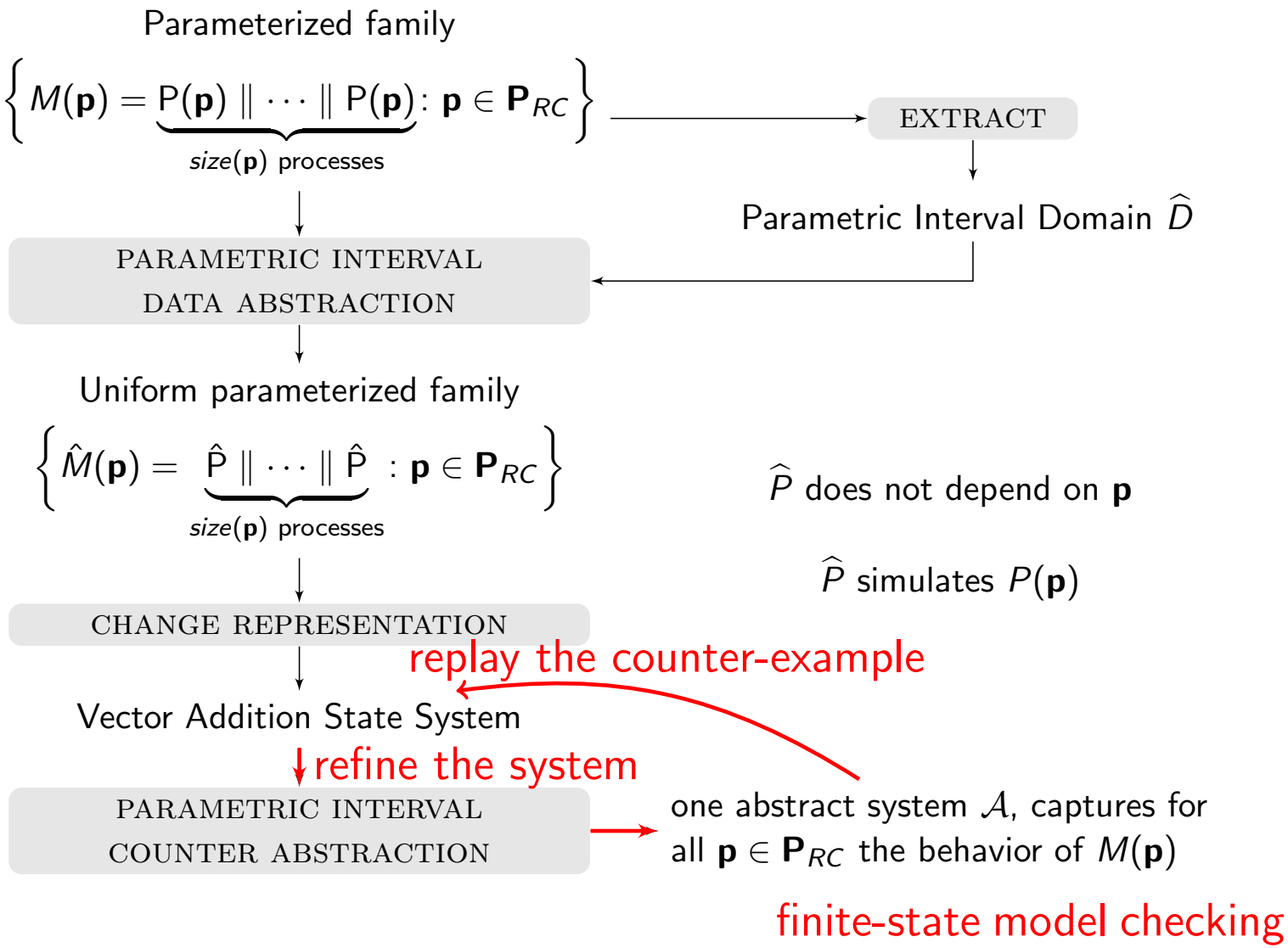
abstraction is an over approximation $\Rightarrow$ possible abstract behavior that does not correspond to a concrete behavior.

3. Refining spurious counter-examples

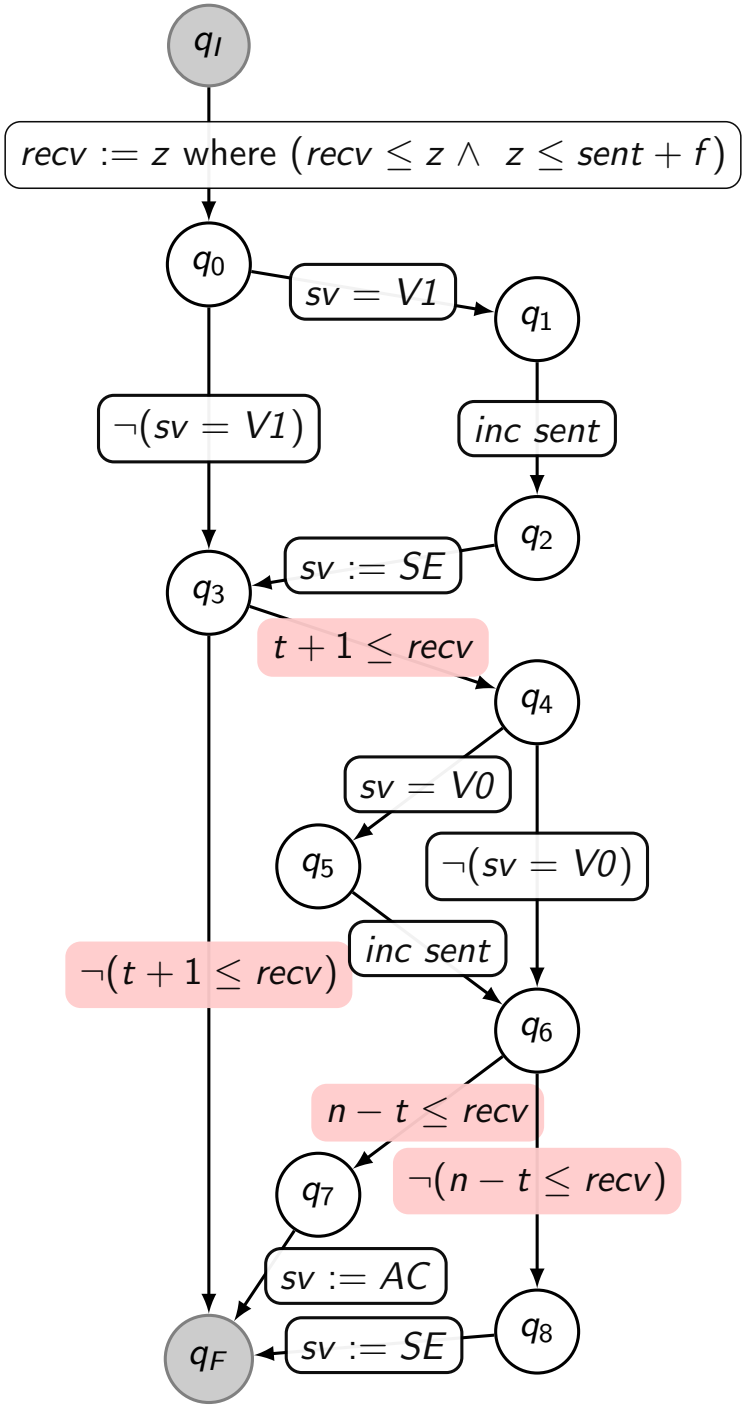
Abstraction overview



Abstraction overview

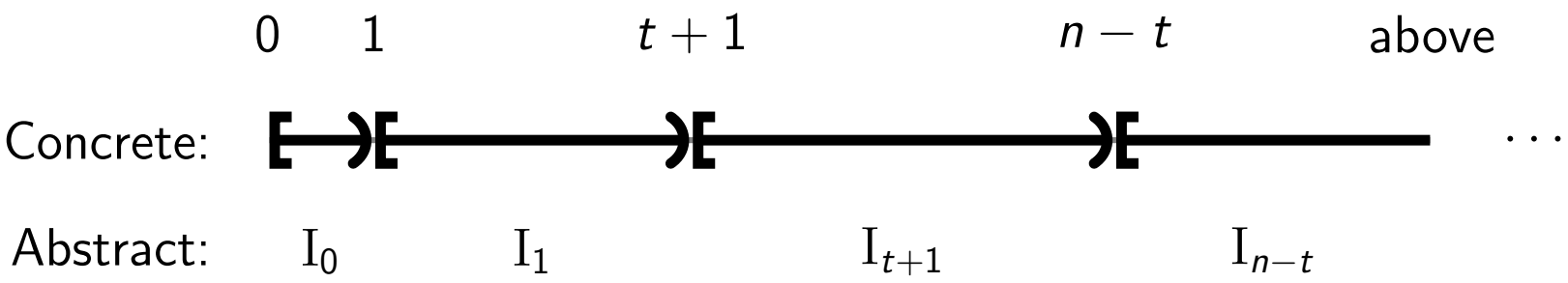


Data abstraction



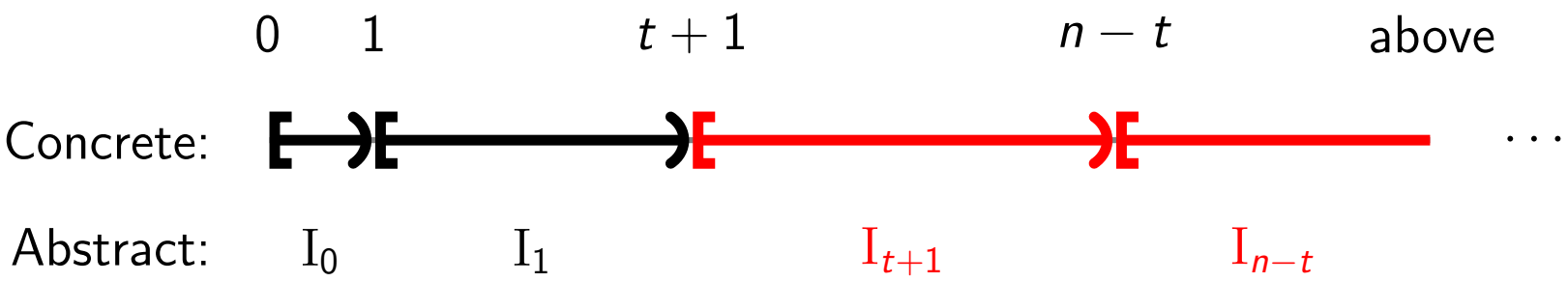
- concrete values are not important
- thresholds are essential:
  $0,\ 1,\ t+1,\ n-t$
- intervals with symbolic boundaries:
  - $\mathrm{I}_0 = [0, 1)$
  - $\mathrm{I}_1 = [1, t+1)$
  - $\mathrm{I}_{t+1} = [t+1, n-t)$
  - $\mathrm{I}_{n-t} = [n-t, \infty)$

Abstract operations



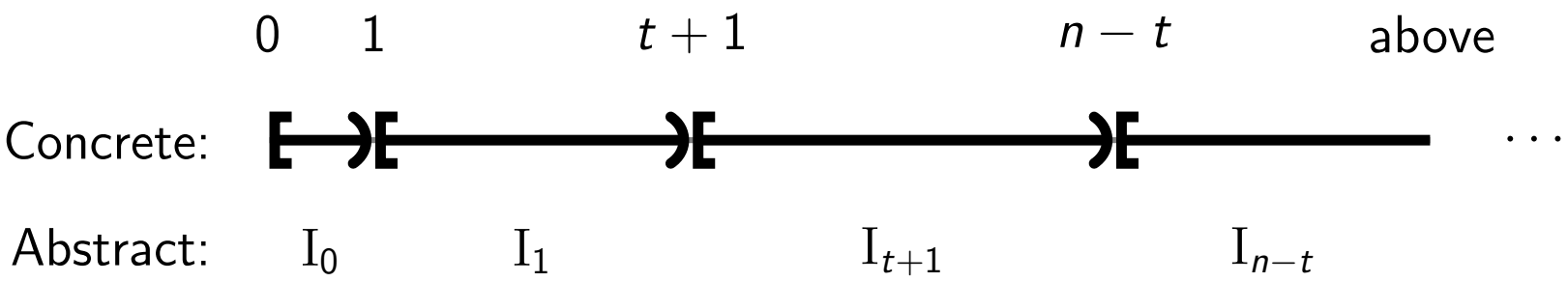
Concrete $t + 1 \leq x$

Abstract operations



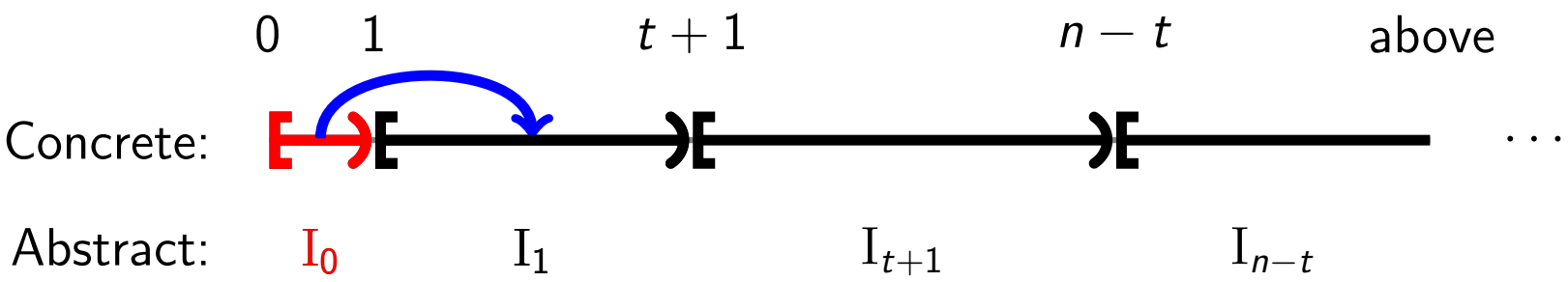
Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

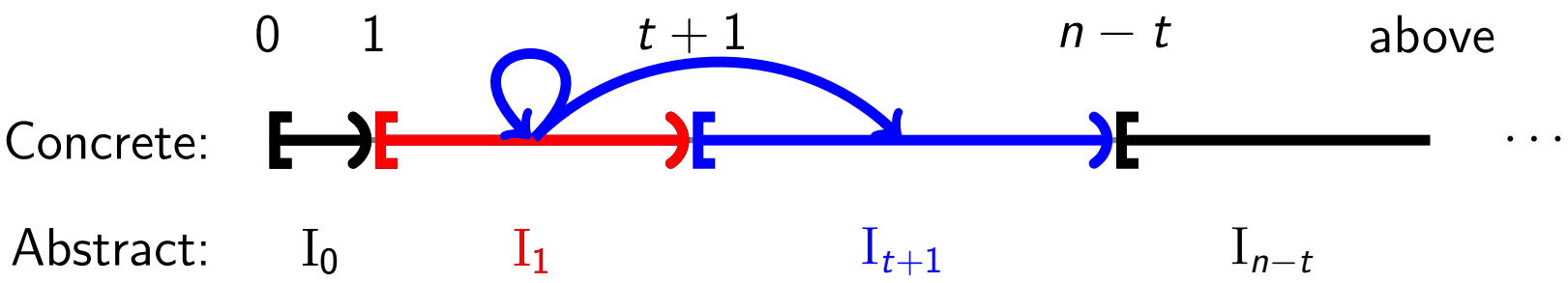


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

$$x = \mathrm{I}_0 \quad \wedge \quad x' = \mathrm{I}_1 \ \ldots$$

Abstract operations

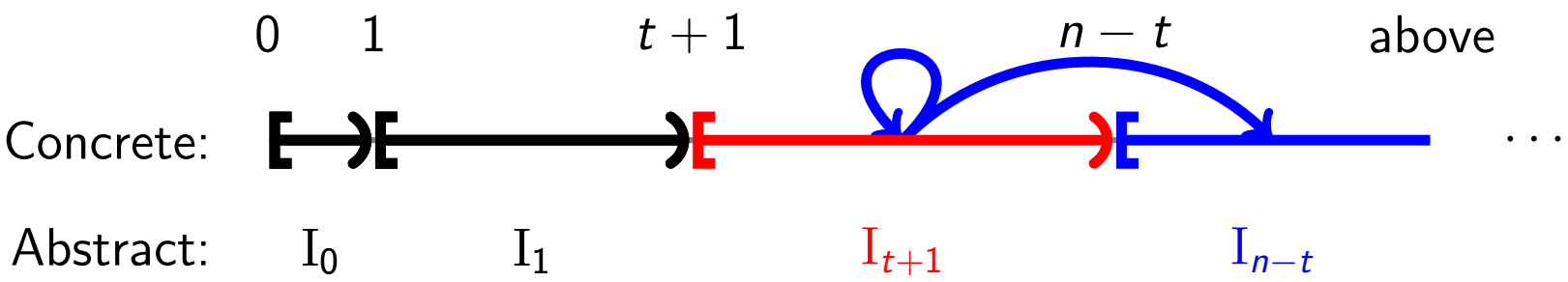


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

$$x = \mathrm{I}_0 \quad \wedge \; x' = \mathrm{I}_1$$
$$\vee x = \mathrm{I}_1 \quad \wedge (x' = \mathrm{I}_1 \quad \vee x' = \mathrm{I}_{t+1}) \ \ldots$$

Abstract operations

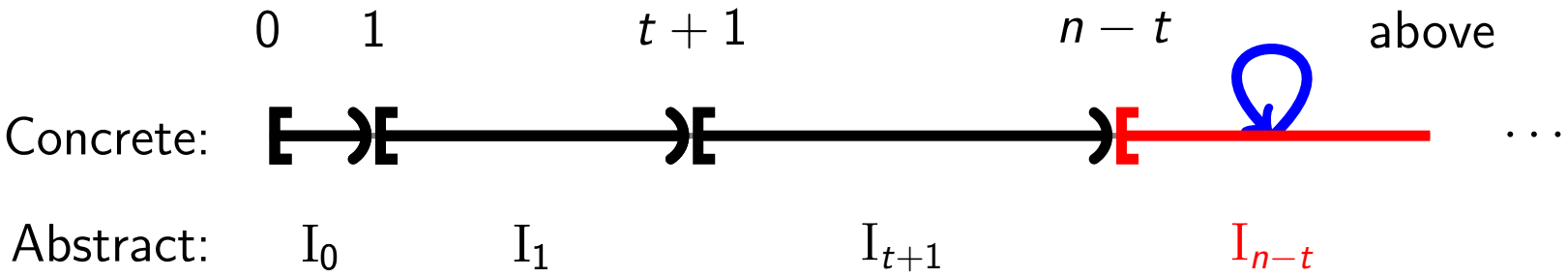


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

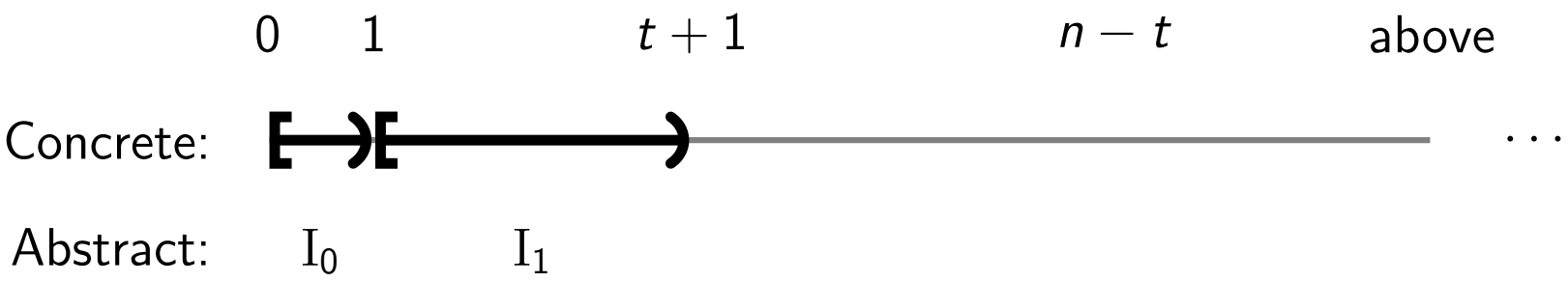


Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

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

Abstract operations



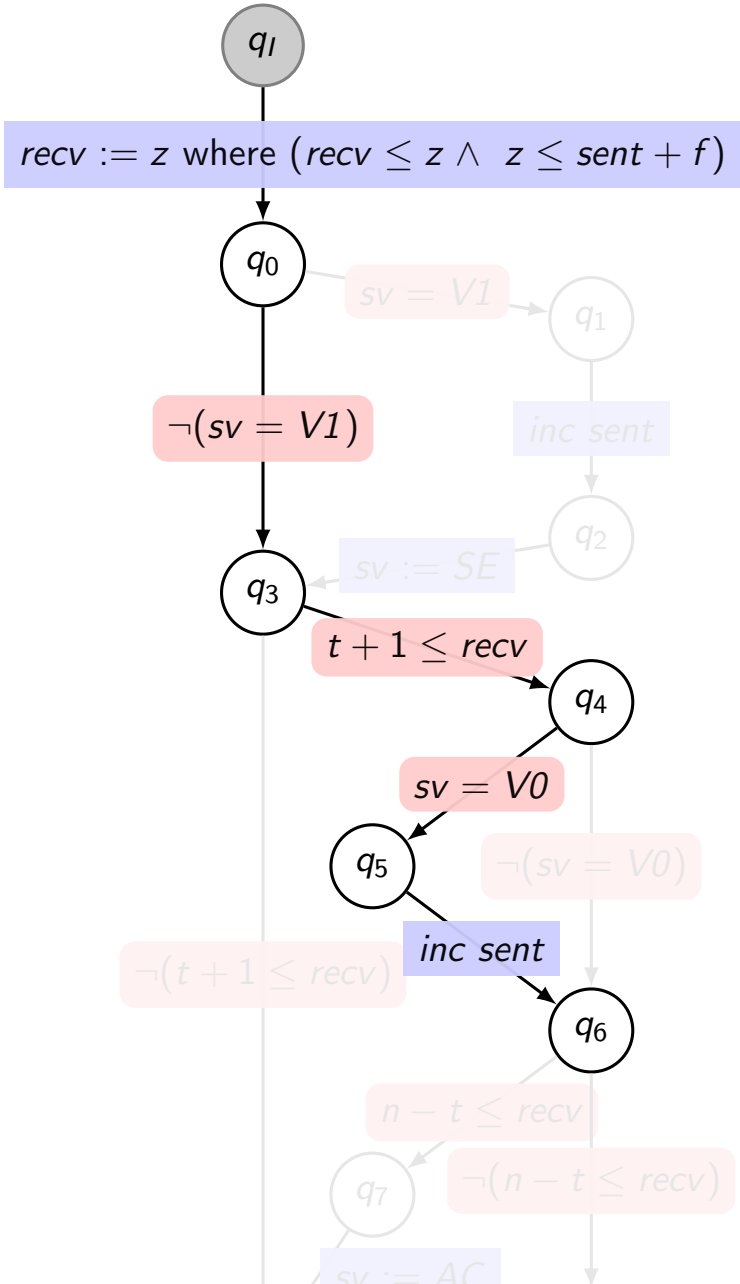
Concrete $t+1 \leq x$ is abstracted as $x = \mathrm{I}_{t+1} \vee x = \mathrm{I}_{n-t}$.

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

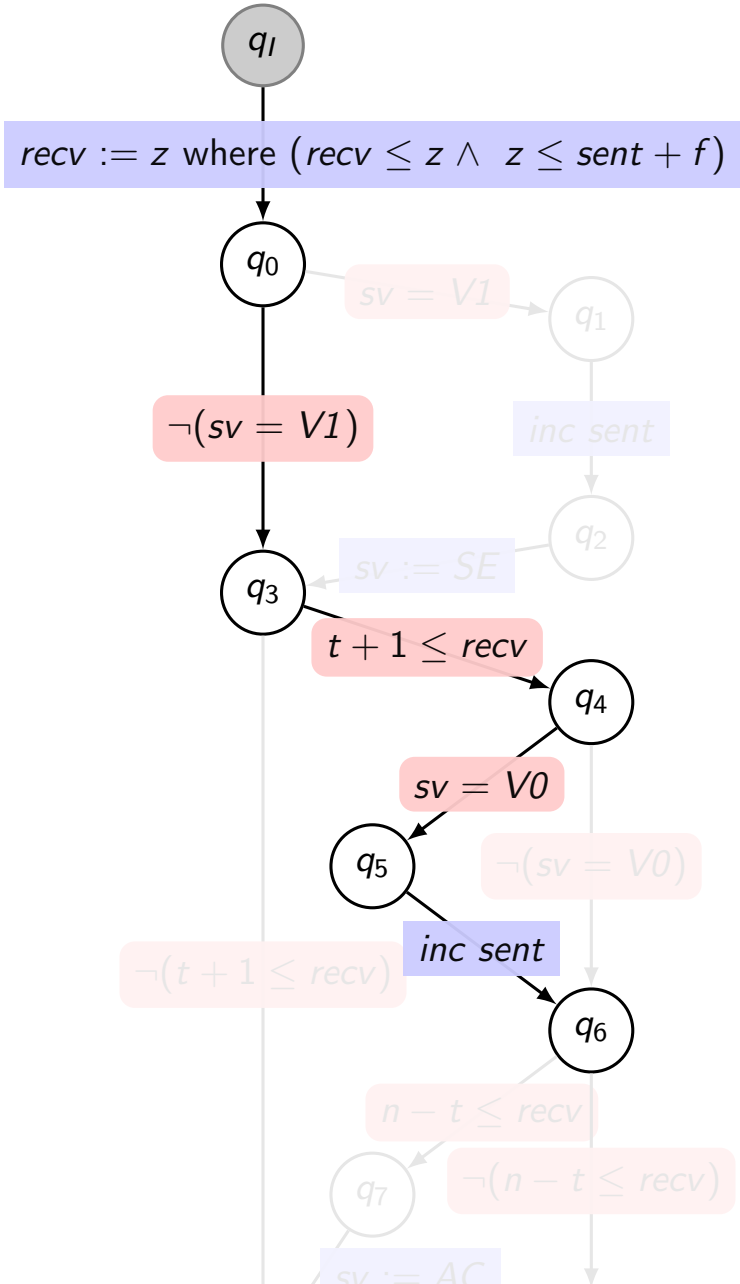
$$
\begin{aligned}
& x = \mathrm{I}_0 \quad \wedge \; x' = \mathrm{I}_1 \\
\vee\, & x = \mathrm{I}_1 \quad \wedge (x' = \mathrm{I}_1 \quad \vee\, x' = \mathrm{I}_{t+1}) \\
\vee\, & x = \mathrm{I}_{t+1} \wedge (x' = \mathrm{I}_{t+1} \vee x' = \mathrm{I}_{n-t}) \\
\vee\, & x = \mathrm{I}_{n-t} \wedge \;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

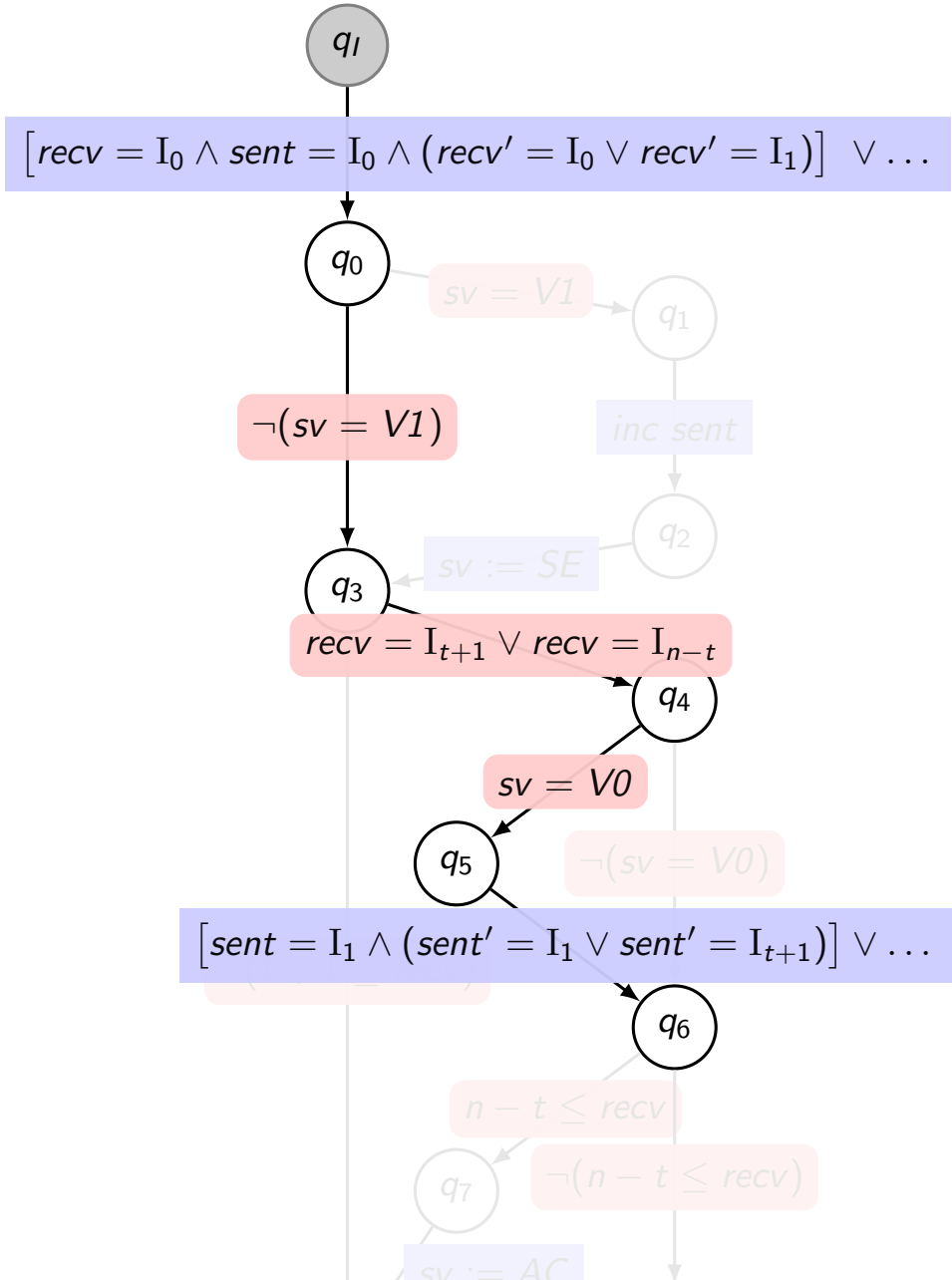
abstract increase may keep the same value!

Abstract CFA

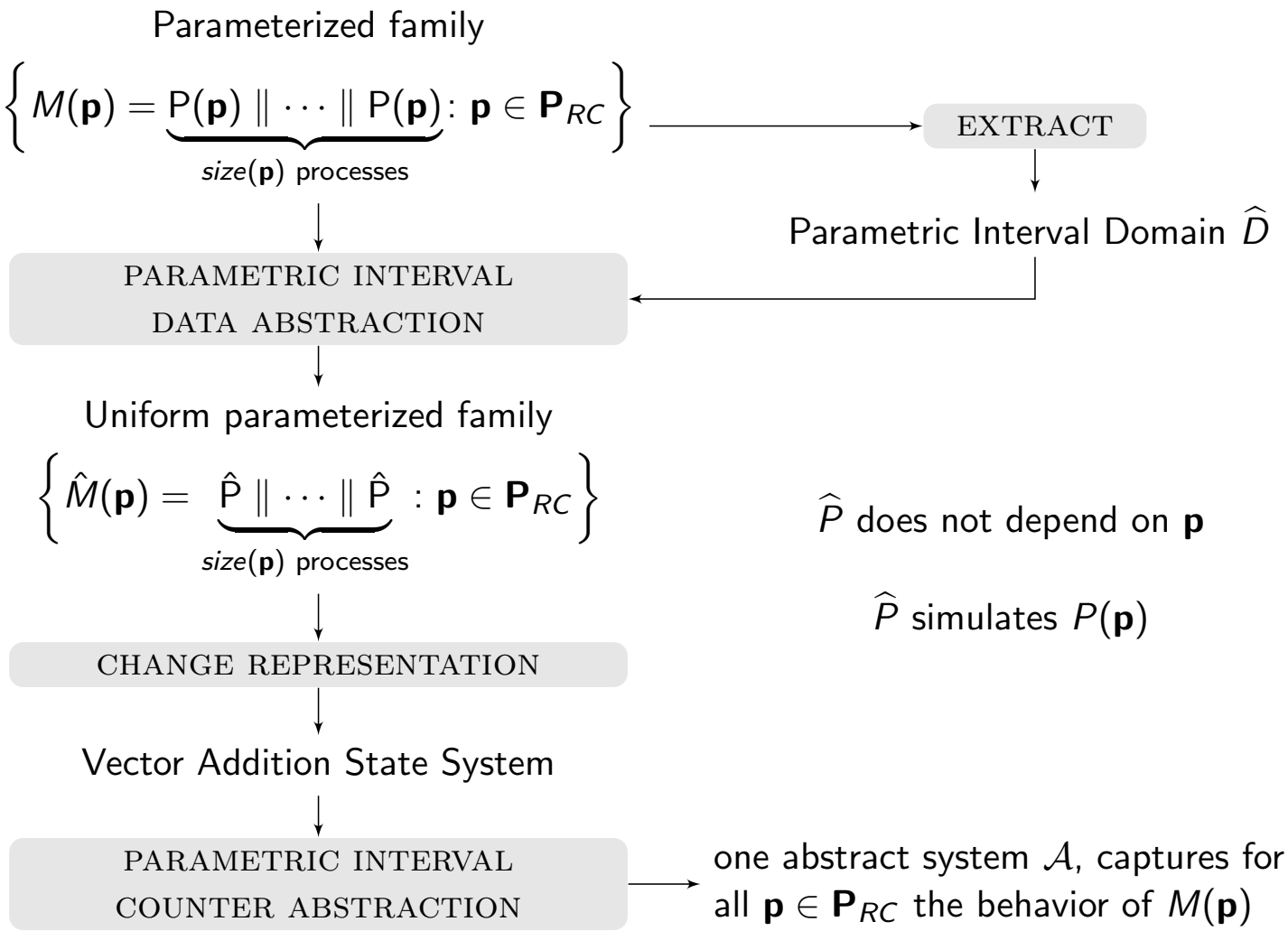


Abstract CFA





Abstraction overview



Counter abstraction

Classic $(0, 1, \infty)$-counter abstraction

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

- finitely many local states,
  e.g., $\{N, T, C\}$.
- based on counter representation:
  for each local states count how many processes are in it

Classic $(0, 1, \infty)$-counter abstraction

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

- finitely many local states,
  e.g., $\{N, T, C\}$.
- based on counter representation:
  for each local states count how many processes are in it
- 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., $\mathbf{G}\,(K(C) = \mathbf{0} \vee K(C) = \mathbf{1})$.

Limits of $(0, 1, \infty)$-counter abstraction

Our parametric data + counter abstraction:

- we require finer counting of processes:
  - $t + 1$ processes in a specific state can force global progress,
  - t processes cannot

- mapping t, $t + 1$, and $n - t$ to **"many"** is too coarse.

Limits of $(0, 1, \infty)$-counter abstraction

Our parametric data + counter abstraction:

- we require finer counting of processes:
  - $t + 1$ processes in a specific state can force global progress,
  - t processes cannot

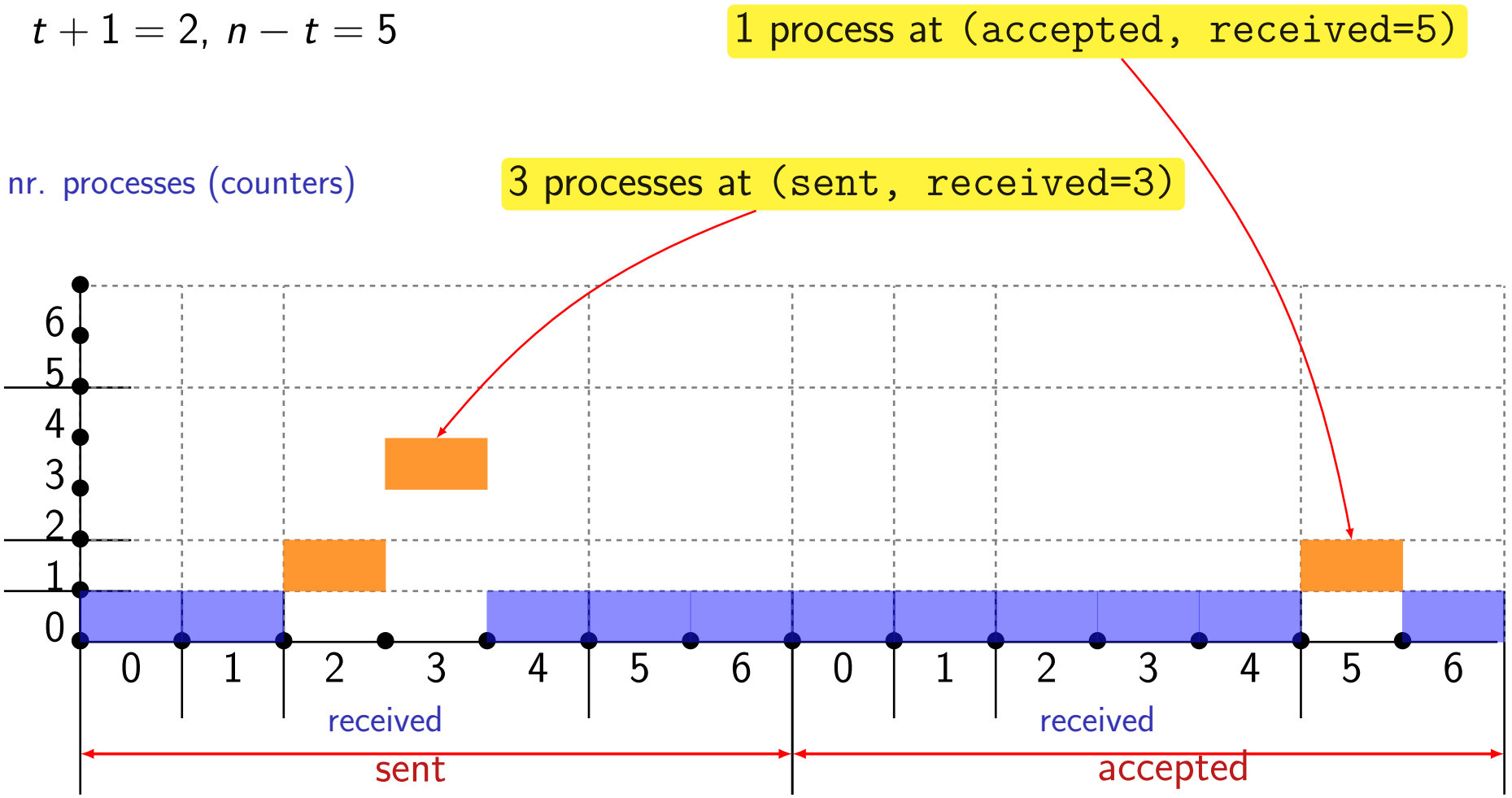
- mapping t, $t + 1$, and $n - t$ to **"many"** is too coarse.

starting point of our approach...

Data + counter abstraction over parametric intervals

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

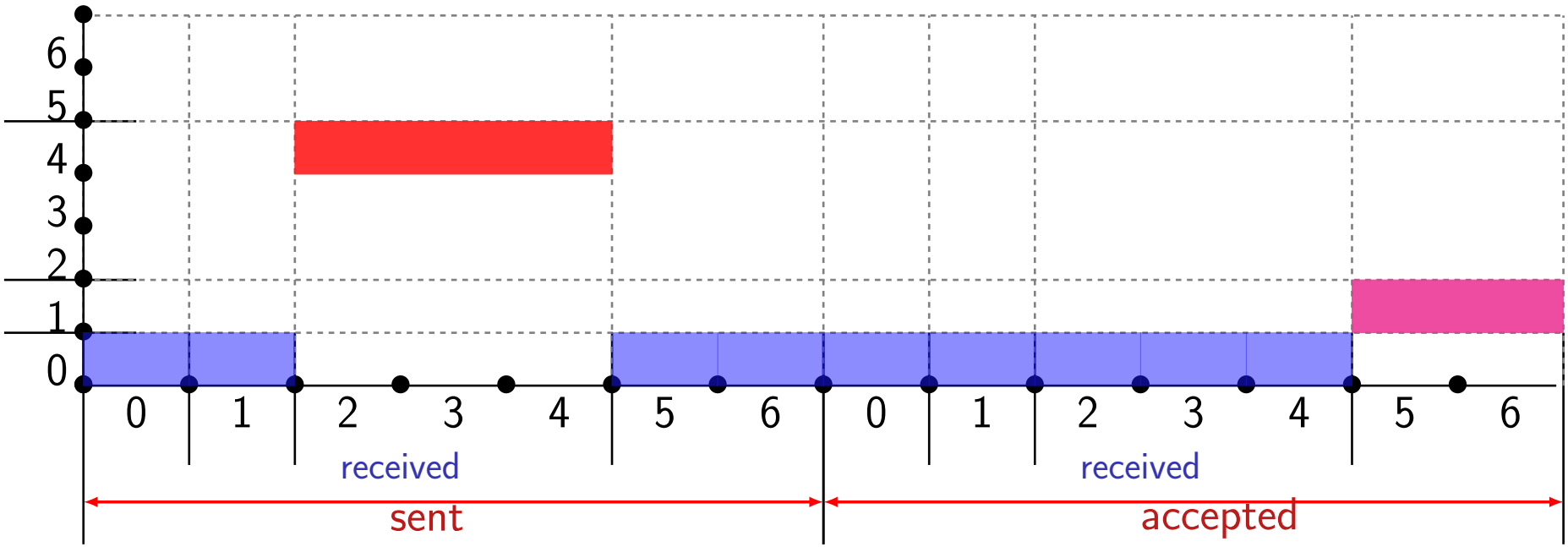
$t + 1 = 2,\ n - t = 5$



Data + counter abstraction over parametric intervals

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

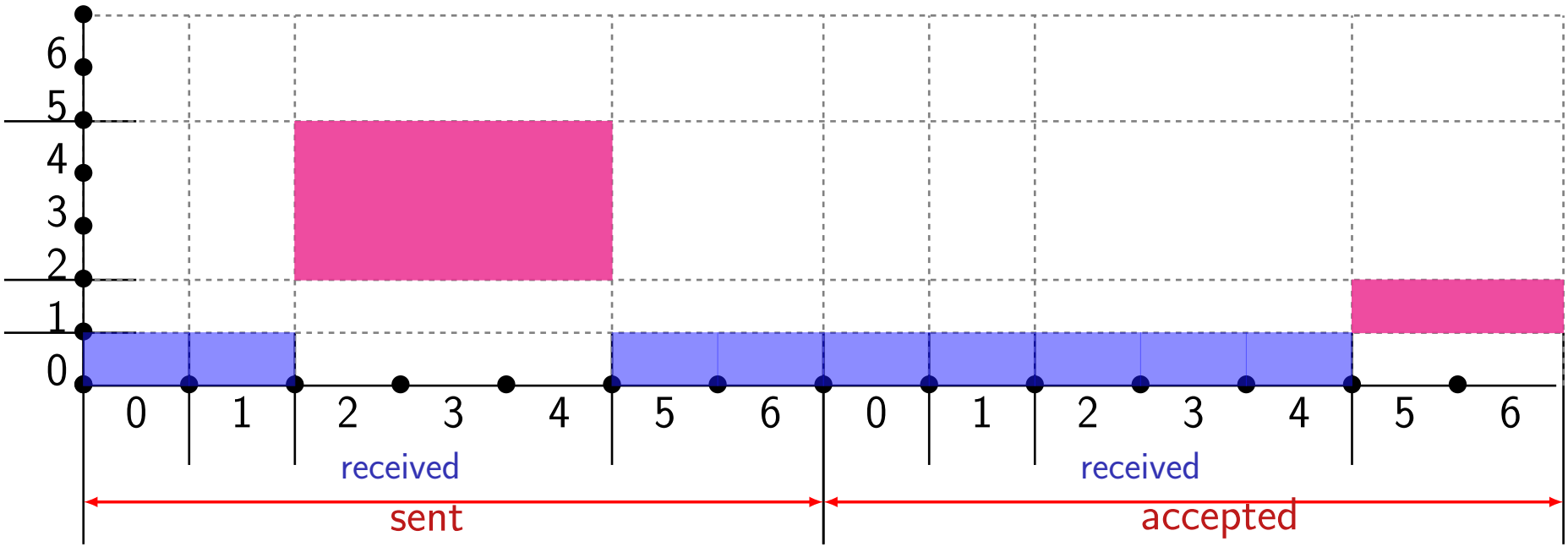
$t + 1 = 2,\ n - t = 5$



Data + counter abstraction over parametric intervals

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

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



Data + counter abstraction over parametric intervals

~~$n \geq 6$, $t \geq 1$, $f \geq 1$~~

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

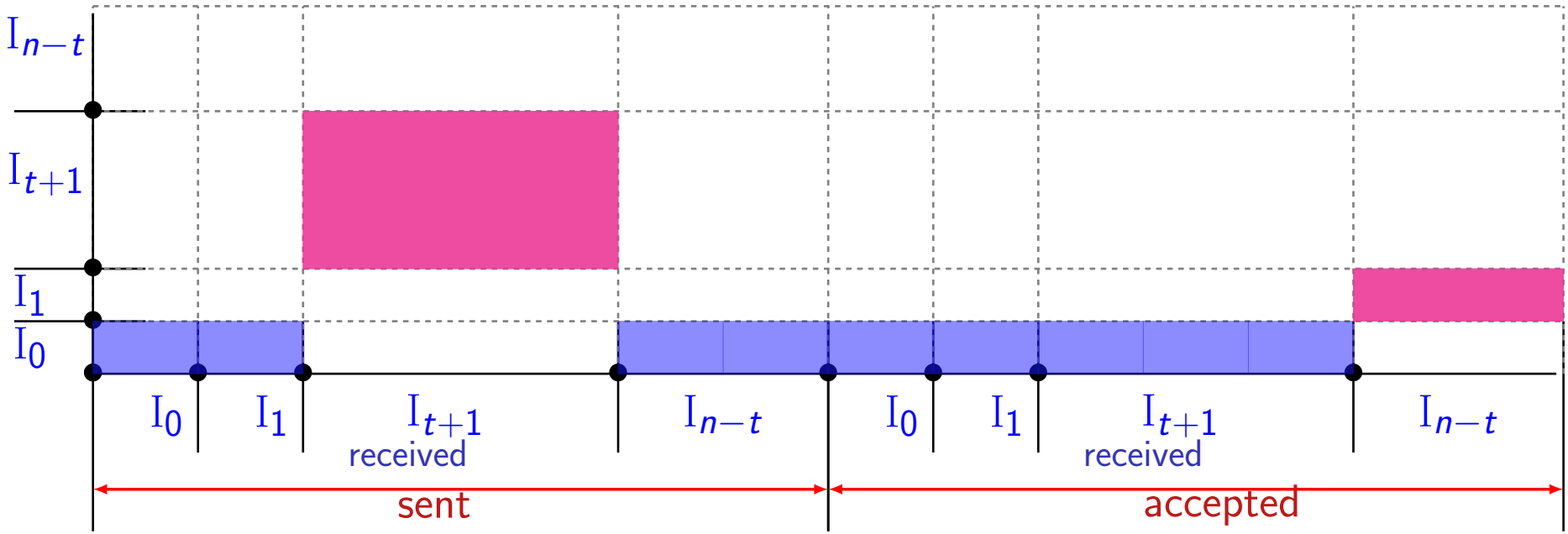
Parametric intervals:

$\mathrm{I}_0 = [0, 1) \quad \mathrm{I}_1 = [1, t+1)$

$\mathrm{I}_{t+1} = [t+1, n-t)$

$\mathrm{I}_{n-t} = [n-t, \infty)$

nr. processes (counters)



Data + counter abstraction over parametric intervals

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

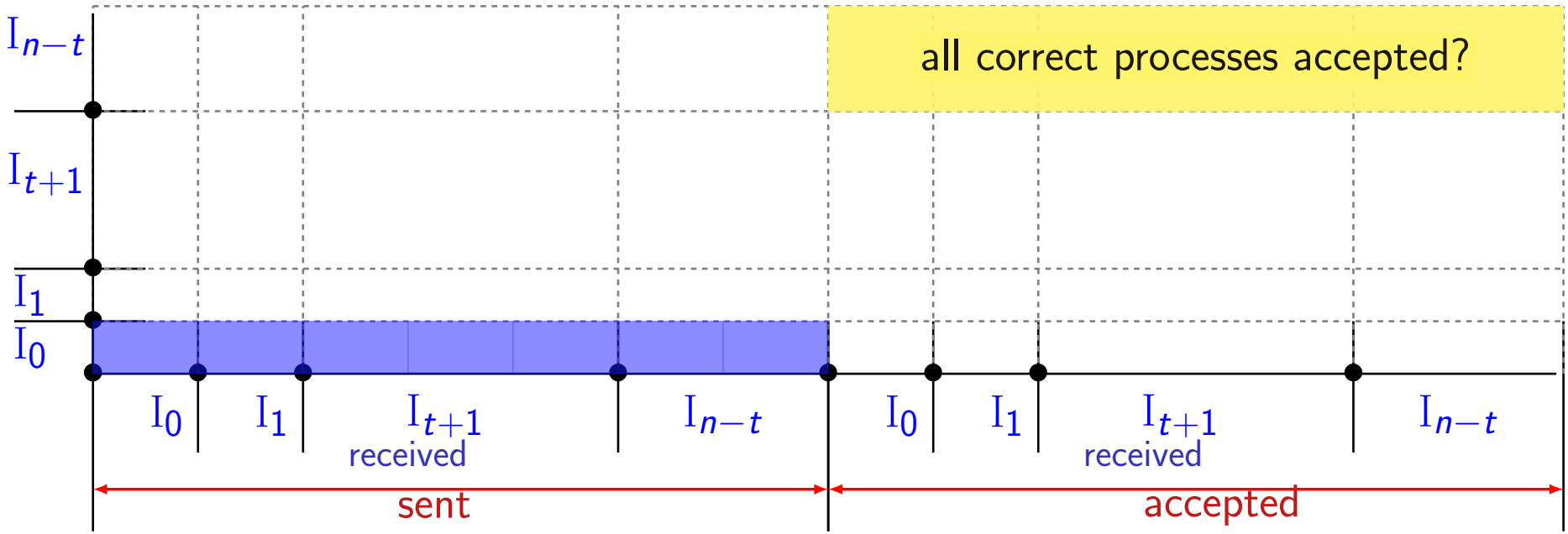
Parametric intervals:

$\mathrm{I}_0 = [0, 1) \quad \mathrm{I}_1 = [1, t+1)$

$\mathrm{I}_{t+1} = [t+1, n-t)$

$\mathrm{I}_{n-t} = [n-t, \infty)$

nr. processes (counters)



Abstraction refinement

Spurious behavior

abstraction adds behaviors (e.g., `x'=x+1` may lead to `x'` being equal to `x`)

Spurious behavior

abstraction adds behaviors (e.g., `x'=x+1` may lead to `x'` being equal to `x`)

$\Rightarrow$ specs that hold in concrete system may be violated in abstract system

- spurious counterexamples
- we have to reduce the behaviors of the abstract system
  make it more concrete
- . . . based on the counterexamples = CEGAR

Spurious behavior

abstraction adds behaviors (e.g., `x'=x+1` may lead to `x'` being equal to `x`)

$\Rightarrow$ specs that hold in concrete system may be violated in abstract system

- spurious counterexamples
- we have to reduce the behaviors of the abstract system
  make it more concrete
- . . . based on the counterexamples = CEGAR

We have observed three sources of spurious behavior

- # processes decreasing or increasing
- # messages sent $\neq$ # processes which have sent a message
- unfair loops

Spurious behavior

abstraction adds behaviors (e.g., `x'=x+1` may lead to `x'` being equal to `x`)

$\Rightarrow$ specs that hold in concrete system may be violated in abstract system

- spurious counterexamples
- we have to reduce the behaviors of the abstract system
  make it more concrete
- . . . based on the counterexamples = CEGAR

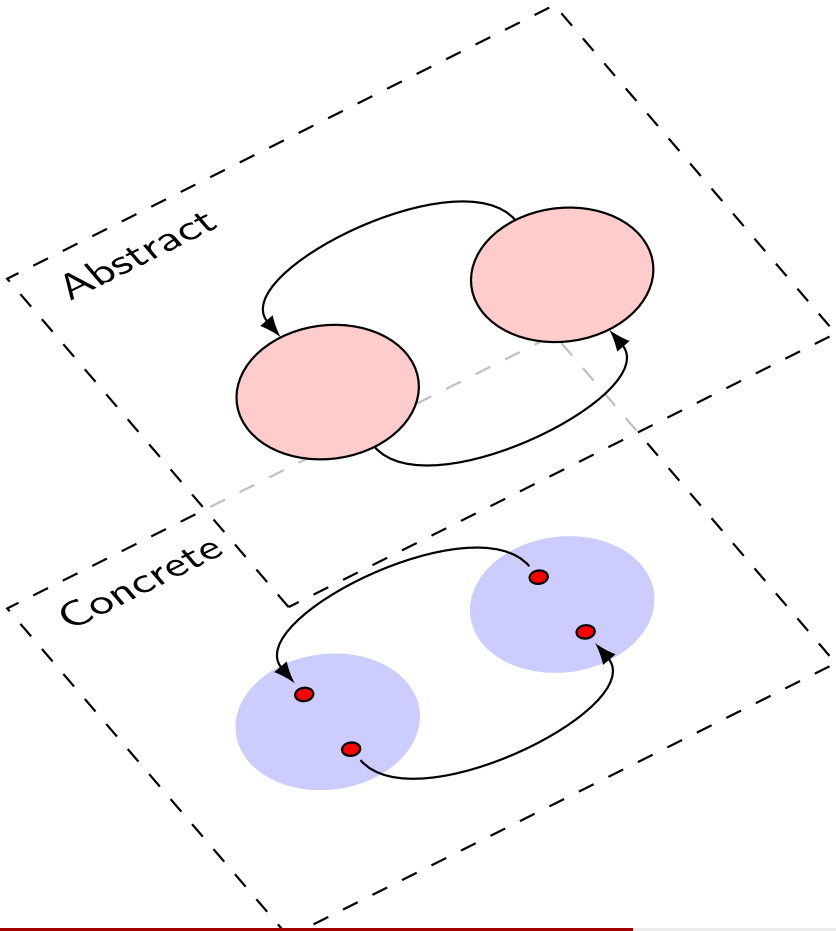
We have observed three sources of spurious behavior

- # processes decreasing or increasing
- # messages sent $\neq$ # processes which have sent a message
- unfair loops

. . . and a new abstraction phenomenon

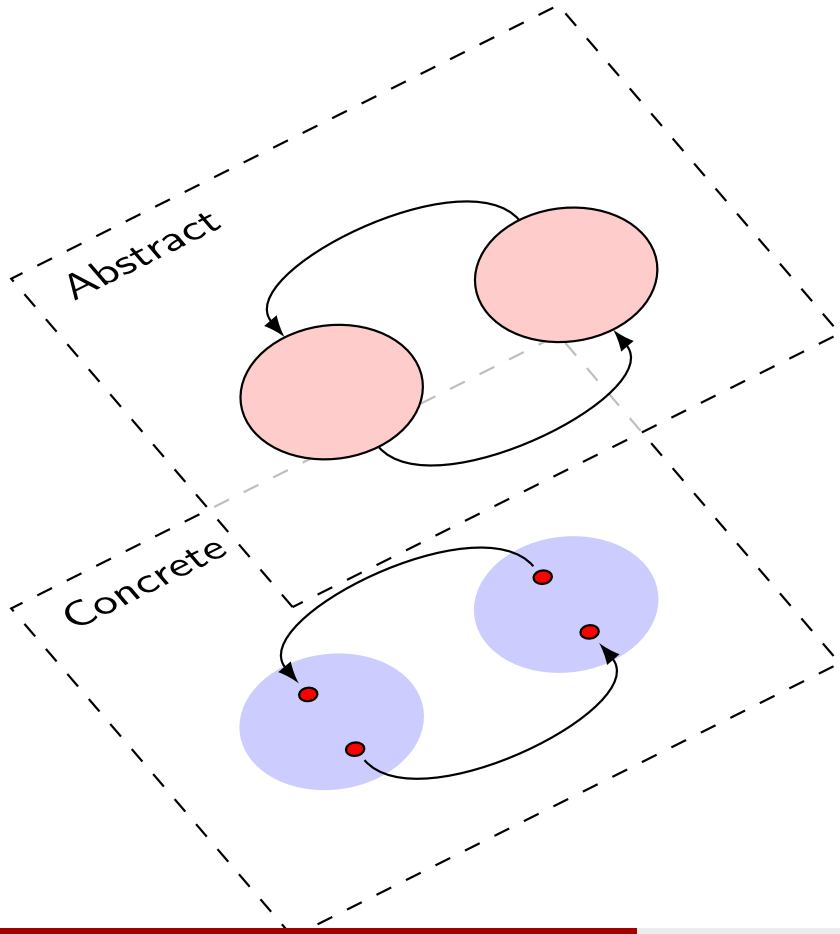
Parametric abst. refinement — uniformly spurious paths

Classic case:

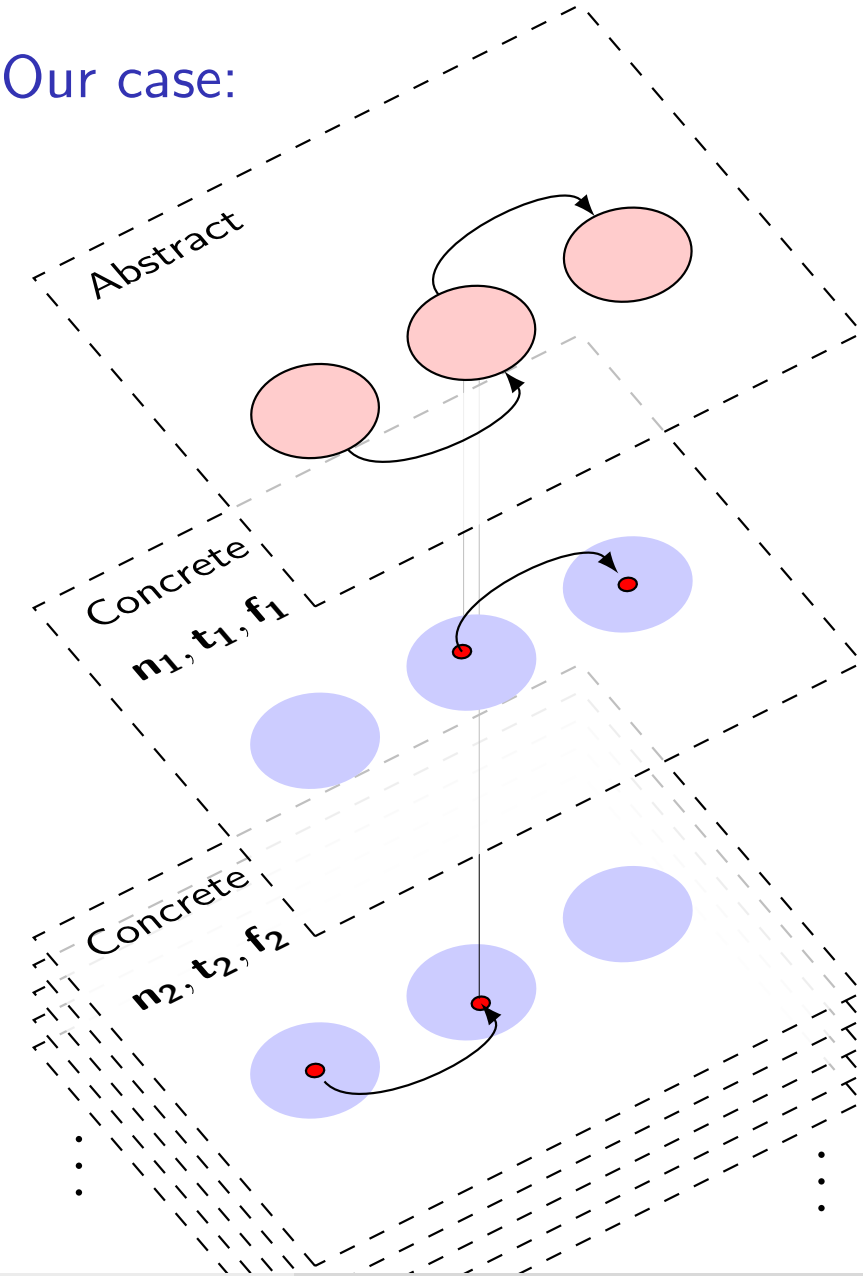


Parametric abst. refinement — uniformly spurious paths

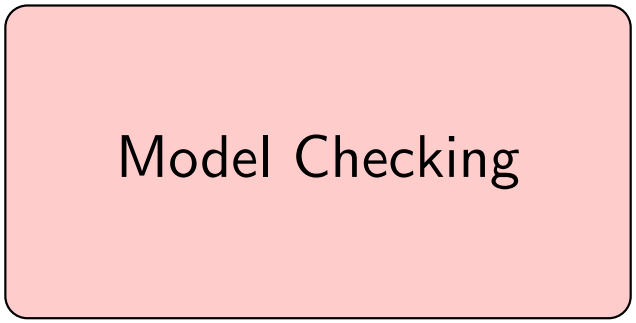
Classic case:



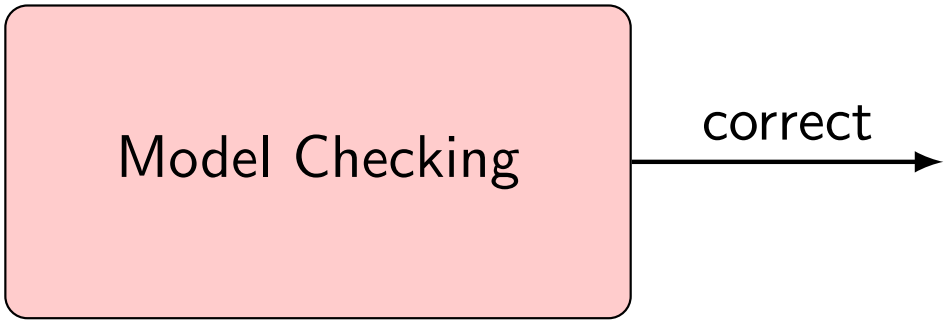
Our case:



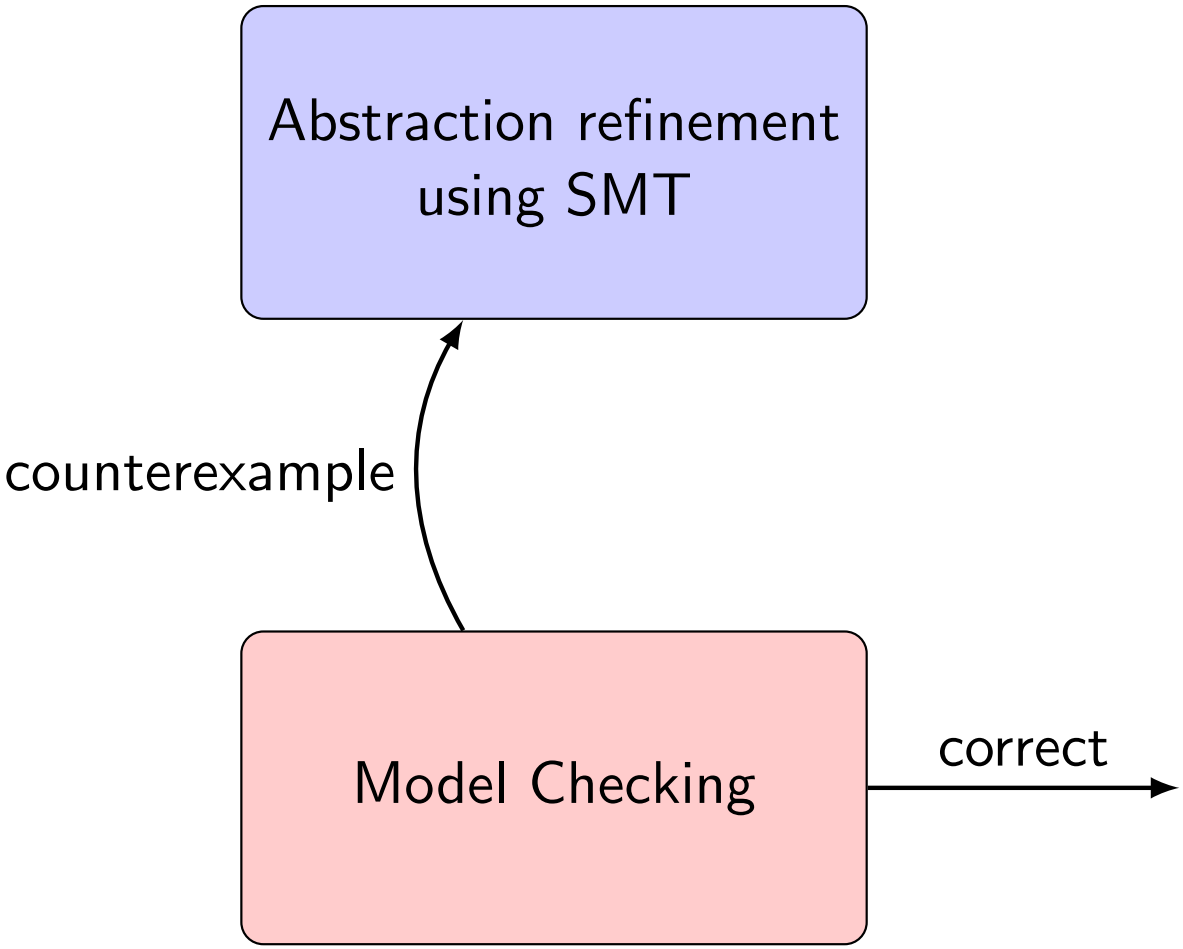
CEGAR — automated workflow



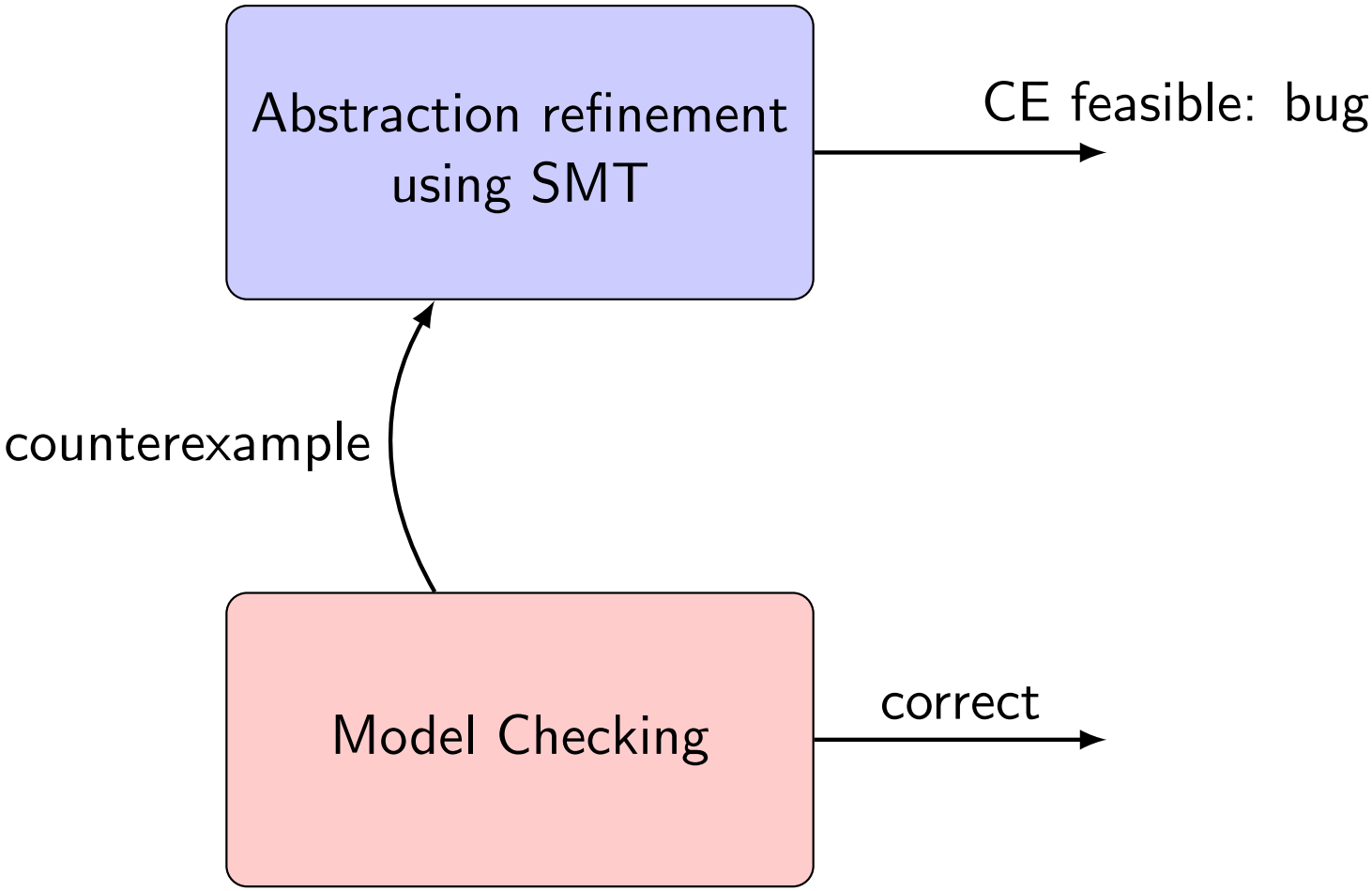
CEGAR — automated workflow



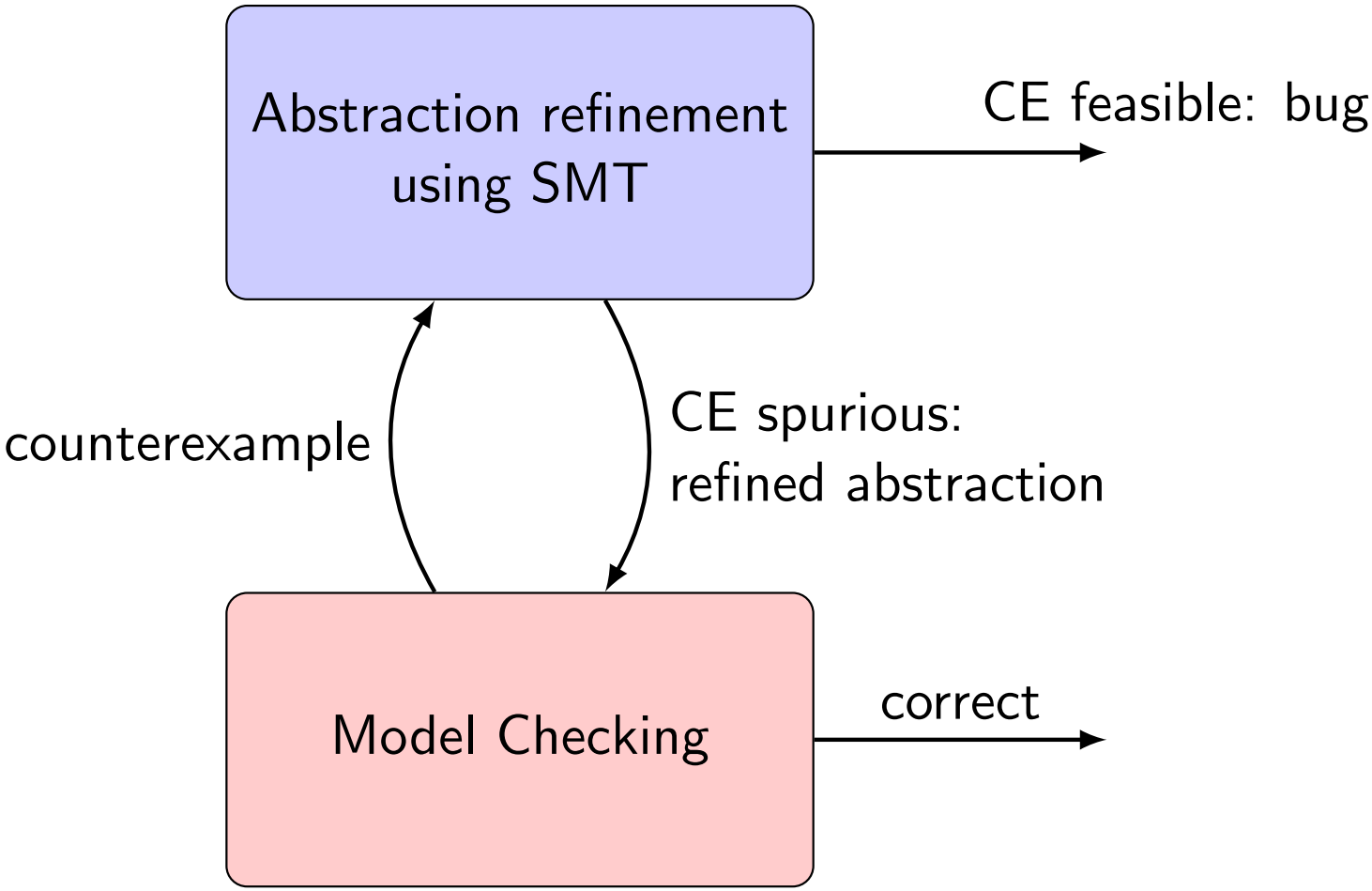
CEGAR — automated workflow



CEGAR — automated workflow



CEGAR — automated workflow



What is SMT?

recall SAT:

- given a Boolean formula, e.g., $(\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d \vee e)$
- is there an assignment of TRUE and FALSE to variables a, b, c, d, e such that the formula evaluates to TRUE?

What is SMT?

recall SAT:

- given a Boolean formula, e.g., $(\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d \vee e)$
- is there an assignment of TRUE and FALSE to variables a, b, c, d, e such that the formula evaluates to TRUE?

Satisfiability Modulo Theories (SMT) :

- here just linear arithmetics
- given a formula, e.g.,

$$x = y \ \wedge \ y = z \wedge u \neq x \wedge (x + y \leq 1 \wedge 2x + y = 1) \vee 3x + 2y \geq 3$$

- is there an assignment of values to u, x, y, z such that formula evaluates to TRUE?
- practically efficient tools: YICES, Z3

Counter example: losing processes

$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 0$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 1$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 2$$

Counter example: losing processes

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 1$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 2$

encode last state in SMT formula:

Resilience condition	$n > 3t \wedge t \geq f \wedge f \geq 0$
non-zero counters	$n - t \leq k[4] \wedge t + 1 \leq k[8] \wedge k[8] < n - t$
zero counters	for $i \in \{0, \ldots 15\} \setminus \{4, 8\}$: $k[i] = 0$
system size	$n - f = k[0] + k[1] + \cdots + k[15]$
msgs sent	$t + 1 \leq nsnt \wedge nsnt < n - t$

Counter example: losing processes

$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\},\ nsnt = 0$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\},\ nsnt = 1$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\},\ nsnt = 2$$

encode last state in SMT formula:

Resilience condition	$n > 3t \wedge t \geq f \wedge f \geq 0$
non-zero counters	$n - t \leq k[4] \wedge t + 1 \leq k[8] \wedge k[8] < n - t$
zero counters	for $i \in \{0, \ldots 15\} \setminus \{4, 8\}$: $k[i] = 0$
system size	$n - f = k[0] + k[1] + \cdots + k[15]$
msgs sent	$t + 1 \leq nsnt \wedge nsnt < n - t$

This provides one large formula T...

Remove transitions

- We ask the SMT solver: is there a satisfiable assignment for T?

Remove transitions

- We ask the SMT solver: is there a satisfiable assignment for T?
- if yes,
  then state is OK, may be part of a real counterexample

Remove transitions

- We ask the SMT solver: is there a satisfiable assignment for T?
- if yes,
  then state is OK, may be part of a real counterexample
- if not, then state is spurious
  - we can remove transitions to that state in the abstract system
    - in fact: unsatisfiable core to remove multiple transitions at once

Coming back to our example

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 1$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 2$

encode last state in SMT formula:

Resilience condition	$n > 3t \wedge t \geq f \wedge f \geq 0$
non-zero counters	$n - t \leq k[4] \wedge t + 1 \leq k[8] \wedge k[8] < n - t$
zero counters	for $i \in \{0, \ldots 15\} \setminus \{4, 8\}$: $k[i] = 0$
system size	$n - f = k[0] + k[1] + \cdots + k[15]$
msgs sent	$t + 1 \leq nsnt \wedge nsnt < n - t$

This provides one large formula T...
contradiction with counters and system size

Counterexample type: losing messages

$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\},\ nsnt = 0$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\},\ nsnt = \mathbf{1}$$
$$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\},\ nsnt = \mathbf{1}$$

- no contradiction within a state

Counterexample type: losing messages

$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 0$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$$
$$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$$

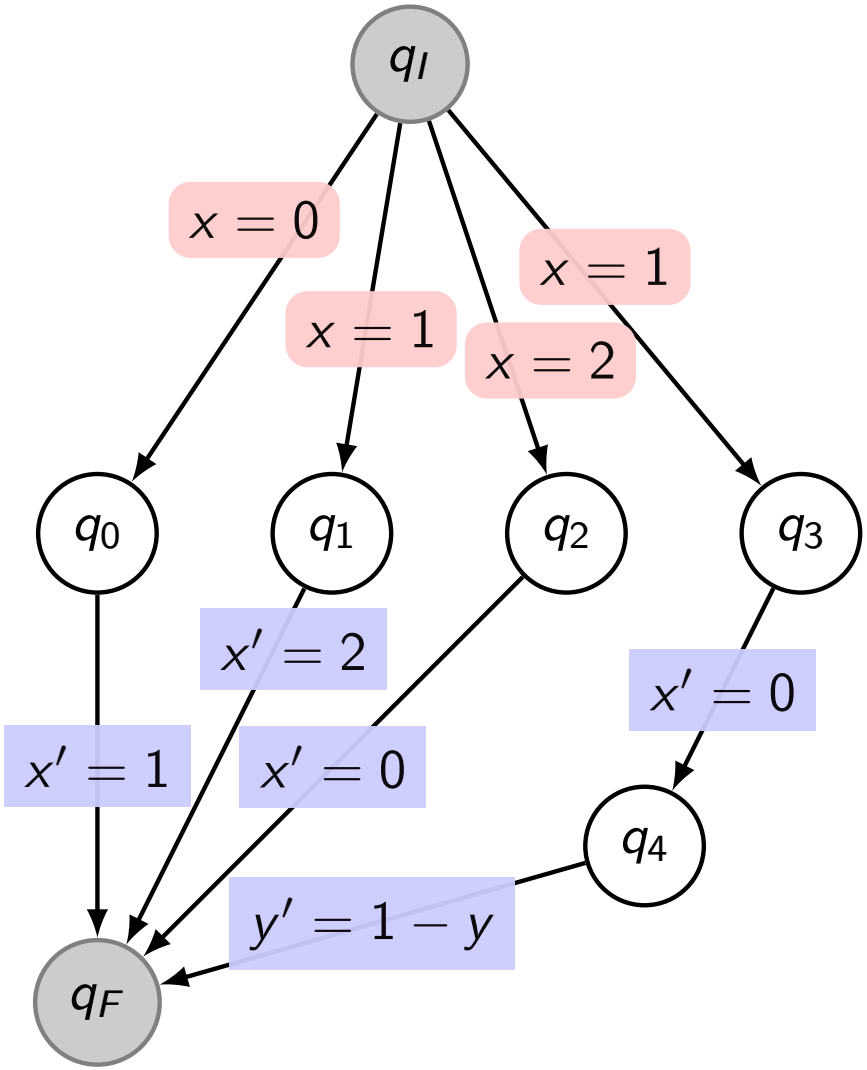
- no contradiction within a state
- when a process sends a message it
  - goes to SE
  - increases nsnt

Counterexample type: losing messages

$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 0$$
$$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$$
$$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$$

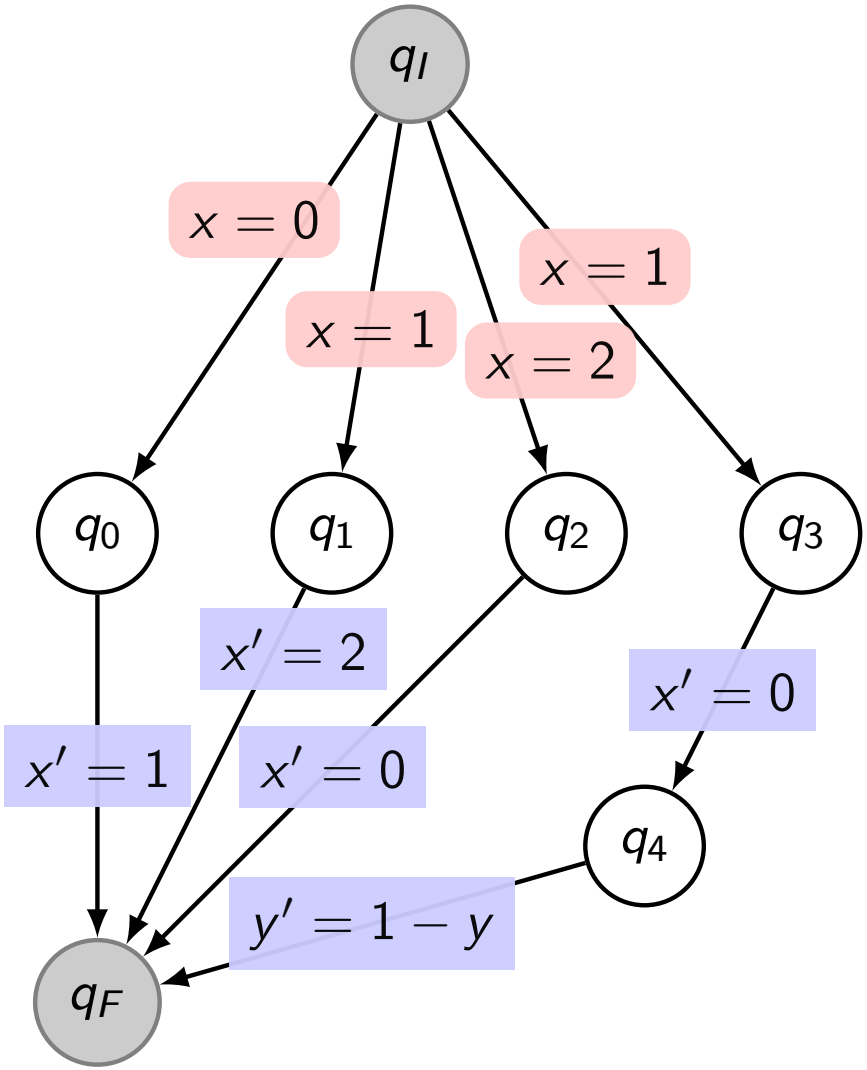
- no contradiction within a state
- when a process sends a message it
  - goes to SE
  - increases nsnt
  - the human sees a correlation
  - the machine has to reason about steps...

Encoding a step



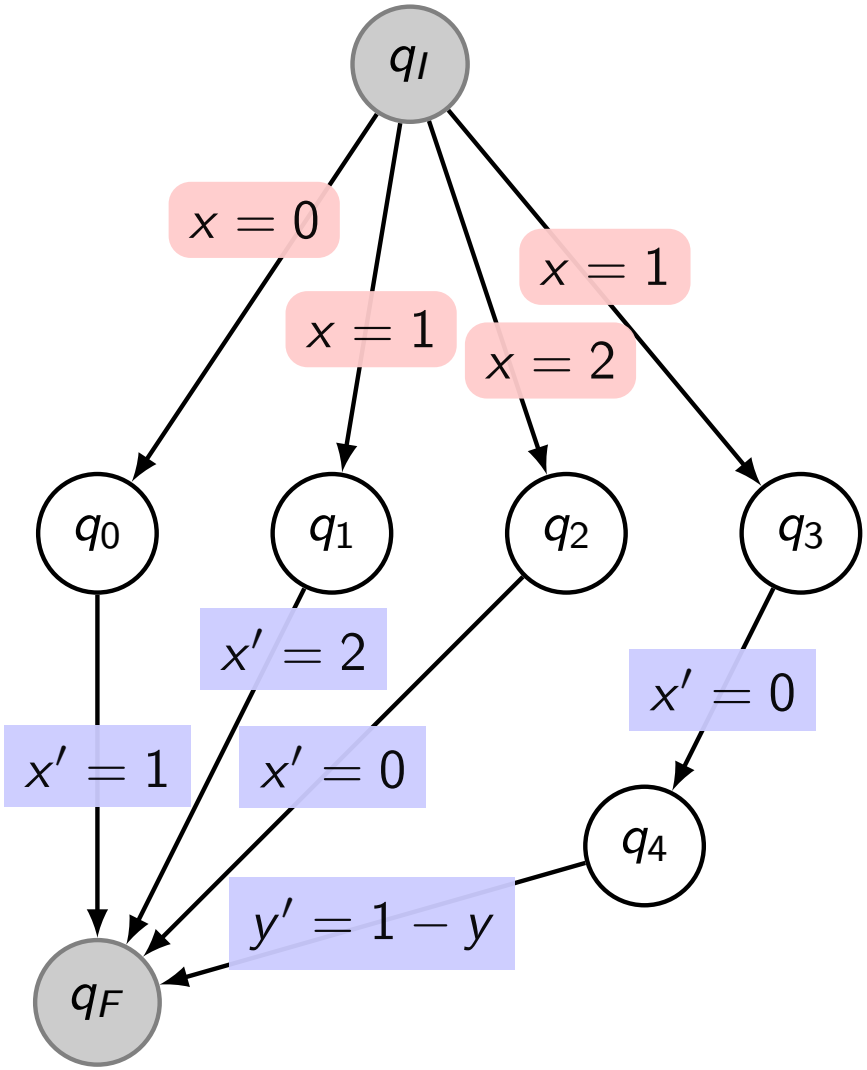
$$step(x, x') =$$

Encoding a step



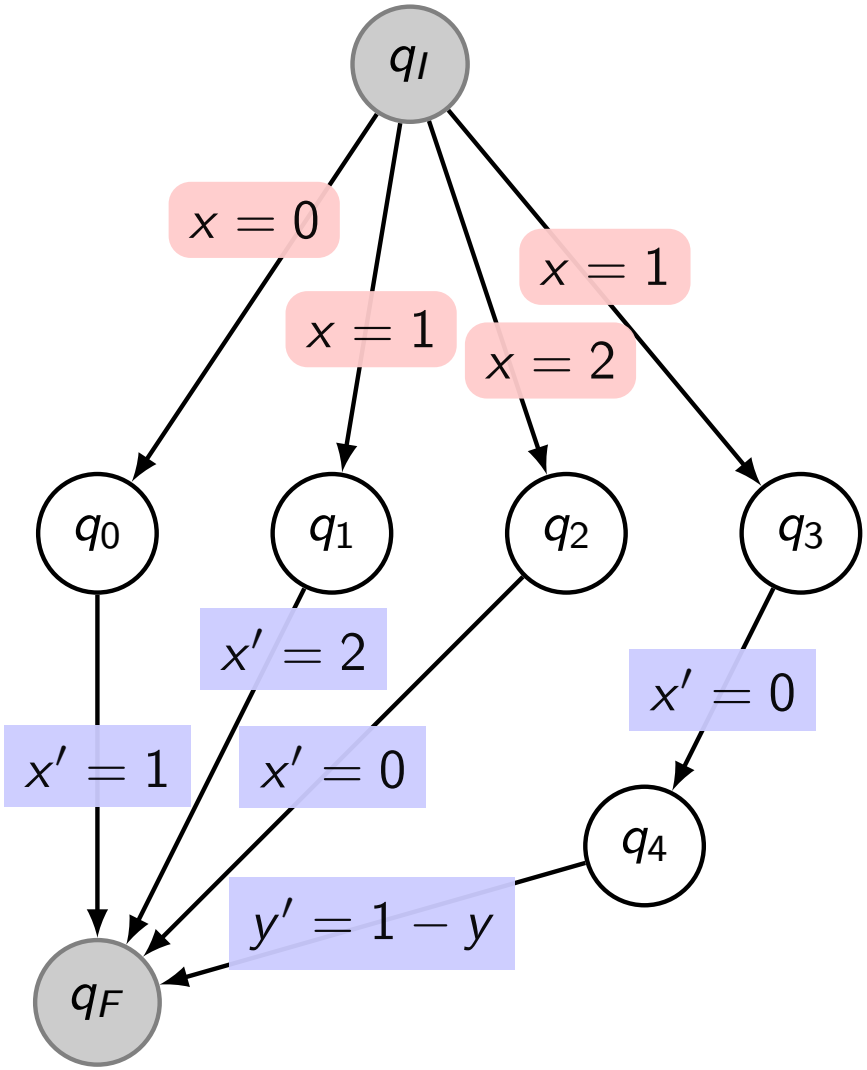
$$step(x, x') =$$
$$(x = 0 \wedge x' = 1)$$

Encoding a step



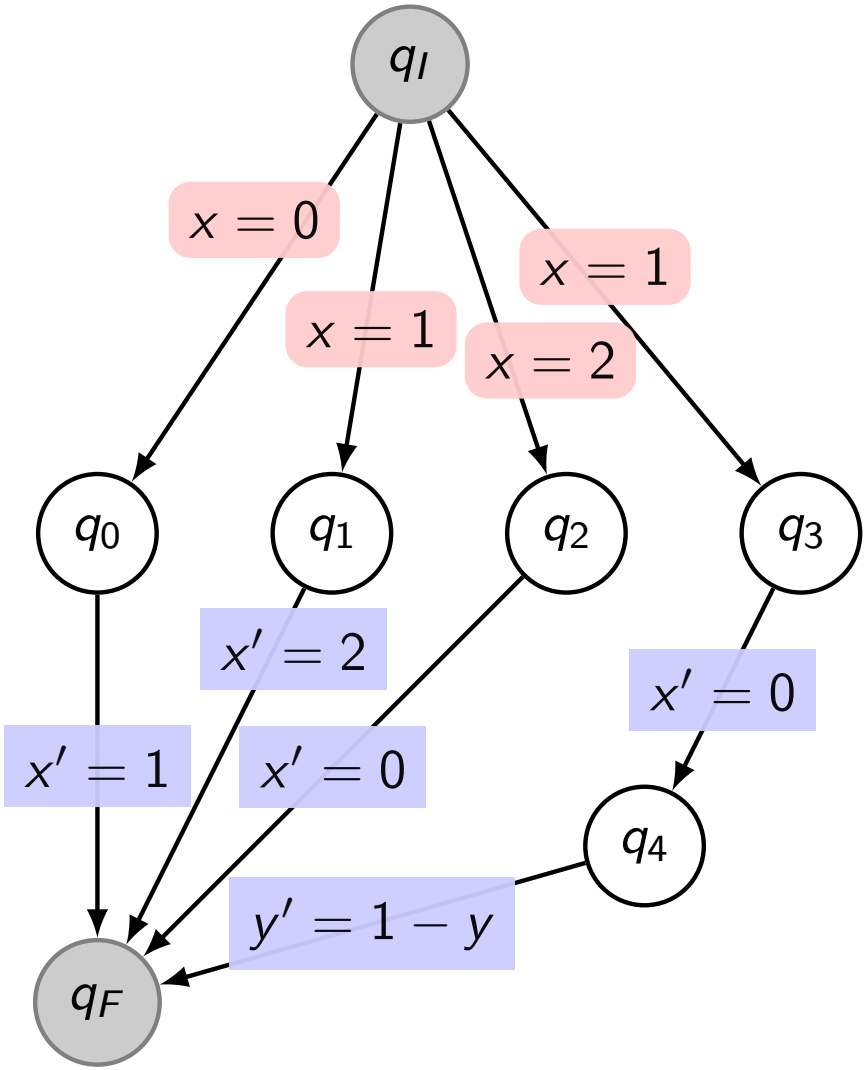
$$step(x, x') =$$
$$(x = 0 \wedge x' = 1)$$
$$\vee$$
$$(x = 1 \wedge x' = 2)$$

Encoding a step



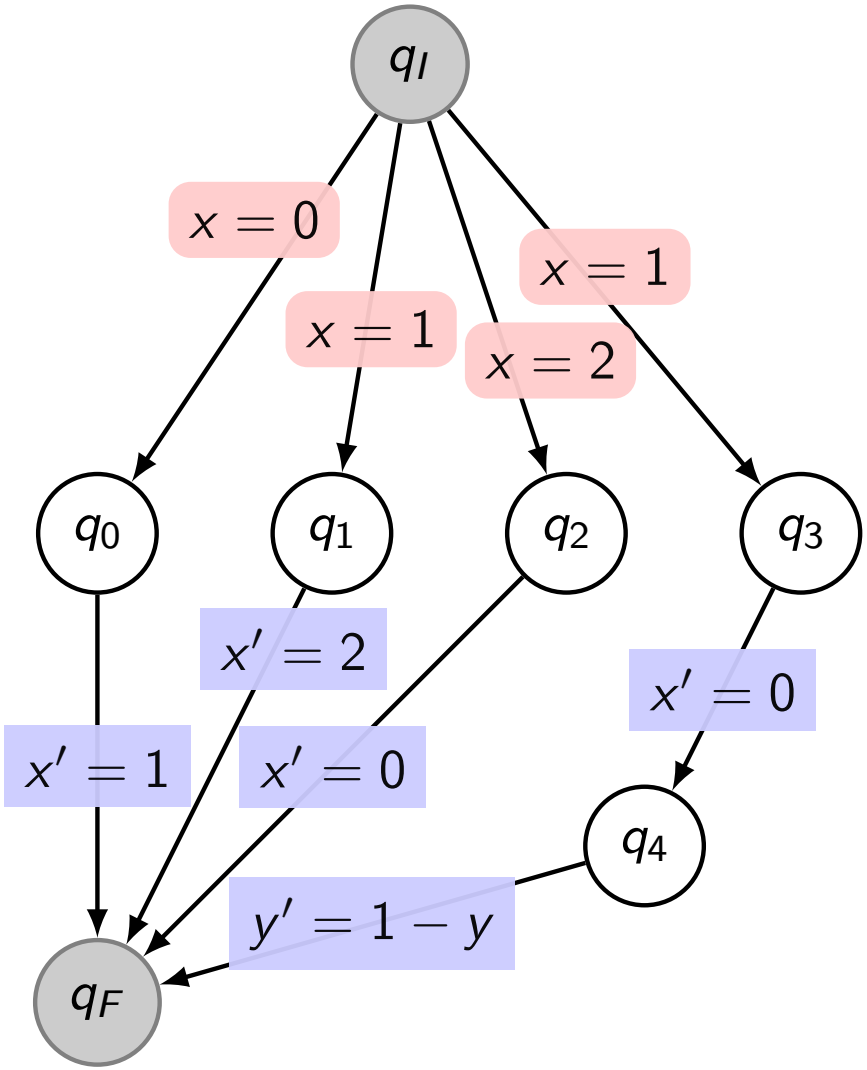
$$step(x, x') =$$
$$(x = 0 \wedge x' = 1)$$
$$\vee$$
$$(x = 1 \wedge x' = 2)$$
$$\vee$$
$$(x = 2 \wedge x' = 0)$$

Encoding a step



$$
\begin{aligned}
&step(x, x') = \\
&(x = 0 \wedge x' = 1) \\
&\vee \\
&(x = 1 \wedge x' = 2) \\
&\vee \\
&(x = 2 \wedge x' = 0) \\
&\vee \\
&(x = 1 \wedge x' = 0 \wedge y' = 1 - y)
\end{aligned}
$$

Encoding a step



$$
\begin{aligned}
&step(x, x') = \\
&(x = 0 \wedge x' = 1) \\
&\vee \\
&(x = 1 \wedge x' = 2) \\
&\vee \\
&(x = 2 \wedge x' = 0) \\
&\vee \\
&(x = 1 \wedge x' = 0 \wedge y' = 1 - y)
\end{aligned}
$$

. . . that is a simplification of what is going on in the tool. . .

Counterexample type: losing messages

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$
$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$

- check whether $(state2(x) \wedge step(x, x')) \rightarrow state3(x')$ is satisfiable

Counterexample type: losing messages

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = \mathbf{1}$
$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = \mathbf{1}$

- check whether $(state2(x) \wedge step(x, x')) \rightarrow state3(x')$ is satisfiable
- surprisingly it is
  - the computer cannot disregard it
  - and it cannot refine it
  - ask the user!

Counterexample type: losing messages

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$
$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}, \; nsnt = \mathbf{1}$

- check whether $(state2(x) \wedge step(x, x')) \rightarrow state3(x')$ is satisfiable
- surprisingly it is
  - the computer cannot disregard it
  - and it cannot refine it
  - ask the user!
- let experts stare at counter examples...
- expert: it is spurious
  formula captures local view, not whole execution
- add global knowledge using invariants.

Invariant candidates — soundness

- given in invariant candidate *inv*
- we want to check whether it is an invariant, i.e., whether

$$\neg \left((inv(x) \wedge step(x, x')) \rightarrow inv(x') \right)$$

  is satisfiable

Invariant candidates — soundness

- given in invariant candidate *inv*
- we want to check whether it is an invariant,
  i.e., whether

$$\neg \left((inv(x) \wedge step(x, x')) \rightarrow inv(x') \right)$$

  is satisfiable
- if not: we have an invariant
- we can add it to the state formula:
  $(state2(x) \wedge inv(x) \wedge step(x, x')) \rightarrow (state3(x') \wedge inv(x'))$
- and check for satisfiability

For our example

Counter example: losing messages

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = \mathbf{1}$
$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = \mathbf{1}$

the invariant candidate that was sufficient for refinement:

the number of messages sent

=

the number of processes who have sent messages

$$nsnt = k[8] + k[9] + \ldots k[15]$$

For our example

Counter example: losing messages

$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = 0$
$k = \{0, 0, 0, 0, \mathbf{3}, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = \mathbf{1}$
$k = \{0, 0, 0, 0, \mathbf{2}, 0, 0, 0, \mathbf{2}, 0, 0, 0, 0, 0, 0, 0\}$, $nsnt = \mathbf{1}$

the invariant candidate that was sufficient for refinement:

the number of messages sent
=
the number of processes who have sent messages

$$nsnt = k[8] + k[9] + \ldots k[15]$$

a triviality for experts ...

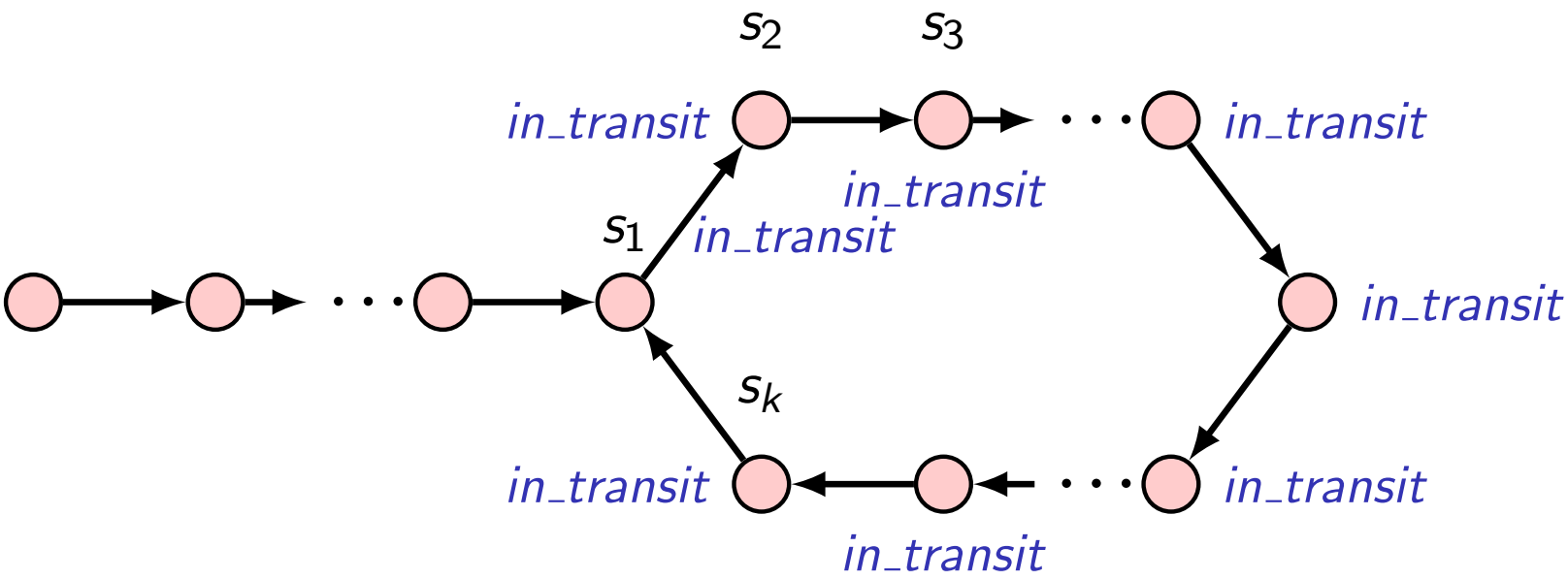
Liveness

- distributed algorithm requires reliable communication
- every message sent is eventually received
- $\neg in\_transit \equiv [\forall i.\ recv_i \geq sent]$
- justice $\mathbf{G}\,\mathbf{F}\,\neg in\_transit$ necessary to verify liveness

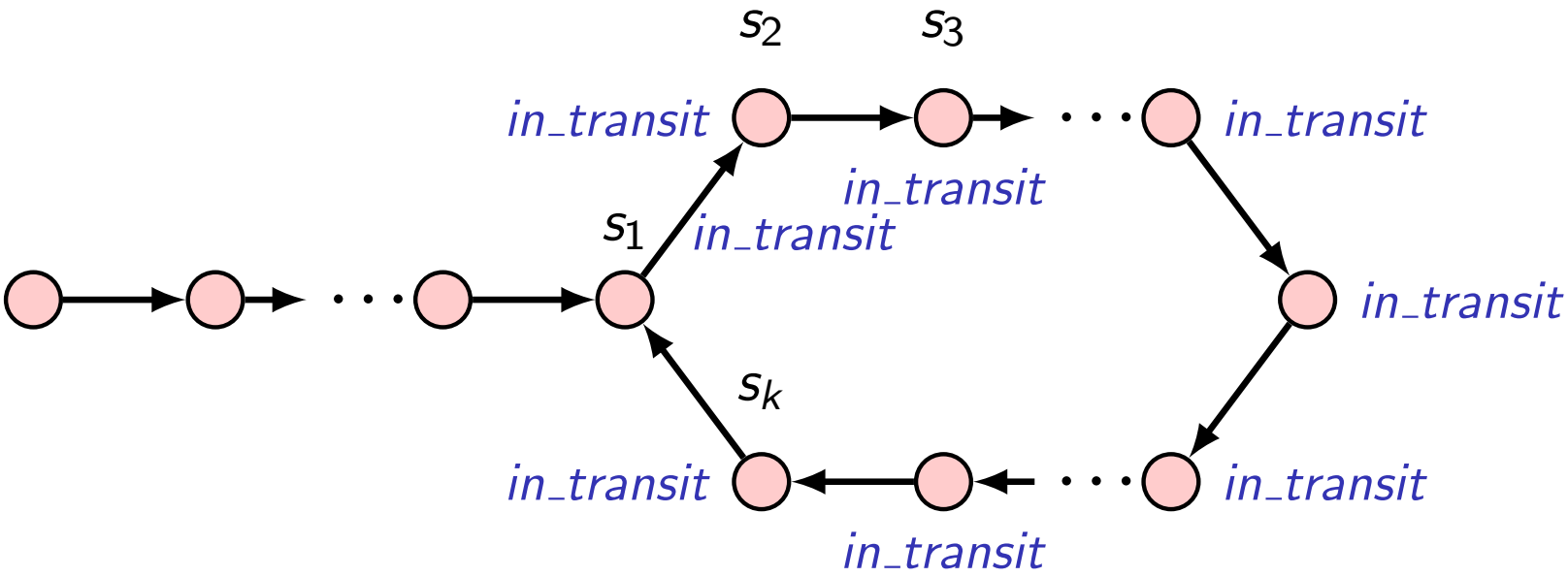
Liveness

- distributed algorithm requires reliable communication
- every message sent is eventually received
- $\neg in\_transit \equiv [\forall i.\ recv_i \geq sent]$
- justice $\mathbf{G}\,\mathbf{F}\,\neg in\_transit$ necessary to verify liveness

counter example (lasso):

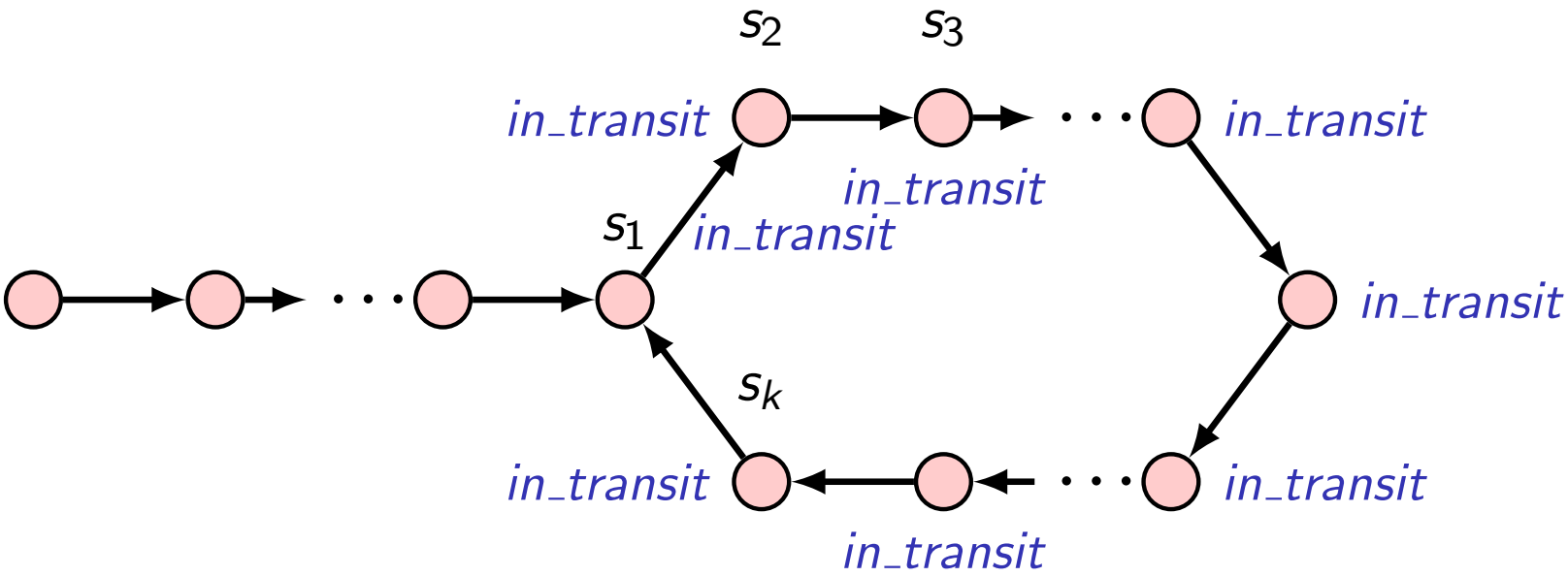


Liveness — justice suppression



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

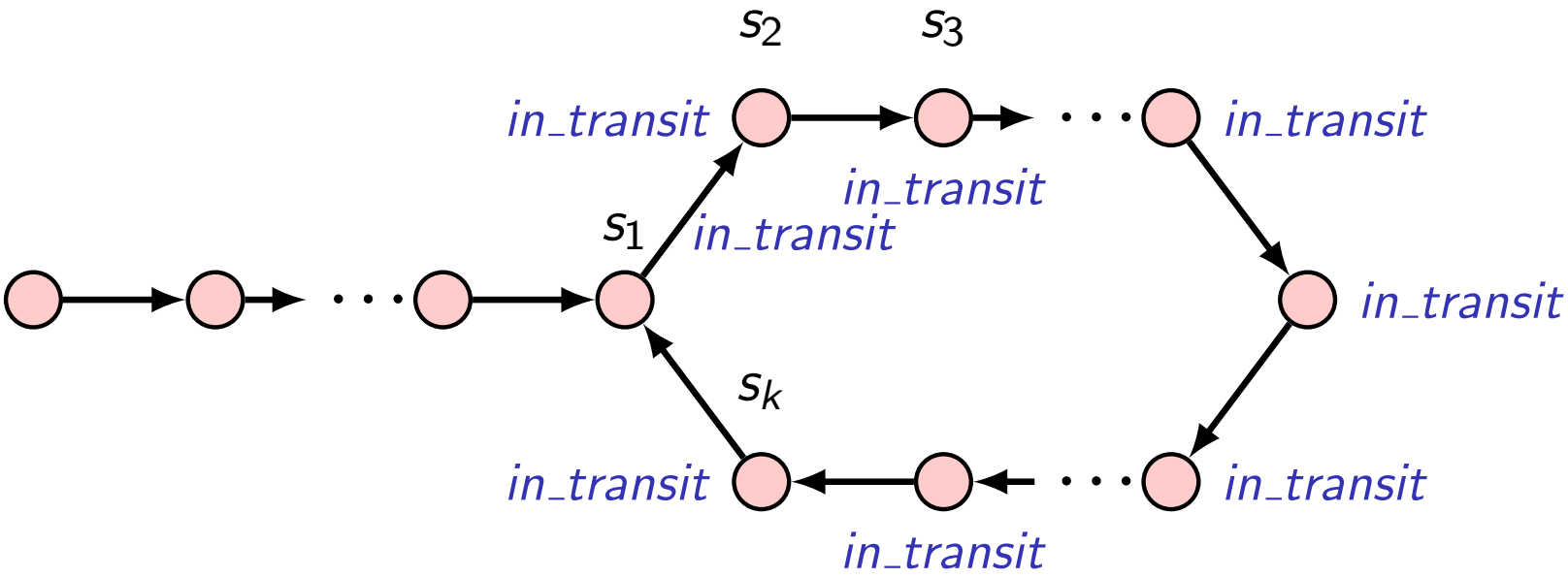
Liveness — justice suppression



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

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

Liveness — justice suppression



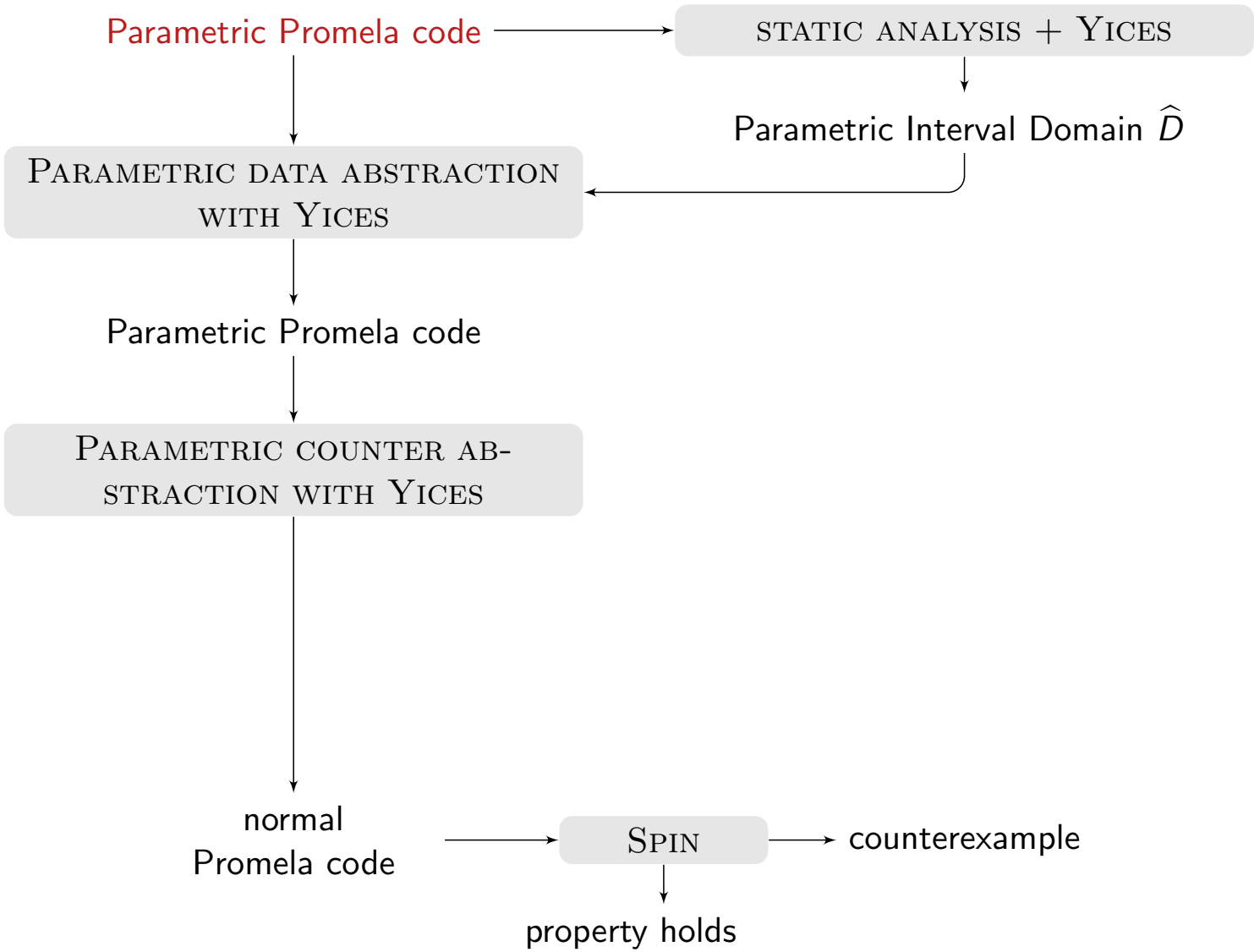
if $\forall j$ all concretizations of s_j violate $\neg$*in_transit*, then CE is spurious.

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

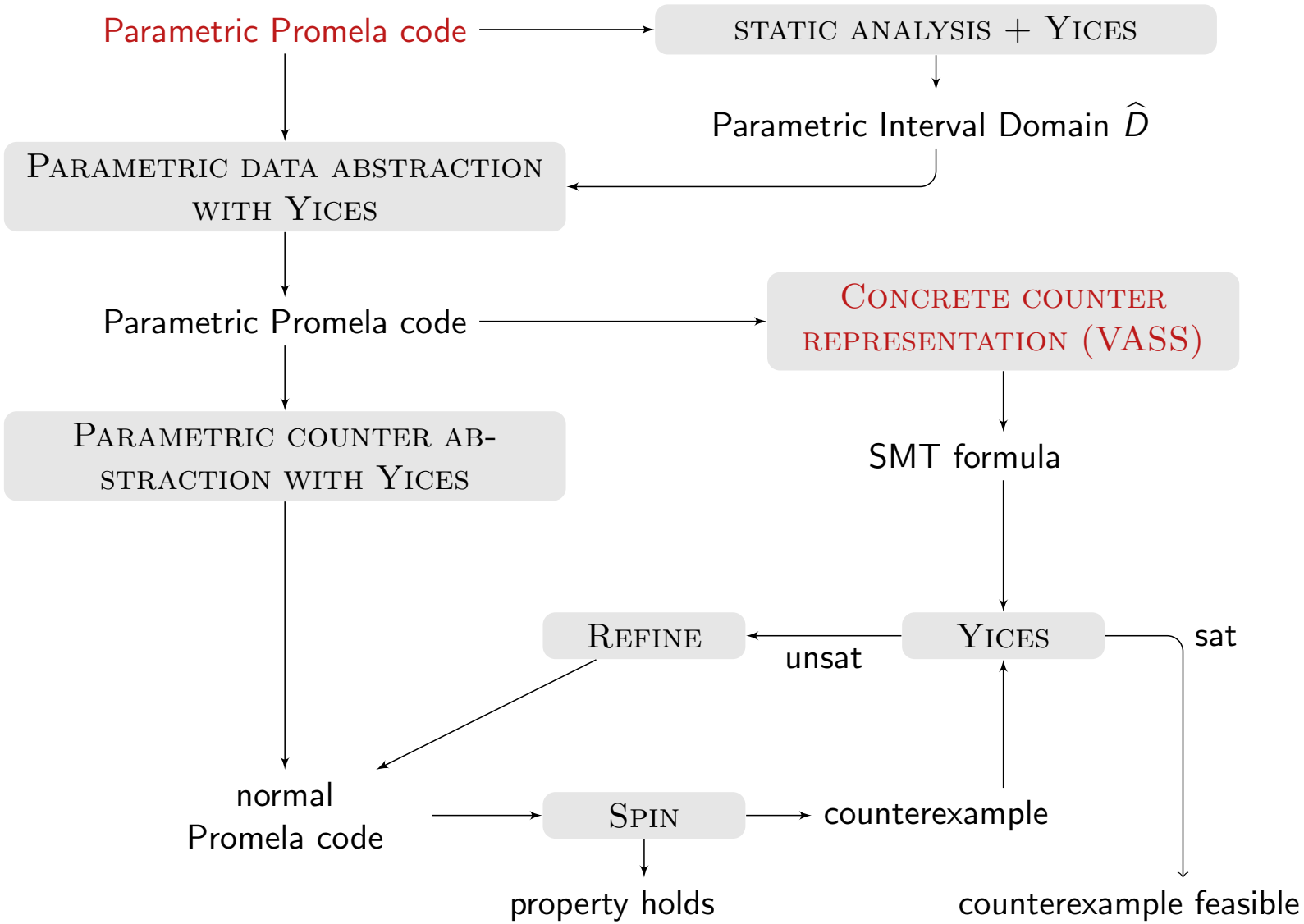
. . . we use unsat cores to refine several loops at once

the implementation

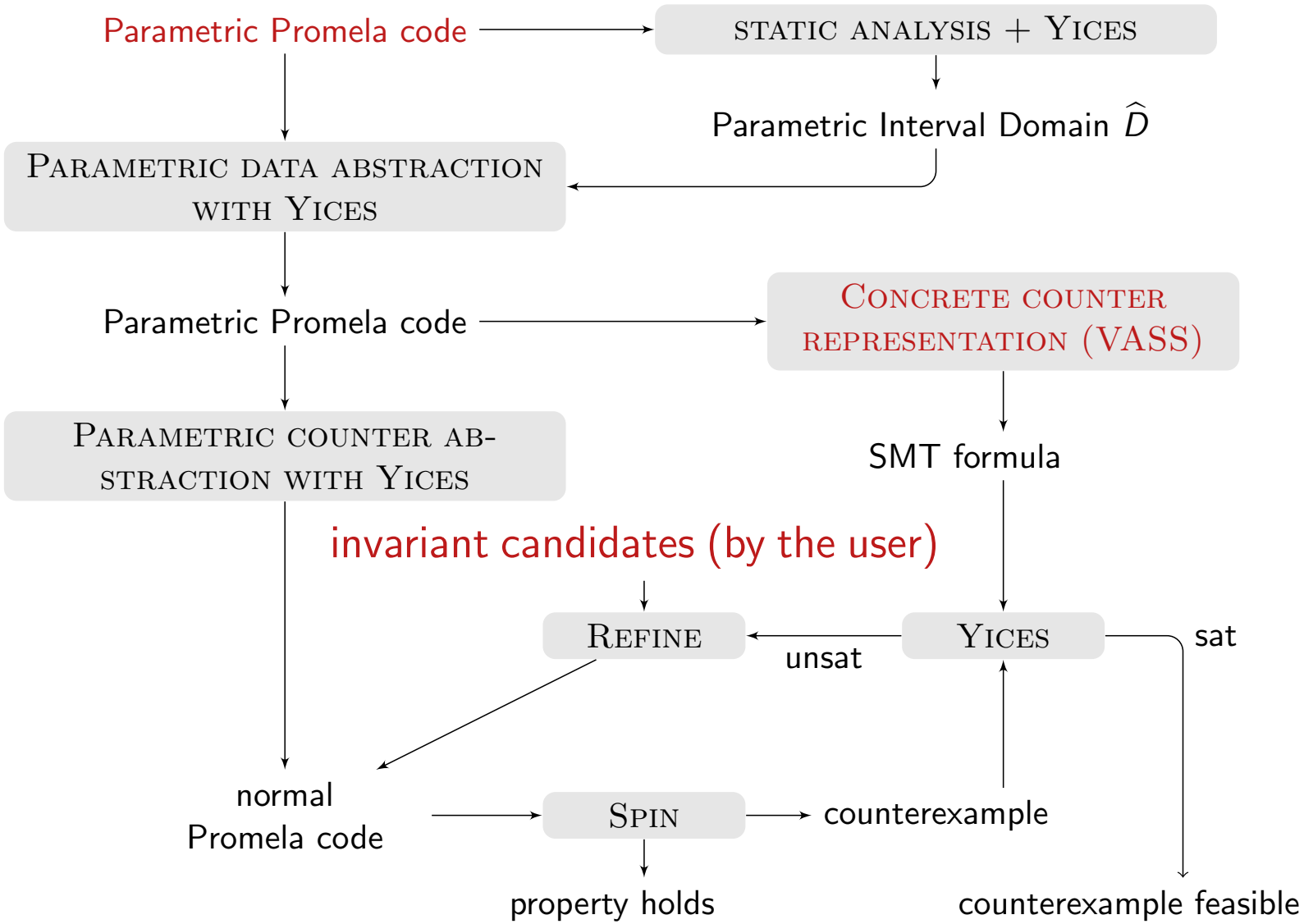
Tool Chain: ByMC



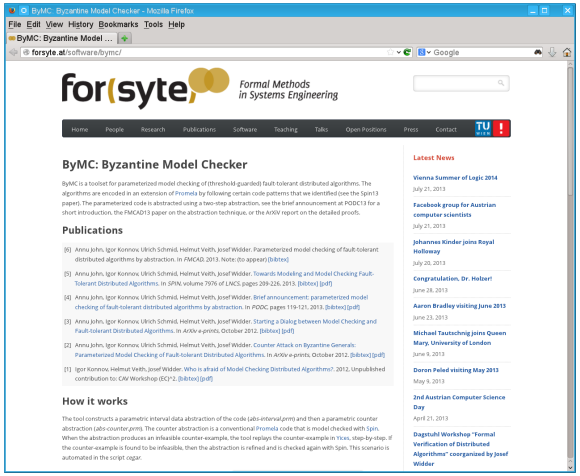
Tool Chain: ByMC

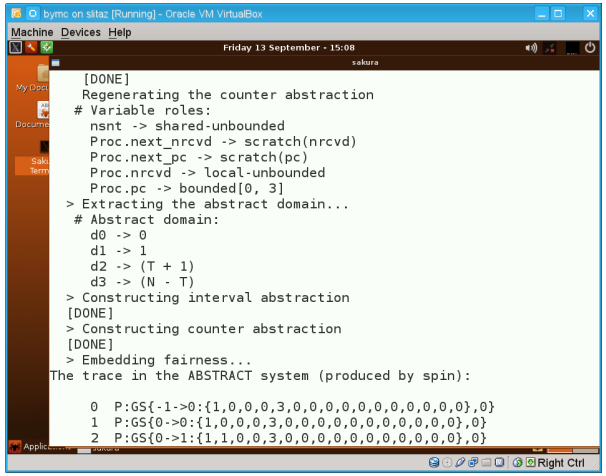


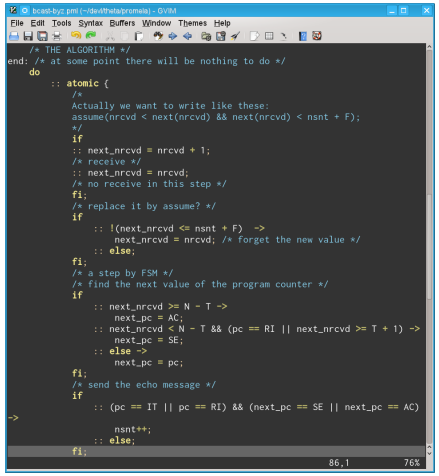
Tool Chain: ByMC



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

Running the tool — concrete case

- user specifies parameter value
- useful to check whether the code behaves as expected
- `$bymc/verifyco-spin "N=4,T=1,F=1" bcast-byz.pml relay`
  - model checking problem in directory "`./x/spin-bcast-byz-relay-N=4,T=1,F=1`"
  - in `concrete.prm`
    - parameters are replaced by numbers
    - process prototype is replaced with $N - F = 3$ active processes

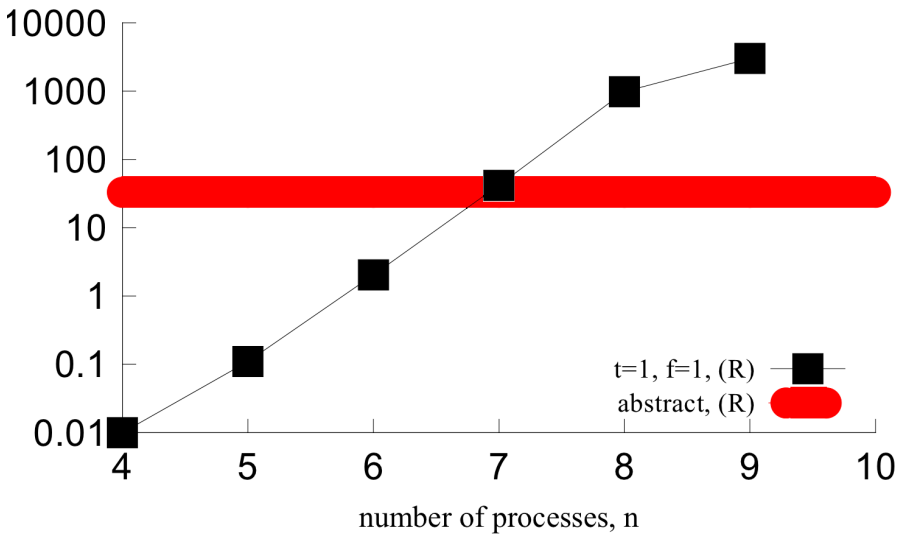
Running the tool — parameterized model checking

- PIA data and counter abstraction
- finite-state model checking on abstract model
- `$bymc/verifypa-spin bcast-omit.pml relay`
  - model checking problem in directory "`./x/bcast-byz-relay-yymmdd-HHMM.*`"
  - directory contains
    - `abs-interval.prm`: result of the data abstraction;
    - `abs-counter.prm`: result of the counter abstraction;
    - `abs-vass.prm`: auxiliary abstraction for abstraction refinement;
    - `mc.out`: the last output by SPIN;
    - `cex.trace`: the counterexample (if there is one);
    - `yices.log`: communication log with YICES.

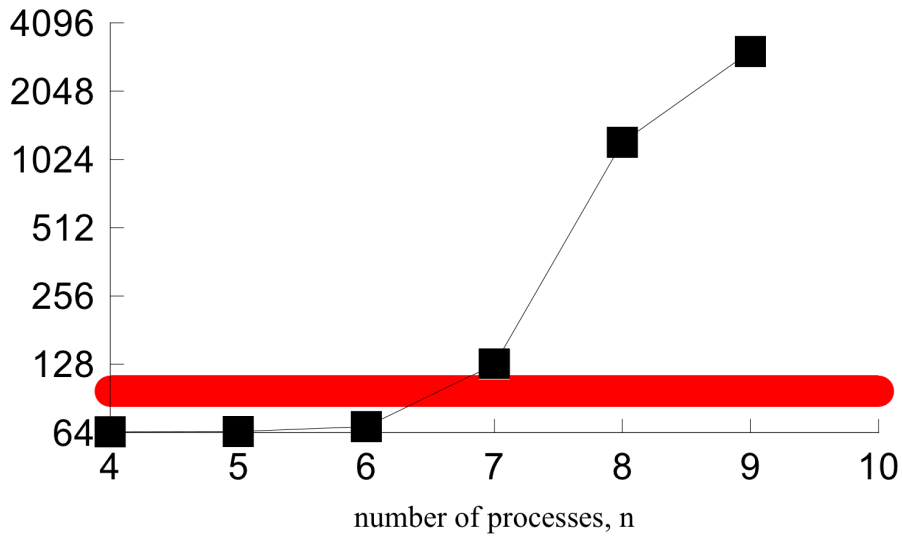
experimental evaluation

Concrete vs. parameterized (Byzantine case)

Time to check relay (sec, logscale)



Memory to check relay (MB, logscale)



- 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	$n \geq 2t \wedge f \leq t$	**U**	✓	0	1 sec.
ST87	Omit	$n \geq 2t \wedge f \leq t$	**C**	✗	0	2 sec.
ST87	Omit	$n \geq 2t \wedge f \leq t$	**R**	✗	0	2 sec.

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:

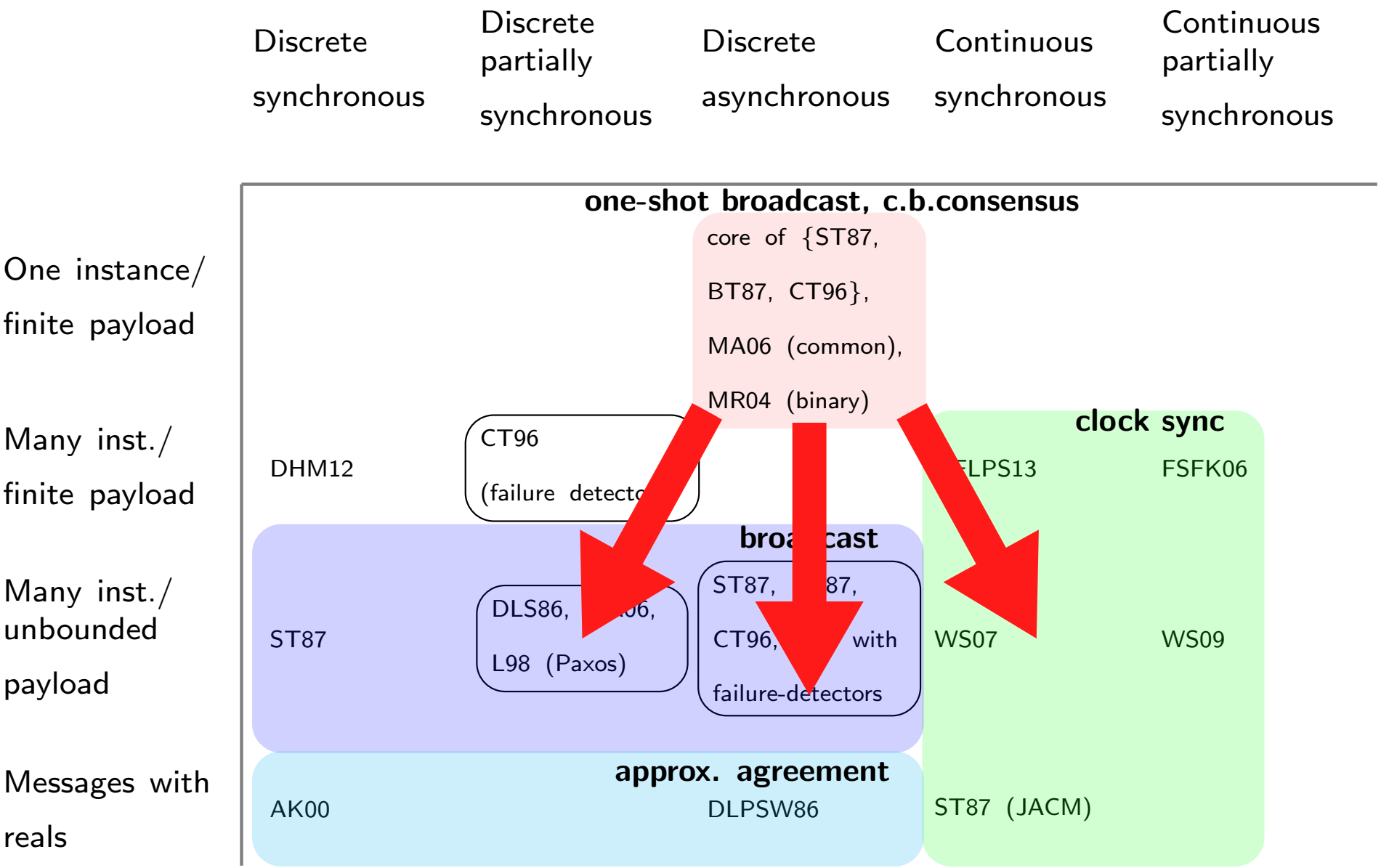
[Alberti, Ghilardi, Pagani, Ranise, Rossi 2010-2012]

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

Our current work

	Discrete synchronous	Discrete partially synchronous	Discrete asynchronous	Continuous synchronous	Continuous partially synchronous
One instance/ finite payload			**one-shot broadcast, c.b.consensus** 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



Formal
Reasoning
In
Distributed
Algorithms

http://vsl2014.at/frida/

Thank you!

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

Fairness, Refinement, and Invariants

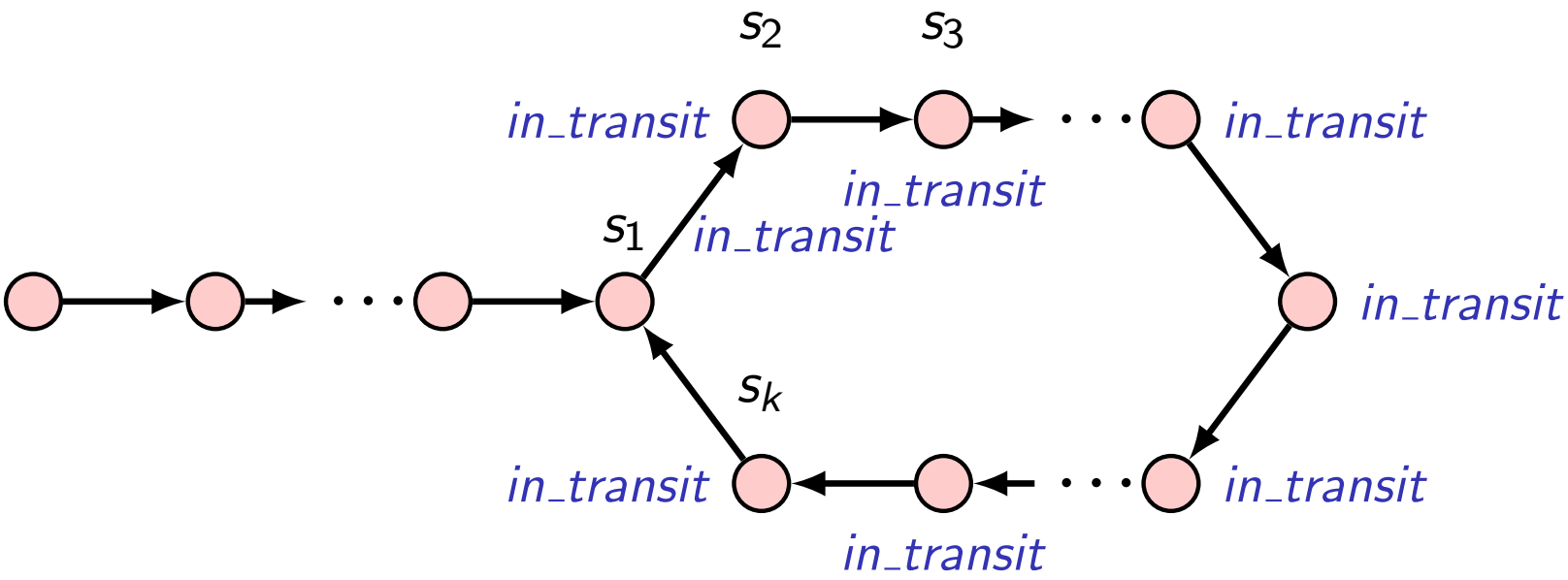
- In the Byzantine case we have $\mathit{in\_transit} : \forall i.\, (\mathit{recv}_i \geq \mathit{sent})$ and $\mathbf{G}\,\mathbf{F}\,\neg \mathit{in\_transit}$.
- In this case communication fairness implies computation fairness.
- But in the abstract version *sent* can deviate from the number of processes 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

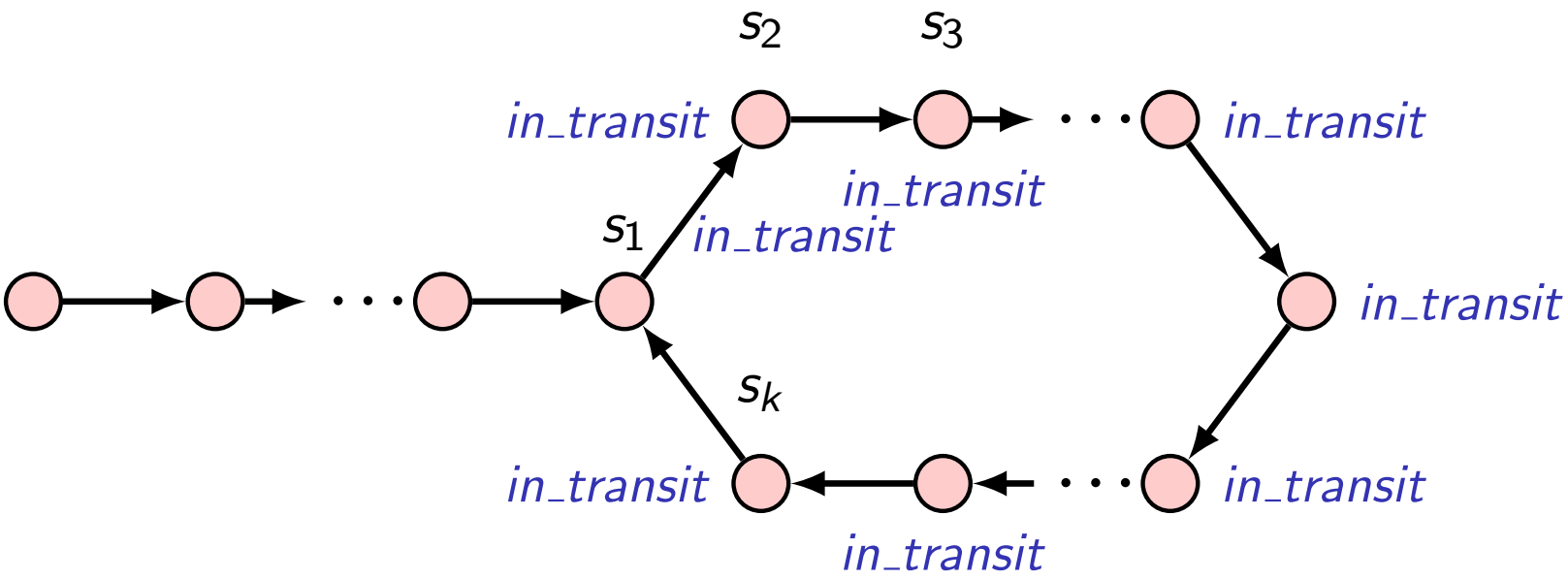
justice $\mathbf{G}\,\mathbf{F}\,\neg in\_transit$ necessary to verify liveness
counter example:



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

Parametric abstraction refinement — justice suppression

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

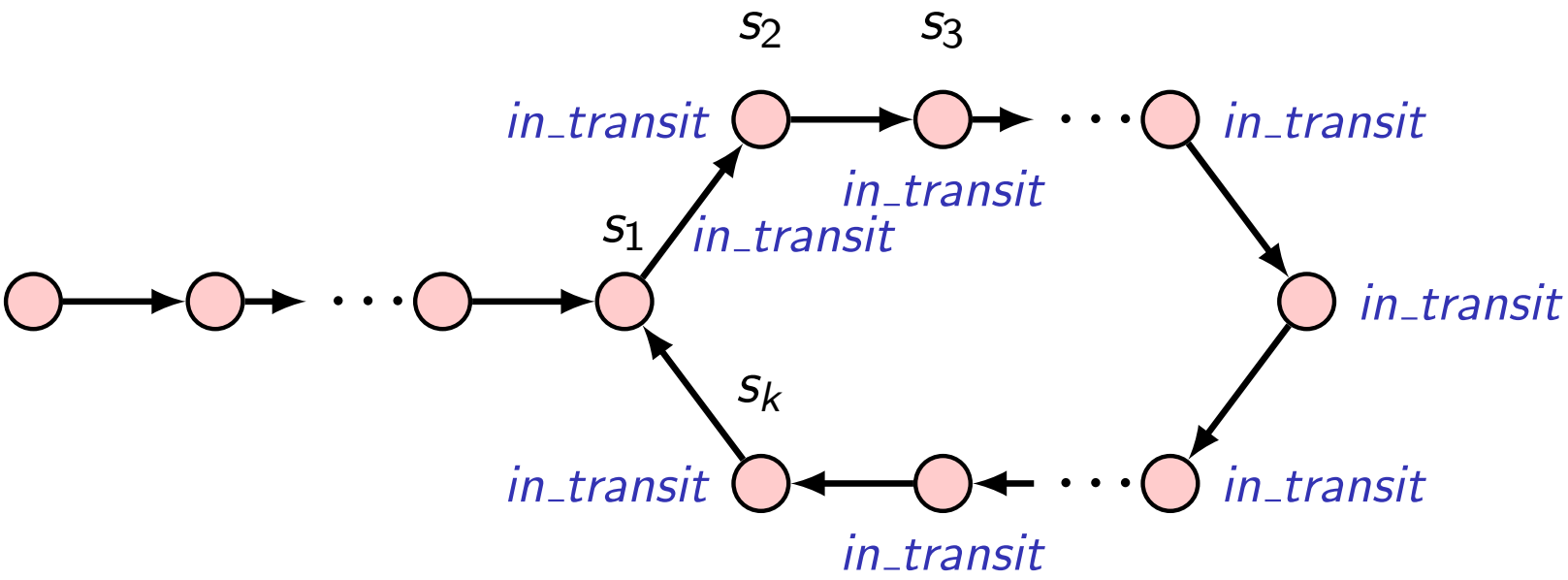


if $\forall j$ all concretizations of s_j violate $\neg in\_transit$, then CE is spurious.

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

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

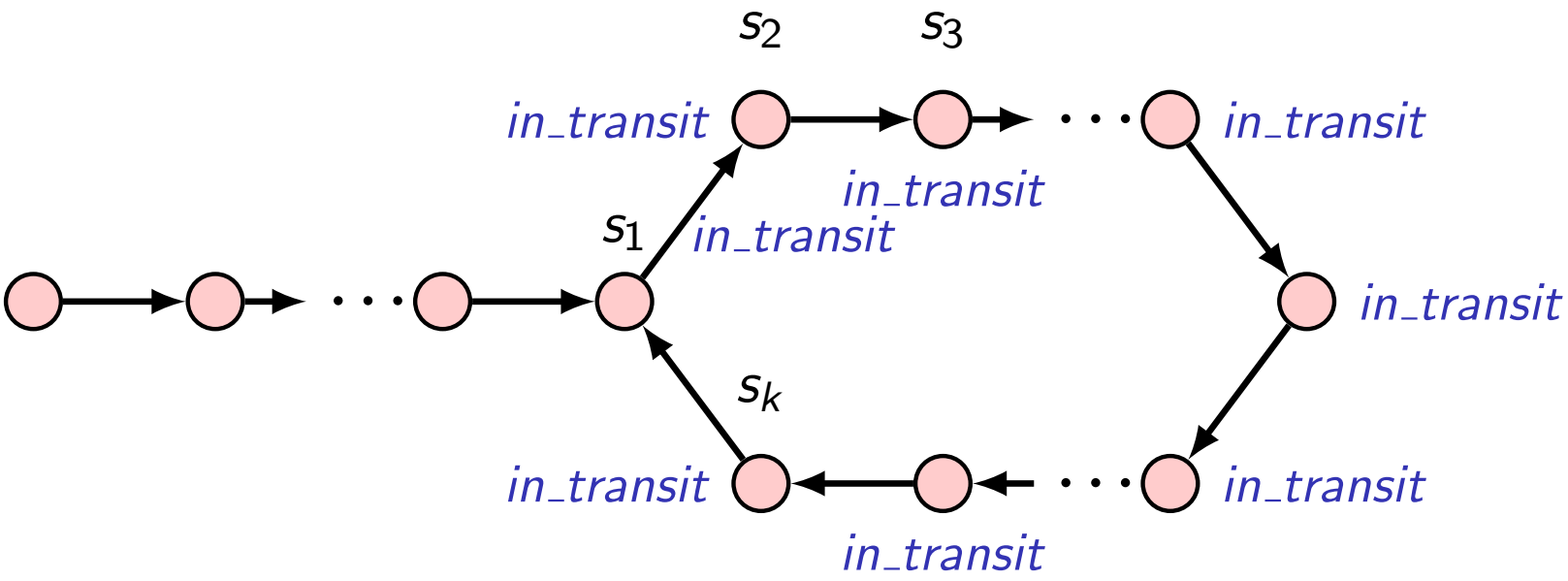
. . . we use unsat cores to refine several loops at once

Parametric abstraction refinement — justice suppression

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

Parametric abstraction refinement — justice suppression

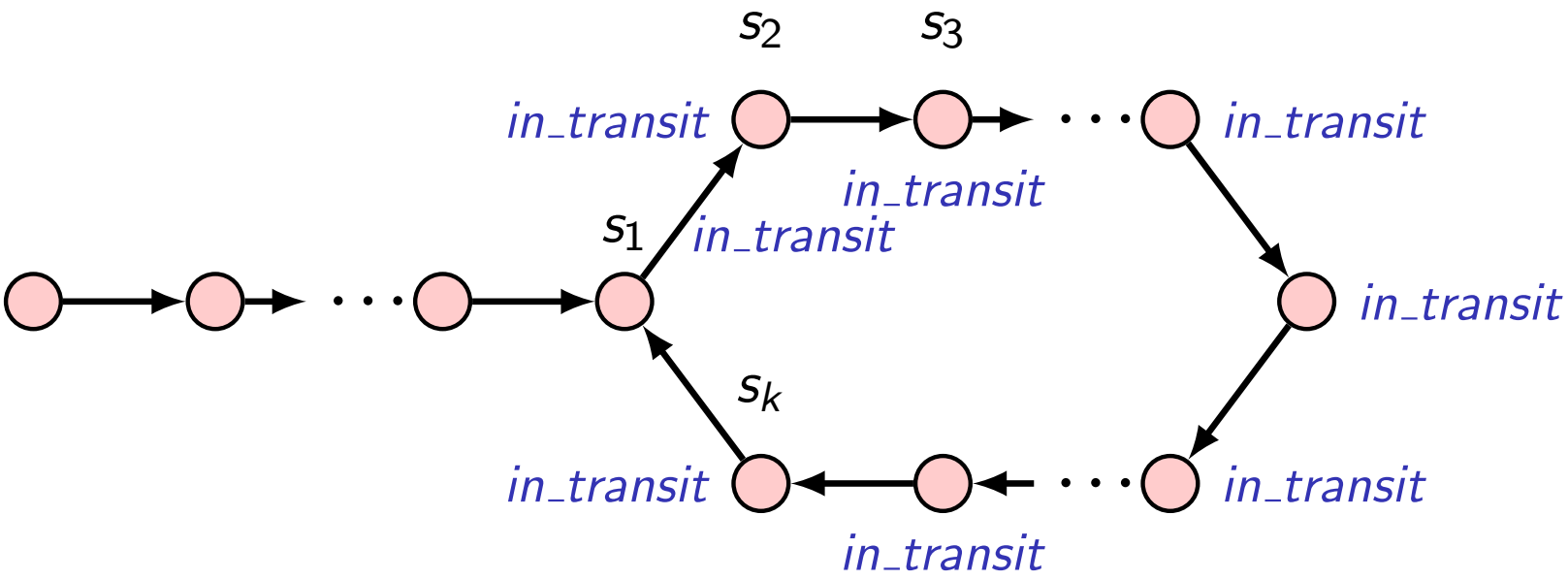
justice $\mathbf{G}\,\mathbf{F}\,\neg in\_transit$ necessary to verify liveness
counter example:



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

Parametric abstraction refinement — justice suppression

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

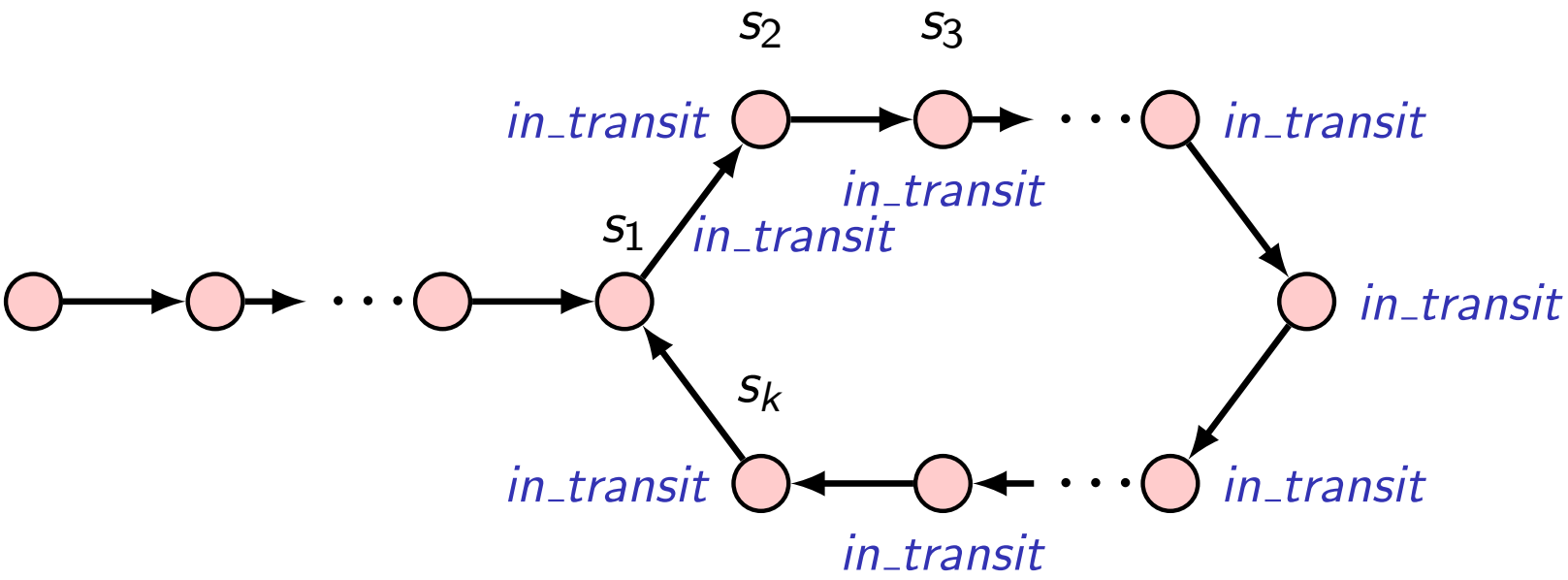


if $\forall j$ all concretizations of s_j violate $\neg in\_transit$, then CE is spurious.

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

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 $accept_i := 1$;

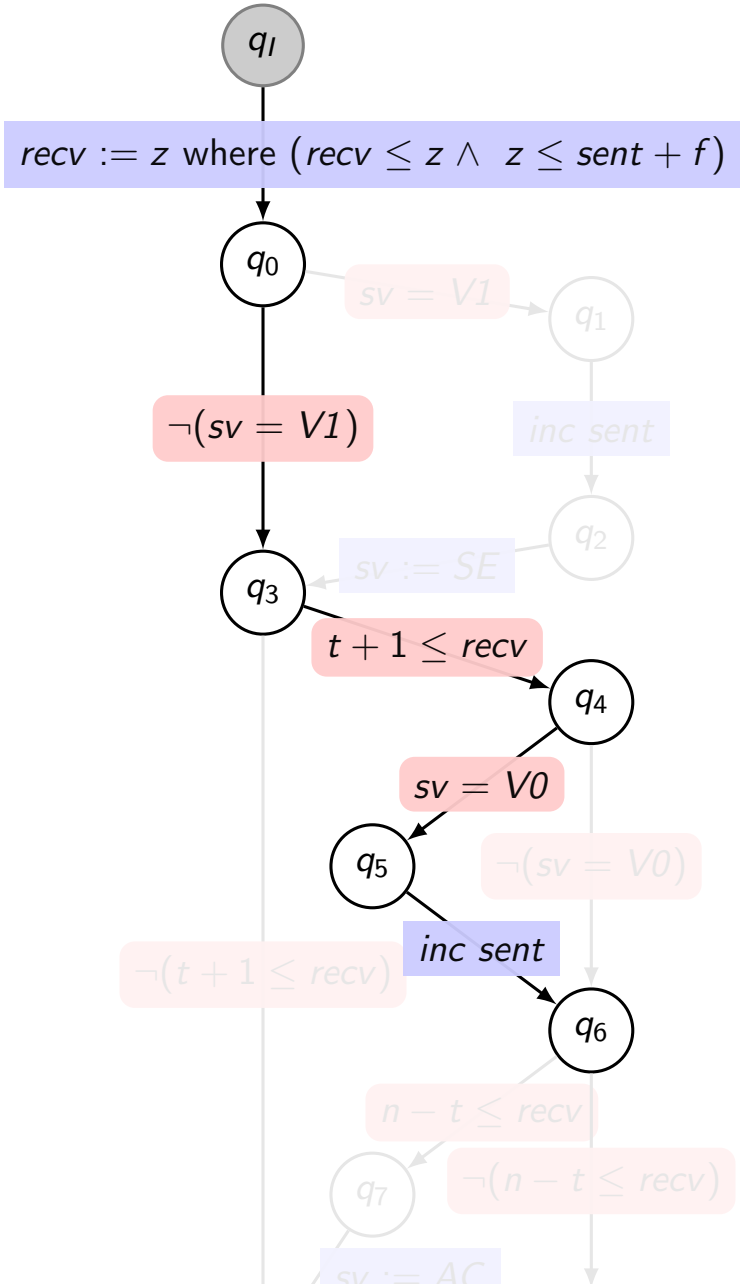
asynchronous

t byzantine faults

correct if $n > 3t$
resilience condition rc

parameterized process
skeleton $p(n, t)$

Abstract CFA



Abstract CFA