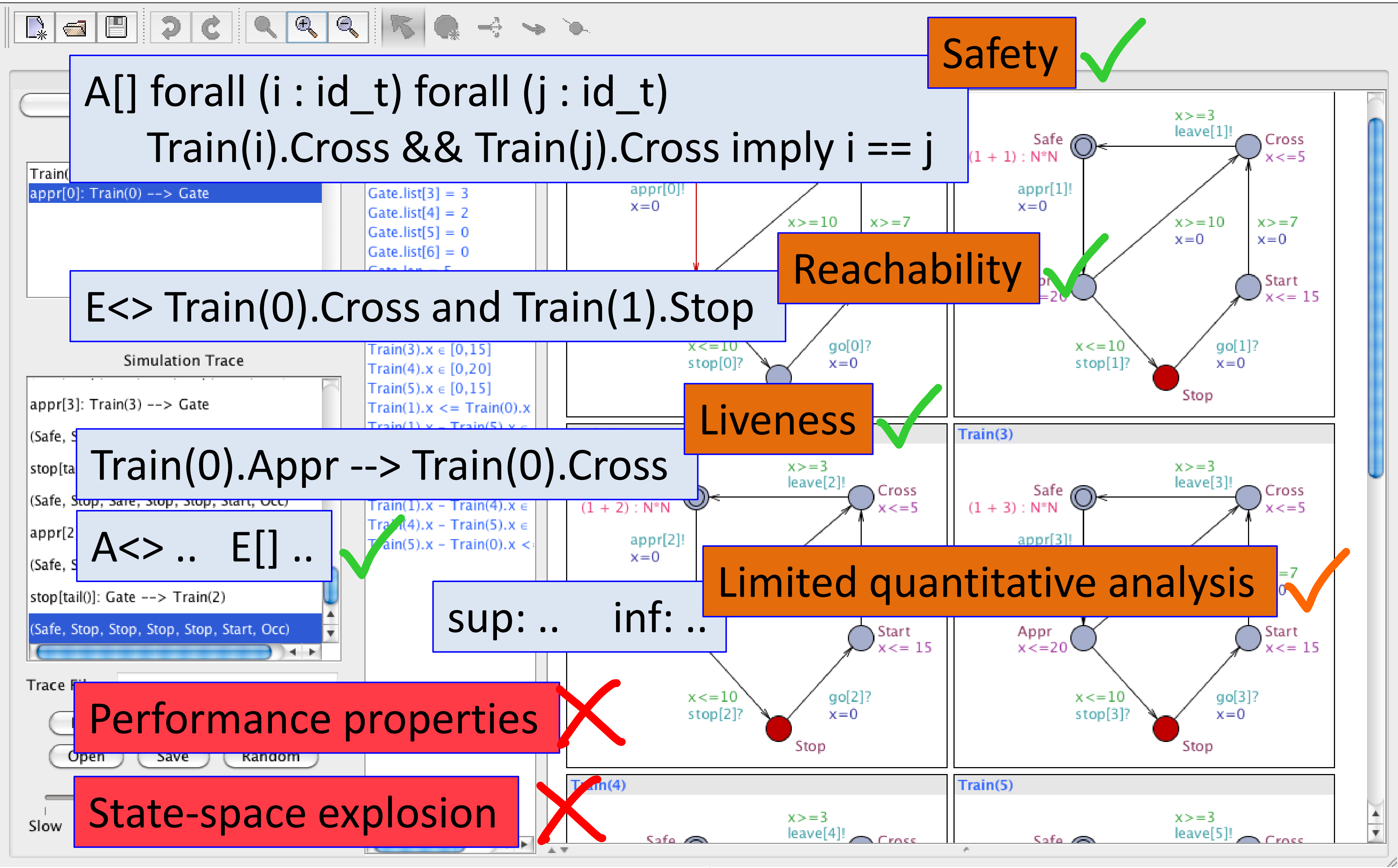
Checking and Distributing Statistical Model-Checking

Peter Bulychev
Alexandre David
Kim G. Larsen
Axel Legay
Marius Mikucionis
Danny Bøgsted Poulsen

Outline

- UPPAAL-SMC in a Nutshell
- Distributing SMC
- Checking DSMC
- Case-studies

UPPAAL



Safety ✓

A[] forall (i : id_t) forall (j : id_t)
Train(i).Cross && Train(j).Cross imply i == j

Reachability ✓

E<> Train(0).Cross and Train(1).Stop

Liveness ✓

Train(0).Appr --> Train(0).Cross

A<> .. E[] .. ✓

sup: .. inf: ..

Limited quantitative analysis ✓

Performance properties ✗

State-space explosion ✗

appr[0]: Train(0) --> Gate

Simulation Trace

appr[3]: Train(3) --> Gate

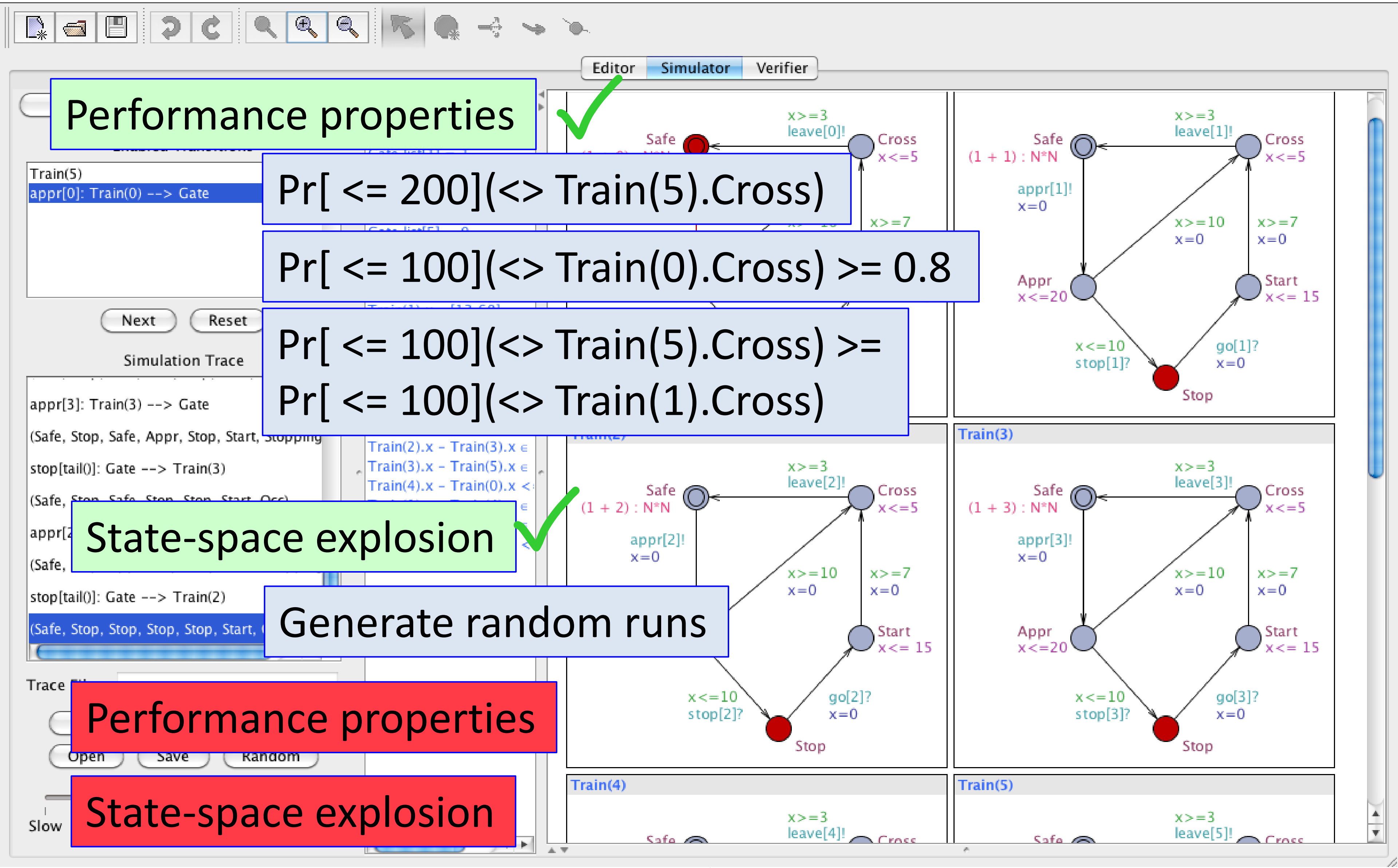
stop[tail()]: Gate --> Train(2)

(Safe, Stop, Stop, Stop, Stop, Start, Occ)

Gate.list[3] = 3
Gate.list[4] = 2
Gate.list[5] = 0
Gate.list[6] = 0

Train(3).x ∈ [0,15]
Train(4).x ∈ [0,20]
Train(5).x ∈ [0,15]
Train(1).x <= Train(0).x

UPPAAL SMC



Train(5)
appr[0]: Train(0) --> Gate

appr[3]: Train(3) --> Gate

(Safe, Stop, Safe, Appr, Stop, Start, Stopping

stop[tail()]: Gate --> Train(3)

(Safe, Stop, Safe, Stop, Stop, Start, Occ)

stop[tail()]: Gate --> Train(2)

(Safe, Stop, Stop, Stop, Stop, Start,

Train(2).x - Train(3).x ∈

Train(3).x - Train(5).x ∈

Train(4).x - Train(0).x <

Performance properties ✓

Pr[<= 200](<> Train(5).Cross)

Pr[<= 100](<> Train(0).Cross) >= 0.8

Pr[<= 100](<> Train(5).Cross) >= Pr[<= 100](<> Train(1).Cross)

State-space explosion ✓

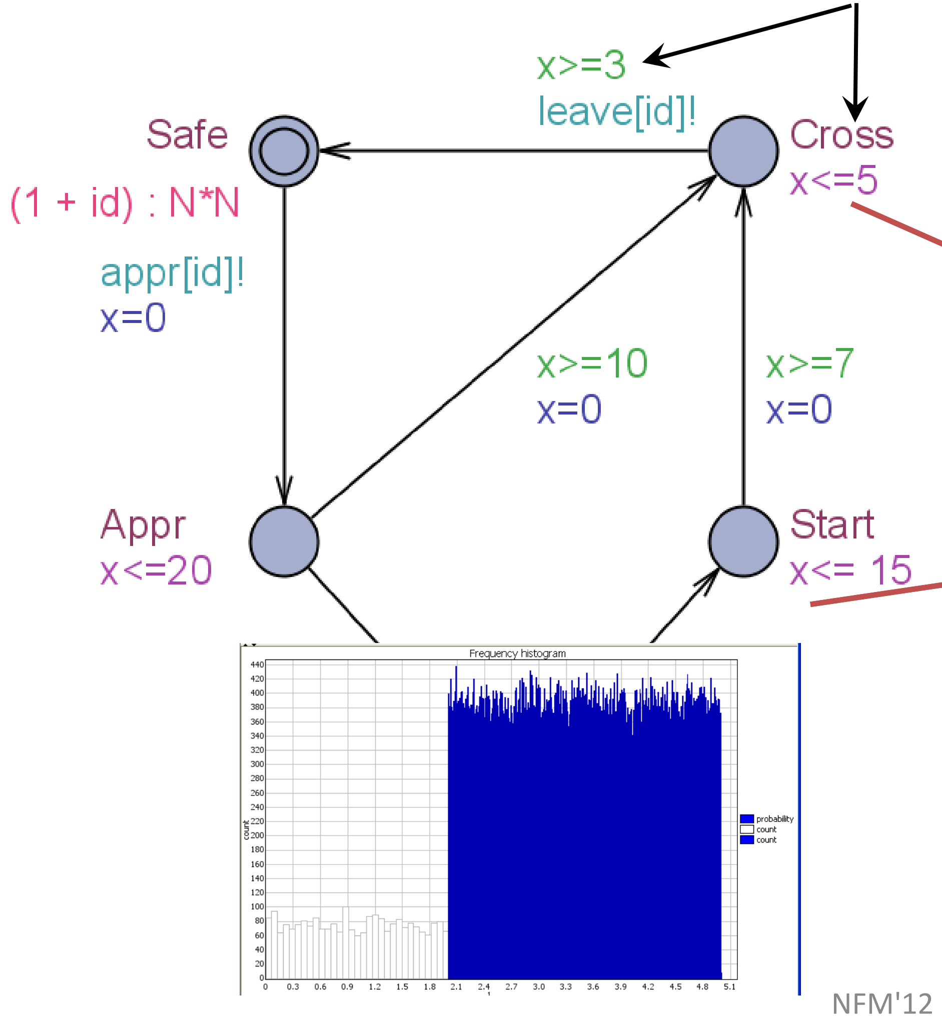
Generate random runs

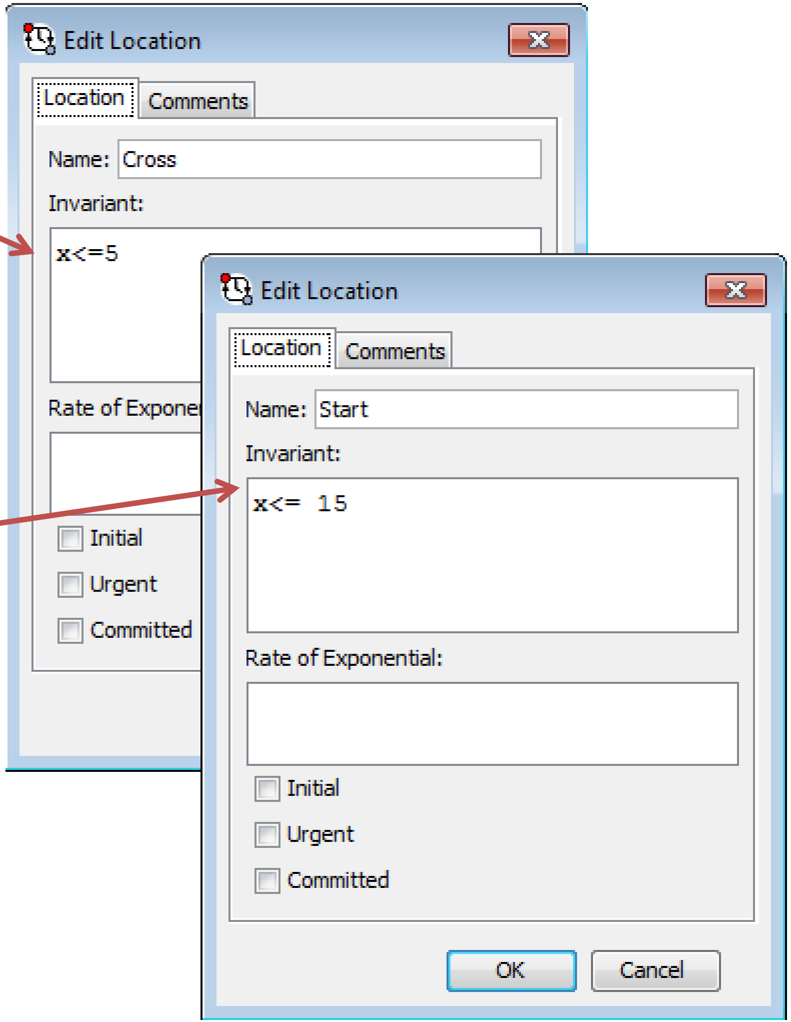
Performance properties

State-space explosion

Stochastic Semantics of UPPAAL TA

Uniform Distribution





Name: Cross

Invariant:

```
x<=5
```

Rate of Exponer

□ Initial

□ Urgent

□ Committed

Name: Start

Invariant:

```
x<= 15
```

Rate of Exponential:

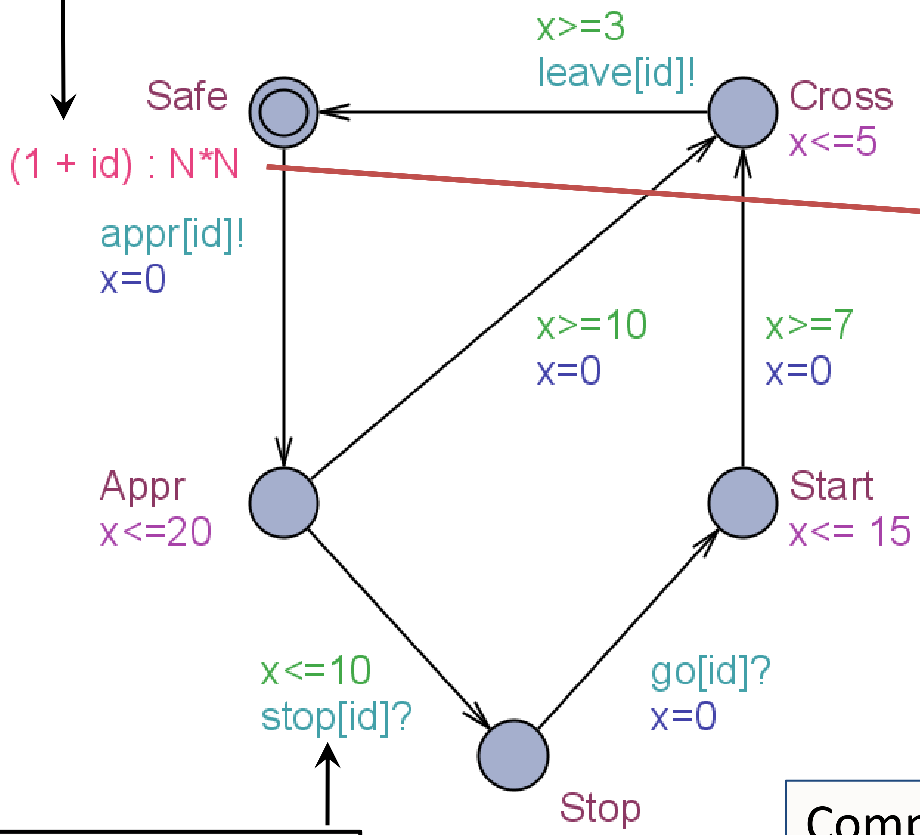
□ Initial

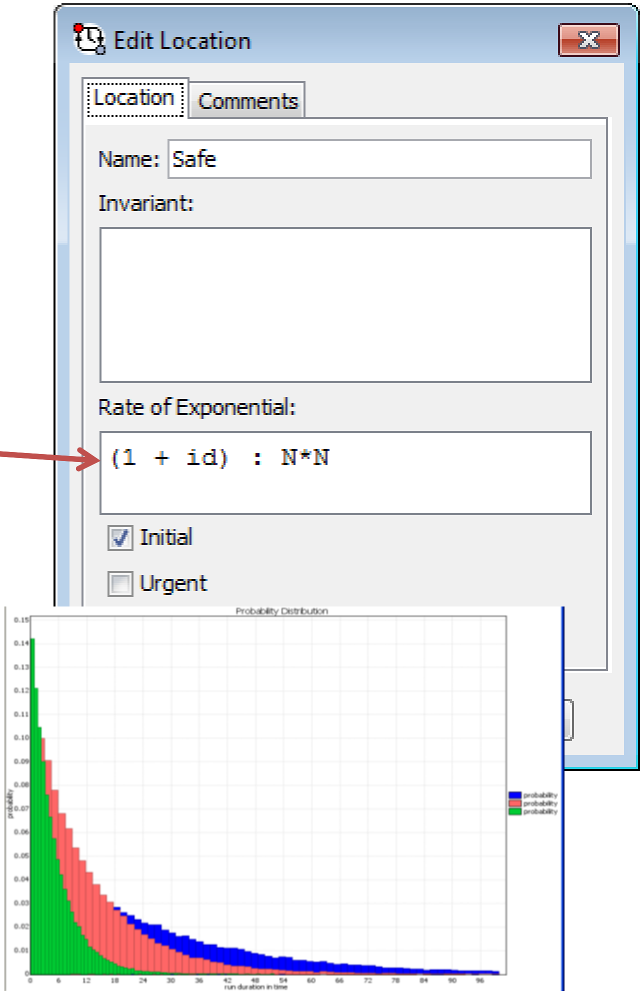
□ Urgent

□ Committed

Stochastic Semantics of UPPAAL TA

Exponential Distribution




Edit Location

Location | Comments

Name: Safe

Invariant:

Rate of Exponential:

```
(1 + id) : N*N
```

☑ Initial

□ Urgent

Input enabled
broadcast channels

Composition =
Repeated races between components

Queries

Syntax

- Hypothesis testing
  ```
  Pr[<=100](<> expr)>=0.1
  x<=100 #<=50 [] expr <=0.5
  ```
- Evaluation
  ```
  Pr[<=100](<> expr)
  ```
- Comparison
  ```
  Pr[<=20](<> e1)>=Pr[<=10](<> e2)
  ```
- Expected value
  ```
  E[<=10;1000](min: expr)
  ```
  Explicit number of runs. Min or max.
- Simulations
  ```
  simulate 10 [<=100]{expr1,expr2}
  ```

Queries

Syntax

- Hypothesis testing
  `Pr[<=100](<> expr)>=0.1`
  `x<=100 #<=50 [] expr <=0.5`
- Evaluation
  `Pr[<=100](<> expr)`
- Comparison
  `Pr[<=20](<> e1)>=Pr[<=10](<> e2)`
- Expected value
  `E[<=10;1000](min: expr)`
  Explicit number of runs. Min or max.
- Simulations
  `simulate 10 [<=100]{expr1,expr2}`

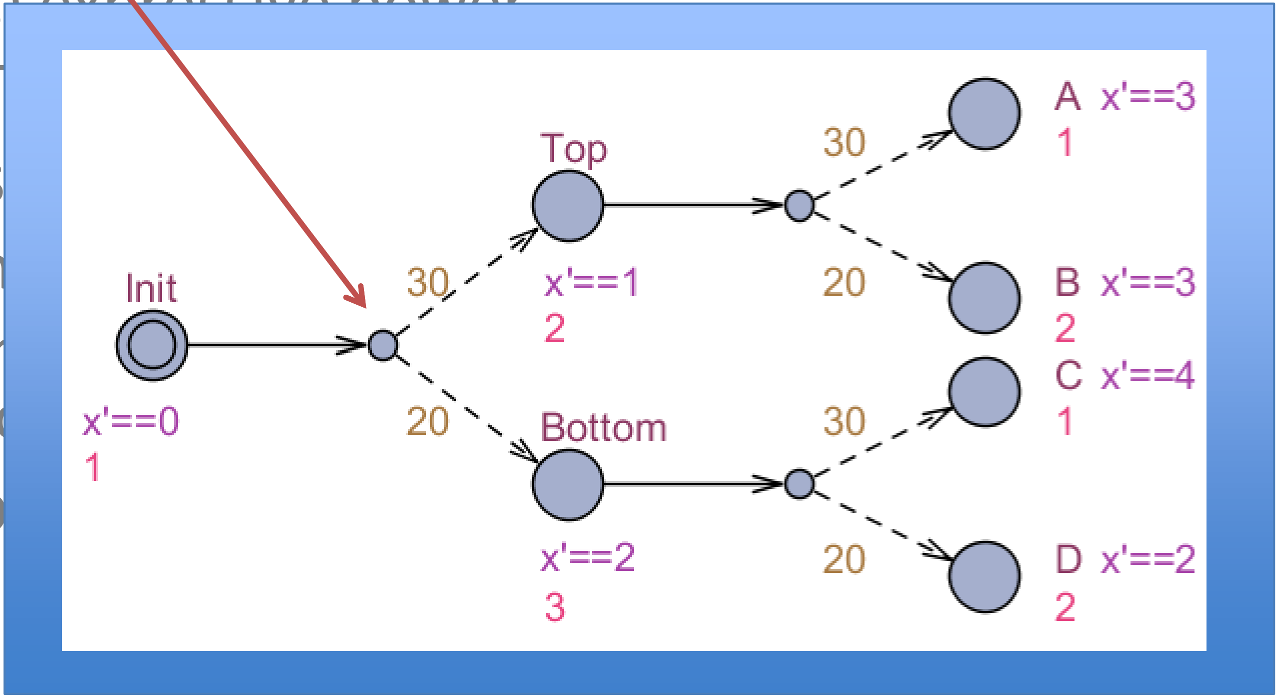
SMC in UPPAAL

Invariants:
x’==0 && y’==bool_fun()

- Constant Slope Timed Automata
  - Clocks may have different (integer) slope in different locations.
  - Branching edges with discrete probabilities (weights).
  - Beyond Priced TA, Energy TA. Equal LHA in (non-stochastic) expressive power.
  - Beyond DTMC, beyond CTMC (with multiple rewards)
- All features of UPPAAL supported
  - User defined functions and types
  - Expressions in guards, invariants, clock-rates, delay-rates (rationals), and weights.
- New GUI for plot-composing and exporting.

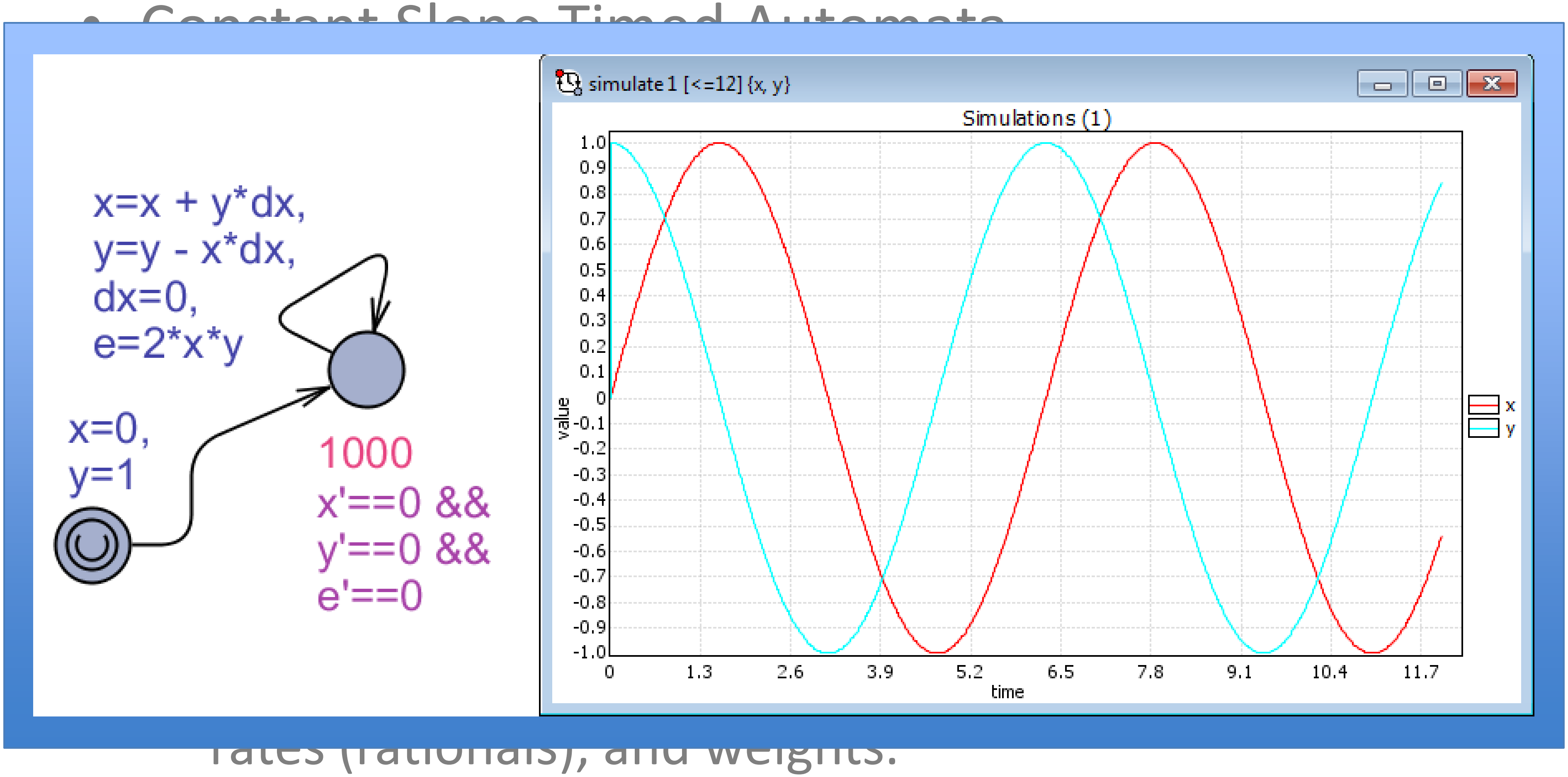
SMC in UPPAAL

- Constant Slope Timed Automata
  - Clocks may have different (integer) slope in different locations.
  - Branching edges with discrete probabilities (weights).
  - Beyond Priced TA, Energy TA. Equal LHA in (non-stochastic) expressive power.
  - Beyond DT
- All features
  - User defin
  - Expression rates (rati
- New GUI fo



SMC in UPPAAL

- Constant Slope Timed Automata



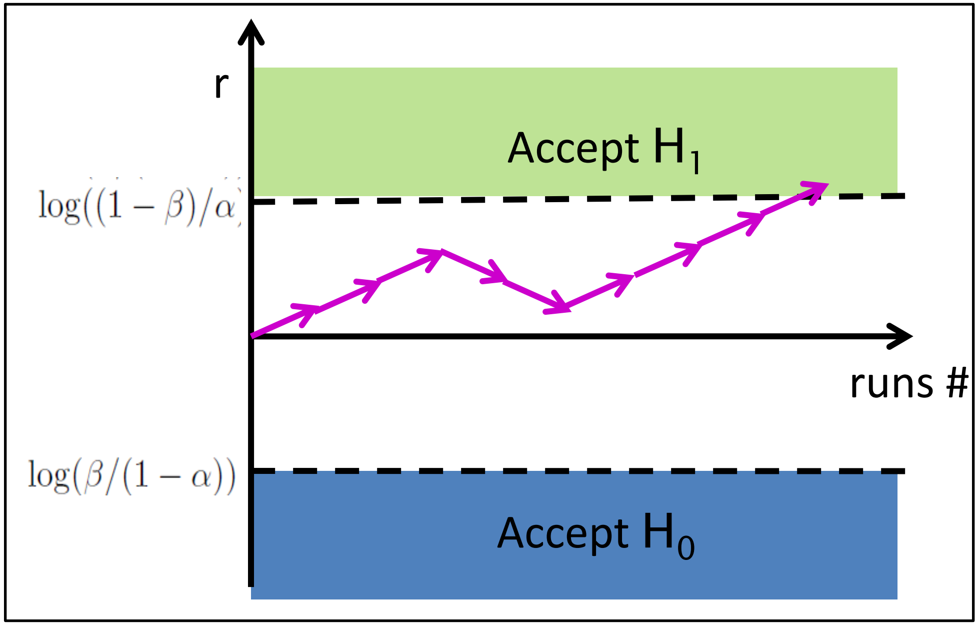
rates (rationals), and weights.

- New GUI for plot-composing and exporting.

Distributing SMC

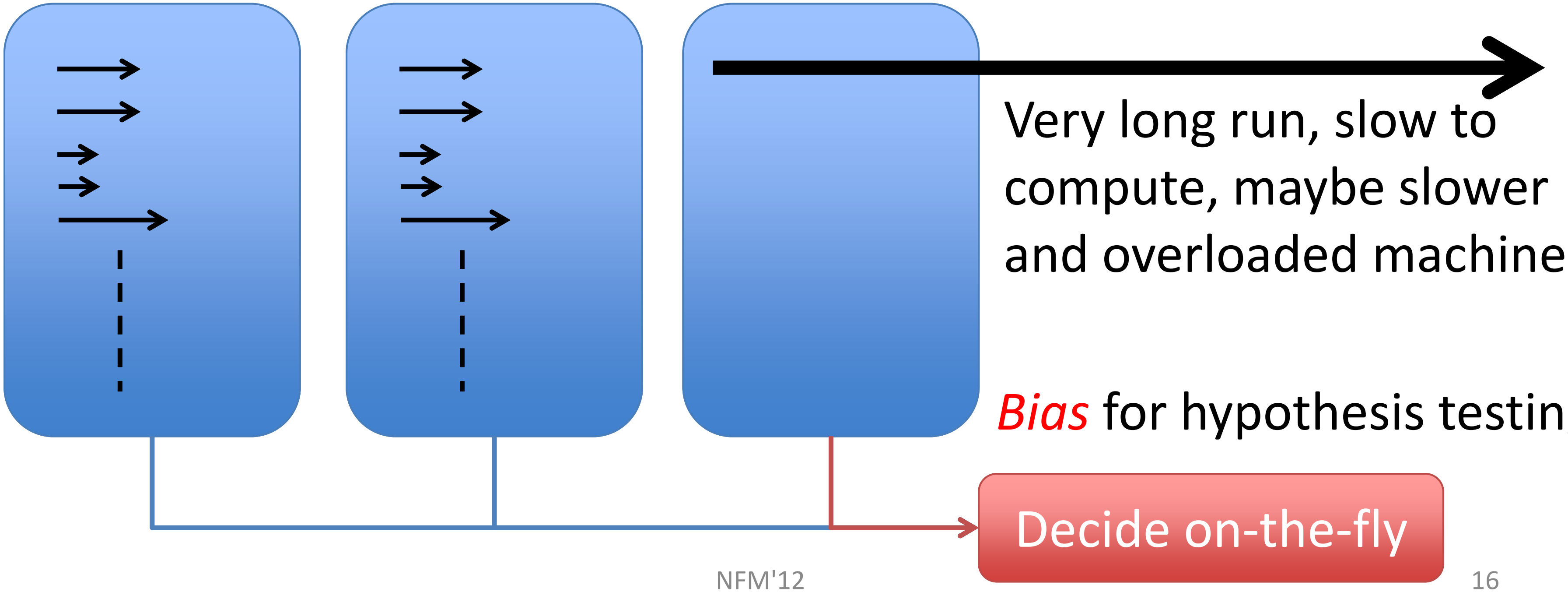
- Distributed SMC
  - Evaluation – trivial to parallelize
  - Hypothesis – careful

Hypothesis testing: intuition.



Distributing SMC

- Distributing hypothesis testing.

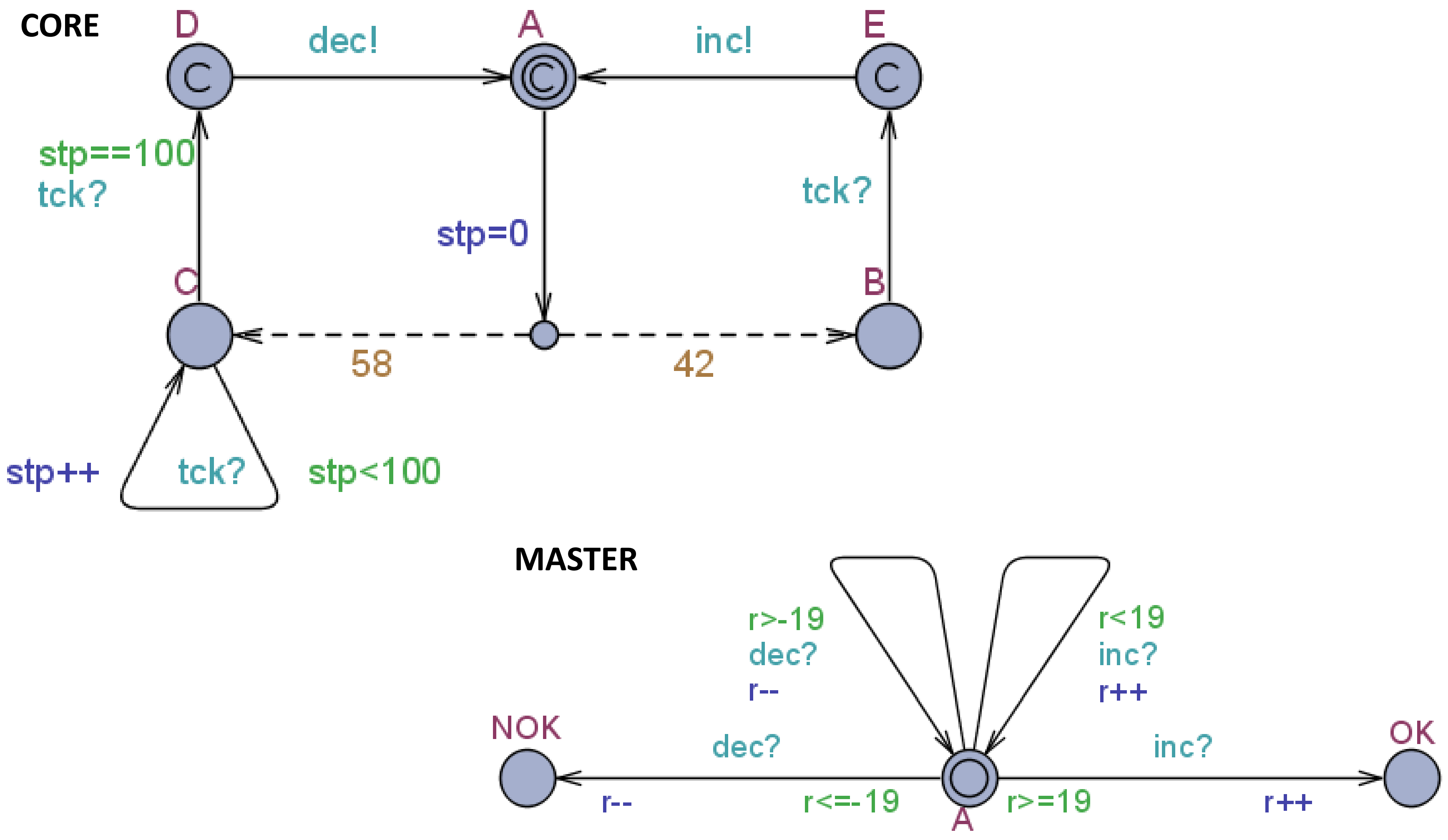


Very long run, slow to compute, maybe slower and overloaded machine.

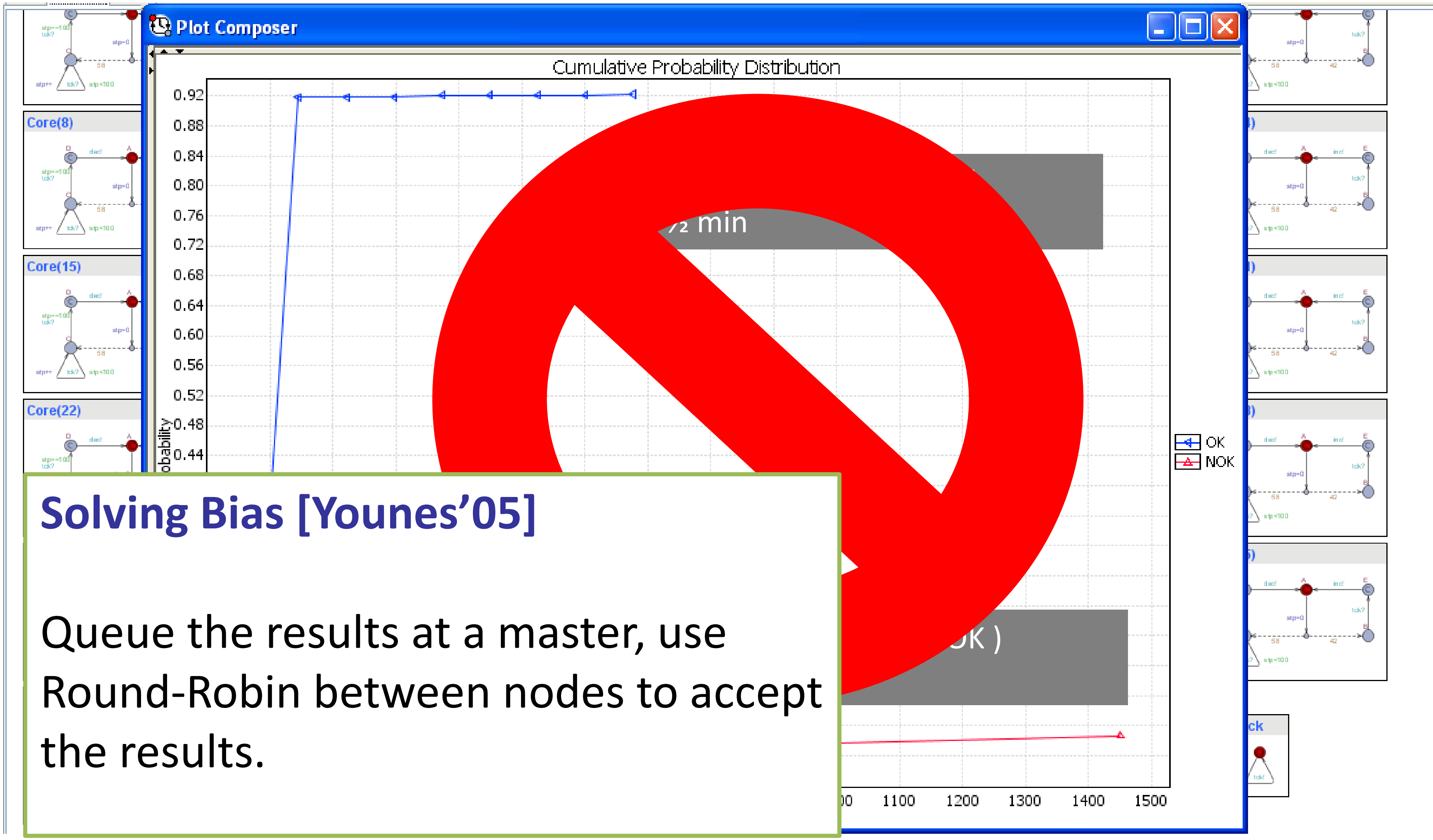
Bias for hypothesis testing

Decide on-the-fly

Distributing SMC – Naïve Approach



Distributing SMC – Naïve Approach

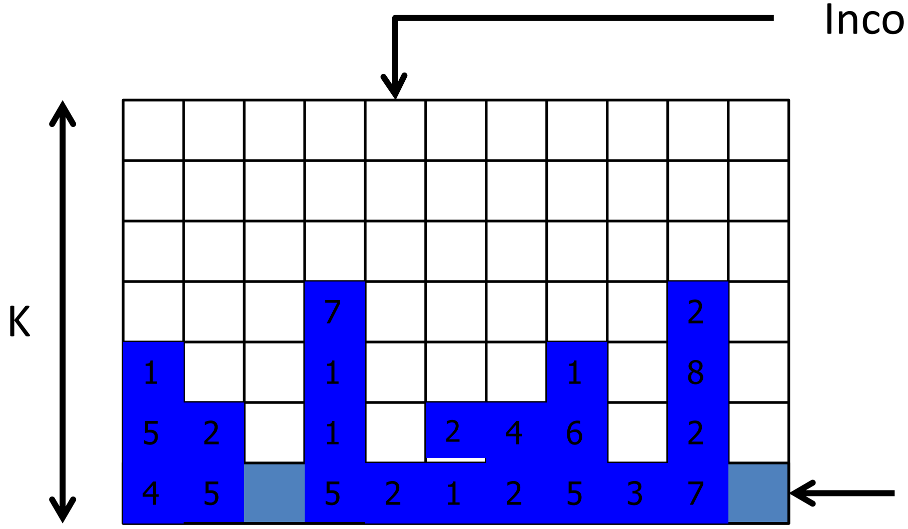


Solving Bias [Younes'05]

Queue the results at a master, use Round-Robin between nodes to accept the results.

Our Implementation

- Use a batch of **B** (e.g 10) runs, transmit one count per batch.
- Use asynchronous communication (MPI)
- Queue results at the master and wait only when the buffer (size=**K**) is full.

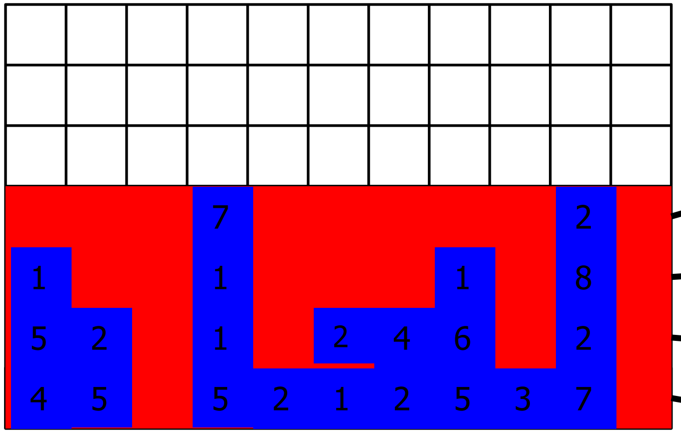


Incoming messages from cores!

Master waits if needed

Our Implementation

- Senders have a buffer of (**K**) asynchronously sent messages and blocks only when the buffer is full.
- The master periodically add results in the buffer.



Update “r”, if can’t decide, continue

Update “r”, if can’t decide, next

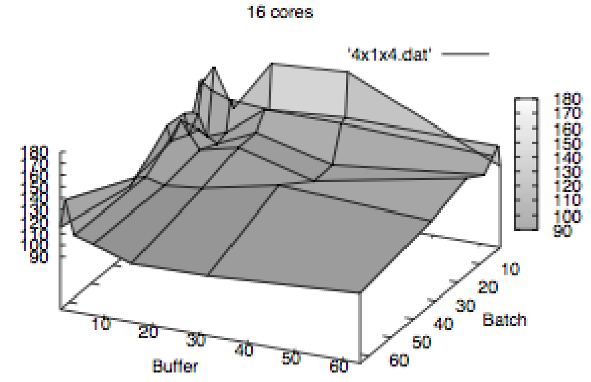
Update “r”, if can’t decide, next

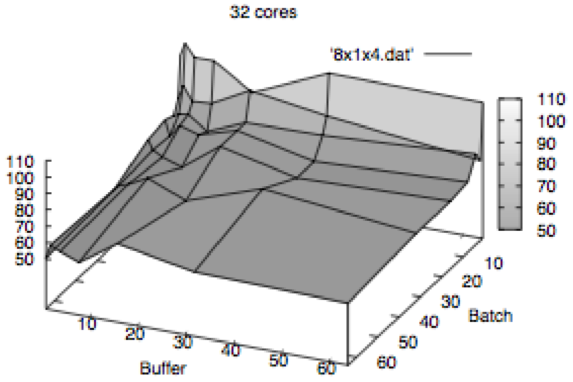
Update “r”, if can’t decide, next

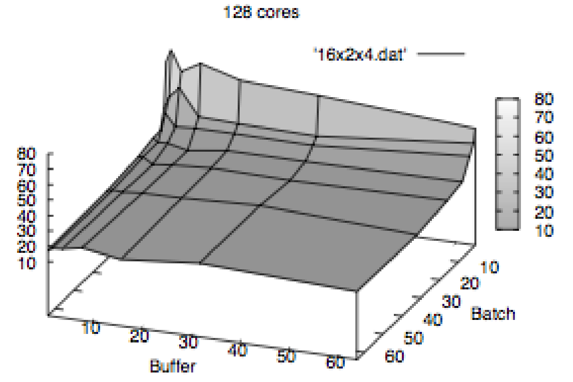
Results

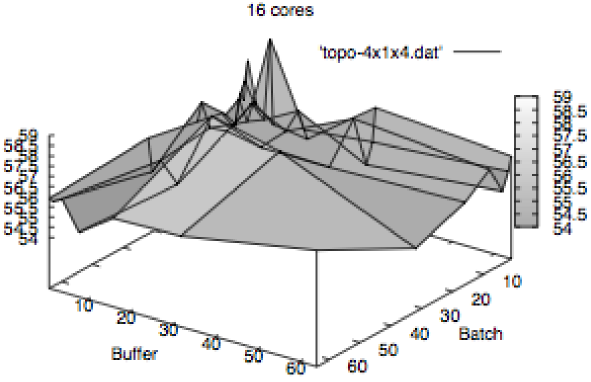
16, 32, 128 cores, Vary Buffer & Batch Sizes

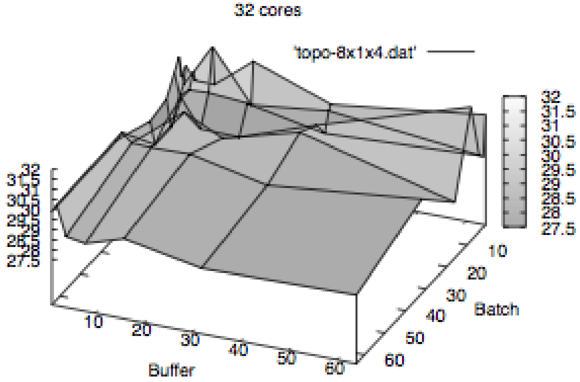
“Small” model: Exhibit expected behaviour.

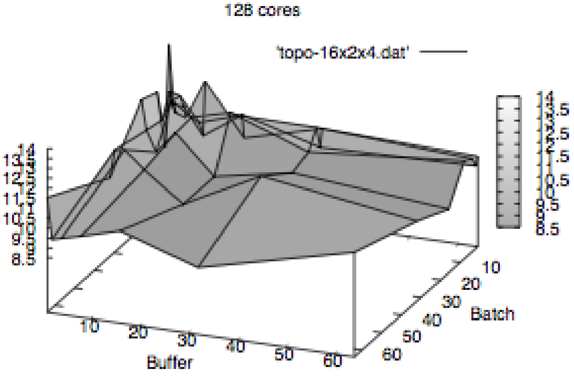








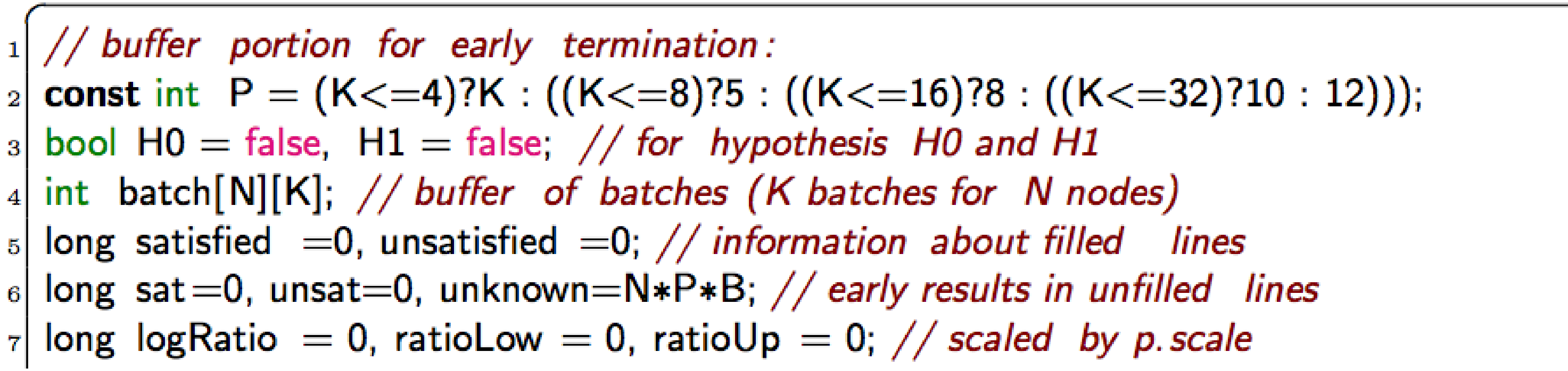




“Big” model: Amortize the differences.

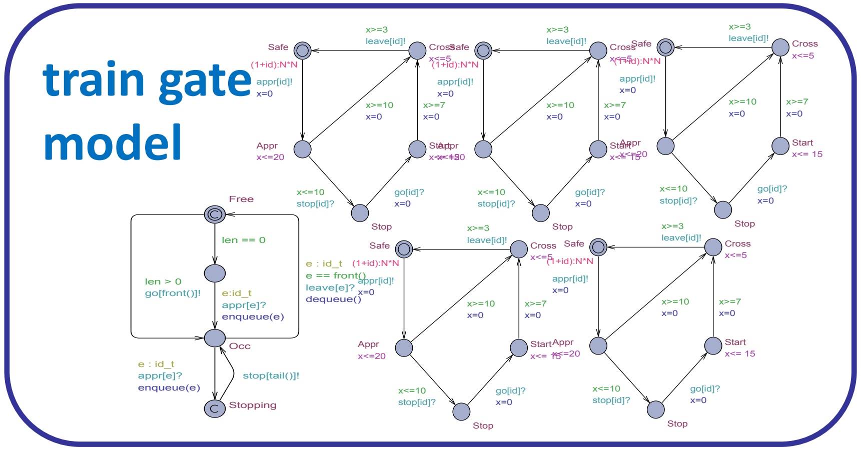
Checking DSMC

- We can model the algorithm inside UPPAAL.
  - Run SMC on it, even DSMC!

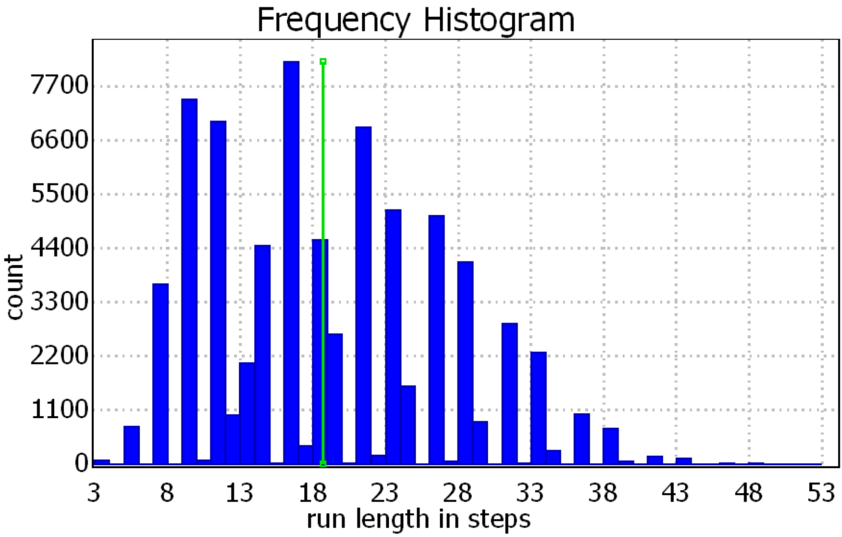


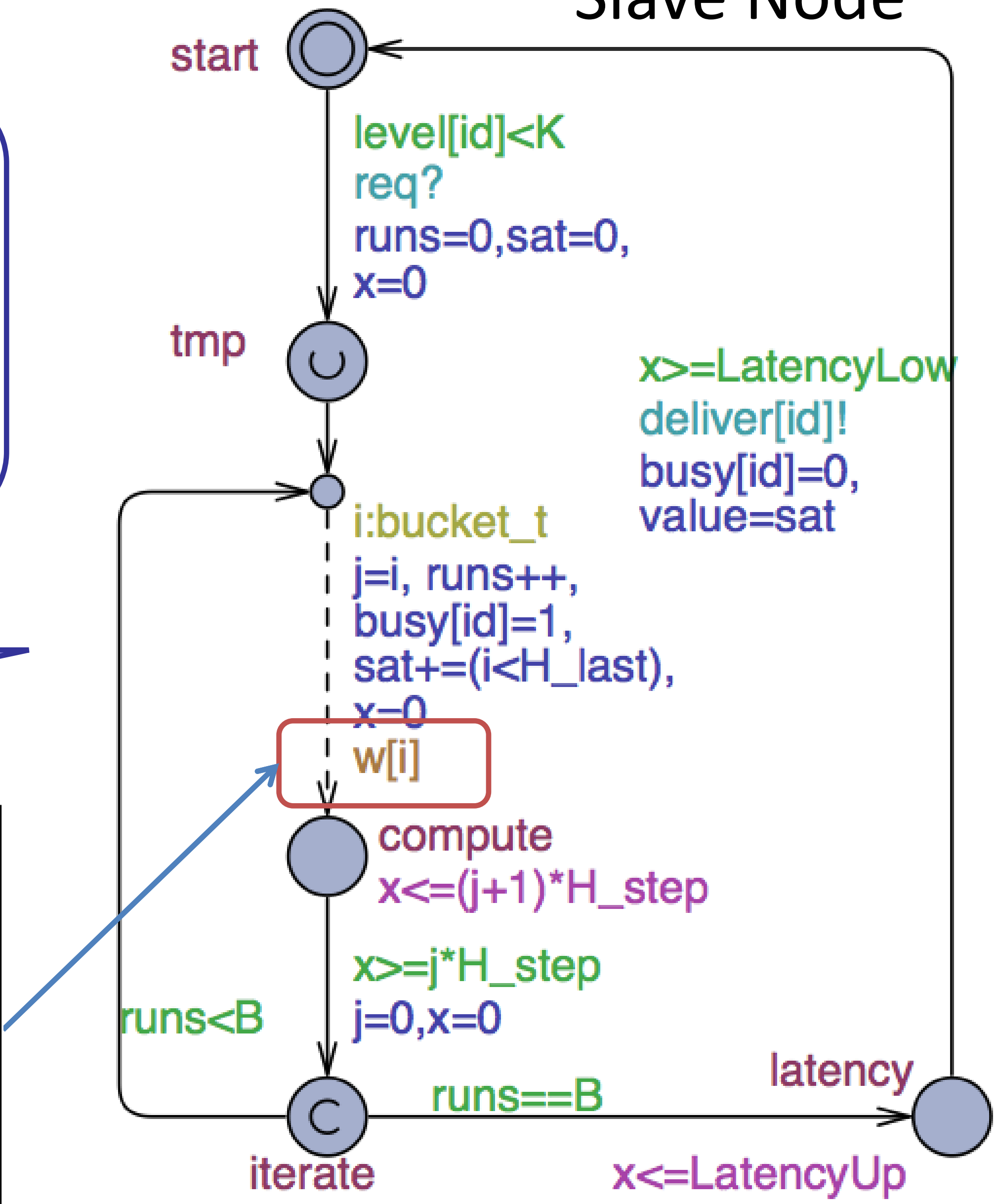
```
// buffer portion for early termination:
const int P = (K<=4)?K : ((K<=8)?5 : ((K<=16)?8 : ((K<=32)?10 : 12)));
bool H0 = false, H1 = false; // for hypothesis H0 and H1
int batch[N][K]; // buffer of batches (K batches for N nodes)
long satisfied =0, unsatisfied =0; // information about filled lines
long sat=0, unsat=0, unknown=N*P*B; // early results in unfilled lines
long logRatio = 0, ratioLow = 0, ratioUp = 0; // scaled by p.scale
```

Slave Node

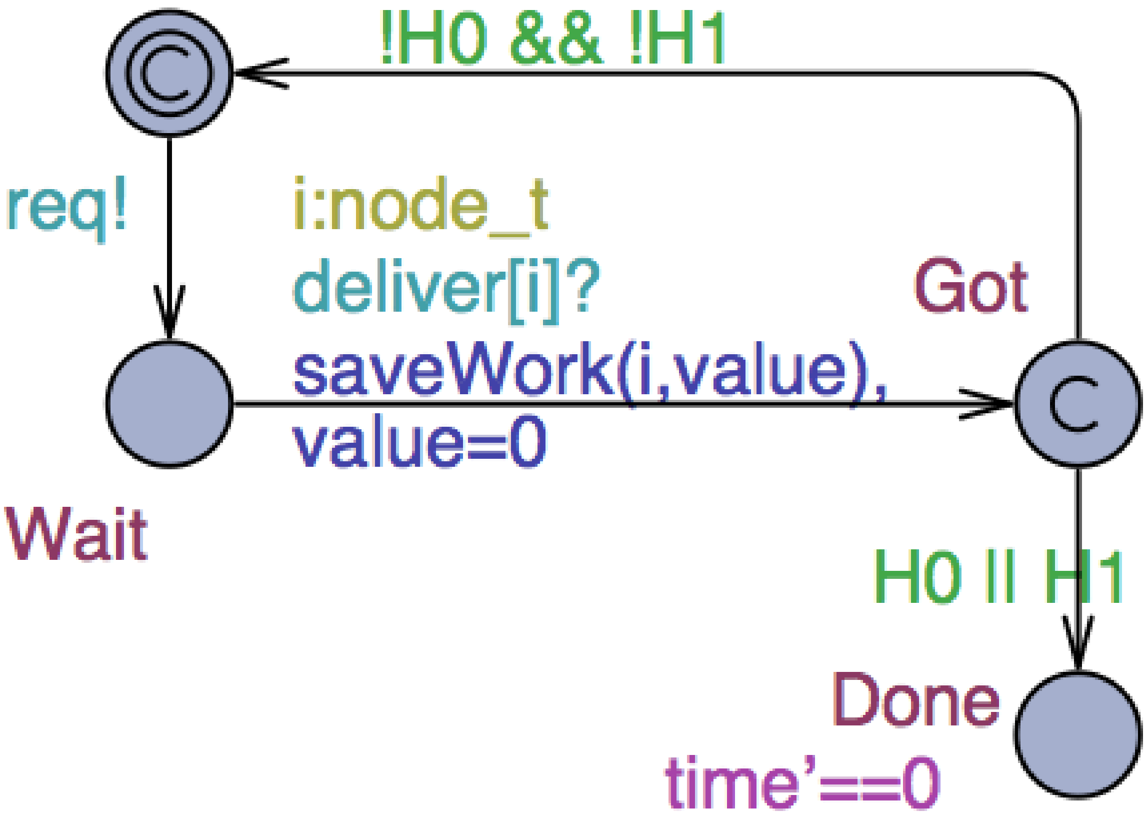


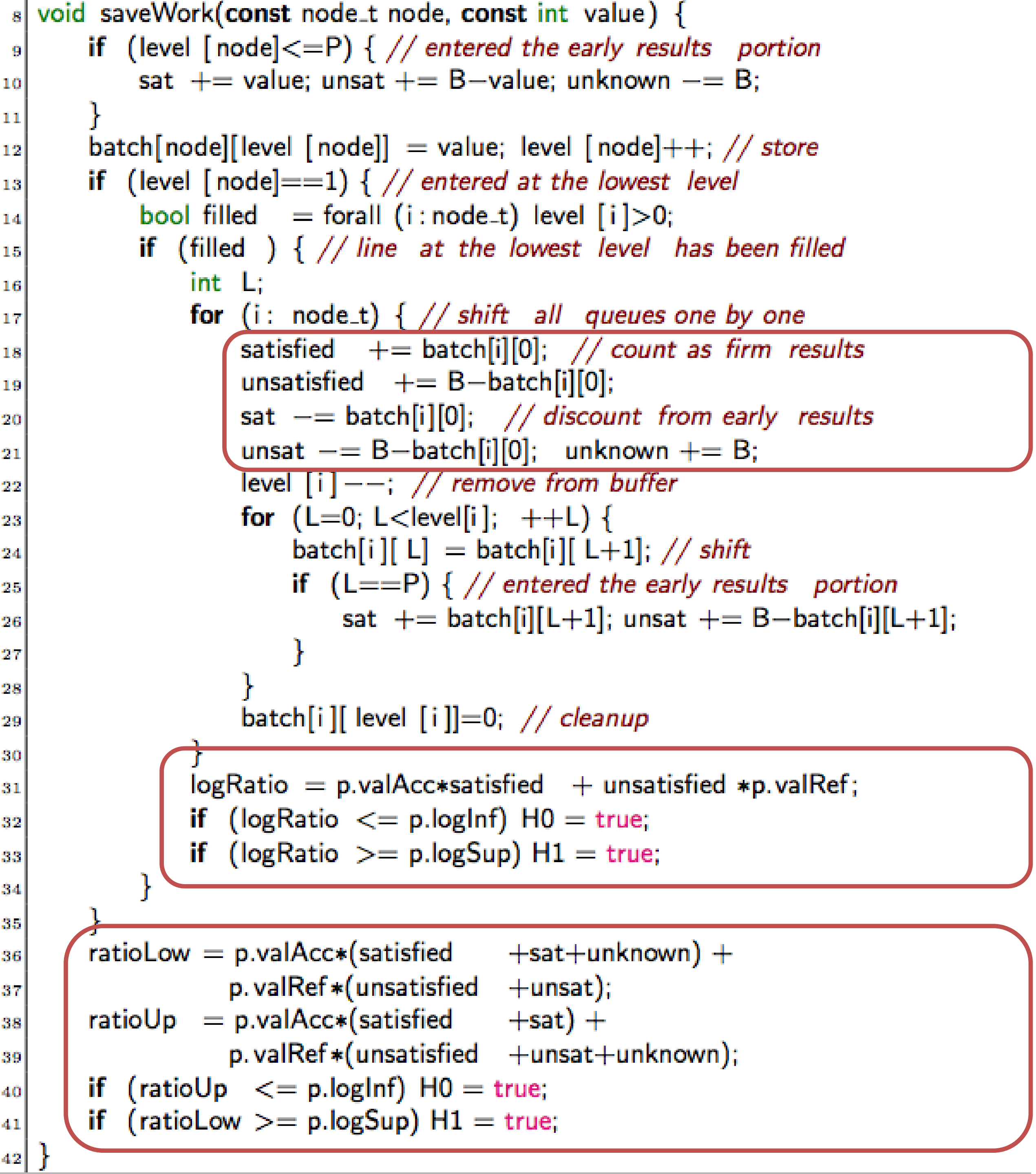
Pr[# <= 100](<> Train(5).Cross)





Master Node


!H0 && !H1
req!
i:node_t
deliver[i]?
saveWork(i,value),
value=0
Got
Wait
H0 || H1
Done
time'==0



```
void saveWork(const node_t node, const int value) {
    if (level [node]<=P) { // entered the early results portion
        sat += value; unsat += B-value; unknown -= B;
    }
    batch[node][level [node]] = value; level [node]++; // store
    if (level [node]==1) { // entered at the lowest level
        bool filled = forall (i : node_t) level [i ]>0;
        if (filled ) { // line at the lowest level has been filled
            int L;
            for (i : node_t) { // shift all queues one by one
                satisfied += batch[i][0]; // count as firm results
                unsatisfied += B-batch[i][0];
                sat -= batch[i][0]; // discount from early results
                unsat -= B-batch[i][0]; unknown += B;
                level [i ]--; // remove from buffer
                for (L=0; L<level[i ]; ++L) {
                    batch[i ][ L] = batch[i][ L+1]; // shift
                    if (L==P) { // entered the early results portion
                        sat += batch[i][L+1]; unsat += B-batch[i][L+1];
                    }
                }
                batch[i ][ level [i ]]=0; // cleanup
            }
            logRatio = p.valAcc*satisfied + unsatisfied *p.valRef;
            if (logRatio <= p.logInf) H0 = true;
            if (logRatio >= p.logSup) H1 = true;
        }
    }
    ratioLow = p.valAcc*(satisfied   +sat+unknown) +
               p. valRef*(unsatisfied +unsat);
    ratioUp  = p.valAcc*(satisfied   +sat) +
               p. valRef*(unsatisfied +unsat+unknown);
    if (ratioUp  <= p.logInf) H0 = true;
    if (ratioLow >= p.logSup) H1 = true;
}
```

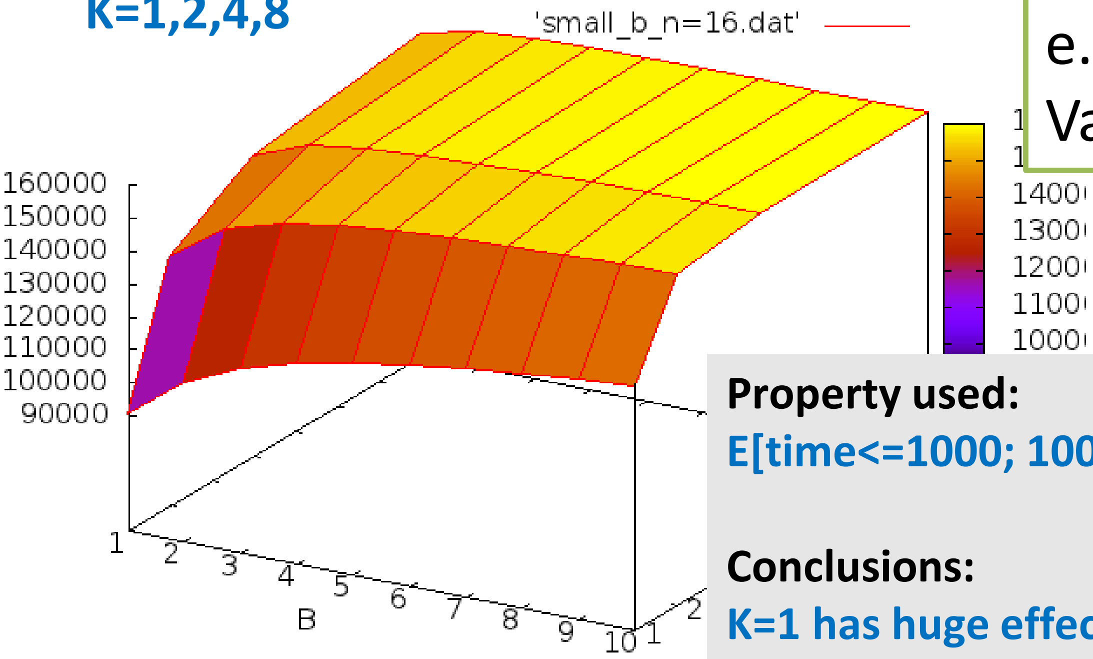
Read batch.

Exact decision.

Safe approximate decision.

Results

N=16
B=1..10
K=1,2,4,8



Can predict performance.
Can derive more information,
e.g., processor usage.
Validate implementation.

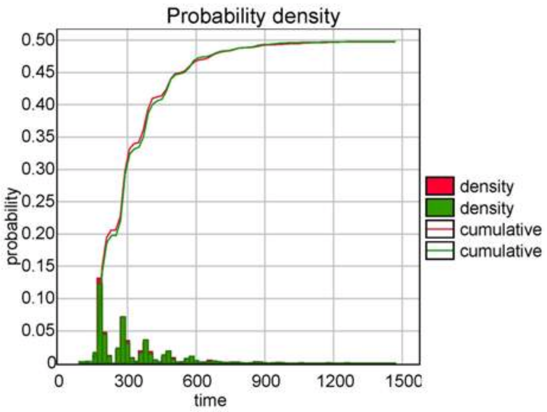
Property used:

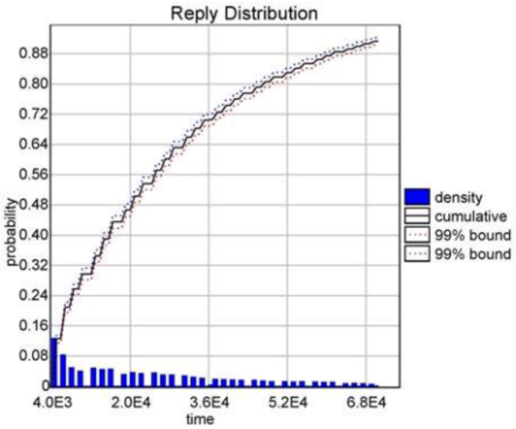
E[time<=1000; 1000] (max: usage)

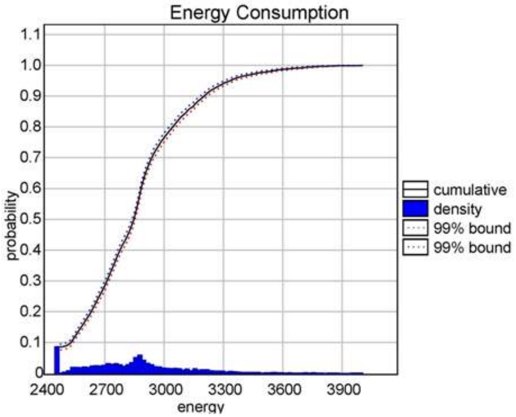
Conclusions:

K=1 has huge effect and should be avoided.
K=2 has effect if B<20.
K>2 are indistinguishable on homogeneous cluster.
K>2 and B>20: number of simulations scale linearly to the number of cores used.

Case Studies

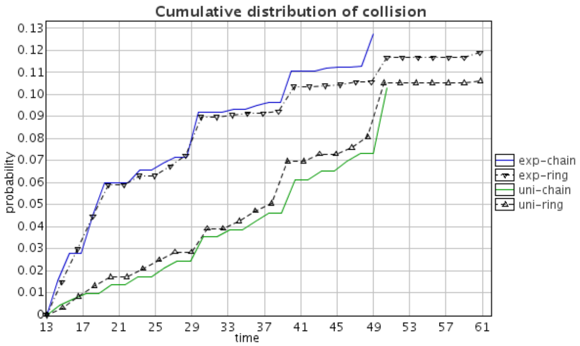


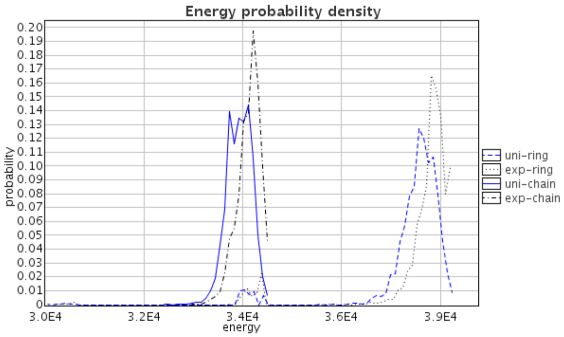




FIREWIRE

BLUETOOTH





LMAC

LMAC

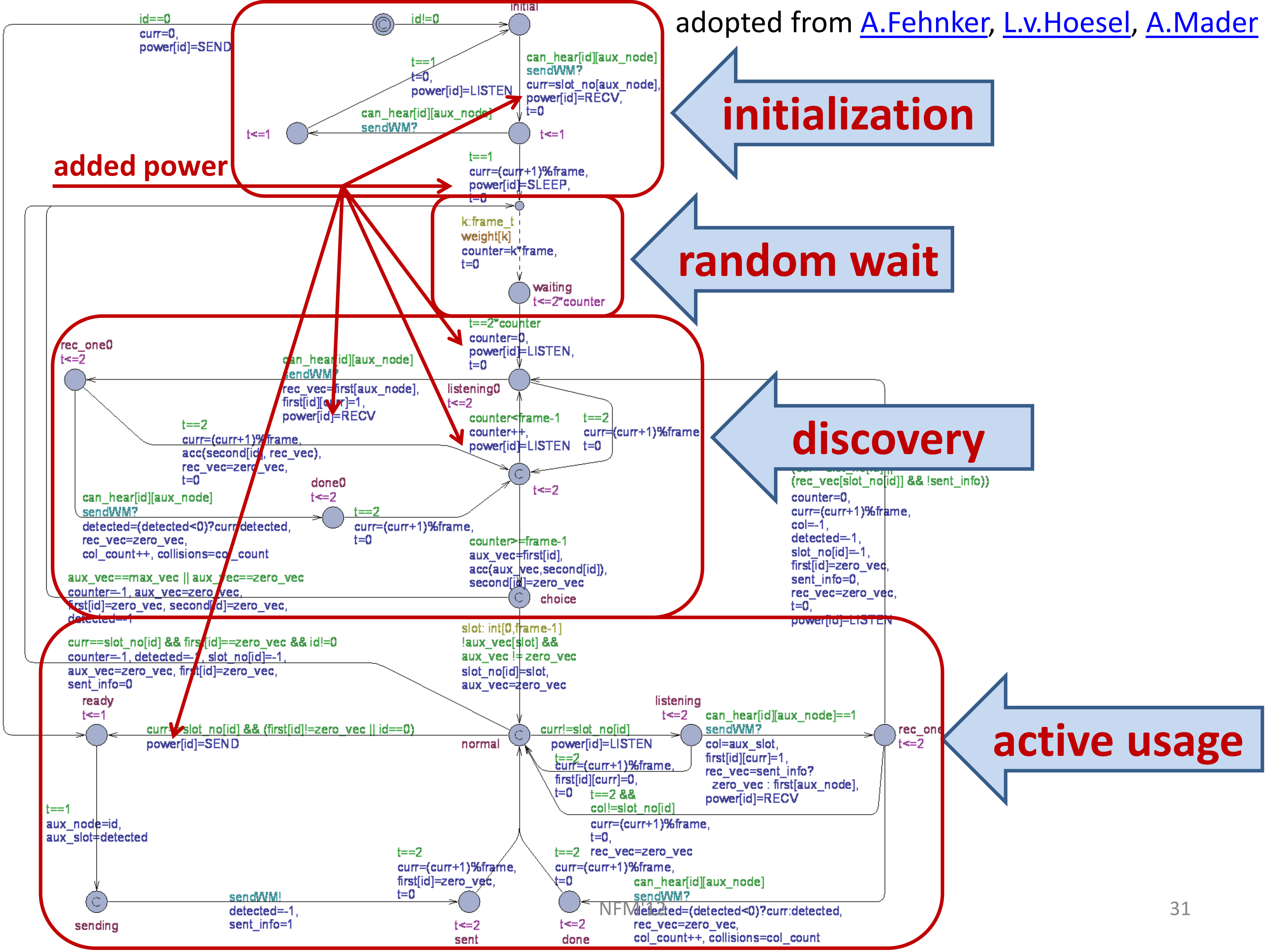
Lightweight Media Access Control (LMAC)

- Problem domain:
  - communication scheduling
- Targeted for:
  - self-configuring networks,
  - collision avoidance,
  - low power consumption
- Application domain:
  - wireless sensor networks

LMAC Protocol Design

- Four phases:
  - Initialization (listen until a neighbor is heard)
  - Waiting (delay a random amount of time frames)
  - Discovery (wait for entire frame and note used slots)
  - Active
    - choose free slot,
    - use it to transmit, including info about detected collisions
    - listen on other slots
    - fallback to Discovery if collision is detected
- Only neighbors can detect collision and tell the user-node that its slot is used by others

adopted from A.Fehnker, L.v.Hoesel, A.Mader



initialization

random wait

discovery

active usage

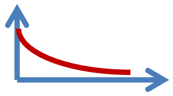
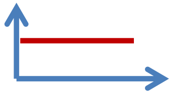
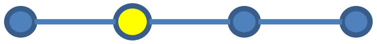
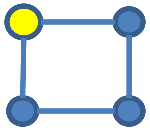
added power

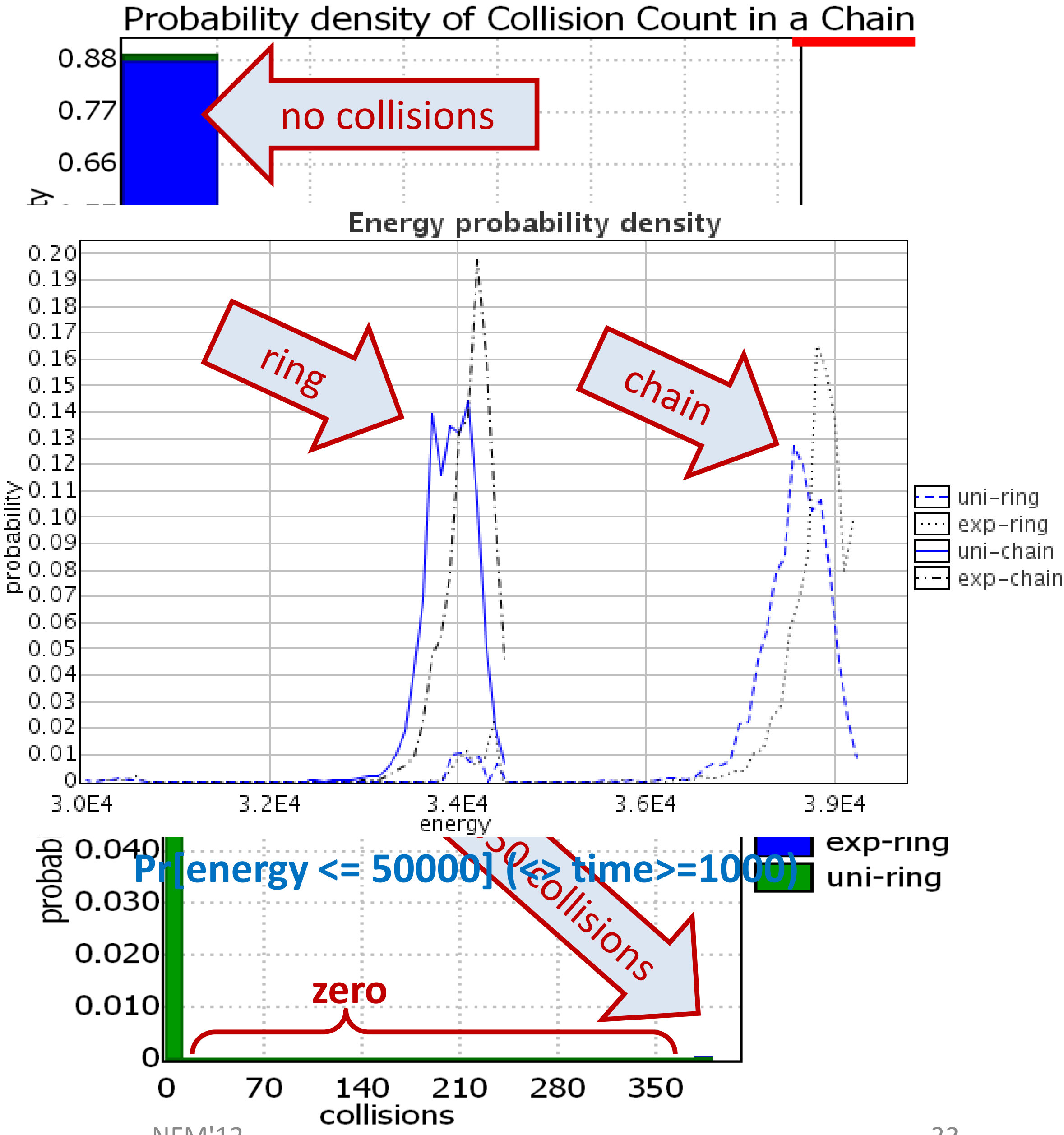
Classical vs. Statistical MC

- A.Fehnker, L.v.Hoesel and A.Mader used UPPAAL to explore 4- and 5-node topologies and found cases with perpetual collisions.
- However they could not know whether the next collisions are inevitable.
- Statistical MC offers an insight by calculating the probability over the number of collisions.

+ estimated cost in terms of energy.

LMAC Simple Statistics for 4 Nodes

- Wait distribution:
  - geometric 
  - uniform 
- Network topology:
  - chain 
  - ring 
- Collision probability
- Collision count
- Power consumption



no collisions

ring

chain

Pr[energy <= 50000] (<> time>=1000)

50 collisions

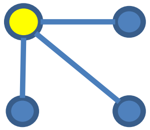
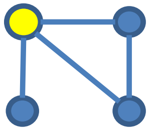
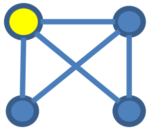
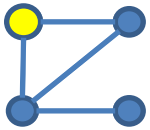
zero

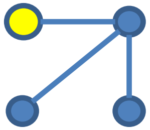
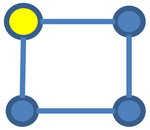
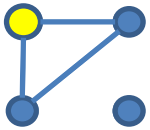
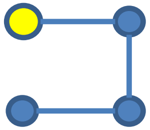
Pr[collisions<=50000] (<> time>=1000)

LMAC with Parameterized Topology

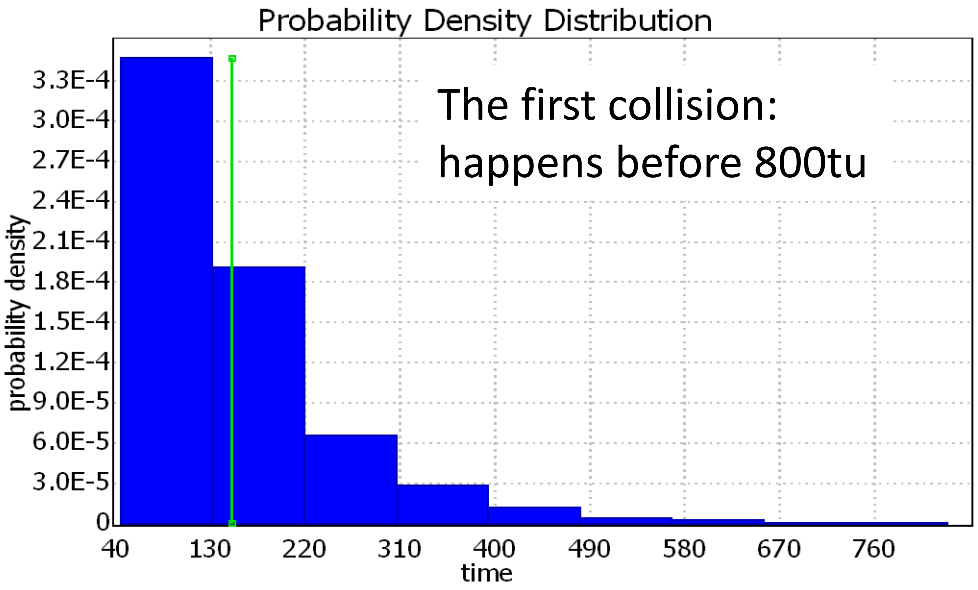
Collision probability in a 4 node network of a randomly generated topology:

Pr[time<=200] (<> col_count>0)

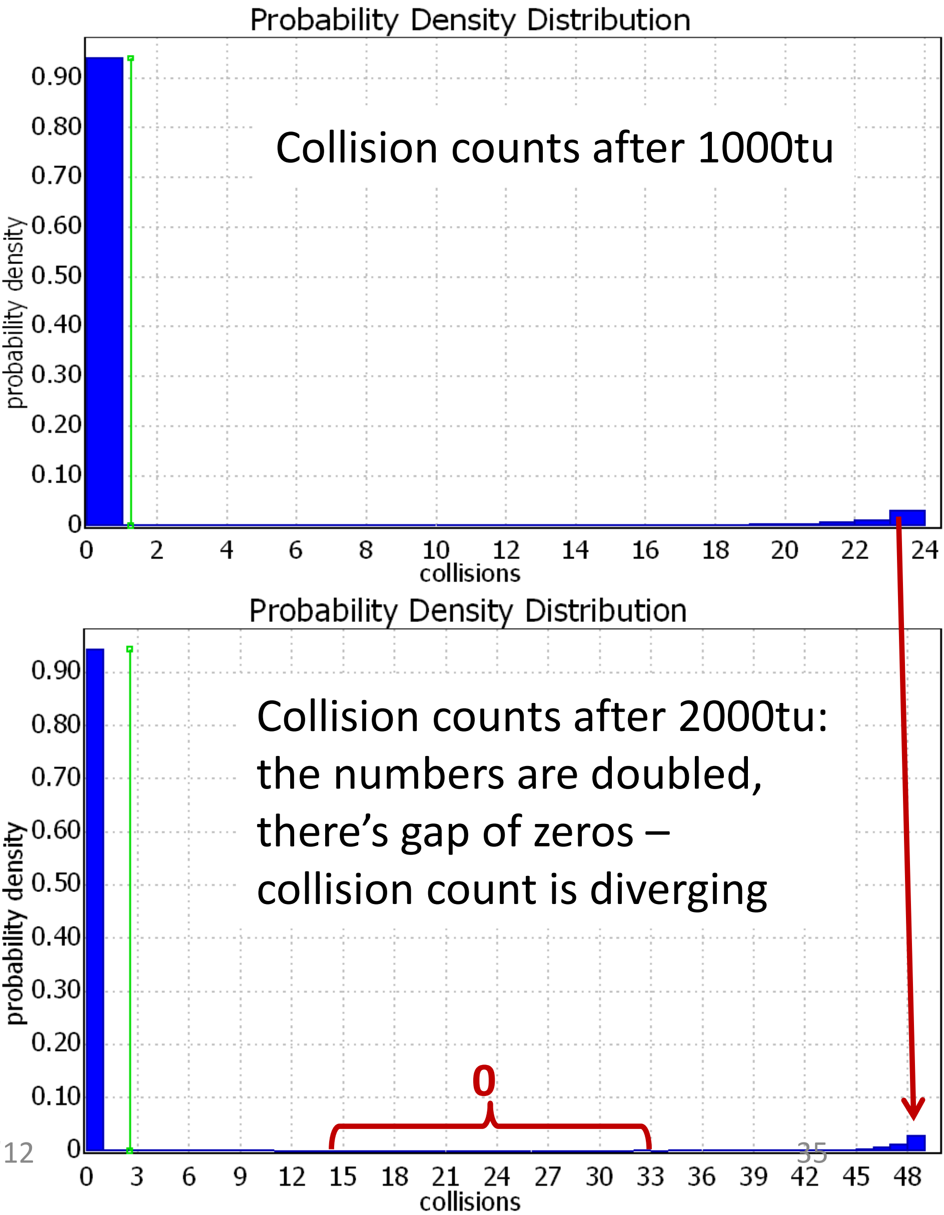
	topology	collision probability
(star)		[0.36; 0.39]
		[0.29; 0.36]
		[0.26; 0.30]
		[0.19; 0.21]

	topology	collision probability
		[0.08; 0.19]
(ring)		[0.11; 0.13]
		[0.08; 0.15]
(chain)		[0.049; 0.050]

10-Node Chain



The first collision: happens before 800tu

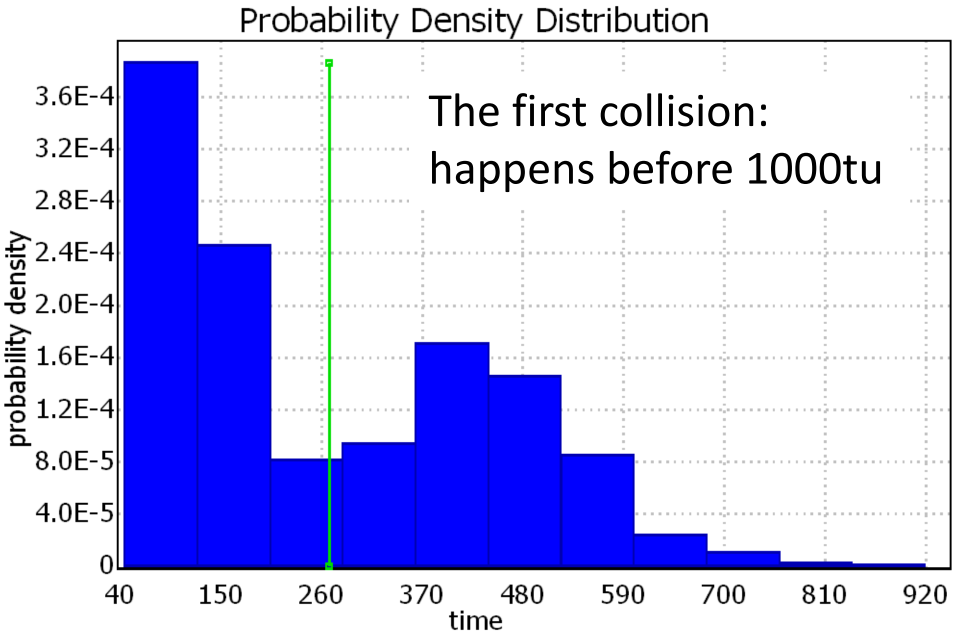


Collision counts after 1000tu

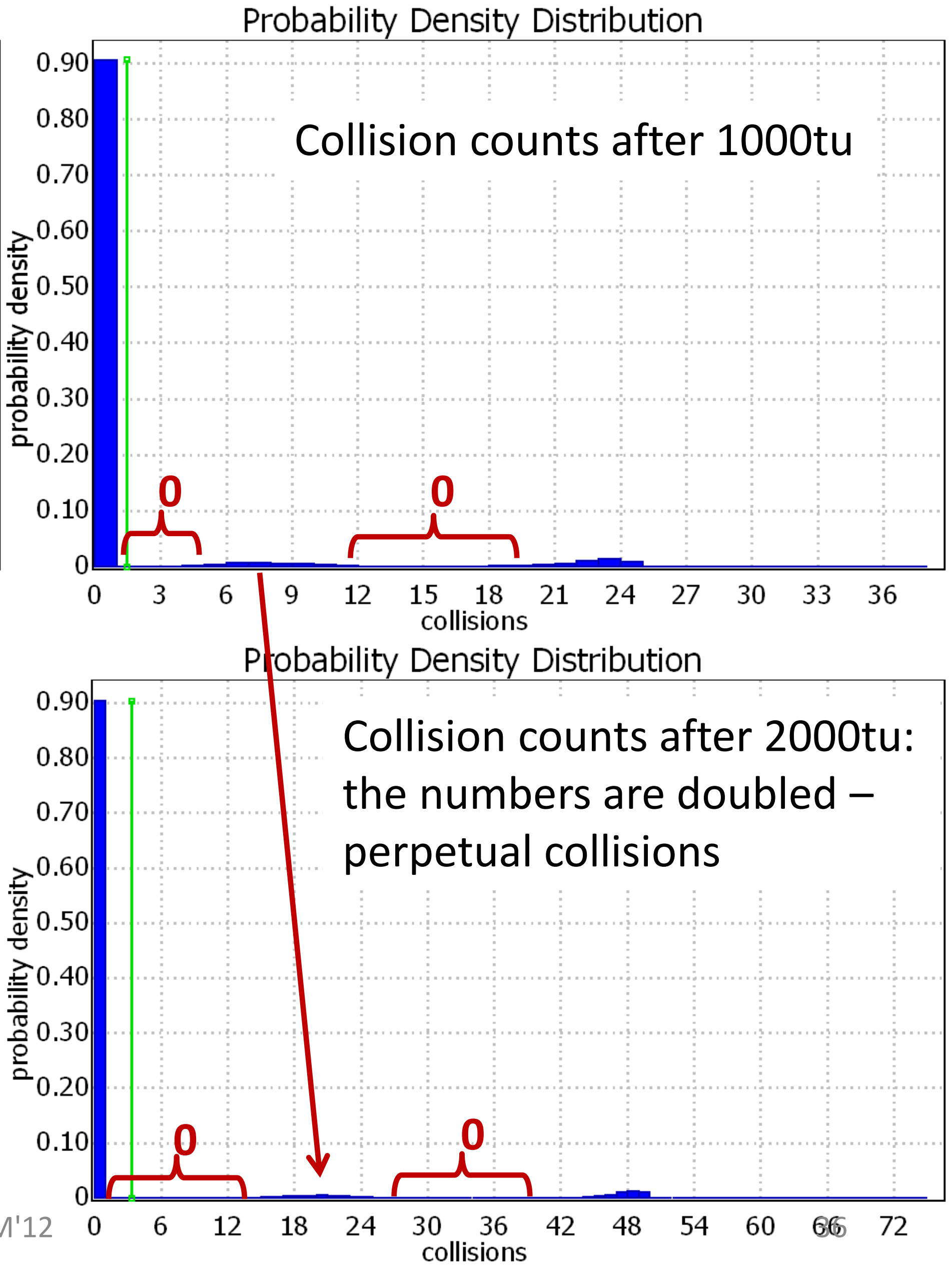
Collision counts after 2000tu: the numbers are doubled, there’s gap of zeros – collision count is diverging

The first collisions can be as late as **800**tu.
It is very likely (**>94%**) that
there will be **0** collisions.
But if they happen, some are perpetual.

10-Node Ring



The first collision: happens before 1000tu

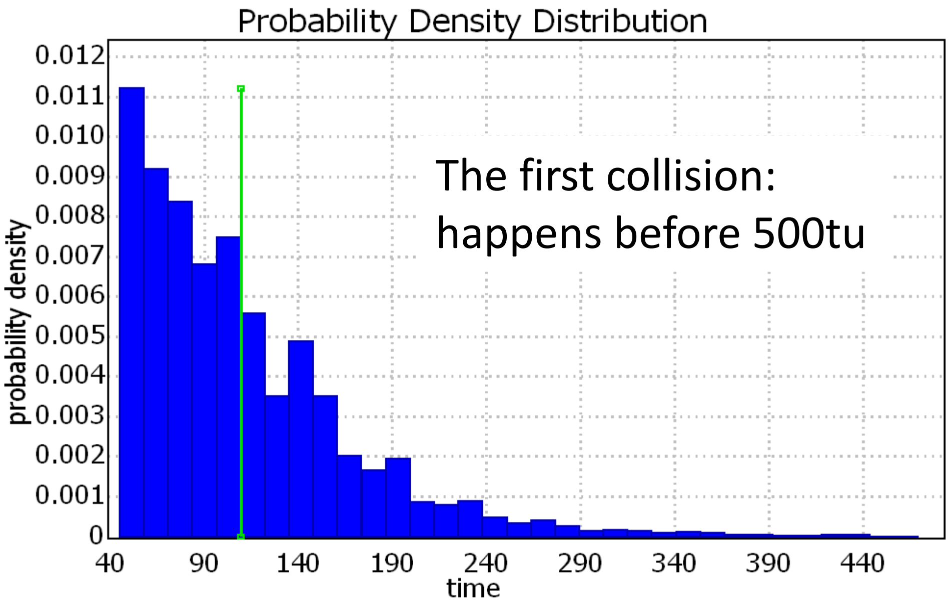


Collision counts after 1000tu

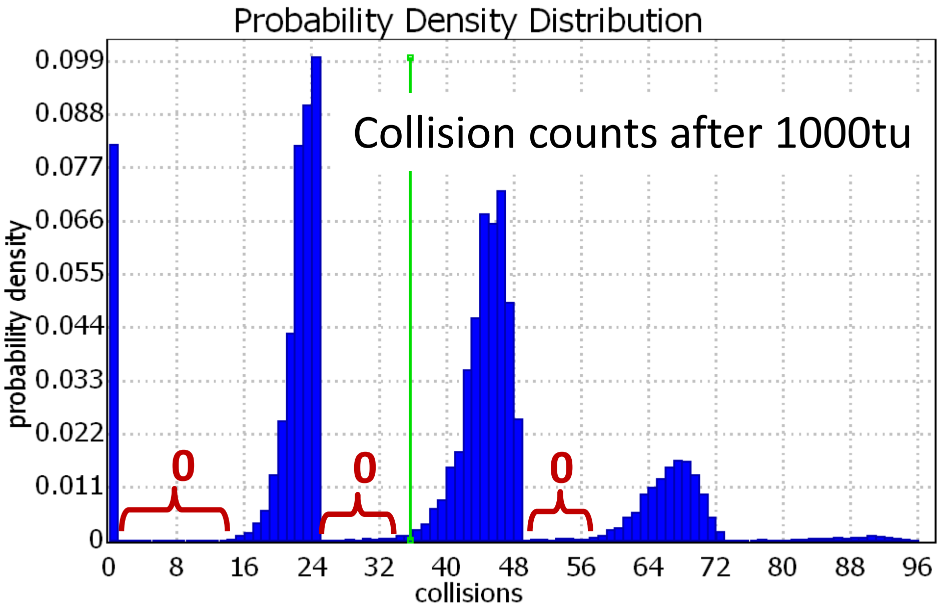
Collision counts after 2000tu: the numbers are doubled – perpetual collisions

The first collisions can be as late as **920**tu.
It is very likely (**>90%**) that
there will be **0** collisions.
But if they happen, they are perpetual.

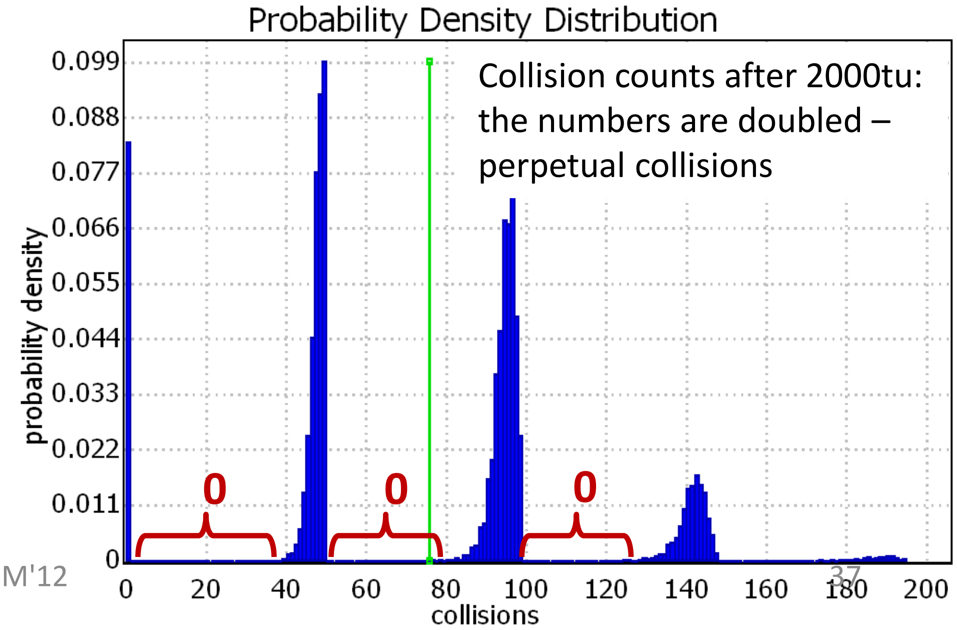
10-Node Star



The first collision: happens before 500tu



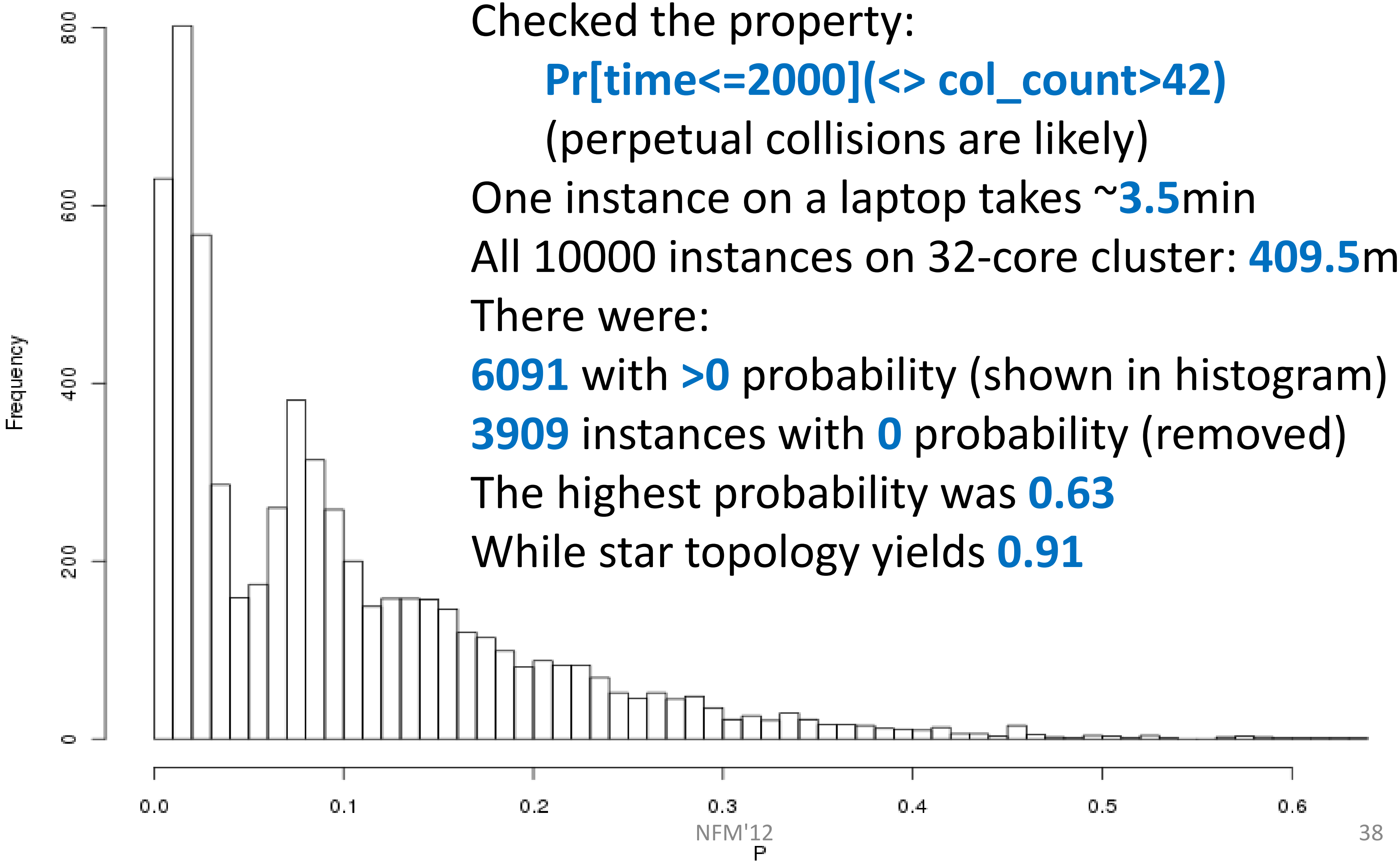
Collision counts after 1000tu



Collision counts after 2000tu: the numbers are doubled – perpetual collisions

The first collisions happen before **500**tu.
It is unlikely (**8.2%**) that
there will be **0** collisions.
And if they happen, they are perpetual.

10-Node Random Topologies



Generated **10000** random topologies

Checked the property:

Pr[time<=2000](<> col_count>42)

(perpetual collisions are likely)

One instance on a laptop takes ~**3.5**min

All 10000 instances on 32-core cluster: **409.5**min

There were:

6091 with **>0** probability (shown in histogram)

3909 instances with **0** probability (removed)

The highest probability was **0.63**

While star topology yields **0.91**

Conclusion

- Preliminary experiments indicate that distributed SMC in UPPAAL scales very nicely.

- More work to identify impact of parameters for distributing individual SMC?

- UPPAAL 4.1.9 available
  (support for SMC, DSMC, 64-bit,..)

End