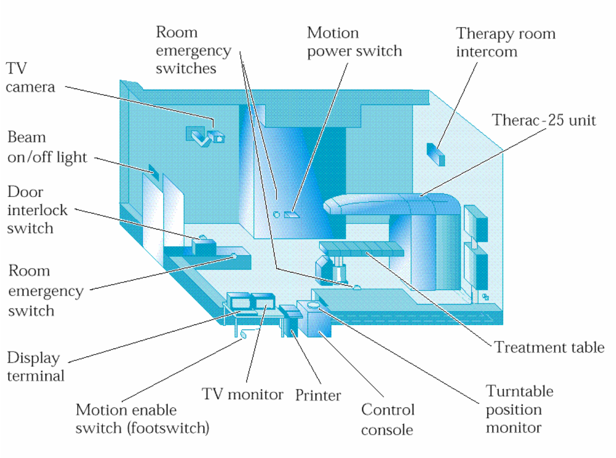
Model Checking of Action-Based Concurrent Systems

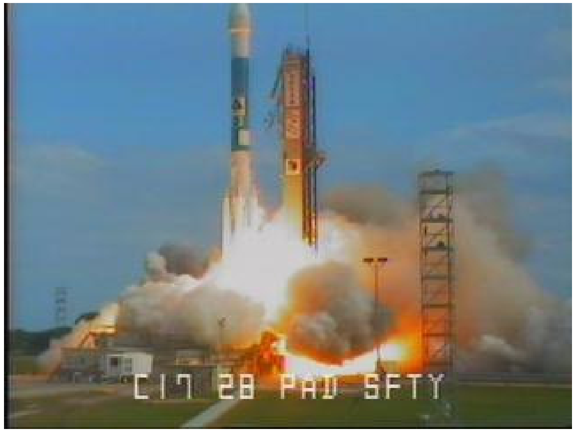
Radu Mateescu
INRIA Rhône-Alpes / VASY
http://www.inrialpes.fr/vasy

Why formal verification?



Therac-25 radiotherapy accidents (1985-1987)

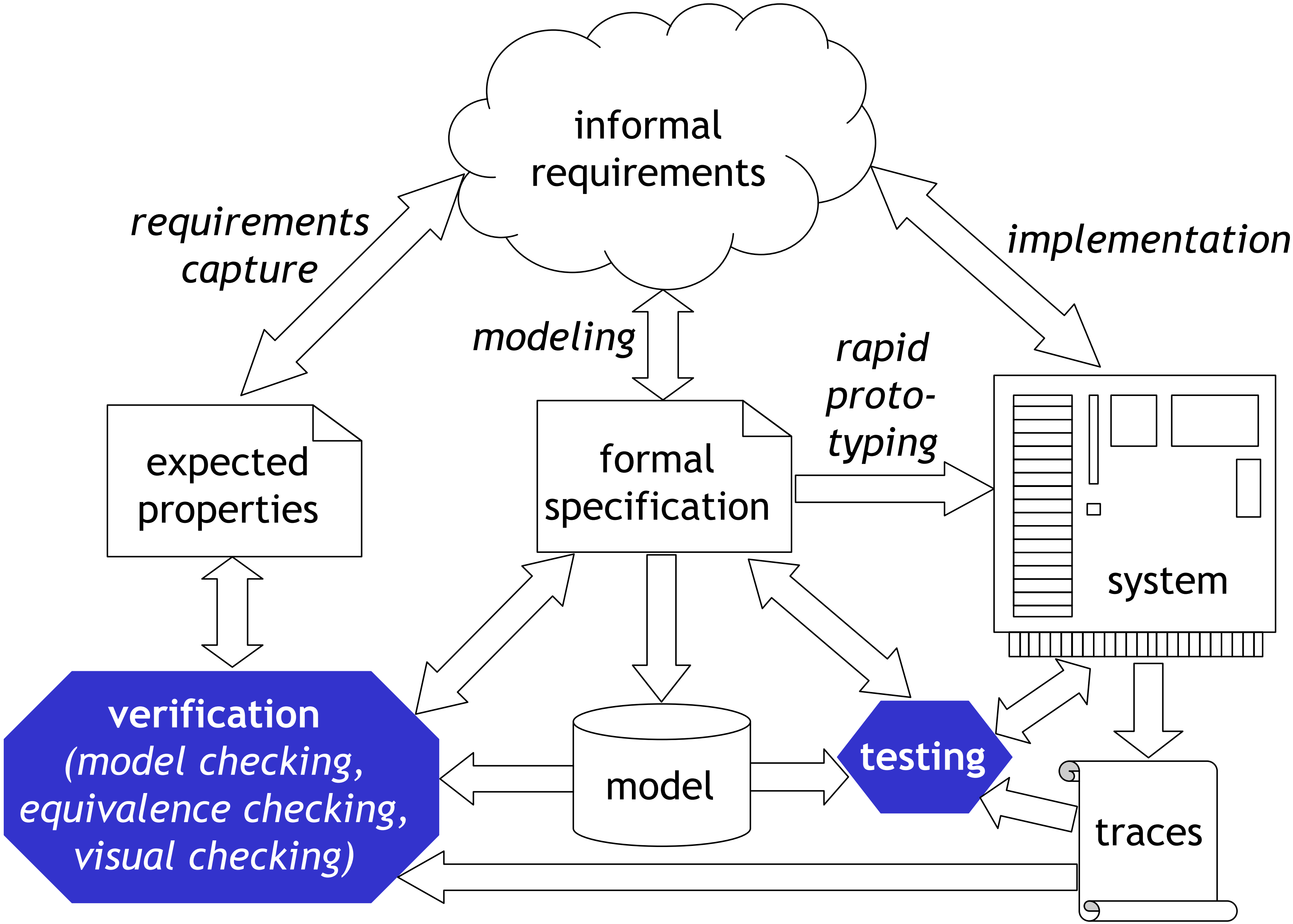
Ariane-5 launch failure (1996)



Mars climate orbiter failure (1999)

- Characteristics of these systems
  - Errors due to software
  - Complex, often involving parallelism
  - Safety-critical

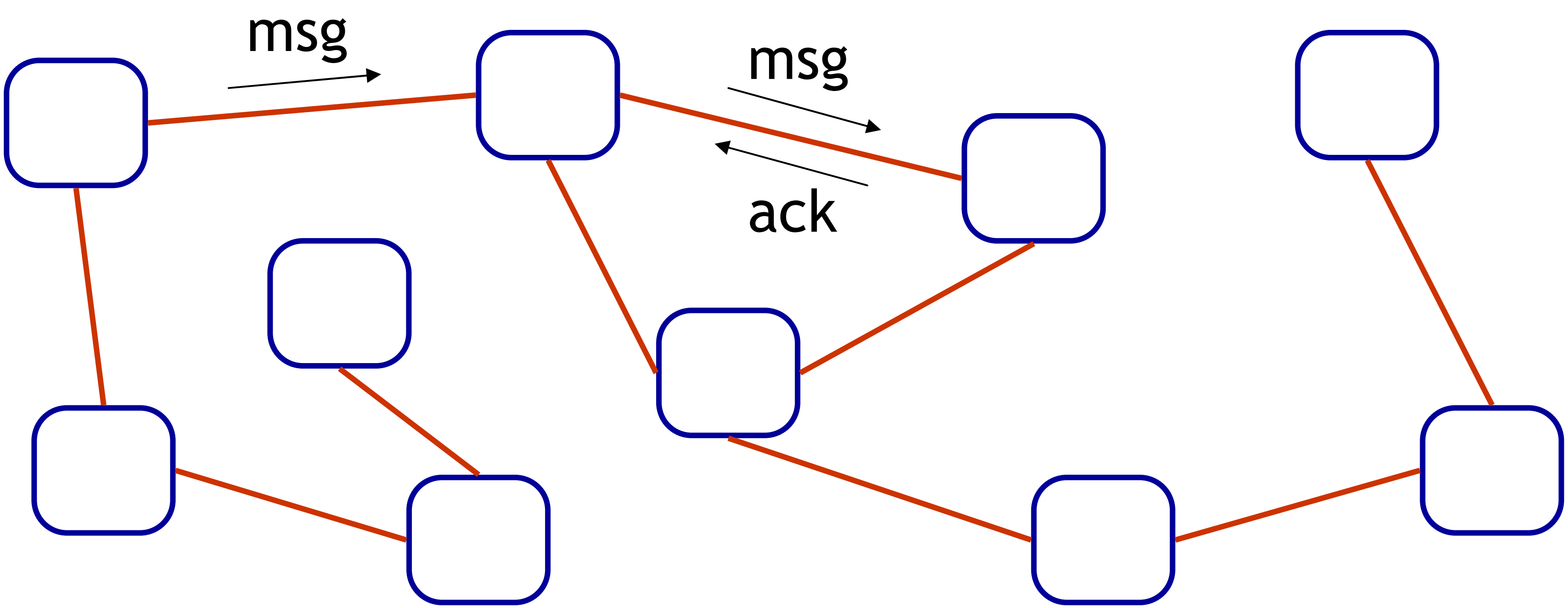
➔ *formal verification is useful for early error detection*


informal
requirements
requirements
capture
implementation
modeling
rapid
proto-
typing
expected
properties
formal
specification
system
verification
(model checking,
equivalence checking,
visual checking)
model
testing
traces

Outline

- Communicating automata
- Process algebraic languages
- Action-based temporal logics
- On-the-fly verification
- Case study
- Discussion and perspectives

Asynchronous concurrent systems



Characteristics:

- Set of distributed processes
- Message-passing communication
- Nondeterminism

Applications:

- Hardware
- Software
- Telecommunications

CADP toolbox:
Construction and Analysis of Distributed Processes

(http://www.inrialpes.fr/vasy/cadp)

- Description languages:
  - ISO standards (LOTOS, E-LOTOS)
  - Networks of communicating automata
- Functionalities:
  - Compilation and rapid prototyping
  - Interactive and guided simulation
  - Equivalence checking and model checking
  - Test generation
- Case-studies and applications:
  - >100 industrial case-studies
  - >30 derived tools
- Distribution: over 400 sites (2008)

Communicating automata

- Basic notions
- Implicit and explicit representations
- Parallel composition and synchronization
- Hiding and renaming
- Behavioural equivalences

Transformational systems

- Work by computing a result in function of the entries
- Absence of termination undesirable
- Upon termination, the result is unique
- Sequential programming (sorting algorithms, graph traversals, syntax analysis, ...)

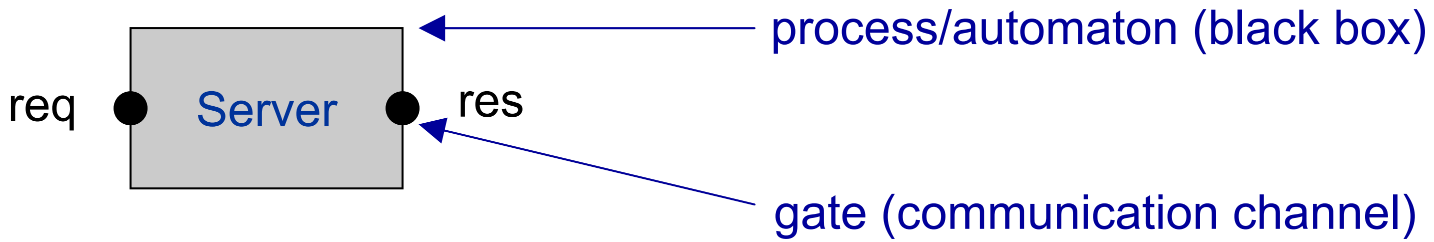
Reactive systems

- Work by reacting to the stimuli of the environment
- Absence of termination desirable
- Different occurrences of the same request may produce different results
- Parallel programming (operating systems, communication protocols, Web services, ...)

- Concurrent execution
- Communication + synchronization

Communicating automata

- Simple formalism describing the behaviour of concurrent systems
- *Black-box* approach:
  - One cannot inspect directly the state of the system
  - The behaviour of the system can be known only through its interactions with the environment



- Synchronization on a gate requires the participation of the process and of its environment (*rendezvous*)

Automaton (LTS)

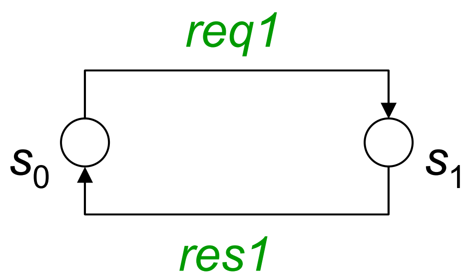
- Labeled Transition System $M = \langle S, A, T, s_0 \rangle$
  - S: set of *states* $(s_1, s_2, \ldots)$
  - A: set of visible *actions* $(a_1, a_2, \ldots)$
  - T: *transition* relation $(s_1 \text{-}a\rightarrow s_2 \in T)$
  - $s_0 \in S$: *initial state*

internal action (noted i or τ)

every state is reachable from the initial state

deadlock (sink) state: no outgoing transitions

- Example: process client$_1$



sequential model of a reactive system behaviour

- Other kinds of automata:
  - Kripke strictures (information associated to states)
  - Input/output automata [Lynch-Tuttle]

LTS representations in CADP
(http://www.inrialpes.fr/vasy/cadp)

Explicit

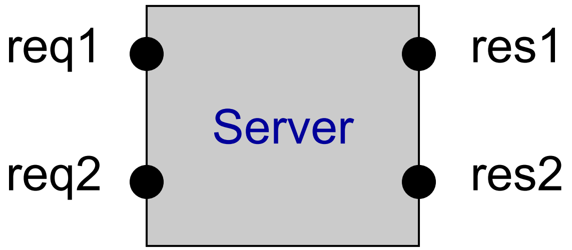
- List of transitions
- Allows forward and backward exploration
- Suitable for global verification
- BCG (Binary Coded Graphs) environment
  - API in C for reading/writing
  - Tools and libraries for explicit graph manipulation (**bcg**_io, **bcg**_draw, **bcg**_info, **bcg**_edit, **bcg**_labels, ...)
  - Global verification tools (XTL)

Implicit

- "Successor" function
- Allows forward exploration only
- Suitable for local (or on-the-fly) verification
- Open/Caesar environment [Garavel-98]
  - API in C for LTS exploration
  - Libraries with data structures for implicit graph manipulation (stacks, tables, edge lists, hash functions, ...)
  - On-the-fly verification tools (Bisimula**tor**, Evalua**tor**, ...)

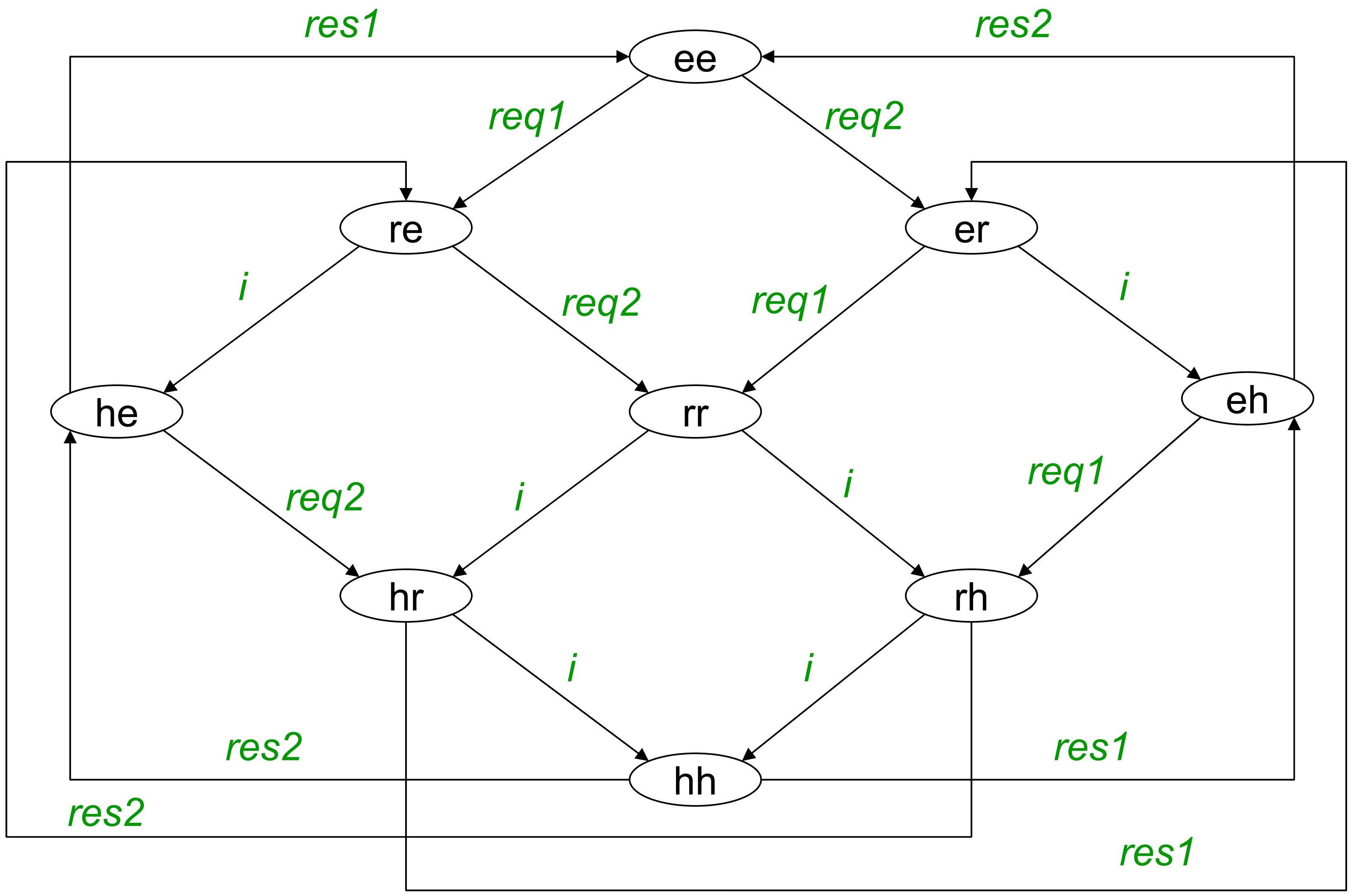
Server example
(modeled using a single automaton)

- Server able to process two requests concurrently
- State variables u_1, u_2 storing the request status:
  - Empty (e)
  - Received (r)
  - Handled (h)
- A state: couple $<u_1, u_2>$
- Initial state: <e, e> (ee for short)
- Gates (actions):
  - req1, req2: receive a request
  - res1, res2: send a response
  - i: internal action



LTS of the server

(9 states, 16 transitions)



Remarks

- All the theoretical states are reachable:

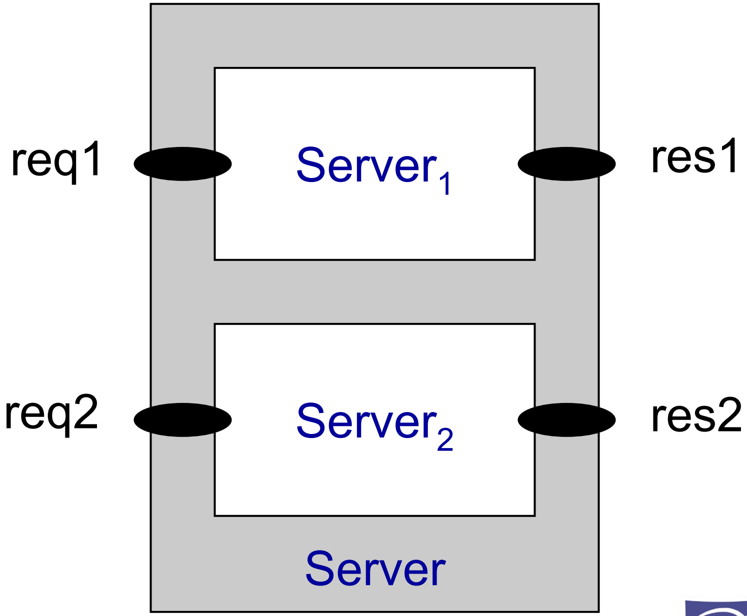
  $$| u_1 | * | u_2 | = 3 * 3 = 9$$

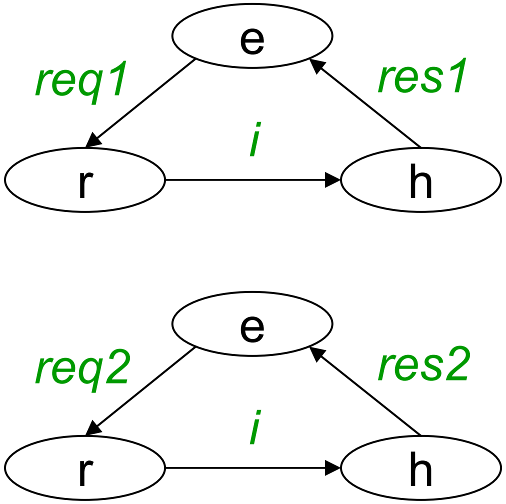
  (no synchronization between request processings)
- There is no sink state (the system is *deadlock-free*)
- From every state, it is possible to reach the initial state again (the server can be re-initialized)
- Shortcomings of modeling with a single automaton:
  - One must predict all the possible request arrival orders
  - For more complex systems, the LTS size grows rapidly

➔ *need of higher-level modeling features*

Server example
(modeled using two concurrent automata)

- Decomposition of the system in two subsystems
  - Every type of request is handled by a subsystem
  - In the server example, subsystems are independent
- Simpler description w.r.t. single automaton:
  3 + 3 = 6 states





Decomposition in concurrent subsystems

Required at physical level

- Modeling of distributed activities
- Multiprocessor/multitasking execution platform

Chosen at logical level

- Simplified design of the system
- Well-structured programs

- Communication and synchronization between subsystems may introduce behavioural errors (e.g., *deadlocks*)
- In practice, even simple parallel programs may reveal difficult to analyze

→ *need of computer-assisted verification*

Parallel composition (“product”) of automata

- Goals:
  - Define internal composition laws

    $$\otimes : \mathrm{LTS} \times \ldots \times \mathrm{LTS} \rightarrow \mathrm{LTS}$$

    expressing the parallel composition of 2 (or more) LTSs
  - Allow synchronizations on one or several actions (gates)
  - Allow hierarchical decomposition of a system
- Consequences:
  - A product of automata can always be translated into a single (sequential) automaton
  - The logical parallelism can be implemented sequentially (e.g., time-sharing OS)

Binary parallel composition
(syntax)

- EXP language [Lang-05]
  - Description of communicating automata
  - Extensive set of operators
    - Parallel compositions (binary, general, ...)
    - Synchronization vectors
    - Hiding / renaming, cutting, priority, ...
  - Exp.Open compiler ➔ implicit LTS representation
- Binary parallel composition:

"lts1.bcg"	|[G1, ..., Gn]|	"lts2.bcg"	with synchronization on G1, ..., Gn
"lts1.bcg"	|||	"lts2.bcg"	without synchronization (interleaving)

Binary parallel composition
(semantics)

Let $M_1 = \langle S_1, A_1, T_1, s_{01} \rangle$, $M_2 = \langle S_2, A_2, T_2, s_{02} \rangle$ and
$L \subseteq A_1 \cap A_2$ a set of visible actions to be synchronized.

$M_1 \; |[\, L \,]| \; M_2 = \langle S, A, T, s_0 \rangle$

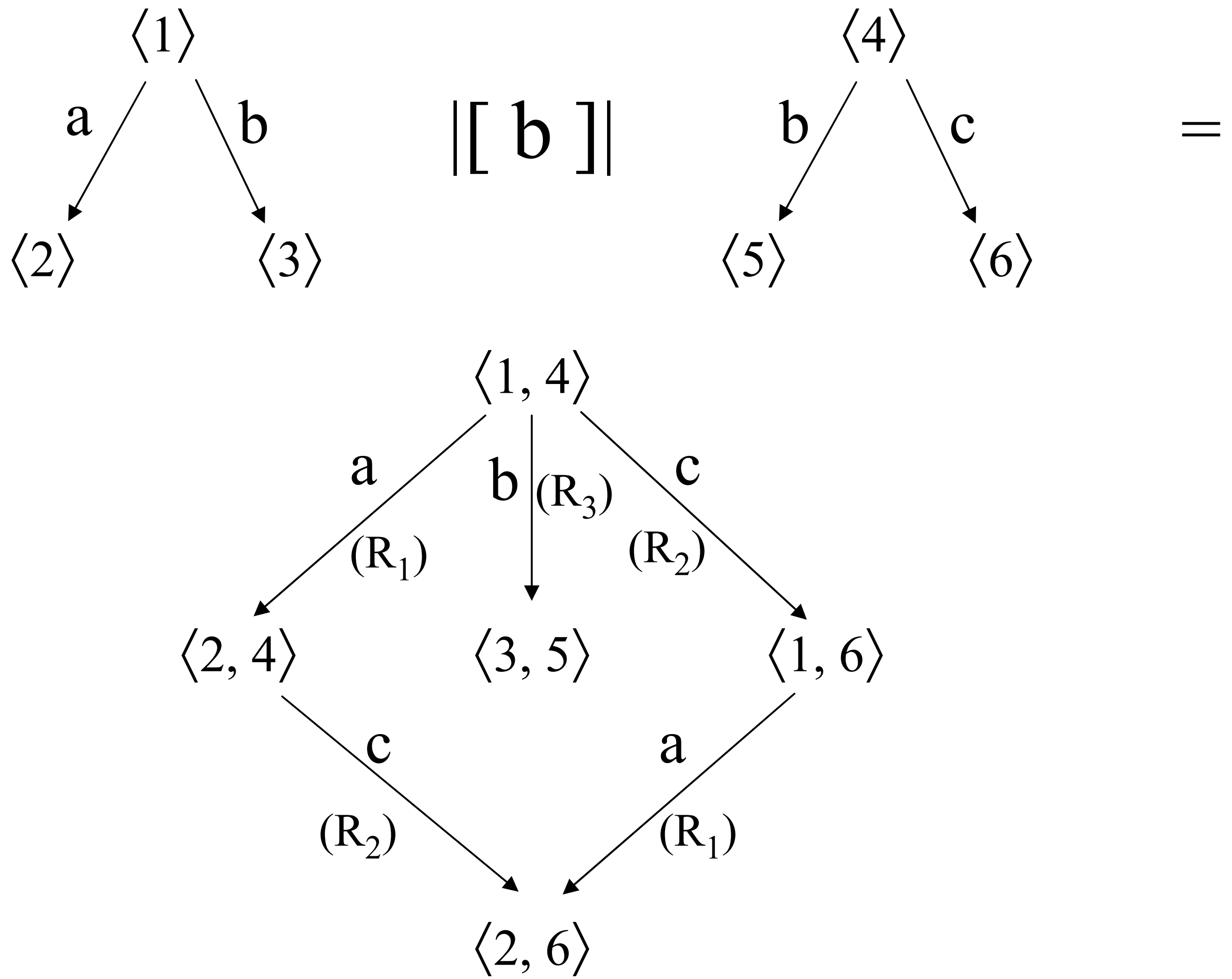
- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $s_0 = \langle s_{01}, s_{02} \rangle$
- $T \subseteq S \times A \times S$
  defined by R_1-R_3

$$(R_1) \quad \frac{s_1 \xrightarrow{a} s'_1 \;\wedge\; a \notin L}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s'_1, s_2 \rangle}$$

$$(R_2) \quad \frac{s_2 \xrightarrow{a} s'_2 \;\wedge\; a \notin L}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1, s'_2 \rangle}$$

$$(R_3) \quad \frac{s_1 \xrightarrow{a} s'_1 \wedge s_2 \xrightarrow{a} s'_2 \wedge\; a \in L}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s'_1, s'_2 \rangle}$$

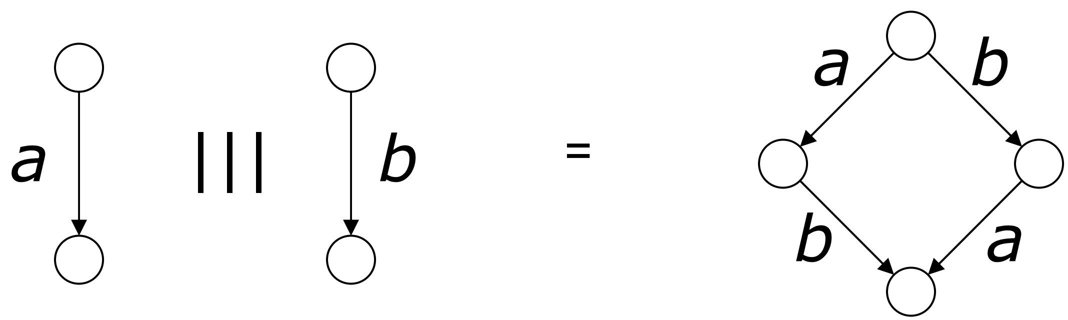
Example



Interleaving semantics

- Hypothesis:
  - Every action is atomic
  - One can observe at most one action at a time

➔ *suitable paradigm for distributed systems*

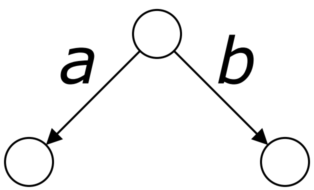


interleaving lozenge

- Parallelism can be expressed in terms of *choice* and *sequence* (*expansion theorem* [Milner-89])

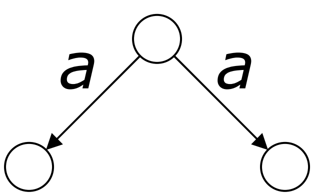
Internal and external choice

- *External* choice (the environment decides which branch of the choice will be executed)



the environment can force the execution of a and b by synchronizing on that action

- *Internal* choice (the system decides)



the environment may synchronize on a, but this will not remove the nondeterminism

Example of modeling with communicating automata

- Mutual exclusion problem:

  Given two parallel processes P_0 and P_1 competing for a shared resource, guarantee that at most one process accesses the resource at a given time.

- Several solutions were proposed *at software level*:
  - In centralized setting (Peterson, Dekker, Knuth, ...)
  - In distributed setting (Lamport, ...)

➔ M. Raynal. *Algorithmique du parallélisme: le problème de l'exclusion mutuelle.* Dunod Informatique, 1984.

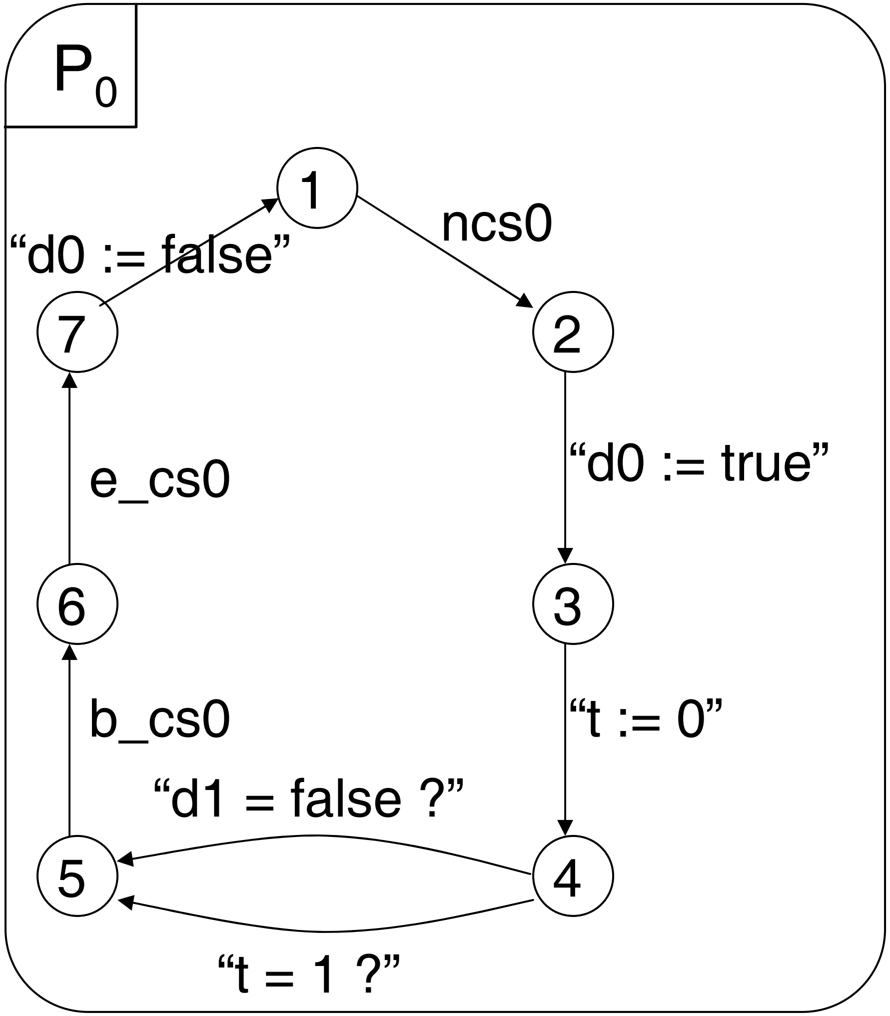
Peterson's algorithm [1968]

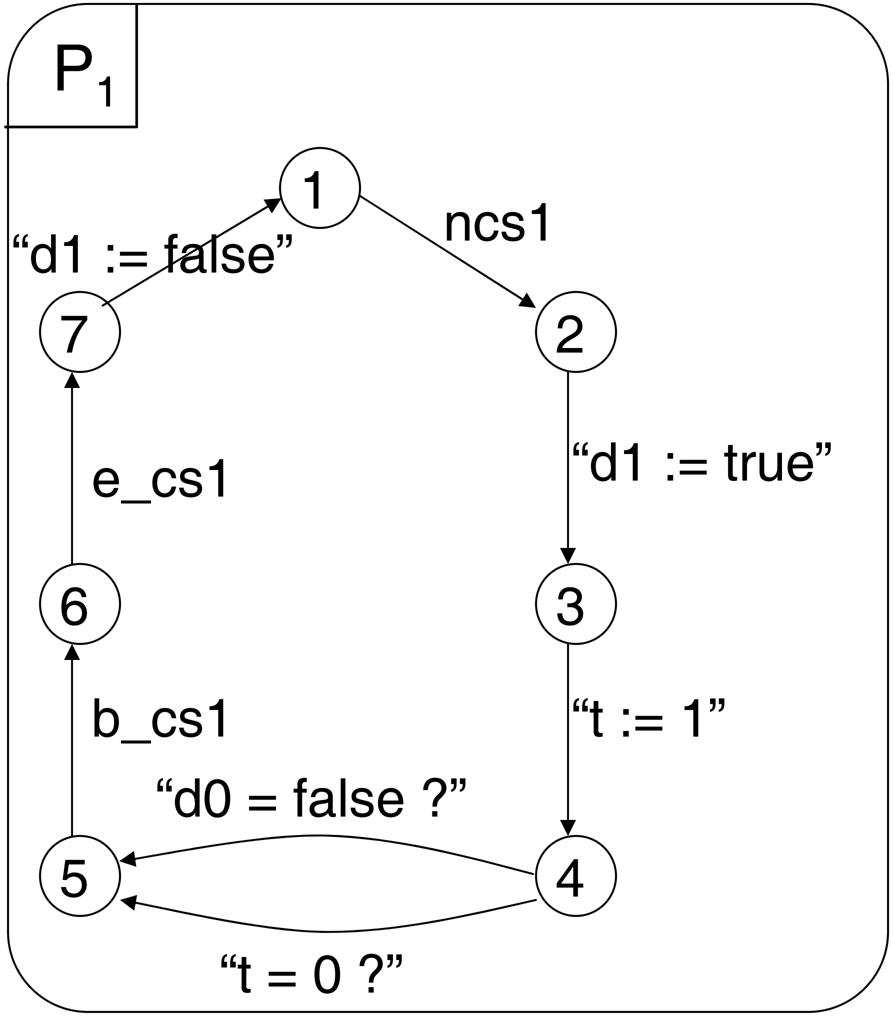
var d0 : bool := false { read by P1, written by P0 }
var d1 : bool := false { read by P0, written by P1 }
var t $\in$ {0, 1} := 0 { read/written by P0 and P1 }

```
loop forever { P0 }
1 : { ncs0 }
2 : d0 := true
3 : t := 0
4 : wait (d1 = false or t = 1)
5 : { b_cs0 }
6 : { e_cs0 }
7 : d0 := false
endloop
```

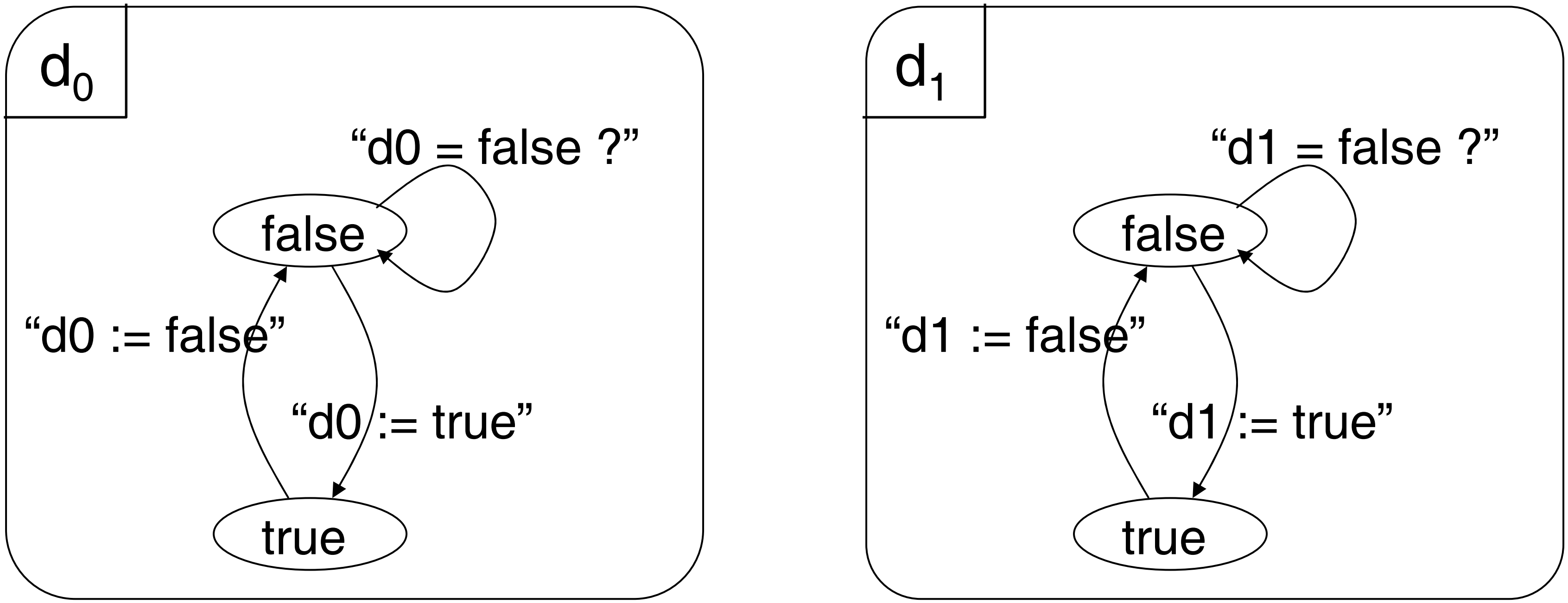
```
loop forever { P1 }
1 : { ncs1 }
2 : d1 := true
3 : t := 1
4 : wait (d0 = false or t = 0)
5 : { b_cs1 }
6 : { e_cs1 }
7 : d1 := false
endloop
```

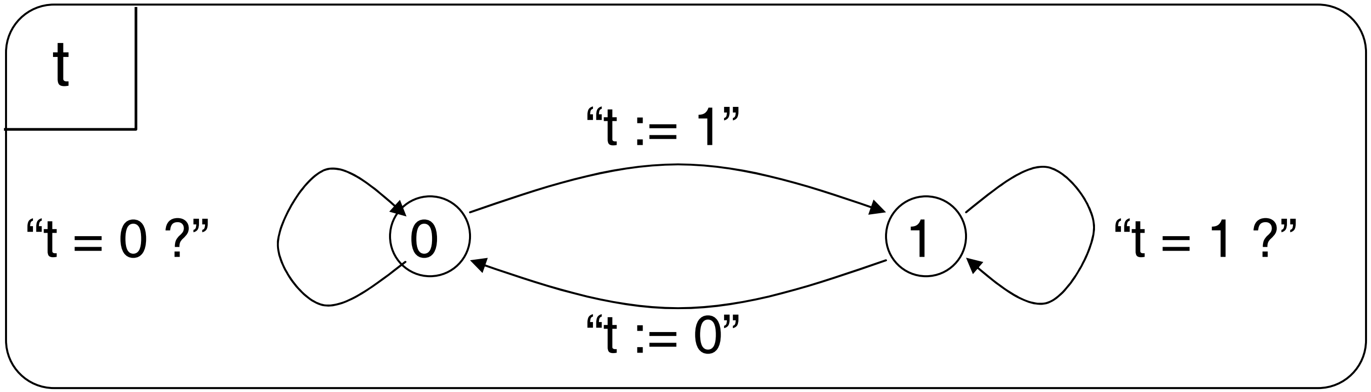
Automata of P_0 and P_1



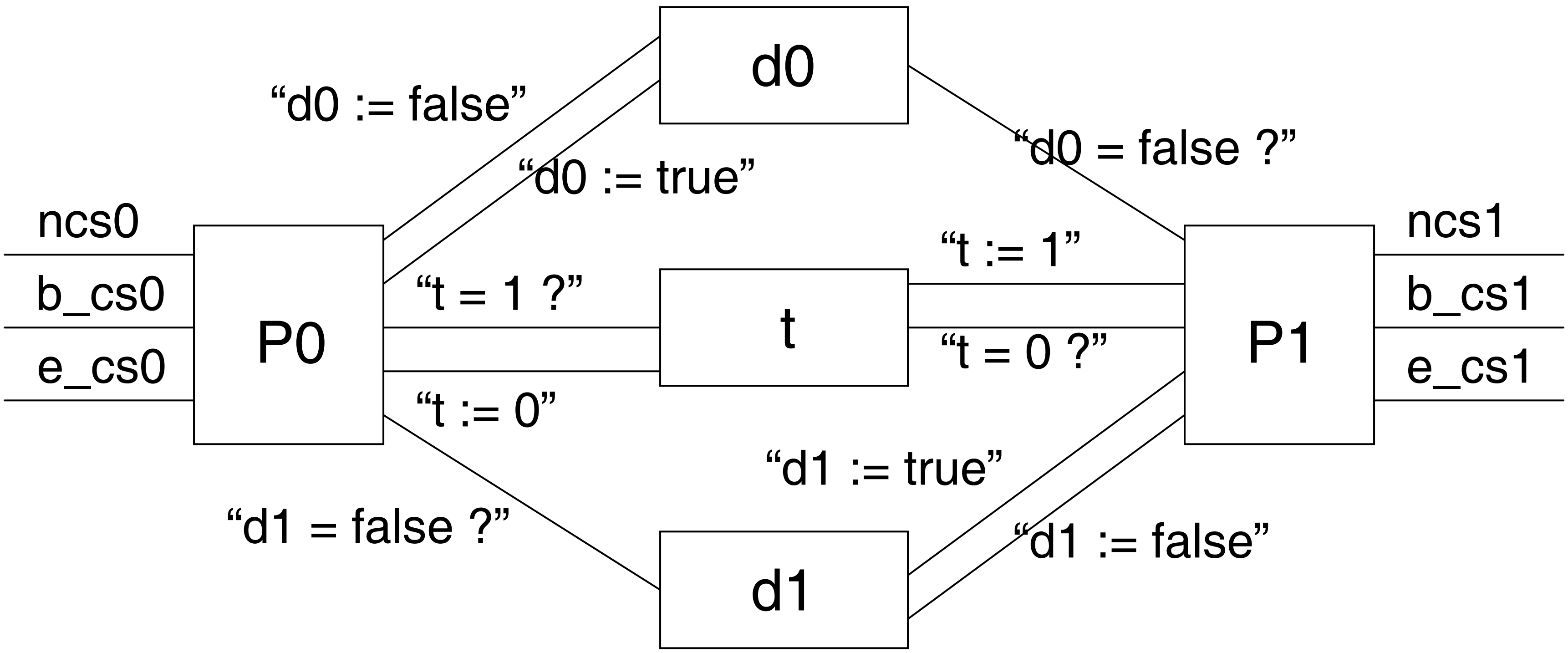


Automata of d_0, d_1, and t





Architecture of the system
(graphical)



- Synchronized actions: «d0:=false», «d0:=true», ...
- Non synchronized actions: ncs0, b_cs0, e_cs0, ...

Architecture of the system
(textual)

- Using binary parallel composition:

```
(P0 ||| P1)
|[ "d0:=false", "d0:=true", ... ]|
(d0 ||| d1 ||| t)
```

- Using general parallel composition:

```
par
       "d0:=false", "d0:=true", ... → P0
||     "d1:=false", "d1:=true", ... → P1
||     "d0:=false", "d0:=true", "d0=false?" → d0
||     "d1:=false", "d1:=true", "d1=false?" → d1
||     "t:=0", "t:=1", "t=0?", "t=1?" → t
end par
```

Construction of the LTS
("product automaton")

- *Explicit-state* method:
  - LTS construction by exploring forward the transition relation, starting at the initial state
  - Transitions are generated by using the R_1, R_2, R_3 rules
  - Detect already visited states in order to avoid cycling
- Several possible exploration strategies:
  - Breadth-first, depth-first
  - Guided by a criterion / property, ...
- Several types of algorithms:
  - Sequential, parallel, distributed, ...

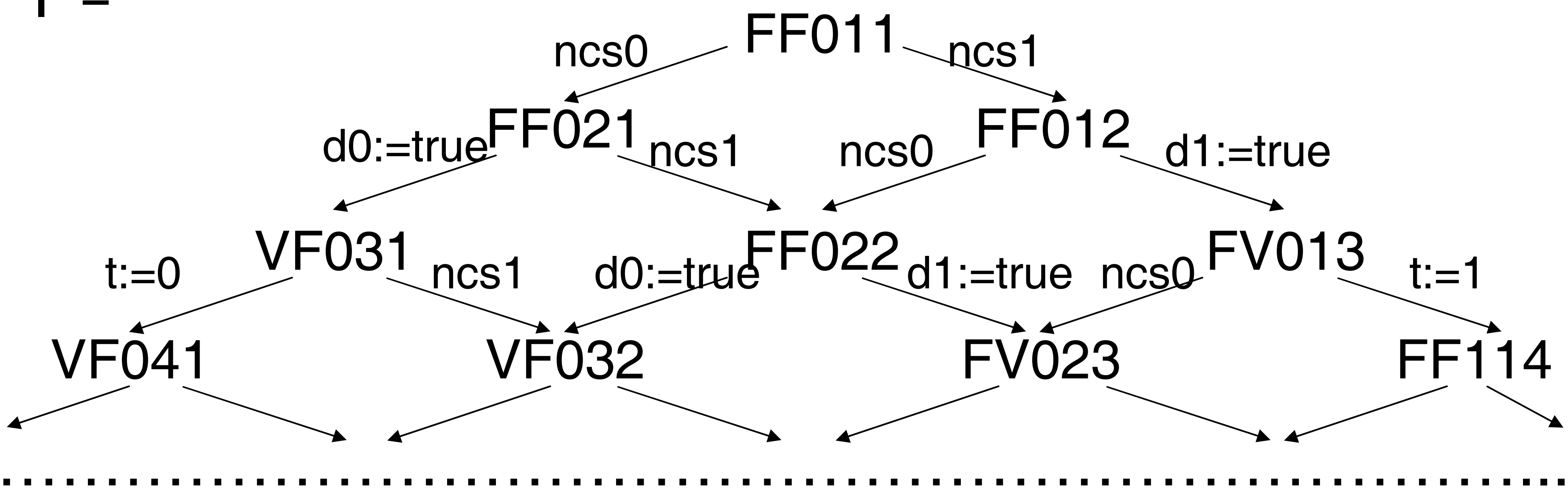
Construction of the LTS

$S = \{ F,V \} \times \{ F,V \} \times \{ 0,1 \} \times \{ 1..7 \} \times \{ 1..7 \}$

A = { ncs0, ncs1, …, “d0:=true”, … }

$s_0 = \langle F, F, 0, 1, 1 \rangle$ = FF011

T =



Remarks

- The LTS of Peterson's algorithm is finite:

  $$| S | \cong 50 \leq 2 \times 2 \times 2 \times 7 \times 7 = 392$$

- In the presence of synchronizations, the number of reachable states is (much) smaller than the size of the cartesian product of the variable domains
- Some tools of CADP for LTS manipulation:
  - OCIS (step-by-step and guided simulation)
  - Executor (random exploration)
  - Exhibitor (search for regular sequences)
  - Terminator (search for deadlocks)

  → can be used in conjunction with Exp.Open

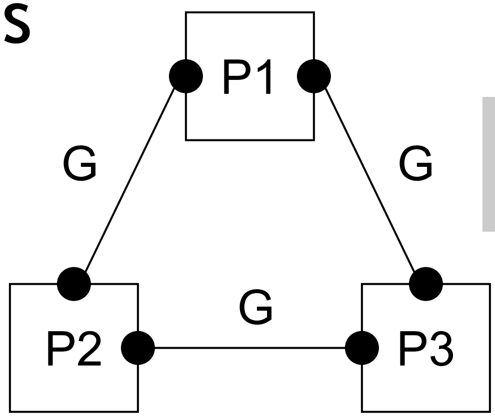
Verification

- Once the LTS is generated, one can formulate and verify automatically the desired properties of the system
- For Peterson's algorithm:
  - Deadlock freedom: each state has at least one successor
  - Mutual exclusion: at most one process can be in the critical section at a given time
  - Liveness: no process can indefinitely overtake the other when accessing its critical section

[see the chapter on temporal logics]

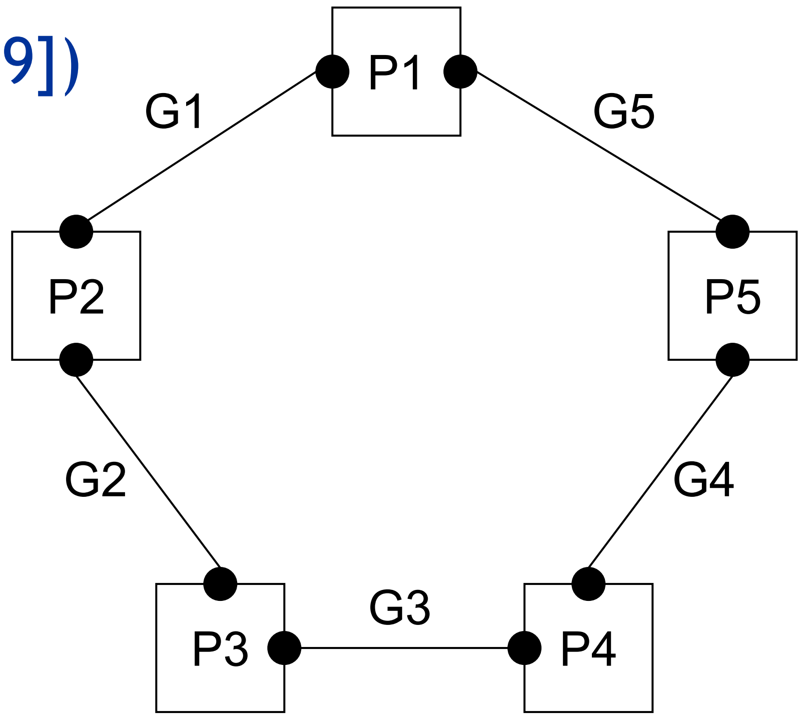
Limitations of binary parallel composition

- Several ways of modeling a process network:
  - Absence of *canonical form*
  - Difficult to determine whether two composition expressions denote the same process network
  - Difficult to retrieve the process network from a composition expression
- The semantics of "|[G_1, ..., G_n]|" (rule R_3) does not prevent that other processes synchronize on G_1, ..., G_n (*maximal cooperation*)
- Some networks cannot be modeled using "|[]|":



binary synchronization on G

Example
(ring network [Garavel-Sighireanu-99])



- Description using binary parallel composition:

$$(P_1 \ |[G_1]| \ P_2 \ |[G_2]| \ P_3 \ |[G_3]| \ P_4)$$
$$|[G_4, G_5]|$$
$$P_5$$

the composition expression does not reflect the symmetry of the process network

General parallel composition

[Garavel-Sighireanu-99]

- "Graphical" parallel composition operator allowing the composition of several automata and their *m* among *n* synchronization:

$$
\begin{array}{ll}
\textbf{par} \; [\, g_1\#m_1, \ldots, g_p\#m_p \; \textbf{in} \,] & \\
\quad \underline{G}_1 \rightarrow B_1 & \\
|| \quad \underline{G}_2 \rightarrow B_2 & \\
\ldots & \\
|| \quad \underline{G}_n \rightarrow B_n & \\
\textbf{end par} &
\end{array}
$$

gates with their associated synchronization degrees

automata (processes)

communication interfaces (gate lists)

General parallel composition

(semantics – rules without synchronization degrees)

$$\frac{\exists\, a, i \,.\, B_i \xrightarrow{a} B_i' \wedge a \notin G_i \wedge \forall\, j \neq i \,.\, B_j' = B_j}{\textbf{par}\ G_1 \rightarrow B_1, \ldots, G_n \rightarrow B_n \xrightarrow{a} \textbf{par}\ G_1 \rightarrow B_1', \ldots, G_n \rightarrow B_n'} \quad \text{(GR1)}$$

mandatory interleaved execution of non-synchronized actions

$$\frac{\exists\, a. \forall\, i \,.\, \text{if } a \in G_i \text{ then } B_i \xrightarrow{a} B_i' \text{ else } B_j' = B_j}{\textbf{par}\ G_1 \rightarrow B_1, \ldots, G_n \rightarrow B_n \xrightarrow{a} \textbf{par}\ G_1 \rightarrow B_1', \ldots, G_n \rightarrow B_n'} \quad \text{(GR2)}$$

execution in maximal cooperation of synchronized actions

Example (1/3)

- Process network unexpressible using "|[]|":

- Description using general parallel composition:

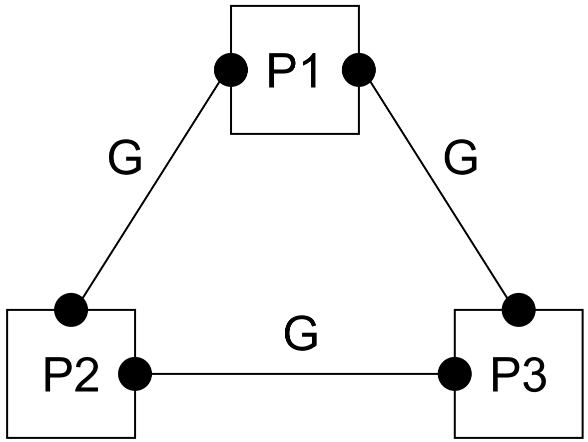
  par G#2 **in**

  $G \rightarrow P_1$

  || $G \rightarrow P_2$

  || $G \rightarrow P_3$

  end par



maximal cooperation avoided by means of synchronization degrees

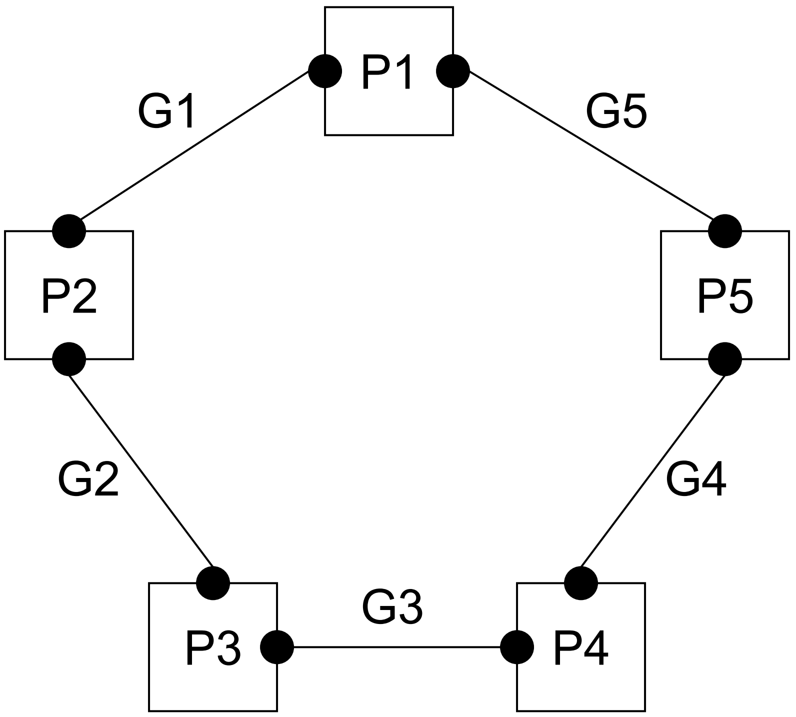
Example (2/3)
(ring network [Garavel-Sighireanu-99])

- Description using general parallel composition:

par

$$G_1, G_5 \rightarrow P_1$$
$$|| \quad G_2, G_1 \rightarrow P_2$$
$$|| \quad G_3, G_2 \rightarrow P_3$$
$$|| \quad G_4, G_3 \rightarrow P_4$$
$$|| \quad G_5, G_4 \rightarrow P_5$$

end par



the symmetry of the process network is also present in the composition expression

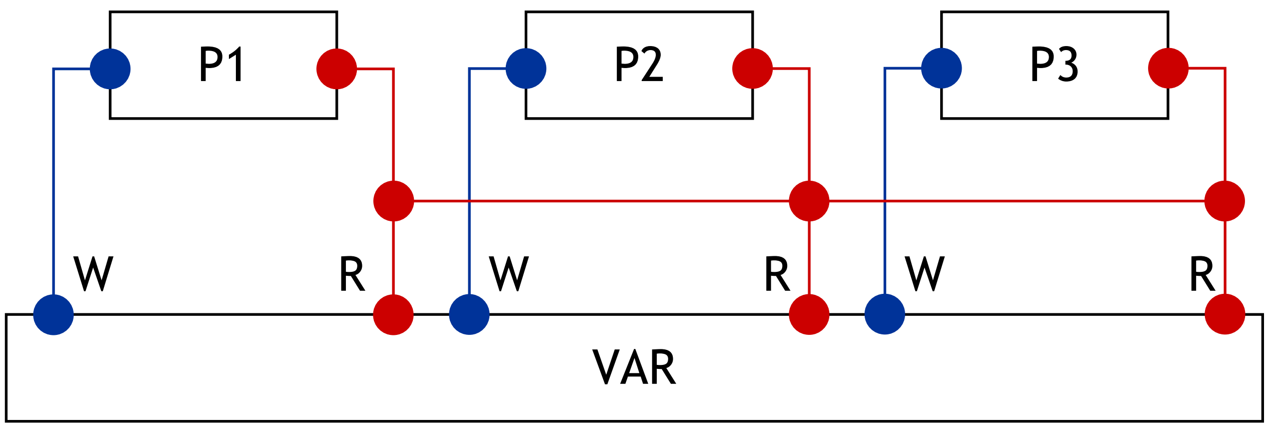
Example (3/3)

- Definition of "|[]|" in terms of "**par**":

B_1 |[G_1, …, G_n]| B_2 = **par** G_1, …, G_n → B_1
|| G_1, …, G_n → B_2
end par

- CREW (Concurrent Read / Exclusive Write):

par W#2 **in**
R, W → P_1
|| R, W → P_2
|| R, W → P_3
|| R, W → VAR
end par



Parallel composition using synchronization vectors

- Primitive form of n-ary parallel composition
- Proposed in various networks of automata:
  MEC [Arnold-Nivat], FC2 [deSimone-Bouali-Madelaine]
- Synchronizations are made explicit by means of *synchronization vectors*
- Syntax in the EXP language [Lang-05]:

par $V_1, \ldots, V_m$ **in**

$B_1 \mid\mid \ldots \mid\mid B_n$

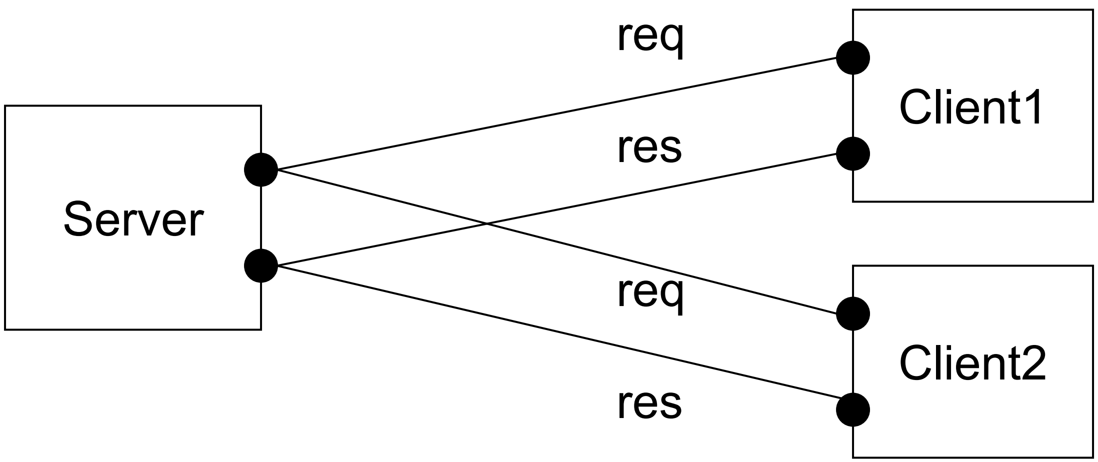
end par

synchronization vectors

$V ::= (G_1 \mid \_) * \ldots * (G_n \mid \_) \rightarrow G_0$

wildcard

Example
(client-server with gate multiplexing)



binary synchronization on gates req *and* res

- Description using synchronization vectors:

par req * _ * req → req, rep * _ * rep → rep,
_ * req * req → req, _ * rep * rep → rep

in

Client$_1$ || Client$_2$ || Server

end par

Behavioural equivalence

- Useful for determining whether two LTSs denote the same behaviour
- Allows to:
  - Understand the semantics of languages (communicating automata, process algebras) having LTS models
  - Define and assess translations between languages
  - Refine specifications whilst preserving the equivalence of their corresponding LTSs
  - Replace certain system components by other, equivalent ones (maintenance)
  - Exploit identities between behaviour expressions (e.g., B_1 |[G]| B_2 = B_2 |[G]| B_1) in analysis tools

Equivalence relations between LTSs

- A large spectrum of equivalence relations proposed:
  - *Trace* equivalence (≅ language equivalence)
  - *Strong* bisimulation [Park-81]
  - *Weak* bisimulation [Milner-89]
  - *Branching* bisimulation [Bergstra-Klop-84]
  - Safety equivalence [Bouajjani-et-al-90]
  - ...

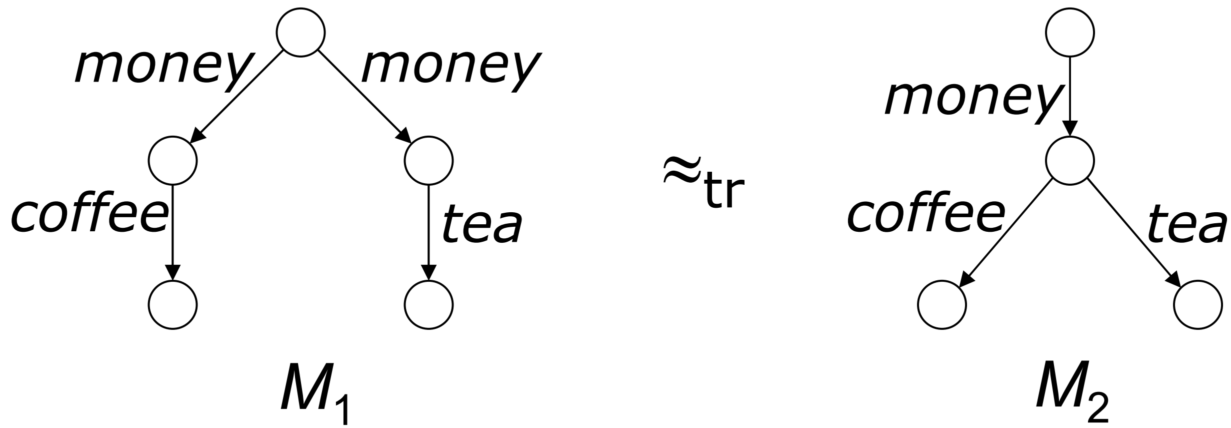
Trace equivalence

- Trace: sequence of visible actions (e.g., $\sigma = req_1\ res_1\ req_2\ res_2$)
- Notations (a = visible action):
  - s =a=>: there exists a transition sequence $s \ -i\rightarrow s_1 \ -i\rightarrow s_2 \ \ldots \ -a\rightarrow s_k$
  - s =σ=>: there exists a transition sequence $s = a_1 \Rightarrow s_1 \ldots = a_n \Rightarrow s_n$ such that $\sigma = a_1 \ldots a_n$
- Two state are trace equivalents iff they are the source of the same traces:

$$s \approx_{tr} s' \quad \text{iff} \quad \forall\sigma\,.\ (s = \sigma \Rightarrow \quad \text{iff} \quad s = \sigma \Rightarrow)$$

Example
(coffee machine)

- The two LTSs below are trace equivalent:



Traces (M_1) = Traces (M_2) =
{ ε, *money*, *money coffee*, *money tea* }

➔ *have the two coffee machines the same behaviour w.r.t. a user?*

M_1: risk of deadlock

Bisimulation

- Trace equivalence is not sufficiently precise to characterize the behaviour of a system w.r.t. its interaction with its environment

  ➔ *stronger relations (bisimulations) are necessary*

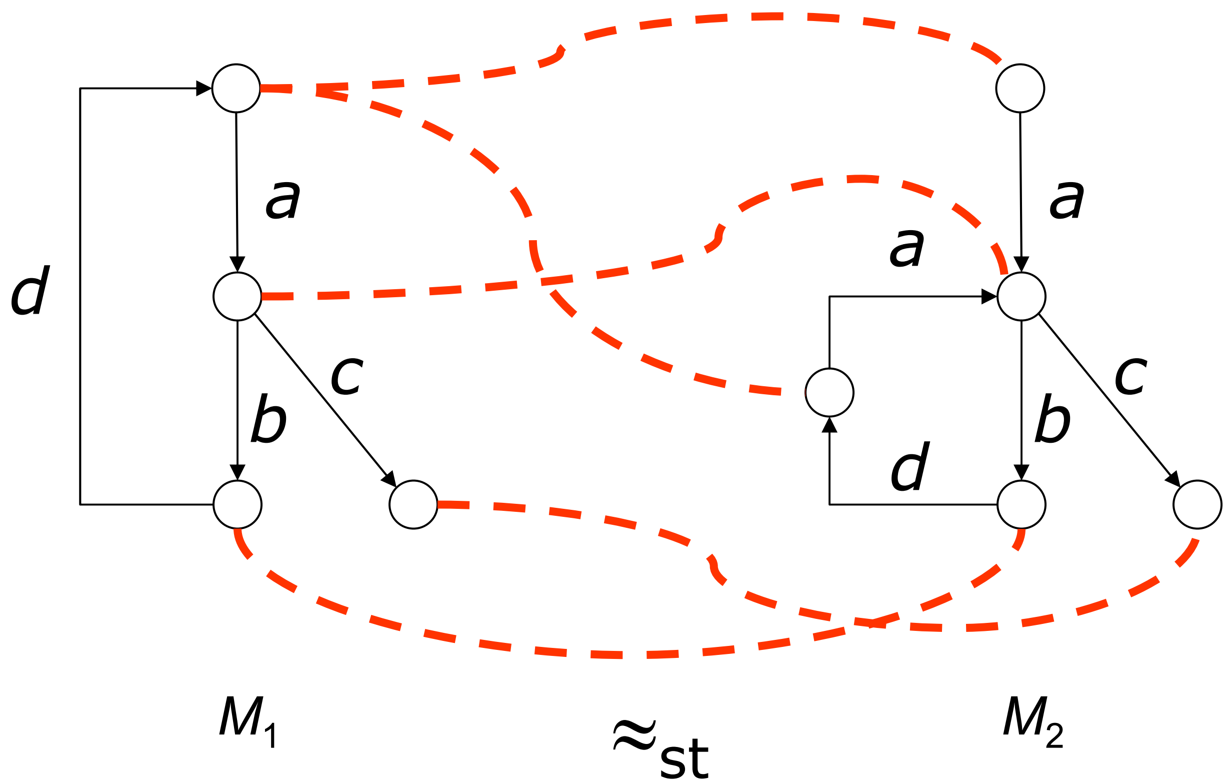
- Two states s_1 et s_2 are *bisimilar* iff they are the origin of the same behaviour (execution tree):

$$\forall\ s_1\text{-}a\rightarrow s_1'\ .\ \exists\ s_2\text{-}a\rightarrow s_2'\ .\ s_1' \approx s_2'$$

$$\forall\ s_2\text{-}a\rightarrow s_2'\ .\ \exists\ s_1\text{-}a\rightarrow s_1'\ .\ s_2' \approx s_1'$$

- Bisimulation is an equivalence relation (reflexive, symmetric, and transitive) on states
- Two LTSs are bisimilar iff $s_{01} \approx s_{02}$

Strong bisimulation



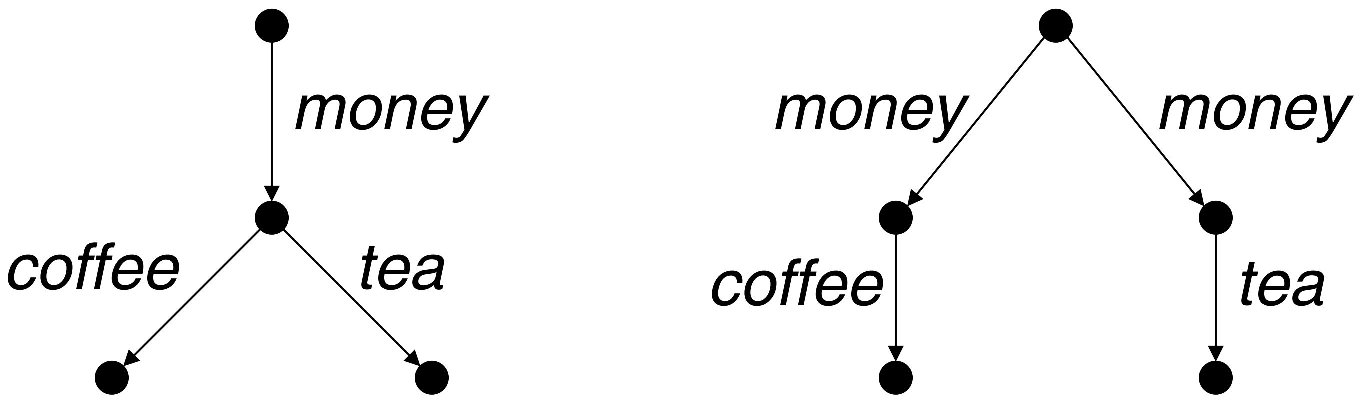
- Strong bisimulation: the largest bisimulation

  ➔ *to show that two LTSs are strongly bisimilar, it is sufficient to find a bisimulation between them*

Is strong bisimulation sufficient?

- *Trace equivalence* ignores internal actions (*i*) and does not capture the branching of transitions

  ➔ *does not distinguish the LTSs below*

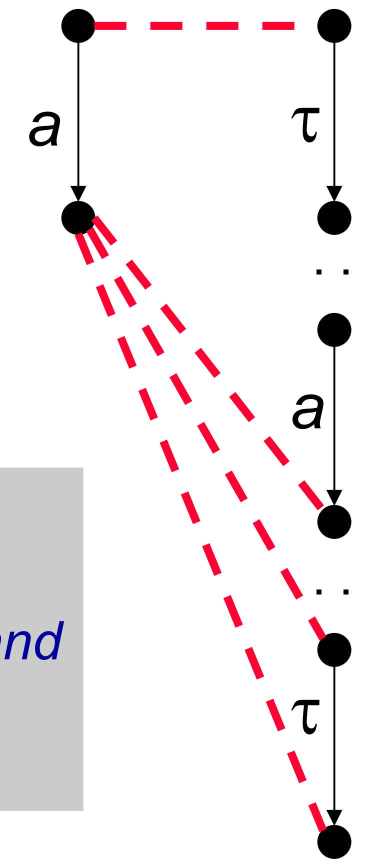


- *Strong bisimulation* captures the branching, but handles internal and visible actions in the same way

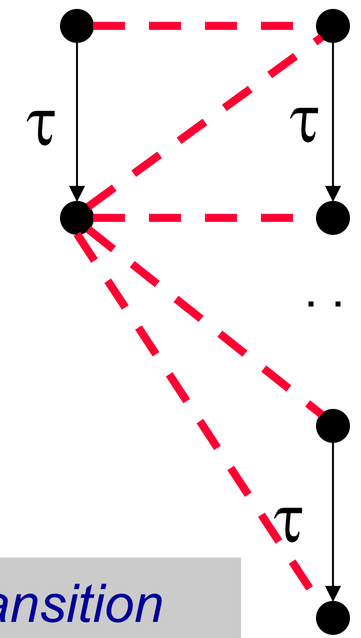
  ➔ *does not abstract away the internal behaviour*

Weak bisimulation
(or *observational equivalence*)

- In practice, it is necessary to compare LTSs
  - By abstracting away internal actions
  - By distinguishing the branching
- *Weak bisimulation* [Milner-89]:



every a-transition corresponds to an a-transition preceded and followed by 0 or more τ-transitions



every τ -transition corresponds to 0 or more τ-transitions

Weak bisimulation

(formal definition)

- Let $M_1 = <S_1, A, T_1, s_{01}>$ and $M_2 = <S_2, A, T_2, s_{02}>$
- A weak bisimulation is a relation $\approx \subseteq S_1 \times S_2$ such that $s_1 \approx s_2$ iff:

  $\forall\ s_1 \text{ -}a\rightarrow s_1' \ .\ \exists\ s_2 \text{ -}\tau^*.a.\tau^*\rightarrow s_2' \ .\ s_1' \text{ eq } s_2'$

  $\forall\ s_1 \text{ -}\tau\rightarrow s_1' \ .\ \exists\ s_2 \text{ -}\tau^*\rightarrow s_2' \ .\ s_1' \text{ eq } s_2'$

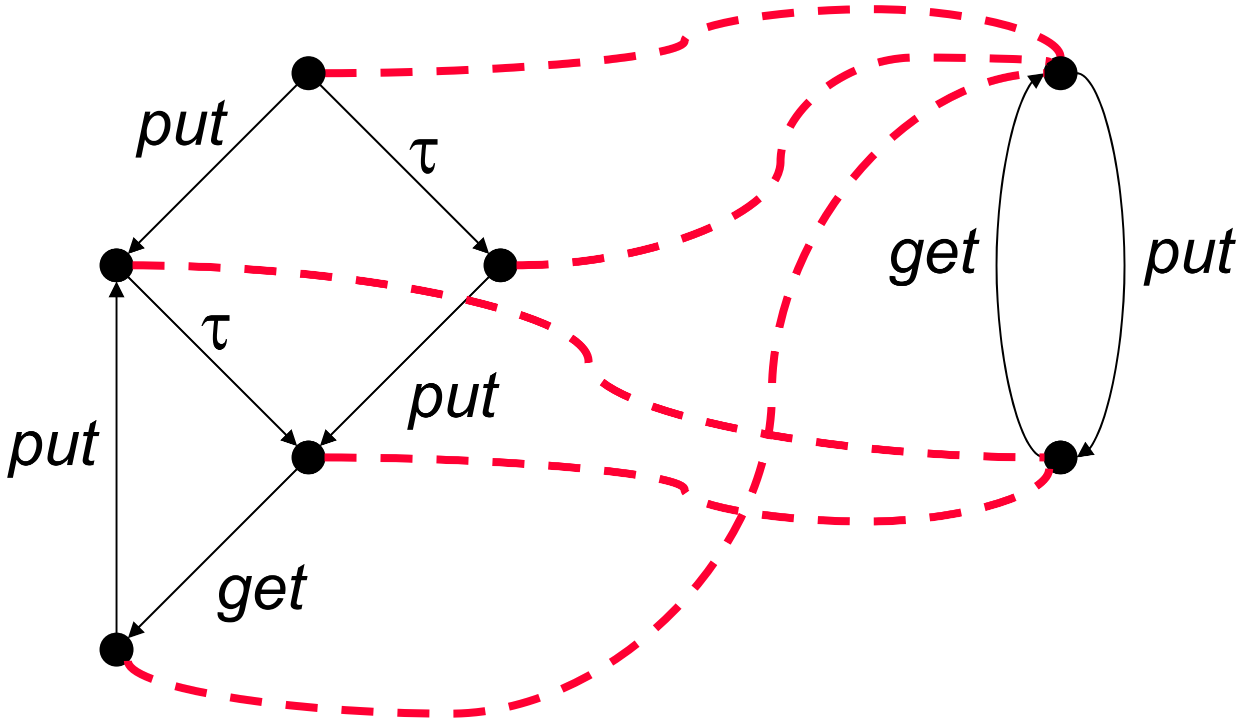
  and

  $\forall\ s_2 \text{ -}a\rightarrow s_2' \ .\ \exists\ s_1 \text{ -}\tau^*.a.\tau^*\rightarrow s_1' \ .\ s_1' \text{ eq } s_2'$

  $\forall\ s_2 \text{ -}\tau\rightarrow s_2' \ .\ \exists\ s_1 \text{ -}\tau^*\rightarrow s_1' \ .\ s_1' \text{ eq } s_2'$
- $\approx_{obs}$ is the largest weak bisimulation
- $M_1 \approx_{obs} M_2$ iff $s_{01} \approx_{obs} s_{02}$

Example

- To show that two LTSs are weakly bisimilar, it is sufficient to find a weak bisimulation between them



Communicating automata
(summary)

- **Advantages:**
  - Simple model for describing concurrency
  - Powerful tools for manipulation
    - MEC (University of Bordeaux)
    - Auto/Autograph/FC2 (INRIA, Sophia-Antipolis)
    - CADP (INRIA, Grenoble)
  - Some industrial applications
- **Shortcomings:**
  - Limited expressiveness
    - No dynamic creation and destruction of automata
    - Impossible to express: A then (B || C) then D
    - No handling of data (each variable = an automaton), unacceptable for complex types (numbers, lists, structures, ...)
  - Maintenance difficult and error-prone (large automata)

Process algebraic languages

- Basic notions
- Parallel composition and hiding
- Sequential composition and choice
- Value-passing and guards
- Process definition and instantiation

Process algebras

- PAs: theoretical formalisms for describing and studying concurrency and communication
- Examples of PAs for asynchronous systems:
  - CCS (*Calculus of Communicating Systems*) [Milner-89]
  - CSP (*Communicating Sequential Processes*) [Hoare-85]
  - ACP (*Algebra of Communicating Processes*) [Bergstra-Klop-84]
- Basic idea of PAs:
  - Provide a small number of operators
  - Construct behaviours by freely combining operators (lego)
- Standardized specification languages:
  - LOTOS [ISO-1988], E-LOTOS [ISO-2001]

LOTOS
(Language Of Temporal Ordering Specification)

- International standard [ISO 8807] for the formal specification of telecommunication protocols and distributed systems

  http://www.inrialpes.fr/vasy/cadp/tutorial

- Enhanced LOTOS (E-LOTOS): revised standard [2001]
- LOTOS contains two “orthogonal” sublanguages:
  - *data* part (for data structures)
  - *process* part (for behaviours)
- Handling data is necessary for describing realistic systems. “Basic LOTOS” (the dataless fragment of LOTOS) is useful only for small examples.

LOTOS - data part

- Based on algebraic abstract data types (ActOne):

```
type Natural is
    sorts Nat
    opns  0 : -> Nat
          succ : Nat -> Nat
          + : Nat, Nat -> Nat
    eqns forall M, N : Nat
      ofsort Nat
          0 + N = N;
          succ(M) + N = succ(M + N);
endtype
```

- Caesar.Adt compiler of CADP [Garavel-Turlier-92]
- ADTs tend to become cumbersome for complex data manipulations (removed in E-LOTOS).

LOTOS - process part

- Combines the best features of the process algebras CCS [Milner-89] and CSP [Hoare-85]
- Terminal symbols (identifiers):
  - Variables: $X_1, \ldots, X_n$
  - Gates: $G_1, \ldots, G_n$
  - Processes: $P_1, \ldots, P_n$
  - Sorts (≈ types): $S_1, \ldots, S_n$
  - Functions: $F_1, \ldots, F_n$
  - Comments: (* … *)
- Caesar compiler of CADP [Garavel-Sifakis-90]

Value expressions and offers

- Value expressions: $V_1, \ldots, V_n$

  $V ::= X$

  $\quad | \quad F\,(V_1, \ldots, V_n)$

  $\quad | \quad V_1\; F\; V_2$

- Offers: $O_1, \ldots, O_n$

  $O ::= \;!\,V$ — emission of a value V

  $\quad | \quad ?\,X : S$ — reception of a value to be stored in a variable X of sort S

Behaviour expressions
(Lots Of Terribly Obscure Symbols :-)

- Behaviours: B_1, ..., B_n

B ::=	**stop**	inaction
|	$G_0\ O_1\ \ldots\ O_n$ [V] ; B_0	action prefix
|	B_1 [] B_2	choice
|	B_1 |[G_1, ..., G_n]| B_2	parallel with synchronization on G_1, ..., G_n
|	B_1 ||| B_2	interleaving
|	**hide** G_1, ..., G_n **in** B_0	hiding
|	[V] -> B_0	guard
|	**let** X : S = V **in** B_0	variable definition
|	**choice** X : S [] B_0	choice over values
|	P [G_1, ..., G_n] (V_1, ..., V_n)	process call

Process definitions

process P [G_1, ..., G_n] (X_1:S_1, ..., X_n:S_n) :=

B

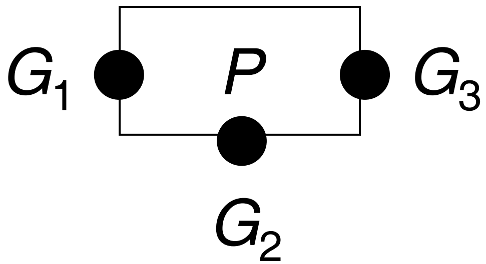
endproc

where:

- P = process name
- G_1, ..., G_n = formal *gate* parameters of P
- X_1, ..., X_n = formal *value* parameters of P,
  of sorts S_1, ..., S_n
- B = body (behaviour) of P

Remarks

- LOTOS process: "black box" equipped with communication points (gates) with the outside



process P [G_1, G_2, G_3] (...) :=

...

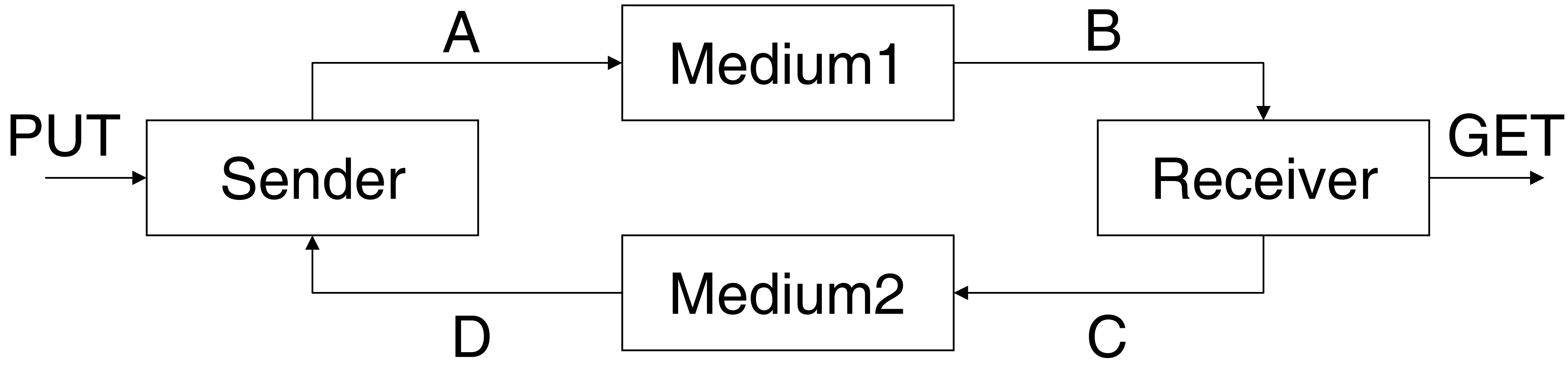
endproc

- Each process has its own local (private) variables, which are not accessible from the outside

  ➔ *communication by rendezvous and not by shared variables*

- Parallel composition and encapsulation of boxes: described using the |[...]|, |||, and **hide** operators

Example



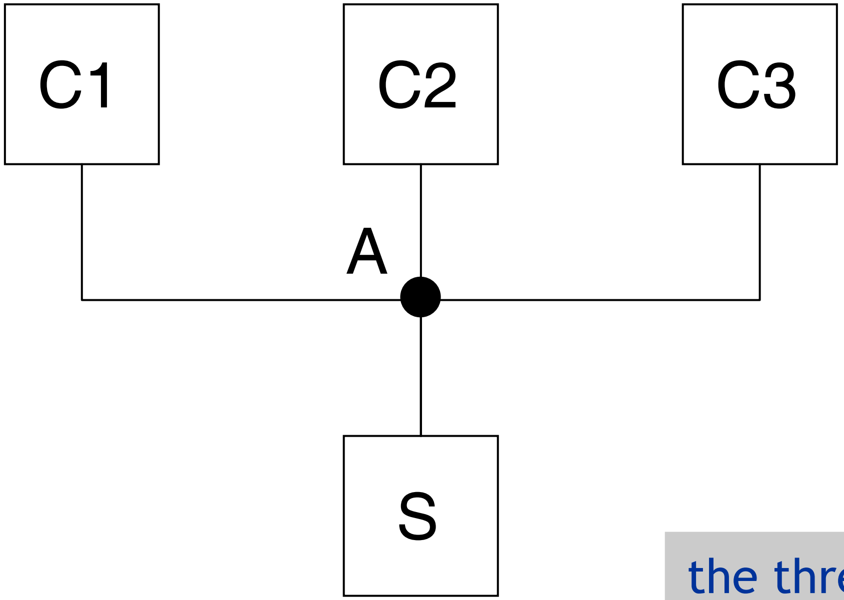
(Sender [PUT, A, D] ||| Receiver [GET, B, C])
|[A, B, C, D]|
(Medium1 [A, B] ||| Medium2 [C, D])

or

(Sender [PUT, A, D] |[A]| Medium1 [A, B])
|[B, D]|
(Receiver [GET, B, C] |[C]| Medium2 [C, D])

Multiple rendezvous

- LOTOS parallel operators allow to specify the synchronization of $n \geq 2$ processes on the same gate



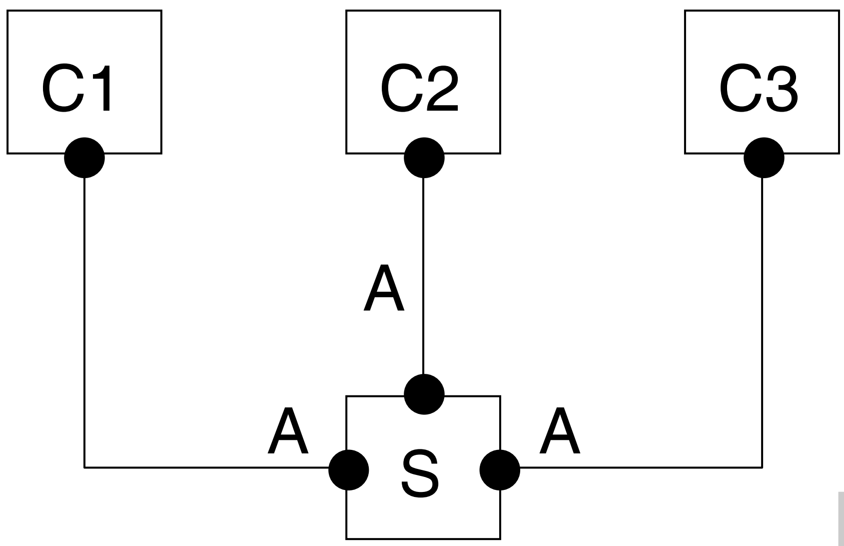
Example (client-server):

```
C1 [A] |[A]| C2 [A] |[A]| C3 [A]
 |[A]|
S [A]
```

the three client processes synchronize with the server on gate A (4-way rendezvous)

Binary rendezvous

- The ||| operator allows to specify binary rendezvous (2 among *n*) on the same gate



Example (client-server):

(C1 [A] ||| C2 [A] ||| C3 [A])
|[A]|
S [A]

the three client processes are competing to access the server on gate A but only one can get access at a given moment

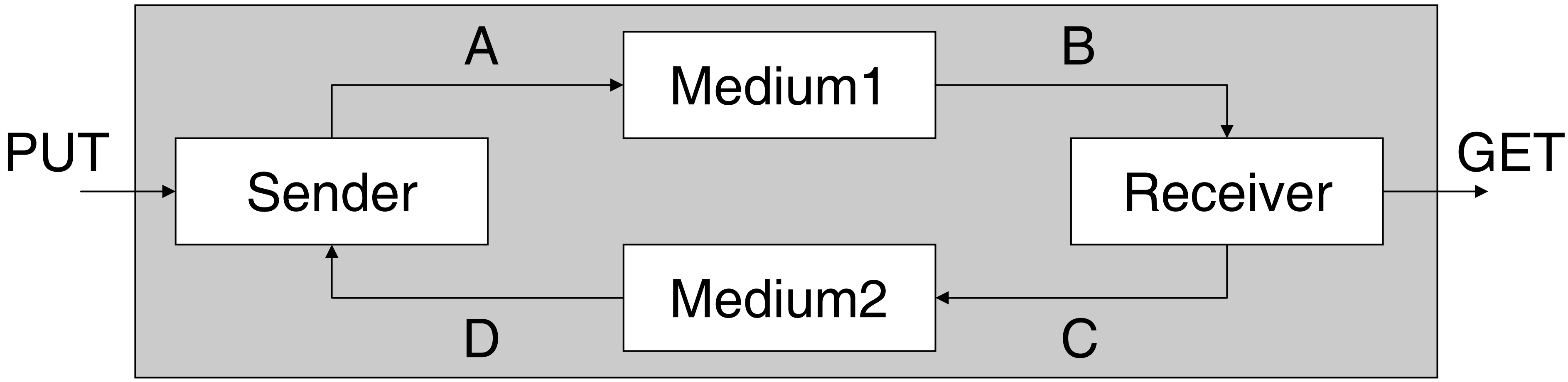
Abstraction
(hiding)

- In LOTOS, when a synchronization takes place on a gate G between two processes, another one can also synchronize on G (*maximal cooperation*)
- If this is undesirable, it can be forbidden by hiding the gate (renaming it into ***i***) using the **hide** operator:

  $$\textbf{hide } G_1, \ldots, G_n \textbf{ in } B$$

  which means that all actions performed by B on gates $G_1, \ldots, G_n$ are hidden
- The gates $G_1, \ldots, G_n$ are “abstracted away” (hidden from the outside world)

Example



process Network [PUT, GET] :=

hide A, B, C, D **in**

(Sender [PUT, A, D] ||| Receiver [GET, B, C])

|[A, B, C, D]|

(Medium1 [A, B] ||| Medium2 [C, D])

endproc

Operational semantics

- Notations:
  - $\underline{G}$: gate list (or set)
  - *L*: action (transition label), of the form

    $$G\ V_1, \ldots, V_n$$

    where *G* is a gate and $V_1, \ldots, V_n$ is the list of values exchanged on *G* during the rendezvous
  - *gate* (*L*) = *G*
  - *B* [*v* / *X*]: syntactic substitution of all free occurrences of *X* inside *B* by a value *v* (having the same sort as *X*)
  - *V* [*v* / *X*]: idem, substitution of *X* by *v* in *V*

Semantics of "|[...]|"

$$\frac{B_1 \rightarrow_L B_1' \wedge gate(L) \notin \underline{G}}{B_1 \;|[\, \underline{G} \,]|\; B_2 \rightarrow_L B_1' \;|[\, \underline{G} \,]|\; B_2}$$

B_1 evolves

$$\frac{B_2 \rightarrow_L B_2' \wedge gate(L) \notin \underline{G}}{B_1 \;|[\, \underline{G} \,]|\; B_2 \rightarrow_L B_1 \;|[\, \underline{G} \,]|\; B_2'}$$

B_2 evolves

$$\frac{B_1 \rightarrow_L B_1' \wedge B_2 \rightarrow_L B_2' \wedge gate(L) \in \underline{G}}{B_1 \;|[\, \underline{G} \,]|\; B_2 \rightarrow_L B_1' \;|[\, \underline{G} \,]|\; B_2'}$$

B_1 and B_2 evolve

- Gates have no direction of communication

Semantics of "hide"

$$\frac{B \rightarrow_L B' \wedge gate\ (L) \notin \underline{G}}{\textbf{hide}\ \underline{G}\ \textbf{in}\ B \rightarrow_L \textbf{hide}\ \underline{G}\ \textbf{in}\ B'}$$

normal gate

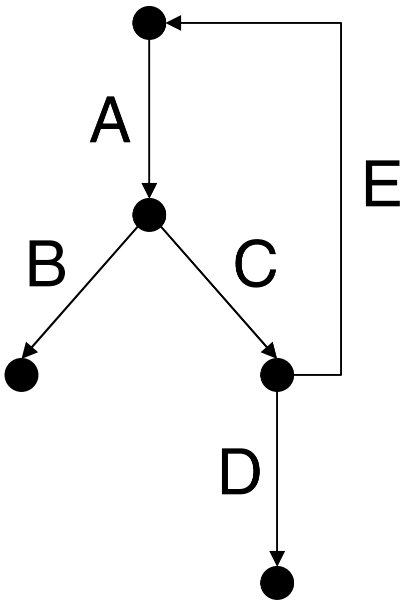
$$\frac{B \rightarrow_L B' \wedge gate\ (L) \in \underline{G}}{\textbf{hide}\ \underline{G}\ \textbf{in}\ B \rightarrow_{\mathbf{i}} \textbf{hide}\ \underline{G}\ \textbf{in}\ B'}$$

hidden gate

- In LOTOS, **i** is a keyword: use with care

Sequential behaviours

- LOTOS allows to encode sequential automata by means of the choice ("[]") and sequence operators (";" and "**stop**"), and recursive processes




```
process P [A, B, C, D, E] : noexit :=
    A; (
        B; stop
        []
        C; (
            D ; stop
            []
            E ; P [A, B, C, D, E]
        )
    )
endproc
```

Remarks

- The description of automata in LOTOS is not far from regular expressions (operators “.”, “|”, “*”), except that:
  - The “;” operator of LOTOS is *asymmetric* (≠ from “.”)

    $G\ O_1 \ldots O_n\ ;\ B$ but not $B_1\ ;\ B_2$
  - There is no iteration operator “*”, one must use a recursive process call instead

- LOTOS allows to describe automata with data values (≈ functions in sequential languages) by using processes with value parameters

Semantics of “stop”

- The “**stop**” operator (inaction) has no associated semantic rule, because no transition can be derived from it

- A call of a “pathological” recursive process like

  process P [A] : **noexit** :=

  P [A]

  endproc

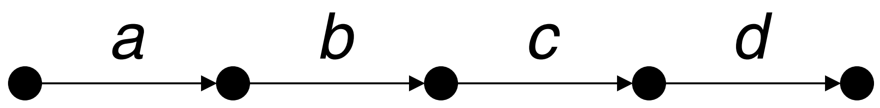
  has a behaviour equivalent to **stop** (unguarded recursion)

Prefix operator (“;”)

- Allows to describe:
  - Sequential composition of actions
  - Communication (emission / reception) of data values

- Simplest variant: actions on gates, without value-passing (basic LOTOS)

a ; *b* ; *c* ; *d* ; **stop**



Semantics of “;”

Case 1: action without reception offers ($?X:S$)

$$\frac{(\forall 1 \le i \le n \;.\; O_i \equiv \,!\, V_i) \wedge V = \text{true}}{G\ O_1 \ldots O_n\ [\,V\,]\ ;\ B \rightarrow_{G\ V1\ \ldots\ Vn} B}$$

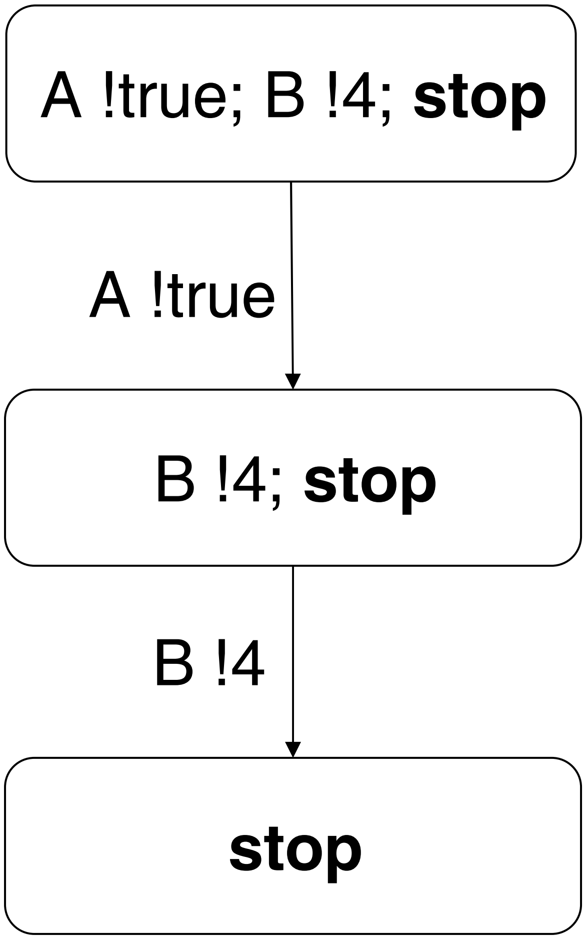
- The boolean guard and the offers are optional
- If the guard V is false, the rendezvous does not happen (deadlock):

$$G\ O_1 \ldots O_n\ [\,V\,]\ ;\ B \quad \approx \quad \textbf{stop}$$

Example (1/2)

Sequential composition:

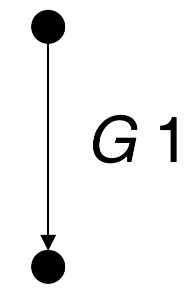
A !true; B !4; **stop**



Example (2/2)

- Synchronization by *value matching*: two processes send to each other the same values on a gate

$G\ !1;\ B_1\ \ |[\ G\ ]|\ \ G\ !1;\ B_2$ — RdV OK



$G\ !1;\ \mathrm{B}_1\ \ |[\ G\ ]|\ \ G\ !2;\ B_2$ — deadlock (different values)

$G\ !1;\ \mathrm{B}_1\ \ |[\ G\ ]|\ \ G\ !\mathrm{true};\ B_2$ — deadlock (different types)

Semantics of ";"

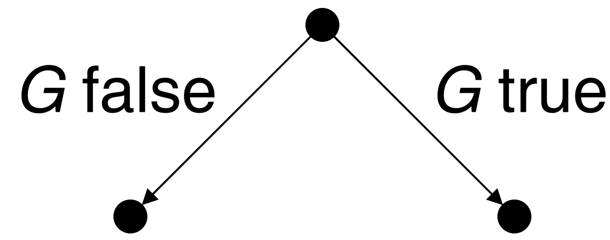
Case 2: action containing reception offer(s) (?X:S)

$$\frac{(v \in S) \wedge (V[v / X] = \text{true})}{G\ ?X{:}S\ [V]\ ;\ B \rightarrow_{G\ v} B\ [v / X]}$$

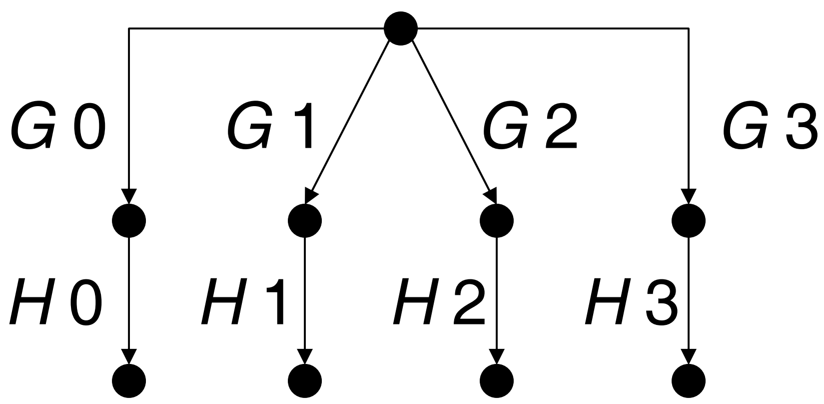
- The variables defined in the offers ?X:S are visible in the boolean guard *V* and inside *B*
- An action can freely mix emission and reception offers

Example (1/3)

G ?*X*:Bool;
stop



G ?*X*:Nat [*X* < 4];
H ! X;
stop



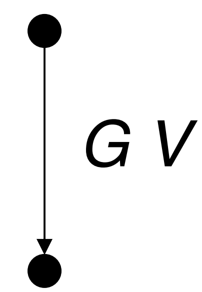
- The semantics handles the reception by branching on all possible values that can be received

Example (2/3)

- Emission of a value = guarded reception:

  $G\ !V \equiv G\ ?X{:}S\ [\ X = V\ ]$

  where $S = type\ (V\ )$

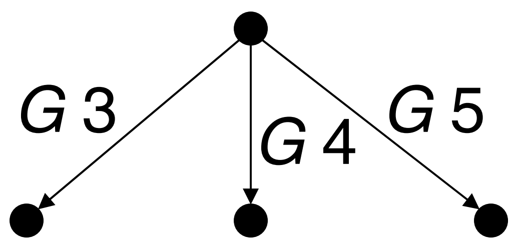


- Synchronization by *value generation*: two processes receive values of the same type on a gate

  $G\ ?n_1{:}Nat\ [\ n_1 <= 5\ ];\ B_1$

  $|[\ G\ ]|$

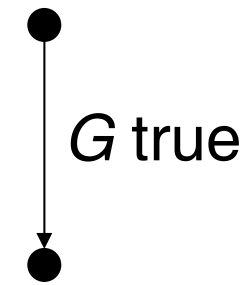
  $G\ ?n_2{:}Nat\ [\ n_2 > 2];\ B_2$



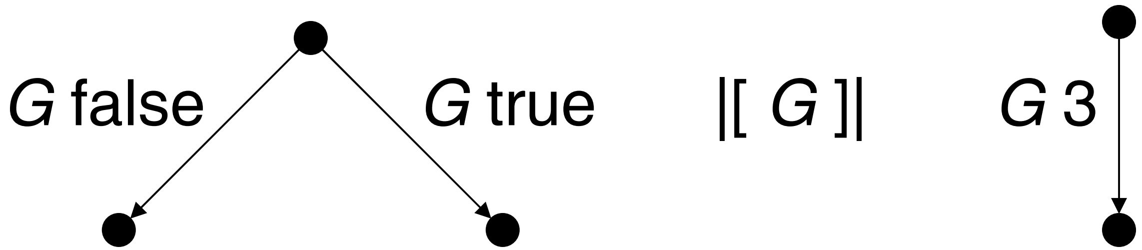
Example (3/3)

- Synchronization by *value-passing*:

G ?*X*:Bool ; **stop** |[*G*]| *G* !true ; **stop**



G ?*X*:Bool ; **stop** |[*G*]| *G* !3 ; **stop**



deadlock: the semantics of the "|[...]|" operator requires that the two labels be identical (same type for the emitted value and the reception offer)

Rendezvous
(summary)

- General form:

  $G\ O_1 \ldots O_m\ [V_1];\ B_1 \quad |[\ \underline{G}\ ]| \quad G'\ O_1' \ldots O_n'[V_2];\ B_2$

- Conditions for the rendezvous:
  - $G = G'$ and $G \in \underline{G}$
  - $m = n$
  - V_1 and V_2 are true in the context of $O_1, \ldots, O_n'$
  - $\forall 1 \leq i \leq n.\ type\ (O_i) = type\ (O_i')$
  - $\forall 1 \leq i \leq n.\ prop\ (O_i) \cap prop\ (O_i') \neq \emptyset$

  where $prop(O)$ = set of values accepted by offer O
  - $prop\ (!V\ ) = \{\ V\ \}$
  - $prop\ (?X{:}S) = S$

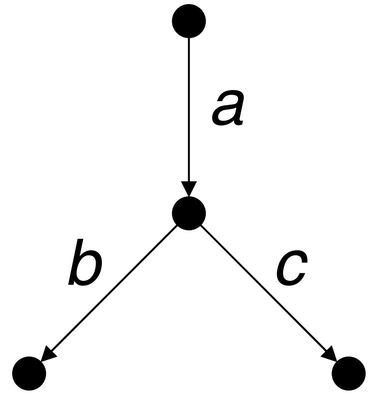
Choice operator ("[]")

- "[]": notation inherited from the programs with guarded commands [Dijkstra]
- Allows to specify the choice between several alternatives:

  $(B_1 \; [] \; B_2 \; [] \; B_3)$

  can execute either B_1, or B_2, or B_3
- Example:

  a ;
  (*b* ; **stop**
  []
  c ; **stop**)



Semantics of "[]"

$$\frac{B_1 \rightarrow_L B_1'}{B_1 \;[]\; B_2 \rightarrow_L B_1'}$$

execution of B_1

$$\frac{B_2 \rightarrow_L B_2'}{B_1 \;[]\; B_2 \rightarrow_L B_2'}$$

execution of B_2

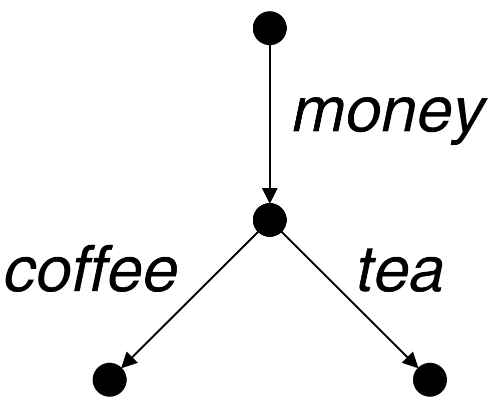
- After the choice, one of the two behaviours disappears (the execution was engaged on a branch of the choice and the other one is abandoned)

Internal / external choice

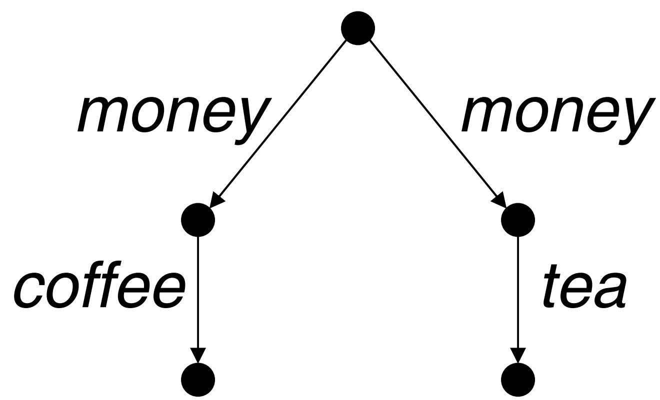
(G_1 ; B_1 [] G_2 ; B_2)

- External choice: the environment can decide which branch will be executed
- Internal choice: the program decides

- Example (coffee machine):



external choice (user)

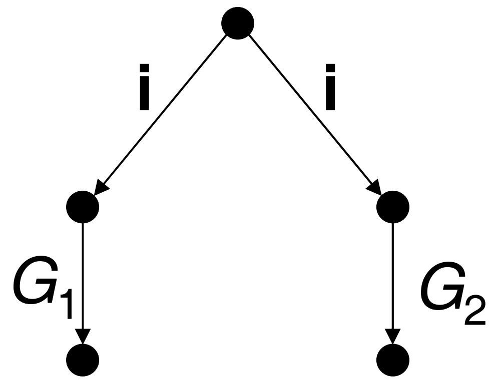


internal choice (machine)

Internal action ("i")

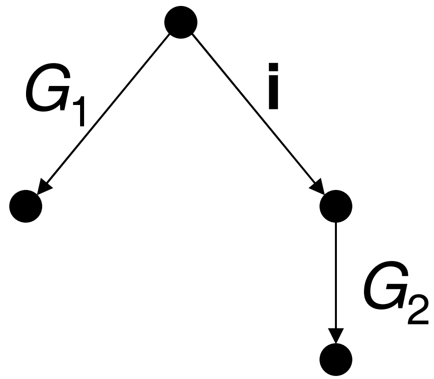
- In LOTOS, the special gate **i** denotes an internal event on which the environment cannot act:

(**i** ; G_1 ; **stop**
[]
i ; G_2 ; **stop**)



internal choice

(${G_1}$; **stop**
[]
i ; G_2 ; **stop**)



still internal choice

Guard operator ("[...] ->")

- LOTOS does not possess an "if-then-else" construct
- *Guards* (boolean conditions) can be used instead
- Informal semantics:

  [*V*] -> *B* ≈ **if** *V* **then** *B* **else stop**

- Frequent usage in conjunction with "[]":

```
READ ?m,n:Nat ;
( [ m >= n ] -> PRINT !m; stop
  []
  [ m < n ]   -> PRINT !n; stop )
```

emission of max (m,n) on gate PRINT

Semantics of “[...] ->”

$$\frac{(V = \text{true}) \wedge B \rightarrow_L B'}{[\ V\ ] \text{ -> } B \rightarrow_L B'}$$

- If the boolean expression *V* evaluates to false, no semantic rule applies (deadlock):

  [false] -> *B* $\approx$ **stop**

Examples

- "if-then-else":

  $[\ V\ ] \text{ -> } B_1$

  []

  $[\ \text{not}\ (V\ )\ ] \text{ -> } B_2$

  "case":

  $[\ X < 0\ ] \text{ -> } B_1$

  []

  $[\ X = 0\ ] \text{ -> } B_2$

  []

  $[\ X > 0\ ] \text{ -> } B_3$

- Beware of overlapping guards:

  $[\ X \leq 0\ ] \text{ -> } B_1$

  []

  $[\ X \geq 0\ ] \text{ -> } B_2$

if $X = 0$ then this is equivalent to an unguarded choice $B1$ [] $B2$

Operator "let"

- LOTOS allows to define variables for storing the results of expressions
- Variable definition:

  let *X*:*S* = *V* **in** *B*

  declares variable *X* and initializes it with the value of *V*. *X* is visible in *B*.
- *Write-once* variables (no multiple assignments):

  let *X*:*Bool* = *true* **in** *G* !*X* ; (* first *X* *)

  let *X*:*Bool* = *false* **in** *G* !*X* ; (* second *X* *)

  stop

Semantics of "let"

$$\frac{B\,[\,V\,/\,X\,] \rightarrow_L B'}{\textbf{let}\ X{:}S = V\ \textbf{in}\ B \rightarrow_L B'}$$

- Example:

let *X*:NatList = cons (0, nil) **in**
G !*X*;
H !cons (1, *X*);
stop

Remarks

LOTOS is a *functional* language:

- No uninitialized variable (forbidden by the syntax)
- No assignment operator (":="), the value of a variable does not change after its initialization
- No "global" or "shared" variables between functions or processes
- Each process has its own local variables
- Communication by rendezvous only
- No side-effects

Operator “choice”

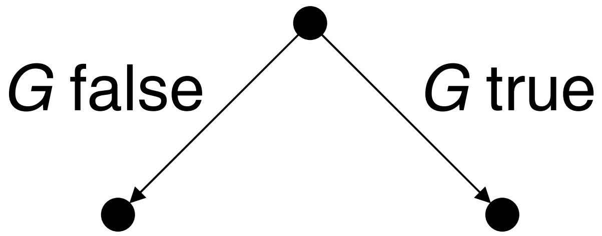
- Operator “**choice**”: similar to “**let**”, except that variable X takes a nondeterministic value in the domain of its sort S
- Semantics:

$$\frac{(v \in S) \wedge B\,[\,v \,/\, X\,] \rightarrow_L B'}{\textbf{choice}\ X{:}S\ [\,]\ B \rightarrow_L B'}$$

- Example:

choice *X*:Bool []

G !*X*; **stop**



Examples

- Reception of a value = particular case of "**choice**":

  G ?X:*S* ; *B* = **choice** *X*:*S* [] *B*

- Iteration over the values of an enumerated type:

  choice *A*:*Addr* []

  SEND !*m* !*A* ; **stop**

- Generation of a random value:

  choice *rand*:*Nat* []

  [*rand* <= 10] -> *PRINT* !*rand* ; **stop**

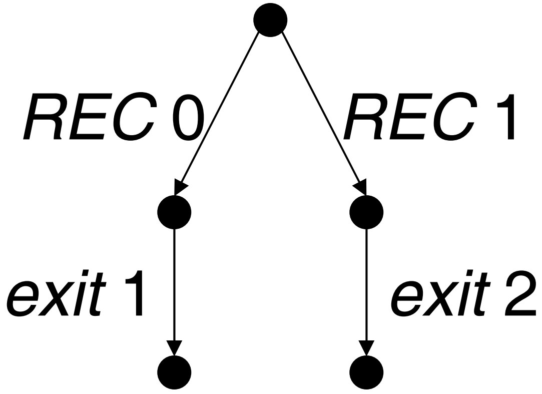
Operator “exit”

- LOTOS allows to express *normal termination* of a behaviour, possibly with the return of one or several values:

  exit (V_1, ..., V_n)

  denotes a behaviour that terminates and produces the values V_1, ..., V_n
- Example:

  REC ?*x*:Nat [*x* < 2] ;
  exit (*x* + 1)



Semantics of “exit”

$$\frac{\text{true}}{\textbf{exit}\ (\ V_1, \ldots, V_n\ ) \rightarrow_{exit\ V1\ \ldots\ Vn} \textbf{stop}}$$

- *exit* = special gate, synchronized by the “|[...]|” operator (see later)
- The values $V_1, \ldots, V_n$ are optional (“**exit**” means normal termination without producing any value)

Operator “>>”

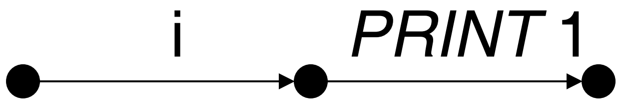
- LOTOS allows to express the sequential composition between a behaviour B_1 that terminates and a behaviour B_2 that begins:

  B_1 >> **accept** X_1:S_1,..., X_n:S_n **in** B_2

  means that when B_1 terminates by producing values V_1,..., V_n, the execution continues with B_2 in which X_1,..., X_n are replaced by the values V_1,..., V_n
- Example:

  exit (1) >> accept n:Nat in

  PRINT !n ; stop



Semantics of “>>”

$$\frac{(B_1 \rightarrow_L B_1') \wedge (gate\ (L) \neq exit\ )}{(B_1 >> \textbf{accept}\ \underline{X}{:}\underline{S}\ \textbf{in}\ B_2) \rightarrow_L (B_1' >> \textbf{accept}\ \underline{X}{:}\underline{S}\ \textbf{in}\ B_2)}$$

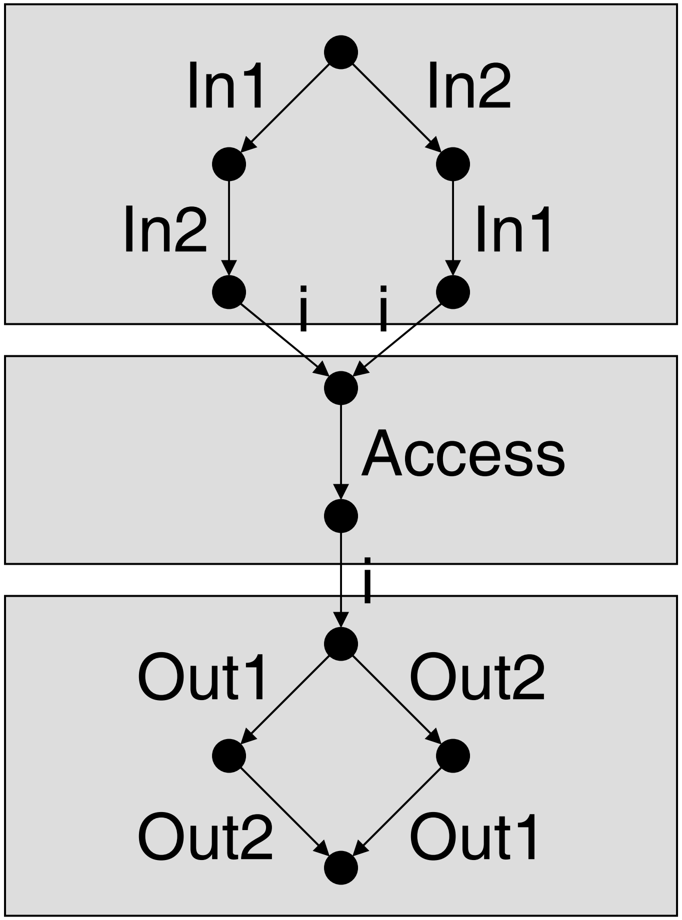
$$\frac{B_1 \rightarrow_{exit\ \underline{V}} B_1'}{(B_1 >> \textbf{accept}\ \underline{X}{:}\underline{S}\ \textbf{in}\ B_2) \rightarrow_{\mathbf{i}} B_2\ [\ \underline{V}\ /\ \underline{X}\ ]}$$

- The $\underline{V}$ values must belong pairwise to the $\underline{S}$ sorts
- The *exit* gate is hidden (renamed into **i**) when sequential composition takes place
- The “>>” operator is also called *enabling* (B_2’s execution is made possible by B_1’s termination)

Example (1/4)

- Sequential composition without value-passing:

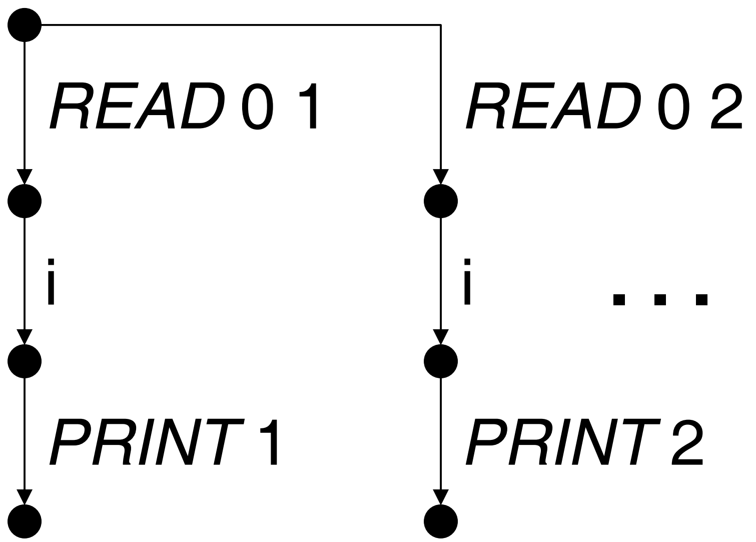
```
(In1; In2; exit
 []
 In2; In1; exit)
>>
(Access; exit)
>>
(Out1; Out2; stop
 []
 Out2; Out1; stop)
```



Example (2/4)

- Sequential composition with value-passing:

```
READ ?m,n:Nat ;
( [ m >= n ] -> exit (m)
  []
  [ m < n ] -> exit (n) )
>>
accept max:Nat in
PRINT !max ; stop
```



Example (3/4)

- Definition of terminating process:

  process Login [LogReq, LogConf, LogAbort] : **exit** :=
      LogReq;
      (i ; LogConf ; exit
        []
        i ; LogAbort ; Login [LogReq, LogConf, LogAbort])
  endproc

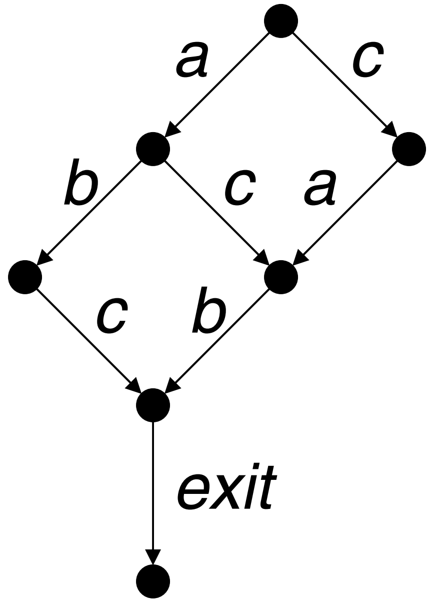
- Example of call:

  Login [Req,Conf,Abort] >> Transfer ; Logout ; stop

Example (4/4)

- Combination of “**exit**” and parallel composition: the two behaviours are synchronized on the *exit* gate (they terminate simultaneously)

(*a* ; *b* ; exit) ||| (*c* ; exit)

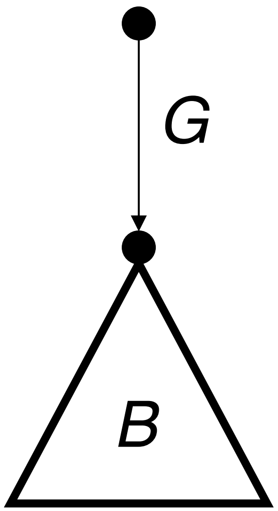


Sequential composition
(summary)

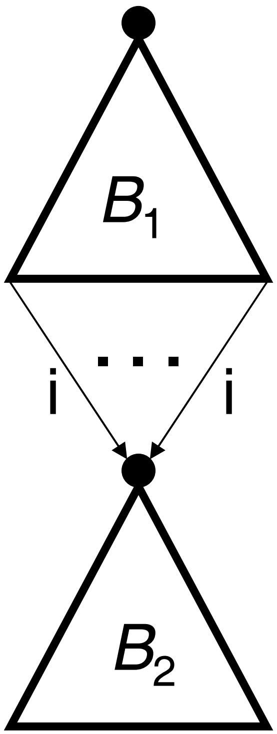
- In LOTOS, difference between
  "**;**" (asymmetric)

  and

  "**>>**" (symmetric):



$G\ ;\ B$



$B_1 >> B_2$

Process call

- Let a process P defined by:

  process P [G_1, …, G_n] (X_1:S_1, …, X_n:S_n) :=

  B

  endproc

- Semantics of a call to P:

$$\frac{B\,[\,g_1 / G_1, \ldots, g_n / G_n\,]\,[\,v_1 / X_1\,, \ldots, v_n / X_n\,] \rightarrow_L B'}{P\,[g_1, \ldots, g_n]\,(v_1, \ldots, v_n) \rightarrow_L B'}$$

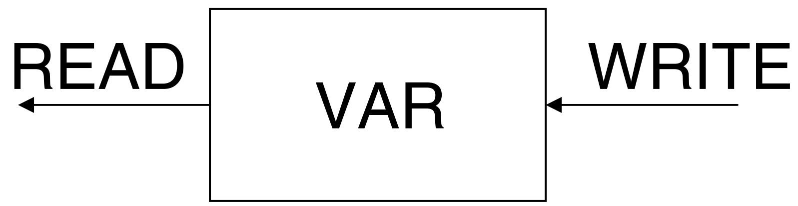
- This semantics explains why a call to

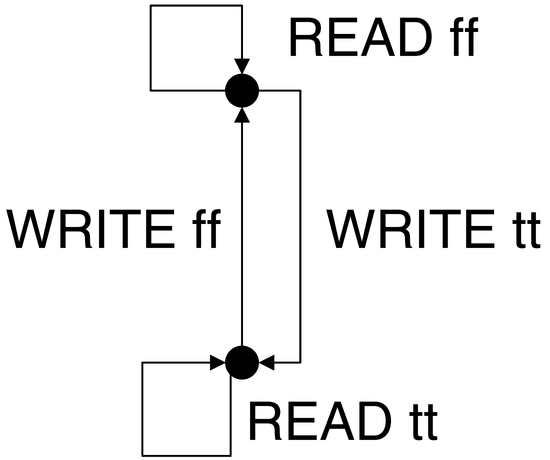
  process P[G] : **noexit** := P[G] **endproc**

  is equivalent to **stop**.

Example

- Boolean variable:






```
process VAR [READ, WRITE] (b:Bool) : noexit :=
  READ !b;
     VAR [READ, WRITE] (b)
  []
  WRITE ?b2:Bool;
     VAR [READ, WRITE] (b2)
endproc
```

Static semantics

(summary)

- Scope of variables inside behaviours:

B ::= G !V_0 ?X:S … [V] ; B_0	p (X) = { V, B_0 }
| hide G in B_0	p (G) = { B_0 }
| let X:S = V in B_0	p (X) = { B_0 }
| choice X:S [] B_0	p (X) = { B_0 }
| B_1 >> accept X:S in B_0	p (X) = { B_0 }

- Scope of process parameters:

process P [G] (X:S) :=	p (G) = { B_0 }
B_0	p (X) = { B_0 }
endproc	

LOTOS specification

- A LOTOS specification is similar to a process definition:

specification Protocol [SEND, RECEIVE] : **noexit** :=

(* ... type definitions *)

behaviour

(* ... behaviour = body of the specification *)

where

(* ... process definitions *)

endspec

Example: Peterson's mutual exclusion algorithm

var d0 : bool := false { read by P1, written by P0 }
var d1 : bool := false { read by P0, written by P1 }
var t ∈ {0, 1} := 0 { read/written by P0 and P1}

loop forever { P0 }	**loop forever** { P1 }
1 : { ncs0 }	1 : { ncs1 }
2 : d0 := true	2 : d1 := true
3 : t := 0	3 : t := 1
4 : **wait** (d1 = false **or** t = 1)	4 : **wait** (d0 = false **or** t = 0)
5 : { cs0 }	5 : { cs1 }
6 : d0 := false	6 : d1 := false
endloop	**endloop**

Description of variables d0, d1

- Each variable: instance of the same process D
- Current value of the variable: parameter of D
- Reading and writing: RdV on gates R et W

```
process D [R, W] (b:Bool) : noexit :=
    R !b ; D [R, W] (b)
    []
    W ?b2:Bool ; D [R, W] (b2)
endproc
```

- d0 ≡ D [R0, W0] (false), d1 ≡ D [R1, W1] (false)

Description of variable t

- Variable t: instance of process T
- Current value of the variable: parameter of T
- Reading and writing: RdV on gates R et W

```
process T [R, W] (n:Nat) : noexit :=
    R !n ; T [R, W] (n)
    []
    W ?n2:Bool ; T [R, W] (n2)
endproc
```

- $t \equiv$ T [RT, WT] (0)

Description of processes P0 and P1

- Process P_m: instance of the same process P
- Index m of the process: parameter of P

```
process P [Rm, Wm, Rn, Wn, RT, WT, NCS, CS]
            (m:Nat) : noexit :=
   NCS !m ; Wm !true ; WT !m ;
      P2 [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m)
endproc
```

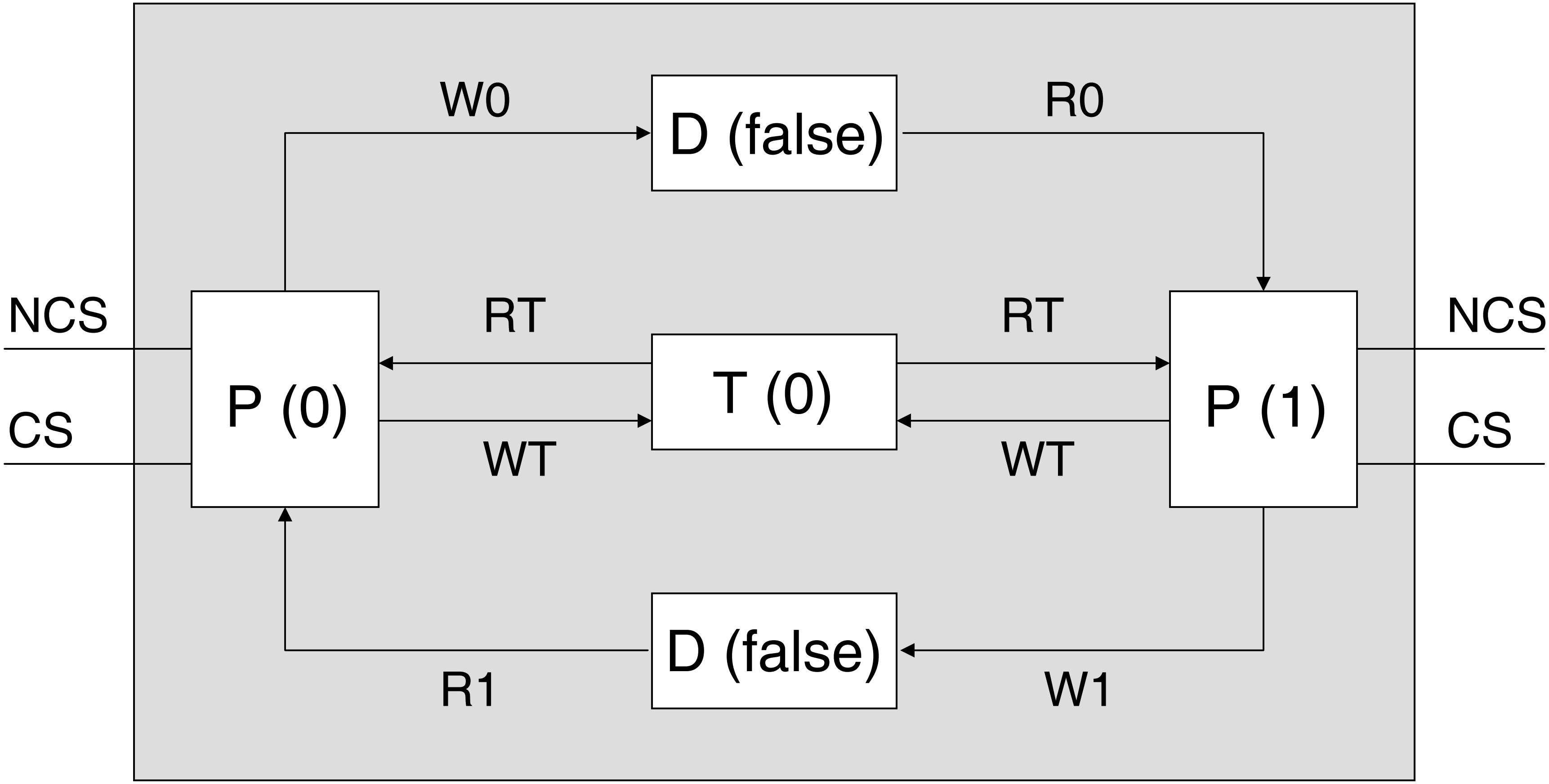
- P0 ≡ P [R0, W0, R1, W1, RT, WT, NCS, CS] (0)
- P1 ≡ P [R1, W1, R0, W0, RT, WT, NCS, CS] (1)

Processes P0 et P1
(continued)

- Auxiliairy process to describe busy waiting:

```
process P2 [Rm, Wm, Rn, Wn, RT, WT, NCS, CS]
              (m:Nat) : noexit :=
   Rn ?dn:Bool ; RT ?t:Nat ;
   ( [ dn and (t eq m) ] ->
        P2 [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m)
     []
     [ not (dn) or (t eq ((m + 1) mod 2)) ] ->
        CS !m ; Wn !false ;
        P [Rm, Wm, Rn, Wn, RT, WT, NCS, CS] (m) )
endproc
```

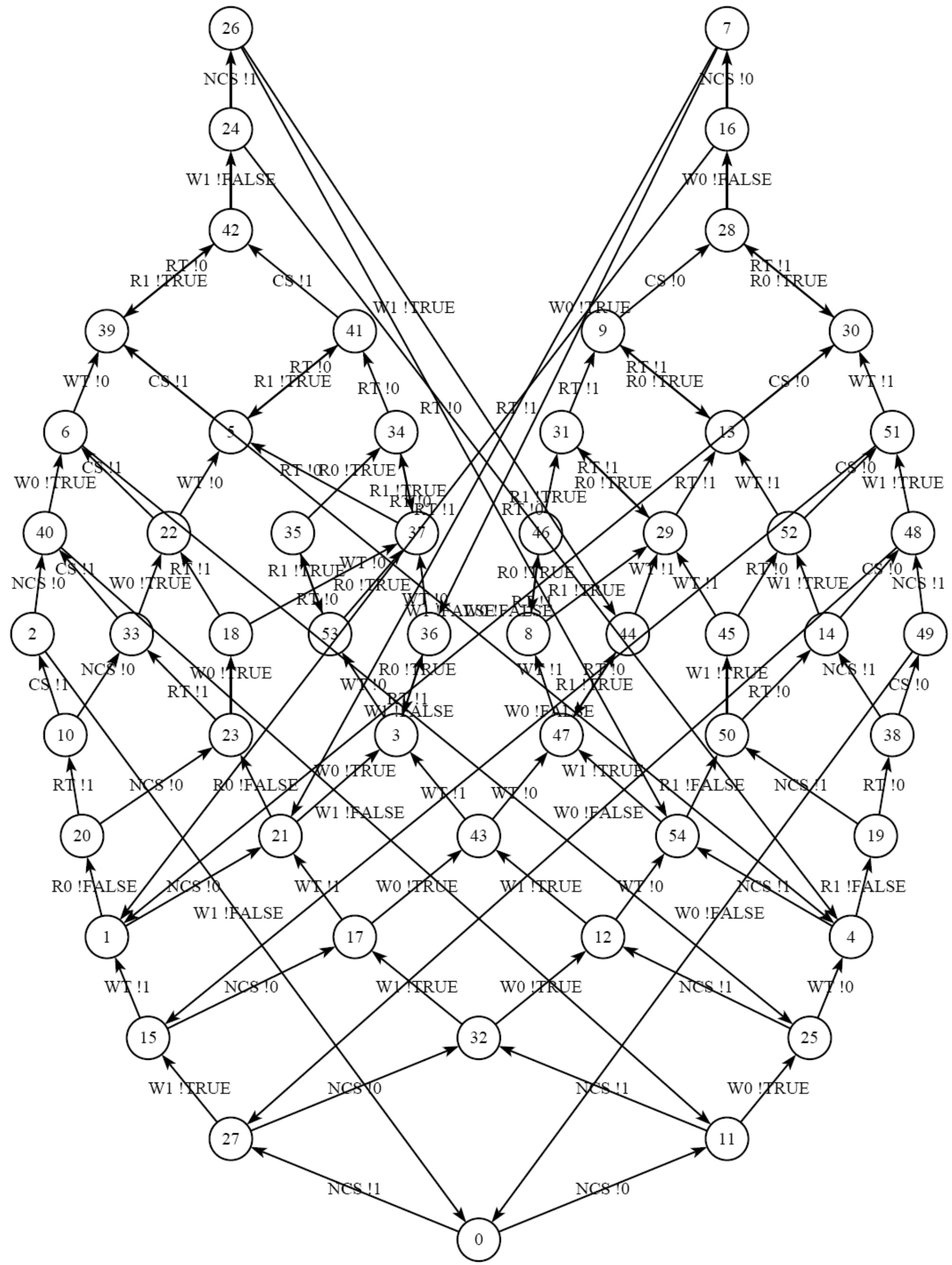
Architecture of the system
(graphical)



Architecture of the system
(textual)

```
hide R0, W0, R1, W1, RT, WT in
    (
        P [R0, W0, R1, W1, RT, WT, NCS, CS] (0)
        |||
        P [R1, W1, R0, W0, RT, WT, NCS, CS] (1)
    )
    |[ R0, W0, R1, W1, RT, WT ]|
    (
        T [RT, WT] (0)
        |||
        D [R0, W0] (false)
        |||
        D [R1, W1] (false)
    )
```

LTS model



- 55 states
- 110 transitions

Process algebraic languages
(summary)

- More concise than communicating automata: process parameterization, value-passing communication (Exercise: model variables d0, d1, t using a single gate allowing both reading / writing)
- In general, there are several ways of describing the parallel composition of processes (Exercise: write a different expression for the architecture of Peterson's algorithm)
- Modeling of nested loops: mutually recursive LOTOS processes (Exercise: model processes P0, P1 using a single LOTOS process)
- But: E-LOTOS process part is much more convenient

Action-based temporal logics

- Introduction
- Modal logics
- Branching-time logics
- Regular logics
- Fixed point logics

Why temporal logics?

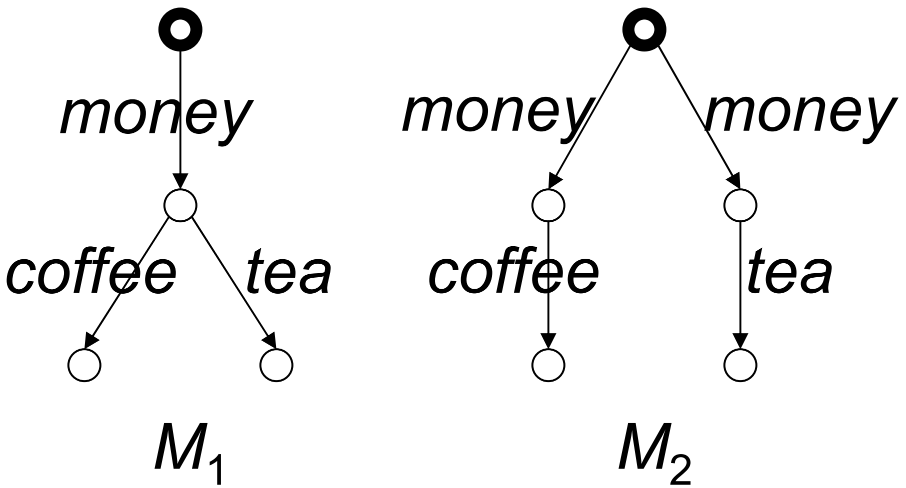
- Formalisms for high-level specification of systems
  - Example: all mutual exclusion protocols should satisfy
    - *Mutual exclusion* (at most one process in critical section)
    - *Liveness* (each process should eventually enter its critical section)
- Temporal logics (TLs):

  formalisms describing the ordering of states (or actions) during the execution of a concurrent program
- TL specification = list of logical formulas, each one expressing a property of the program
- Benefits of TL [Pnueli-77]:
  - *Abstraction*: properties expressed in TL are independent from the description/implementation of the system
  - *Modularity*: one can add/remove a property without impacting the other properties of the specification

(Rough) classification of TLs

	State-based	Action-based
Linear-time (properties about execution sequences)	LTL (SPIN tool) linear mu-calculus	TLA (TLA+ tool) action-based LTL (LTSA tool)
Branching-time (properties about execution trees)	CTL (nuSMV tool) CTL*	ACTL (JACK tool) ACTL* modal mu-calculus (CWB, Concurrency Factory, CADP tools)

Example
(coffee machine)



$L(M_1) = L(M_2) =$
$\{ money.coffee, money.tea \}$

- A linear-time TL cannot distinguish the two LTSs M_1 and M_2, which have the same set of execution sequences, but are not behaviourally equivalent (modulo strong bisimulation)
- A branching-time TL can capture nondeterminism and thus can distinguish M_1 and M_2

Interpretation of (branching-time) TLs on LTSs

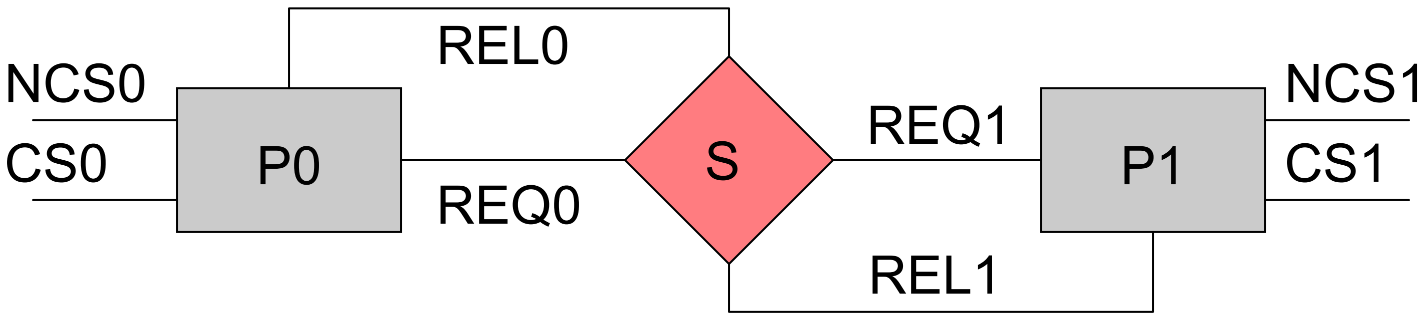
- LTS model $M = \langle S, A, T, s_0 \rangle$, where:
  - S: set of states
  - A: set of actions (events)
  - $T \in S \times A \times S$: transition relation
  - $s_0 \in S$: initial state
- Interpretation of a formula φ on M:

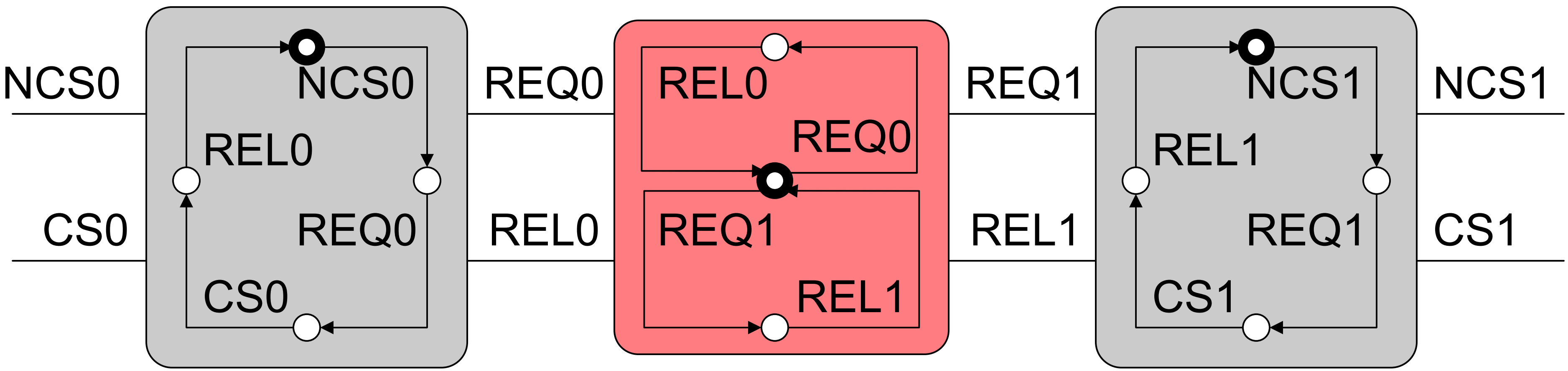
  $$[[\varphi]] = \{ s \in S \mid s \models \varphi \}$$

  ($[[\varphi]]$ defined inductively on the structure of φ)
- An LTS M satisfies a TL formula φ ($M \models \varphi$) iff its initial state satisfies φ :

$$M \models \varphi \quad \Leftrightarrow \quad s_0 \models \varphi \quad \Leftrightarrow \quad s_0 \in [[\varphi]]$$

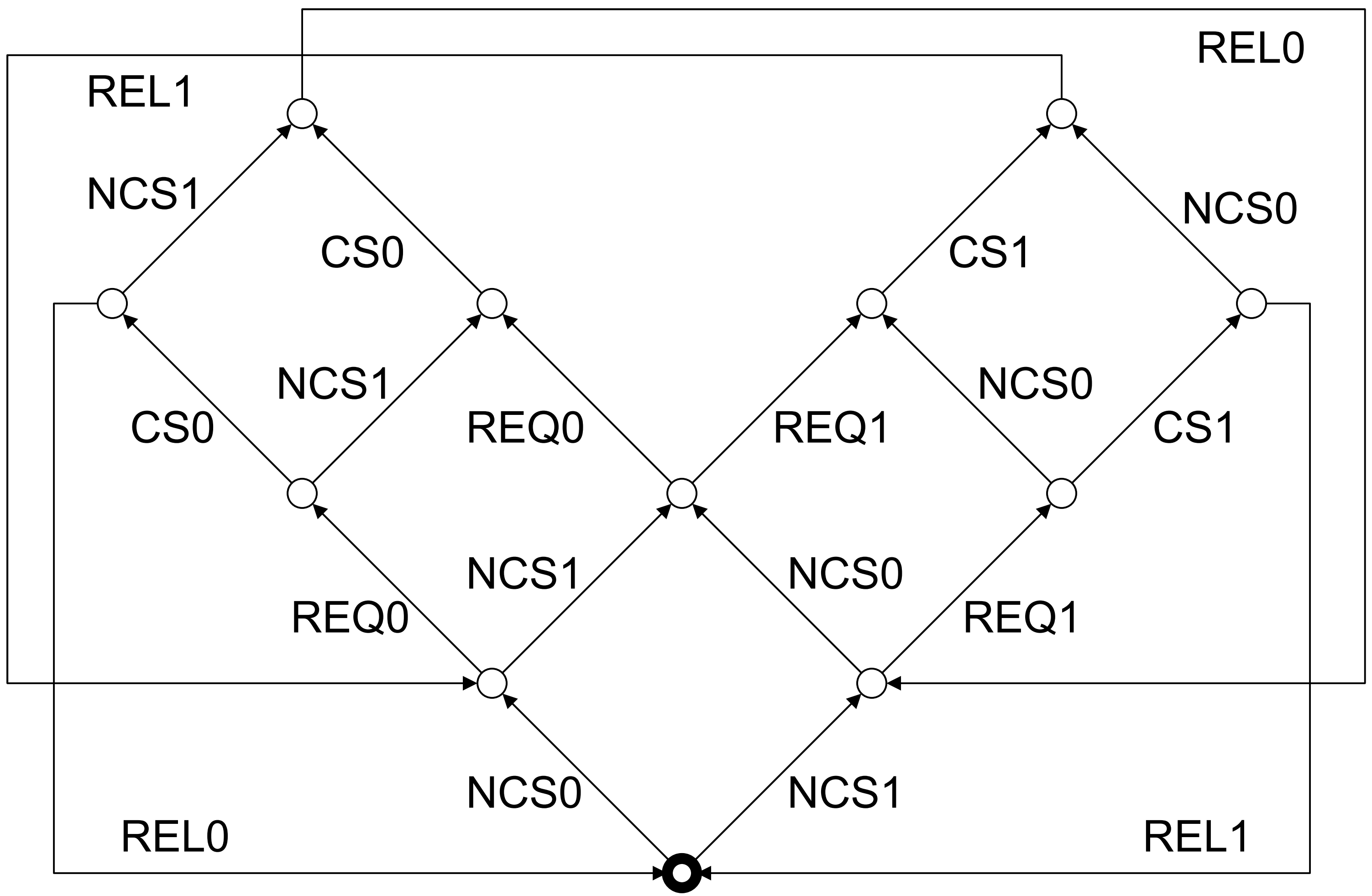
Running example: mutual exclusion with a semaphore





Description using communicating automata

LTS model



REL0
REL1
NCS1
CS0
NCS0
CS1
NCS1
CS0
REQ0
NCS0
REQ1
CS1
NCS1
REQ0
NCS0
REQ1
NCS0
NCS1
REL0
REL1

Modal logics

- They are the simplest logics allowing to reason about the sequencing and branching of transitions in an LTS
- Basic modal operators:
  - *Possibility*

    from a state, there exists (at least) an outgoing transition labeled by a certain action and leading to a certain state
  - *Necessity*

    from a state, all the outgoing transitions labeled by a certain action lead to certain states
- Hennessy-Milner Logic (HML) [Hennessy-Milner-85]

Action predicates
(syntax)

α ::=	a	atomic proposition ($a \in A$)
|	tt	constant "true"
|	ff	constant "false"
|	$\alpha_1 \vee \alpha_2$	disjunction
|	$\alpha_1 \wedge \alpha_2$	conjunction
|	$\neg\alpha_1$	negation
|	$\alpha_1 \Rightarrow \alpha_2$	implication ($\neg\alpha_1 \vee \alpha_2$)
|	$\alpha_1 \Leftrightarrow \alpha_2$	equivalence ($\alpha_1 \Rightarrow \alpha_2 \wedge \alpha_1 \Rightarrow \alpha_2$)

Action predicates
(semantics)

Let $M = (S, A, T, s_0)$. Interpretation $[[\ \alpha\ ]] \subseteq A$:

- $[[\ a\ ]] = \{\ a\ \}$
- $[[\ \mathrm{tt}\ ]] = A$
- $[[\ \mathrm{ff}\ ]] = \varnothing$
- $[[\ \alpha_1 \vee \alpha_2\ ]] = [[\ \alpha_1\ ]] \cup [[\ \alpha_2\ ]]$
- $[[\ \alpha_1 \wedge \alpha_2\ ]] = [[\ \alpha_1\ ]] \cap [[\ \alpha_2\ ]]$
- $[[\ \neg\alpha_1\ ]] = A \setminus [[\ \alpha_1\ ]]$
- $[[\ \alpha_1 \Rightarrow \alpha_2\ ]] = (A \setminus [[\ \alpha_1\ ]]) \cup [[\ \alpha_2\ ]]$
- $[[\ \alpha_1 \Leftrightarrow \alpha_2\ ]] = ((A \setminus [[\ \alpha_1\ ]]) \cup [[\ \alpha_2\ ]])$
  $\cap\ ((A \setminus [[\ \alpha_2\ ]]) \cup [[\ \alpha_1\ ]])$

Examples

A = { NCS_0, NCS_1, CS_0, CS_1, REQ_0, REQ_1, REL_0, REL_1 }

- [[tt]] = { NCS_0, NCS_1, CS_0, CS_1, REQ_0, REQ_1, REL_0, REL_1 }
- [[ff]] = $\varnothing$
- [[NCS_0]] = { NCS_0 }
- [[$\neg NCS_0$]] = { NCS_1, CS_0, CS_1, REQ_0, REQ_1, REL_0, REL_1 }
- [[$NCS_0 \wedge \neg NCS_1$]] = { NCS_0 } = [[NCS_0]]
- [[$NCS_0 \vee NCS_1$]] = { NCS_0, NCS_1 }
- [[$(NCS_0 \vee NCS_1) \wedge (NCS_0 \vee REQ_0)$]] = { NCS_0 }
- [[$NCS_0 \wedge NCS_1$]] = $\varnothing$ = [[ff]]
- [[$NCS_0 \vee \neg NCS_0$]] =
  { NCS_0, NCS_1, CS_0, CS_1, REQ_0, REQ_1, REL_0, REL_1 } = [[tt]]

HML logic

(syntax)

φ ::=		tt	constant "true"
	|	ff	constant "false"
	|	$\varphi_1 \vee \varphi_2$	disjunction
	|	$\varphi_1 \wedge \varphi_2$	conjunction
	|	$\neg\varphi_1$	negation
	|	$\langle \alpha \rangle \varphi_1$	possibility
	|	$[\alpha] \varphi_1$	necessity

- Duality: $[\alpha] \varphi = \neg\langle \alpha \rangle \neg\varphi$

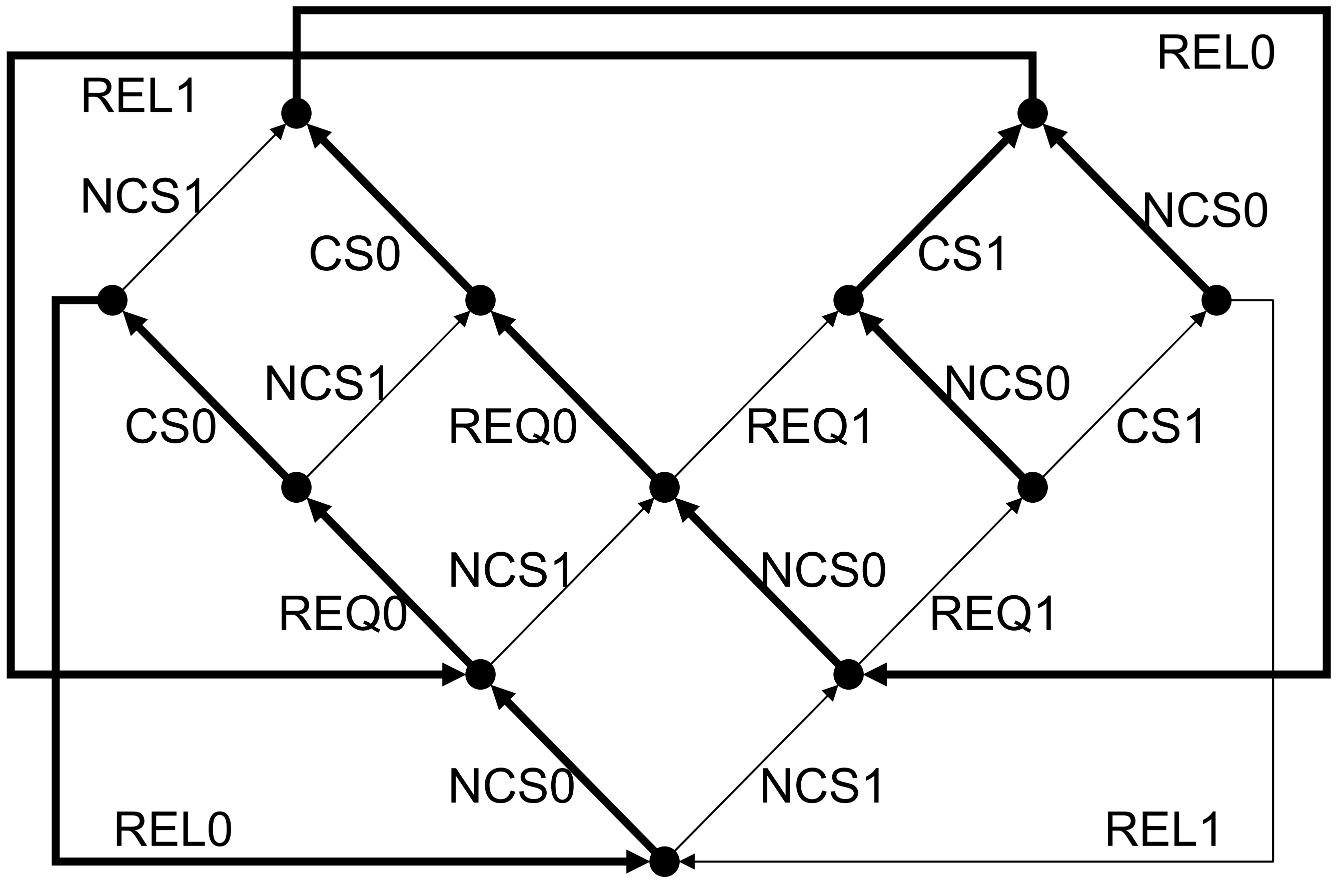
HML logic
(semantics)

Let $M = (S, A, T, s_0)$. Interpretation $[[\ \varphi\ ]] \subseteq S$:

- $[[\ \mathrm{tt}\ ]] = S$
- $[[\ \mathrm{ff}\ ]] = \emptyset$
- $[[\ \varphi_1 \vee \varphi_2\ ]] = [[\ \varphi_1\ ]] \cup [[\ \varphi_2\ ]]$
- $[[\ \varphi_1 \wedge \varphi_2\ ]] = [[\ \varphi_1\ ]] \cap [[\ \varphi_2\ ]]$
- $[[\ \neg\varphi_1\ ]] = S \setminus [[\ \varphi_1\ ]]$
- $[[\ \langle\ \alpha\ \rangle\ \varphi_1\ ]] = \{\ s \in S \mid \exists\ (s, a, s') \in T\ .$
  $a \in [[\ \alpha\ ]] \wedge s' \in [[\ \varphi_1\ ]]\ \}$
- $[[\ [\ \alpha\ ]\ \varphi_1\ ]] = \{\ s \in S \mid \forall\ (s, a, s') \in T\ .$
  $a \in [[\ \alpha\ ]] \Rightarrow s' \in [[\ \varphi_1\ ]]\ \}$

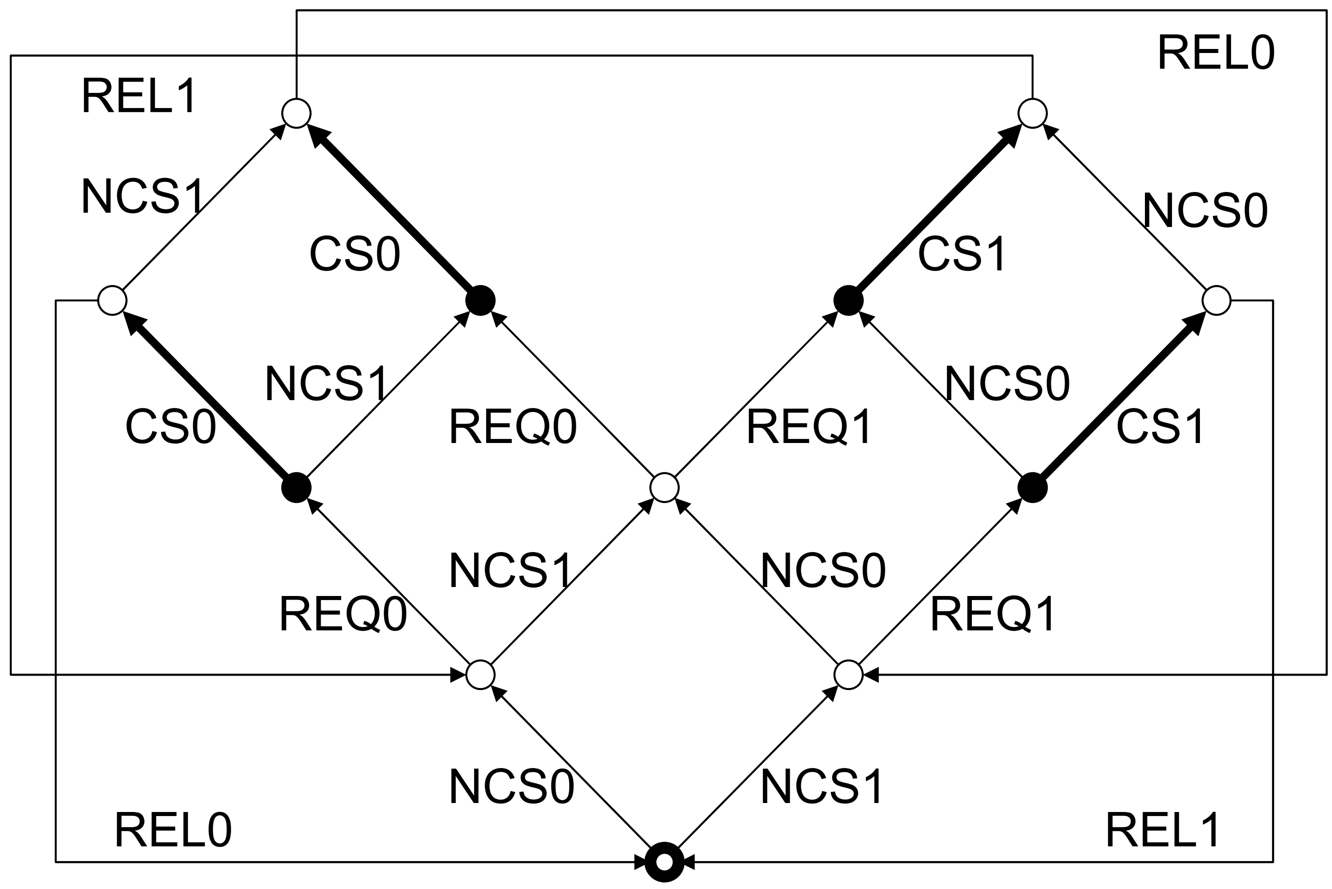
Example (1/4)

Deadlock freedom: $\langle \mathrm{tt} \rangle \mathrm{tt}$



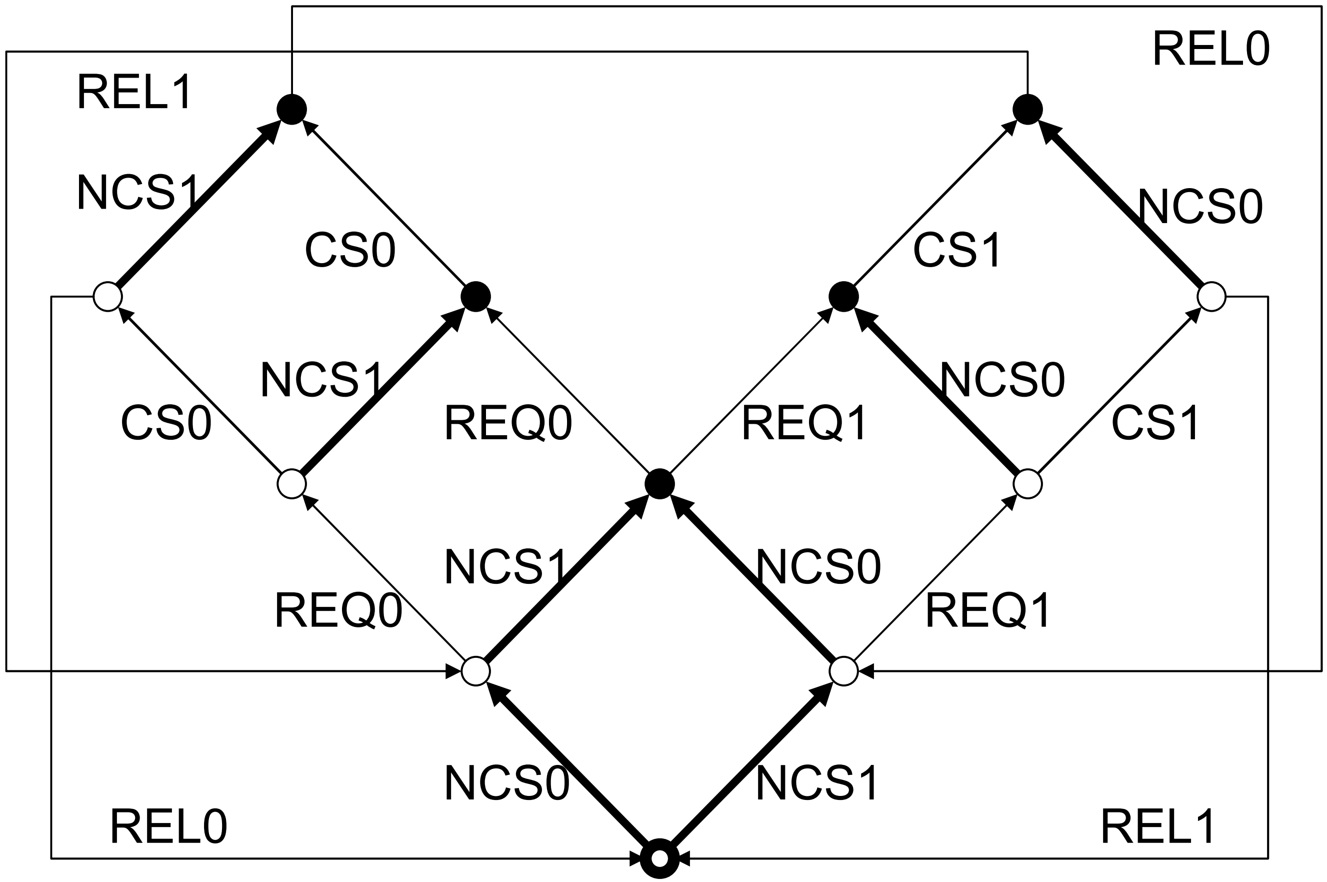
Example (2/4)

Possible execution of a set of actions: $\langle CS_0 \vee CS_1 \rangle tt$



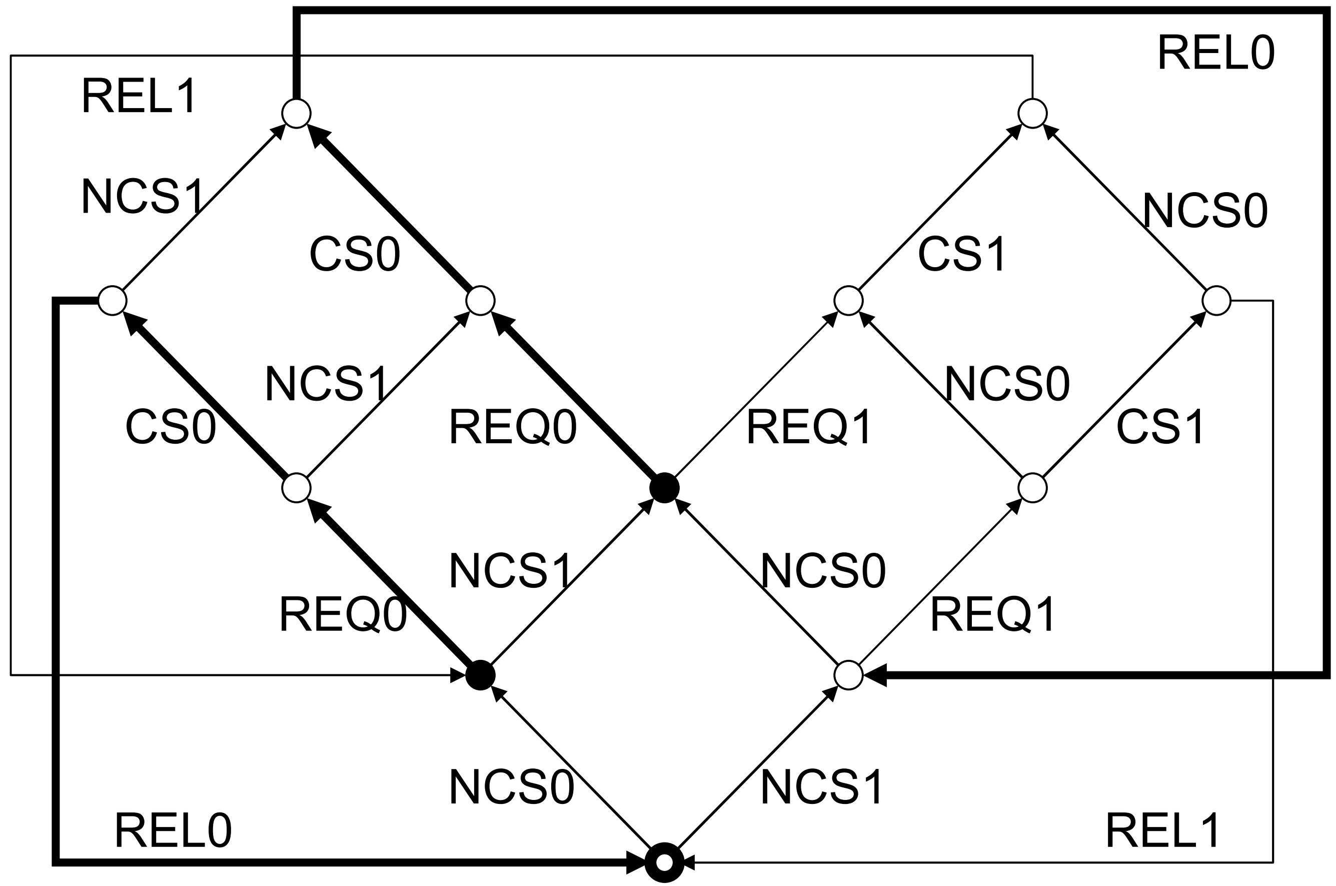
Example (3/4)

Forbidden execution of a set of actions: $[\ NCS_0 \vee NCS_1\ ]\ ff$



Example (4/4)

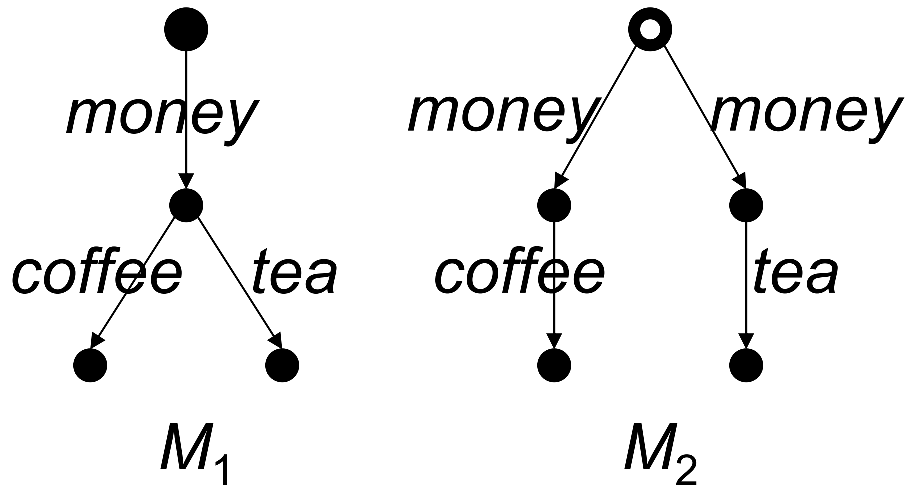
Execution of an action sequence: $\langle\ REQ_0\ \rangle \langle\ CS_0\ \rangle \langle\ REL_0\ \rangle$ tt



Some identities

- Tautologies:
  - $\langle \alpha \rangle ff = \langle ff \rangle \varphi = ff$
  - $[\alpha] tt = [ff] \varphi = tt$
- Distributivity of modalities over $\vee$ and $\wedge$:
  - $\langle \alpha \rangle \varphi_1 \vee \langle \alpha \rangle \varphi_2 = \langle \alpha \rangle (\varphi_1 \vee \varphi_2)$
  - $\langle \alpha_1 \rangle \varphi \vee \langle \alpha_2 \rangle \varphi = \langle \alpha_1 \vee \alpha_2 \rangle \varphi$
  - $[\alpha] \varphi_1 \wedge [\alpha] \varphi_2 = [\alpha] (\varphi_1 \wedge \varphi_2)$
  - $[\alpha_1] \varphi \wedge [\alpha_2] \varphi = [\alpha_1 \vee \alpha_2] \varphi$
- Monotonicity of modalities over φ and α:
  - $(\varphi_1 \Rightarrow \varphi_2) \Rightarrow (\langle \alpha \rangle \varphi_1 \Rightarrow \langle \alpha \rangle \varphi_2) \wedge ([\alpha] \varphi_1 \Rightarrow [\alpha] \varphi_2)$
  - $(\alpha_1 \Rightarrow \alpha_2) \Rightarrow (\langle \alpha_1 \rangle \varphi \Rightarrow \langle \alpha_2 \rangle \varphi) \wedge ([\alpha_2] \varphi \Rightarrow [\alpha_1] \varphi)$

Characterization of branching



- Modal formula distinguishing between M_1 and M_2:

$$\varphi = [\, money \,] \; (\langle coffee \rangle \mathrm{tt} \wedge \langle tea \rangle \mathrm{tt})$$

$$M_1 \models \varphi \quad \text{and} \quad M_2 \not\models \varphi$$

Modal logics
(summary)

- Are able to express simple branching-time properties involving states $s \in S$ and actions $a \in A$ of an LTS
- But:
  – Take into account only a finite number of steps around a state (nesting of modalities)
  – Cannot express properties about transition sequences or subtrees of arbitrary length
- Example: the property

  "from a state *s*, there exists a sequence leading to a state *s'* where the action *a* is executable"

  cannot be expressed in modal logic

  (it would need a formula $\langle \text{tt} \rangle \langle \text{tt} \rangle \ldots \langle \text{tt} \rangle \langle a \rangle \text{tt}$)

Branching-time logics

- They are logics allowing to reason about the (infinite) execution trees contained in an LTS
- Basic temporal operators:
  - *Potentiality*

    from a state, there exists an outgoing, finite transition sequence leading to a certain state
  - *Inevitability*

    from a state, all outgoing transition sequences lead, after a finite number of steps, to certain states
- Action-based Computation Tree Logic (ACTL) [DeNicola-Vaandrager-90]

ACTL logic
(syntax)

φ ::=	tt | ff	boolean constants
|	$\varphi_1 \vee \varphi_2$ | $\neg\varphi_1$	connectors
|	$E [\varphi_{1\alpha 1} \, U \, \varphi_2]$	potentiality 1
|	$E [\varphi_{1\alpha 1} \, U_{\alpha 2} \, \varphi_2]$	potentiality 2
|	$A [\varphi_{1\alpha 1} \, U \, \varphi_2]$	inevitability 1
|	$A [\varphi_{1\alpha 1} \, U_{\alpha 2} \, \varphi_2]$	inevitability 2

ACTL logic
(derived operators)

- $EF_\alpha\ \varphi = E\ [\ tt_\alpha\ U\ \varphi\ ]$ basic potentiality
- $AF_\alpha\ \varphi = A\ [\ tt_\alpha\ U\ \varphi\ ]$ basic inevitability

- $AG_\alpha\ \varphi = \neg\ EF_\alpha\ \neg\varphi$ invariance
- $EG_\alpha\ \varphi = \neg\ AF_\alpha\ \neg\varphi$ trajectory

- $\langle\ \alpha\ \rangle\ \varphi = E\ [\ tt_{ff}\ U_\alpha\ \varphi\ ]$ possibility
- $[\ \alpha\ ]\ \varphi = \neg\ \langle\ \alpha\ \rangle\ \neg\ \varphi$ necessity

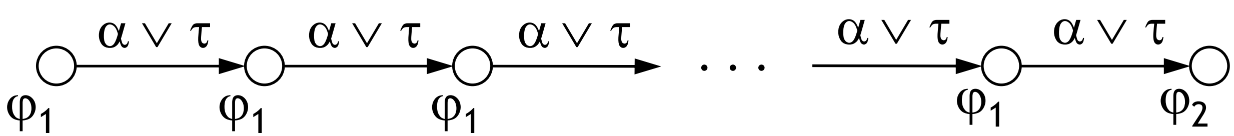
dualities

ACTL logic

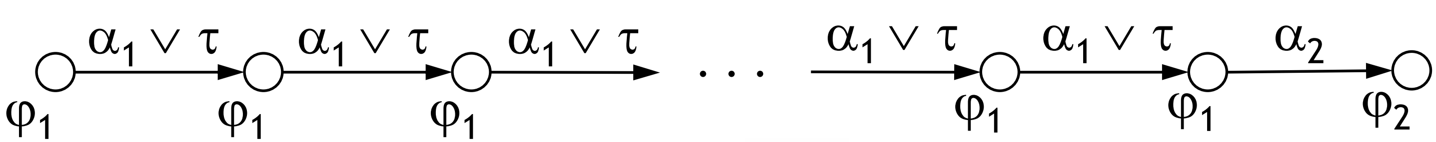
(semantics – potentiality operators)

Let $M = (S, A, T, s_0)$. Interpretation $[[\ \varphi\ ]] \subseteq S$:

- $[[\ E\ [\ \varphi_{1\alpha}\ U\ \varphi_2\ ]\ ]] = \{\ s \in S \mid \exists s(=s_0) \to^{a0} s_1 \to^{a1} s_2 \to \ldots\ .$ $\exists k \geq 0.\ \forall 0 \leq i < k.\ (s_i \in [[\ \varphi_1\ ]] \wedge a_i \in [[\ \alpha \vee \tau\ ]]) \wedge$ $s_k \in [[\ \varphi_2\ ]]\ \}$

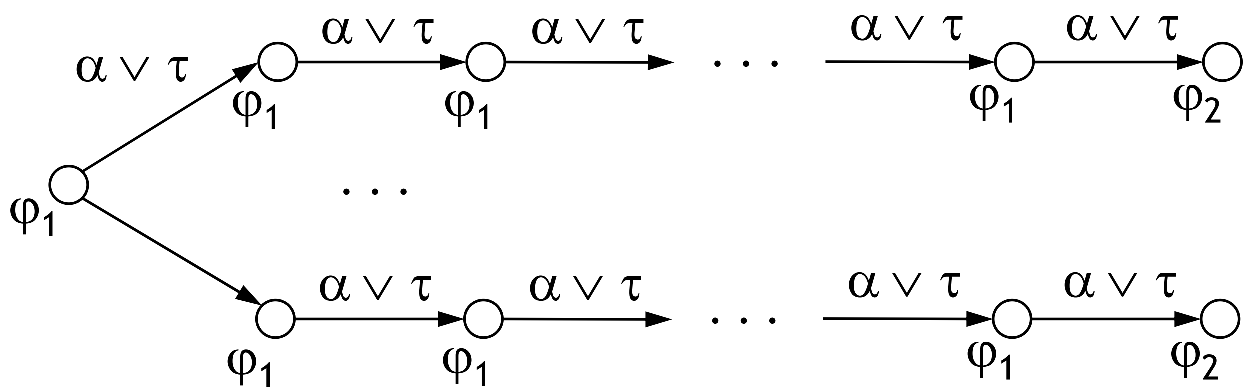


- $[[\ E\ [\ \varphi_{1\alpha 1}\ U_{\alpha 2}\ \varphi_2\ ]\ ]] = \{\ s \in S \mid \forall s(=s_0) \to^{a0} s_1 \to^{a1} s_2 \to \ldots\ .$ $\exists k \geq 0.\ \forall 0 \leq i < k.\ (s_i \in [[\ \varphi_1\ ]] \wedge a_i \in [[\ \alpha_1 \vee \tau\ ]] \wedge$ $s_k \in [[\ \varphi_1\ ]] \wedge a_k \in [[\ \alpha_2\ ]] \wedge s_{k+1} \in [[\ \varphi_2\ ]]\ \}$

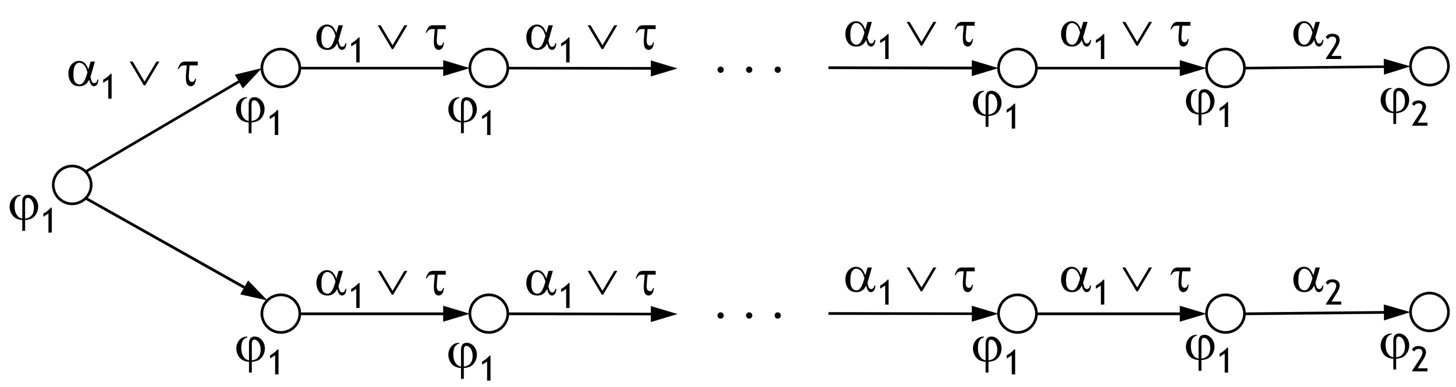


ACTL logic
(semantics – inevitability operators)

- $[[\ A\ [\ \varphi_{1\alpha}\ U\ \varphi_2\ ]\ ]]$:

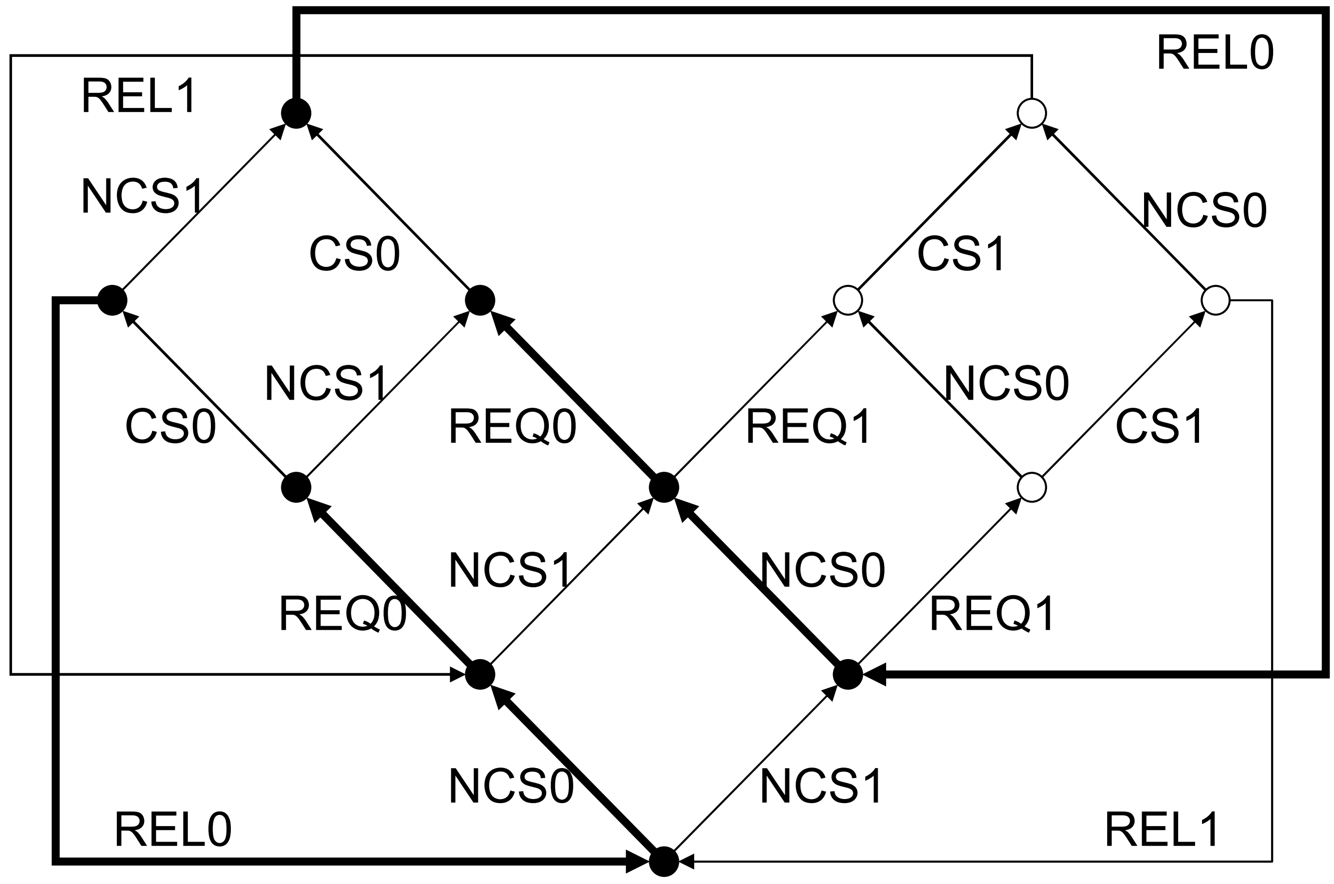


- $[[\ A\ [\ \varphi_{1\alpha 1}\ U_{\alpha 2}\ \varphi_2\ ]\ ]]$:



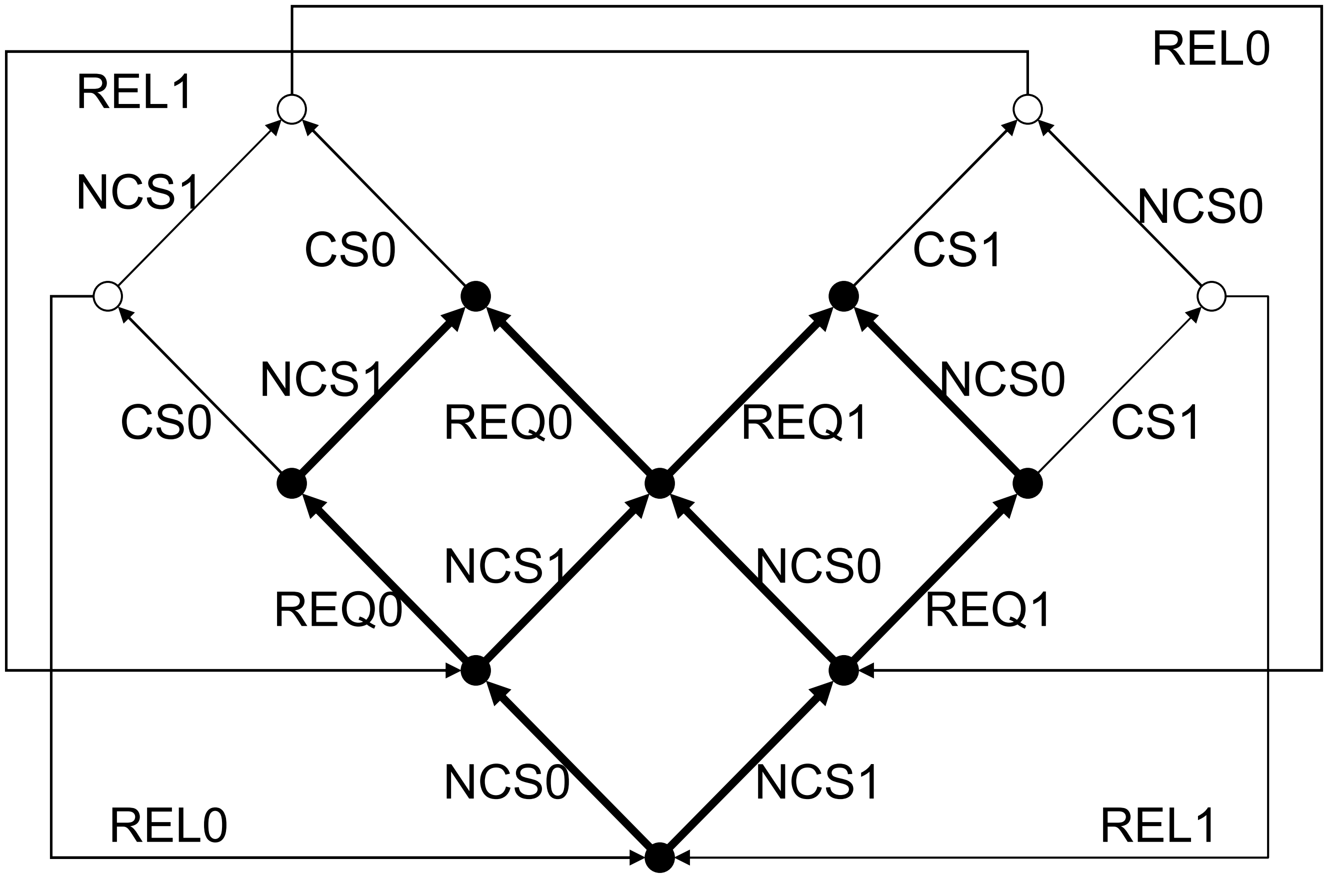
Example (1/4)

Potential reachability: $EF_{\neg REL1} \langle CS_0 \rangle tt$



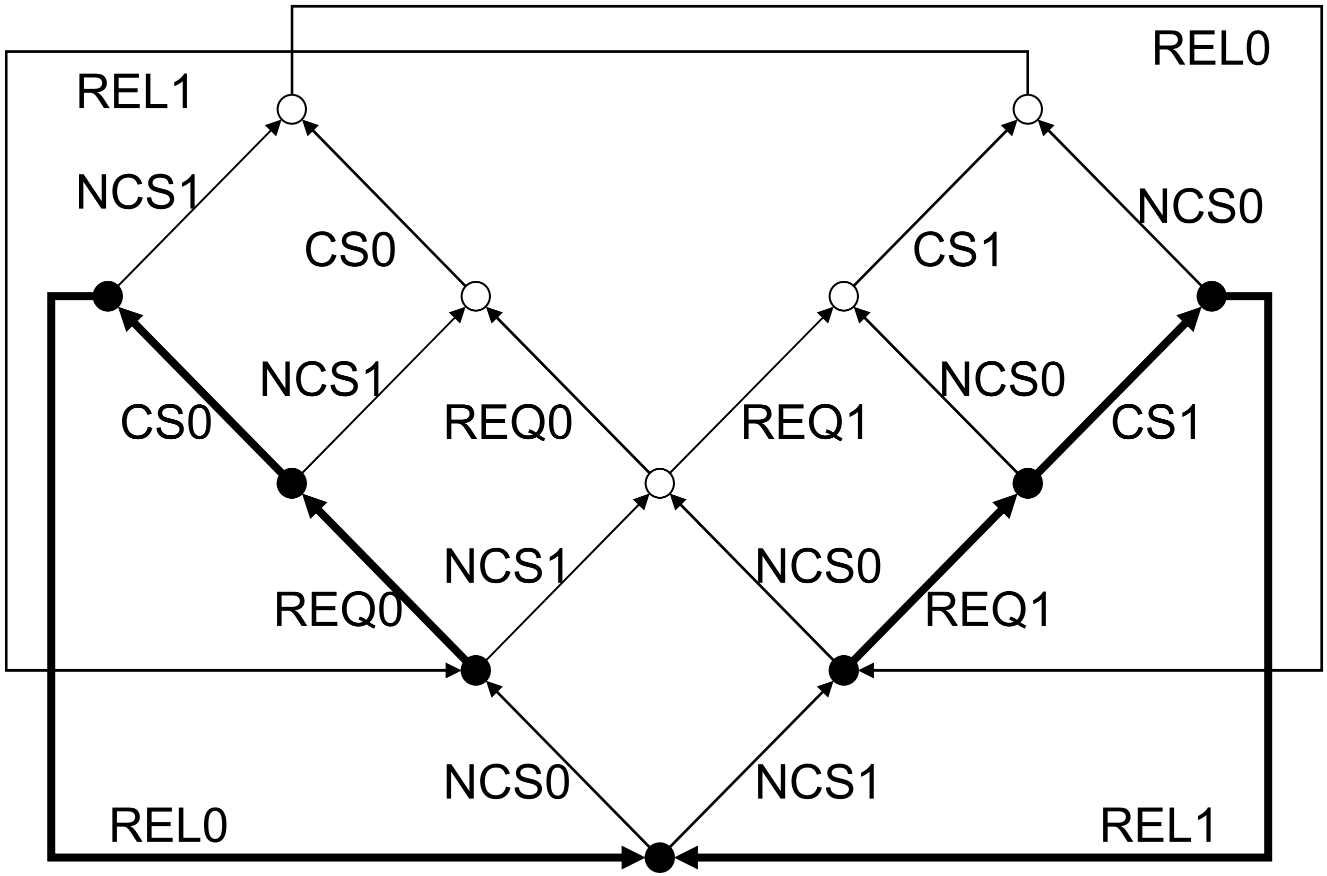
Example (2/4)

Inevitable reachability: $AF_{\neg\,(REL0 \vee REL1)} \langle CS_0 \vee CS_1 \rangle tt$



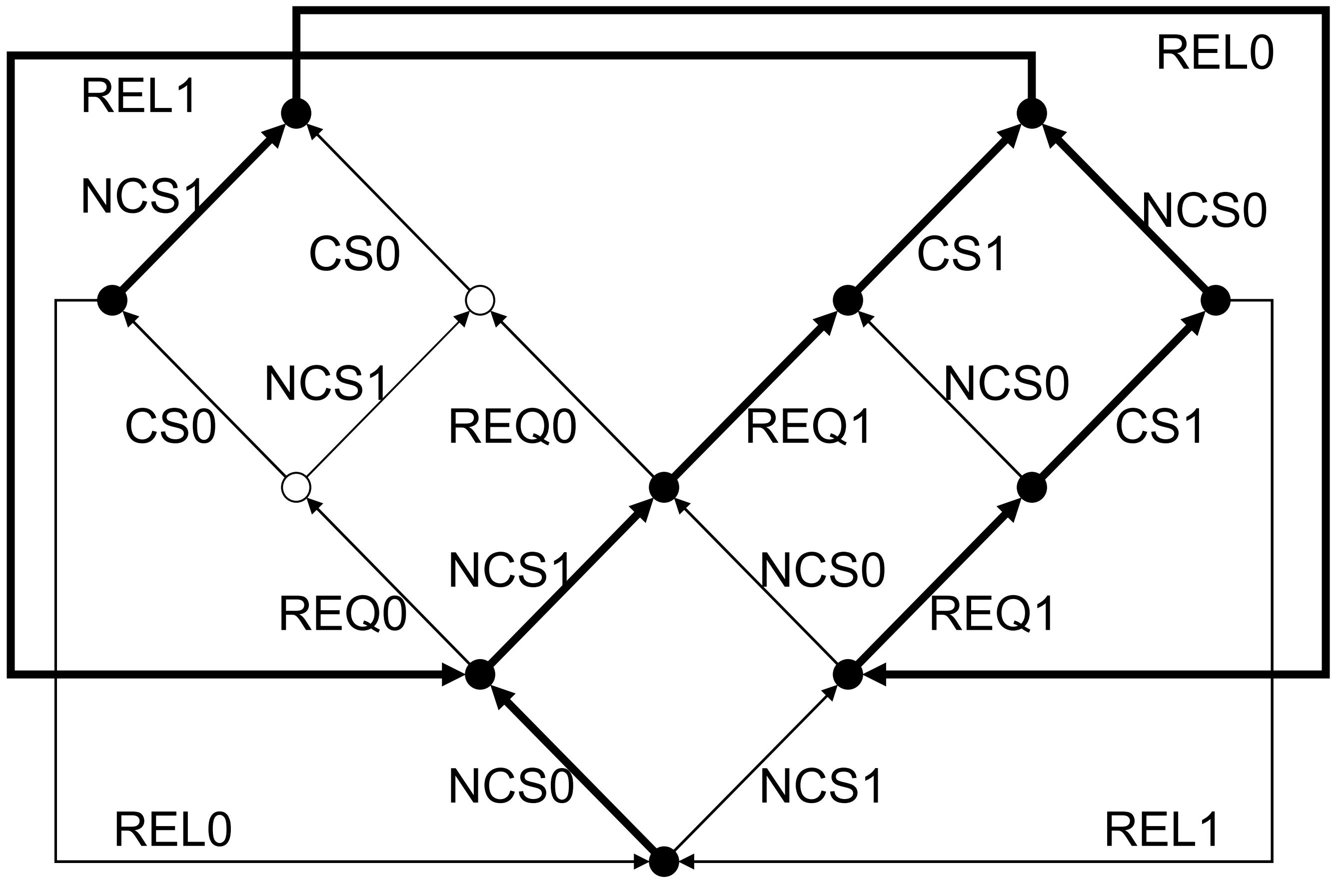
Example (3/4)

Invariance: $AG_{\neg (NCS0 \vee NCS1)} \langle NCS_0 \vee NCS_1 \rangle tt$



Example (4/4)

Trajectory: $EG_{\neg CS0}\,[\,CS_0\,]\,ff$

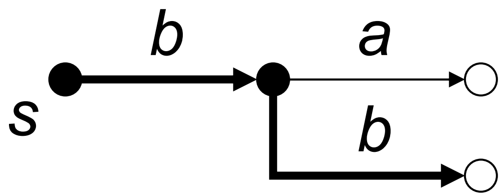


Remark about inevitability

- *Inevitable reachability*: all sequences going out of a state lead to states where an action *a* is executable

  $AF_{tt} \langle a \rangle tt$

- *Inevitable execution*: all sequences going out of a state contain the action *a*
- Inevitable execution $\Rightarrow$ inevitable reachability but the converse does not hold:



$s \models AF_{tt} \langle a \rangle tt$

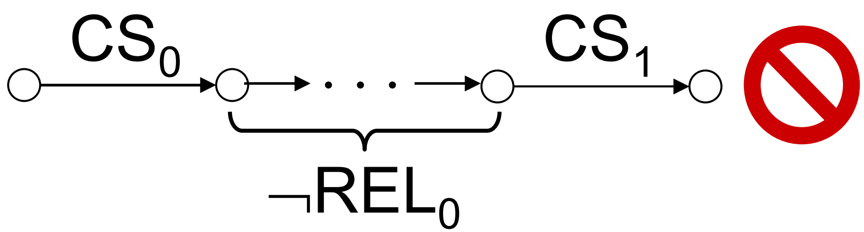
- Inevitable execution must be expressed using the inevitability operators of ACTL:

$$s \not\models A [tt_{tt} U_a tt]$$

Safety properties

- Informally, safety properties specify that
  "something bad never happens"
  during the execution of the system
- One way of expressing safety properties:
  forbid undesirable execution sequences
  - Mutual exclusion:

$\neg \langle CS_0 \rangle EF_{\neg REL0} \langle CS_1 \rangle tt$
$= [CS_0] AG_{\neg REL0} [CS_1] ff$



- In ACTL, forbidding a sequence is expressed by combining the $[\alpha] \varphi$ and $AG_\alpha \varphi$ operators

Liveness properties

- Informally liveness properties specify that
  "something good eventually happens"
  during the execution of the system
- One way of expressing liveness properties:
  require desirable execution sequences / trees
  - Potential release of the critical section:
    $\langle NCS_0 \rangle EF_{tt} \langle REQ_0 \rangle EF_{tt} \langle REL_0 \rangle tt$
  - Inevitable access to the critical section:
    $A [tt_{tt} U_{CS0} tt]$
- In ACTL, the existence of a sequence is expressed by combining the $\langle \alpha \rangle \varphi$ and $EF_{\alpha} \varphi$ operators

Branching-time logics
(summary)

- The temporal operators of ACTL: strictly more powerful than the HML modalities $\langle \alpha \rangle \varphi$ and $[\alpha] \varphi$
- They allow to express branching-time properties on an unbounded depth in an LTS
- But:
  - They do not allow to express the unbounded repetition of a subsequence
- Example: the property

  "from a state *s*, there exists a sequence *a.b.a.b* ... *a.b* leading to a state *s*' where an action *c* is executable"

  cannot be expressed in ACTL

Regular logics

- They allow to reason about the regular execution sequences of an LTS
- Basic operators:
  - *Regular formulas*

    two states are linked by a sequence whose concatenated actions form a word of a regular language
  - *Modalities on sequences*

    from a state, some (all) outgoing regular transition sequences lead to certain states
- Propositional Dynamic Logic (PDL) [Fischer-Ladner-79]

Regular formulas
(syntax)

β ::=	α	one-step sequence
|	nil	empty sequence
|	$\beta_1 \, . \, \beta_2$	concatenation
|	$\beta_1 \mid \beta_2$	choice
|	$\beta_1{}^*$	iteration ($\geq$ 0 times)
|	$\beta_1{}^+$	iteration ($\geq$ 1 times)

- Some identities:

  nil = ff * $\beta^+ = \beta \, . \, \beta^*$

Regular formulas

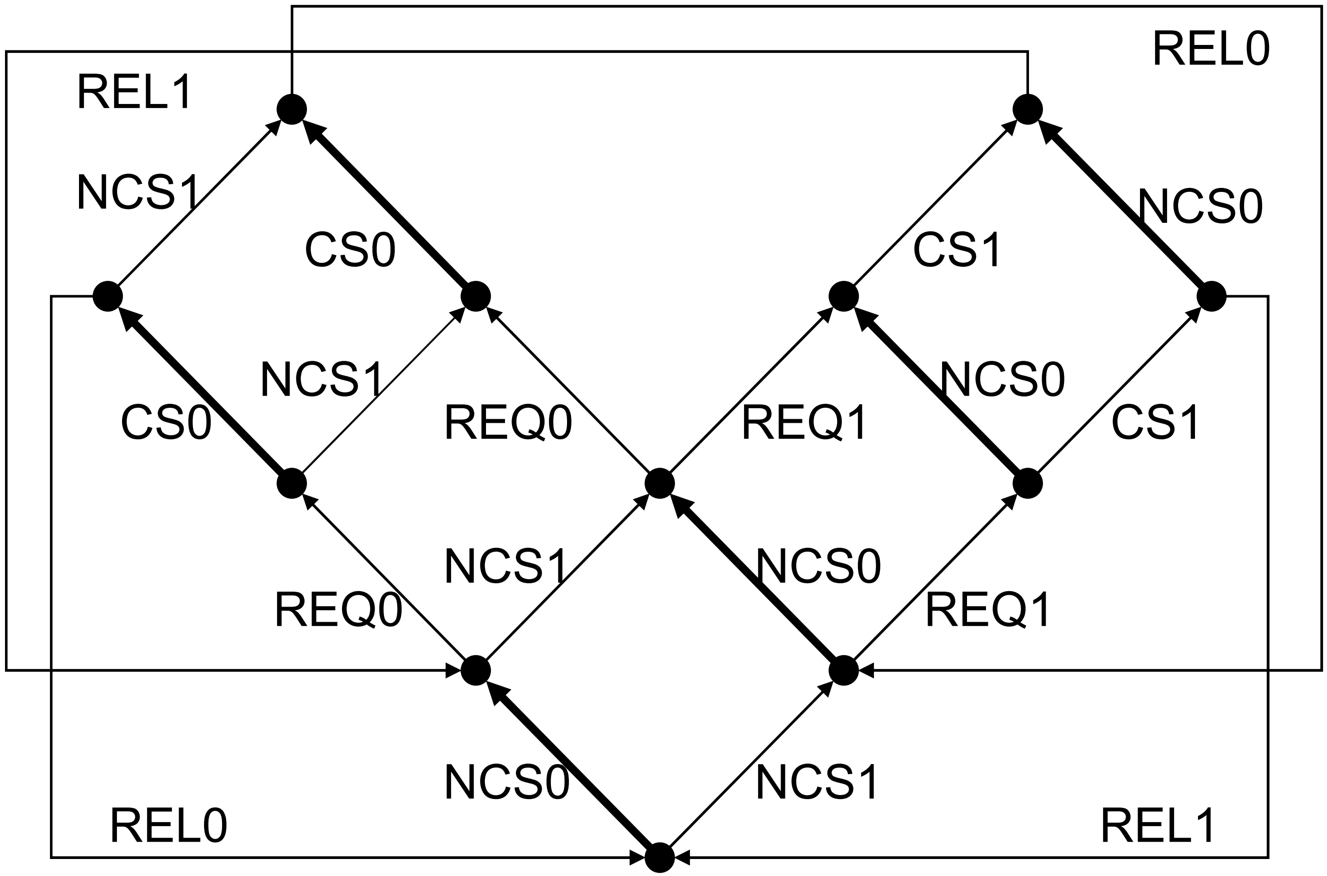
(semantics)

Let $M = (S, A, T, s_0)$. Interpretation $[[\ \beta\ ]] \subseteq S \times S$:

- $[[\ \alpha\ ]] = \{\ (s, s') \mid \exists a \in [[\ \alpha\ ]]\ .\ (s, a, s') \in T\ \}$
- $[[\ \text{nil}\ ]] = \{\ (s, s) \mid s \in S\ \}$ (identity)
- $[[\ \beta_1\ .\ \beta_2\ ]] = [[\ \beta_1\ ]] \circ [[\ \beta_2\ ]]$ (composition)
- $[[\ \beta_1 \mid \beta_2\ ]] = [[\ \beta_1\ ]] \cup [[\ \beta_2\ ]]$ (union)
- $[[\ \beta_1^*\ ]] = [[\ \beta_1\ ]]^*$ (transitive refl. closure)
- $[[\ \beta_1^+\ ]] = [[\ \beta_1\ ]]^+$ (transitive closure)

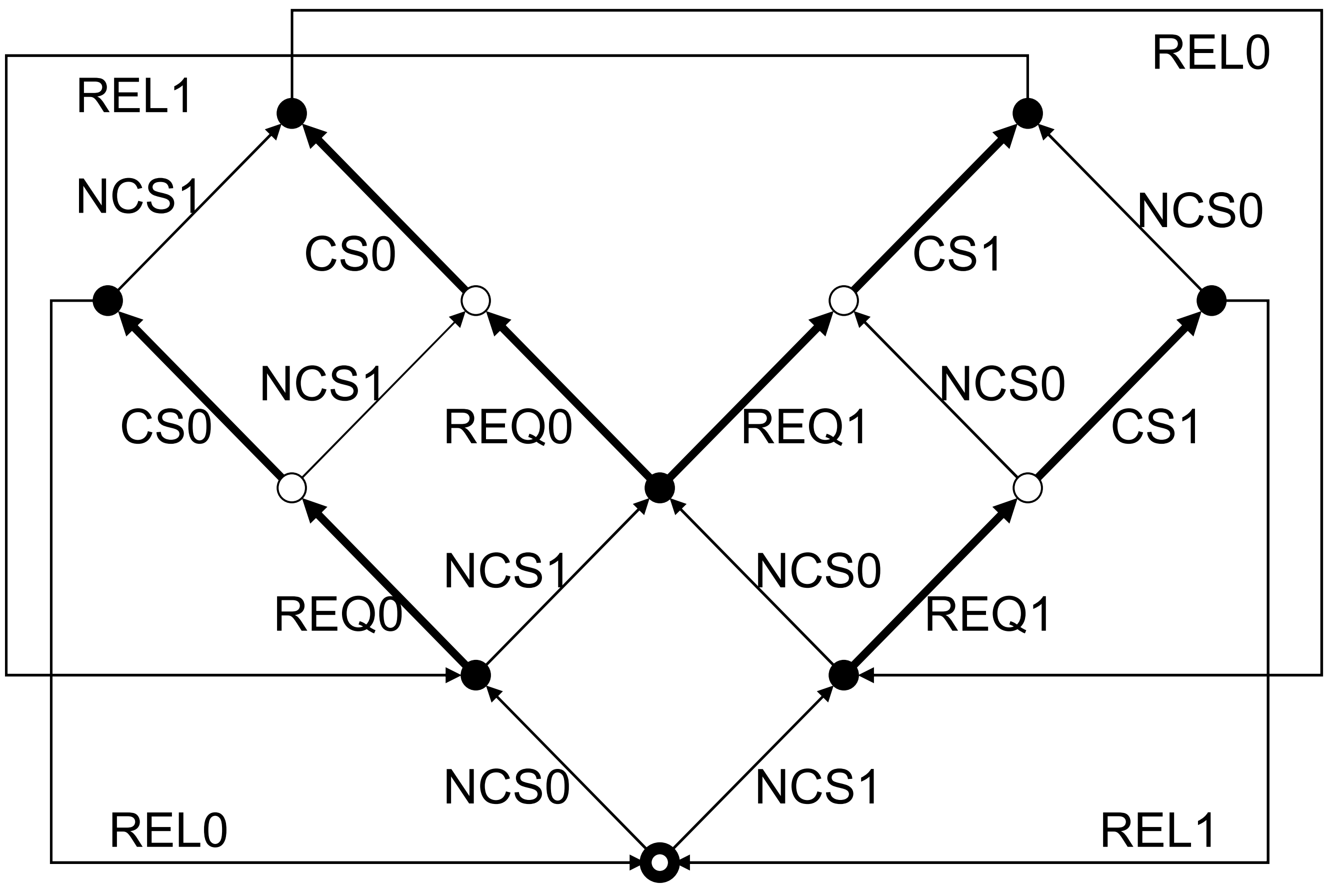
Example (1/3)

One-step sequences: $NCS_0 \vee CS_0$



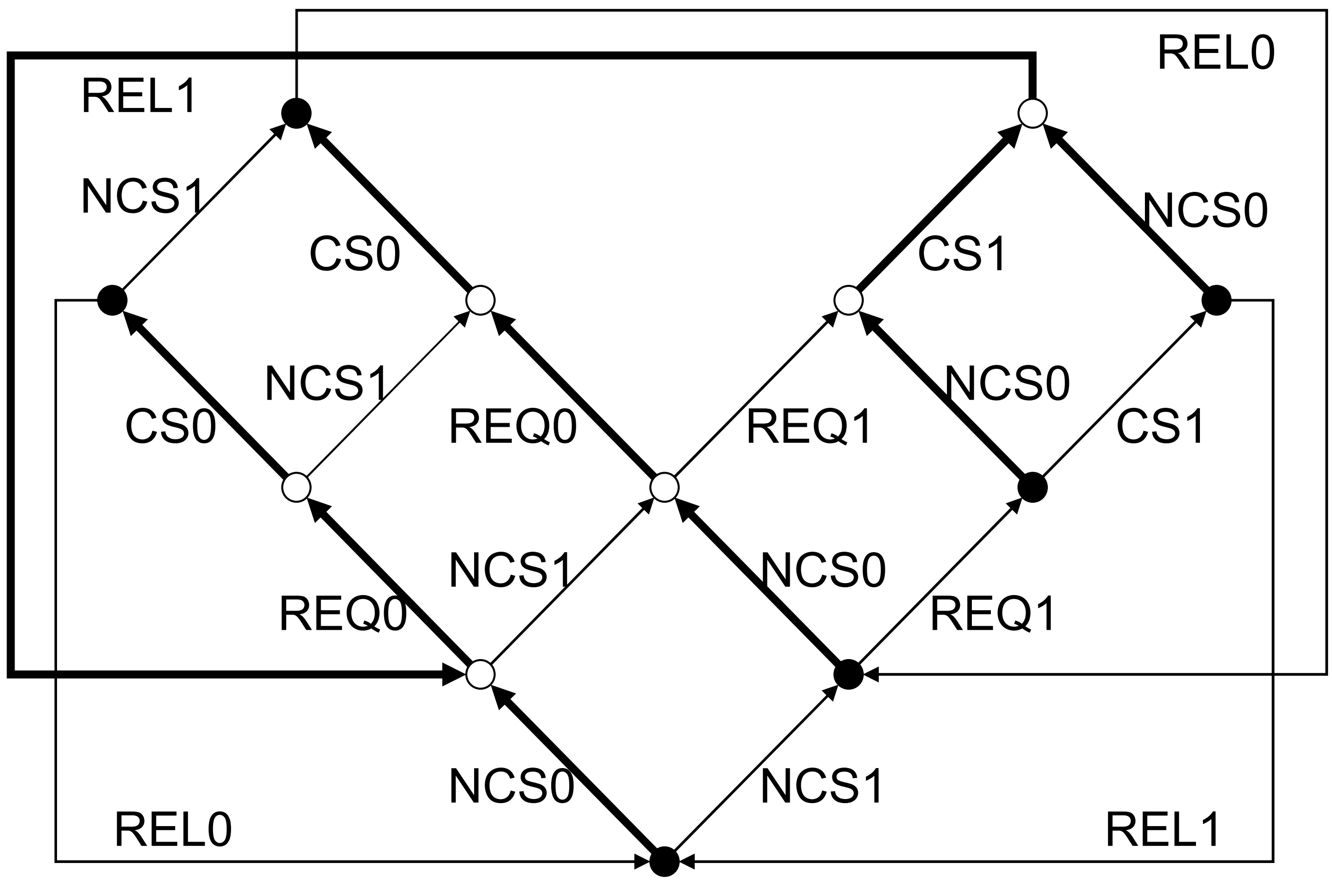
Example (2/3)

Alternative sequences: $(REQ_0 \,.\, CS_0) \mid (REQ_1 \,.\, CS_1)$



Example (3/3)

Sequences with repetition: NCS_0 . $(\neg NCS_1)^*$. CS_0



PDL logic
(syntax)

$\varphi ::=$	tt | ff	boolean constants
|	$\varphi_1 \vee \varphi_2$	disjunction
|	$\varphi_1 \wedge \varphi_2$	conjunction
|	$\neg \varphi_1$	negation
|	$\langle \beta \rangle \varphi_1$	possibility
|	$[\beta] \varphi_1$	necessity

- Duality: $[\beta]\varphi = \neg \langle \beta \rangle \neg \varphi$

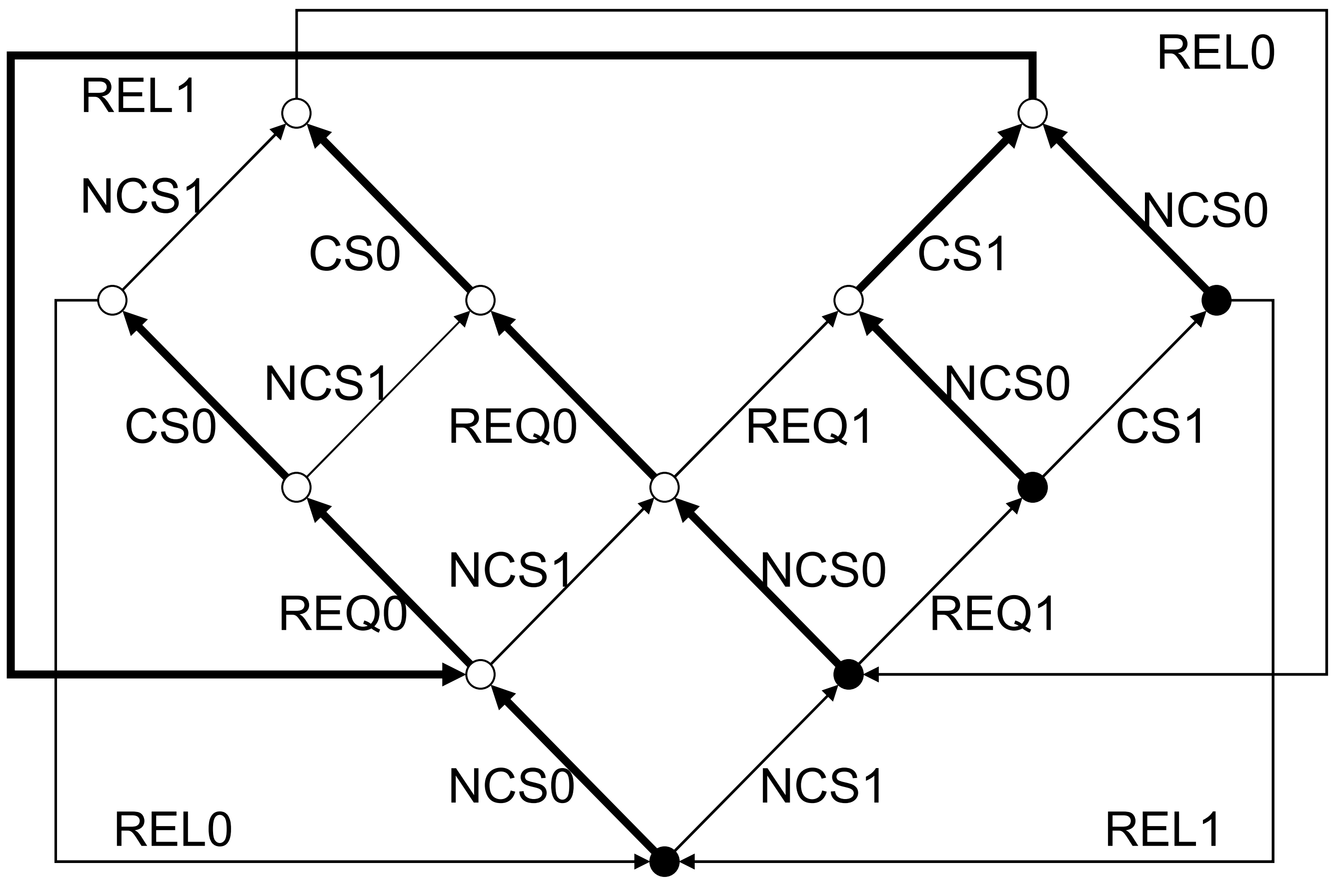
PDL logic
(semantics)

Let $M = (S, A, T, s_0)$. Interpretation $[[\ \varphi\ ]] \subseteq S$:

- $[[\ tt\ ]] = S$
- $[[\ ff\ ]] = \varnothing$
- $[[\ \varphi_1 \vee \varphi_2\ ]] = [[\ \varphi_1\ ]] \cup [[\ \varphi_2\ ]]$
- $[[\ \varphi_1 \wedge \varphi_2\ ]] = [[\ \varphi_1\ ]] \cap [[\ \varphi_2\ ]]$
- $[[\ \neg\varphi_1\ ]] = S \setminus [[\ \varphi_1\ ]]$
- $[[\ \langle\ \beta\ \rangle\ \varphi_1\ ]] = \{\ s \in S \mid \exists\ s' \in S\ .\ (s, s') \in [[\ \beta\ ]] \wedge s' \in [[\ \varphi_1\ ]]\ \}$
- $[[\ [\ \beta\ ]\ \varphi_1\ ]] = \{\ s \in S \mid \forall\ s' \in S\ .\ (s, s') \in [[\ \beta\ ]] \Rightarrow s' \in [[\ \varphi_1\ ]]\ \}$

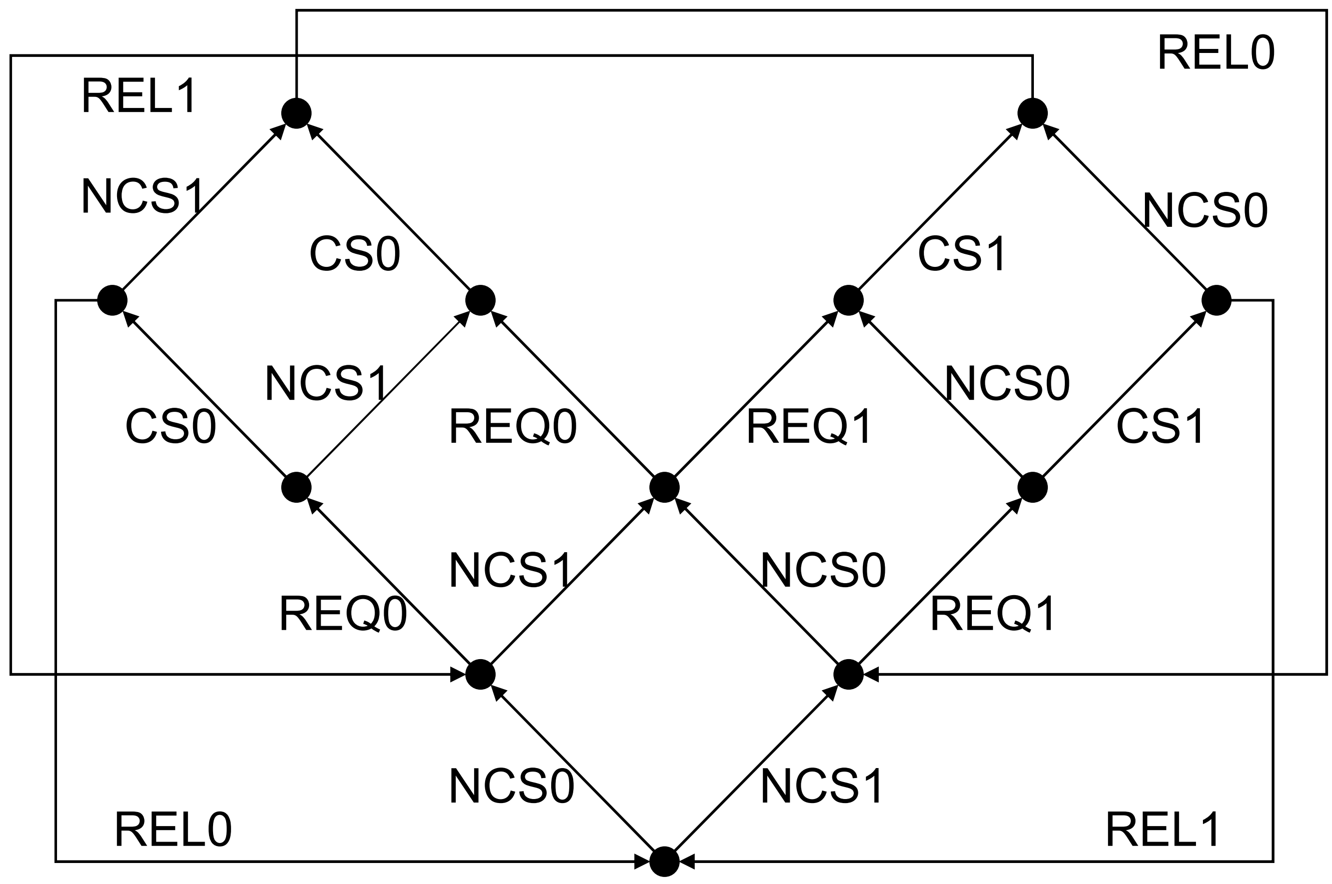
Example (1/2)

Potential reachability of critical section: $\langle\ NCS_0\ .\ tt\ ^*\ .\ CS_0\ \rangle\ tt$



Example (2/2)

Mutual exclusion: $[\ CS_0\ .\ (\neg REL_0)^*\ .\ CS_1\ ]\ ff$

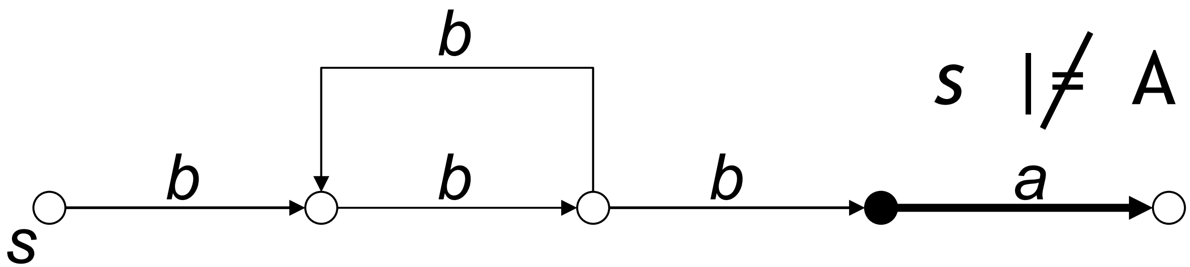


Some identities

- Distributivity of regular operators over $\langle\ \rangle$ and [ ]:
  - $\langle \beta_1 . \beta_2 \rangle \varphi = \langle \beta_1 \rangle \langle \beta_2 \rangle \varphi$
  - $\langle \beta_1 \mid \beta_2 \rangle \varphi = \langle \beta_1 \rangle \varphi \vee \langle \beta_2 \rangle \varphi$
  - $\langle \beta^* \rangle \varphi = \varphi \vee \langle \beta \rangle \langle \beta^* \rangle \varphi$
  - $[\beta_1 . \beta_2] \varphi = [\beta_1] [\beta_2] \varphi$
  - $[\beta_1 \mid \beta_2] \varphi = [\beta_1] \varphi \wedge [\beta_2] \varphi$
  - $[\beta^*] \varphi = \varphi \wedge [\beta] [\beta^*] \varphi$
- Potentiality and invariance operators of ACTL:
  - $EF_\alpha \varphi = \langle \alpha^* \rangle \varphi$
  - $AG_\alpha \varphi = [\alpha^*] \varphi$

Fairness properties

- Problem: from the initial state of the LTS, there is no inevitable execution of action $CS_0 \Rightarrow$ process P_1 can enter its critical section indefinitely often



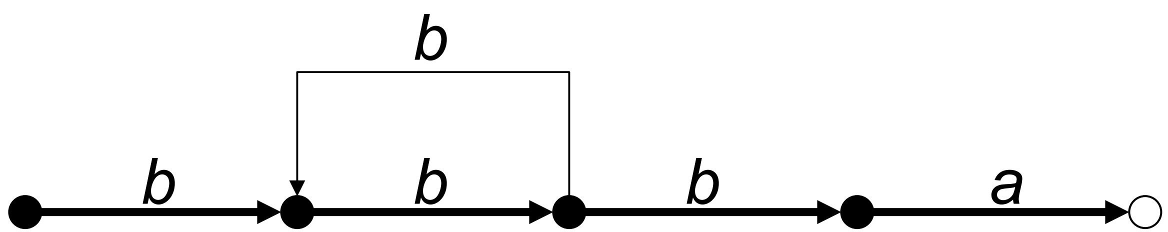
$s \not\models A [tt_{tt} \; U_a \; tt]$

- *Fair execution* of an action *a*: from a state, all transition sequences that do not cycle indefinitely contain action *a*
- Action-based counterpart of the *fair reachability of predicates* [Queille-Sifakis-82]

Fair execution

- Fair execution of an action *a* expressed in PDL:

$$\text{fair}(a) = [\,(\neg a)^*\,]\,\langle\, tt^*.\ a\,\rangle\, tt$$

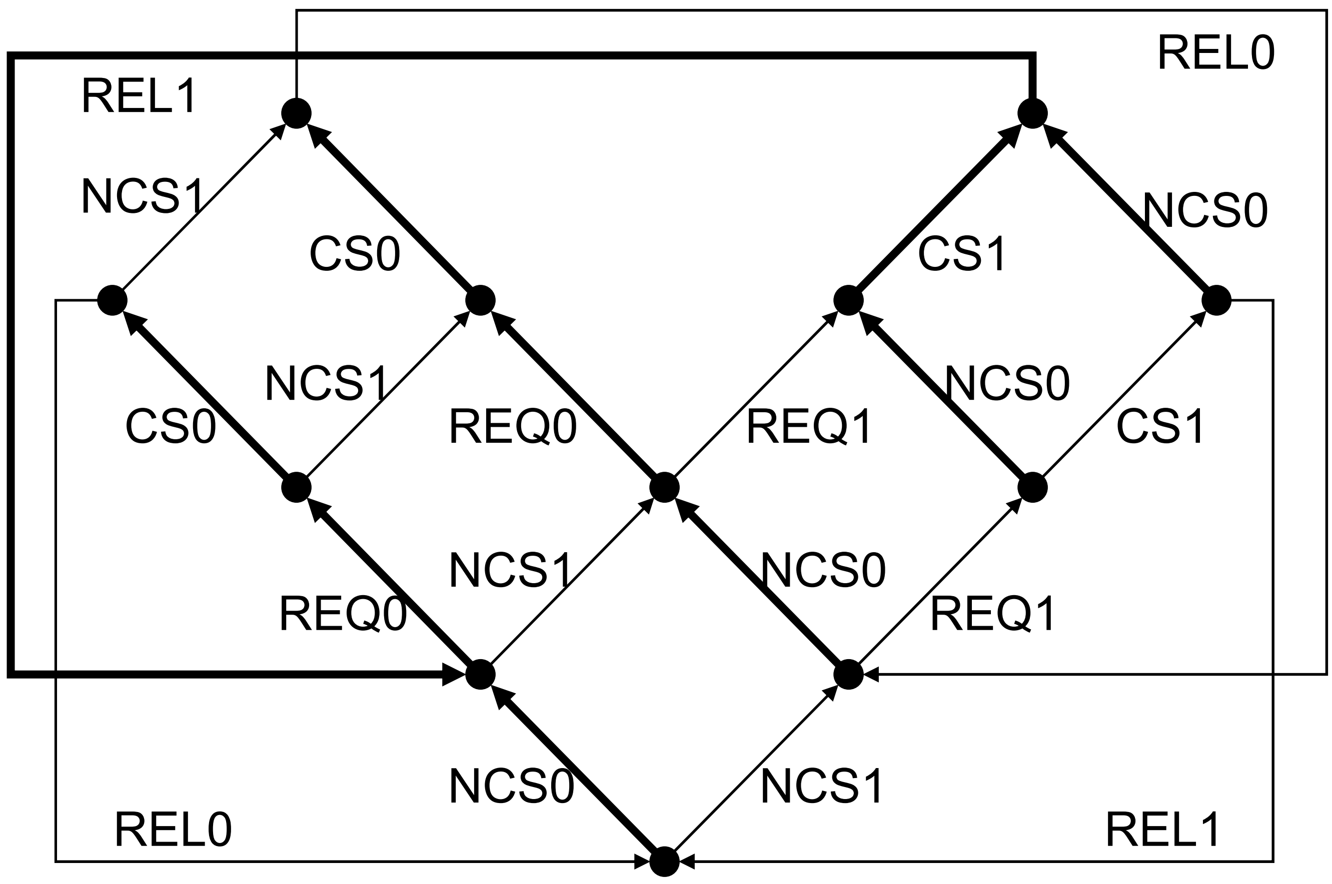


- Equivalent formulation in ACTL:

$$\text{fair}(a) = AG_{\neg a}\, EF_{tt}\,\langle\, a\,\rangle\, tt$$

Example

Fair execution of critical section: $[\ (\neg CS_0)^*\ ]\ \langle\ tt^*.\ CS_0\ \rangle\ tt$



Regular logics
(summary)

- They allow a direct and natural description of regular execution sequences in LTSs

- More intuitive description of safety properties:
  - Mutual exclusion:

    $[\ CS_0\ ]\ AG_{\neg REL0}\ [\ CS_1\ ]\ ff \quad =$ (in ACTL)

    $[\ CS_0\ .\ (\neg REL_0)^*\ .\ CS_1\ ]\ ff$ (in PDL)

- But:
  - Not sufficiently powerful to express inevitability operators (expressiveness uncomparable with branching-time logics)

Fixed point logics

- Very expressive logics ("temporal logic assembly languages") allowing to characterize finite or infinite tree-like patterns in LTSs
- Basic temporal operators:
  - *Minimal fixed point* (μ)

    "recursive function" defined over the LTS: *finite* execution trees going out of a state
  - *Maximal fixed point* (ν)

    dual of the minimal fixed point operator: *infinite* execution trees going out of a state
- Modal mu-calculus [Kozen-83,Stirling-01]

Modal mu-calculus

(syntax)

φ ::=	tt | ff	boolean constants
|	$\varphi_1 \vee \varphi_2$ | $\neg\varphi_1$	connectors
|	$\langle\, \alpha \,\rangle\, \varphi_1$	possibility
|	$[\, \alpha \,]\, \varphi_1$	necessity
|	X	propositional variable
|	$\mu X \,.\, \varphi_1$	minimal fixed point
|	$\nu X \,.\, \varphi_1$	maximal fixed point

- Duality: $\nu X \,.\, \varphi = \neg\, \mu X \,.\, \neg\, \varphi\, [\neg\, X \,/\, X\,]$

Syntactic restrictions

- Syntactic monotonicity [Kozen-83]
  - Necessary to ensure the existence of fixed points
  - In every formula $\sigma X \,.\, \varphi\,(X)$, where $\sigma \in \{\, \mu, \nu \,\}$, every free occurrence of X in φ falls in the scope of an even number of negations

$$\mu X \,.\, \langle\, a \,\rangle\, X \vee \neg \langle\, b \,\rangle\, X$$

- Alternation depth 1 [Emerson-Lei-86]
  - Necessary for efficient (linear-time) verification
  - In every formula $\mu X \,.\, \varphi\,(X)$, every maximal subformula $\nu Y \,.\, \varphi'\,(Y)$ of φ is closed

$$\mu X \,.\, \langle\, a \,\rangle\, \nu Y \,.\, ([\, b \,]\, Y \wedge [\, c \,]\, X)$$

Modal mu-calculus
(semantics)

Let $M = (S, A, T, s_0)$ and $\rho : \mathcal{X} \rightarrow 2^S$ a context mapping propositional variables to state sets. Interpretation $[[\ \varphi\ ]] \subseteq S$:

- $[[\ X\ ]]\ \rho = \rho\ (X\ )$
- $[[\ \mu X\ .\ \varphi\ ]]\ \rho = \bigcup_{k \geq 0} \Phi_\rho{}^k\ (\varnothing)$
- $[[\ \nu X\ .\ \varphi\ ]]\ \rho = \bigcap_{k \geq 0} \Phi_\rho{}^k\ (S)$

where $\Phi_\rho : 2^S \rightarrow 2^S$,

$$\Phi_\rho\ (U) = [[\ \varphi\ ]]\ \rho\ [\ U\ /\ X\ ]$$

Minimal fixed point

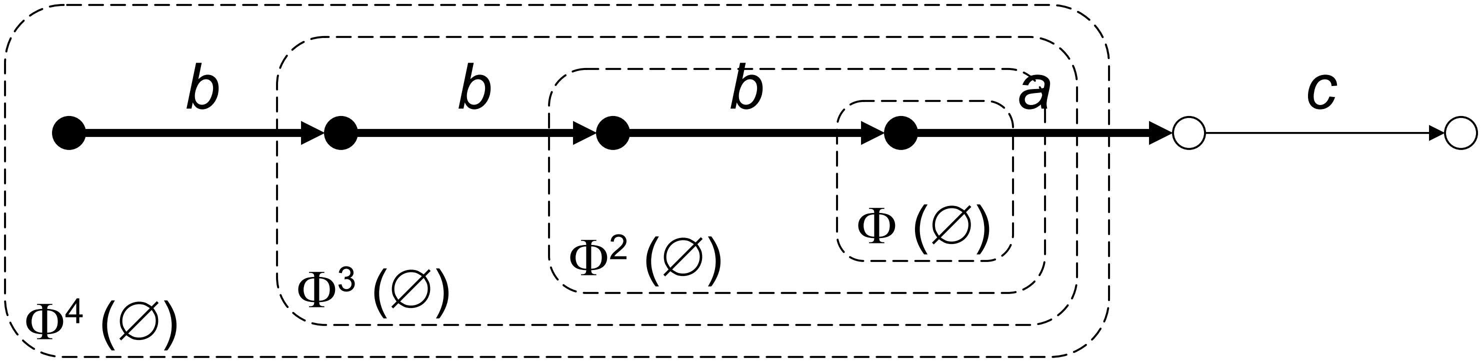
- Potential reachability of an action *a* (existence of a sequence leading to a transition labeled by *a*):

$$\mu X \,.\, \langle\, a \,\rangle\, \mathrm{tt} \vee \langle\, \mathrm{tt} \,\rangle\, X$$

- Associated functional:

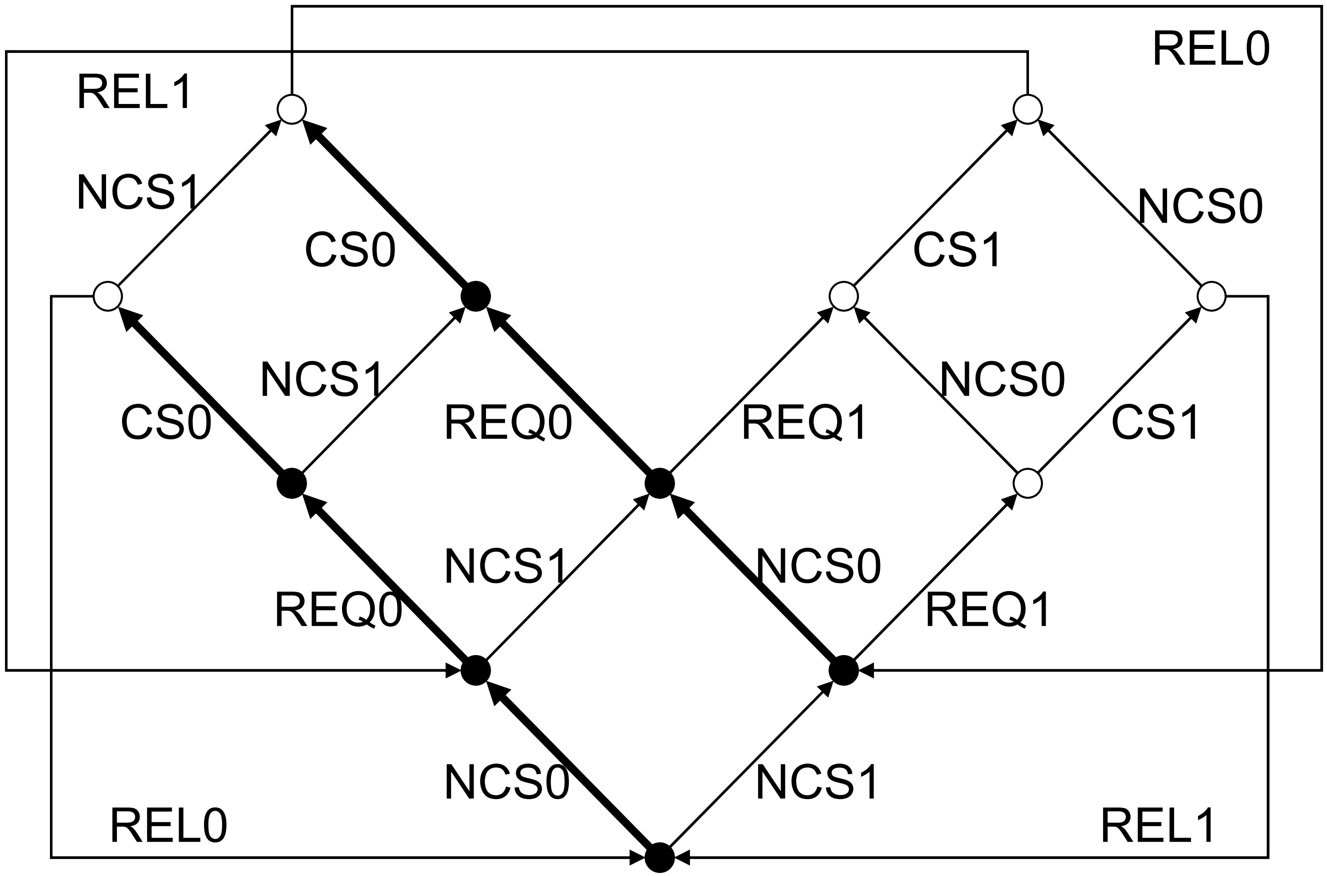
$$\Phi\ (U) = [[\ \langle\, a \,\rangle\, \mathrm{tt} \vee \langle\, \mathrm{tt} \,\rangle\, X\ ]]\ \ [\ U\ /\ X\ ]$$

- Evaluation on an LTS:



Example

Potential reachability: $\mu X \,.\, \langle\, CS_0 \,\rangle\, tt \vee \langle\, \neg(REL_1 \vee REL_0) \,\rangle\, X$



Maximal fixed point

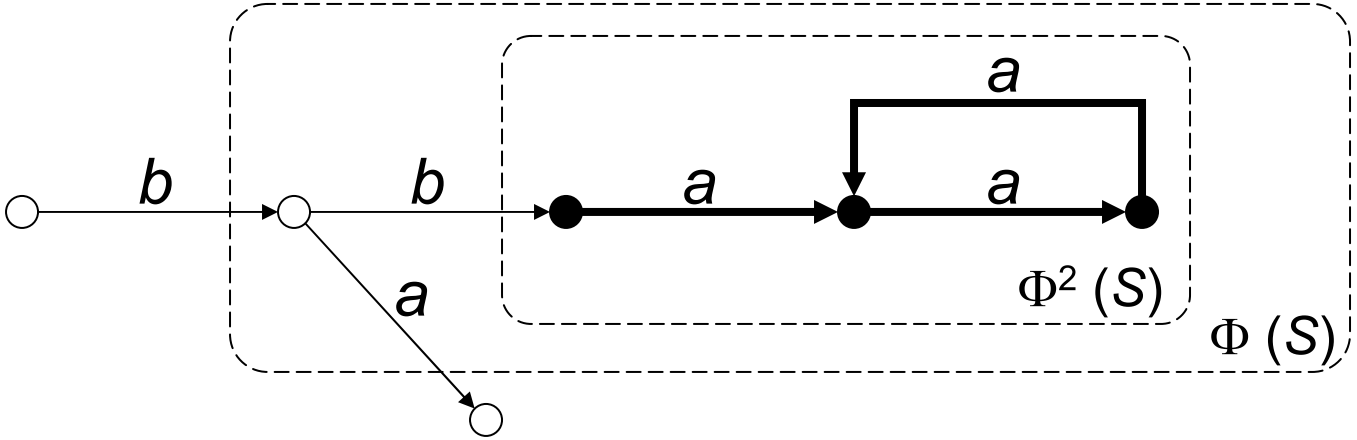
- Infinite repetition of an action *a* (existence of a cycle containing only transitions labeled by *a*):

  $\nu X \,.\, \langle a \rangle X$

- Associated functional:

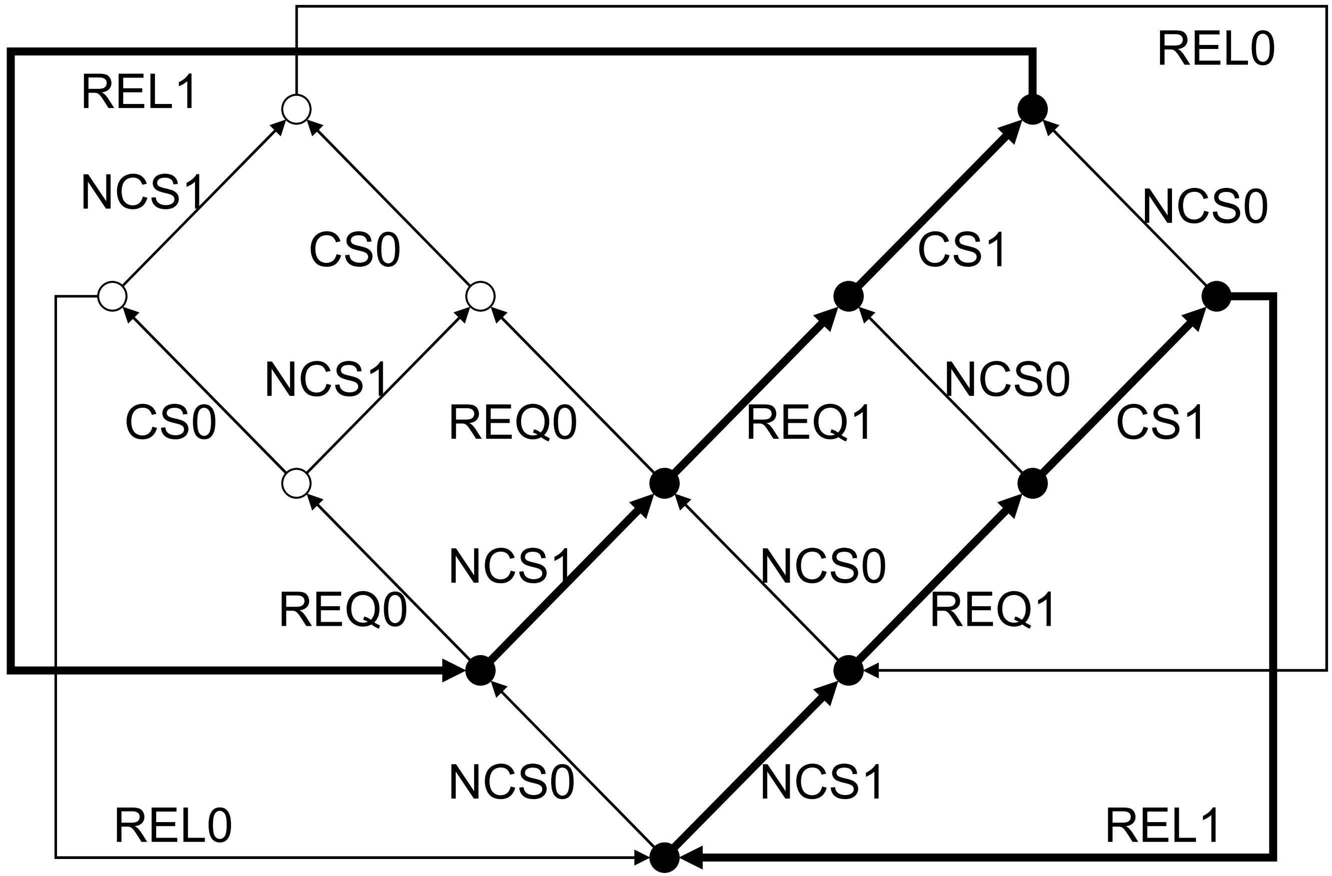
  $\Phi\,(U) = [[\,\langle a \rangle X\,]]\;[\,U\,/\,X\,]$

- Evaluation on an LTS:



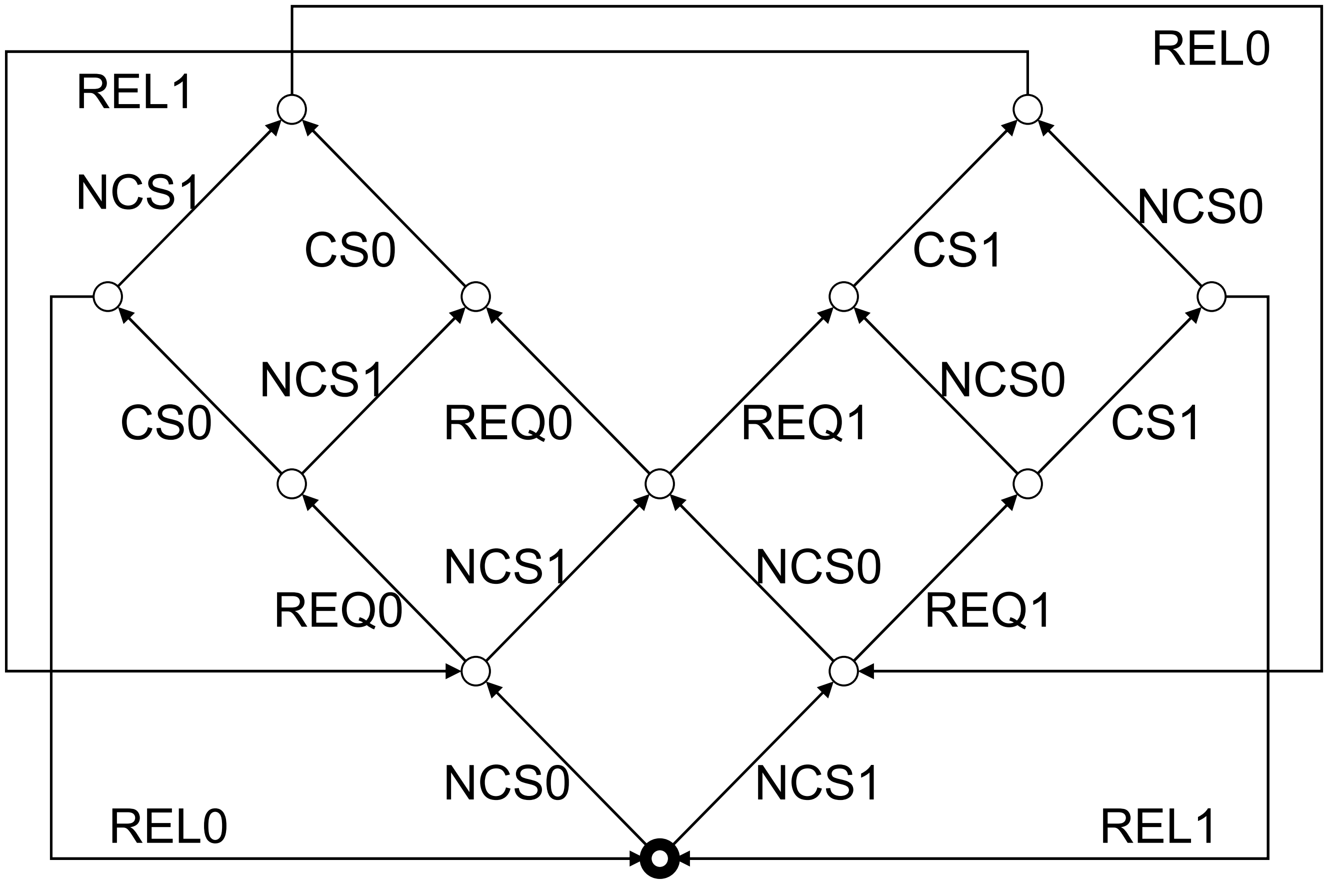
Example

Infinite repetition: $\nu X \,.\, \langle \mathrm{NCS}_1 \vee \mathrm{REQ}_1 \vee \mathrm{CS}_1 \vee \mathrm{REL}_1 \rangle X$



Exercise

Evaluate the formula: $\mu X \,.\, \langle\, CS_0 \,\rangle\, tt \vee ([\, NCS_0 \,]\, ff \wedge \langle\, tt \,\rangle\, X\,)$



Some identities

- Description of (some) ACTL operators:
  - $E[\varphi_{1\alpha 1} U_{\alpha 2} \varphi_2] = \mu X . \varphi_1 \wedge (\langle \alpha_2 \rangle \varphi_2 \vee \langle \alpha_1 \rangle X)$
  - $A[\varphi_{1\alpha 1} U_{\alpha 2} \varphi_2] = \mu X . \varphi_1 \wedge \langle tt \rangle tt \wedge [\neg(\alpha_1 \vee \alpha_2)] ff$
    $\wedge [\neg\alpha_1 \wedge \alpha_2] \varphi_2 \wedge [\neg\alpha_2] X \wedge [\alpha_1 \wedge \alpha_2] (\varphi_2 \vee X)$
  - $EF_\alpha \varphi = \mu X . \varphi \vee \langle \alpha \rangle X$
  - $AF_\alpha \varphi = \mu X . \varphi \vee (\langle tt \rangle tt \wedge [\neg\alpha] ff \wedge [\alpha] X)$
- Description of the PDL operators:
  - $\langle \beta^* \rangle \varphi = \mu X . \varphi \vee \langle \beta \rangle X$
  - $[\beta^*] \varphi = \nu X . \varphi \wedge [\beta] X$

Inevitable reachability

- Inevitable reachability of an action *a*:

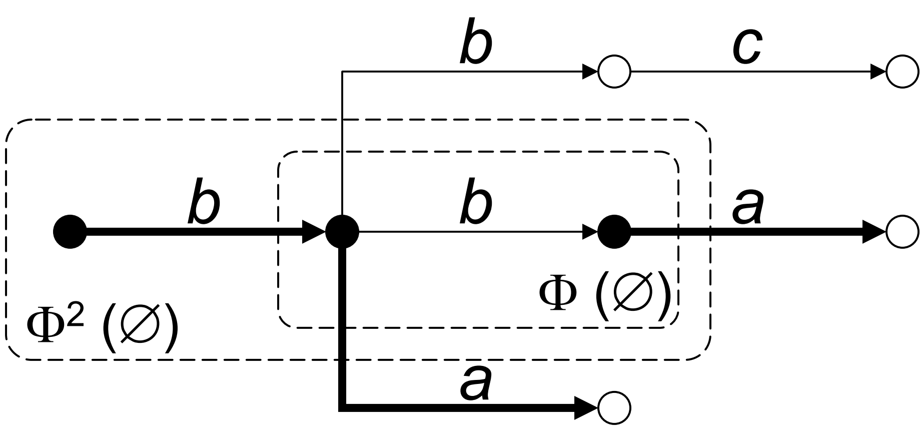
  $\text{access}(a) = AF_{tt} \langle a \rangle tt =$

  $\mu X . \langle a \rangle tt \vee (\langle tt \rangle tt \wedge [tt] X)$

- Associated functional:

  $\Phi (U) = [[\langle a \rangle tt \vee (\langle tt \rangle tt \wedge [tt] X)]] \; [U / X]$

- Evaluation on an LTS:



Inevitable execution

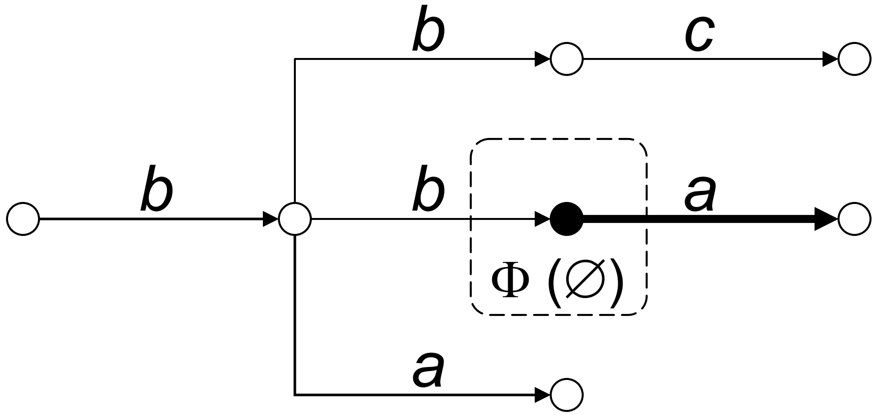
- Inevitable execution of an action *a*:

$$\text{inev}\ (a) = \mu X\ .\ \langle\ \text{tt}\ \rangle\ \text{tt} \wedge [\ \neg a\ ]\ X$$

- Associated functional:

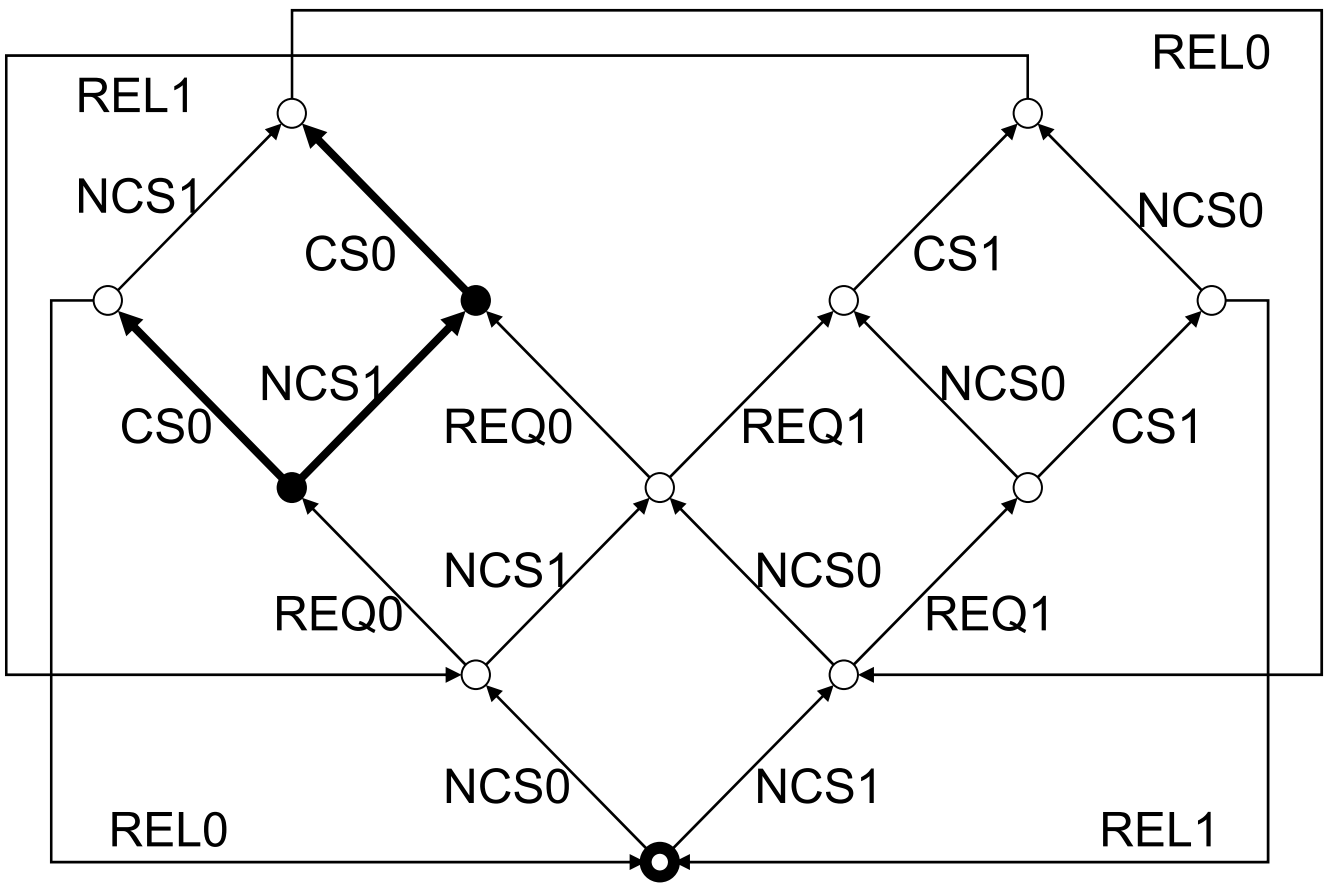
$$\Phi\ (U) = [[\ \langle\ \text{tt}\ \rangle\ \text{tt} \wedge [\ \neg a\ ]\ X\ ]]\ \ [\ U\ /\ X\ ]$$

- Evaluation on an LTS:



Example

Inevitable execution: $\mu X \,.\, \langle\, tt \,\rangle\, tt \wedge [\, \neg CS_0 \,]\, X$



Fair execution

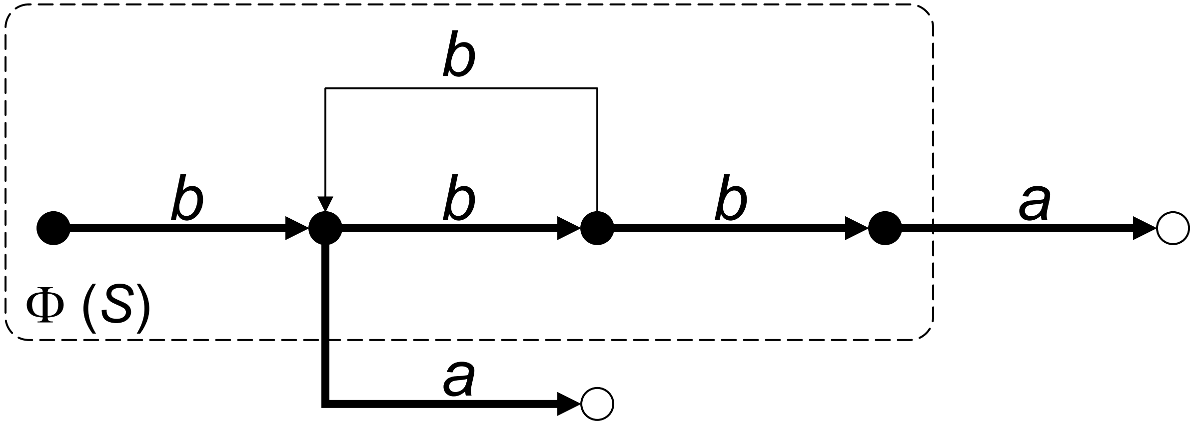
- Fair execution of an action a:

$$\text{fair}\,(a) = [\,(\neg a)^*\,]\,\langle\, tt^*.\ a \,\rangle\, tt$$
$$= \nu X\,.\,\langle\, tt^*.\ a \,\rangle\, tt \wedge [\,\neg a\,]\,X$$

- Associated functional:

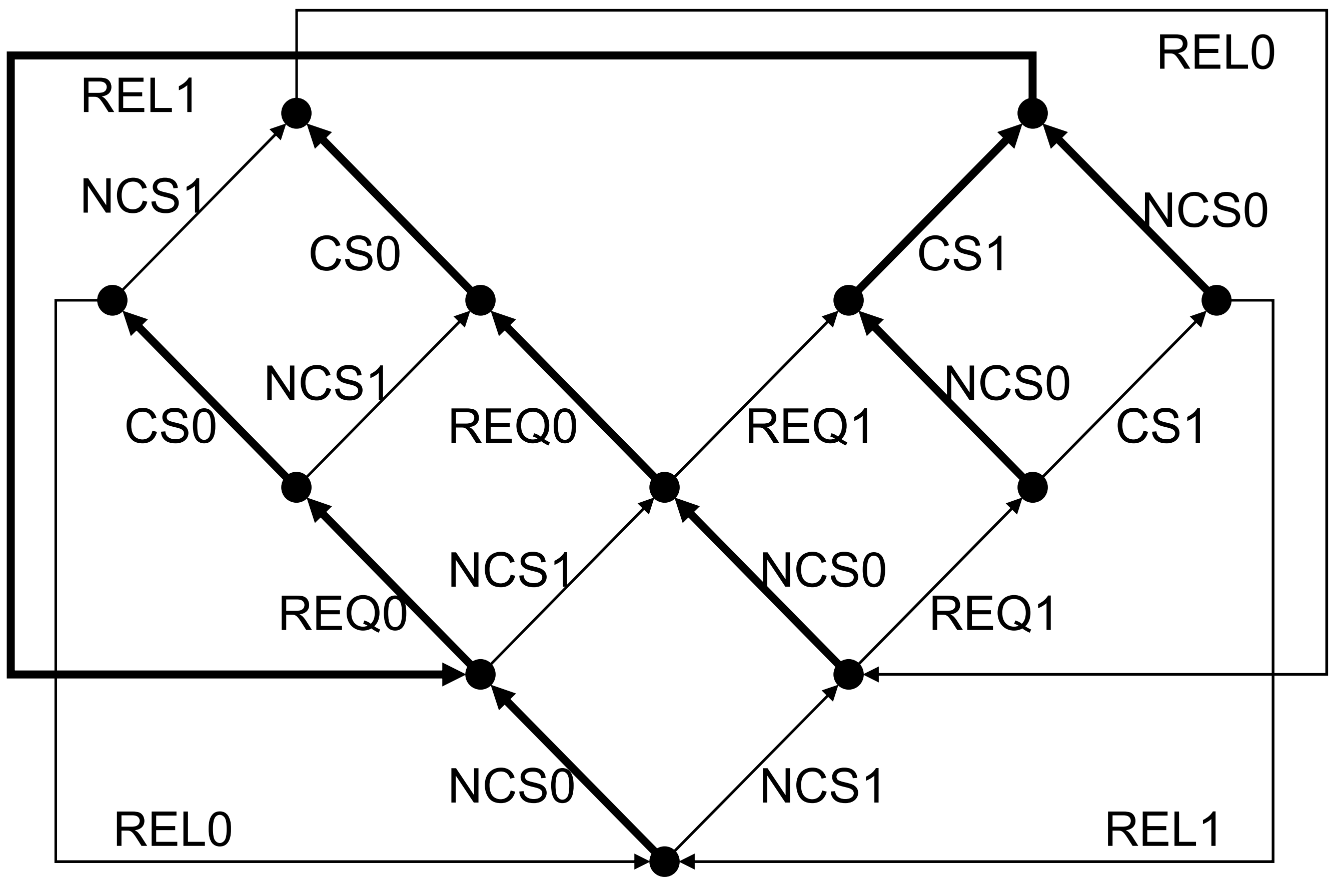
$$\Phi\,(U) = [\![\,\langle\, tt^*.\ a \,\rangle\, tt \wedge [\,\neg a\,]\,X\,]\!]\ [\,U\,/\,X\,]$$

- Evaluation on an LTS:



Example

Fair execution: $[\ (\neg CS_0)^*\ ]\ \langle\ tt^*.\ CS_0\ \rangle\ tt$



Fixed point logics
(summary)

- They allow to encode virtually all TL proposed in the literature
- Expressive power obtained by *nesting* the fixed point operators:

$$\langle\, (a \,.\, b^*)^* \,.\, c \,\rangle\, \mathrm{tt} =$$

$$\mu X \,.\, \langle\, c \,\rangle\, \mathrm{tt} \vee \langle\, a \,\rangle\, \mu Y \,.\, (X \vee \langle\, b \,\rangle\, Y\,)$$

- Alternation depth of a formula: degree of mutual recursion between μ and ν fixed points

  Example of alternation depth 2 formula:

$$\nu X \,.\, \langle\, a^* . \, b \,\rangle\, X \;=\; \nu X \,.\, \mu Y \,.\, \langle\, b \,\rangle\, X \vee \langle\, a \,\rangle\, Y$$

Some verification tools
(for action-based logics)

- CWB (Edinburgh)

  and

- Concurrency Factory (State University of New York)
  - Modal μ-calculus (fixed point operators)

- JACK (University of Pisa, Italy)
  - μ-ACTL (modal μ-calculus combined with ACTL)

- CADP / Evaluator 3.x (INRIA Rhône-Alpes / VASY)
  - Regular alternation-free μ-calculus (PDL modalities and fixed point operators)

Extensions of μ-calculus with data

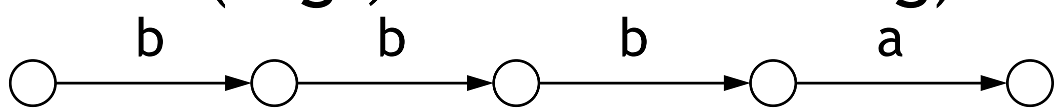
- Temporal logics (ACTL, PDL, ...) and μ-calculi
  - No data manipulation (basic LOTOS, pure CCS, ...)
  - Too low-level operators (complex formulas)

➔ *Extended temporal logics are needed in practice*

- Several μ-calculus extensions with data:
  - For polyadic pi-calculus [Dam-94]
  - For symbolic transition systems [Rathke-Hennessy-96]
  - For μCRL [Groote-Mateescu-99]
  - For full LOTOS [Mateescu-Thivolle-08]

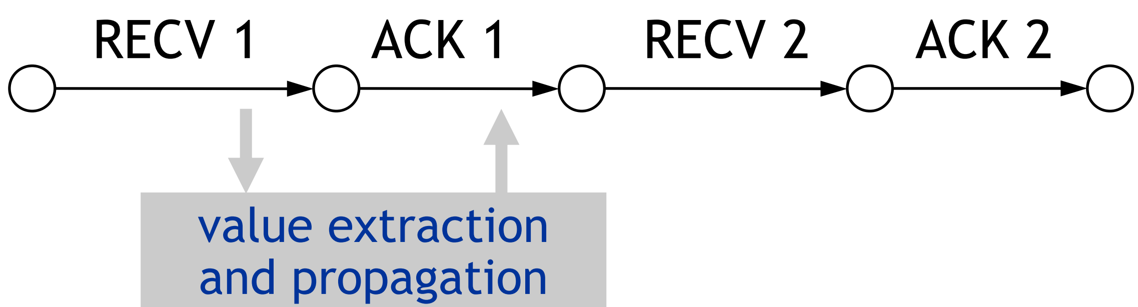
Why to handle data?

- Some properties are cumbersome to express without data (e.g., action counting):



$\langle\, b \,\rangle \langle\, b \,\rangle \langle\, b \,\rangle \langle\, a \,\rangle\ \mathrm{tt}$ or $\langle\, b\ \{3\}\ .\ a \,\rangle\ \mathrm{tt}$?

- LTSs produced from value-passing process algebraic languages (full CCS, LOTOS, ...) contain values on transition labels

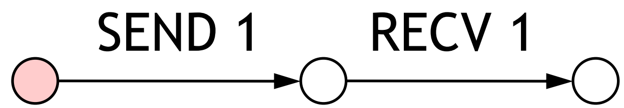


Model Checking Language

- Based on EVALUATOR 3.5 input language
  - standard μ-calculus
  - regular operators
- Data-handling mechanisms
  - data extraction from LTS labels
  - regular operators with counters
  - variable declaration
  - parameterized fixed point operators
  - expressions
- Constructs inspired from programming languages

Parameterized modalities

- Possibility:



< {SEND ?msg:Nat} > < {RECV !msg} > true

value extraction and propagation

- Necessity:

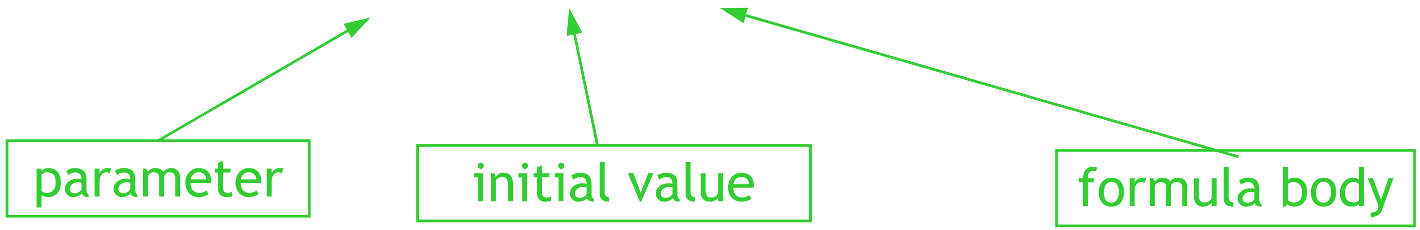
[{RECV ?msg:Nat}] (msg < 6)

value extraction and propagation

Parameterized fixed points

- (basic) syntax:

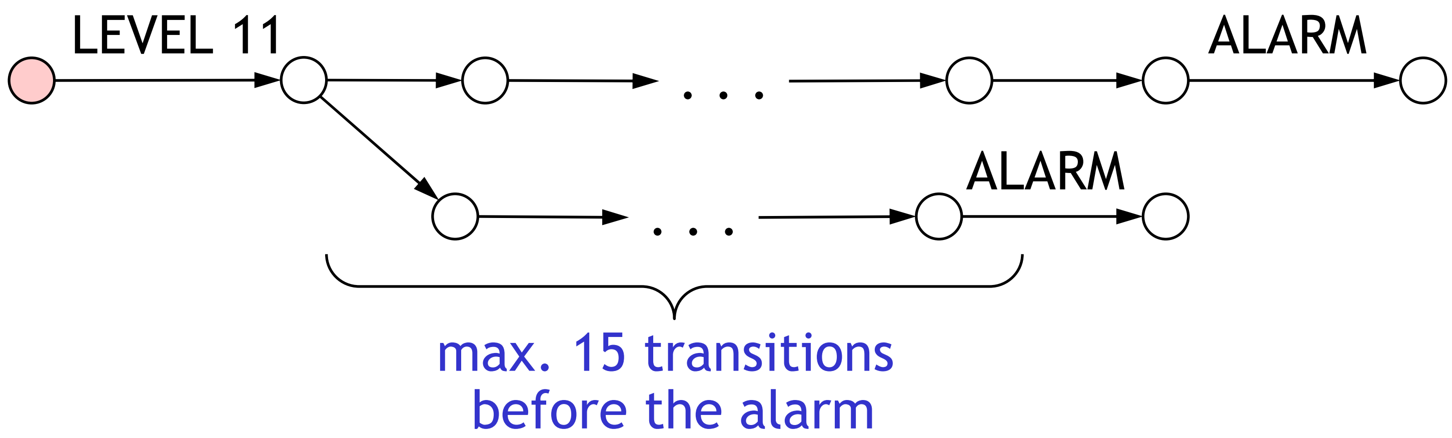
  mu X (y:T := E) . P



  – P contains « calls » X (E')
  – Allows to perform computations and store intermediate results while exploring the PLTS

Example (1/3)

- Counting of actions (e.g., clock ticks):



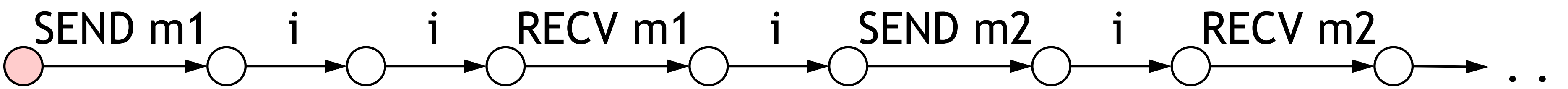
[{LEVEL ?l:Nat where l > 10}]

nu X (c:Nat := 15) .

[not ALARM] (c > 0 and X (c - 1))

Example (2/3)

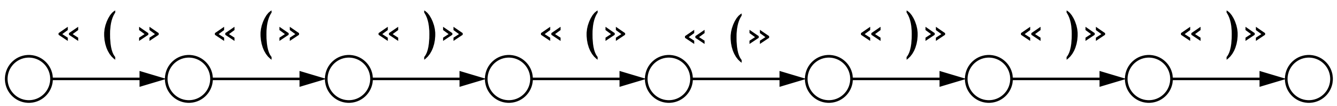
- Alternation of two actions and value propagation:




```
nu X (s:Bool := true, m:Msg := nil) . (
     [ {SEND ?p:Msg} ] (s and X (false, p))
    and
    [ {RECV ?q:Msg} ] (not s and q = m and X (true, nil))
    and
    [ not ({RECV any} or {SEND any}) ] X (s, m)
)
```

Example (3/3)

- Syntax analysis on sequences:




```
mu X (op_cl:nat := 0) . (
    (([ true ] false) implies (op_cl = 0))      and
    < “(” > X (op_cl + 1)                       and
    < “)” > ((op_cl > 0) and X (op_cl - 1))
)
```

- Allows to simulate pushdown automata (by storing the stack in a parameter)

Quantifiers

- Existential quantifier:

  exists x:T among { E_1 ... E_2 } . P

  limits of the subdomain of T

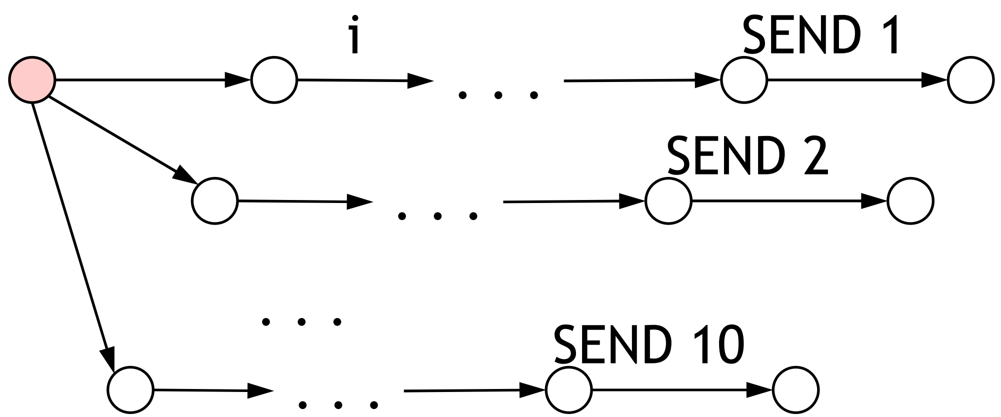
- Universal quantifier:

  forall x:T among { E_1 ... E_2 } . P

➔ *shorthands for large disjunctions and conjunctions*

Example

- Broadcast of messages:



forall msg:Nat among { 1 … 10 } .

mu X . (< {SEND !msg} > true or < true > X)

Conditional operators (1/2)

- Branching operator:

  if P_1 then P_1'
      elsif P_2 then P_2'
      ...
      else P_n' ← mandatory clause
  end if

- Semantics:

  (P_1 and P_1') or
  ((not (P_1) and P_2) and P_2') or ...
  ((not (P_1 or P_2 or ... P_{n-1})) and P_n')

Syntactic restrictions

- State formulas present in conditions must be *propositionally closed* (to ensure syntactic monotonicity)
- Example (illegal):

  mu X . (...

  if X then P_1 else P_2 end if

)

  boolean translation:

  mu X . (...

  (X and P_1) or (not X and P_2)

)

negative occurrence of X

Example

- Counting of actions (revisited):

```
[ {LEVEL ?l:Nat where l > 10} ]
    nu X (c:Nat := 0) .
        if c < 15 then
            [ not ALARM ] X (c + 1)
        else
            [ not ALARM ] false
        end if
```

Conditional operators (2/2)

- Selection operator:

  case E is
  M_1 -> P_1
  | ...
  | any -> P_n
  end case

  mandatory exhaustiveness

- Semantics:

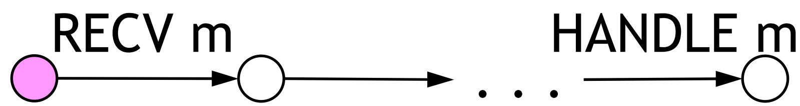
  ((E match M_1) and P_1) or ... or
  (not ((E match M_1) or ... or (E match M_{n-1})) and P_n)

Example

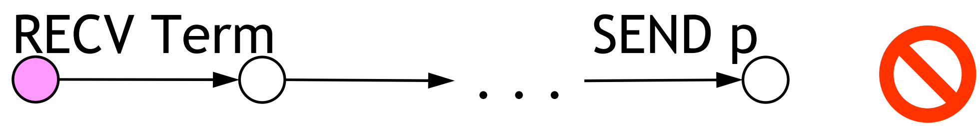
- Message handling (event/reaction):

[{RECV ?m:Msg}]

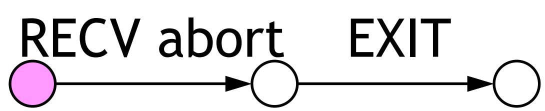
case kind (m) is



Norm -> mu X . < {HANDLE !m} > true or < true > X



| Term -> nu Y . [{SEND any}] false and [true] Y



| Abort -> < true > true and [not EXIT] false

end case

Variable definition

- Initialisation operator:

```
let x:T := E in
      P
end let
```

- Example:

```
[ {RECV ?l:NatList} ]
  let n:Nat := sum (l) in
    < {DELIVER !n} > < {ACK !n} > true
  end let
```

Extended regular formulas

- Counting operators:

R { E }	*repetition E times*
R { E_1 ... }	*repetition at least E_1 times*
R { E_1 ... E_2 }	*repetition between E_1 and E_2 times*

- Some identities:

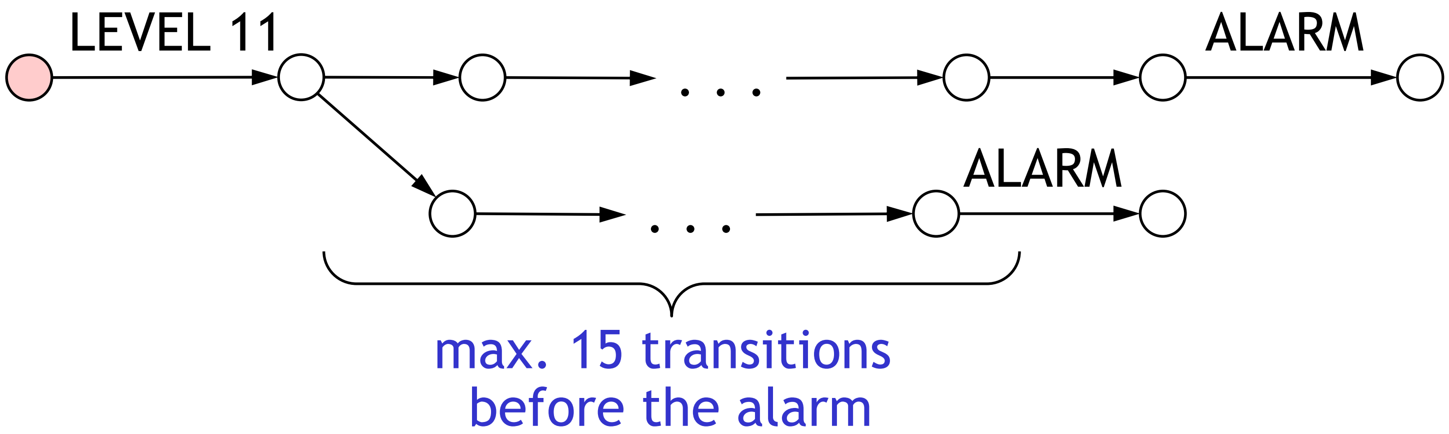
nil = false *	R + = R . R*
R * = R { 0 ... }	R ? = R { 0 ... 1 }
R + = R { 1 ... }	R { E } = R { E ... E }

Translations to basic MCL

-
```
< R { E ... } > P =
mu X (c:Nat := 0) .
    if c < E then
        < R > X (c+1)
    else
        P or < R > X (c)
    end if
```

-
```
< R { E1 ... E2 } > P =
mu X (c:Nat := 0) .
    if c < E1 then
        < R > X (c+1)
    elsif c < E2 then
        P or < R > X (c+1)
    else
        P
    end if
```

Example
(action counting revisited)



- Formulation using counting operators:

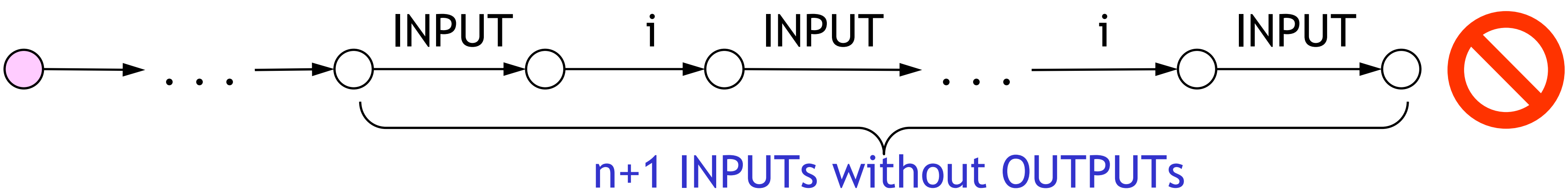
[{LEVEL ?l:Nat where l > 10} . (not ALARM) { 16 }] false

Example

(safety of a n-place buffer)

- Formulation using extended regular operators:

  [true* . ((not OUTPUT)* . INPUT) { n + 1 }] false



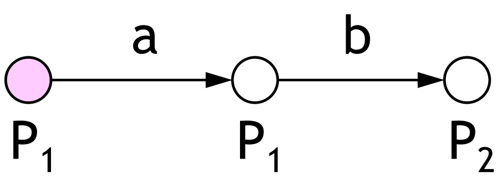
- Formulation using parameterized fixed points:

```
nu X . (nu Y (c:Nat:=0) . (
                  [not OUTPUT] Y (c) and
                  if c = n+1 then [INPUT] false
                         else [INPUT] Y (c+1)
                  end if)
    and [ true ] X)
```

Testing operator of PDL

- PDL with tests [Fischer-Ladner-79]:
  - Express properties of intermediate states of sequences denoted by a regular formula
  - Add a “test” operator on regular formulas
- Syntax (PDL): $P\ ?$
- Semantics: $< P_1 ? > P_2 = P_1 \text{ and } P_2$
- Example: $< P_1 ? . a . P_1 ? . b > P_2 =$

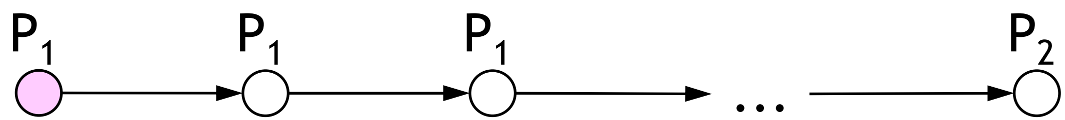
$P_1 \text{ and } < a > (P_1 \text{ and } < b > P_2)$



$P\ ? = \text{if } P \text{ then nil else false end if}$

Example

- Operator E(.U.) of CTL:



E (P_1 U P_2) =

mu X . (P_2 or (P_1 and < true > X)) =

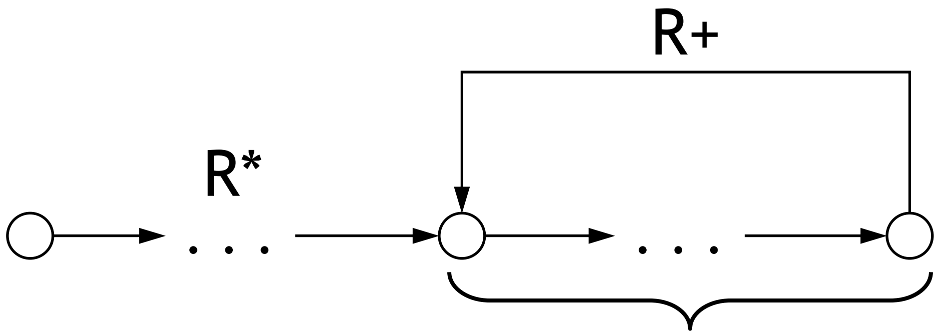
< if P_1 then true end if * > P_2

- "else" clause not mandatory:
  if P then R end if = if P then R else nil end if

Looping operator (from PDL-delta)

- ⊿ R operator added to PDL to specify infinite behaviours [Streett-82]

- MCL syntax: < R > @



cycle containing one or more repetitions of R

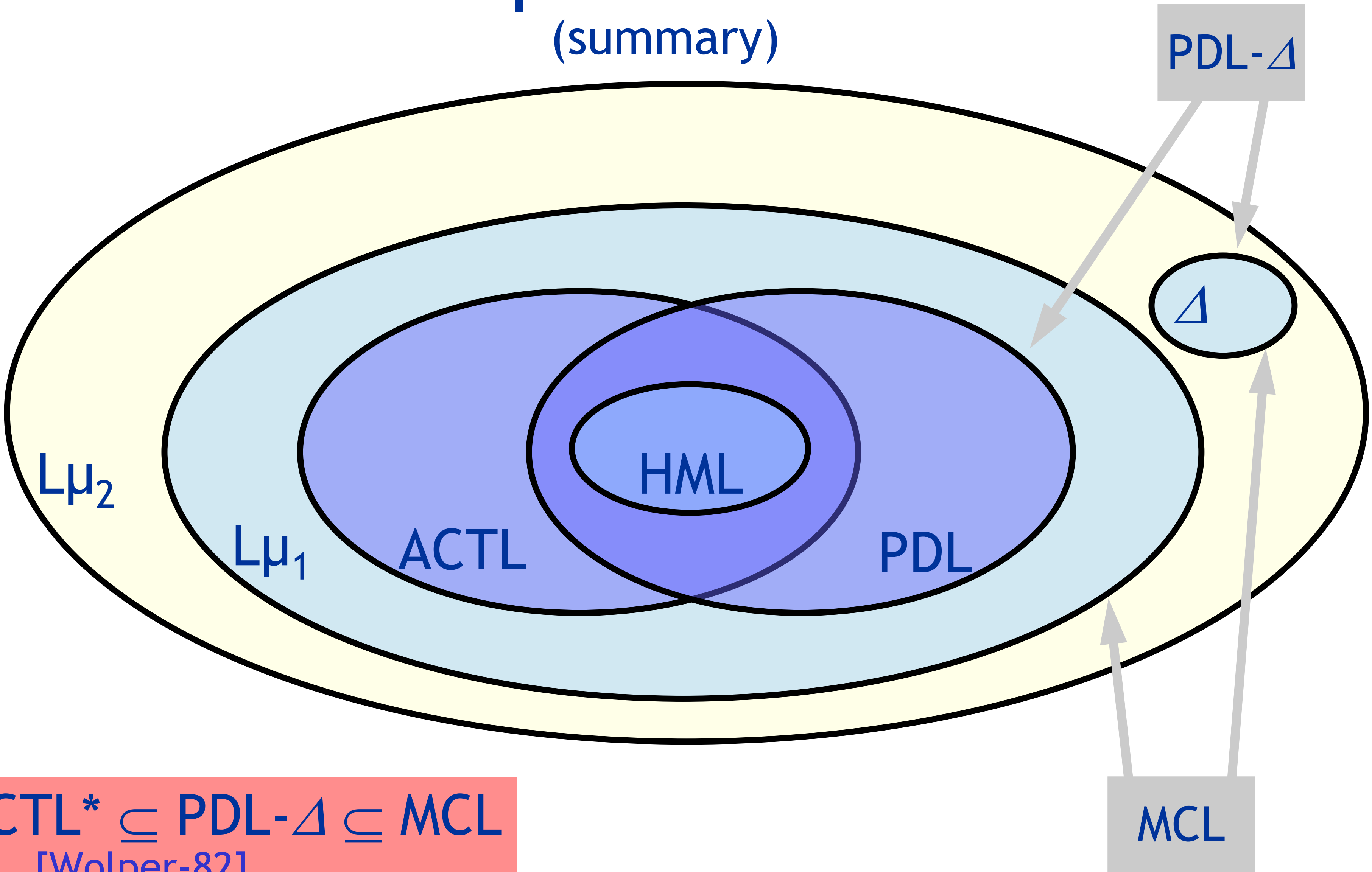
- Examples:
  - process overtaking

    [REQ_0] < (not GET_0)* . REQ_1 . (not GET_0)* . GET_1 > @
  - Büchi acceptance condition

    < true* . if $P_{accepting}$ then true end if > @

  ➔ *allows to encode LTL model checking*

Expressiveness
(summary)



$CTL^* \subseteq$ PDL-$\Delta \subseteq$ MCL
[Wolper-82]

Adequacy with equivalence relations

- A temporal logic L is adequate with an equivalence relation $\approx$ iff for all LTSs M_1 and M_2

$$M_1 \approx M_2 \quad \text{iff} \quad \forall \varphi \in L \,.\, (M_1 \models \varphi \;\Leftrightarrow\; M_2 \models \varphi)$$

- HML:
  - Adequate with strong bisimulation
  - HMLU (HML with Until): weak bisimulation
- ACTL-X (fragment presented here):
  - Adequate with branching bisimulation
- PDL and modal mu-calculus:
  - Adequate with strong bisimulation
  - Weak mu-calculus: weak bisimulation

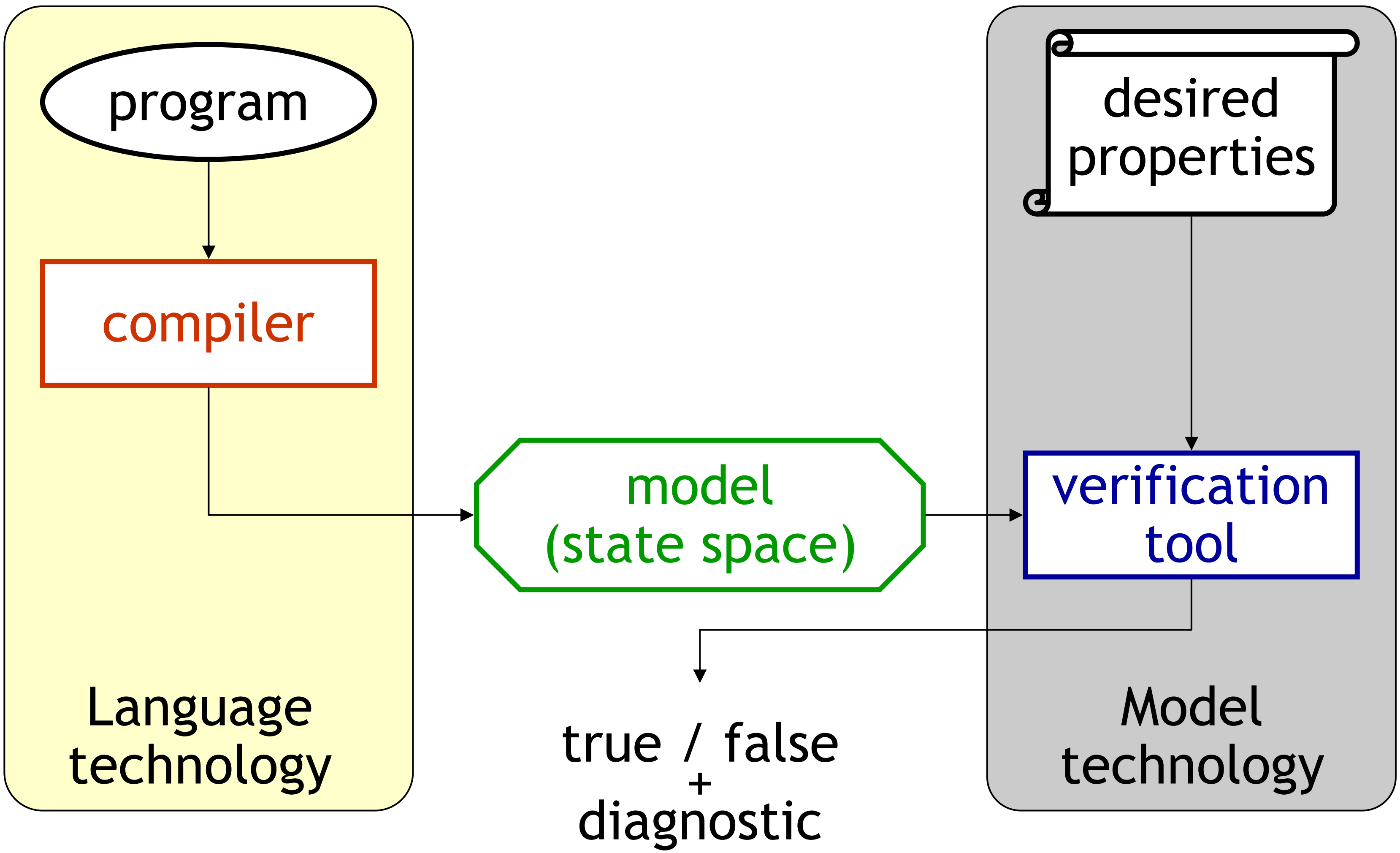
$\langle\langle \;\; \rangle\rangle\, \varphi = \langle\, \tau^* \,\rangle\, \varphi$

$\langle\langle\, a \,\rangle\rangle\, \varphi = \langle\, \tau^* . \, a \, . \, \tau^* \,\rangle\, \varphi$

On-the-fly verification

- Principles
- Alternation-free boolean equation systems
- Local resolution algorithms
- Applications:
  - Equivalence checking
  - Model checking
  - Tau-confluence reduction
- Implementation and use

Principle of explicit-state verification



On-the-fly verification

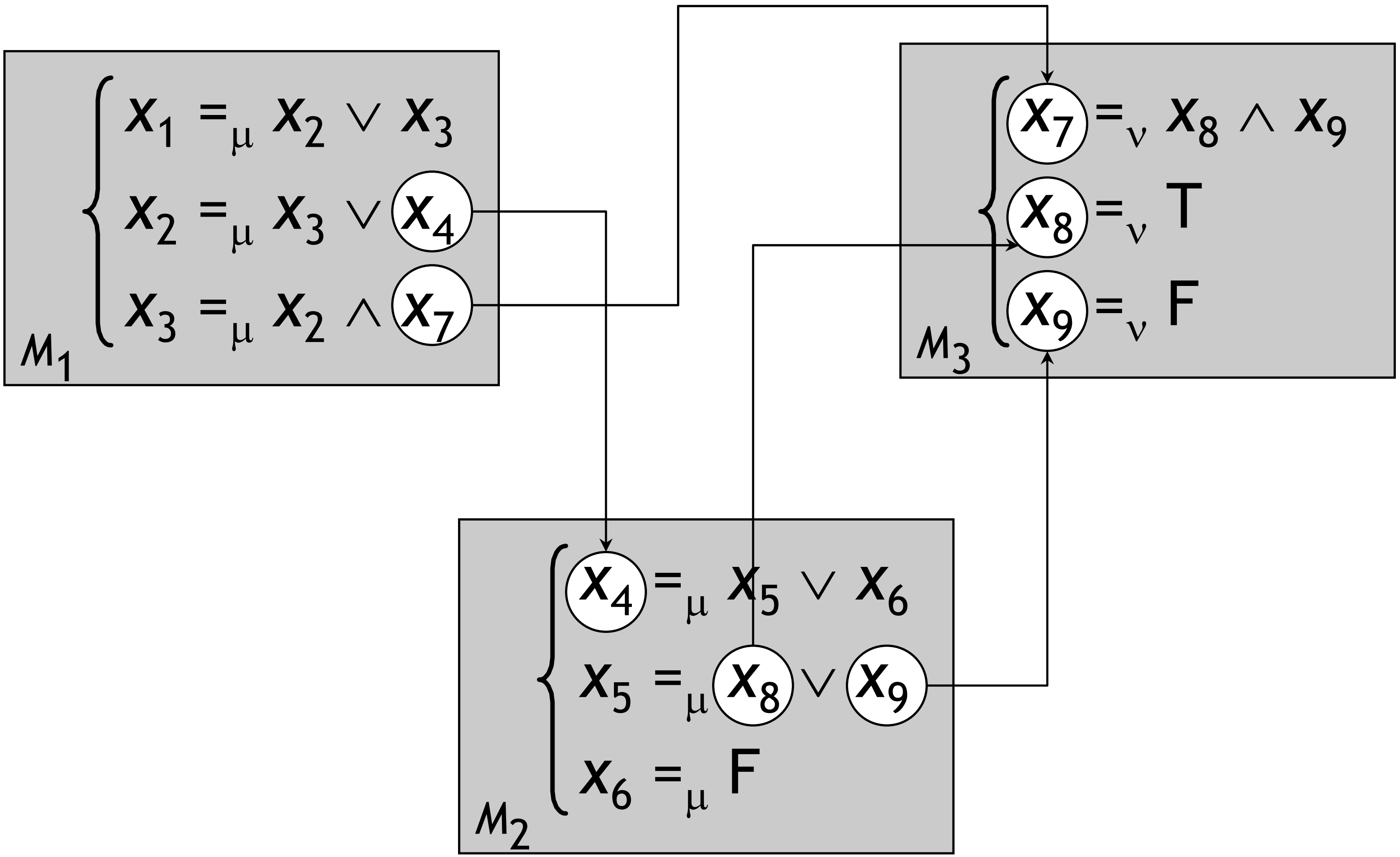
- Incremental construction of the state space
  - Way of fighting against state explosion
  - Detection of errors in complex systems
- "Traditional" methods:
  - Equivalence checking
  - Model checking
- Solution adopted:
  - Translation of the verification problem into the resolution of a *boolean equation system* (BES)
  - Generation of *diagnostics* (fragments of the state space) explaining the result of verification

Boolean equation systems
(syntax)

A BES is a tuple $B = (x, M_1, \ldots, M_n)$, where

- $x \in \boldsymbol{X}$: main boolean variable
- $M_i = \{ x_j =\sigma_i \, op_j \, \boldsymbol{X}_j \}_{j \in [1, mi]}$: equation blocks
  - $\sigma_i \in \{ \mu, \nu \}$: fixed point sign of block i
  - $op_j \in \{ \vee, \wedge \}$: operator of equation j
  - $\boldsymbol{X}_j \subseteq \boldsymbol{X}$: variables in the right-hand side of equation j
  - F = $\vee\emptyset$ (empty disjunction), T = $\wedge\emptyset$ (empty conjunction)
  - x_j depends upon x_k iff $x_k \in \boldsymbol{X}_j$
  - M_i depends upon M_l iff a x_j of M_i depends upon a x_k of M_l
  - *Closed* block: does not depend upon other blocks
- *Alternation-free* BES: M_i depends upon $M_{i+1} \ldots M_n$

Example



x1 =μ x2 ∨ x3
x2 =μ x3 ∨ x4
x3 =μ x2 ∧ x7
M1
x7 =ν x8 ∧ x9
x8 =ν T
x9 =ν F
M3
x4 =μ x5 ∨ x6
x5 =μ x8 ∨ x9
x6 =μ F
M2

Particular blocks

- *Acyclic* block:
  - No cyclic dependencies between variables of the block
- Var. x_i disjunctive (conjunctive): $op_i = \vee$ ($op_i = \wedge$)
- *Disjunctive* block:
  - contains disjunctive variables
  - and conjunctive variables
    - with a single non constant successor in the block (the last one in the right-hand side of the equation)
    - all other successors are constants or free variables (defined in other blocks)
- *Conjunctive* block: dual definition

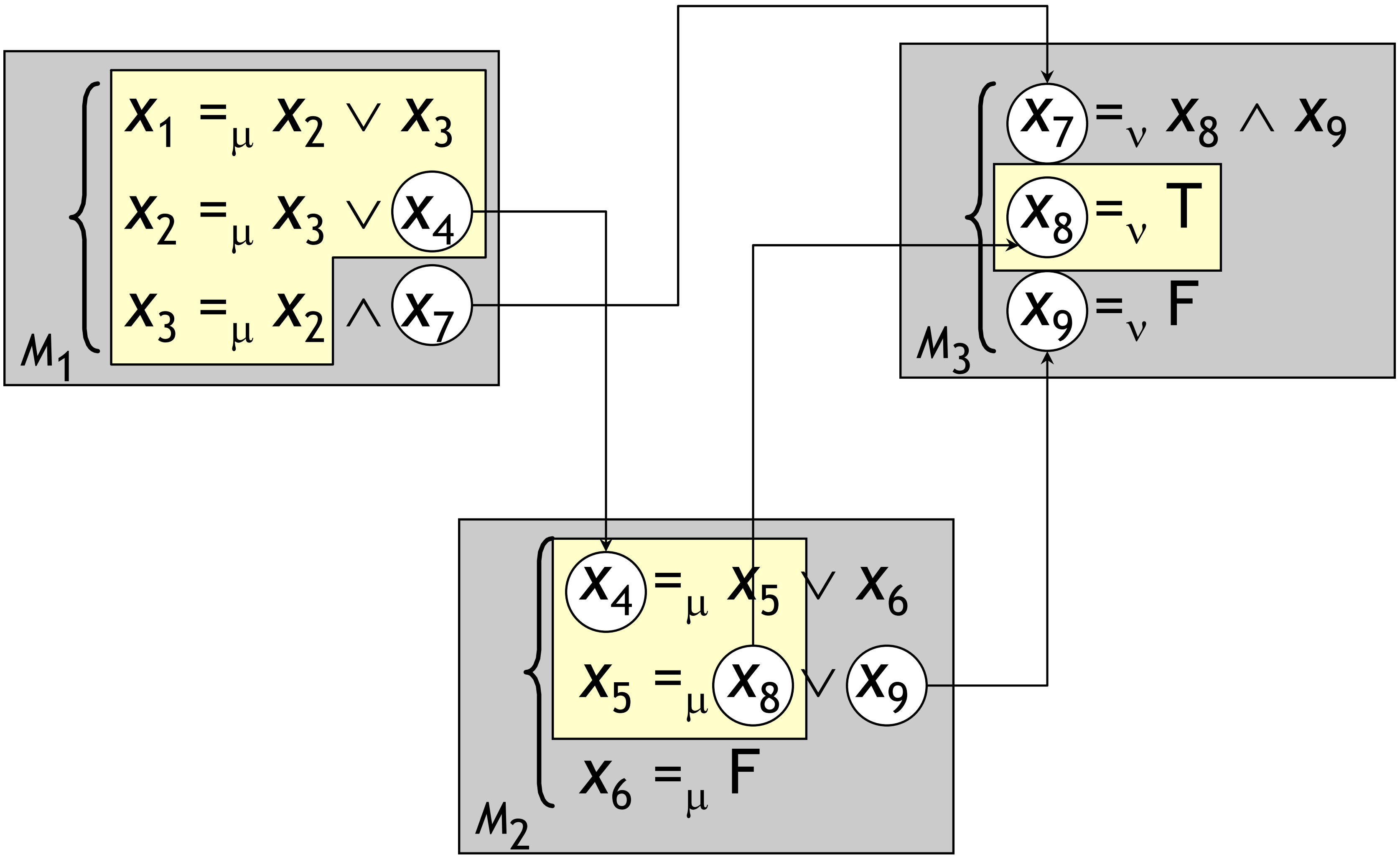
Boolean equation systems
(semantics)

- Context: partial function $\delta : \boldsymbol{X} \rightarrow \text{Bool}$
- Semantics of a boolean formula:
  - $[[\ op\ \{ x_1, \ldots, x_p \}\ ]]\ \delta = op\ (\delta\ (x_1), \ldots, \delta\ (x_p))$
- Semantics of a block:
  - $[[\ \{ x_j =\sigma\ op_j\ \boldsymbol{X}_j \}_{j \in [1, m]}\ ]]\ \delta = \sigma\Phi_\delta$
  - $\Phi_\delta : \text{Bool}^m \rightarrow \text{Bool}^m$
  - $\Phi_\delta\ (b_1, \ldots, b_m) = ([[\ op_j\ \boldsymbol{X}_j\ ]]\ (\delta \oplus [b_1/x_1, \ldots, b_m/x_m]))_{j \in [1, m]}$
- Semantics of a BES:
  - $[[\ (x, M_1, \ldots, M_n)\ ]] = \delta_1\ (x)$
  - $\delta_n = [[\ M_n\ ]]\ []$ (M_n closed)
  - $\delta_i = ([[\ M_i\ ]]\ \delta_{i+1}) \oplus \delta_{i+1}$ (M_i depends upon $M_{i+1} \ldots M_n$)

Local resolution

- Alternation-free BES $B = (x, M_1, \ldots, M_n)$
- Primitive: compute a variable of a block
  - A resolution routine R_i associated to M_i
  - $R_i (x_j)$ computes the value of x_j in M_i
  - Evaluation of the rhs of equations + substitution
  - Call stack $R_1 (x) \rightarrow \ldots \rightarrow R_n (x_k)$ bounded by the depth of the dependency graph between blocks
  - "Coroutine-like" style: each R_i must keep its context
- Advantages:
  - Simple resolution routines (a single type of fixed point)
  - Easy to optimize for particular kinds of blocks

Example


$x_1 =_\mu x_2 \vee x_3$
$x_2 =_\mu x_3 \vee x_4$
$x_3 =_\mu x_2 \wedge x_7$
M_1
$x_4 =_\mu x_5 \vee x_6$
$x_5 =_\mu x_8 \vee x_9$
$x_6 =_\mu F$
M_2
$x_7 =_\nu x_8 \wedge x_9$
$x_8 =_\nu T$
$x_9 =_\nu F$
M_3

Local resolution algorithms

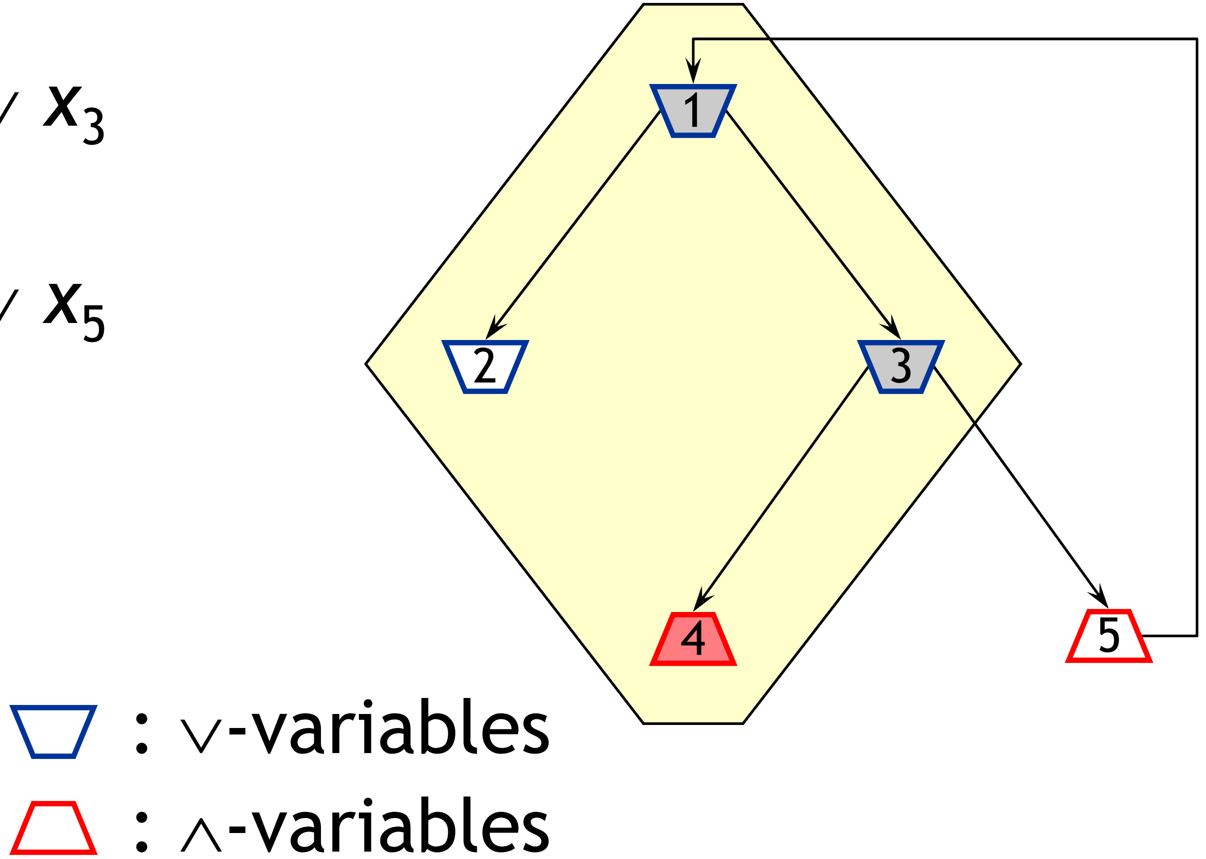
- Representation of blocks as *boolean graphs* [Andersen-94]
- To a block $M = \{ x_j =_\mu op_j\ \boldsymbol{X}_j \}_{j \text{ in } [1, m]}$ we associate the boolean graph $G = (V, E, L, \mu)$, where:
  - $V = \{ x_1, \ldots, x_m \}$: set of vertices (variables)
  - $E = \{ (x_i, x_j) \mid x_j \in \boldsymbol{X}_i \}$: set of edges (dependencies)
  - $L : V \rightarrow \{ \vee, \wedge \}$, $L(x_j) = op_j$: vertex labeling
- Principle of the algorithms:
  - *Forward* exploration of G starting at $x \in V$
  - *Backward* propagation of stable (computed) variables
  - Termination: x is stable or G is completely explored

Example

BES (μ-block)

$$\begin{cases} x_1 =_\mu x_2 \vee x_3 \\ x_2 =_\mu F \\ x_3 =_\mu x_4 \vee x_5 \\ x_4 =_\mu T \\ x_5 =_\mu x_1 \end{cases}$$

boolean graph



Three effectiveness criteria

[Mateescu-06]

For each resolution routine *R*:

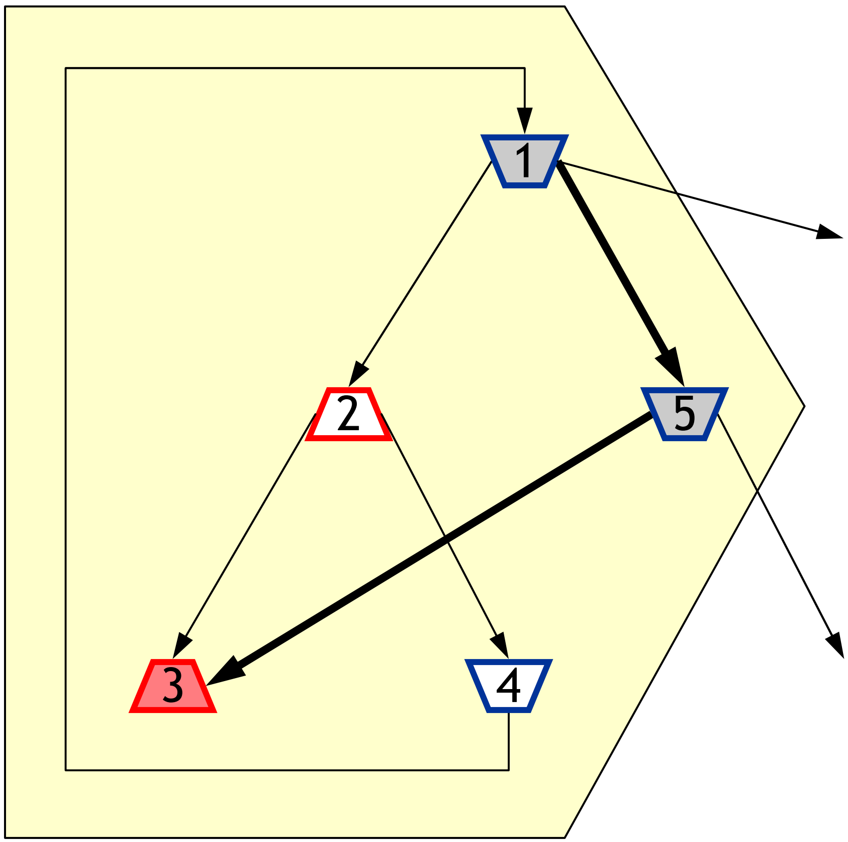
A. The worst-case complexity of a call *R* (*x*) must be $O(|V|+|E|)$
   ➔ *linear-time complexity for the overall BES resolution*

B. While executing *R* (*x*), every variable explored must be « linked » to *x* via unstable variables
   ➔ *graph exploration limited to "useful" variables*

C. After termination of *R* (*x*), all variables explored must be stable
   ➔ *keep resolution results between subsequent calls of R*

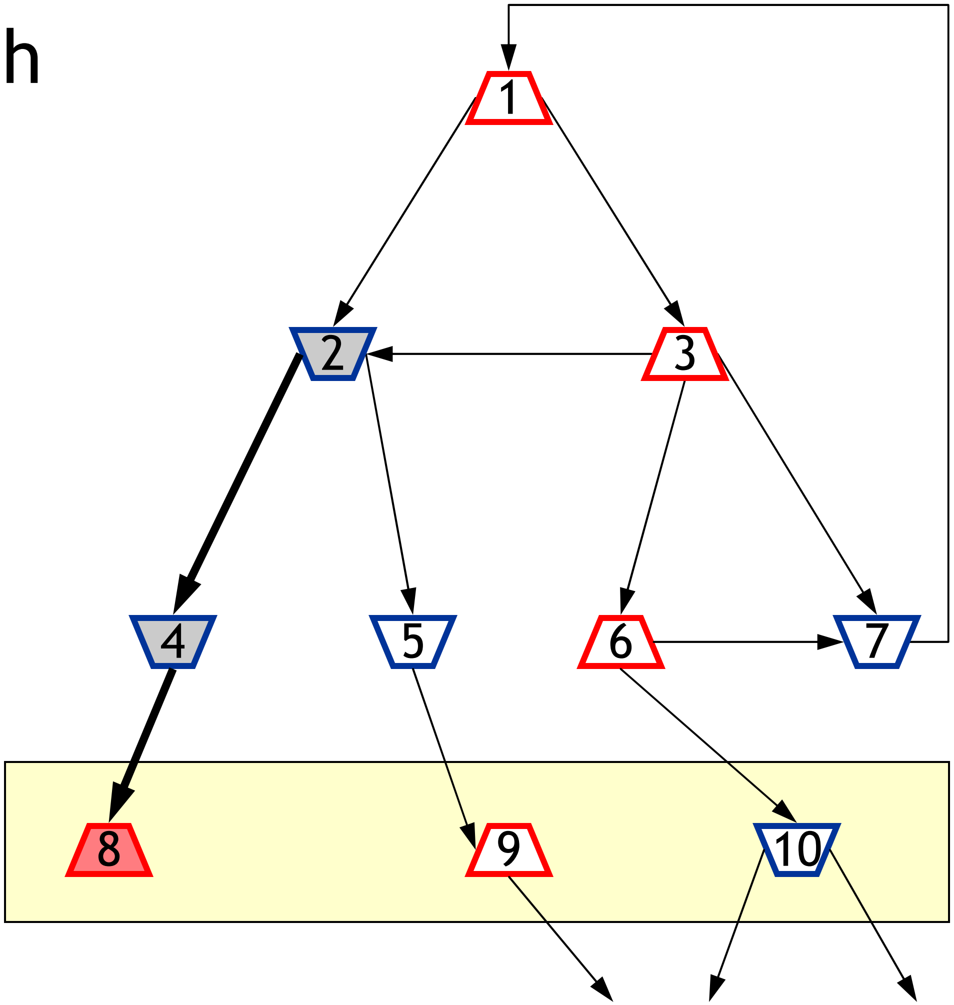
Algorithm A0
(general)

- DFS of the boolean graph
- Satisfies A, B, C
- Memory complexity
  $O(|V|+|E|)$
- Optimized version of
  [Andersen-94]
- Developed for model checking regular alternation-free μ-calculus
  [Mateescu-Sighireanu-00,03]



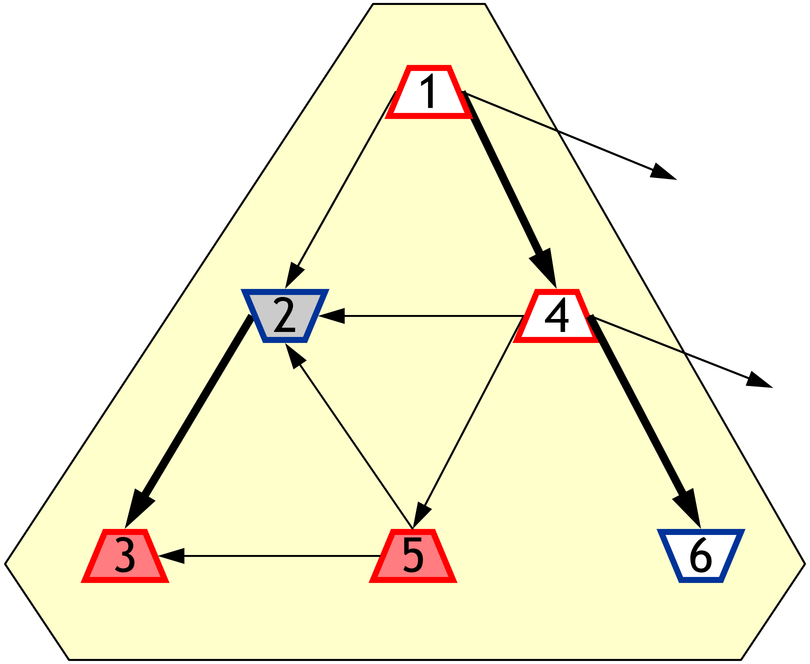
Algorithm A1
(general)

- BFS of the boolean graph
- Satisfies A, C
  (risk of computing useless variables)
- Slightly slower than A0
- Memory complexity
  $O(|V|+|E|)$
- Low-depth diagnostics



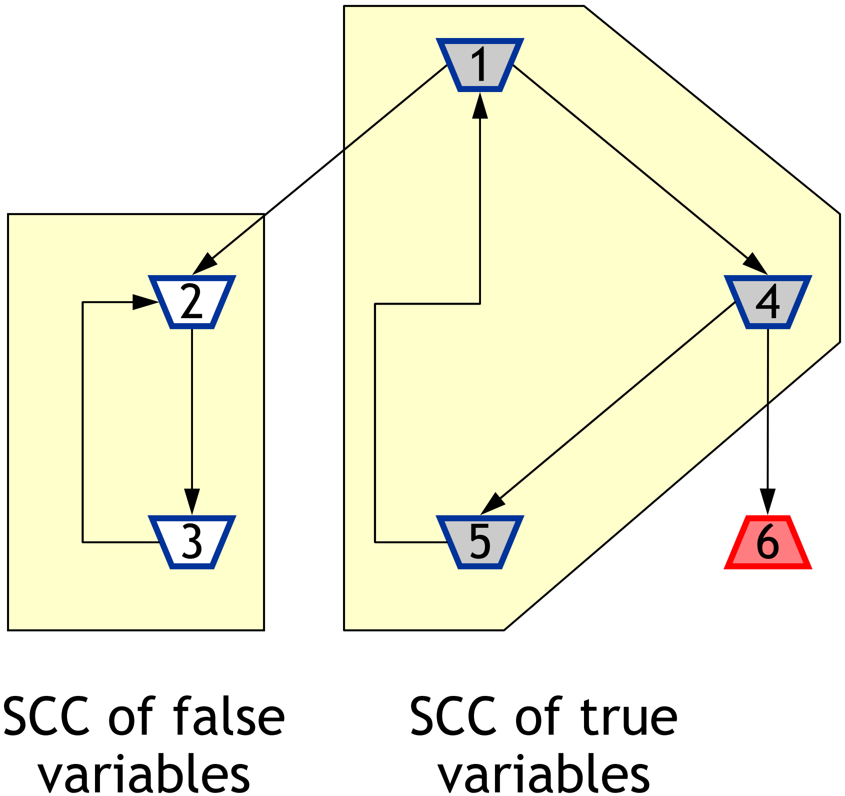
Algorithm A2
(acyclic)

- DFS of the boolean graph
- Back-propagation of stable variables on the DFS stack only
- Satisfies A, B, C
- Avoids storing edges
- Memory complexity $O(|V|)$
- Developed for trace-based verification [Mateescu-02]



Algorithm A3 / A4
(disjunctive / conjunctive)

- DFS of the boolean graph
- Detection and stabilization of SCCs
- Satisfies A, B, C
- Avoids storing edges
- Memory complexity $O(|V|)$
- Developed for model checking CTL, ACTL, and PDL



Resolution algorithms

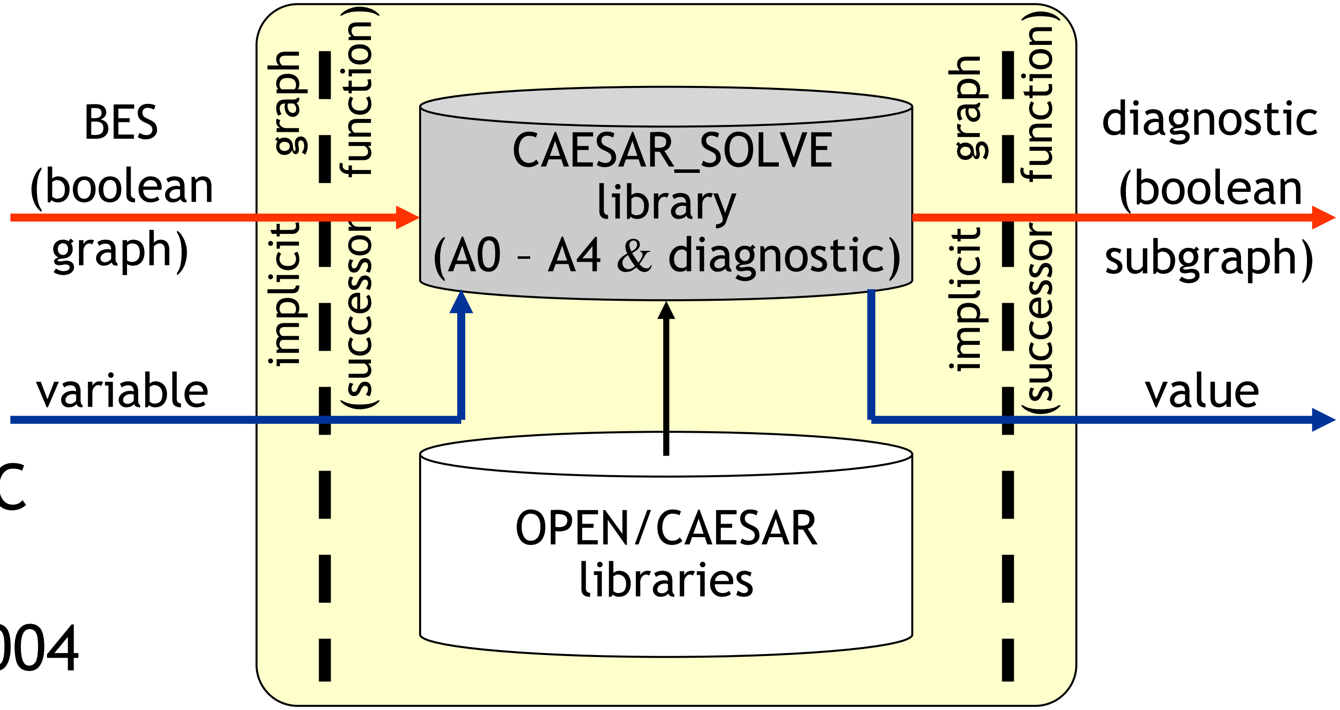
(summary)

- A0 (DFS, general)
  - Satisfies A, B, C
  - Memory complexity $O(|V|+|E|)$
- A1 (BFS, general)
  - Satisfies A, C + « small » diagnostics
  - Memory complexity $O(|V|+|E|)$
- A2 (DFS, acyclic)
  - Satisfies A, B, C
  - Memory complexity $O(|V|)$
- A3/A4 (DFS, disjunctive/conjunctive)
  - Satisfies A, B, C
  - Memory complexity $O(|V|)$

} Time complexity $O(|V|+|E|)$

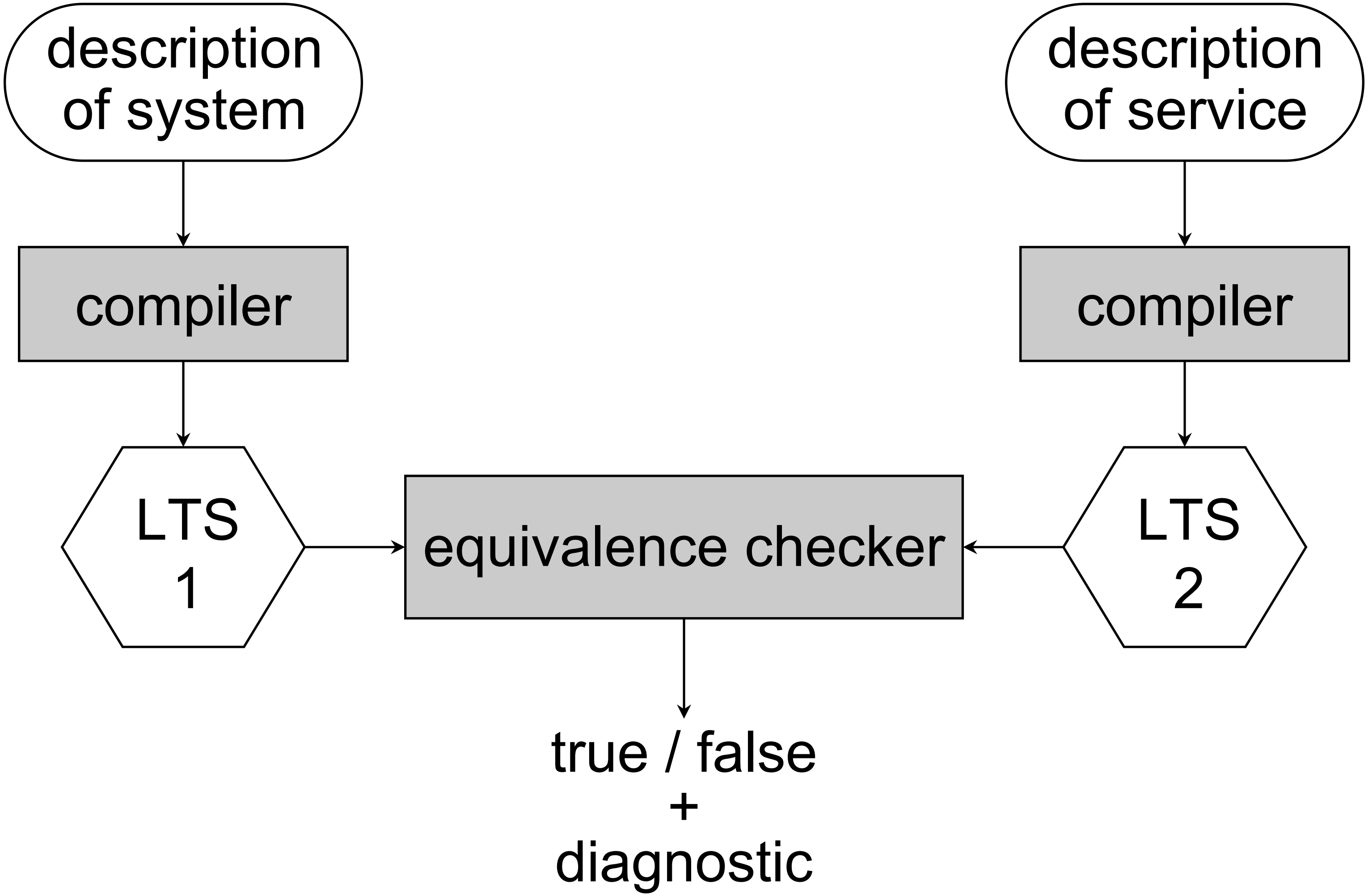
Caesar_Solve library of CADP

[Mateescu-03,06]



- 15 000 lines of C
- Integrated into CADP in Dec. 2004
- Diagnostic generation features [Mateescu-00]
- Used as verification back-end for Bisimulator, Evaluator 3.5 and 4.0, Reductor 5.0

Equivalence checking
(principle)



Strong equivalence

- $M_1 = (Q_1, A, T_1, q_{01}), M_2 = (Q_2, A, T_2, q_{02})$
  $\approx \subseteq Q_1 \times Q_2$ is the maximal relation s.t. $p \approx q$ iff

  $\forall a \in A. \forall p \rightarrow_a p' \in T_1. \exists q \rightarrow_a q' \in T_2. p' \approx q'$
  and
  $\forall a \in A. \forall q \rightarrow_a q' \in T_2. \exists p \rightarrow_a p' \in T_1. p' \approx q'$

- $M_1 \approx M_2$ iff $q_{01} \approx q_{02}$

Translation to a BES

- Principle: $p \approx q$ iff $X_{p,q}$ is true
- General BES:

$$\begin{cases} X_{p,q} =_\nu (\bigwedge_{p \to_a p'} \bigvee_{q \to_a q'} X_{p',q'}) \\ \qquad \wedge \\ \qquad (\bigwedge_{q \to_a q'} \bigvee_{p \to_a p'} X_{p',q'}) \end{cases}$$

- Simple BES:

$$\begin{cases} X_{p,q} =_\nu (\bigwedge_{p \to_a p'} Y_{a,p',q}) \wedge (\bigwedge_{q \to_a q'} Z_{a,p,q'}) \\ Y_{a,p',q} =_\nu \bigvee_{q \to_a q'} X_{p',q'} \\ Z_{a,p,q'} =_\nu \bigvee_{p \to_a p'} X_{p',q'} \end{cases}$$

$p \leq q$ (preorder)

Tau*.a and safety equivalences

- $M_1 = (Q_1, A_\tau, T_1, q_{01})$, $M_2 = (Q_2, A_\tau, T_2, q_{02})$
  $A_\tau = A \cup \{ \tau \}$
- Tau*.a equivalence:

$$\begin{cases} X_{p,q} =_\nu (\wedge_{p \rightarrow \tau^*.a\, p'} \vee_{q \rightarrow \tau^*.a\, q'} X_{p',q'}) \\ \qquad \wedge \\ \qquad (\wedge_{q \rightarrow \tau^*.a\, q'} \vee_{p \rightarrow \tau^*.a\, p'} X_{p',q'}) \end{cases}$$

- Safety equivalence:

$$\begin{cases} X_{p,q} =_\nu Y_{p,q} \wedge Y_{q,p} \\ Y_{p,q} =_\nu \wedge_{p \rightarrow \tau^*.a\, p'} \vee_{q \rightarrow \tau^*.a\, q'} Y_{p',q'} \end{cases}$$

Observational and branching equivalences

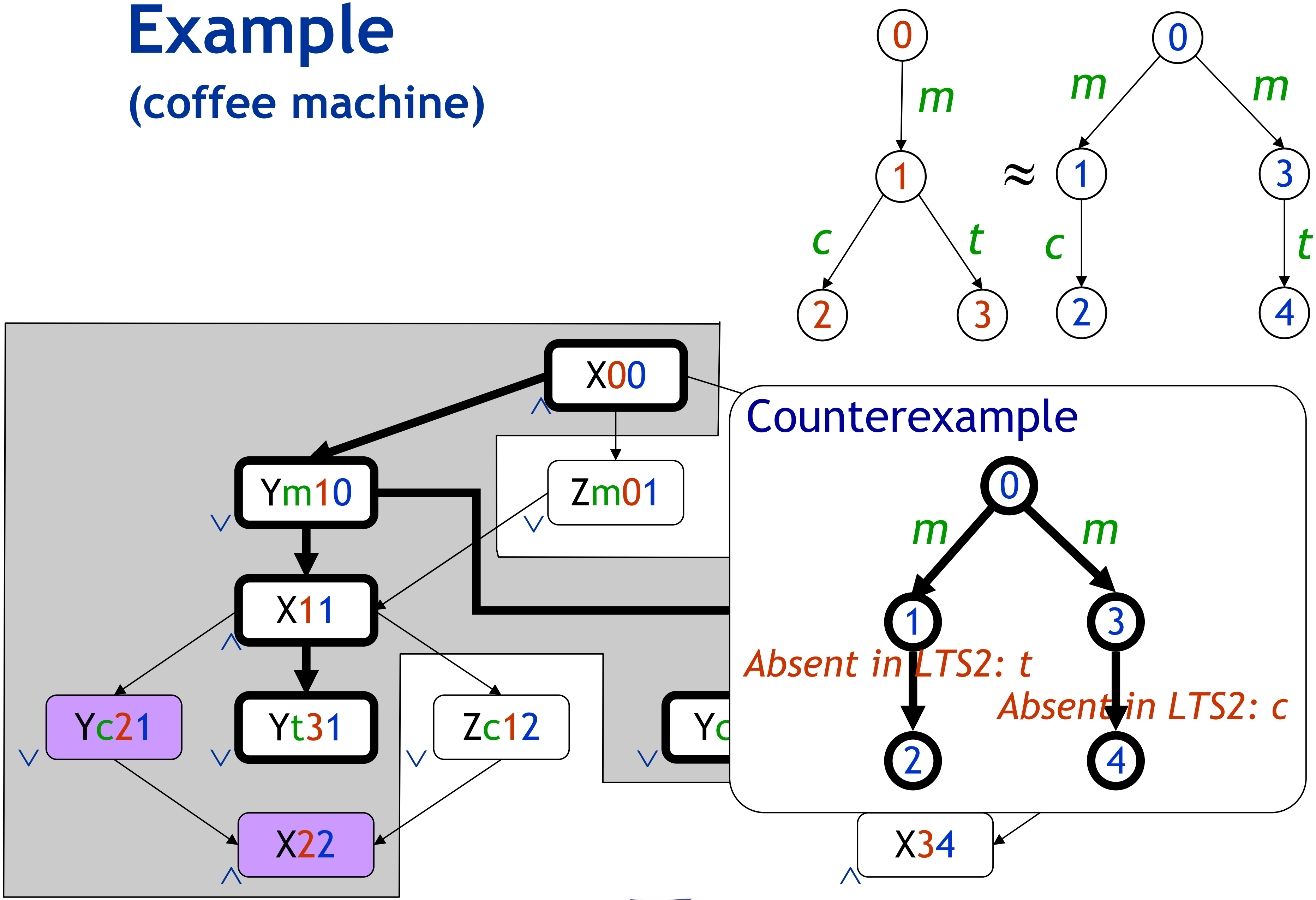
- Observational equivalence:

$$\begin{cases} X_{p,q} =_\nu (\bigwedge_{p \to_\tau p'} \bigvee_{q \to_{\tau^*} q'} X_{p',q'}) \wedge (\bigwedge_{p \to_a p'} \bigvee_{q \to_{\tau^*.a.\tau^*} q'} X_{p',q'}) \\ \qquad \wedge \\ \qquad (\bigwedge_{q \to_\tau q'} \bigvee_{p \to_{\tau^*} p'} X_{p',q'}) \wedge (\bigwedge_{q \to_a q'} \bigvee_{p \to_{\tau^*.a.\tau^*} p'} X_{p',q'}) \end{cases}$$

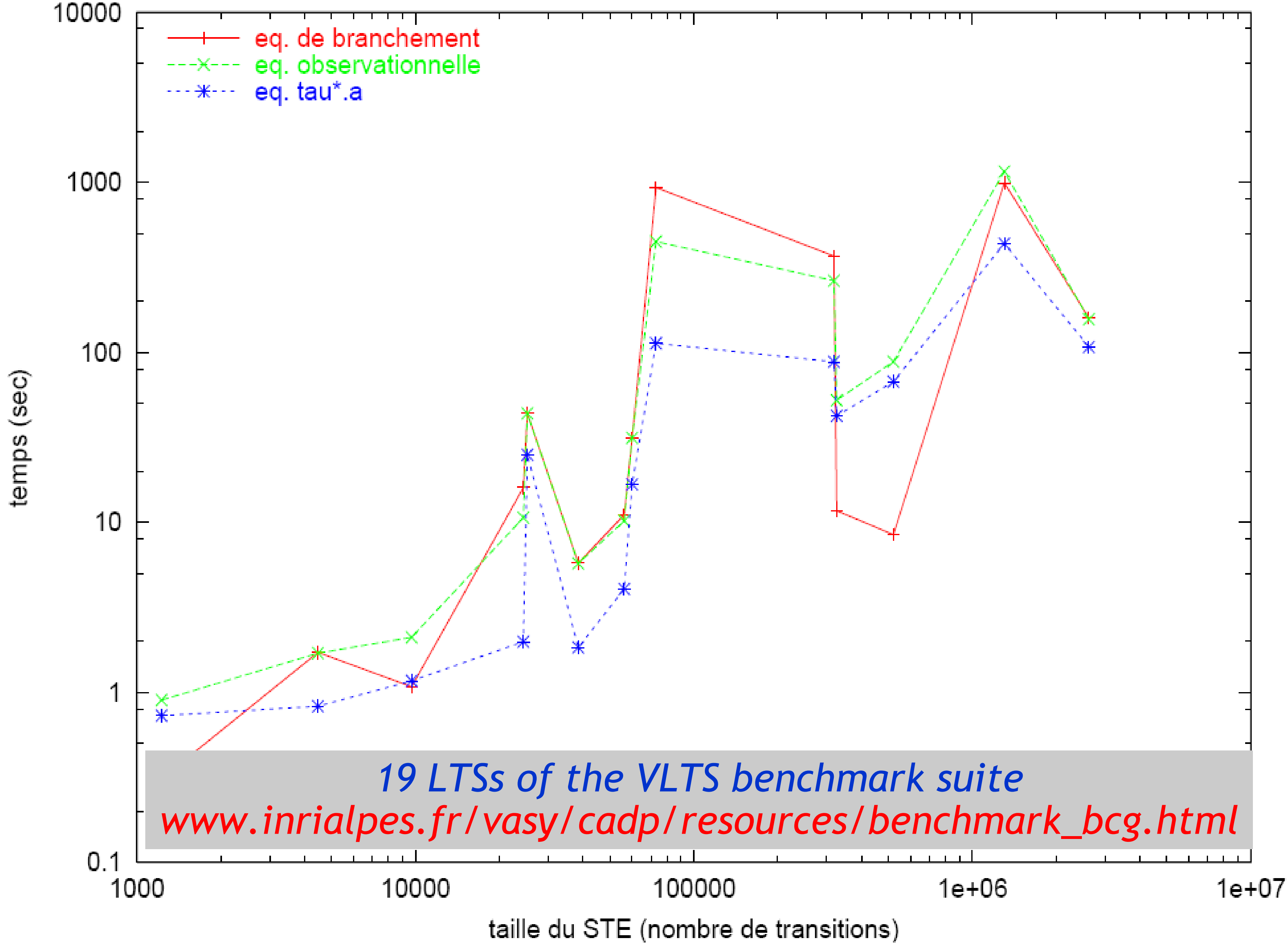
- Branching equivalence:

$$\begin{cases} X_{p,q} =_\nu \bigwedge_{p \to_b p'} ((b=\tau \wedge X_{p',q}) \vee \bigvee_{q \to_{\tau^*} q' \to_b q''} (X_{p,q'} \wedge X_{p',q''}) \\ \qquad \wedge \\ \qquad \bigwedge_{q \to_b q'} ((b=\tau \wedge X_{p,q'}) \vee \bigvee_{p \to_{\tau^*} p' \to_b p''} (X_{p',q} \wedge X_{p'',q'}) \end{cases}$$

Example
(coffee machine)



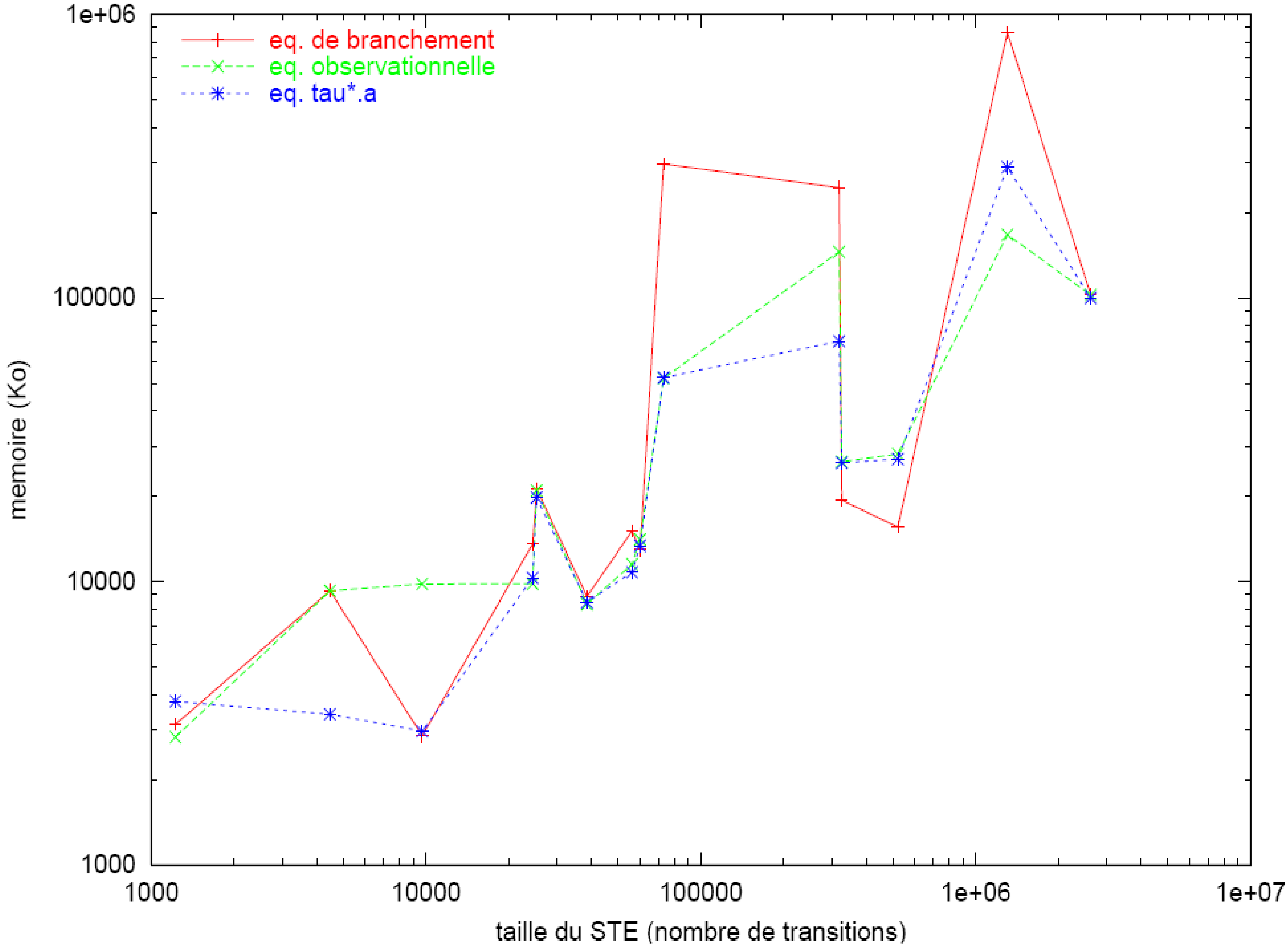
Equivalence checking (time)



19 LTSs of the VLTS benchmark suite

www.inrialpes.fr/vasy/cadp/resources/benchmark_bcg.html

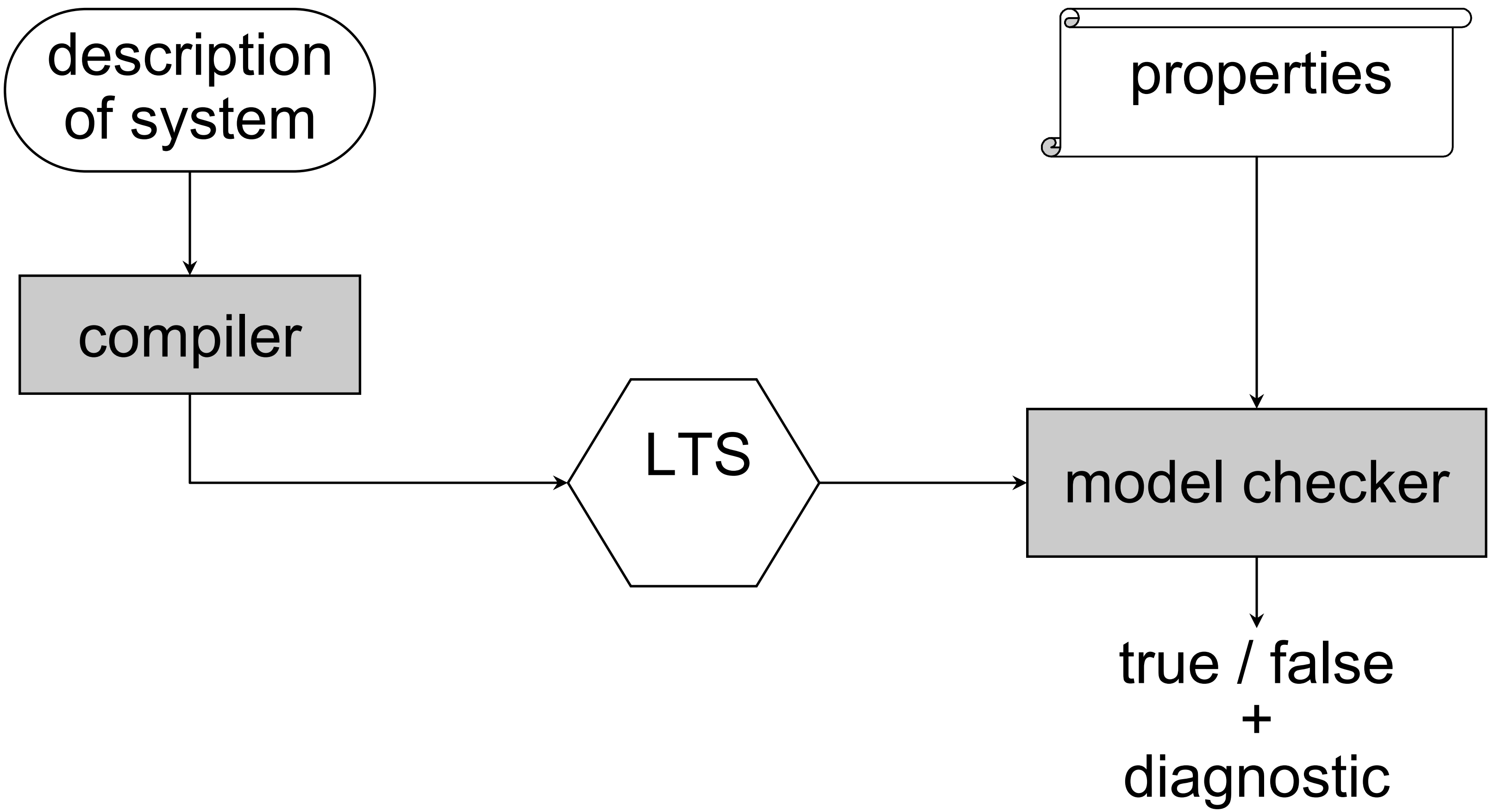
Equivalence checking (memory)



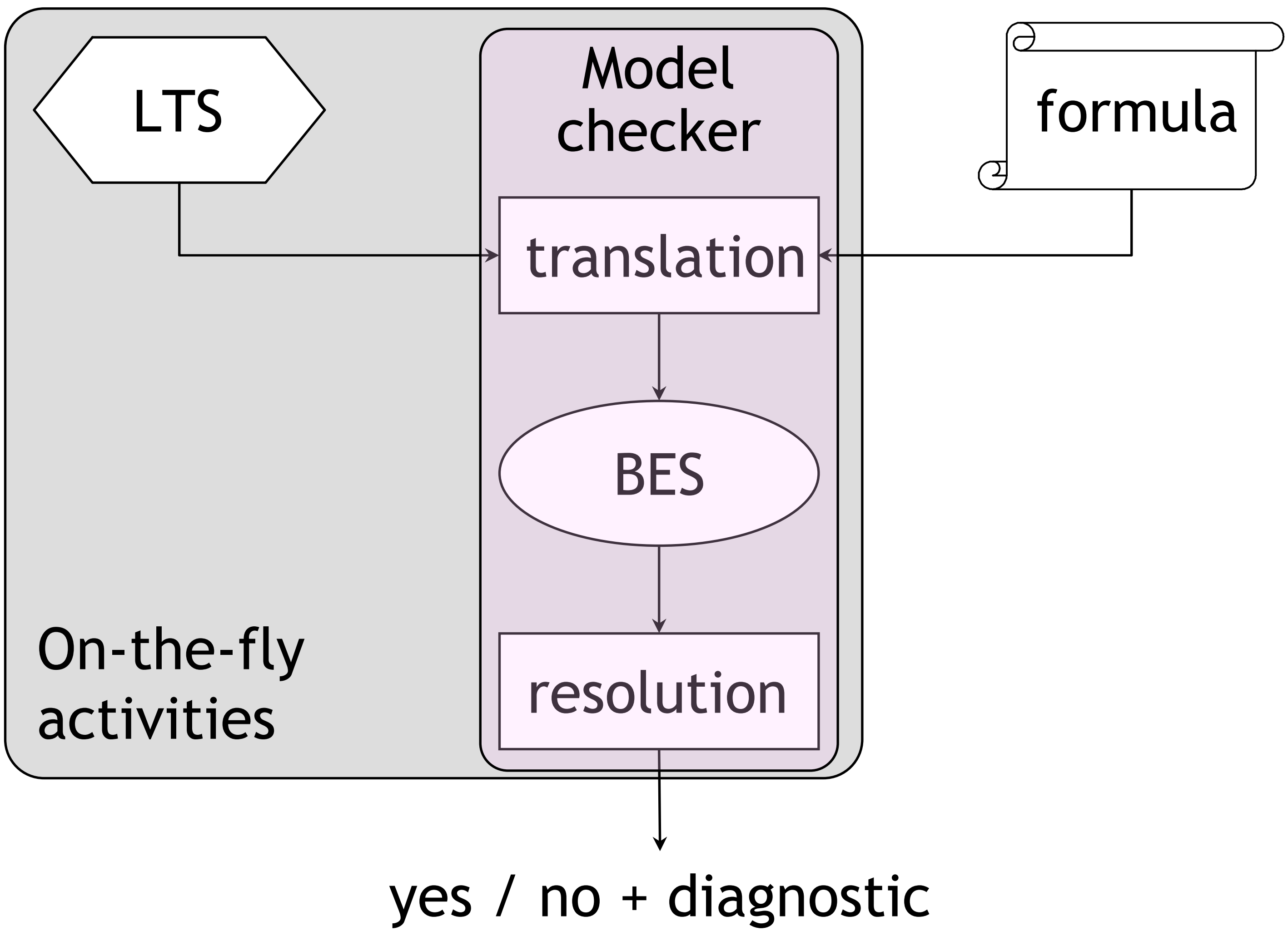
Equivalence checking
(summary)

- *General* boolean graph:
  - All equivalences and their preorders
  - Algorithms A0 and A1 (counterexample depth ↓)
- *Acyclic* boolean graph:
  - Strong equivalence: one LTS acyclic
  - $\tau^*.a$ and safety: one LTS acyclic (τ-circuits allowed)
  - Branching and observational: both LTS acyclic
  - Algorithm A2 (memory ↓)
- *Conjunctive* boolean graph:
  - Strong equivalence: one LTS deterministic
  - Weak equivalences: one LTS deterministic and τ-free
  - Algorithm A4 (memory ↓)

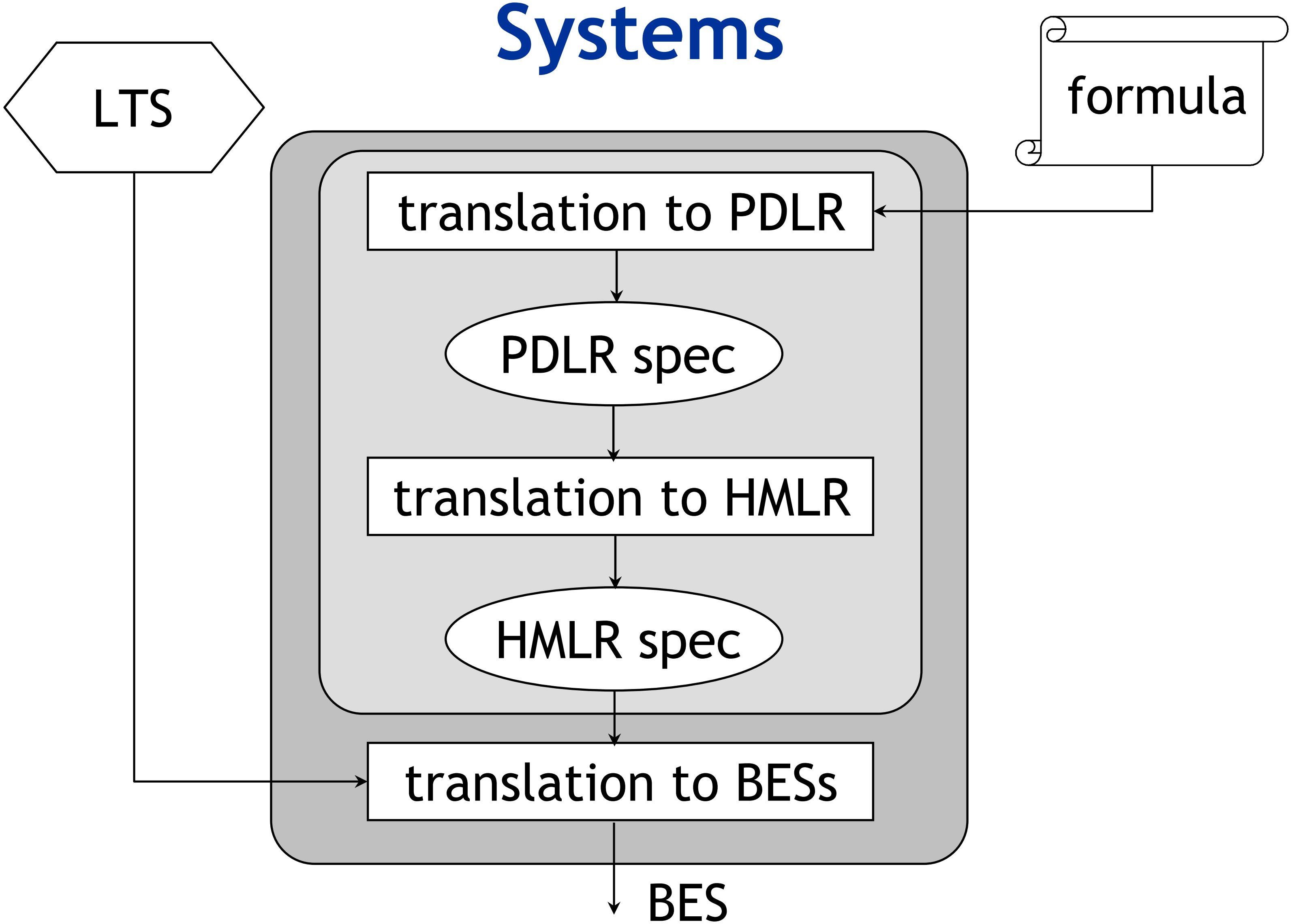
Model checking
(principle)



On-the-fly model checking in CADP
(Evaluator 3.x)



Translation to Boolean Equation Systems



Translation to PDL with recursion

- State formula (expanded):

  nu Y_0 . [true* . SEND]

  mu Y_1 . ⟨ true ⟩ true and [not RECV] Y_1

- PDLR specification [Mateescu-Sighireanu-03]:

$Y_0 =_{nu}$ [true* . SEND] Y_1

$Y_1 =_{mu}$ ⟨ true ⟩ true and [not RECV] Y_1

Simplification

- PDLR specification:

$$Y_0 =_{nu} [\ \mathrm{true}^* \ .\ \mathrm{SEND}\ ]\ Y_1$$

$$Y_1 =_{mu} \langle\ \mathrm{true}\ \rangle\ \mathrm{true\ and}\ [\ \mathrm{not\ RECV}\ ]\ Y_1$$

- Simple PDLR specification:

$$Y_0 =_{nu} [\ \mathrm{true}^* \ .\ \mathrm{SEND}\ ]\ Y_1$$

$$Y_1 =_{mu} Y_2\ \mathrm{and}\ Y_3$$

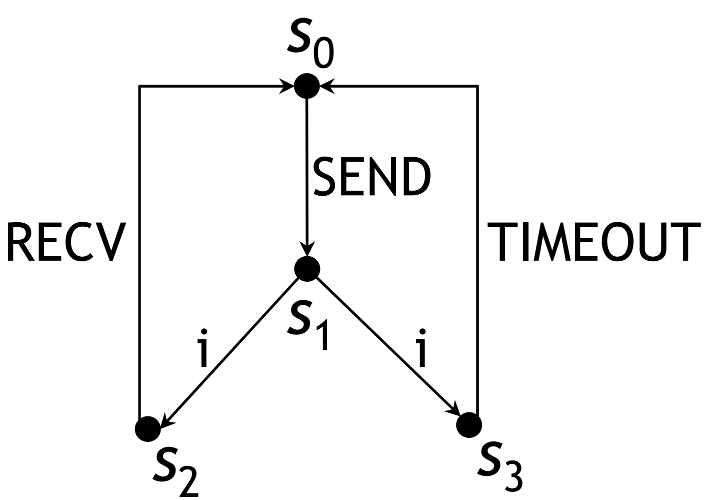
$$Y_2 =_{mu} \langle\ \mathrm{true}\ \rangle\ \mathrm{true}$$

$$Y_3 =_{mu} [\ \mathrm{not\ RECV}\ ]\ Y_1$$

Translation to BESs

$Y_0 =_{nu} Y_4$ and Y_5
$Y_4 =_{nu}$ [SEND] Y_1
$Y_5 =_{nu}$ [true] Y_0

$Y_1 =_{mu} Y_2$ and Y_3
$Y_2 =_{mu} \langle$ true $\rangle$ true
$Y_3 =_{mu}$ [not RECV] Y_1

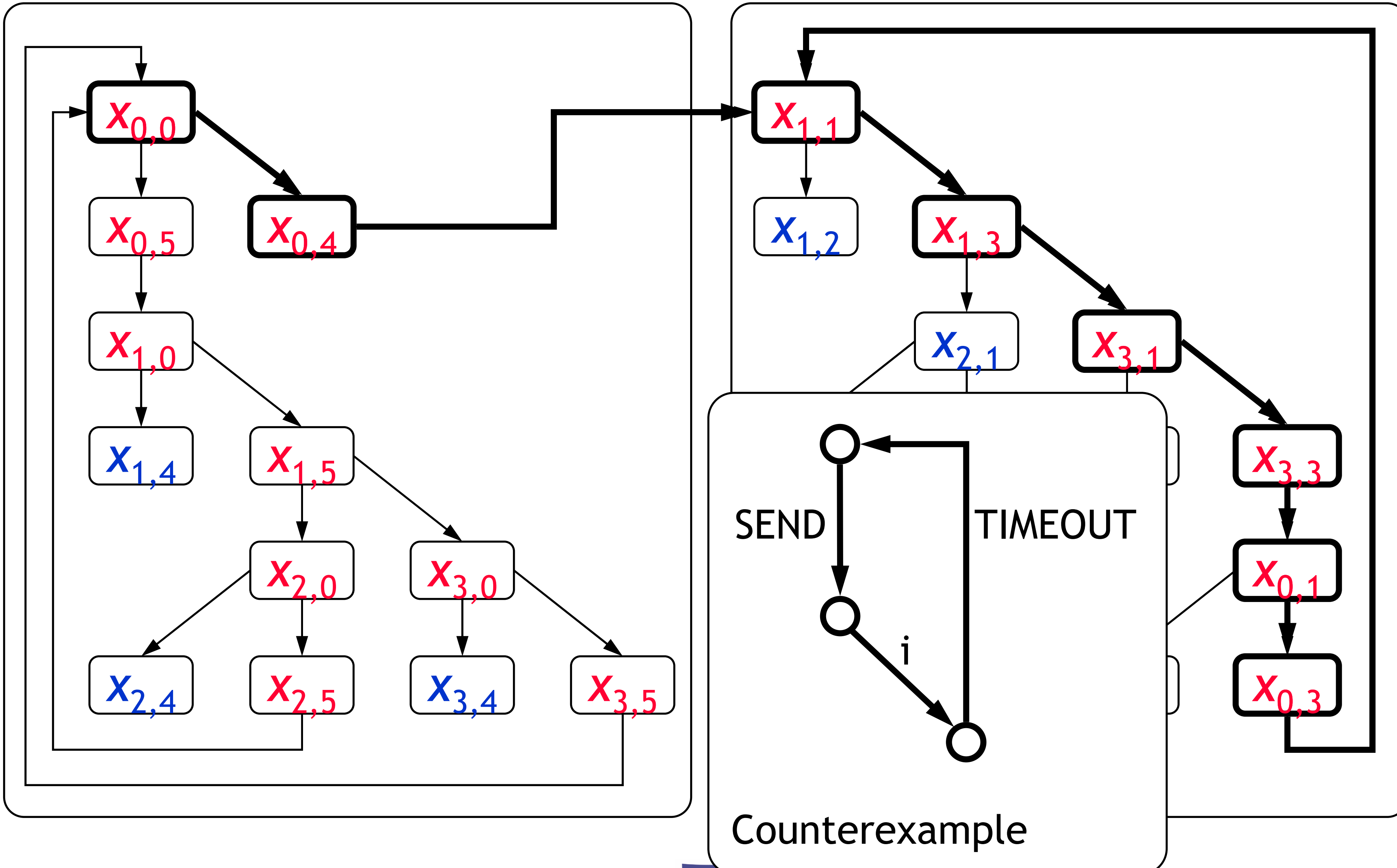


Boolean variables: $x_{i,\,j} \;\equiv\; s_i \models Y_j$

$$
\begin{aligned}
x_{0,0} &=_\nu x_{0,4} \wedge x_{0,5}\\
x_{0,4} &=_\nu x_{1,1}\\
x_{0,5} &=_\nu x_{1,0}\\
x_{1,0} &=_\nu x_{1,4} \wedge x_{1,5}\\
x_{1,4} &=_\nu \text{true}\\
x_{1,5} &=_\nu x_{2,0} \wedge x_{3,0}\\
x_{2,0} &=_\nu x_{2,4} \wedge x_{2,5}\\
x_{2,4} &=_\nu \text{true}\\
x_{2,5} &=_\nu x_{0,0}\\
x_{3,0} &=_\nu x_{3,4} \wedge x_{3,5}\\
x_{3,4} &=_\nu \text{true}\\
x_{3,5} &=_\nu x_{0,0}
\end{aligned}
$$

$$
\begin{aligned}
x_{1,1} &=_\mu x_{1,2} \wedge x_{1,3}\\
x_{1,2} &=_\mu \text{true}\\
x_{1,3} &=_\mu x_{2,1} \wedge x_{3,1}\\
x_{2,1} &=_\mu x_{2,2} \wedge x_{2,3}\\
x_{2,2} &=_\mu \text{true}\\
x_{2,3} &=_\mu \text{true}\\
x_{3,1} &=_\mu x_{3,2} \wedge x_{3,3}\\
x_{3,2} &=_\mu \text{true}\\
x_{3,3} &=_\mu x_{0,1}\\
x_{0,1} &=_\mu x_{0,2} \wedge x_{0,3}\\
x_{0,2} &=_\mu \text{true}\\
x_{0,3} &=_\mu x_{1,1}
\end{aligned}
$$

Local BES resolution with diagnostic



Additional operators

- Mechanisms for macro-definition (overloaded) and library inclusion
- Libraries encoding the operators of CTL and ACTL

  EU (φ_1, φ_2) = mu Y . φ_2 or $(\varphi_1$ and $\langle$ true $\rangle$ $Y)$

  EU $(\varphi_1, \alpha_1, \alpha_2, \varphi_2)$ = mu Y . $\langle \alpha_2 \rangle \varphi_2$ or $(\varphi_1$ and $\langle \alpha_1 \rangle$ $Y)$

- Libraries of high-level property patterns [Dwyer-99]
  - Property classes:
    - Absence, existence, universality, precedence, response
  - Property scopes:
    - Globally, before *a*, after *a*, between *a* and *b*, after *a* until *b*
  - More info:
    - http://www.inrialpes.fr/vasy/cadp/resources

Disjunctive BES

- *Disjunctive* boolean graph:
  - *Potentiality* operator of CTL

    $E\ [\varphi_1\ U\ \varphi_2] = \mu X\ .\ \varphi_2 \vee (\varphi_1 \wedge \langle\ T\ \rangle\ X)$

    $\{\ X =_\mu \varphi_2 \vee Y\ \ ,\ Y =_\mu \varphi_1 \wedge Z\ \ ,\ Z =_\mu \langle\ T\ \rangle\ X\ \}$

    $\{\ X_s =_\mu \varphi_{2s} \vee Y_s\ \ ,\ Y_s =_\mu \varphi_{1s} \wedge Z_s\ \ ,\ Z_s =_\mu \vee_{s \to s'} X_{s'}\ \}$
  - *Possibility* modality of PDL

    $\langle\ (a\ |\ b)^*\ .\ c\ \rangle\ T$

    $\{\ X =_\mu \langle\ c\ \rangle\ T \vee \langle\ a\ \rangle\ X \vee \langle\ b\ \rangle\ X\ \}$

    $\{\ X_s =_\mu (\vee_{s \to c\ s'}\ T) \vee (\vee_{s \to a\ s'}\ X_{s'}) \vee (\vee_{s \to b\ s'}\ X_{s'})\ \}$
- Algorithm A3 (memory ↓)

Linear-time model checking

(looping operator of PDL-delta)

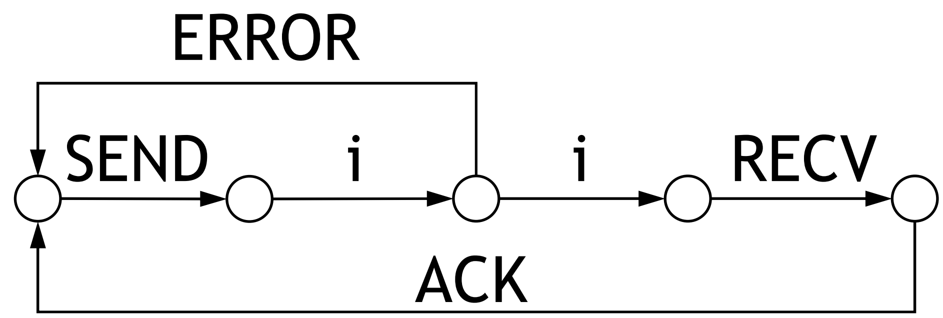
- Translation in mu-calculus of alternation depth 2 [Emerson-Lei-86]:

  $< R > @ = \text{nu } X \;.\; < R > X$

  if R contains *-operators, the formula is of alternation depth 2

- But still checkable in linear-time:
  - Mark LTS states potentially satisfying X
  - Leads to marked variables in the disjunctive BES
  - Computation of boolean SCCs containing marked variables
  - $A3_{cyc}$ algorithm [Mateescu-Thivolle-08]
    - Can serve for LTL model checking
    - Allows linear-time handling of repeated invocations

Model checking of data-based properties
(Evaluator 4.0)



- Every SEND is followed by a RECV after 2 steps:

```
[ true* . SEND ] < true { 2 } . RECV > true =
nu X . ( [ SEND ] mu Y (c:Nat := 2) .
                         if c = 0 then < RECV > true
                              else < true > Y (c - 1)
                         end if
      and
      [ true ] X )
```

Translation into HMLR

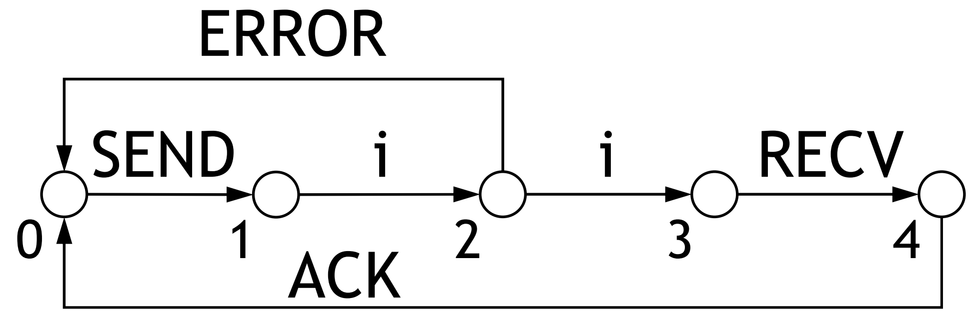
```
nu X . [ SEND ]

        and [ true ] X
```

```
mu Y (c:Nat := 2) .
   if c = 0 then < RECV > true
          else < true > Y (c - 1)
end if
```

```
{ X =nu
      [ SEND ] Y (2)
      and
      [ true ] X
}
```

```
{ Y (c:Nat) =mu
      if c = 0 then < RECV > true
         else < true > Y (c - 1)
      end if
}
```

Translation into BES and resolution



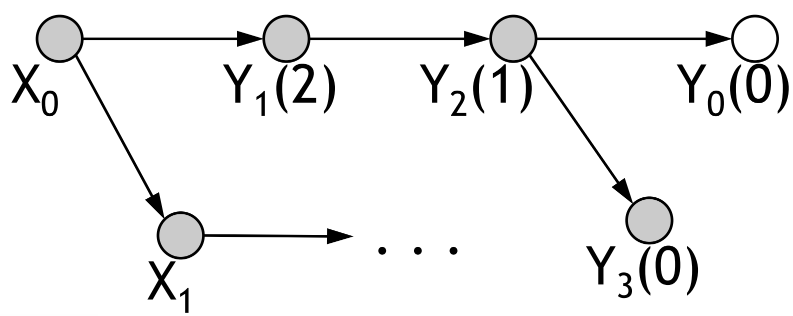

```
{ X =nu
      [ SEND ] Y (2)
      and
      [ true ] X
}
```

```
{ Y (c:Nat) =mu
      if c = 0 then < RECV > true
         else < true > Y (c - 1)
      end if
}
```

- Principle:

$X_s \;=\; « \; s \;|= X \; »$

$Y_s\ (c) \;=\; « \; s \;|= Y\ (c) \; »$

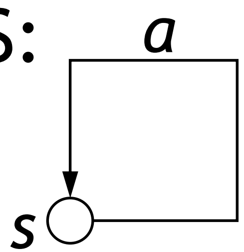


Divergence

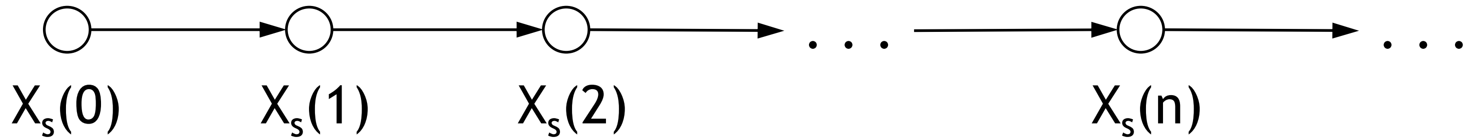
- In presence of data parameters of infinite types, termination of model checking is not guaranteed anymore
- (pathological) property:

  mu X (n:Nat := 0) . < *a* > X (n + 1)

LTS:



- BES : $\{ X_s \text{ (n:Nat)} =_{mu} \text{OR}_{s \to_a s'} X_{s'} \text{ (n + 1) } \} = \{ X_s \text{ (n:Nat)} =_{mu} X_s \text{ (n + 1) } \}$



Conjunctive BES

- *Conjunctive* boolean graph:
  - *Inevitability* operator of CTL

    $A\,[\varphi_1 \;U\; \varphi_2] = \mu X \,.\, \varphi_2 \vee (\varphi_1 \wedge \langle\, T \,\rangle\, T \wedge [\, T \,]\, X)$

    $\{\, X =_\mu \varphi_2 \vee Y \;\;,\; Y =_\mu \varphi_1 \wedge Z \wedge [\, T \,]\, X \,,\; Z =_\mu \langle\, T \,\rangle\, T \,\}$

    $\{\, X_s =_\mu \varphi_{2s} \vee Y_s \;\;,\; Y_s =_\mu \varphi_{1s} \wedge Z_s \wedge (\wedge_{s \to s'} X_{s'}) \;,\; Z_s =_\mu \vee_{s \to s'} T \,\}$
  - *Necessity* modality of PDL

    $[\, (a \mid b)^* \,.\, c \,]\, F$

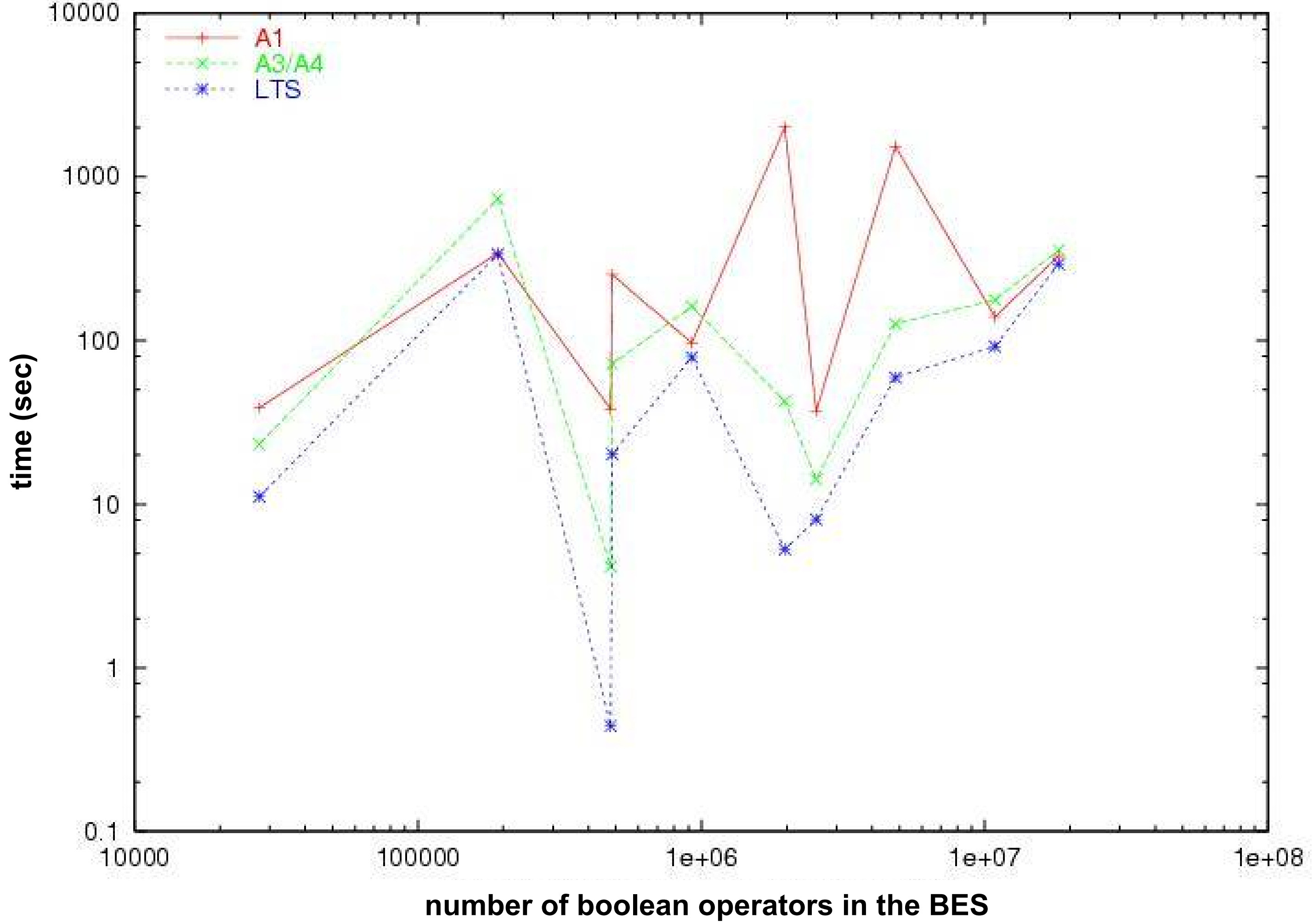
    $\{\, X =_\mu [\, c \,]\, F \wedge [\, a \,]\, X \wedge [\, b \,]\, X \,\}$

    $\{\, X_s =_\mu (\wedge_{s \to c\, s'} F) \wedge (\wedge_{s \to a\, s'} X_{s'}) \wedge (\wedge_{s \to b\, s'} X_{s'}) \,\}$
- Algorithm A4 (memory ↓)

Acyclic BES

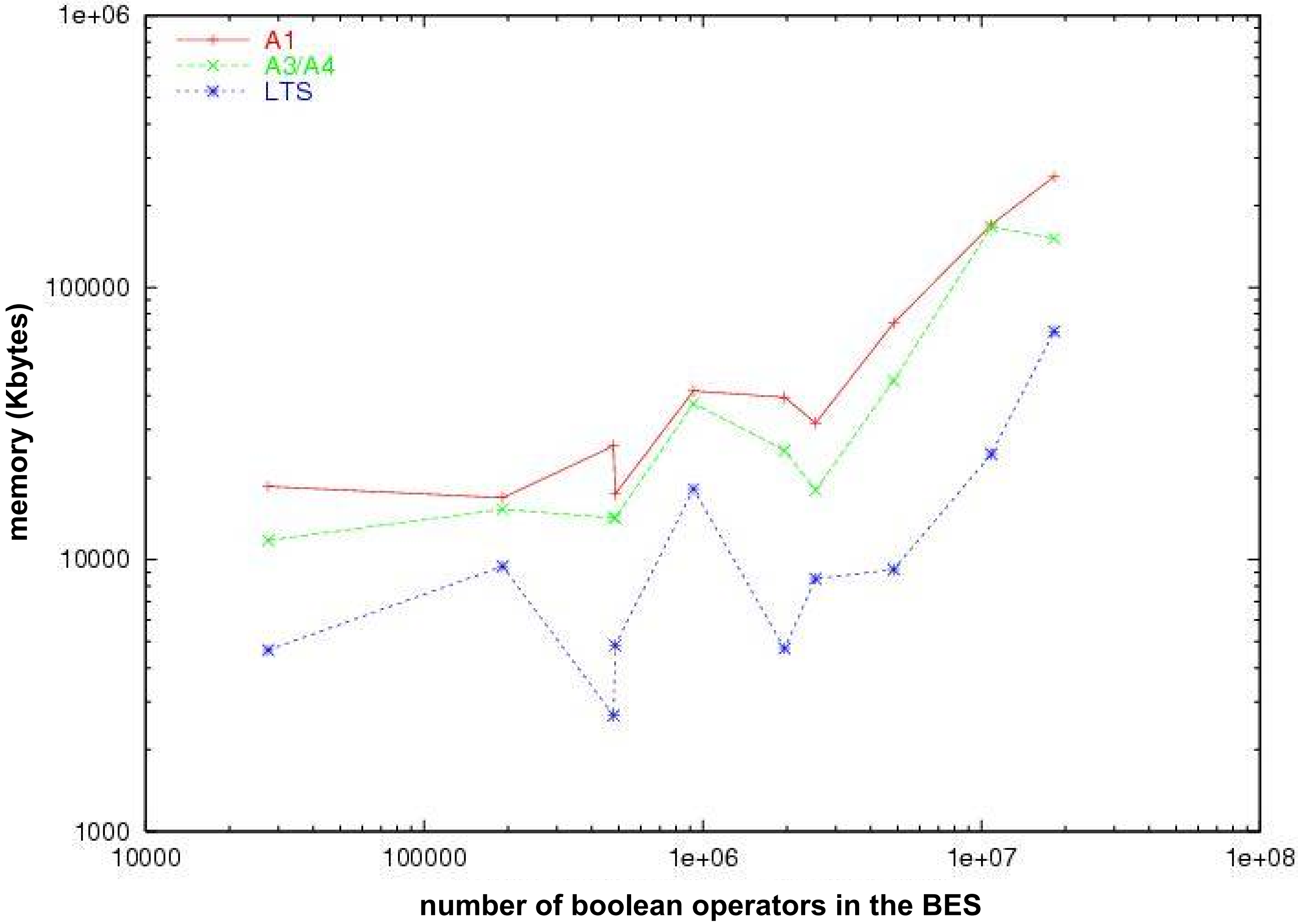
- *Acyclic* boolean graph:
  - *Acyclic* LTS and *guarded* formulas [Mateescu-02]
- Handling of CTL (and ACTL) operators:
  - $E\,[\varphi_1\ U\ \varphi_2] = \mu X\ .\ \varphi_2 \vee (\varphi_1 \wedge \langle\ T\ \rangle\ X)$
  - $A\,[\varphi_1\ U\ \varphi_2] = \mu X\ .\ \varphi_2 \vee (\varphi_1 \wedge \langle\ T\ \rangle\ T \wedge [\ T\ ]\ X)$
- Handling of full mu-calculus
  - Translation to guarded form
  - Conversion from maximal to minimal fixed points [Mateescu-02]
- Algorithm A2 (memory $\downarrow$)

Algorithm A1 vs. A3/A4
(execution time – CADP demos)

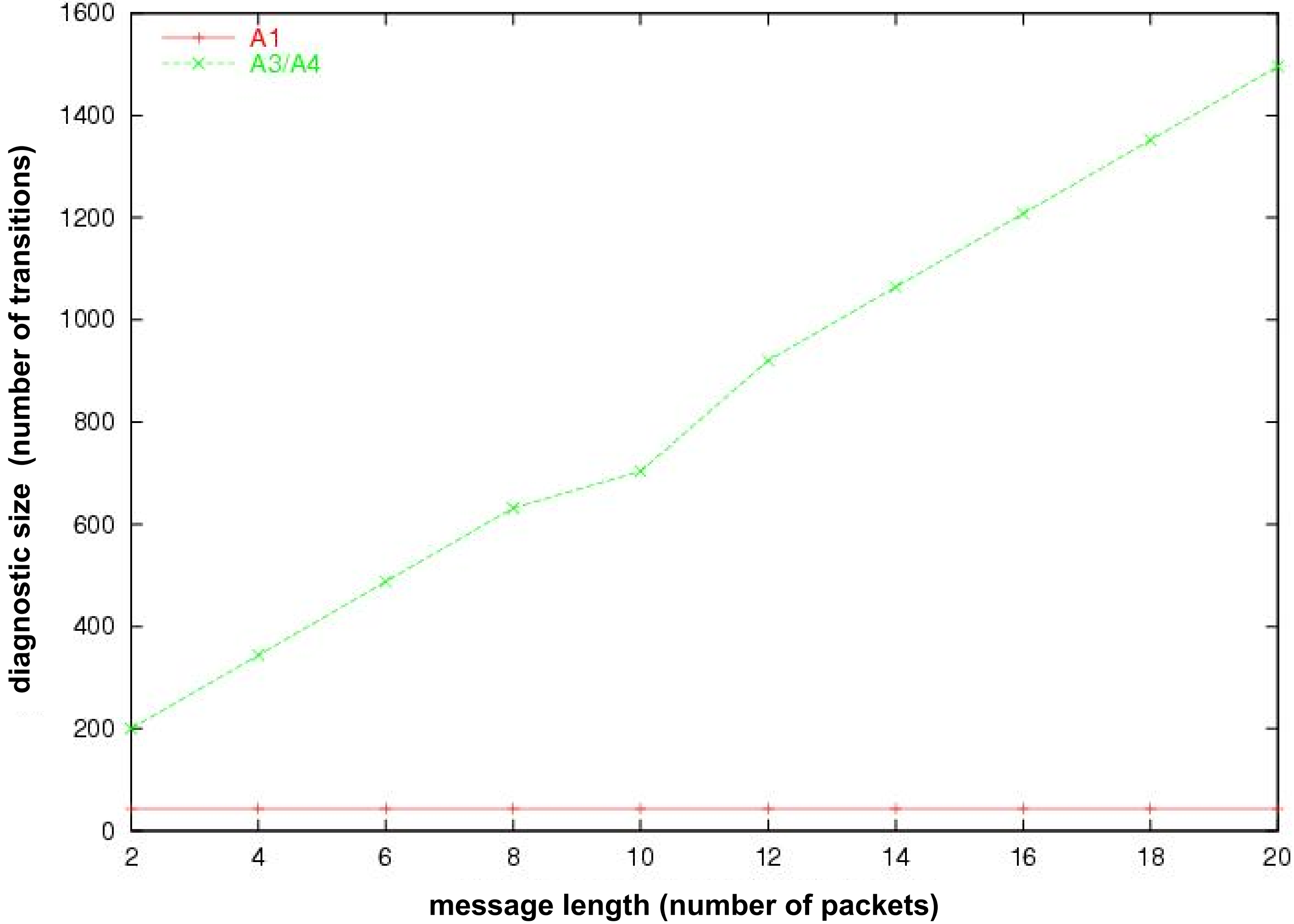


Algorithm A1 vs. A3/A4

(memory consumption – CADP demos)



Algorithm A1 vs. A3/A4
(diagnostic size – BRP protocol)



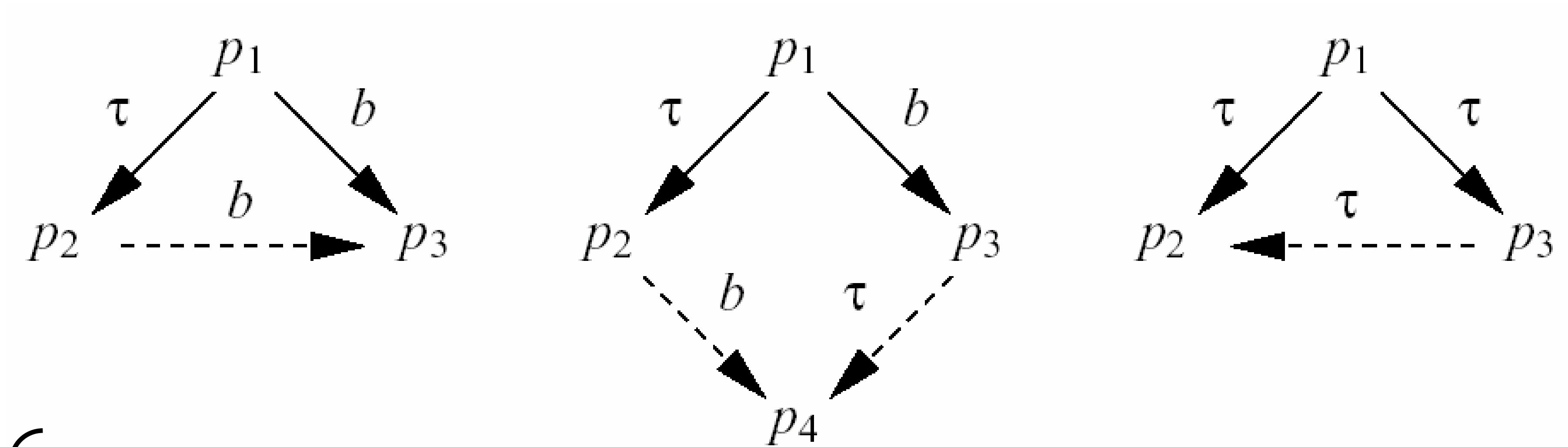
Model checking
(summary)

- *General* boolean graph:
  - Any LTS and any alternation-free μ-calculus formula
  - Algorithms A0 and A1 (diagnostic depth ↓)
- *Acyclic* boolean graph:
  - Acyclic LTS and guarded formula (CTL, ACTL)
  - Acyclic LTS and μ-calculus formula (via reduction)
  - Algorithm A2 (memory ↓)
- *Disjunctive/conjunctive* boolean graph:
  - Any LTS and any formula of CTL, ACTL, PDL
  - Algorithm A3/A4 (memory ↓)
  - Matches the best local algorithms dedicated to CTL [Vergauwen-Lewi-93]

Partial order reduction

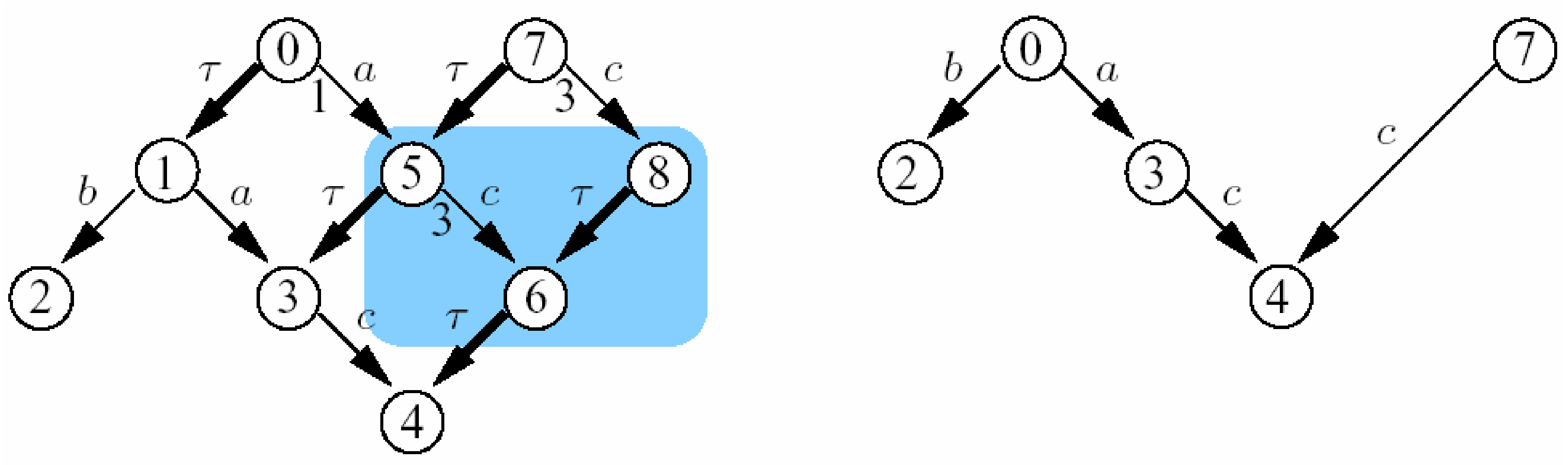
- *τ-confluence* [Groote-vandePol-00]
  - Form of partial-order reduction defined on LTSs
  - Preserves branching bisimulation
- Principle
  - Detection of τ-confluent transitions
  - Elimination of "neighbour" transitions (*τ-prioritisation*)
- On-the-fly LTS reduction
  - Direct approach [Blom-vandePol-02]
  - BES-based approach [Pace-Lang-Mateescu-03]
    - Define τ-confluence in terms of a BES
    - Detect τ-confluent transitions by locally solving the BES
    - Apply τ-prioritisation and compression on sequences

Translation to a BES



$$
\left\{
\begin{array}{l}
X_{p1,p2} =_{\nu} \bigwedge_{p1 \to_b p3} (\\
\qquad p2 \to_b p3 \vee \\
\qquad \bigvee_{p2 \to_b p4,\ p3 \to_\tau p4} X_{p3,p4} \vee \\
\qquad ((b = \tau) \wedge \bigvee_{p3 \to_\tau p2} X_{p3,p2}) \\
\quad)
\end{array}
\right.
$$

Tau-prioritisation and compression



Original LTS

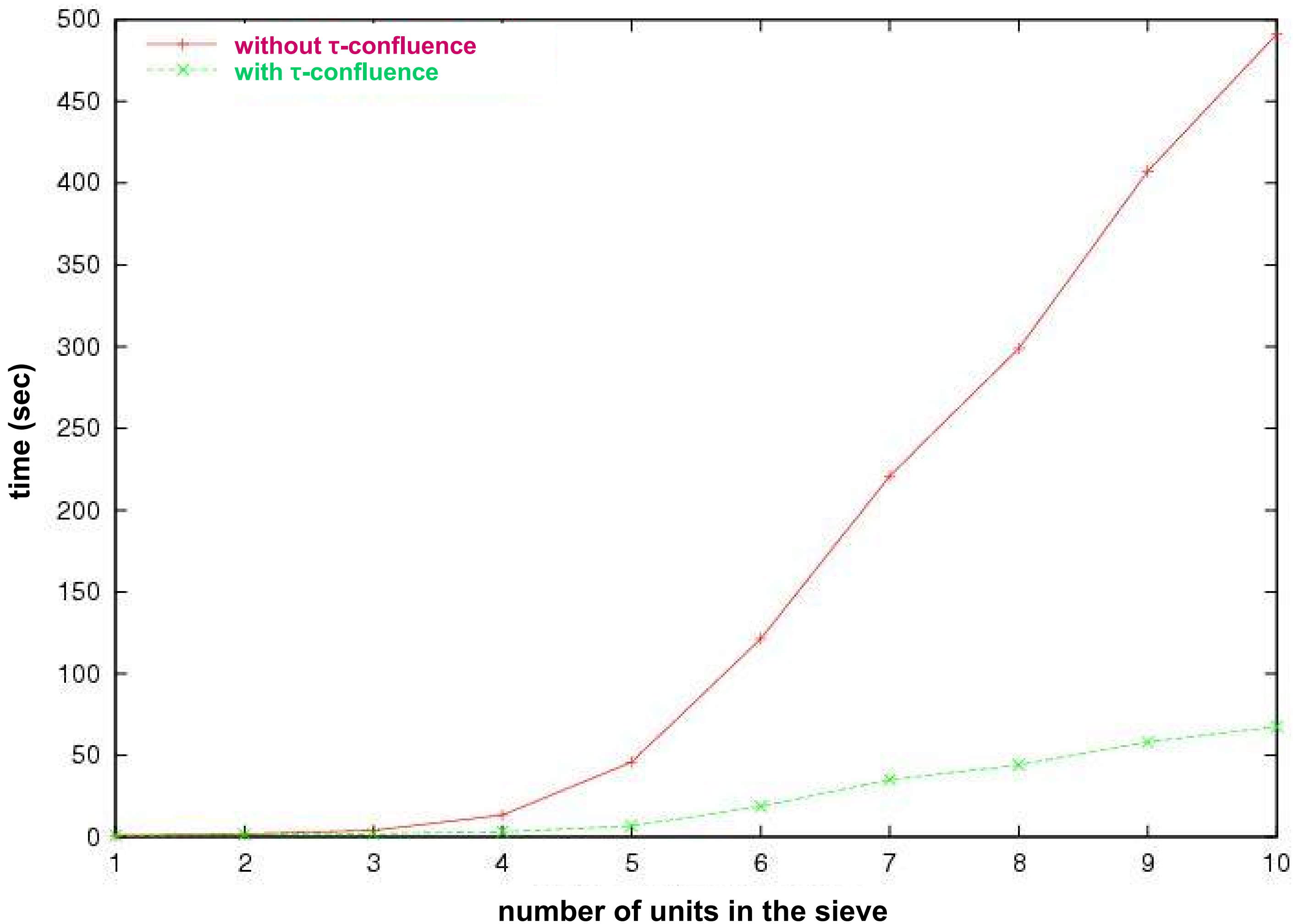
Reduced LTS

(exploration from s_0 and s_7)

- In practice: reductions of a factor 10^2 – 10^3 [Mateescu-05]

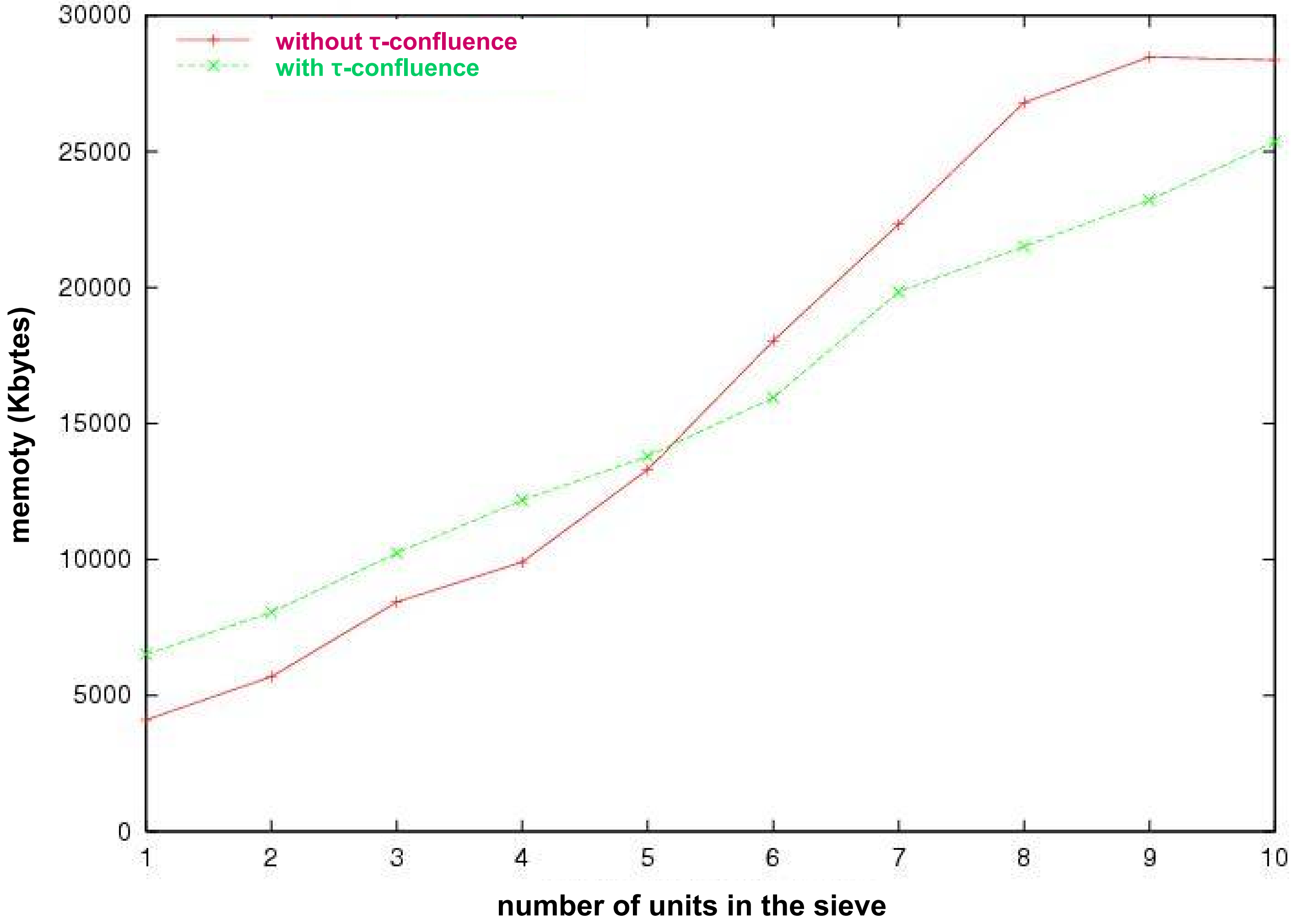
Model checking using A3/A4

(effect of τ-confluence reduction – time – Erathostene's sieve)



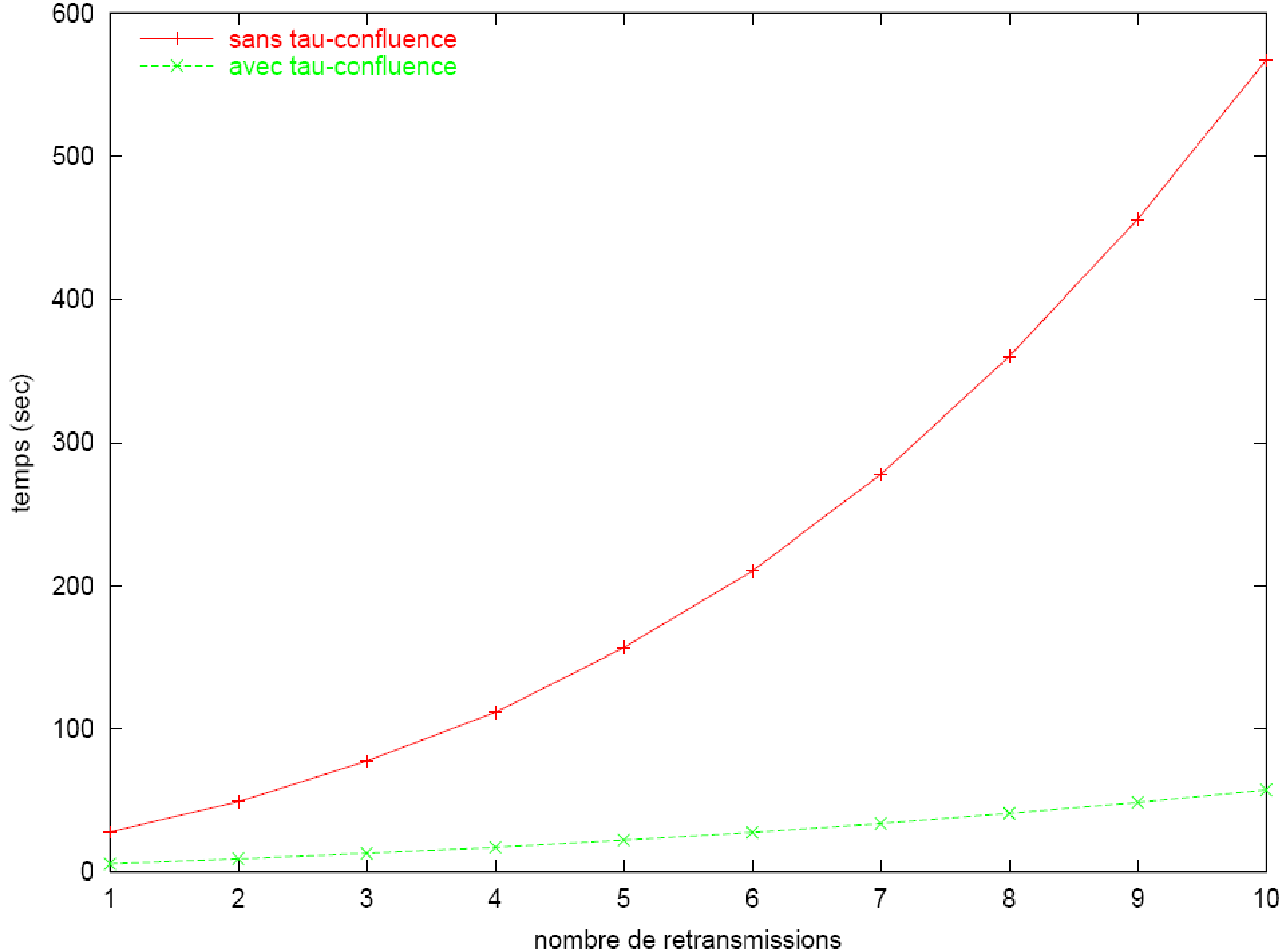
Model checking using A3/A4

(effect of τ-confluence reduction – memory – Erathostene's sieve)



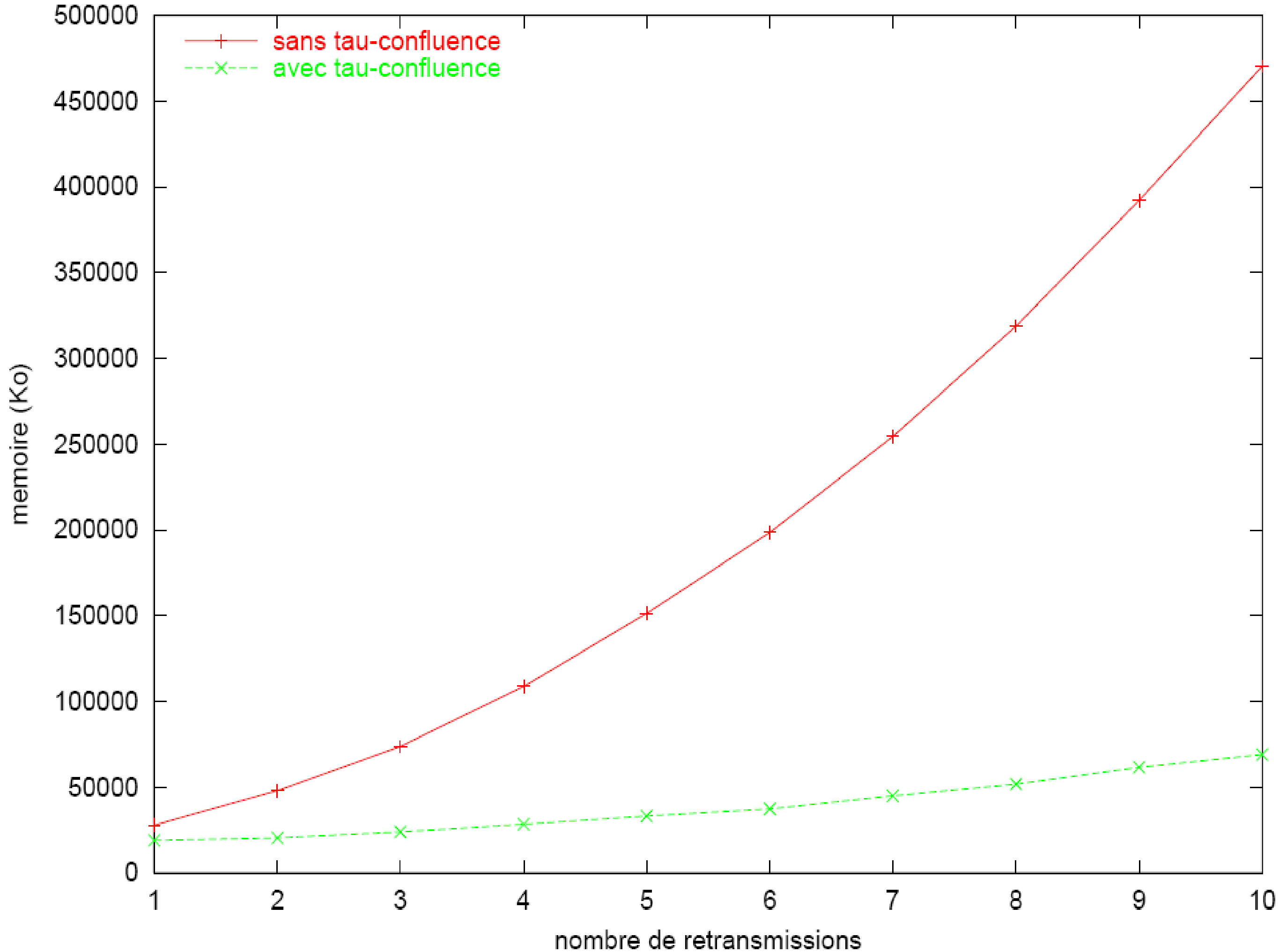
Checking branching bisimulation

(effect of τ-confluence reduction – time – BRP protocol)



Checking branching bisimulation

(effect of τ-confluence reduction – memory – BRP protocol)



On-the-fly verification
(summary)

Already available:

- Generic Caesar_Solve library [Mateescu-03,06]
- 9 local BES resolution algorithms (A8 added in 2008)
- Diagnostic generation features
- Applications: Bisimulator, Evaluator 3.5, Reductor 5.0

Ongoing:

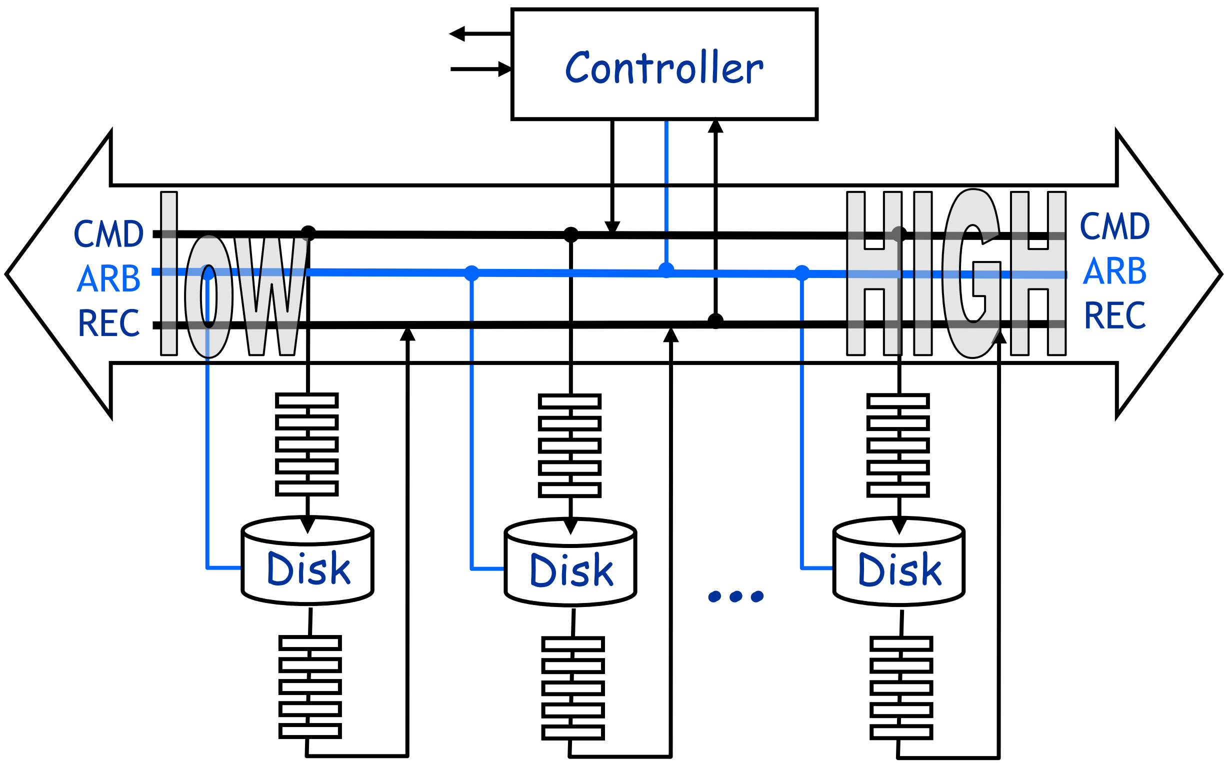
- Distributed BES resolution algorithms on clusters of machines [Joubert-Mateescu-04,05,06]
- New applications
  - Test generation
  - Software adaptation
  - Discrete controller synthesis

Case study

- SCSI-2 bus arbitration protocol
- Description in LOTOS
- Specification of properties in TL
- Verification using Evaluator 3.5 and 4.0
- Interpretation of diagnostics

SCSI-2 bus arbitration protocol

- Prioritized arbitration mechanism, based on static IDs on bus (devices numbered from 0 to n – 1)
- Fairness problem (starvation of low-priority disks)



Architecture of the system

```
(
    DISK [ARB, CMD, REC] (0, 0)
    |[ARB]|
    DISK [ARB, CMD, REC] (1, 0)
    |[ARB]|
    ...
    |[ARB]|
    DISK [ARB, CMD, REC] (6, 0)
)
|[ARB, CMD, REC]|
CONTROLLER [ARB, CMD, REC] (NC, ZERO)
```

8-ary rendezvous on gate ARB

binary rendezvous on gates CMD, REC

Synchronization constraints
(bus arbitration policy)

- Synchronizations on gate ARB:

  ARB ?r0, …,r7:Bool [C (r0, …, r7, n)] ; …

  where:
  - r0, …, r7 = values of the electric signals on the bus
  - n = index of the current device

- Two particular cases for guard condition C:
  - P (r0, …, r7, n): device n does not ask the bus
  - A (r0, …, r7, n): device n asks and obtains access to bus

Guard conditions

- Predicate P (r0, ..., r7, n) = $\neg r_n$

  P (r0, ..., r7, 0) = not (r0)

  P (r0, ..., r7, 1) = not (r1)

  ...

  P (r0, ..., r7, 7) = not (r7)

- Predicate A (r0, ..., r7, n) = $r_n \wedge \forall i \in [n+1, 7] \, . \, \neg r_i$

  A (r0, ..., r7, 0) = r0 and not (r1 or ... or r7)

  A (r0, ..., r7, 1) = r1 and not (r2 or ... or r7)

  ...

  A (r0, ..., r7, 7) = r7

Controller process

```
process Controller [ARB, CMD, REC] (C:Contents) : noexit :=
    (* communicate with disk N *)
    choice N:Nat []
        [(N >= 0) and (N <= 6)] ->
                Controller2 [ARB, CMD, REC] (C, N)
    []
    (* does not request the bus *)
    ARB ?r0, ..., r7:Bool [P (r0, ..., r7, 7)];
        Controller [ARB, CMD, REC] (C)
endproc
```

Controller process

```
process Controller2 [ARB, CMD, REC] (C:Contents, N:Nat) :
noexit :=
   [not_full (C, N)] ->
       (* request and obtain the bus *)
       ARB ?r0, ..., r7:Bool [A (r0, ..., r7, 7)];
           CMD !N; (* send a command *)
               Controller [ARB, CMD, REC] (incr (C, N))
   []
   REC !N;    (* receive an acknowledgement *)
       Controller [ARB, CMD, REC] (decr (C, N))
endproc
```

Disk process

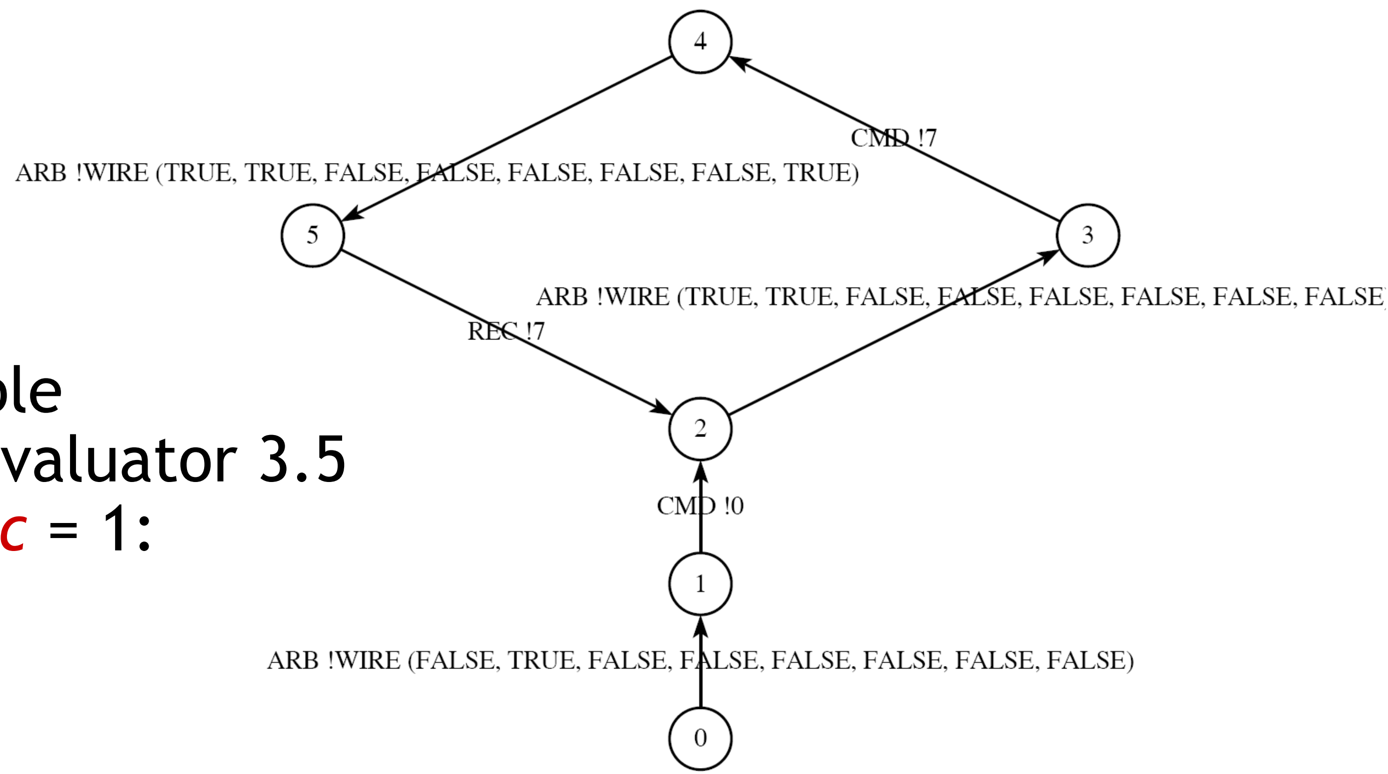
```
process DISK [ARB, CMD, REC] (N, L:Nat) : noexit :=
   CMD !N;  DISK [ARB,CMD,REC] (N, L+1)
   []
   [L > 0] -> (
        ARB ?r0, ..., r7:Bool [A (r0, ..., r7, N)];
                REC !N;  DISK [ARB, CMD, REC] (N, L-1)
        []
        ARB ?r0, ..., r7:Bool [not (A (r0, ..., r7, N)) and
                                  not (P (r0, ..., r7, N))];
                DISK [ARB, CMD, REC] (N, L)
   )
   []
   [L = 0] ->  ARB ?r0, ..., r7:Bool [P (r0, ..., r7, N)];
                        DISK [ARB, CMD, REC] (N, L)
endproc
```

Absence of starvation property
(PDL+ACTL formulation)

"Every time a disk i *receives a command from the controller, it will be able to gain access to the bus in order to send the corresponding acknowledgement"*

$$[\ true^* \ .\ cmd_i\ ]\ A\ [\ true_{true}\ U_{reci}\ true\ ]$$

- Property fails for *i* < *nc*
- Counterexample produced by Evaluator 3.5 for *i* = 0 and *nc* = 1:



Starvation property
(MCL formulation)

"Every time a disk i with priority lower than the controller nc receives a command, its access to the bus can be continuously preempted by any other disk j with higher priority"

```
[ true*. {cmd ?i:Nat where i < nc} ]
        forall j:Nat among { i + 1 ... n − 1 } .
                (j <> nc) implies
                < (not {rec !i})*. {cmd !j} .
                   (not {rec !i})*. {rec !j} > @
```

Safety property

(MCL formulation)

"The difference between the number of commands received and reconnections sent by a disk i varies between 0 and 8 (the size of the buffers associated to disks)"

```
forall i:Nat among { 0 … n – 1 } .
    nu Y (c:Nat:=0) . (
        [ {cmd !i} ] ((c < 8) and Y (c + 1))
        and
        [ {rec !i} ] ((c > 0) and Y (c – 1))
        and
        [ not ({cmd !i} or {rec !i}) ] Y (c)
    )
```

Safety property
(standard mu-calculus formulation)

```
nu CMD_REC_0 . (
    [ CMD_i ] nu CMD_REC_1 . (
      [ CMD_i ] nu CMD_REC_2 . (
        [ CMD_i ] nu CMD_REC_3 . (
          [ CMD_i ] nu CMD_REC_4 . (
            [ CMD_i ] nu CMD_REC_5 . (
              [ CMD_i ] nu CMD_REC_6 . (
                [ CMD_i ] nu CMD_REC_7 . (
                  [ CMD_i ] nu CMD_REC_8 . (
                    [ CMD_i ] false
                    and
                    [ REC_i ] CMD_REC_7
                    and
                    [ not ((CMD_i) or (REC_i)) ] CMD_REC_8
                  )
                  and
                  [ REC_i ] CMD_REC_6
                  and
                  [ not ((CMD_i) or (REC_i)) ] CMD_REC_7
                )
                and
                [ REC_i ] CMD_REC_5
                and
                [ not ((CMD_i) or (REC_i)) ] CMD_REC_6
              )
              and
              [ REC_i ] CMD_REC_4
              and
              [ not ((CMD_i) or (REC_i)) ] CMD_REC_5
            )
            and
            [ REC_i ] CMD_REC_3
            and
            [ not ((CMD_i) or (REC_i)) ] CMD_REC_4
          )
          and
          [ REC_i ] CMD_REC_2
          and
          [ not ((CMD_i) or (REC_i)) ] CMD_REC_3
        )
        and
        [ REC_i ] CMD_REC_1
        and
        [ not ((CMD_i) or (REC_i)) ] CMD_REC_2
      )
      and
      [ REC_i ] CMD_REC_0
      and
      [ not ((CMD_i) or (REC_i)) ] CMD_REC_1
    )
    and
    [ REC_i ] false
    and
    [ not ((CMD_i) or (REC_i)) ] CMD_REC_0
)
```

Discussion and perspectives

- Model-based verification techniques:
  - Bug hunting, useful in early stages of the design process
  - Confronted with (very) large models
  - Temporal logics extended with data (XTL, Evaluator 4.0)
  - Machinery for on-the-fly verification (Open/Caesar)

- Perspectives:
  - Parallel and distributed algorithms
    - State space construction
    - BES resolution
  - New applications
    - Analysis of genetic regulatory networks