Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols

Radu Mateescu and Wendelin Serwe

INRIA Grenoble Rhône-Alpes / LIG / VASY

http://vasy.inria.fr

Overview

- Mutual exclusion on shared-memory machines
- Formal description of mutex protocols in LNT
- Functional analysis by model checking using MCL
- Performance evaluation using IMCs
- Conclusion and future work

Mutual exclusion on shared-memory machines

- Long-standing problem in concurrent programming [Dijkstra-65]:
  - Protect a shared resource against concurrent non-atomic accesses from competing processes
  - Processes communicate by atomic read/write operations on shared variables
- Mutual exclusion protocols:
  - Ensure that at most one process accesses the resource
  - Guarantee the progress of execution
- Dozens of protocols proposed in the literature (see survey in [Anderson-Kim-Herman-03])
- Performance assessment mainly by experimental measures

➔ *our goal: provide model-based quantitative analysis*

Formal specification of mutual exclusion protocols

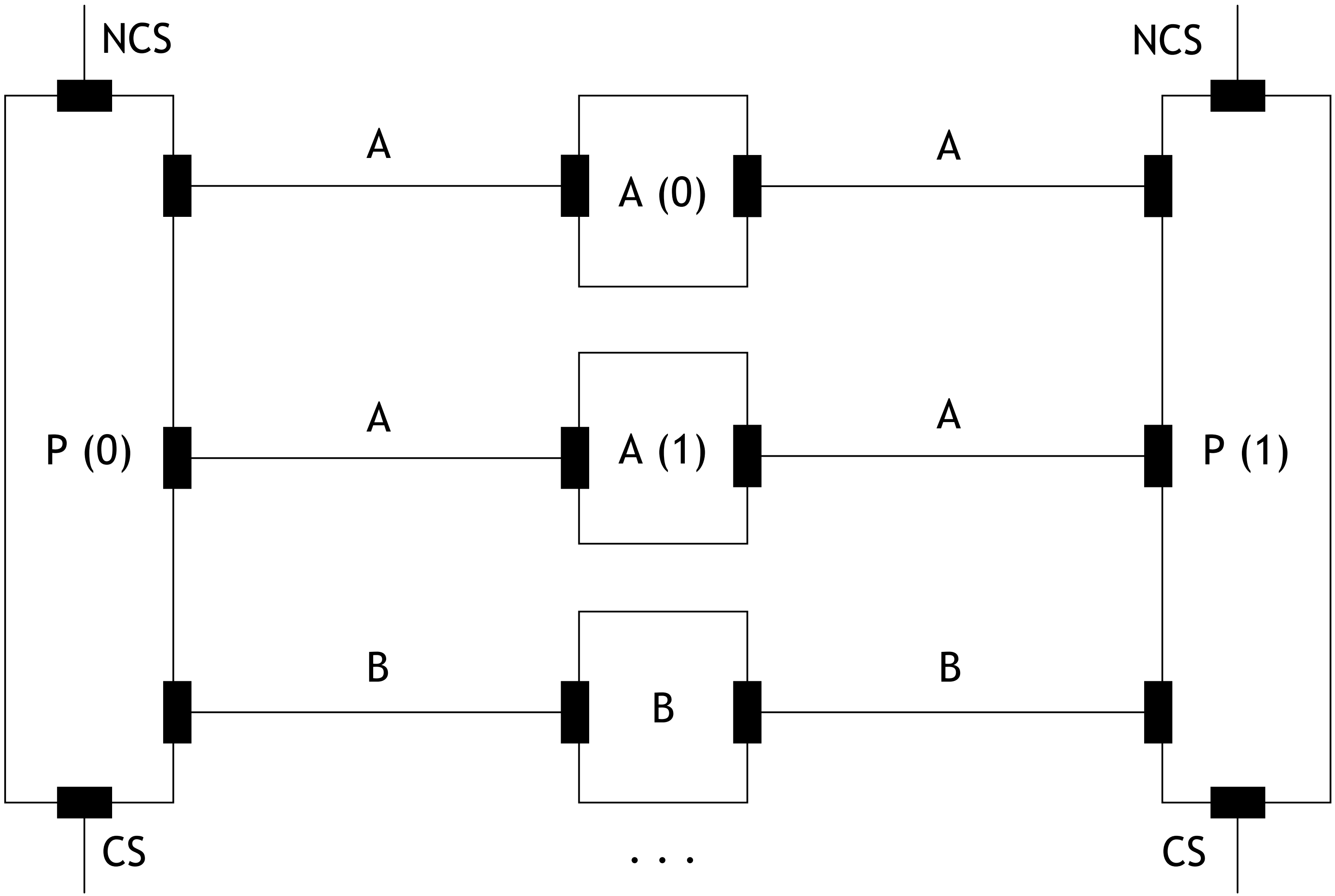
- LNT (LOTOS NT) language:
  - Combines process algebraic and imperative programming features
  - User-friendly syntax and formal semantics
  - Accepted as input by the CADP verification toolbox
- Specification of 27 mutex protocols in LNT:
  - Burns&Lynch [80], Craig and Landin&Hagersten [93-94], Dekker [68]
  - Dijkstra [65], Peterson [81], Knuth [66], Lamport [87]
  - Kessels [82], Mellor-Crummey&Scott [91], Szymanski [88]
  - black-white bakery protocol [Taubenfeld-04]
  - 12 protocols generated automatically [Bar-David-Taubenfeld-03]
  - array-based queue lock [Anderson-90]
  - test-and-set (TAS), test/test-and-set (TTAS) protocols [Anderson-90]
  - 1 trivial (incorrect) one-bit protocol for benchmarking purposes
- Analysis using the CADP toolbox (http://cadp.inria.fr)

Mutual exclusion protocols

- Structure of a concurrent process P competing for the access to the shared resource:

loop	
non critical section ;	➔ *may loop forever*
entry section ;	➔ *access shared variables*
critical section ;	➔ *access resource* *must terminate*
exit section	➔ *access shared variables*
end loop	

Architecture for two processes
(three shared variables)



Knuth's protocol
[Knuth-66]

three shared variables
A[0], A[1], B

$j \in \{ 0, 1 \}$
other process:
$k = 1 - j$

Process P_j

```
loop
      non critical section ;
      loop
         A[j] := 1 ;
         await B == j or A[k] == 0 ;
         A[j] := 2 ;
         if A[k] != 2 then break ;
      end loop ;
      B := j ;
      critical section ;
      B := k ;
      A[j] := 0
end loop
```

entry section

exit section

LNT specification
(architecture of the system)

```
par A, B, CS, NCS in
 par A, B in
  par
   P [NCS, CS, A, B] (0 of Nat)  ||  P [NCS, CS, A, B] (1 of Nat)
  end par
 ||
  par
   A [A] (0 of Nat, 0 of Nat)  ||  A [A] (1 of Nat, 0 of Nat)
  ||
   B [B] (0 of Nat)
  end par
 end par
||
 L [A, B, CS, NCS, MU]
end par
```

all shared variables are initially 0

LNT specification
(process Pj)

```
process P [NCS:Pid, CS:Access, A, B:Operation] (j:Nat) is
  var k, a_k, b:Nat in   k := 1 - j;
  loop
    NCS (!j);
    loop L1 in
      A (!Write, !j, !1 of Nat, !j);
      loop L2 in
        B (!Read, ?b, !j); A (!Read, !k, ?a_k, !j);
        if (b == j) or (a_k == 0) then break L2 end if
      end loop;
      A (!Write, !j, !2 of Nat, !j);
      A (!Read, !k, ?a_k, !j); if a_k != 2 then break L1 end if
    end loop;
    B (!Write, !j, !j);                          entry section
    CS (!Enter, !j); CS (!Leave, !j);
    B (!Write, !k, !j);
    A (!Write, !j, !0 of Nat, !j)                exit section
  end loop
  end var
end process
```

LNT specification
(shared variables)

```
process A [A:Operation]
              (index, val:Nat) is
  loop
    select
      A (!Read, !index, !val, ?any Nat)
    []
      A (!Write, !index, ?val, ?any Nat)
    end select
  end loop
end process
```

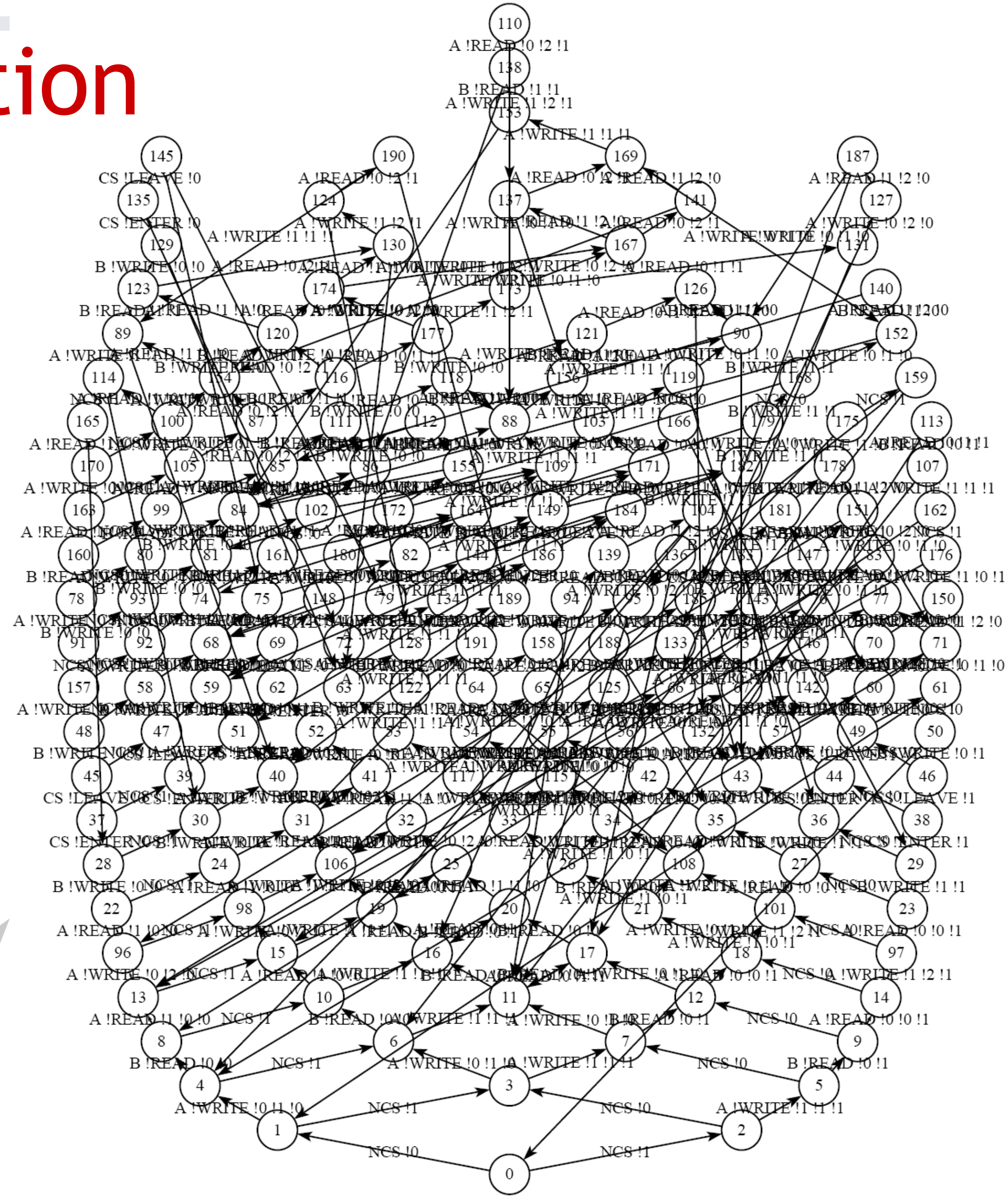
index (0, 1) of the two-cell array

```
process B [B:Operation] (val:Nat) is
  loop
    select
      B (!Read, !val, ?any Nat)
    []
      B (!Write, ?val, ?any Nat)
    end select
  end loop
end process
```

Labeled transition system

Tool support:
LNT.OPEN

- OPEN/CAESAR compliant compiler for LNT
- Allows the on-the-fly exploration of the LTSs corresponding to LNT specifications



LTS of Knuth's protocol
192 states, 384 transitions

Functional analysis by model checking

- Formulate the essential properties of mutex protocols in an action-based setting:
  - Mutual exclusion (safety)
  - Livelock freedom (liveness)
  - Starvation freedom (fairness)
  - Degree of overtaking (fairness)
  - Independent progress (fairness)
- Verify the properties on the LNT specifications:
  - Express properties in MCL
  - Use LNT.OPEN and EVALUATOR 4.0
  - Interpret diagnostics

MCL (Model Checking Language) [Mateescu-Thivolle-08]

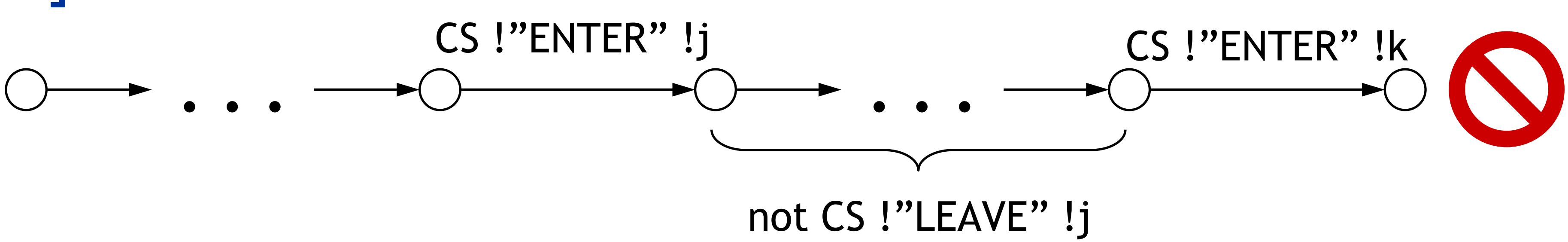
- Extension of modal μ-calculus with:
  - Regular expressions over action sequences [Mateescu-Sighireanu-03]
  - Modalities that extract data values from LTS labels
  - Fixed point operators parameterized by data variables
  - Constructs inspired from programming languages
- Tool support: EVALUATOR 4.0
  - On-the-fly verification of MCL formulas on LTSs
  - Diagnostic generation (examples and counterexamples)
  - Reusable libraries of derived operators (CTL, ACTL, ...) and property patterns [Dwyer-et-al-99]

Mutual exclusion
(safety)

Two processes can never execute simultaneously their critical sections.

```
[  true* .
   { CS !"ENTER" ?j:Nat } .
   (not { CS !"LEAVE" !j })* .
   { CS !"ENTER" ?k:Nat where k <> j }
]  false
```

> *fully parametric*
> MCL formula (depends only on information present on LTS transitions)



Livelock freedom
(first formulation – 2 processes)

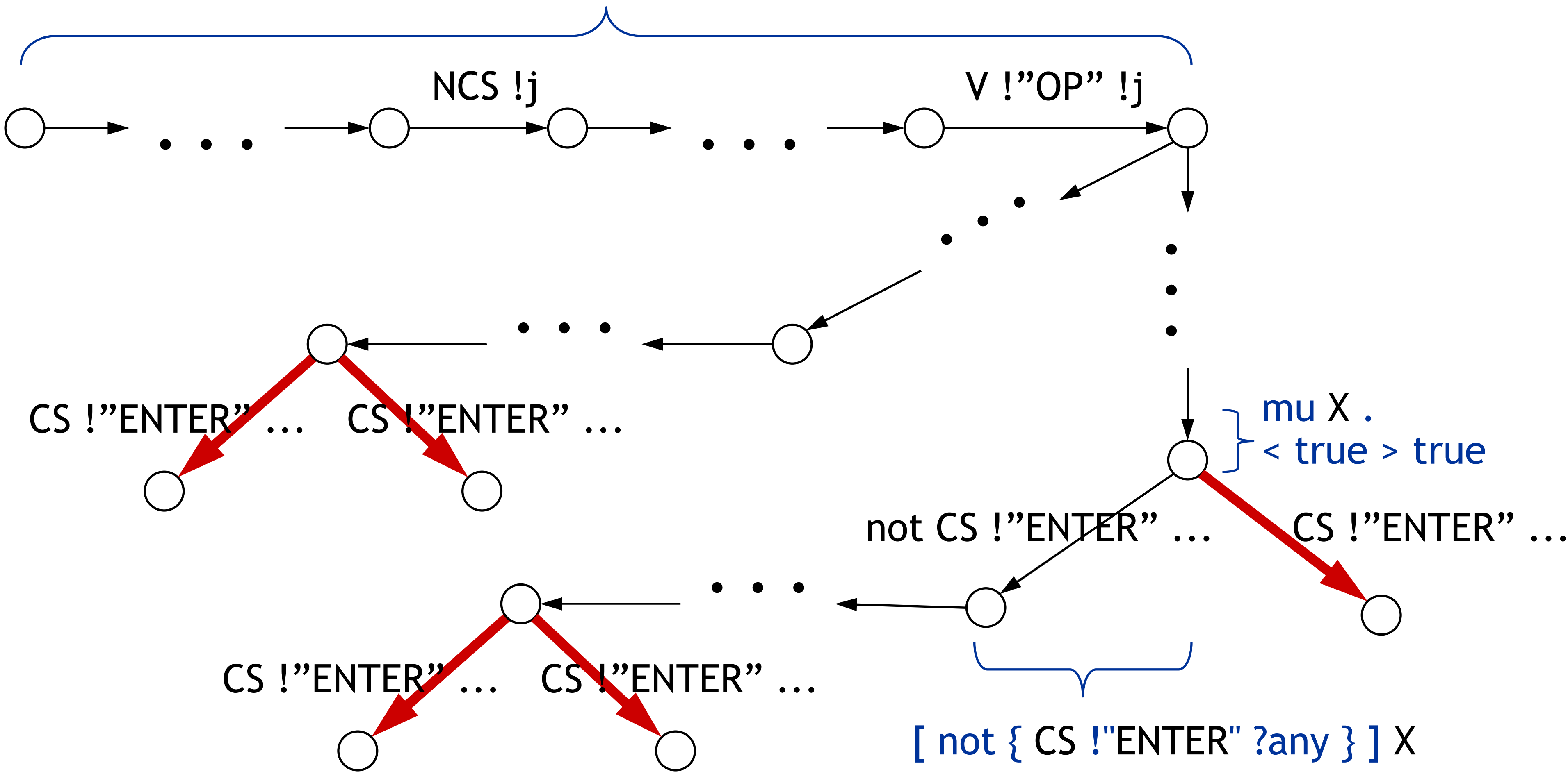
Each time a process is in its entry section, then some process will eventually enter its critical section.

```
[  true* . { NCS ?j:Nat } .
   (not { ?any ?"READ"|"WRITE" ... !j })* .
   { ?any ?"READ"|"WRITE" ... !j }
]  mu X . (< true > true and
                [ not { CS !"ENTER" ?any } ] X)
```

➔ *this formula fails on all mutex protocols!*

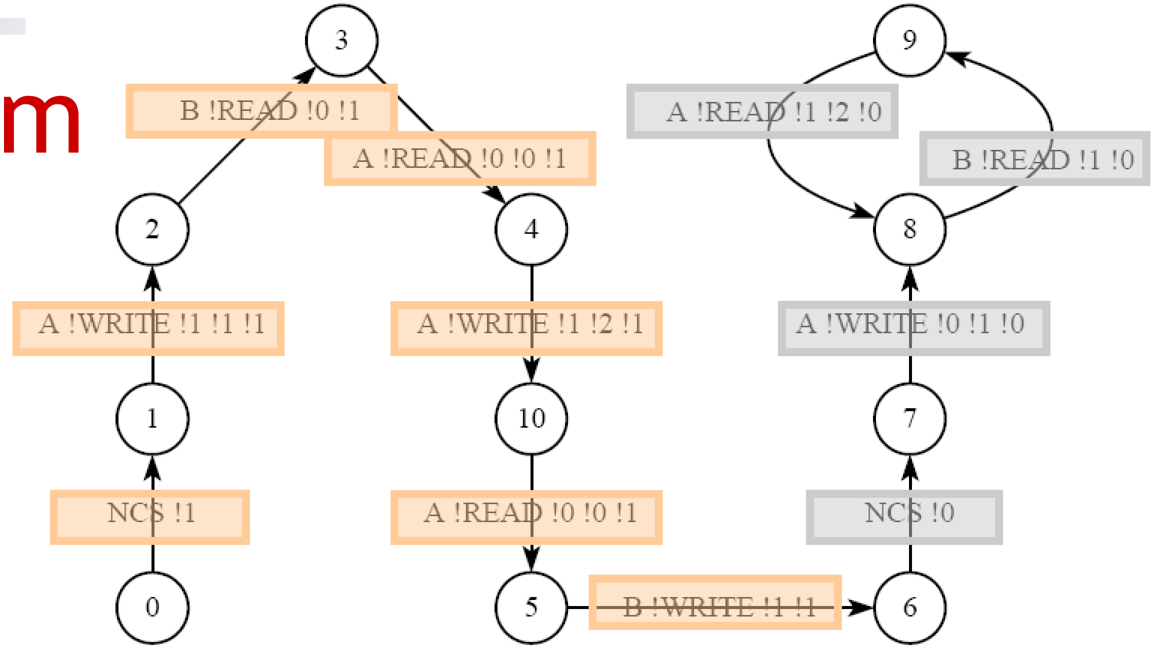
Livelock freedom – LTS view
(first formulation – 2 processes)

[true* . { NCS ?j:Nat } . (not { ?any ?“R”|”W” … !j })* . { ?any !“R”|”W” … !j }]



Livelock freedom
(first formulation)

- Counterexample for Knuth's protocol:




```
loop
  non critical section ;
  loop
    A[0] := 1 ;
    await B == 0 or A[1] == 0 ;
    A[0] := 2 ;
    if A[1] != 2 then break ;
  end loop ;
  B := 0 ;
  critical section ;
  B := 1 ; A[0] := 0
end loop
```

P_0

```
loop
  non critical section ;
  loop
    A[1] := 1 ;
    await B == 1 or A[0] == 0 ;
    A[1] := 2 ;
    if A[0] != 2 then break ;
  end loop ;
  B := 1 ;
  critical section ;
  B := 0 ; A[1] := 0
end loop
```

P_1

Livelock freedom

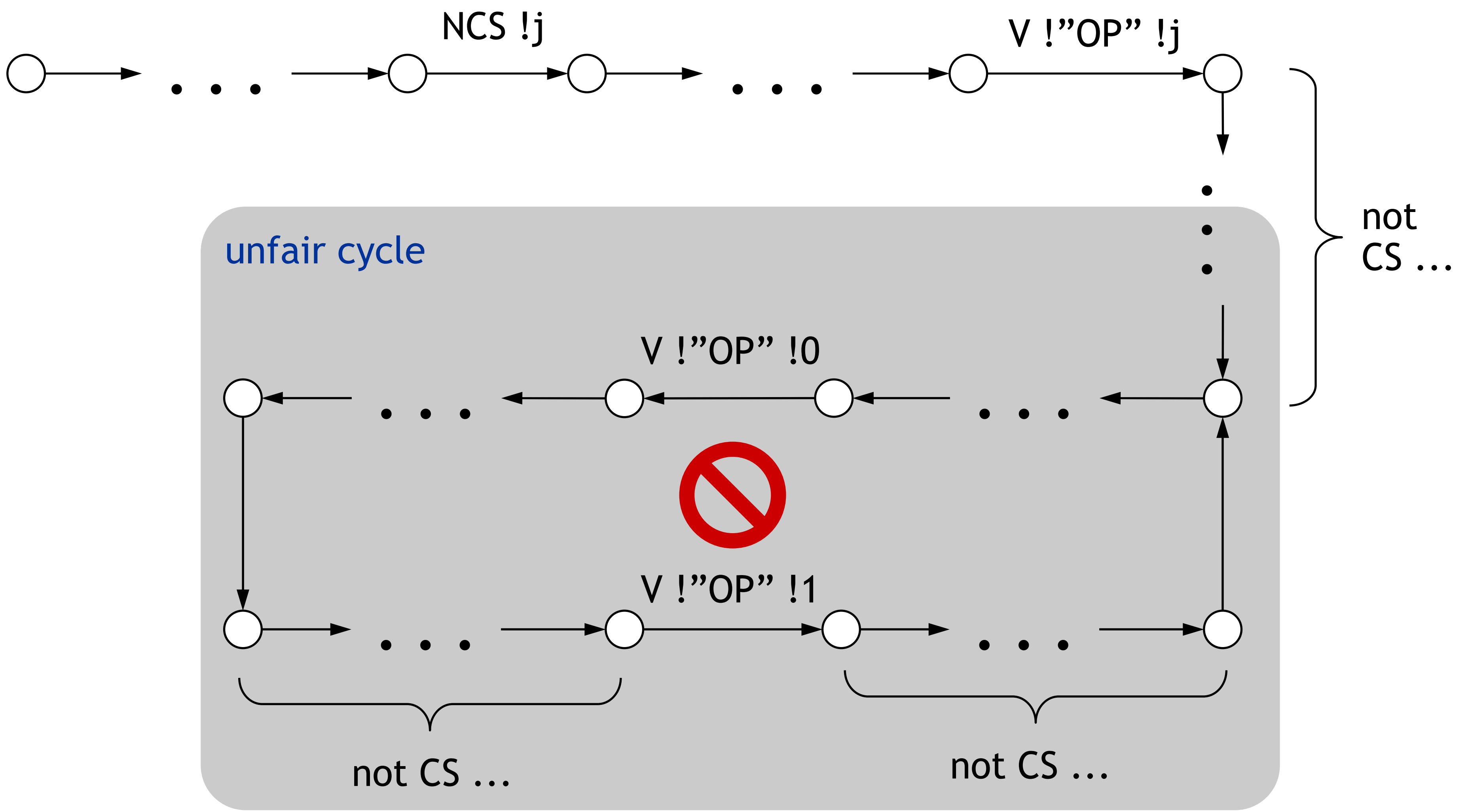
(second formulation – 2 processes [BDT-03])

There is no cycle in which each process executes an instruction but no one enters its critical section.

```
[  true* . { NCS ?j:Nat } .
   (not { ?any ?"READ"|"WRITE" ... !j })* .
   { ?any ?"READ"|"WRITE" ... !j }
] not < (not { CS ... })* .
          { ?G:String ... ?k:Nat where G <> "CS" } .
          (not { CS ... })* .
          { ?G:String ... !1 - k where G <> "CS" }
      > @
```

➔ *holds on all mutex protocols*

Livelock freedom – LTS view
(negation of second formulation – 2 processes)



Livelock freedom
(second formulation – *n* processes)

There is no cycle in which each process executes an instruction but no one enters its critical section.

```
[  true* . { NCS ?j:Nat } .
   (not { ?any ?"READ"|"WRITE" ... !j })* .
   { ?any ?"READ"|"WRITE" ... !j }
] not < for j:Nat from 0 to n – 1 do
              (not { CS ... })* .
              { ?G:String ... !j where G <> "CS" }
          end for
        > @
```

complex cycle containing a set of events (generalized Büchi automaton)

➔ *holds on all mutex protocols*

Starvation freedom
(fairness – 2 processes)

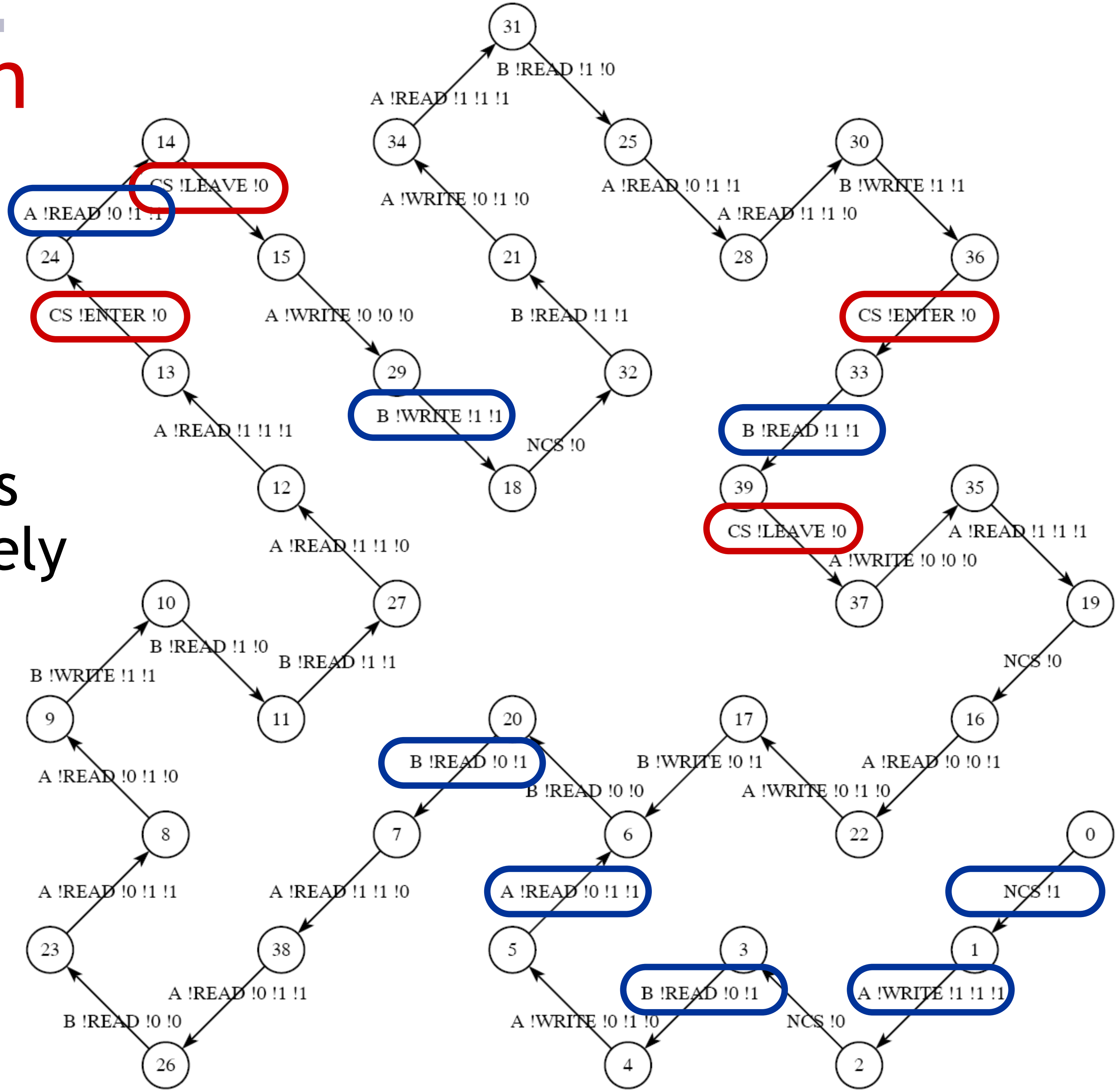
Each time a process is in its entry section, then that process will eventually enter its critical section.

```
[  true* . { NCS ?j:Nat } .
   (not { ?any ?"READ"|"WRITE" ... !j })* .
   { ?any ?"READ"|"WRITE" ... !j }
] not < (not { CS ... !j })* . { ?G:String ... ?k:Nat
                            where (G <> "CS") or (k <> j) } .
          (not { CS ... !j })* . { ?G:String ... !1 – k
                            where (G <> "CS") or ((1 - k) <> j) }
      > @
```

➔ *holds on some mutex protocols*

Starvation witness

- Protocol 3b_p2 [BDT-03]
- P_0 overtakes P_1 indefinitely



Bounded overtaking

(fairness)

How many times a process P_i can be overtook by another process P_j in accessing the critical section?

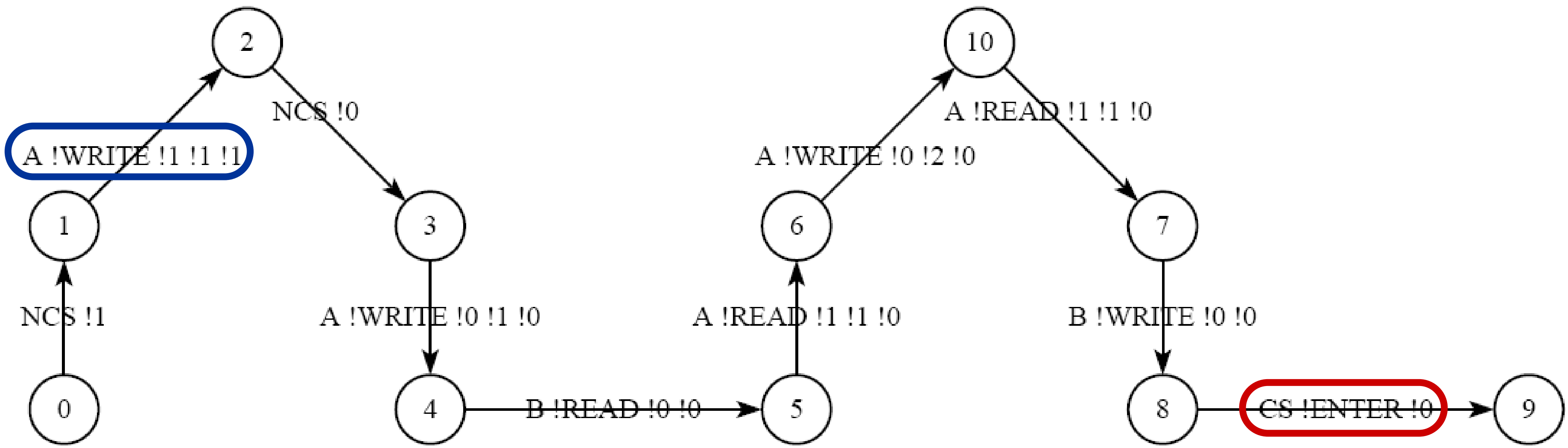
```
< true* . { NCS !i } .
    (not { ?any ?"READ" | "WRITE" ... !i })* .
    { ?any ?"READ" | "WRITE" ... !i } .
    (   (not { CS ?any !i })* .
        { ?G:String ... !i where G <> "CS" } .
        (not { CS ?any !i })* . { CS !"ENTER" !j }
    ) { overtaking_times }
> true
```

regular formula with counting: overtaking degree of P_i by P_j

P_j overtakes P_i

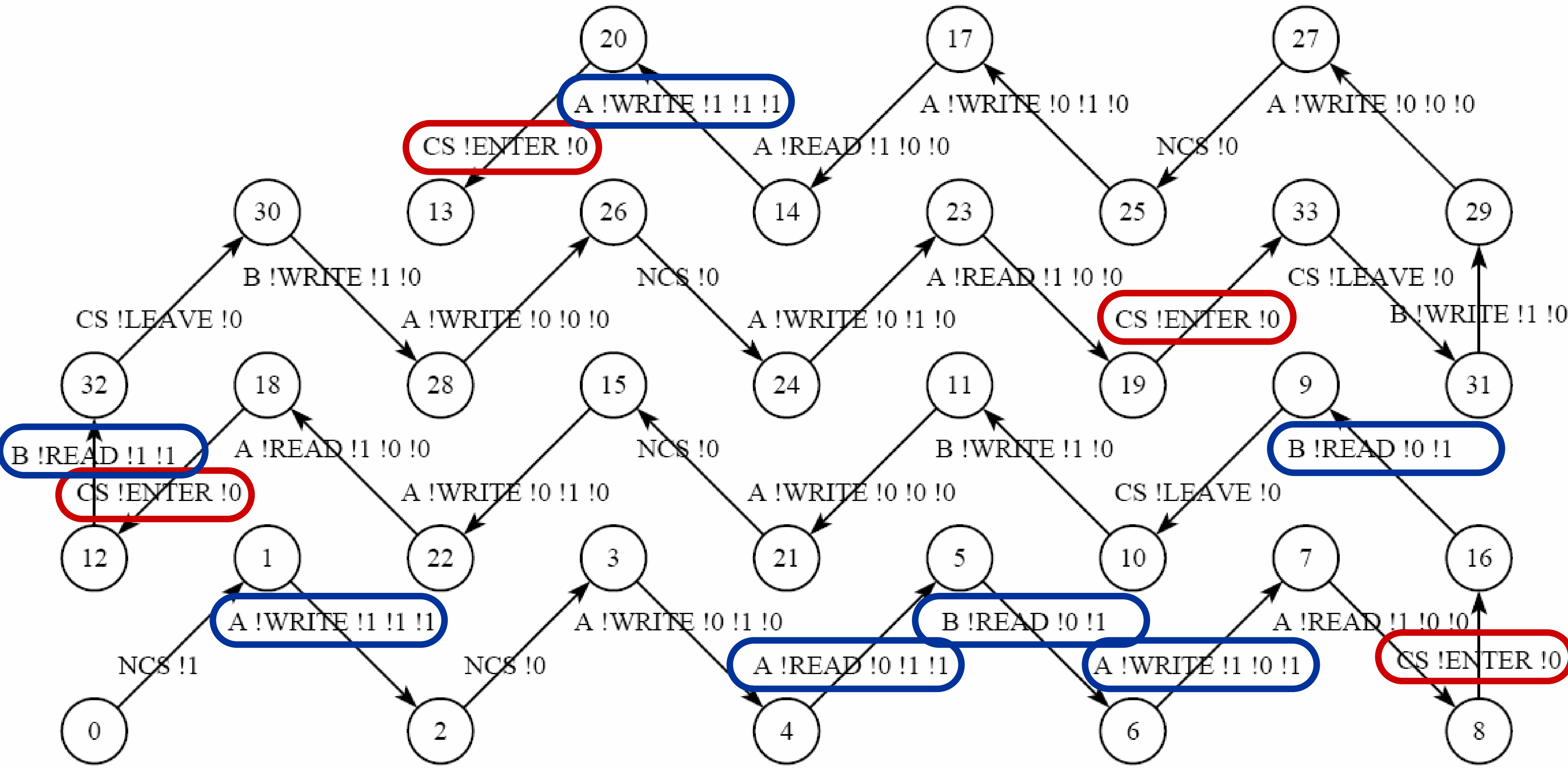
Witness of maximum overtaking

- Knuth's protocol for two processes (at most 1 overtake of P_1 by P_0):



Witness of maximum overtaking

- Dekker's protocol for two processes (at most 4 overtakes of P_1 by P_0):



Independent progress

[Dijkstra-65]

If a process stops in its non critical section, the other processes can still access their critical sections.

```
forall j:Nat among { 0 … 1 } .
   [ true* ] (
      < { NCS !1 - j } > true
      implies
      <  { … !j }* . { CS !"ENTER" !j } .
          { … !j }* . { CS !"LEAVE" !j }
      > @
   )
```

P_k stops at the beginning of its entry section

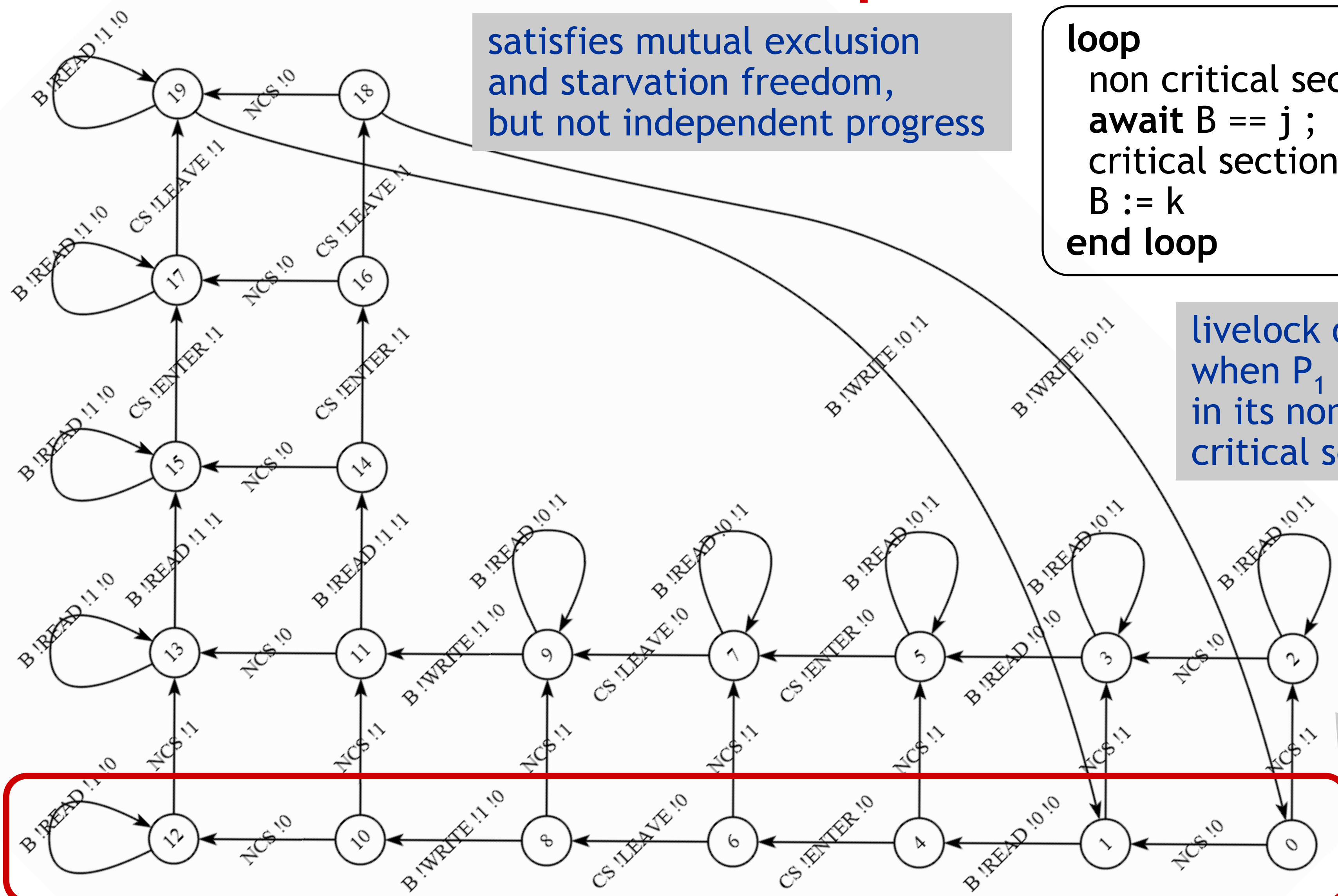
➔ *holds on all mutex protocols, but should be checked separately*

Trivial one-bit protocol

satisfies mutual exclusion
and starvation freedom,
but not independent progress

```
loop
  non critical section ;
  await B == j ;
  critical section ;
  B := k
end loop
```

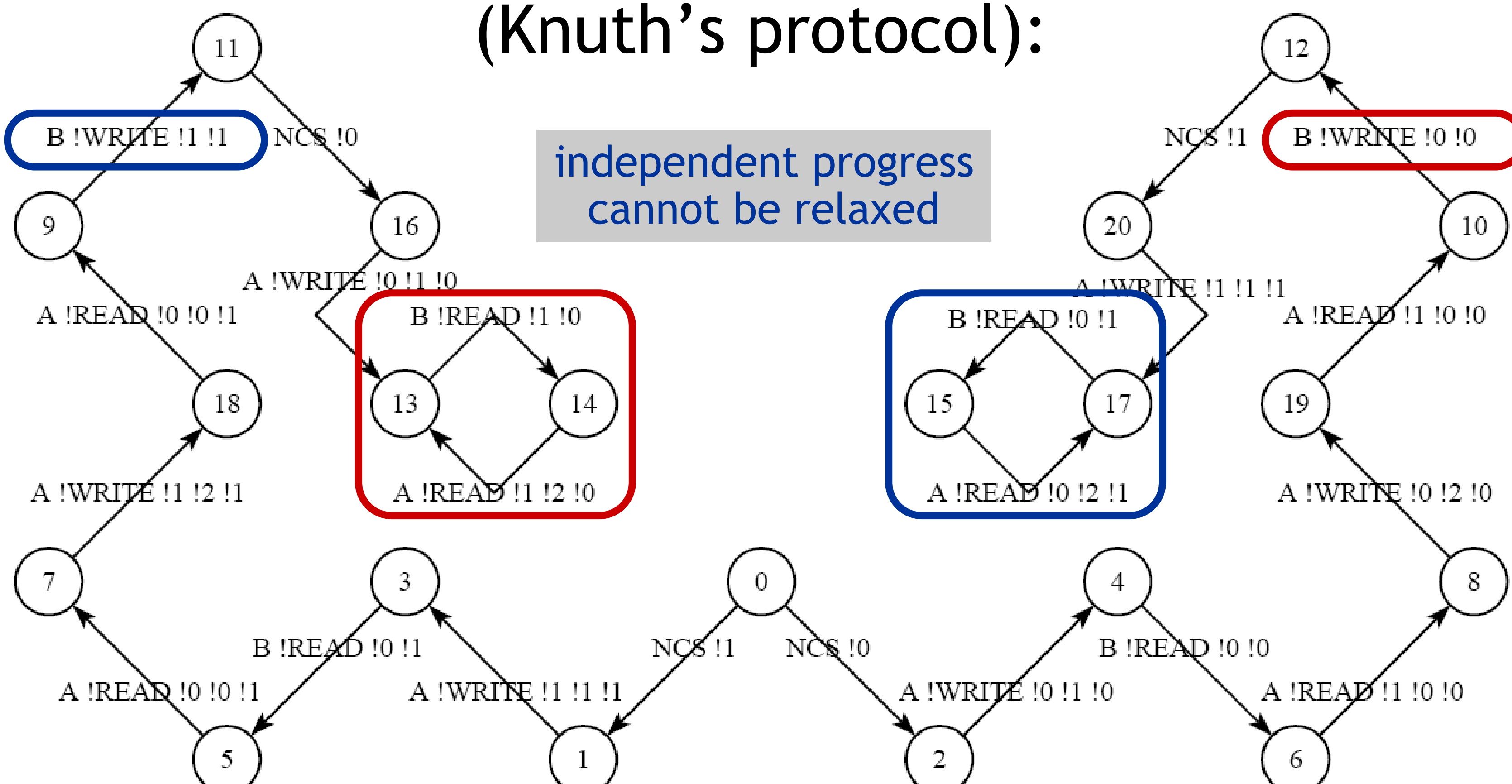
P_j



livelock of P_0 when P_1 stops in its non critical section

Livelock upon crash
(outside the non critical sections)

Livelock of each process when the other one "has decided to stop" in its entry section (Knuth's protocol):



Model checking summary (2 processes)

Protocol (2 processes)	Livelock-free	Starvation-free	Independent progress	Overtaking	
				P_0/P_1	P_1/P_0
Anderson	all	all	all	1	1
Burns & Lynch	all	P_0	all	∞	1
B&W Bakery	all	all	all	2	2
CLH	all	all	all	1	1
Dekker	all	all	all	4	4
Dijkstra	all	none	all	∞	∞
Kessels	all	all	all	2	2
Knuth	all	all	all	1	1
Lamport	all	none	all	∞	∞
MCS	all	all	all	1	1
Peterson	all	all	all	1	1
Peterson_t	all	all	all	1	1
Szymanski	all	all	all	2	1

Model checking summary (2 processes)

Protocol (2 processes)	Livelock-free	Starvation-free	Independent progress	Overtaking	
				P_0/P_1	P_1/P_0
2b_p1	all	P_0	all	∞	1
2b_p2	all	P_0	all	∞	1
2b_p3	all	P_1	all	1	∞
3b_p1	all	all	all	2	2
3b_p2	all	P_0	all	∞	1
3b_c_p1_orig	all	all	all	1	1
3b_c_p1	all	all	all	1	1
3b_c_p2	all	all	all	1	1
3b_c_p3	all	all	all	1	1
4b_p1	all	P_0	all	∞	1
4b_p2	all	all	all	2	2
4b_c_p1	all	P_0	all	∞	1
4b_c_p2	all	P_1	all	1	∞
tas	all	none	all	∞	∞
ttas	all	none	all	∞	∞
trivial	all	all	none	1	1

Model checking summary (3 processes)

Protocol (3 processes)	Livelock-free	Starv.-free	Indep. progress	Overtaking					
				P_0/P_1	P_0/P_2	P_1/P_0	P_1/P_2	P_2/P_0	P_2/P_1
Anderson	all	all	all	1	1	1	1	1	1
Burns & Lynch	all	P_0	all	∞	∞	1	∞	1	∞
B&W Bakery	all	all	all	2	2	2	2	2	2
CLH	all	all	all	1	1	1	1	1	1
Dijkstra	all	none	all	∞	∞	∞	∞	∞	∞
Knuth	all	all	all	1	2	2	1	1	2
Lamport	all	none	all	∞	∞	∞	∞	∞	∞
MCS	all	all	all	1	1	1	1	1	1
Peterson	all	all	all	6	6	6	6	6	6
Peterson_t	all	all	all	1	1	1	1	12	12
Szymanski	all	all	all	2	2	1	2	1	1
tas	all	none	all	∞	∞	∞	∞	∞	∞
ttas	all	none	all	∞	∞	∞	∞	∞	∞
trivial	all	all	none	1	1	1	1	1	1

Performance evaluation using IMCs

- A single model for both
  functional verification + *performance evaluation*
- Enrich LNT model with (exponential) delays
  - constraint-oriented style: composition with a process L
  - each action corresponds to the begin of a delay
  - process L enforces alternation of delays and actions
- Compute steady-state probabilities on the underlying continuous time Markov chain (CTMC)
- Tool support by CADP
  - BCG_MIN: minimization
  - BCG_STEADY: computation of steady-state probabilities
  - CUNCTATOR: on-the-fly steady-state simulation

LNT specification
(auxiliary process for delay insertion)

```
process L [A, B: Operation, CS: Access, NCS: Pid, MU: Latency] is
    var index, pid:Nat, sig:Signal in
        loop
            select
                A (!Read, ?index, ?any Nat, ?pid);  MU (!Read, !index, !pid)
             [] A (!Write, ?index, ?any Nat, ?pid); MU (!Write, !index, !pid)
             [] B (!Read, ?any Nat, ?pid);  MU (!Read, !pid)
             [] B (!Write, ?any Nat, ?pid); MU (!Write, !pid)
             [] CS (?sig, ?pid); if sig == Enter then MU (!sig, !pid) end if
             [] NCS (?pid); MU (!Work, !pid)
            end select
        end loop
    end var
end process
```

Continuous-Time Markov Chains (CTMCs) in the BCG format

- Syntax of actions (transition labels):
  - Stochastic transitions "rate %f"
  - Labeled stochastic transition "*action*; rate %f"
  - Internal transition "i"

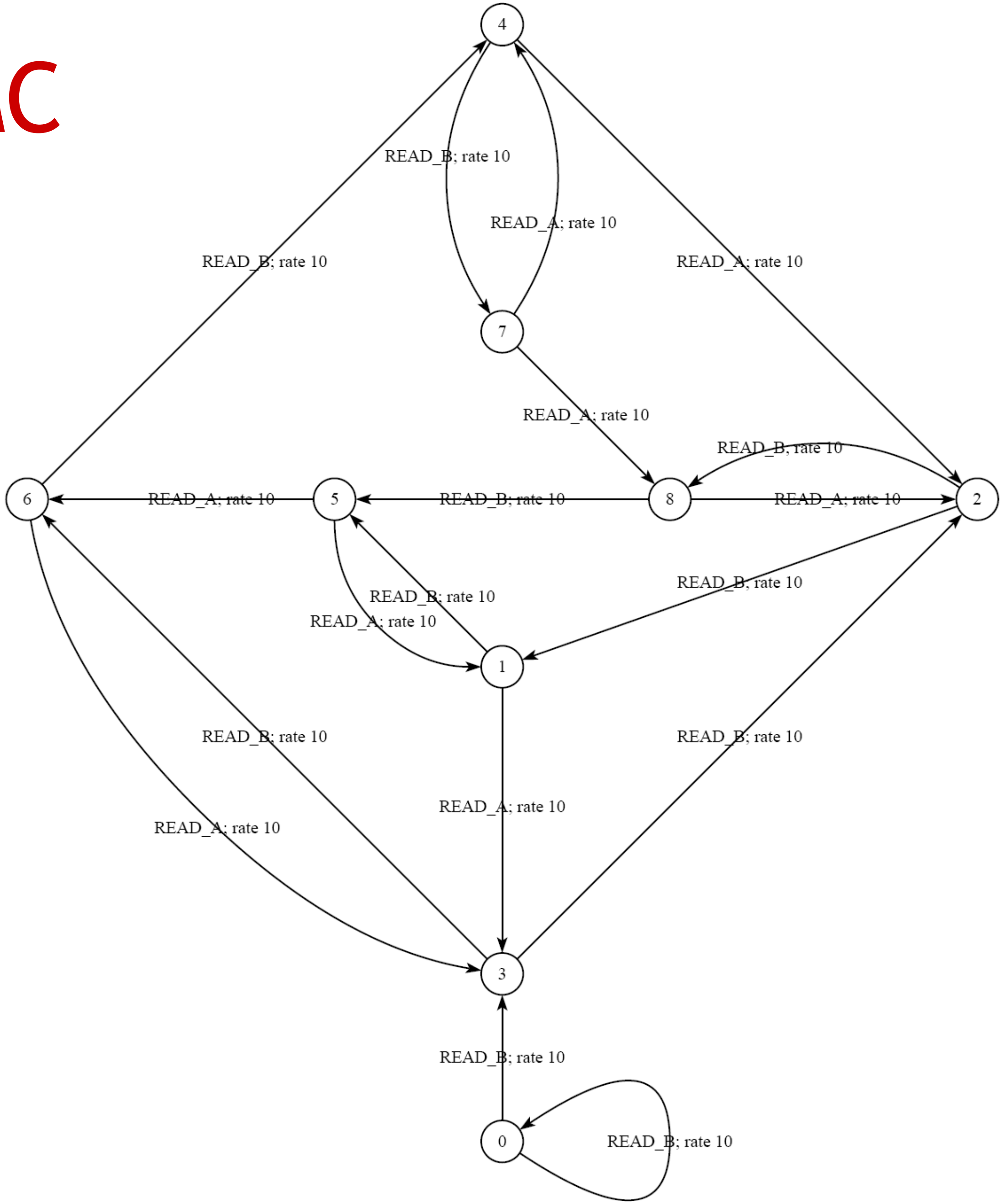
strictly positive floating-point number

character string without ';'

- Terminology for states:
  - **Stable** state (without i-successors)
  - **Unstable** state (with some i-successors)
  - **Nondeterministic** state (with at least two i-successors)

Example of CTMC

- Mutual exclusion protocol with three shared variables
- CTMC contains only read accesses to shared variables



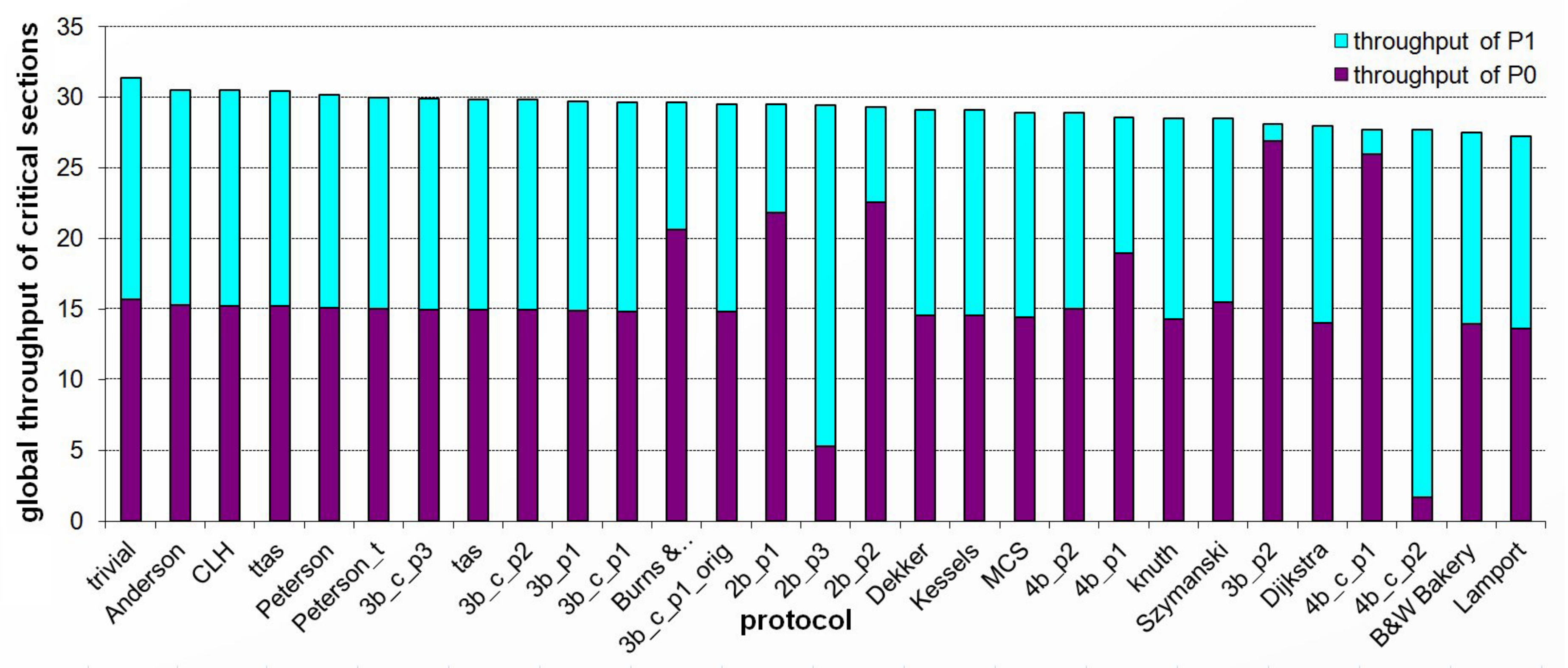
Dealing with nondeterminism

- Numerous nondeterministic (2-branch) choices due to concurrent accesses of P_0, P_1 to shared variables
- Work-around: model a *fair scheduler* replacing an equiprobable probabilistic choice
- Performance evaluation approach:
  - hide accesses to shared variables
  - minimize for stochastic branching bisimulation
  - rename remaining "i"-transitions into "prob 0.5"
    ➔ yields a "continuous-time probabilistic Markov chain" a graph with stochastic and probabilistic transitions
  - compute steady-state throughputs using BCG_STEADY (on constructed graphs) or CUNCTATOR (on the fly)

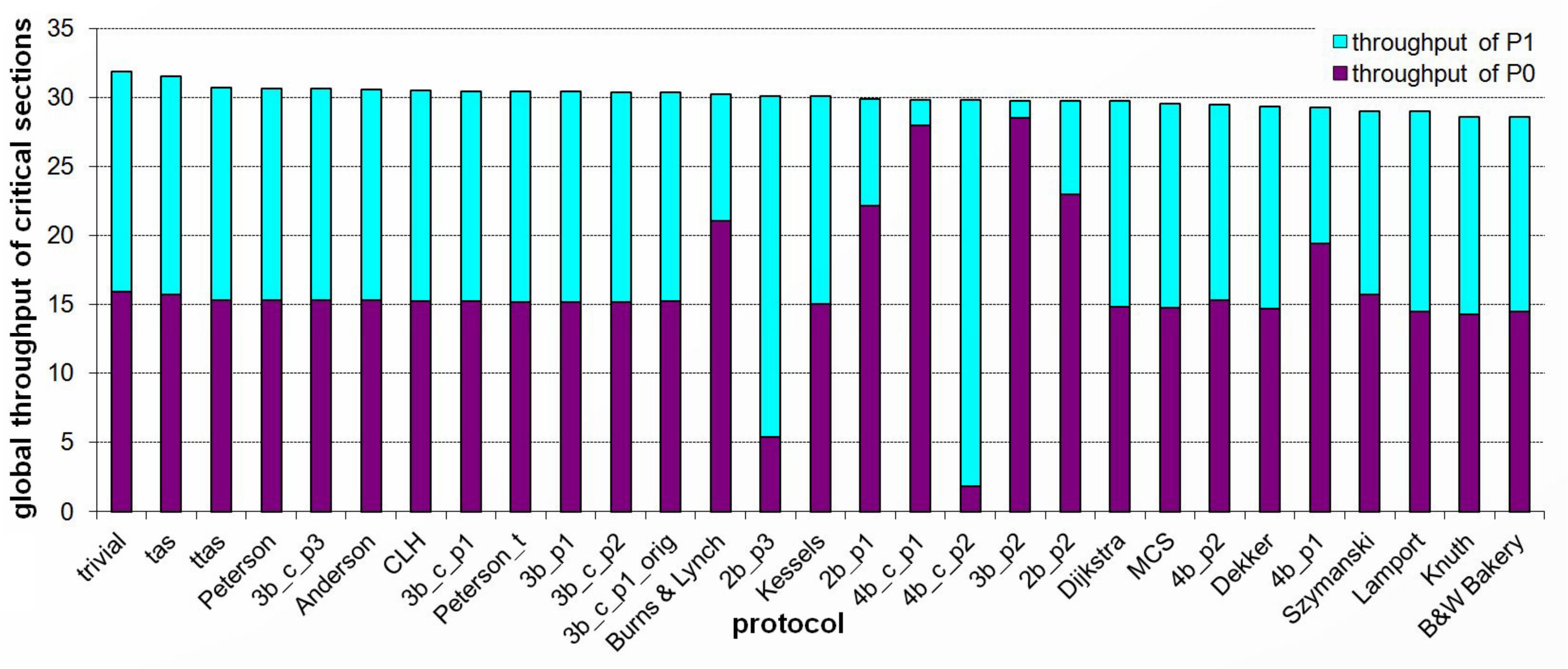
Performance experiments

- Goal: detect tendencies, no absolute values
- Throughput of the critical section:
  - relative (one process only)
  - cumulative (sum of both processes)
- Common rate parameters:
  - read access:
    - 3000 (global memory), 150000 (local cache)
  - write/fetch&store/compare&swap access:
    - 2000 (global memory), 135000 (local cache)
  - critical section: 100
- Varying rate for the non-critical section(s)

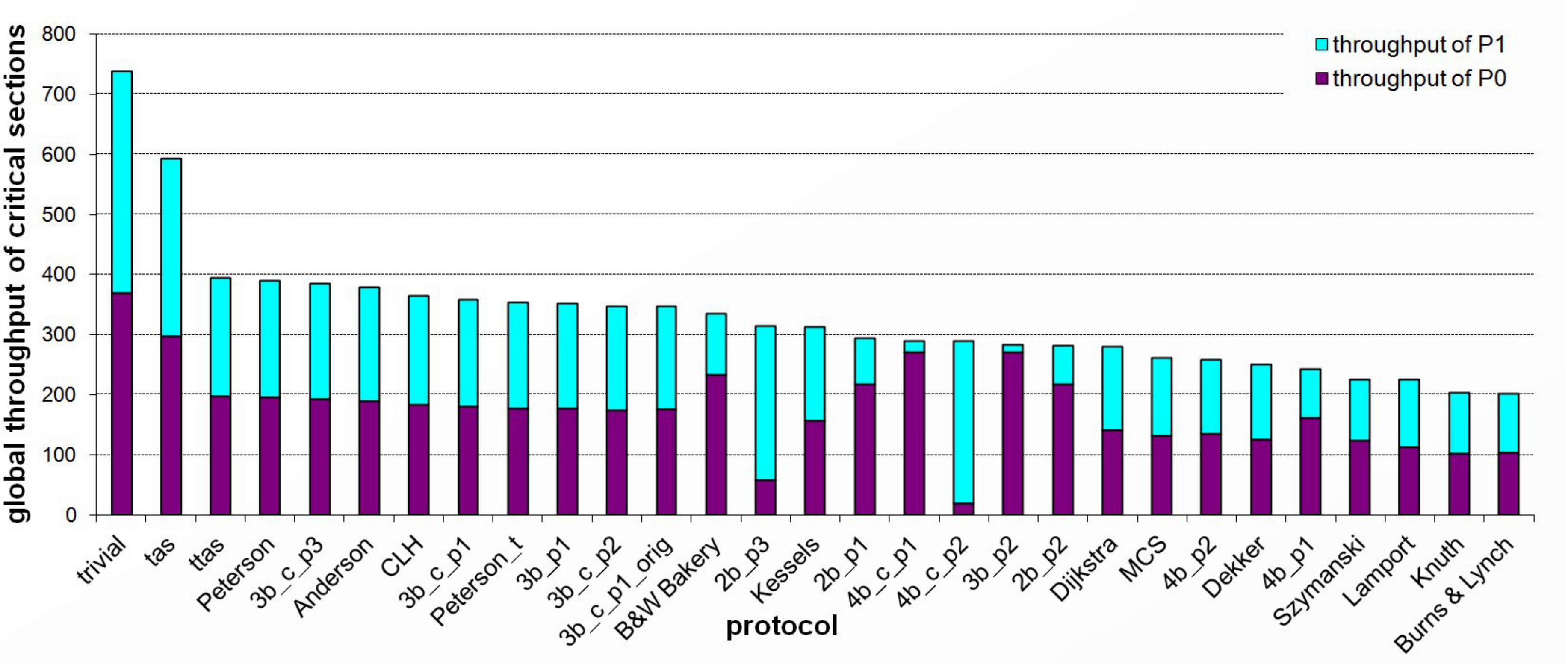
Global throughput without caching
(2 processes)



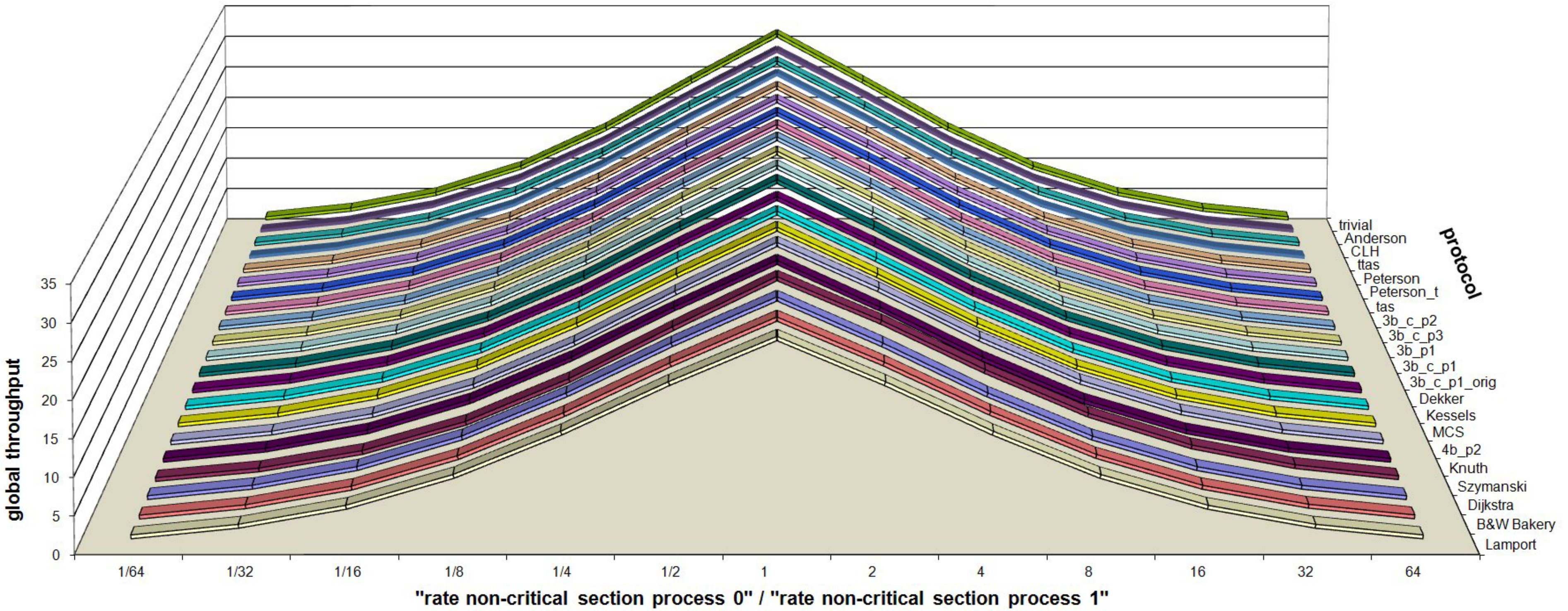
Global throughput with caching
(2 processes)



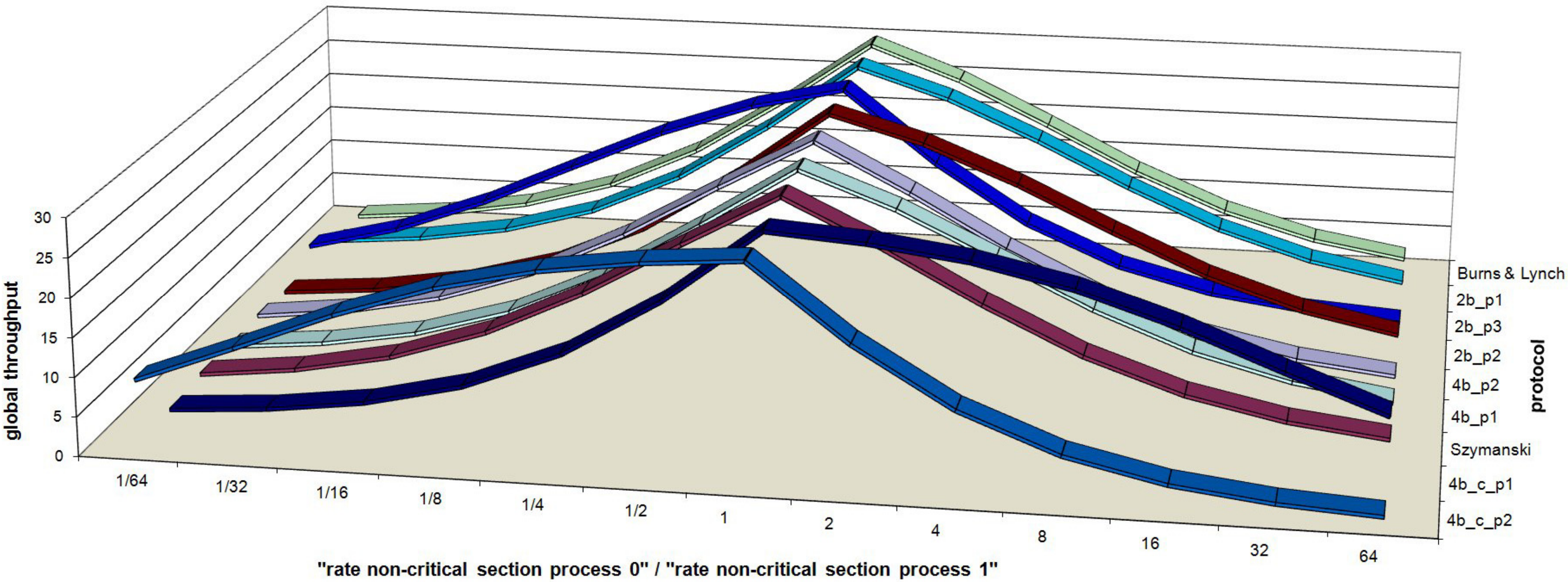
Global throughput with caching
(2 processes, very short critical section)



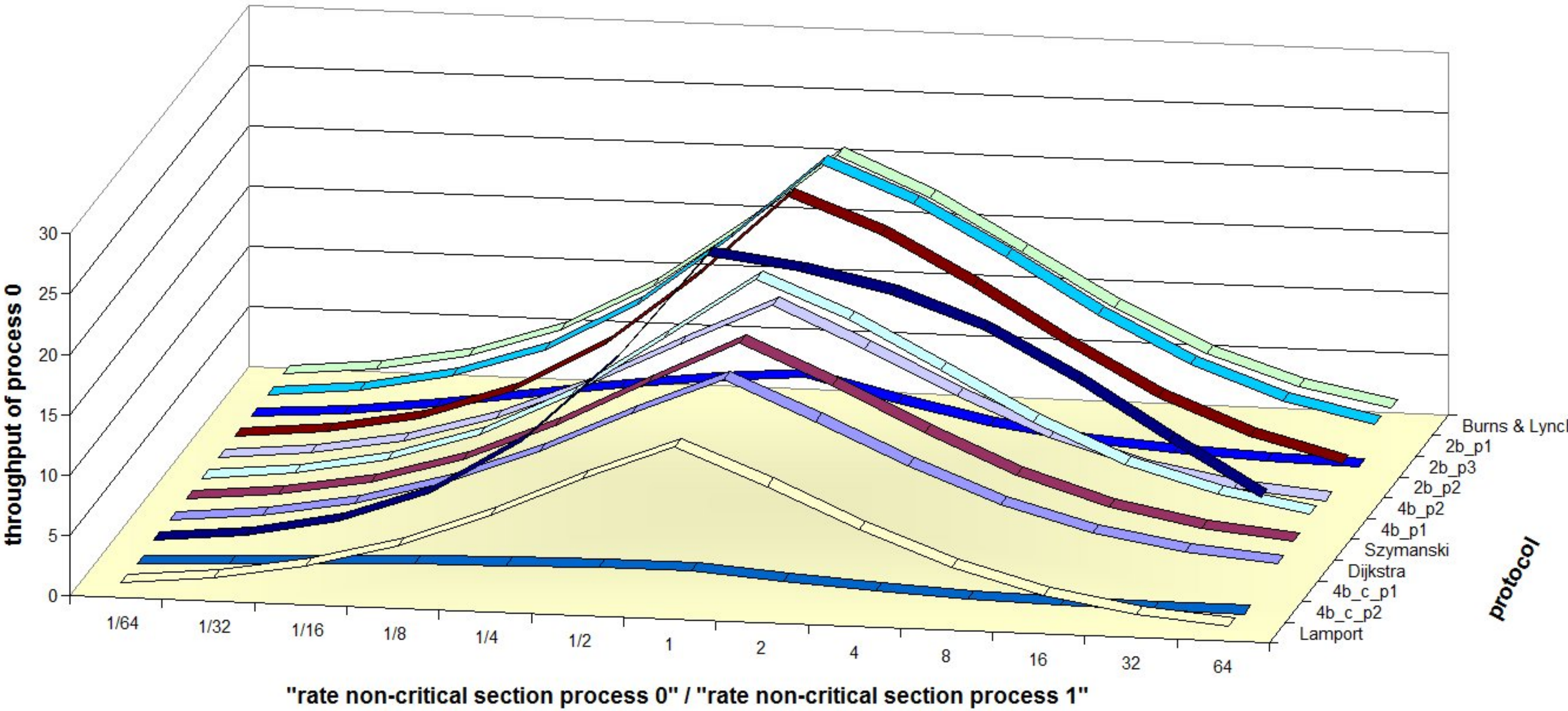
Global throughput for symmetric protocols
(2 processes)



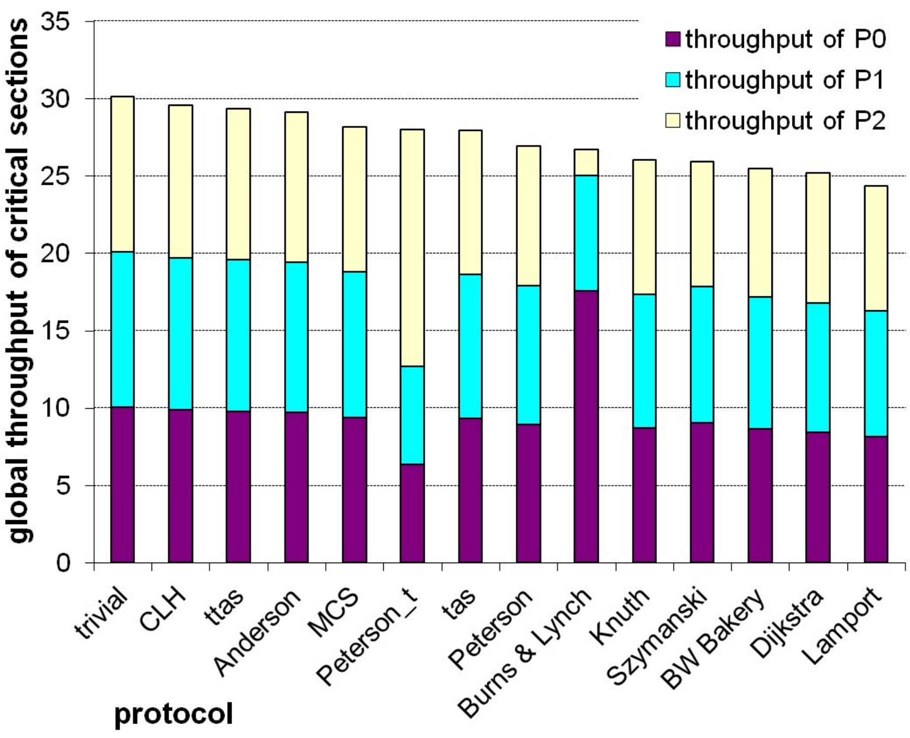
Global throughput for asymmetric protocols
(2 processes)

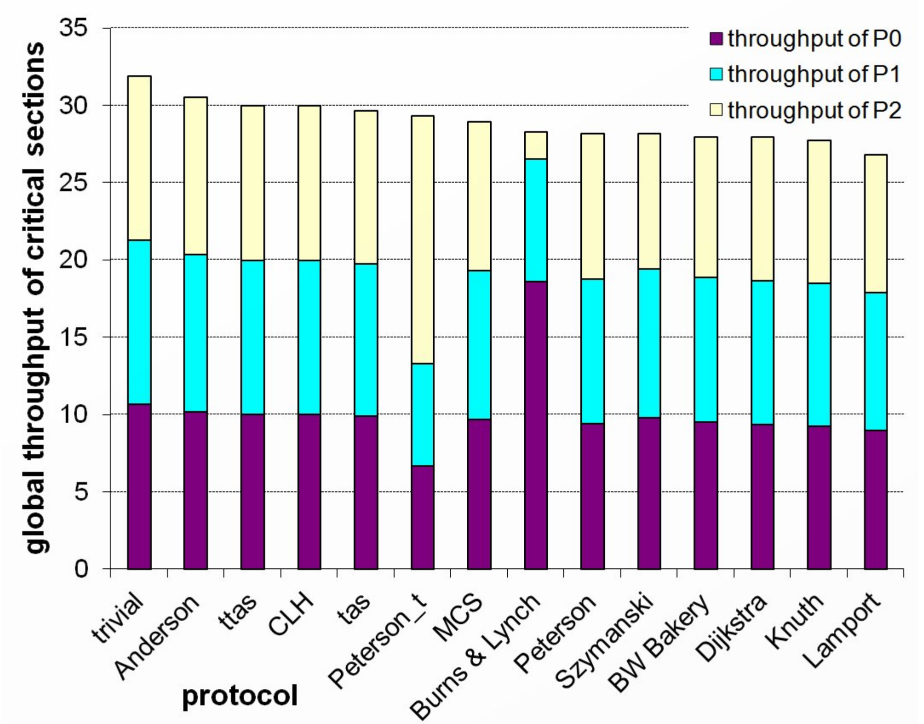


Throughput of process P_0 for asymmetric protocols
(2 processes)

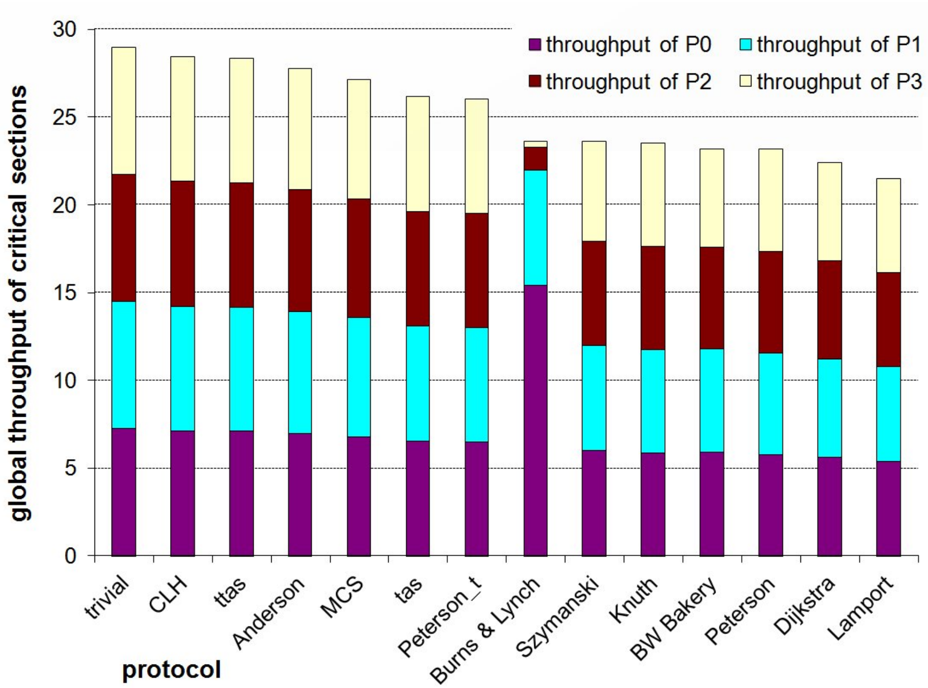


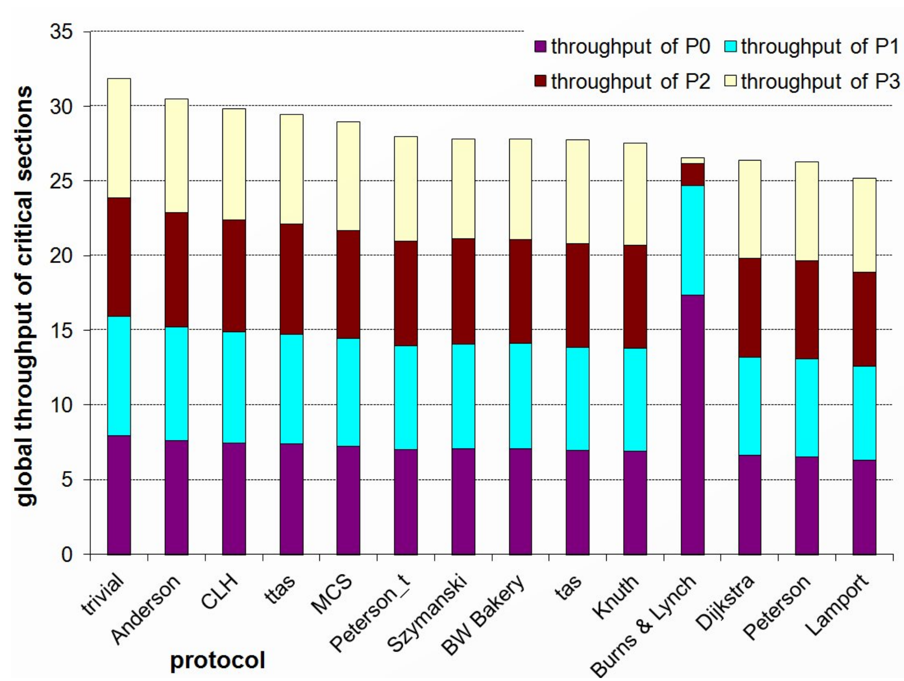
Global throughput with/without caching
(3 processes, CS twice as fast as NCS)



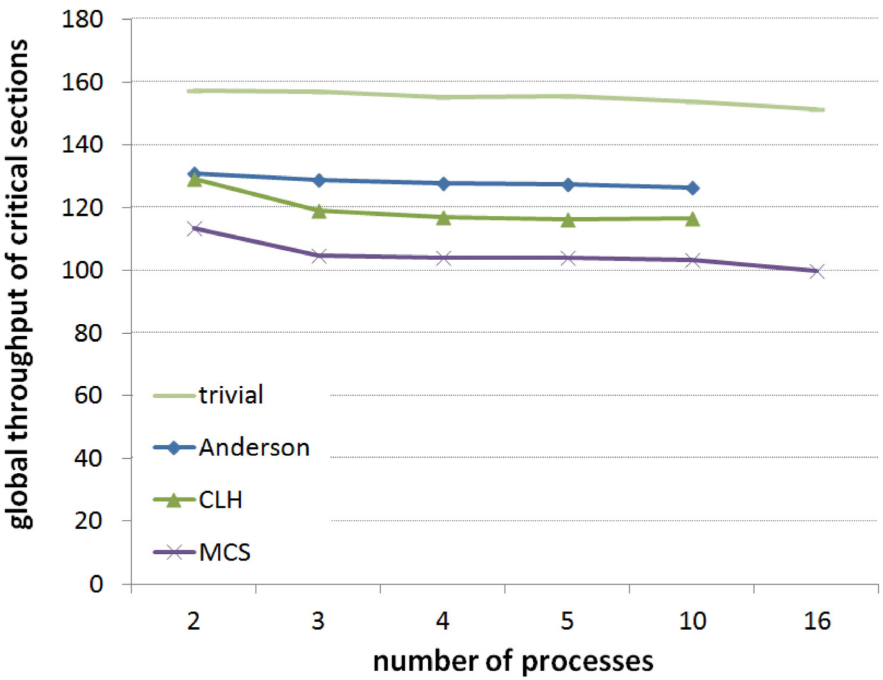


Global throughput with/without caching
(4 processes, CS twice as fast as NCS)

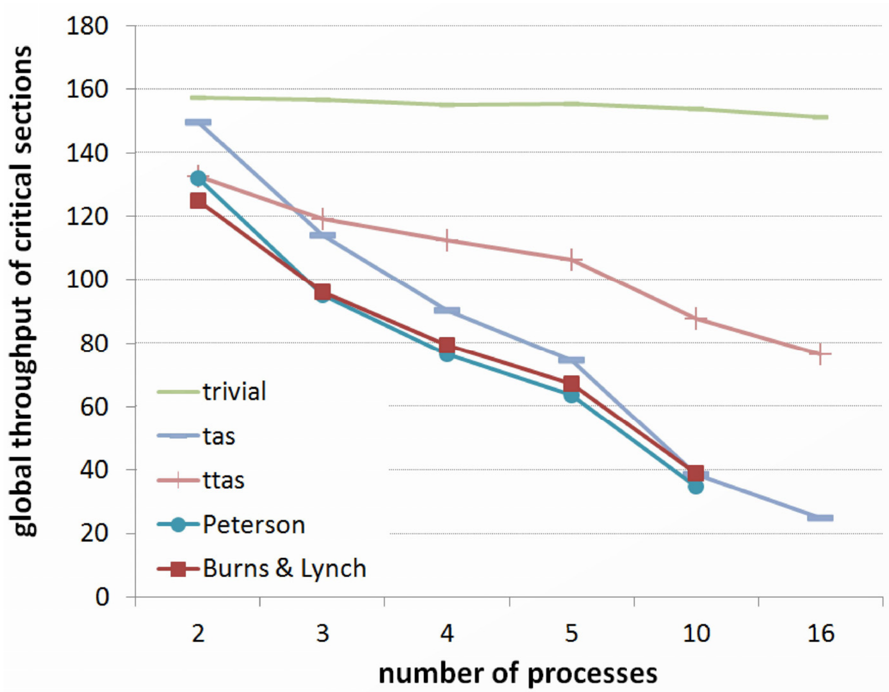




Global throughput
(increasing number of processes)



Scalable procotols



Unscalable procotols

Conclusion and future work

- Formal analysis and performance evaluation of mutual exclusion protocols on a single model
- Automated analysis using CADP (LNT, MCL, SVL)
- (More?) proper handling of nondeterminism

- Extend performance study to
  - Determine variable placement

    ➔ *frequent accesses should be local, not remote*
  - Analyze performance w.r.t. degree of contention (e.g., Lamport's fast mutex protocol)