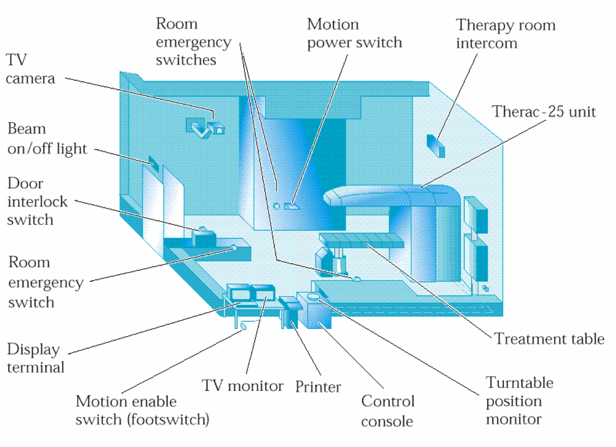
Model Checking of Action-Based Concurrent Systems

Radu Mateescu

INRIA Rhône-Alpes / VASY

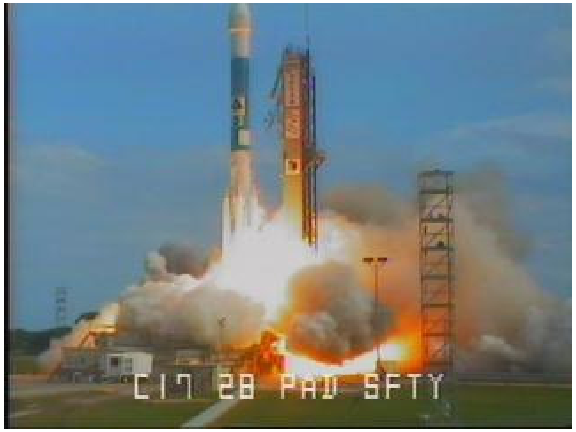
http://www.inrialpes.fr/vasy

Why formal verification?



Therac-25 radiotherapy accidents (1985-1987)

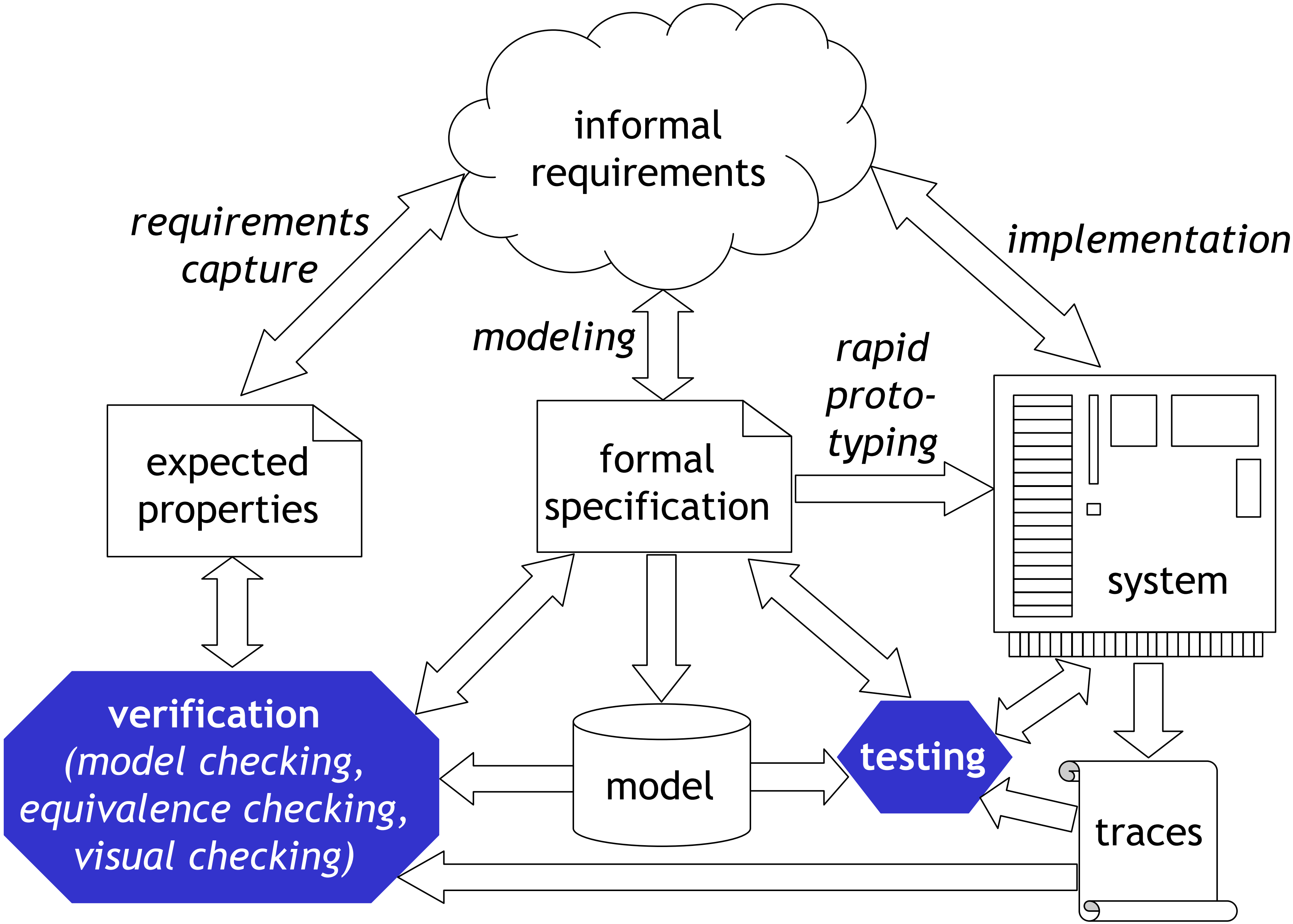
Ariane-5 launch failure (1996)



Mars climate orbiter failure (1999)

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

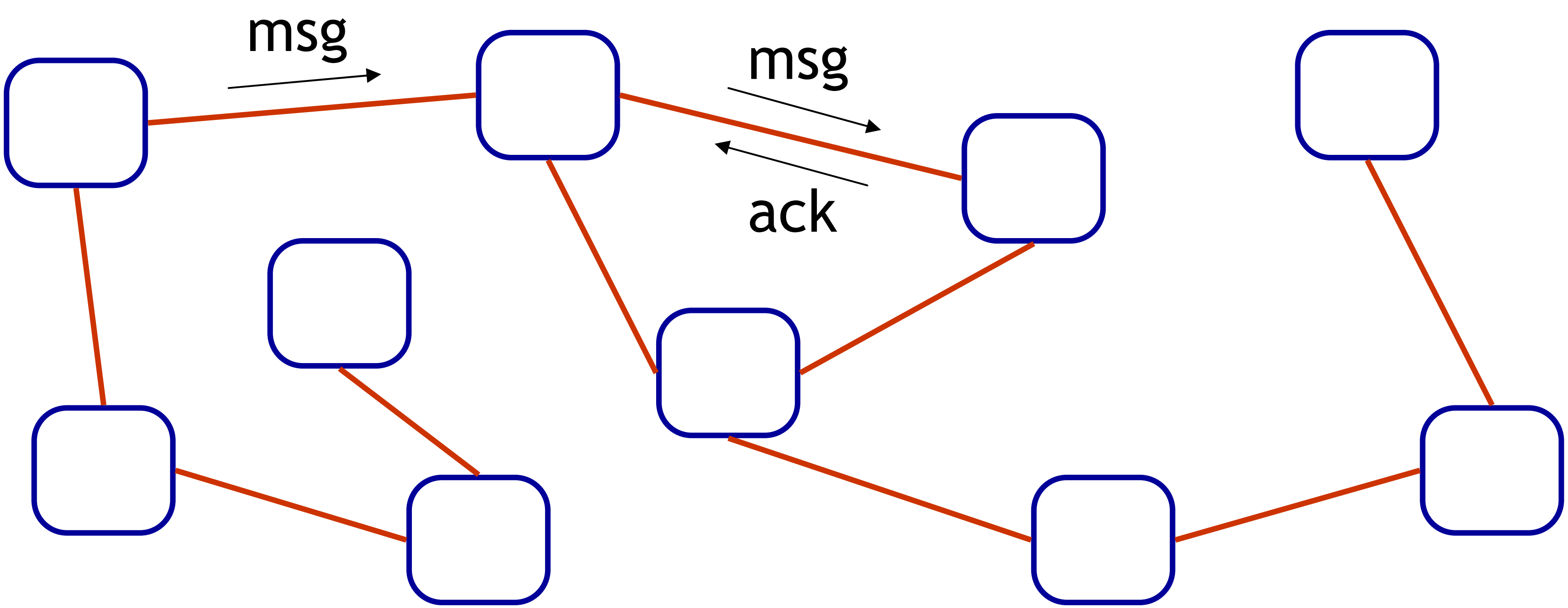
➔ *formal verification is useful for early error detection*


informal requirements
requirements capture
modeling
implementation
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, ...)

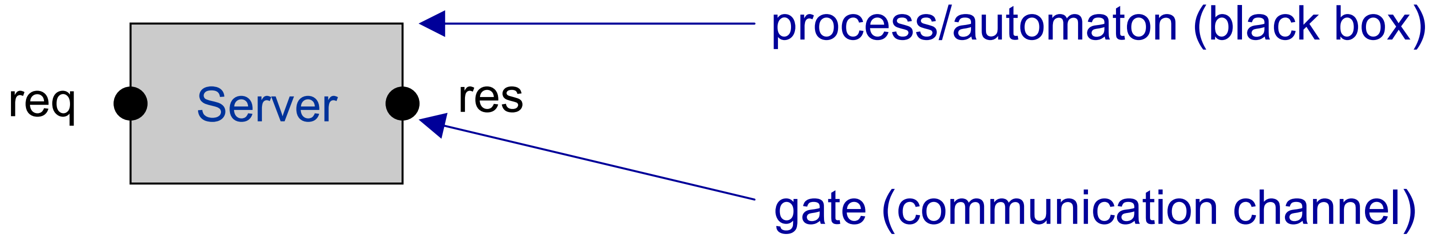
Reactive systems

- 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

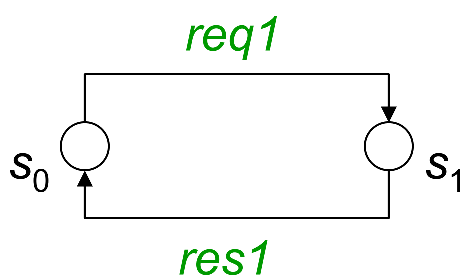
- Simple formalism describing the behaviour of concurrent systems
- *Black-box* approach:
  - One cannot inspect directly the state of the system
  - The behaviour of the system can be known only through its interactions with the environment



- 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*
- Example:
  process client$_1$



internal action (noted i or τ)

every state is reachable from the initial state

deadlock (sink) state: no outgoing transitions

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

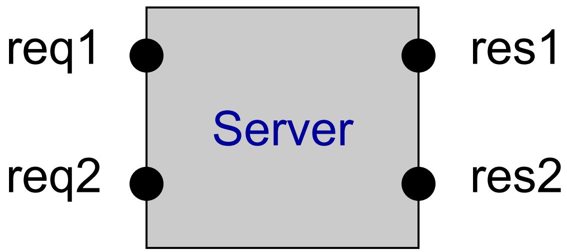
- List of transitions
- Allows forward and backward exploration
- Suitable for global verification
- BCG (Binary Coded Graphs) environment
  - API in C for reading/writing
  - Tools and libraries for explicit graph manipulation (**bcg**_io, **bcg**_draw, **bcg**_info, **bcg**_edit, **bcg**_labels, ...)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

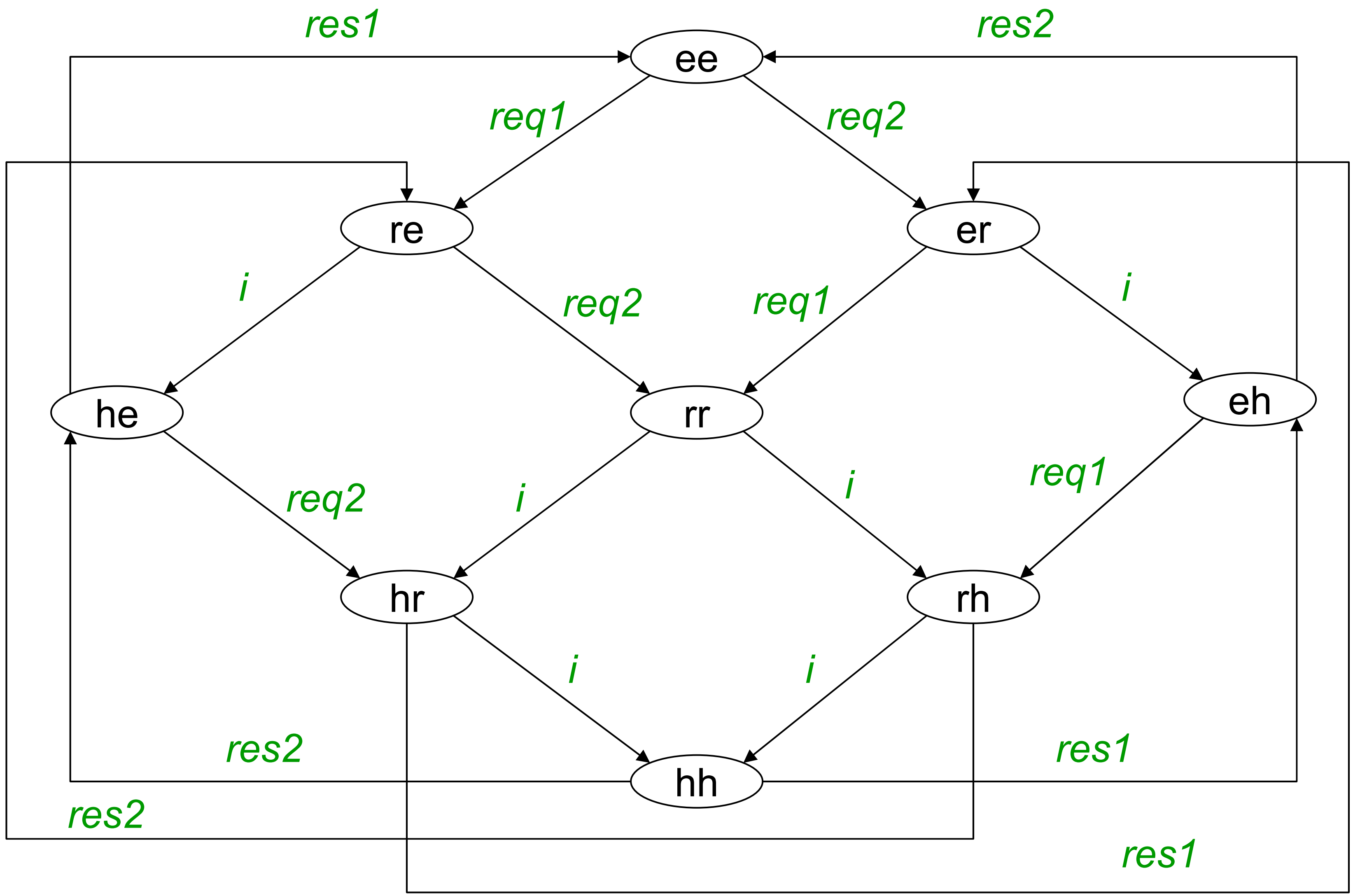
Server example
(modeled using a single automaton)

- 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:

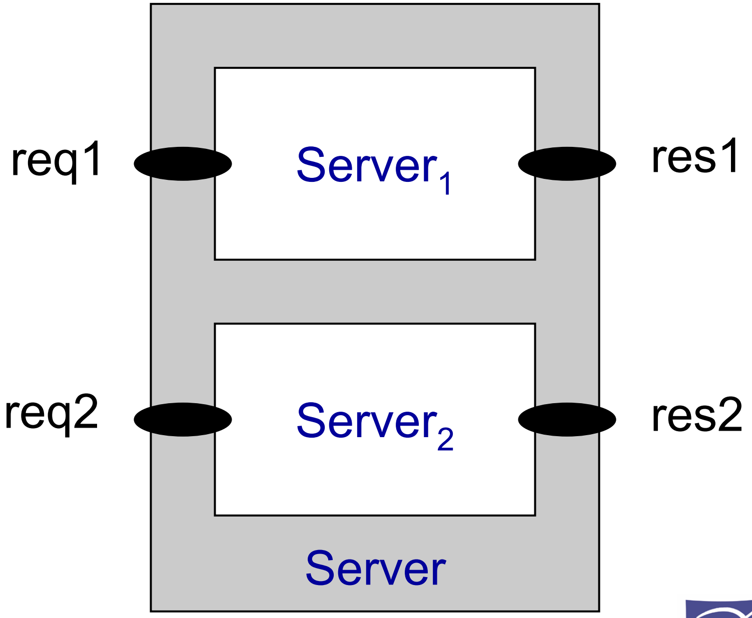
  $$| u_1 | * | u_2 | = 3 * 3 = 9$$

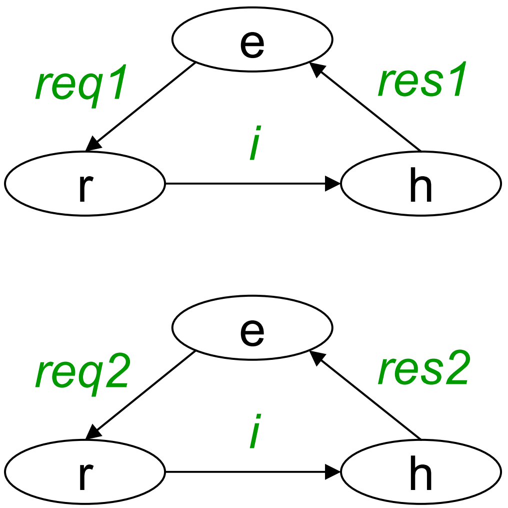
  (no synchronization between request processings)
- There is no sink state (the system is *deadlock-free*)
- From every state, it is possible to reach the initial state again (the server can be re-initialized)
- Shortcomings of modeling with a single automaton:
  - One must predict all the possible request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

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

    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.

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

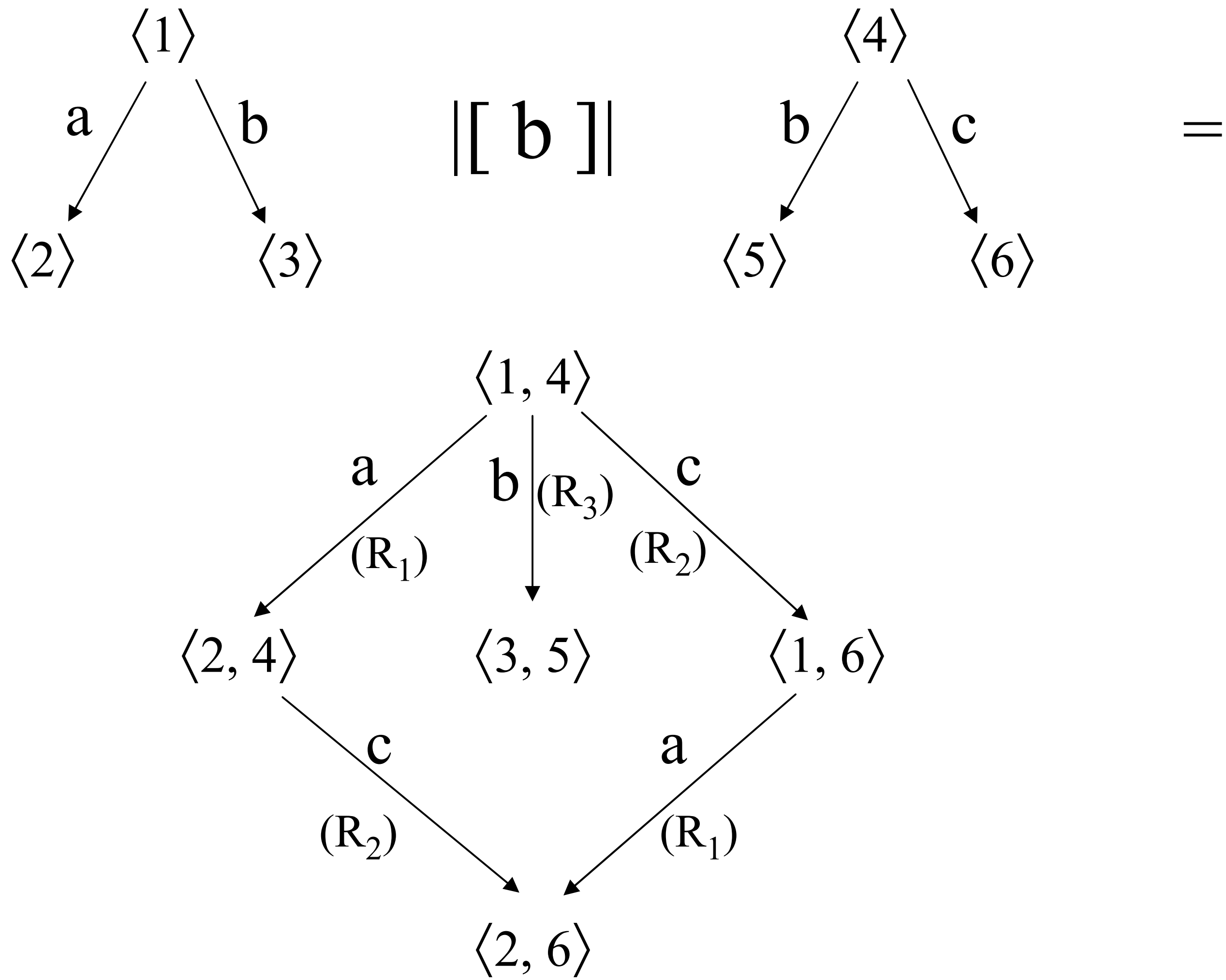
- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $s_0 = \langle s_{01}, s_{02}\rangle$
- $T \subseteq S \times A \times S$ defined by R_1-R_3

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

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

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

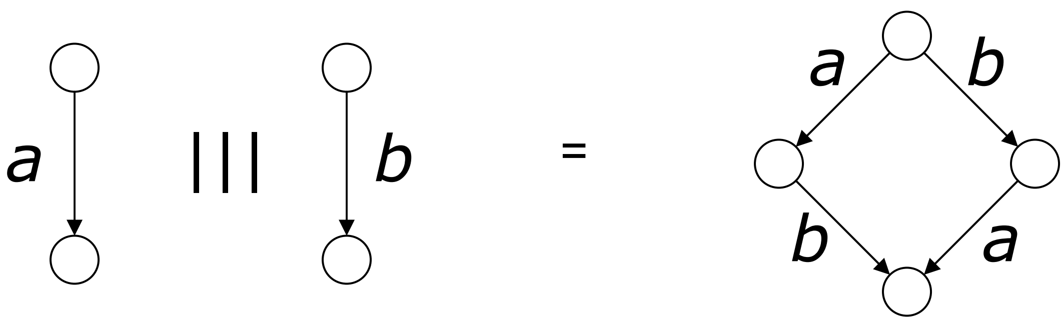
Example



Interleaving semantics

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

➔ *suitable paradigm for distributed systems*

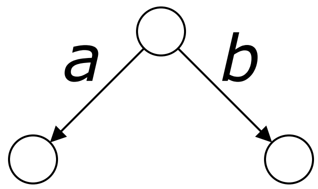


interleaving lozenge

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

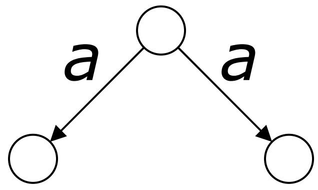
Internal and external choice

- *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.

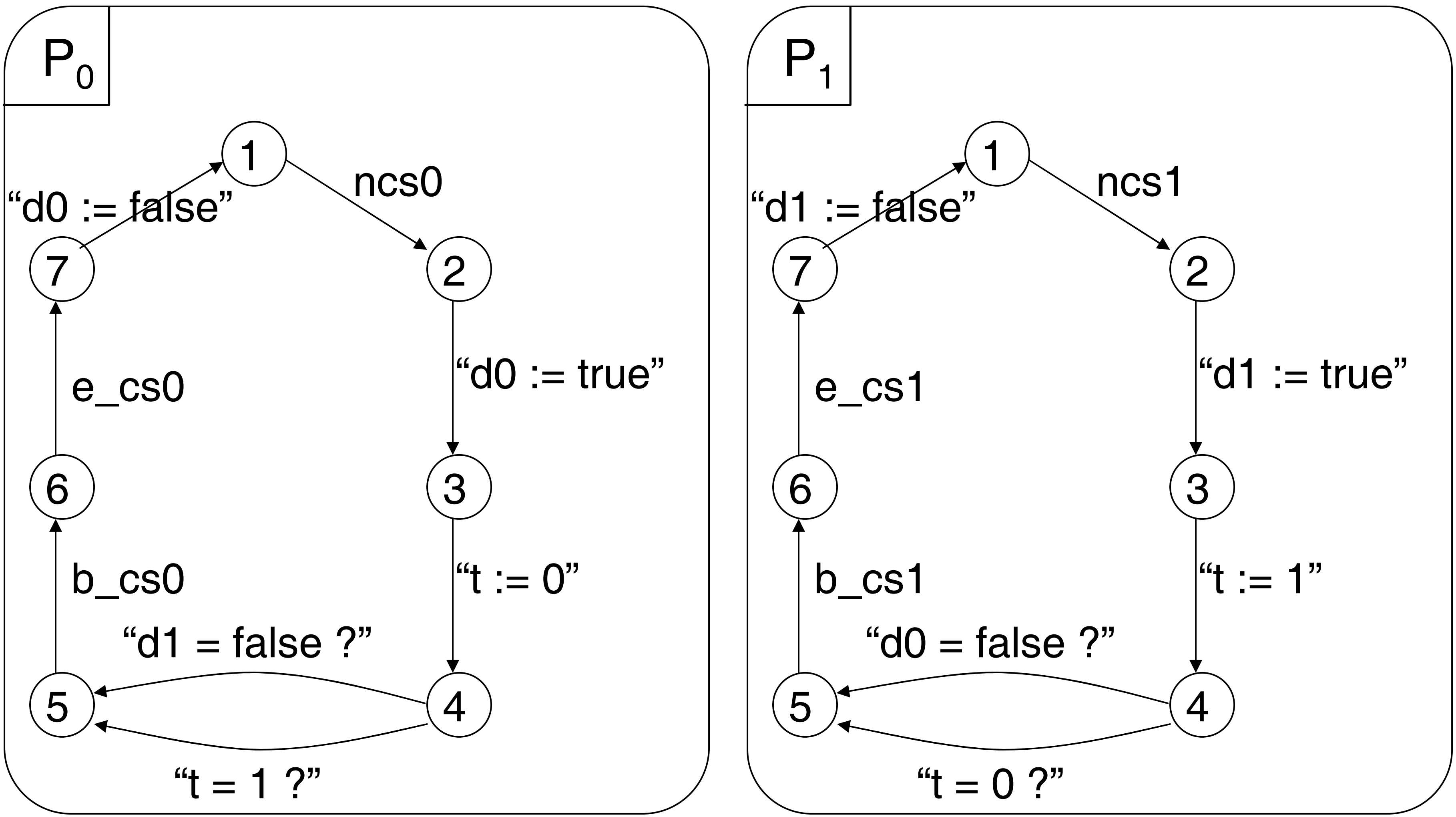
Peterson's algorithm [1968]

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 }

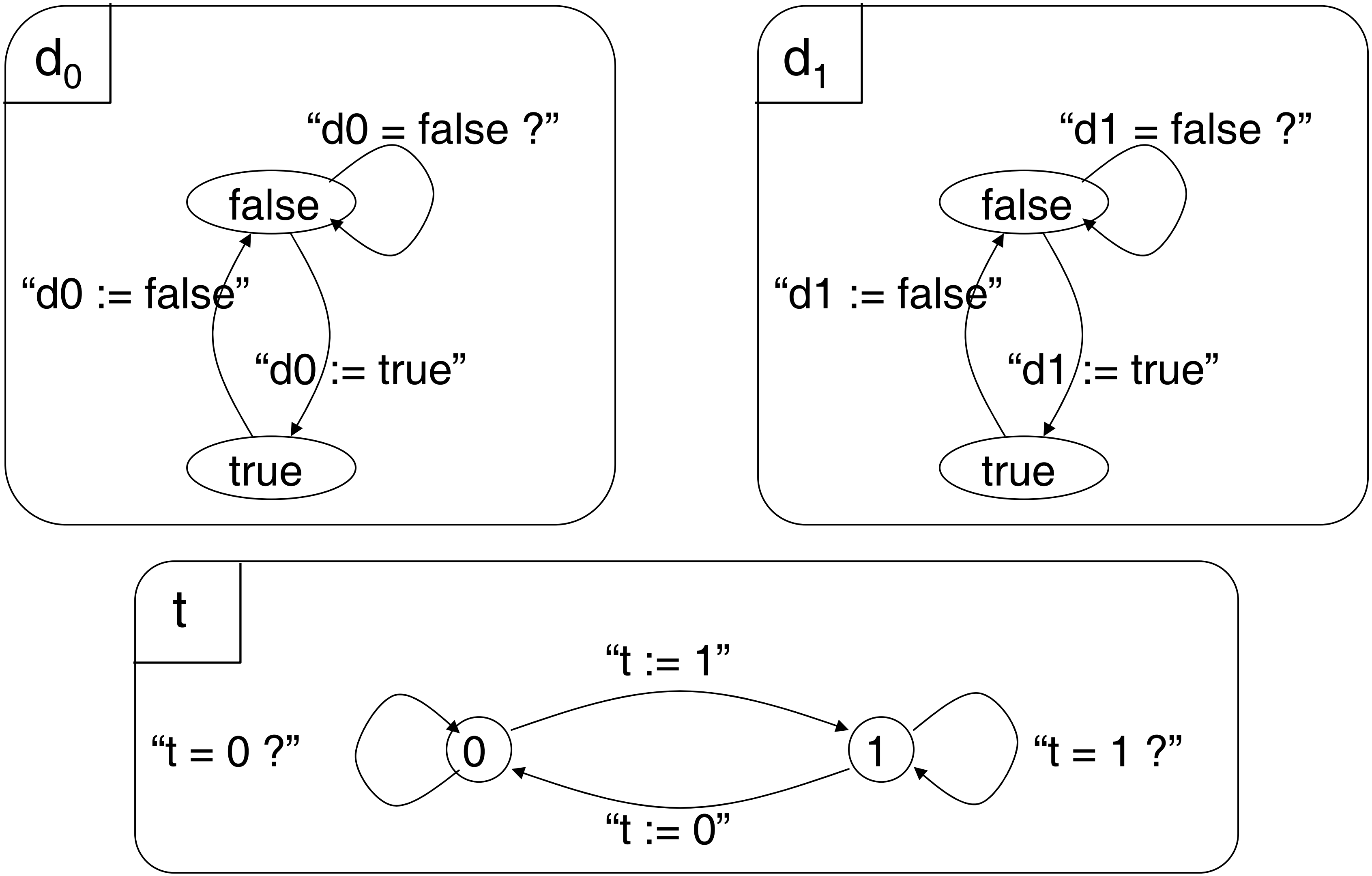
loop forever { P0 }
1 : { ncs0 }
2 : d0 := true
3 : t := 0
4 : **wait** (d1 = false **or** t = 1)
5 : { b_cs0 }
6 : { e_cs0 }
7 : d0 := false
endloop

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

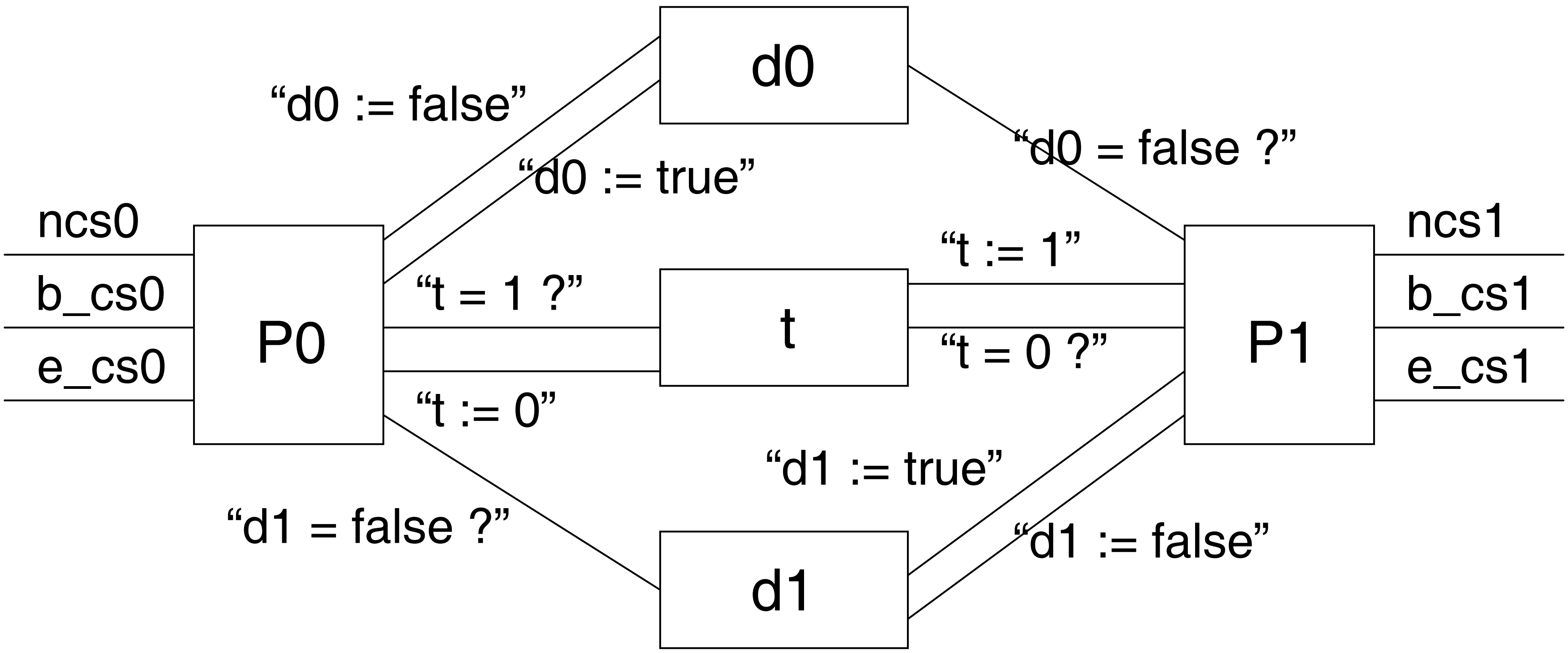
Automata of P_0 and P_1



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, ...

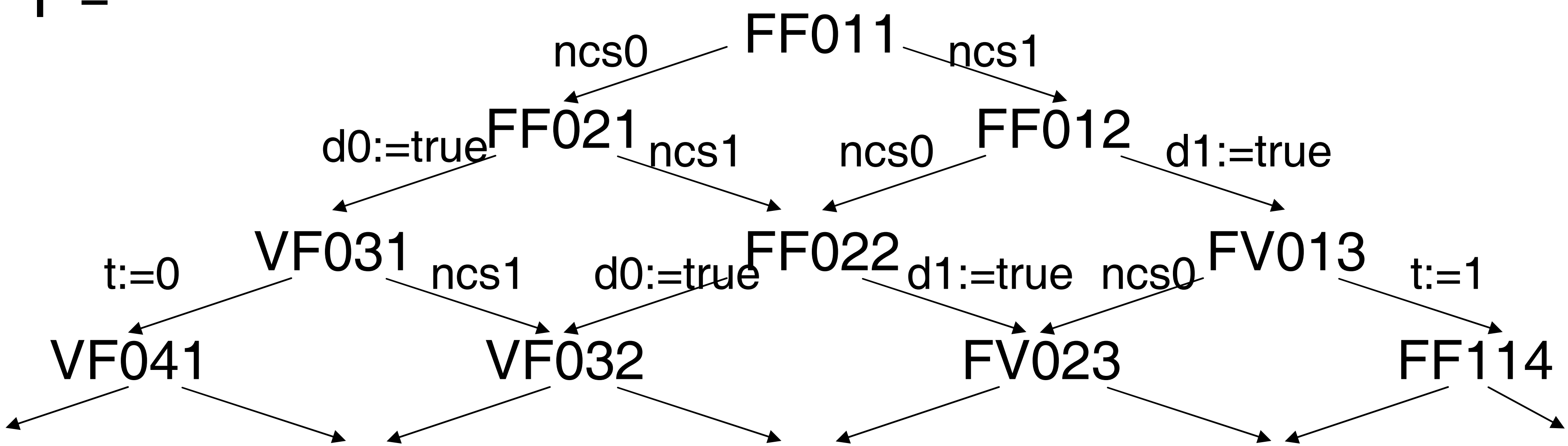
Construction of the LTS

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

$A = \{$ 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

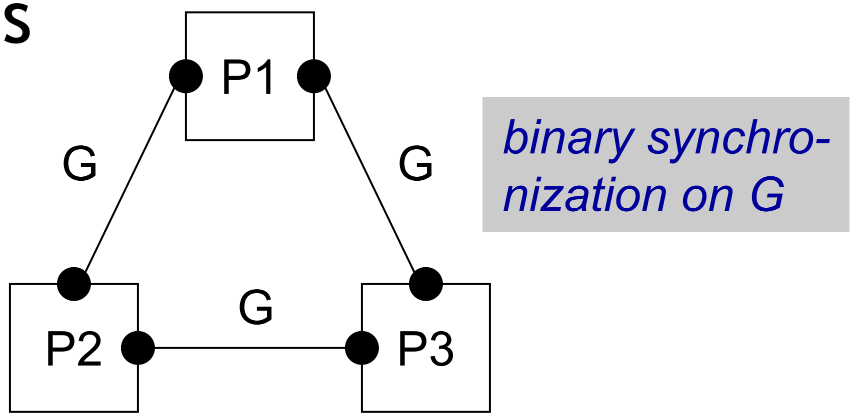
Verification

- 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

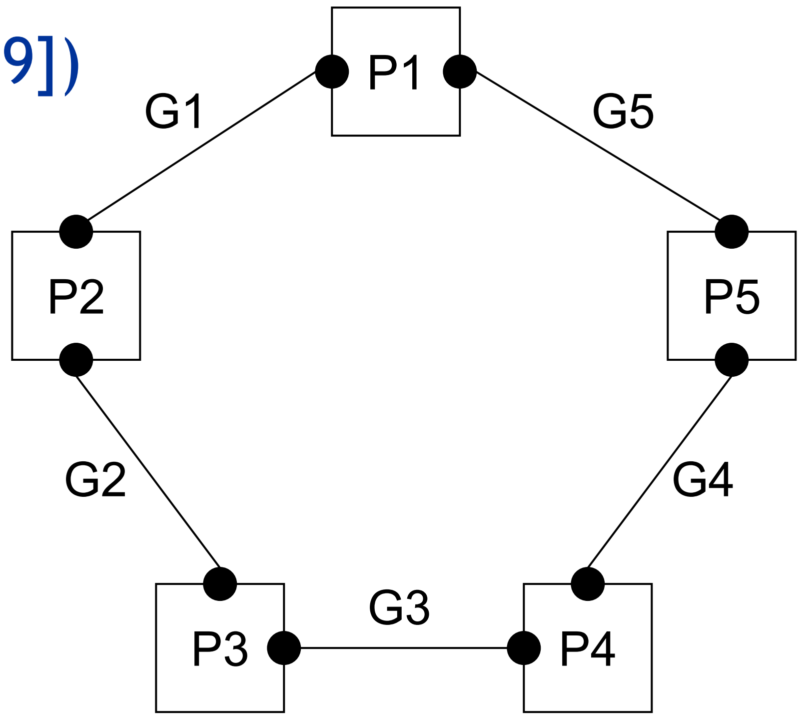
[see the chapter on temporal logics]

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 "|[]|":



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:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textbf{par } [\ g_1\#m_1, \ldots, g_p\#m_p \textbf{ in }]} \\
 & \underline{G}_1 \rightarrow B_1 \\
|| & \underline{G}_2 \rightarrow B_2 \\
\ldots & \\
|| & \underline{G}_n \rightarrow B_n \\
\multicolumn{2}{l}{\textbf{end par}}
\end{array}
$$

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:

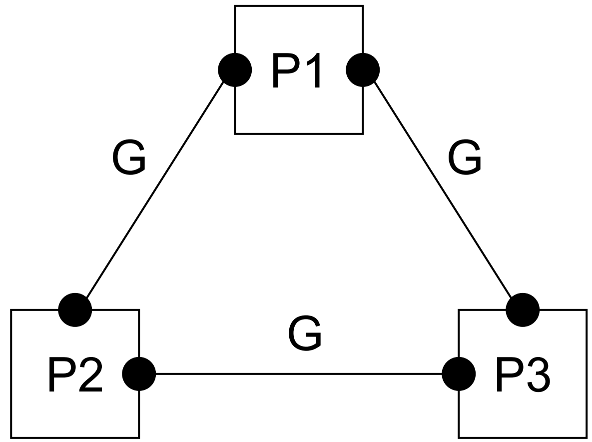
par G#2 **in**

$\quad G \rightarrow P_1$

$|| \quad G \rightarrow P_2$

$|| \quad G \rightarrow P_3$

end par



maximal cooperation avoided by means of synchronization degrees

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

- Description using general parallel composition:

par

$G_1, G_5 \rightarrow P_1$

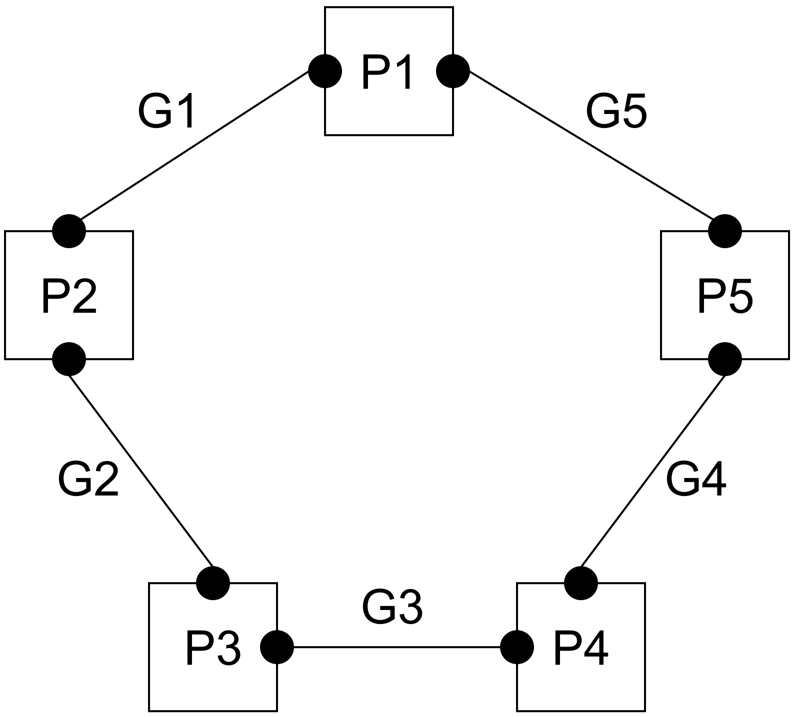
$||\quad G_2, G_1 \rightarrow P_2$

$||\quad G_3, G_2 \rightarrow P_3$

$||\quad G_4, G_3 \rightarrow P_4$

$||\quad G_5, G_4 \rightarrow P_5$

end par



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

Example (3/3)

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

$$B_1 \ |[G_1, \ldots, G_n]| \ B_2 \quad = \quad \textbf{par} \quad G_1, \ldots, G_n \rightarrow B_1$$
$$|| \quad G_1, \ldots, G_n \rightarrow B_2$$
$$\textbf{end par}$$

- CREW (Concurrent Read / Exclusive Write):

par W#2 **in**

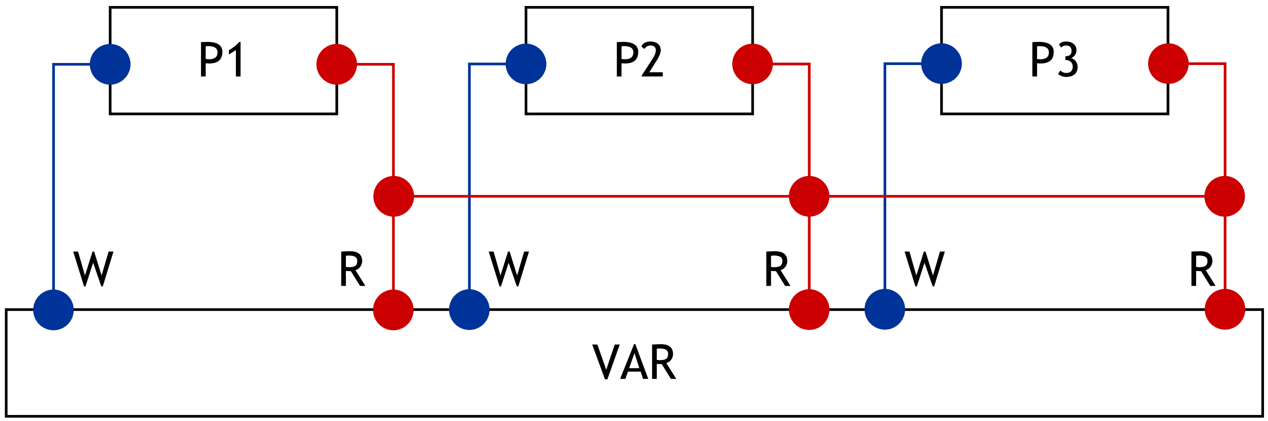
R, W → P_1

|| 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

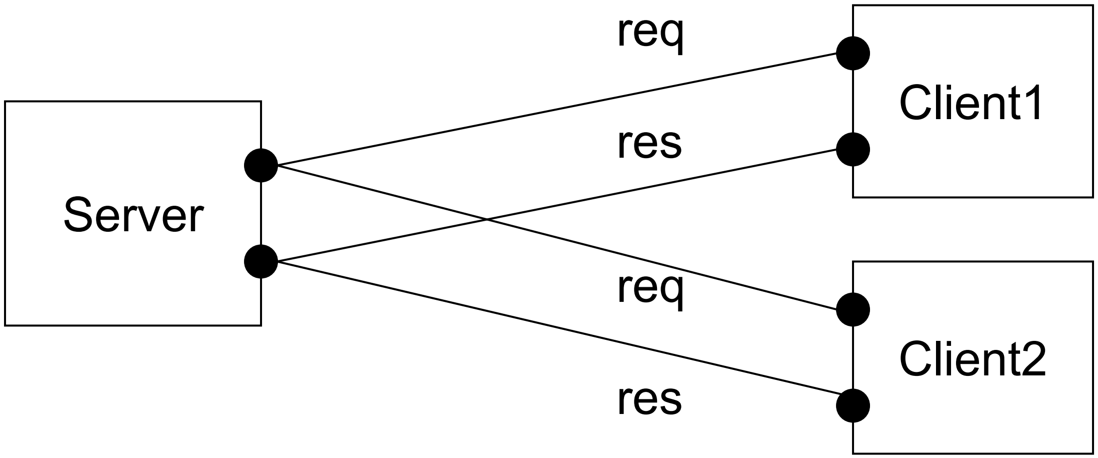
synchronization vectors

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

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]
  - ...

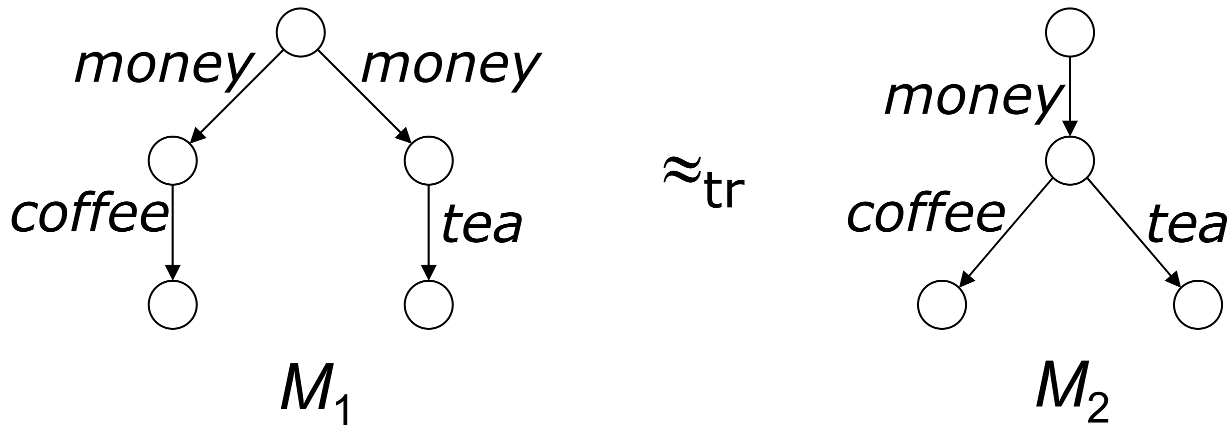
Trace equivalence

- 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 $s \text{ -i}\rightarrow s_1 \text{ -i}\rightarrow s_2 \ldots \text{ -}a\rightarrow s_k$
  - s =σ=>: there exists a transition sequence $s \text{ =}a_1\text{=> } s_1 \ldots \text{ =}a_n\text{=> } s_n$ such that $\sigma = a_1 \ldots a_n$
- Two state are trace equivalents iff they are the source of the same traces:

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

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*

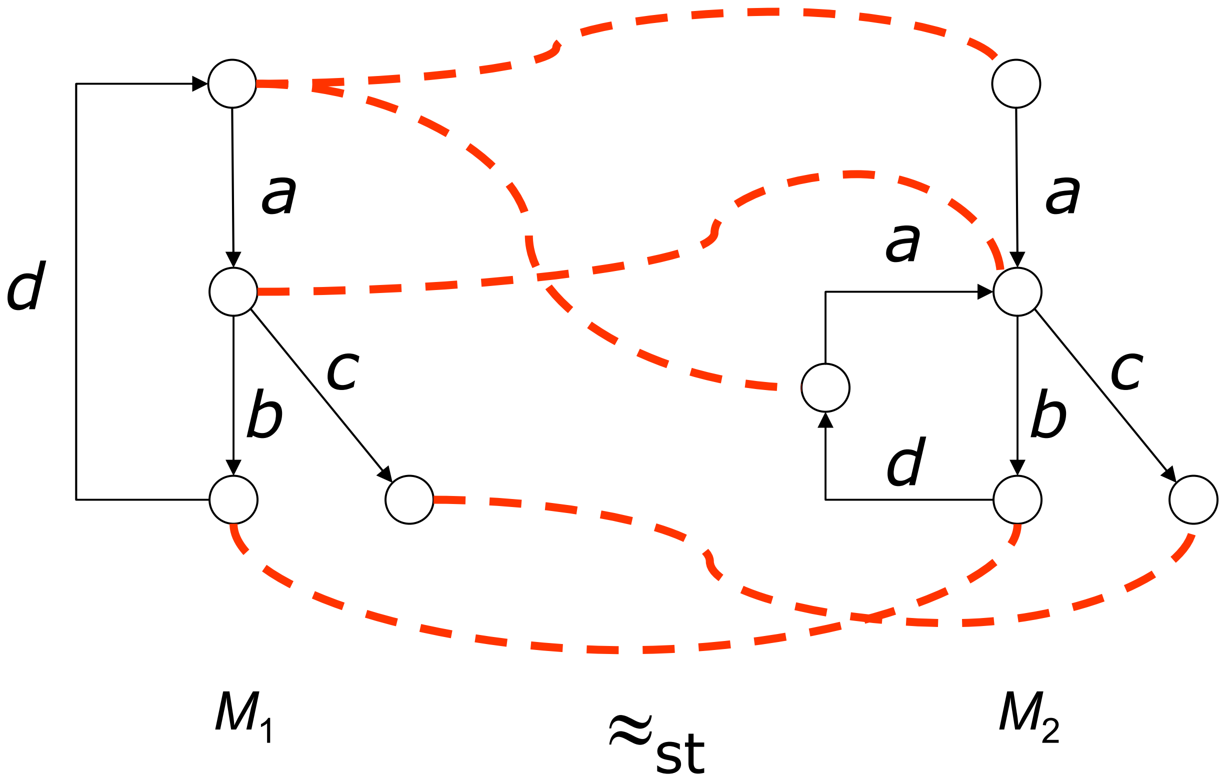
- Two states s_1 et s_2 are *bisimilar* iff they are the origin of the same behaviour (execution tree):

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

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

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

Strong bisimulation

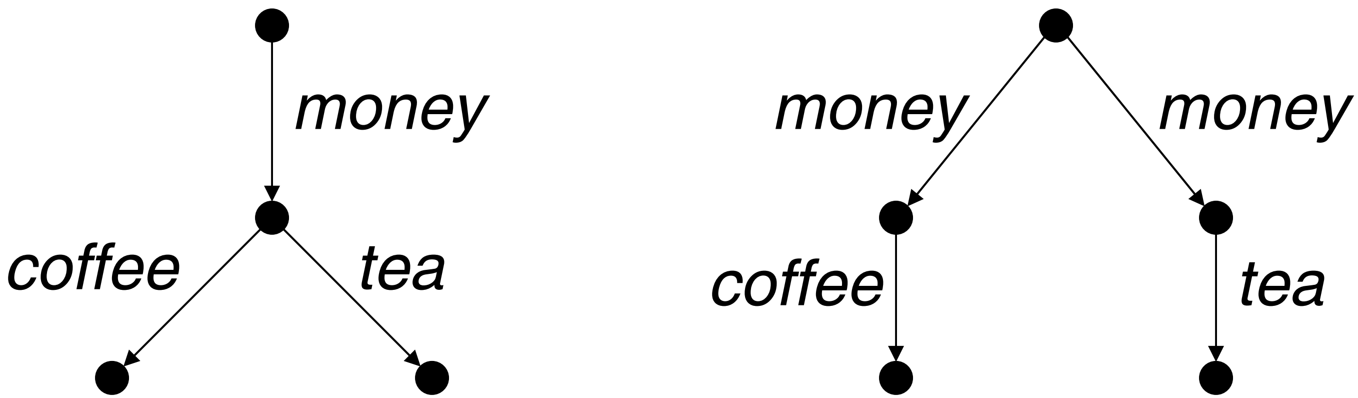


- Strong bisimulation: the largest bisimulation
  - ➔ *to show that two LTSs are strongly bisimilar, it is sufficient to find a bisimulation between them*

Is strong bisimulation sufficient?

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

  ➔ *does not distinguish the LTSs below*



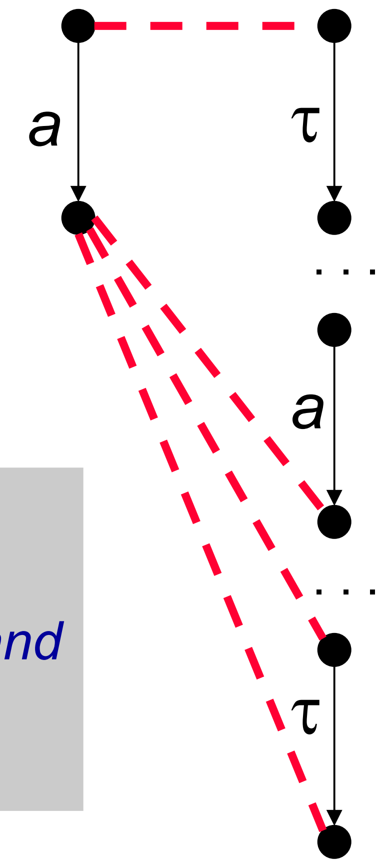
- *Strong bisimulation* captures the branching, but handles internal and visible actions in the same way

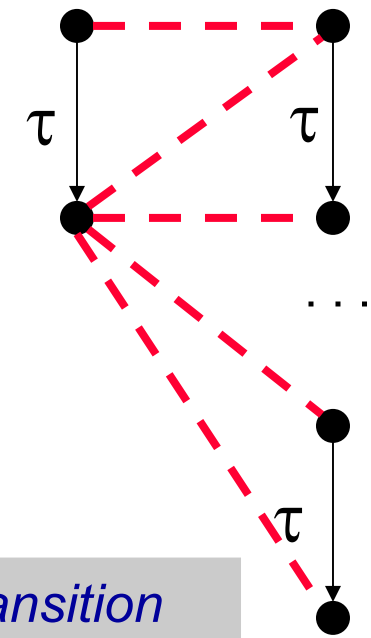
  ➔ *does not abstract away the internal behaviour*

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:

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

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

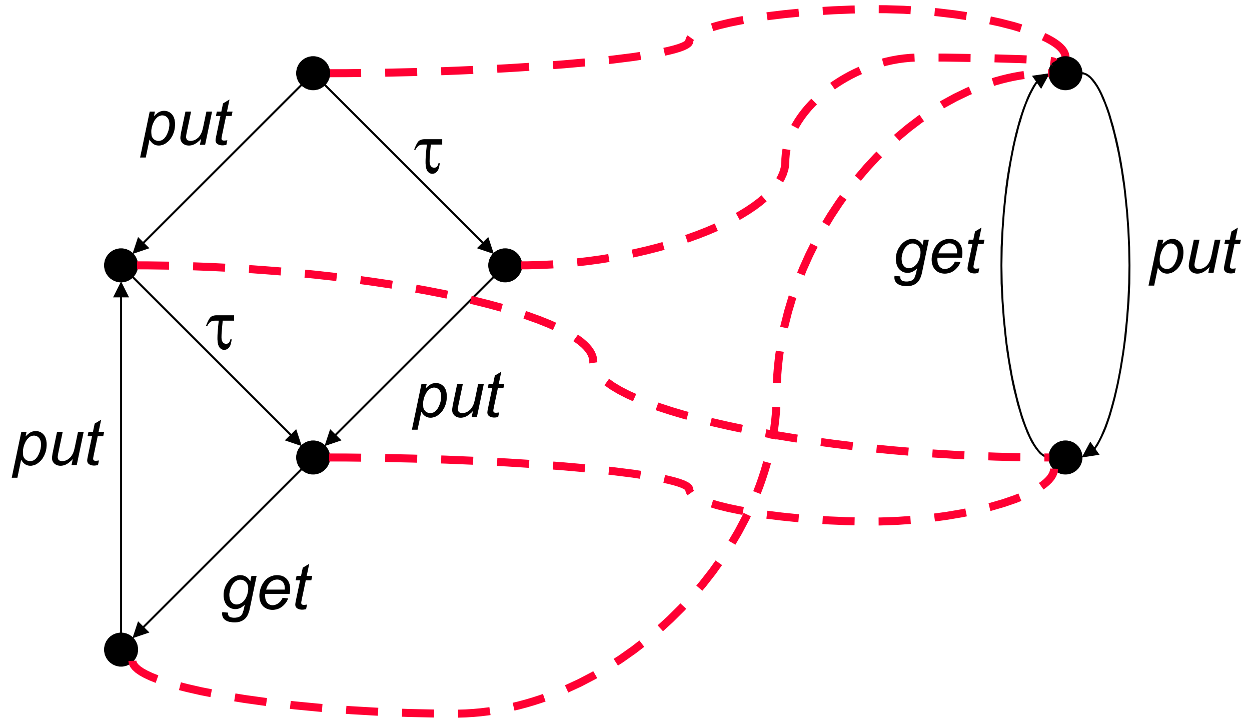
  and

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

  $\forall\ s_2 \text{-}\tau\rightarrow s_2'\ .\ \exists\ s_1 \text{-}\tau^*\rightarrow s_1'\ .\ s_1' \text{ eq } s_2'$
- $\approx_{obs}$ 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, ..., X_n
  - Gates: G_1, ..., G_n
  - Processes: P_1, ..., P_n
  - Sorts (≈ types): S_1, ..., S_n
  - Functions: F_1, ..., F_n
  - Comments: (* ... *)
- Caesar compiler of CADP [Garavel-Sifakis-90]

Value expressions and offers

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

$$
\begin{array}{lll}
V ::= X & & \\
 & | & F\,(V_1, \ldots, V_n) \\
 & | & V_1\;F\;V_2
\end{array}
$$

- Offers: $O_1, \ldots, O_n$

O ::= $!\,V$	emission of a value V
$\mid\ \ ?\,X : S$	reception of a value to be stored in a variable X of sort S

Behaviour expressions
(Lots Of Terribly Obscure Symbols :-)

- Behaviours: $B_1, \ldots, B_n$

B ::=	**stop**	inaction
|	$G_0\ O_1 \ldots O_n$ [V] ; B_0	action prefix
|	B_1 [] B_2	choice
|	B_1 |[$G_1, \ldots, G_n$]| B_2	parallel with synchronization on $G_1, \ldots, G_n$
|	B_1 ||| B_2	interleaving
|	**hide** $G_1, \ldots, G_n$ **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 [$G_1, \ldots, G_n$] ($V_1, \ldots, V_n$)	process call

Process definitions

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

B

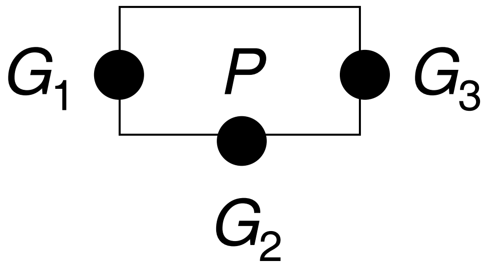
endproc

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 $[G_1, G_2, G_3]$ (...) :=

...

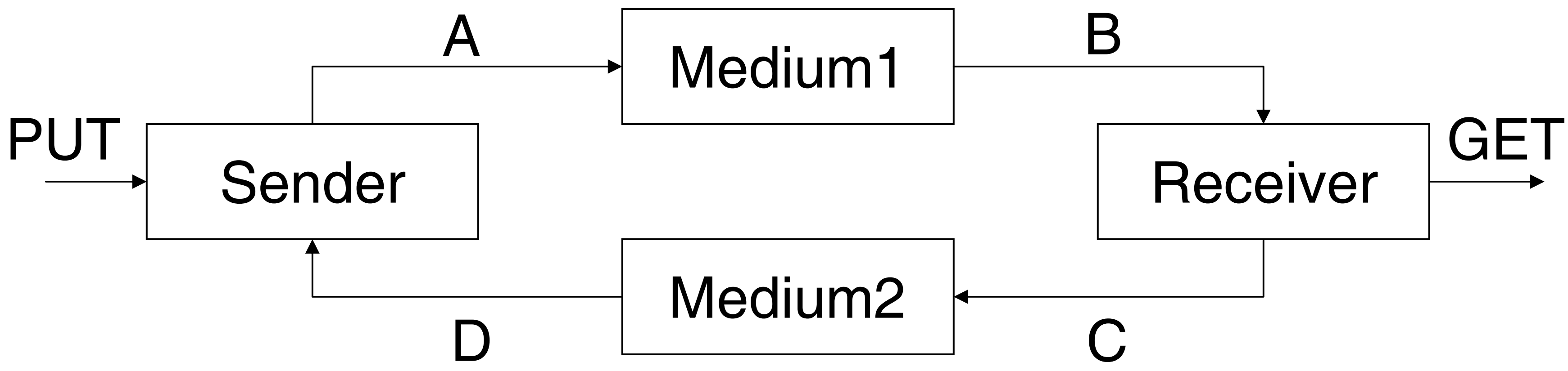
endproc

- 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



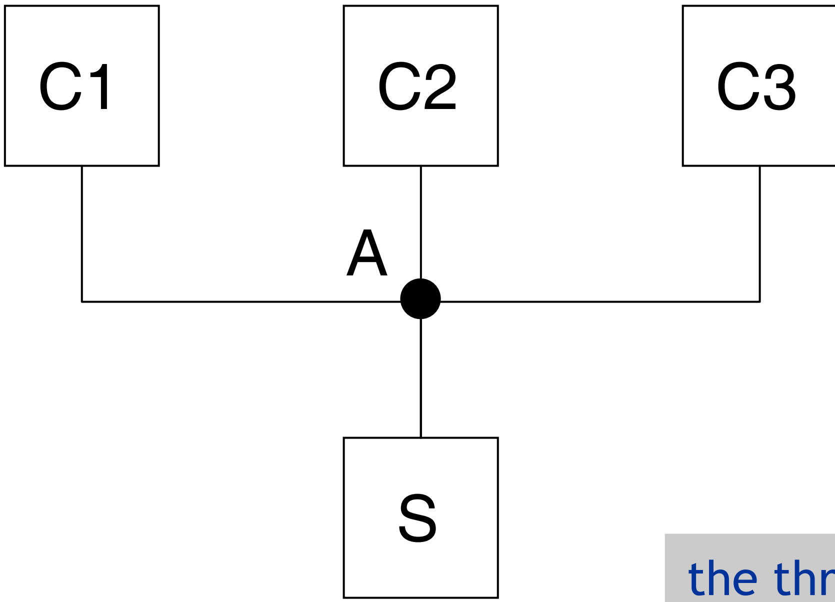
(Sender [PUT, A, D] ||| Receiver [GET, B, C])
|[A, B, C, D]|
(Medium1 [A, B] ||| Medium2 [C, D])

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



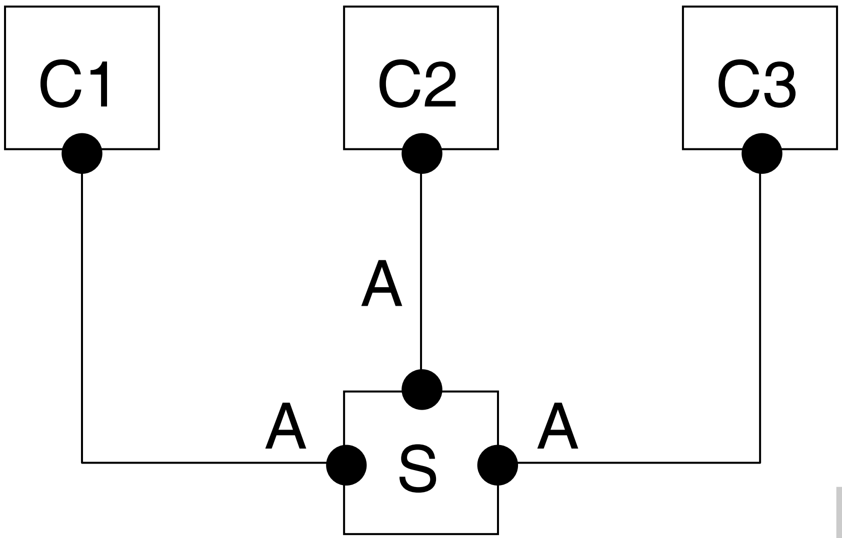
Example (client-server):

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

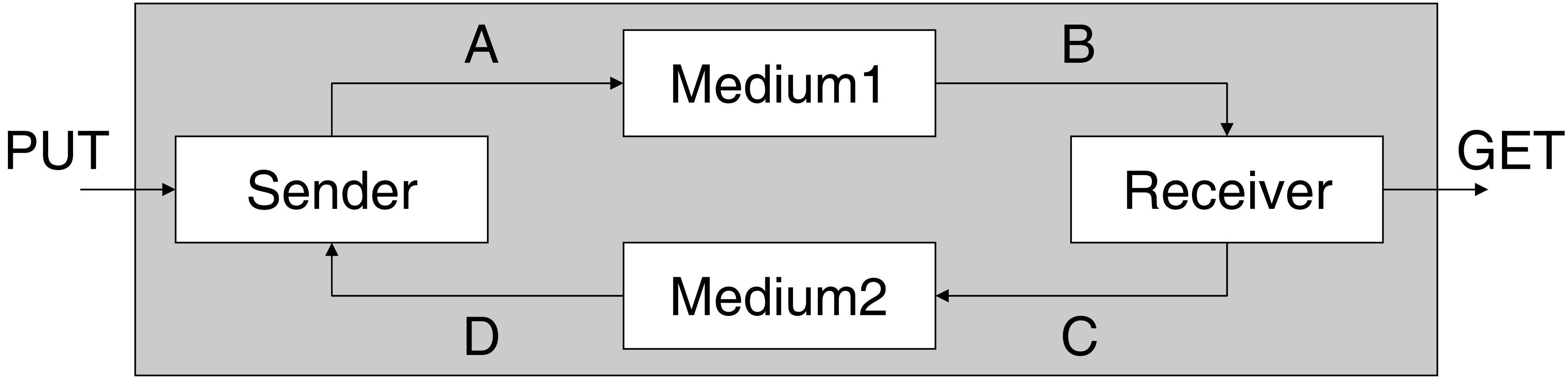
Abstraction
(hiding)

- 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:
  - <u>*G*</u>: 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"

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

normal gate

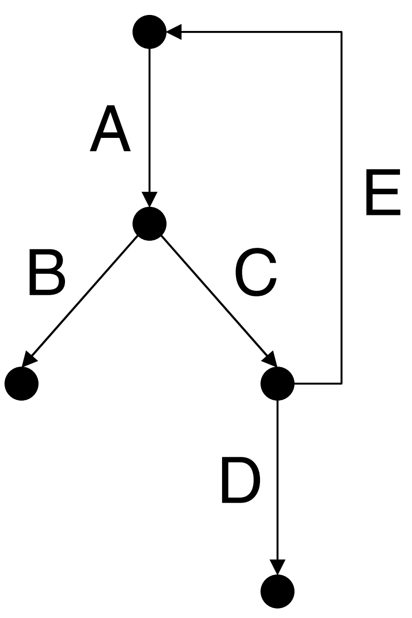
$$\frac{B \rightarrow_L B' \wedge \mathit{gate}\ (L) \in \underline{G}}{\textbf{hide}\ \underline{G}\ \textbf{in}\ B \rightarrow_{\mathbf{i}} \textbf{hide}\ \underline{G}\ \textbf{in}\ B'}$$

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* ($\neq$ 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 ($\approx$ 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

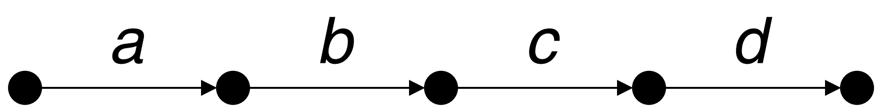
  has a behaviour equivalent to **stop** (unguarded recursion)

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)

$a \;;\; b \;;\; c \;;\; d \;;\; \textbf{stop}$



Semantics of “;”

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

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

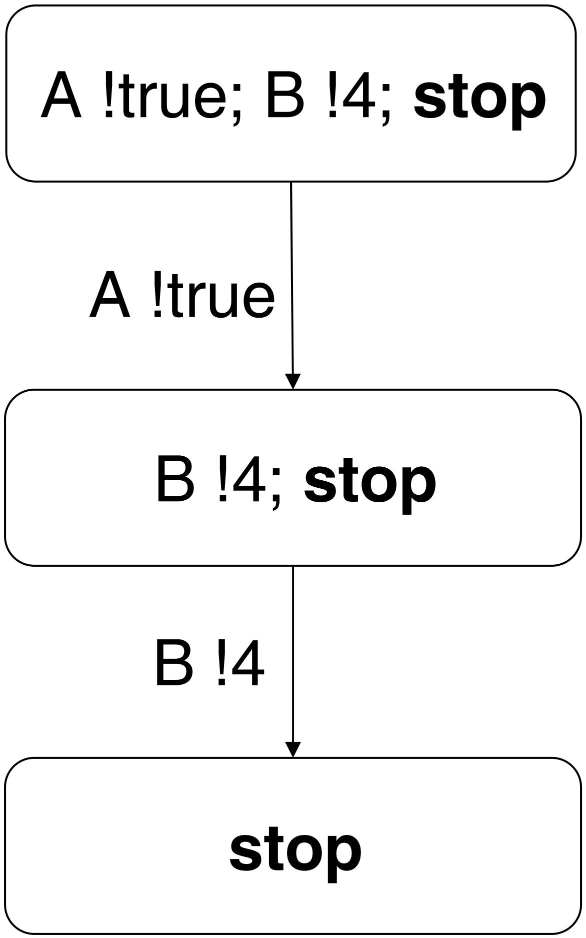
- The boolean guard and the offers are optional
- If the guard V is false, the rendezvous does not happen (deadlock):

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

Example (1/2)

Sequential composition:

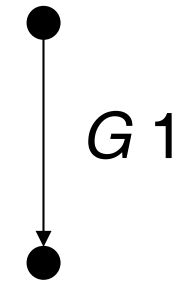
A !true; B !4; **stop**



Example (2/2)

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

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



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

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

Semantics of ";"

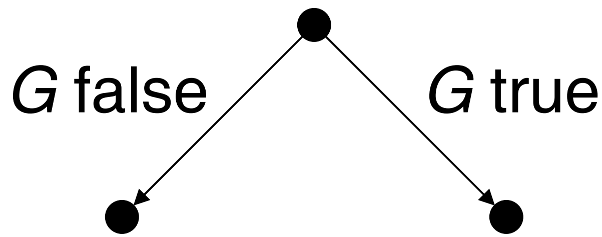
<u>Case 2</u>: action containing reception offer(s) (?X:S)

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

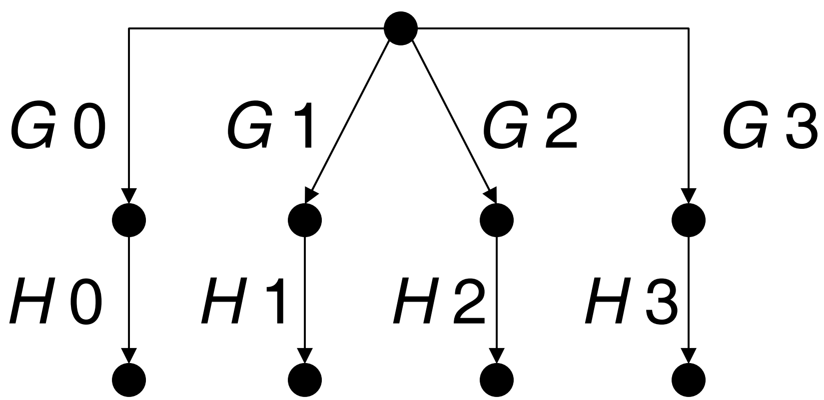
- The variables defined in the offers ?X:S are visible in the boolean guard *V* and inside *B*
- An action can freely mix emission and reception offers

Example (1/3)

G ?*X*:Bool;
  stop



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



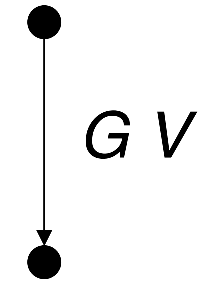
- The semantics handles the reception by branching on all possible values that can be received

Example (2/3)

- Emission of a value = guarded reception:

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

  where $S = type\ (V\ )$

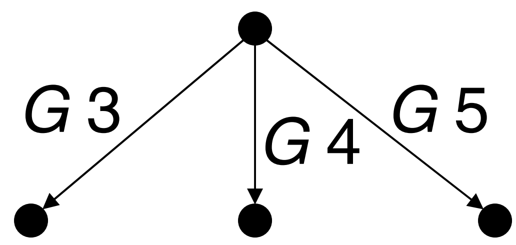


- Synchronization by *value generation*: two processes receive values of the same type on a gate

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

  $$|[\ G\ ]|$$

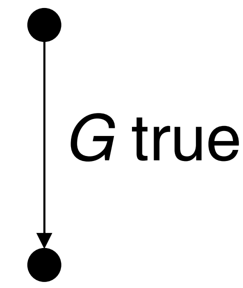
  $$G\ ?n_2{:}Nat\ [\ n_2 > 2];\ B_2$$



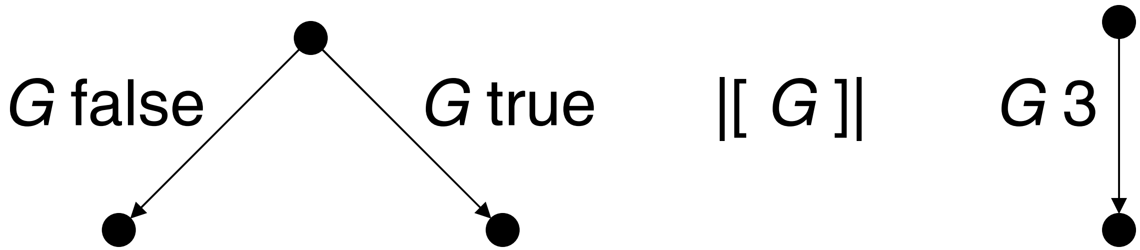
Example (3/3)

- 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

  - $prop\ (!V\ ) = \{\ V\ \}$
  - $prop\ (?X{:}S) = S$

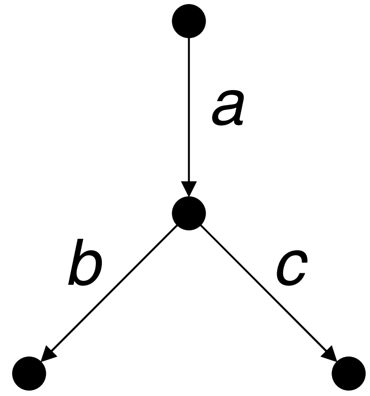
Choice operator ("[]")

- "[]": 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

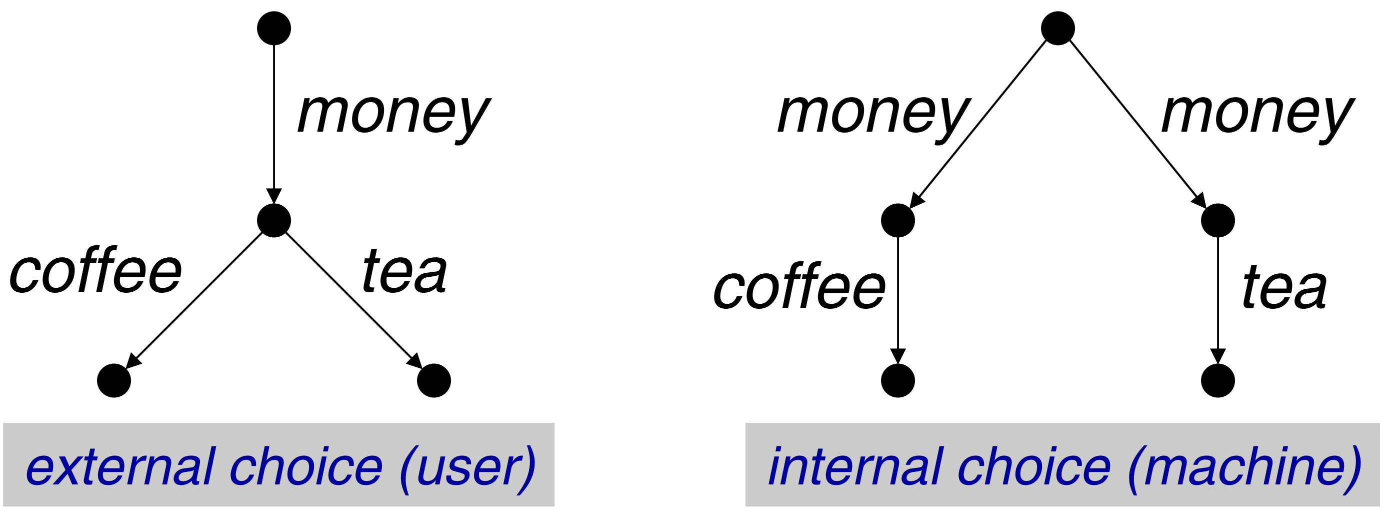
- After the choice, one of the two behaviours disappears (the execution was engaged on a branch of the choice and the other one is abandoned)

Internal / external choice

$(G_1 \;;\; B_1 \quad [] \quad G_2 \;;\; B_2\,)$

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

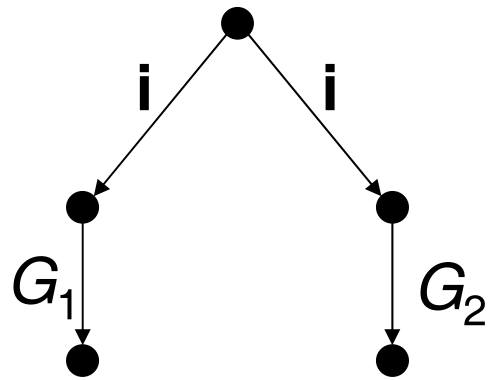
- Example (coffee machine):



Internal action ("i")

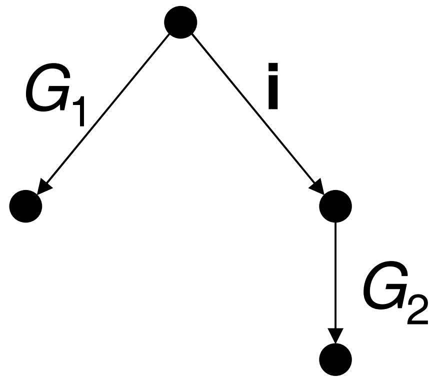
- In LOTOS, the special gate **i** denotes an internal event on which the environment cannot act:

(**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":

  $[\ V\ ]$ -> B_1
  []
  $[\ \mathrm{not}\ (V\ )\ ]$ -> B_2

  "case":

  $[\ X < 0\ ]$ -> B_1
  []
  $[\ X = 0\ ]$ -> B_2
  []
  $[\ X > 0\ ]$ -> B_3

- Beware of overlapping guards:

  $[\ \mathrm{X} \leq 0\ ]$ -> B_1
  []
  $[\ \mathrm{X} \geq 0\ ]$ -> 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**

  $\quad$ *G* !*X*;

  $\qquad$ *H* !cons (1, *X*);

  $\qquad\quad$ **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"

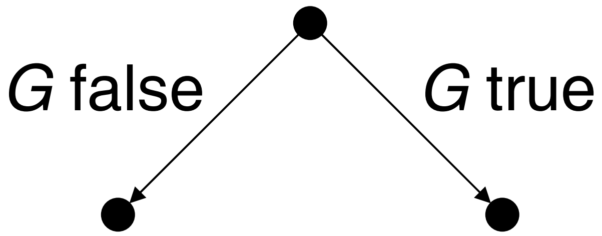
- Operator "**choice**": similar to "**let**", except that variable X takes a nondeterministic value in the domain of its sort S
- Semantics:

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

- 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"

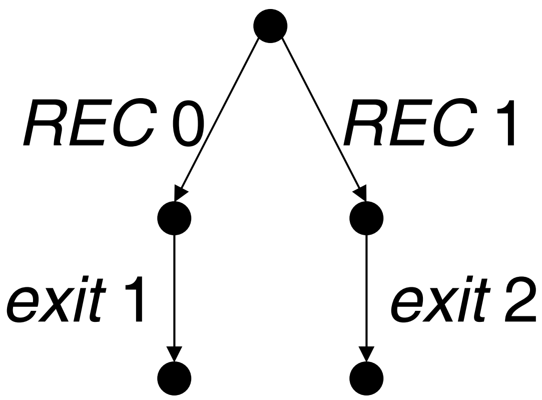
- LOTOS allows to express *normal termination* of a behaviour, possibly with the return of one or several values:

  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, ..., V_n are optional (“**exit**” means normal termination without producing any value)

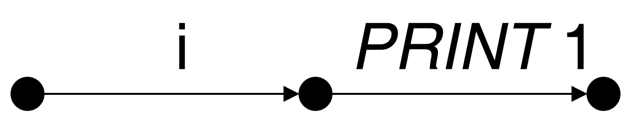
Operator ">>"

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

  B_1 >> **accept** $X_1{:}S_1,\ldots,\ X_n{:}S_n$ **in** B_2

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

```
exit (1) >> accept n:Nat in
    PRINT !n ; stop
```



Semantics of “>>”

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

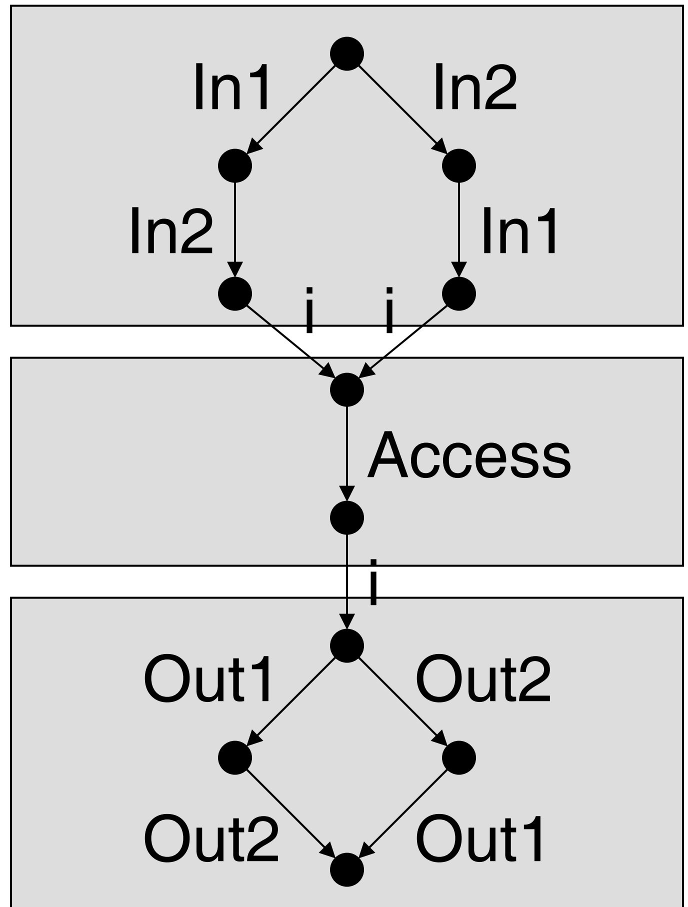
$$\frac{B_1 \rightarrow_{\mathit{exit}\ \underline{V}} B_1'}{(B_1 >> \textbf{accept}\ \underline{X}{:}\underline{S}\ \textbf{in}\ B_2) \rightarrow_{\mathbf{i}} B_2\ [\ \underline{V}\ /\ \underline{X}\ ]}$$

- 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:

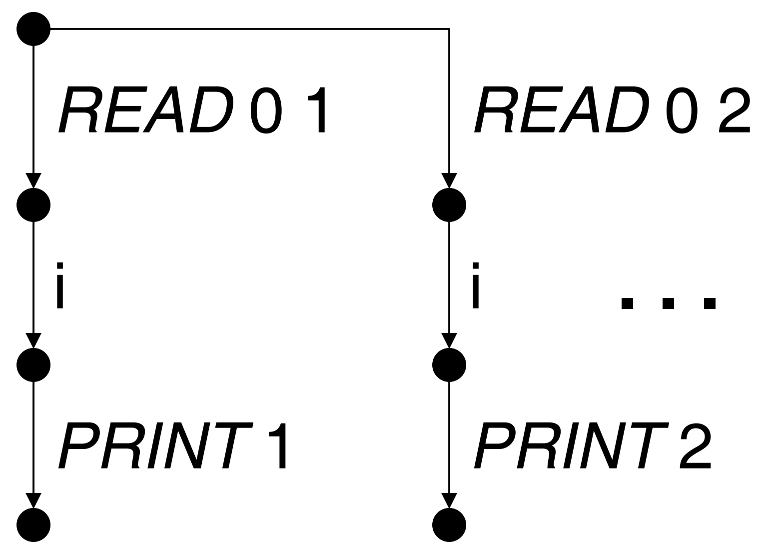
```
(In1; In2; exit
 []
 In2; In1; exit)
>>
(Access; exit)
>>
(Out1; Out2; stop
 []
 Out2; Out1; stop)
```



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

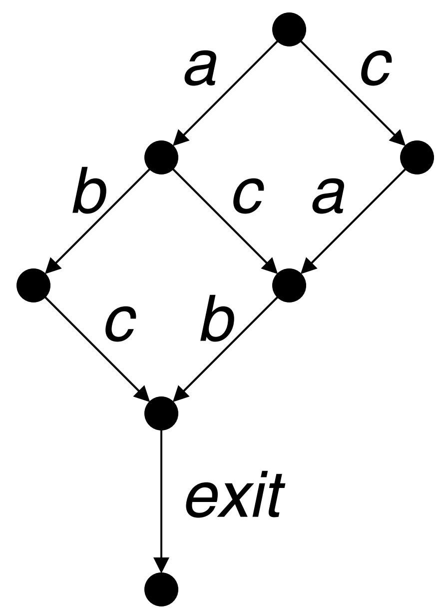
- Example of call:

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

Example (4/4)

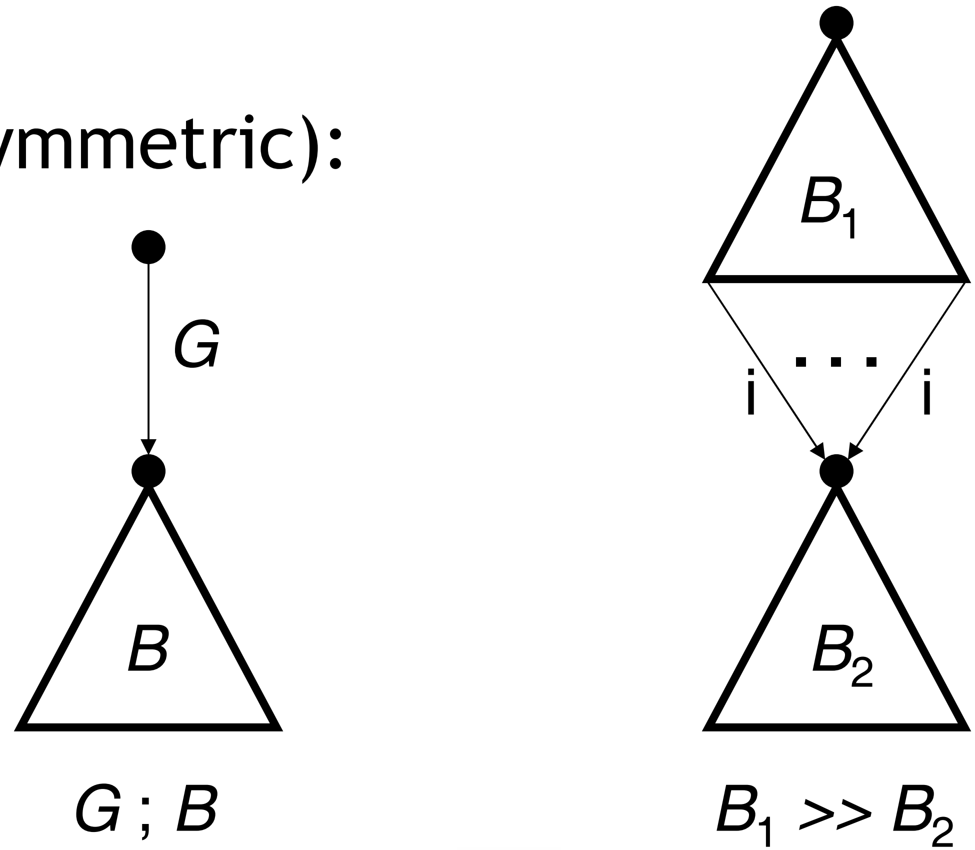
- Combination of "**exit**" and parallel composition: the two behaviours are synchronized on the *exit* gate (they terminate simultaneously)

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



Sequential composition
(summary)

- In LOTOS, difference between
  "$;$" (asymmetric)
  and
  "$>>$" (symmetric):



Process call

- Let a process *P* defined by:

  process $P\ [G_1, \ldots, G_n]\ (X_1{:}S_1, \ldots, X_n{:}S_n)$:=

  B

  endproc

- Semantics of a call to *P*:

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

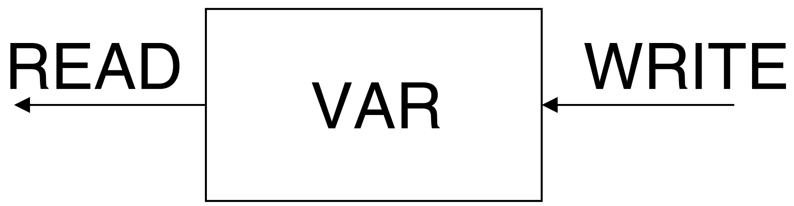
- This semantics explains why a call to

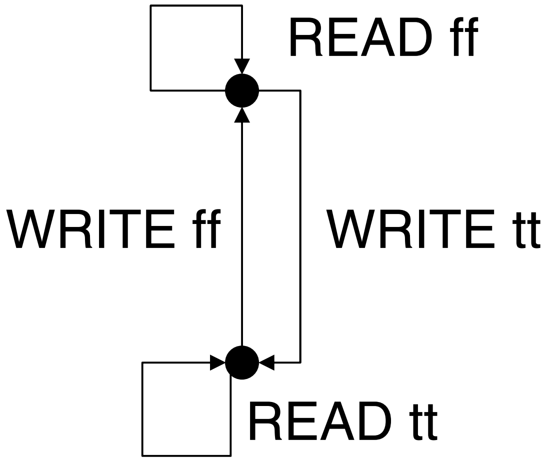
  process P[G] : **noexit** := P[G] **endproc**

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

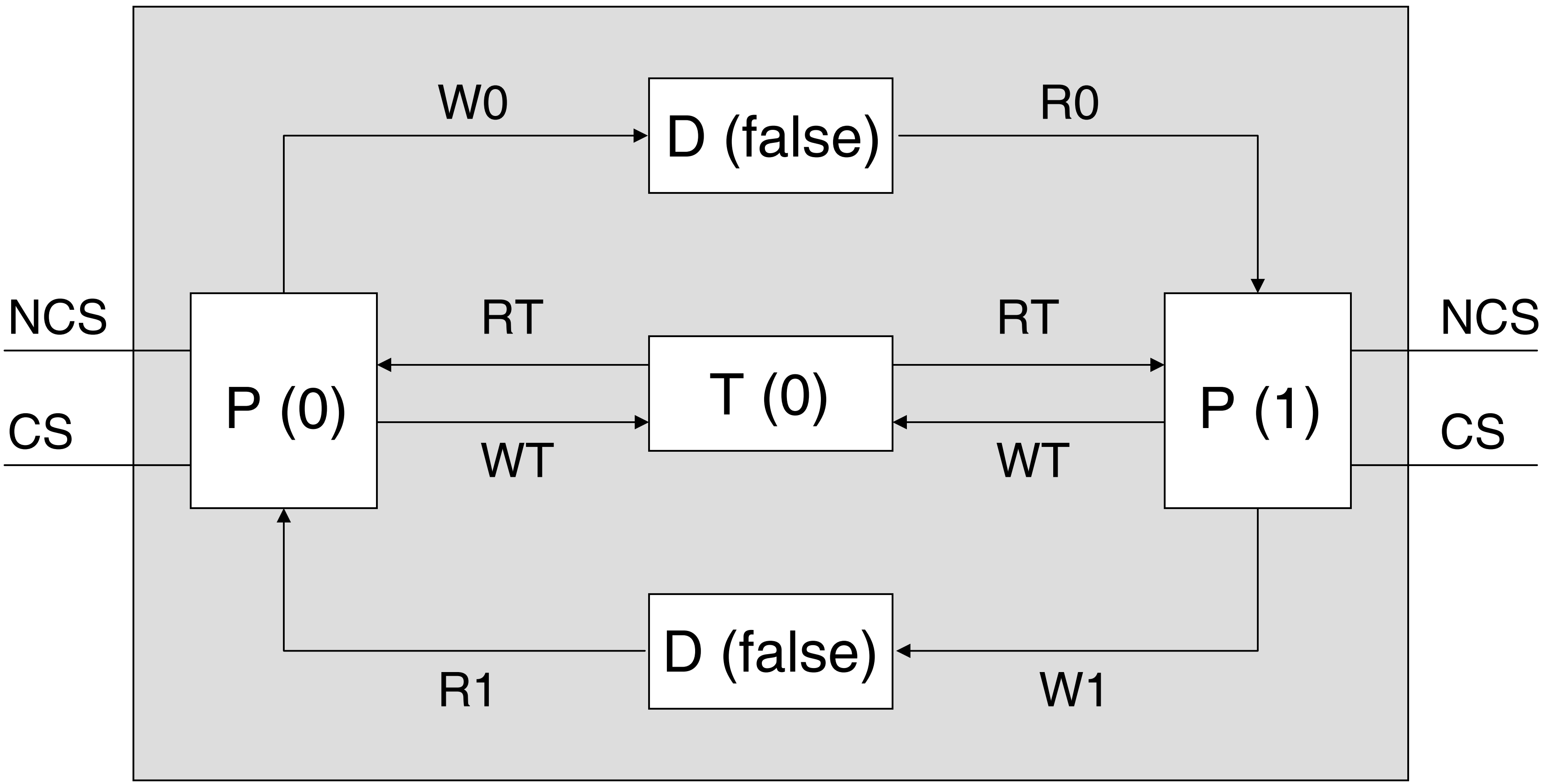
Processes P0 et P1
(continued)

- 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)



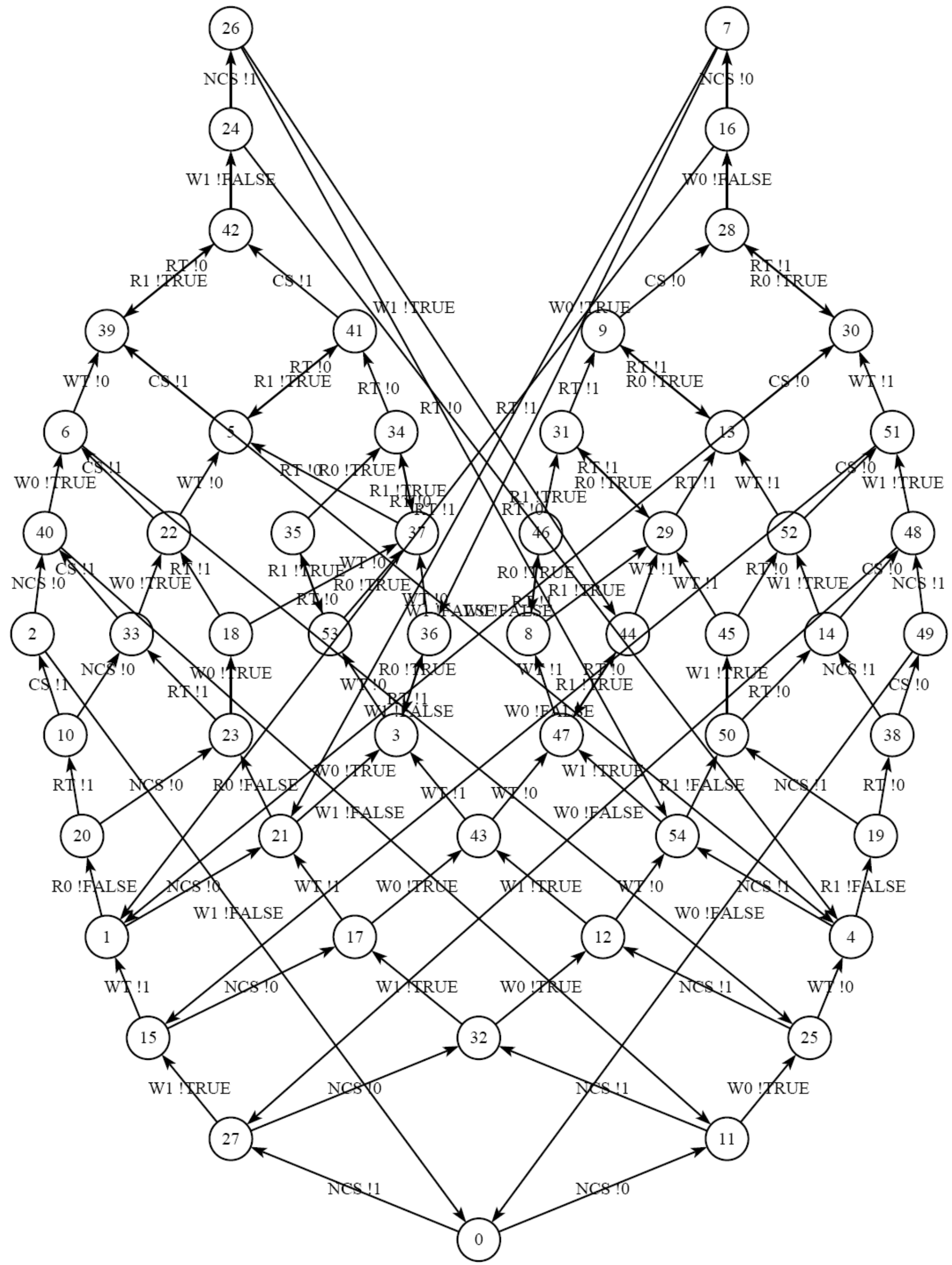
Architecture of the system

(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