Communicating automata

Communicating automata (CA)

- Simple formalism to describe asynchronous concurrent systems
- Allow the basic concepts studied in this course to be introduced
- Important notions:
  - System description using an automaton
  - Decomposition into communicating automata
  - automata product (parallelisation)
  - automata synchronisation
  - construction of the state graph

Example: car lights

dashboard

V, V' → swith on/ off sidelight (*veilleuses*)
C, C' → switch on / off dimmers (*codes*)
P, P' → switch on / off high beam (*phares*)

G, G' → switch on / off left (gauche) turn signal
D, D' → switch on / off right (droit) turn signal

lights (bulbs)

$v \in \{0, 1\}$
$c \in \{0, 1\}$
$p \in \{0, 1\}$
$k \in \{0, g, d\}$

State variables:

$v = 1$	iff	sidelight on
$c = 1$	iff	dimmers on
$p = 1$	iff	high beam on
$k \in \{g / d\}$	iff	left/ right turn signal on

Modeling with a single automaton (1/2)

- states: $\langle v, c, p, k \rangle$
- initial state: $\underline{\langle 0, 0, 0, 0 \rangle}$ (all lights off)
- transitions labeled by commands (V, V',…)

$$\begin{tikzcd}[column sep=huge, row sep=large]
\langle 0, 0, 0, g\rangle \arrow[r, shift right, "G'"'] \arrow[d, shift right, "V"'] & \underline{\langle 0, 0, 0, 0\rangle} \arrow[l, shift right, "G"'] \arrow[r, shift left, "D"] \arrow[d, shift right, "V"'] & \langle 0, 0, 0, d\rangle \arrow[l, shift left, "D'"] \arrow[d, shift right, "V"'] \\
\langle 1, 0, 0, g\rangle \arrow[u, shift right, "V'"'] \arrow[r, shift right, "G'"'] \arrow[d, shift right, "C"'] & \langle 1, 0, 0, 0\rangle \arrow[u, shift right, "V'"'] \arrow[l, shift right, "G"'] \arrow[r, shift left, "D"] \arrow[d, shift right, "C"'] & \langle 1, 0, 0, d\rangle \arrow[u, shift right, "V'"'] \arrow[l, shift left, "D'"] \arrow[d, shift right, "C"'] \\
\langle 0, 1, 0, g\rangle \arrow[u, shift right, "C'"'] \arrow[r, shift right, "G'"'] \arrow[d, shift right, "P"'] & \langle 0, 1, 0, 0\rangle \arrow[u, shift right, "C'"'] \arrow[l, shift right, "G"'] \arrow[r, shift left, "D"] \arrow[d, shift right, "P"'] & \langle 0, 1, 0, d\rangle \arrow[u, shift right, "C'"'] \arrow[l, shift left, "D'"] \arrow[d, shift right, "P"'] \\
\langle 0, 0, 1, g\rangle \arrow[u, shift right, "P'"'] \arrow[r, shift right, "G'"'] & \langle 0, 0, 1, 0\rangle \arrow[u, shift right, "P'"'] \arrow[l, shift right, "G"'] \arrow[r, shift left, "D"] & \langle 0, 0, 1, d\rangle \arrow[u, shift right, "P'"'] \arrow[l, shift left, "D'"]
\end{tikzcd}$$

12 states, 34 transitions

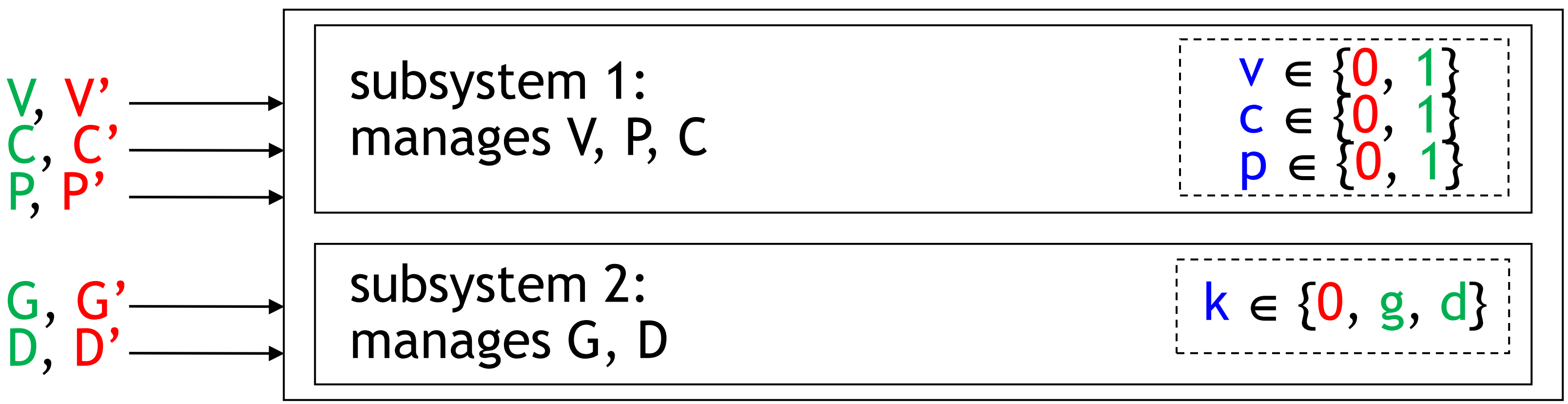
Modeling with a single automaton (2/2)

Remarks :

- Theoretical number of states $\langle v, c, p, k \rangle$:
  $2 \times 2 \times 2 \times 3 = 24$
- There are only 12 states $\Rightarrow$ some states are not reachable from the initial state due to the system constraints
- There is no *sink state*: all states have at least one successor
- From each state, the initial state is reachable (the system is *reinitialisable*)

Modeling with parallel automata (1/4)

Use the independance of subsystems VPC and GD



Automaton of subsystem 1: (state: ⟨v, c, p⟩)

$$\underline{\langle 0, 0, 0\rangle} \underset{V'}{\overset{V}{\rightleftarrows}} \langle 1, 0, 0\rangle \underset{C'}{\overset{C}{\rightleftarrows}} \langle 0, 1, 0\rangle \underset{P'}{\overset{P}{\rightleftarrows}} \langle 0, 0, 1\rangle$$

4 states, 6 transitions

Automaton of subsystem 2: (state: ⟨k⟩)

$$\langle g\rangle \underset{G'}{\overset{G}{\leftrightarrows}} \underline{\langle 0\rangle} \underset{D'}{\overset{D}{\rightleftarrows}} \langle d\rangle$$

3 states, 4 transitions

Modeling with parallel automata (2/4)

Remarks :

- Complexities are added instead of multiplied (« divide and conquer ») :

  4 + 3 << 12 states

  6 + 4 << 34 transitions

- Generally, a system can be decomposed into subsystems that are not completely independent by adding communications and synchronisations
- But then the decomposition can introduce consistency problems (e.g. deadlocks)

Modeling with parallel automata (3/4)

The modular decomposition…

- … can be imposed at the physical level
  - Modeling physically distributed activities
  - Example: multiprocessor or multitask execution
- … can be chosen at the logical level
  - Can simplify the system design
  - Better program structuration

Modeling with parallel automata (4/4)

Two notions to be distinguished:

- Parallelism of description (*logic* parallelism)

  = conceptual means to decompose a system into subsystems
- Parallelism of implementation (*physical* parallelism)

  = execution on several processors or a mutitask OS

« Orthogonal » notions:

- A program containing parallelism of description can be implemented sequentially (e.g., Lustre, Esterel) or in a distributed way (e.g., Ada)
- A program without parallelism of description can be implemented in parallel (e.g., parallelisation of Fortran code)

Automata: definition

An *automaton* or *Labeled Transition System* (LTS) is a 4-tuple $M = \langle S, A, T, s_0 \rangle$, where:

- S is a set of *states*
- A is a set of *labels* (*actions*)
- $T \subseteq S \times A \times S$ is the *transition relation*
- $s_0 \in S$ is the *initial state*

finite or not

Notation : $(\forall\ s_1, s_2 \in S, a \in A)$

$(s_1, a, s_2) \in T$ (or s_1 –a-> s_2) $\Leftrightarrow$ there exists a transition labeled by a that goes from s_1 to s_2

Automata

- LTS is the class of automata studied in this course: the information (labels) is attached to transitions
- There are other classes of automata:
  - with information attached to states: Kripke structures
  - with actions structured in the form of input/outputs: Mealy and Moore automata
- Advantages of the LTS model:
  - simplicity
  - adapted to the description of systems based on actions, e.g., systems communicating by messages exchanged on a network

Product of automata

Objective:

- Define an internal composition law

  $$\otimes : LTS \times LTS \rightarrow LTS$$

  which expresses the parallel composition of two automata LTS_1 and LTS_2
- Synchronise LTS_1 and LTS_2 on one or several actions

Goal: To be able to analyse the system behaviour

Product of asynchrounous automata

- Principle : independent actions cannot be observed simultaneously [Milner-89]

$$\downarrow a \;\otimes\; \downarrow b \;\rightarrow\; \begin{tikzcd} & \cdot \arrow[dl, "a"'] \arrow[dr, "b"] & \\ \cdot \arrow[dr, "b"'] & & \cdot \arrow[dl, "a"] \\ & \cdot & \end{tikzcd}$$

interleaving semantics

$\Rightarrow$ expansion of parallelism into choice and sequence (Milner's *expansion theorem*)

- But some actions can be synchronized

$$\downarrow a \;\otimes\; \downarrow a \;\rightarrow\; \downarrow a$$

if a is an action to be synchronized

Automata product: definition

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

$M_1 \otimes_L M_2 = \langle S, A, T, s_0 \rangle$
where:

- $S = S_1 \times S_2$
- $A = A_1 \cup A_2$
- $s_0 = \langle s_{01}, s_{02} \rangle$
- T is defined by rules R_1, R_2 and R_3

M_1 evolves alone: **R_1**

$$\frac{s_1 \xrightarrow{a} s_1' \wedge a \notin L}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2 \rangle}$$

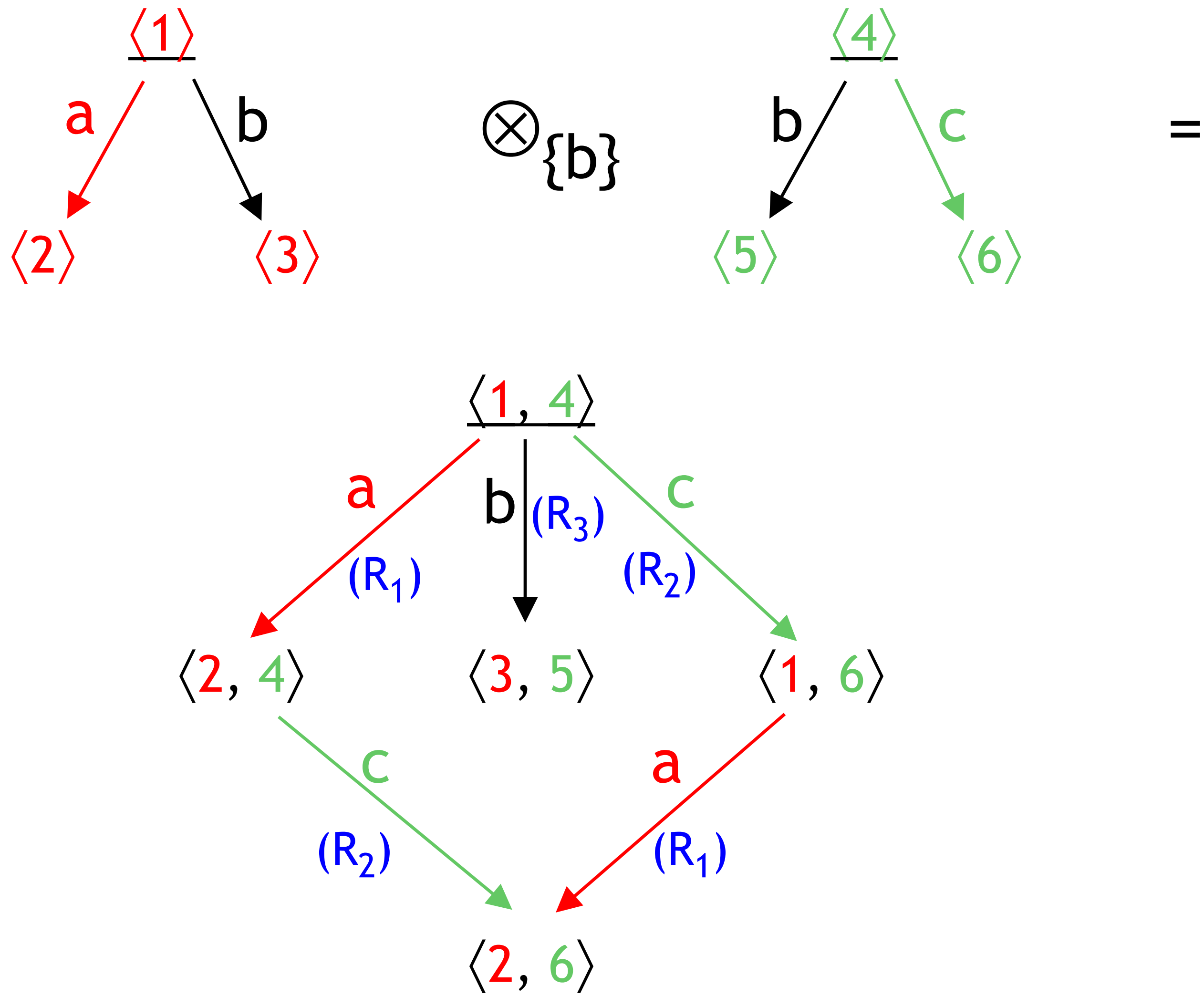
M_2 evolves alone: **R_2**

$$\frac{s_2 \xrightarrow{a} s_2' \wedge a \notin L}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1, s_2' \rangle}$$

M_1 and M_2 synchronize: **R_3**

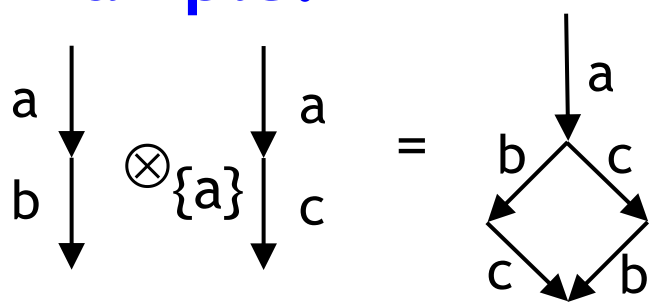
$$\frac{s_1 \xrightarrow{a} s_1' \wedge s_2 \xrightarrow{a} s_2' \wedge a \in L}{\langle s_1, s_2 \rangle \xrightarrow{a} \langle s_1', s_2' \rangle}$$

Example

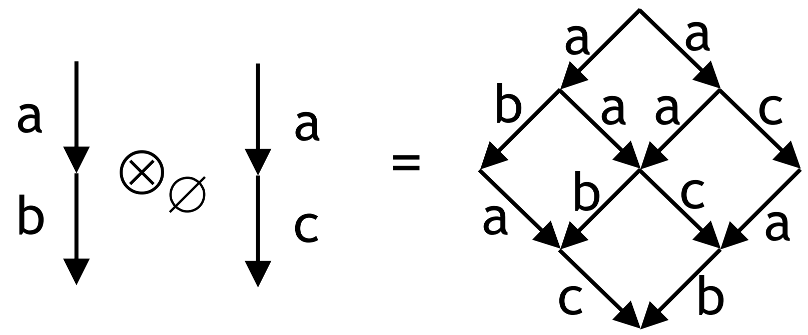


Remarks

- if $L = \emptyset$ (no action to be synchronized): M_1 and M_2 evolve fully asynchronously
- $\otimes_L$ may create **nondeterminism** (ND) if an action present in both LTS is not in L
  Example:



deterministic



nondeterministic for a

- Concurrent systems are generally ND

Example of modeling by CA

The mutual exclusion (ME) problem:

Given two processes P_0 and P_1 with a shared memory, can the mutual exclusion of accesses to this memory be guaranteed?

Several solutions « at the software level » were proposed to resolve the ME problem: algorithms of Peterson, Dekker, Knuth, ...

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

Example: The Peterson algorithm

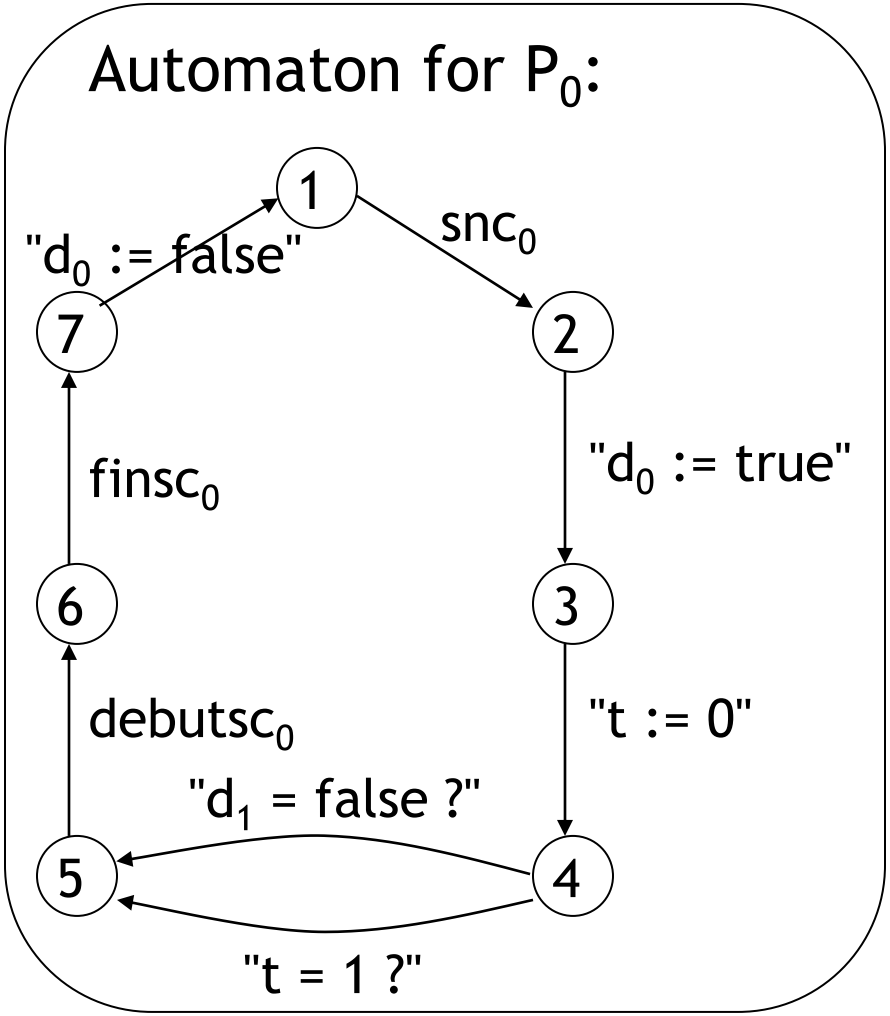
var d_0 : bool := false { read by P_1, written by P_0 }
var d_1 : bool := false { read by P_0, written by P_1 }
var t $\in$ {0, 1} := 0 { read/written by P_0 and P_1 }

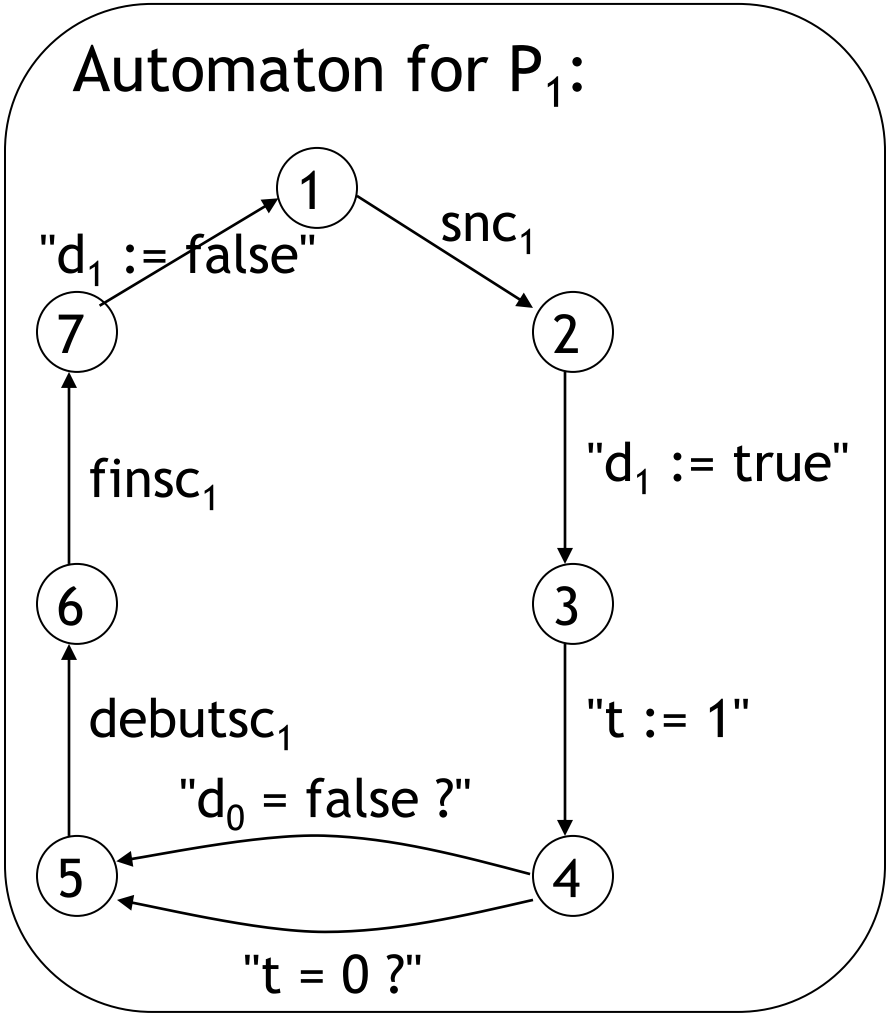
```
loop forever { P0 }
1 : { snc0 }
2 : d0 := true
3 : t := 0
4 : wait (d1 = false or t = 1)
5 : { debutsc0 }
6 : { finsc0 }
7 : d0 := false
endloop
```

```
loop forever { P1 }
1 : { snc1 }
2 : d1 := true
3 : t := 1
4 : wait (d0 = false or t = 0)
5 : { debutsc1 }
6 : { finsc1 }
7 : d1 := false
endloop
```

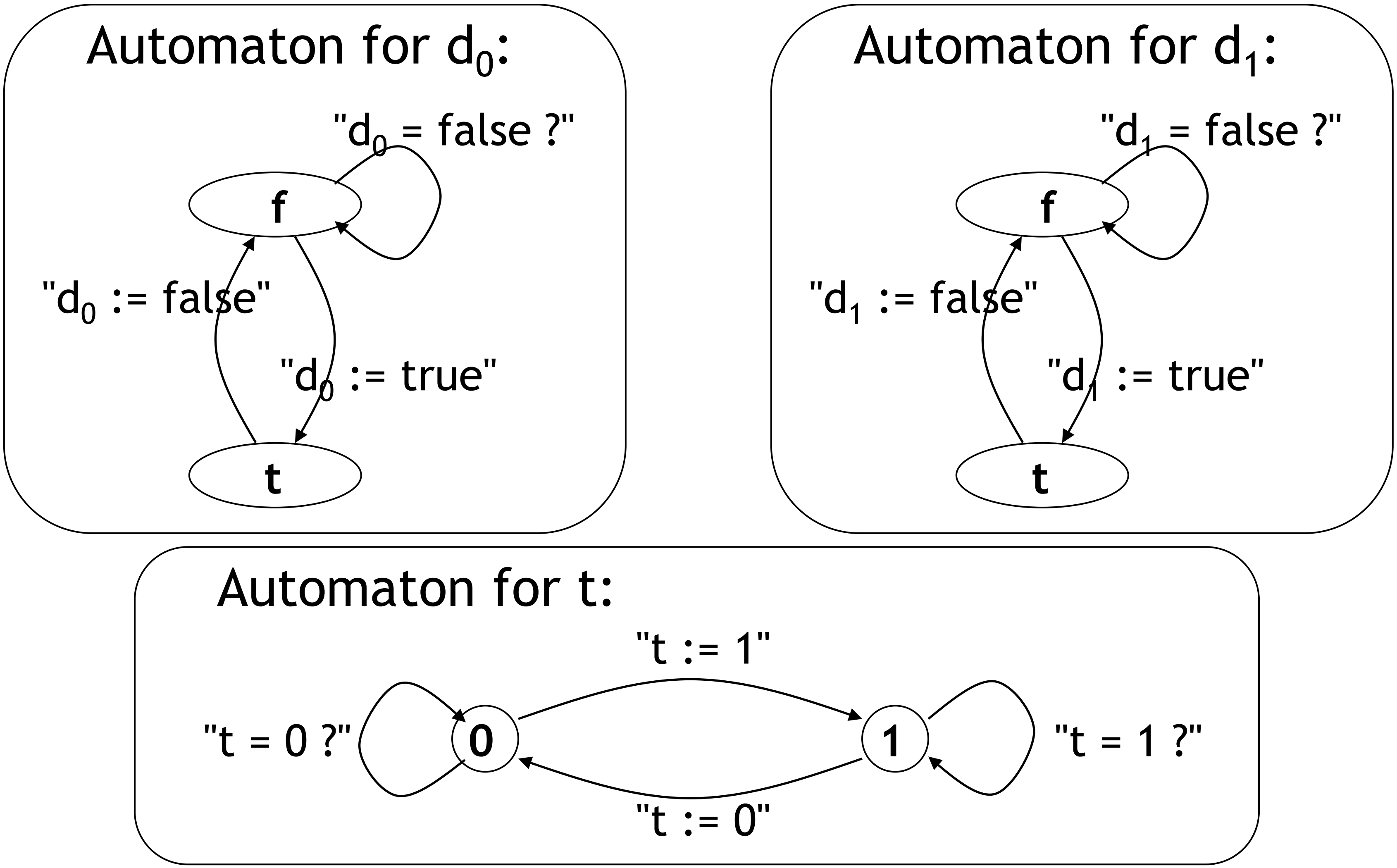
G. L. Peterson. *Myths about the mutual exclusion problem*. Information Processing Letters 12(3):115-116, June 13, 1981

Peterson: Automata for P_0 and P_1

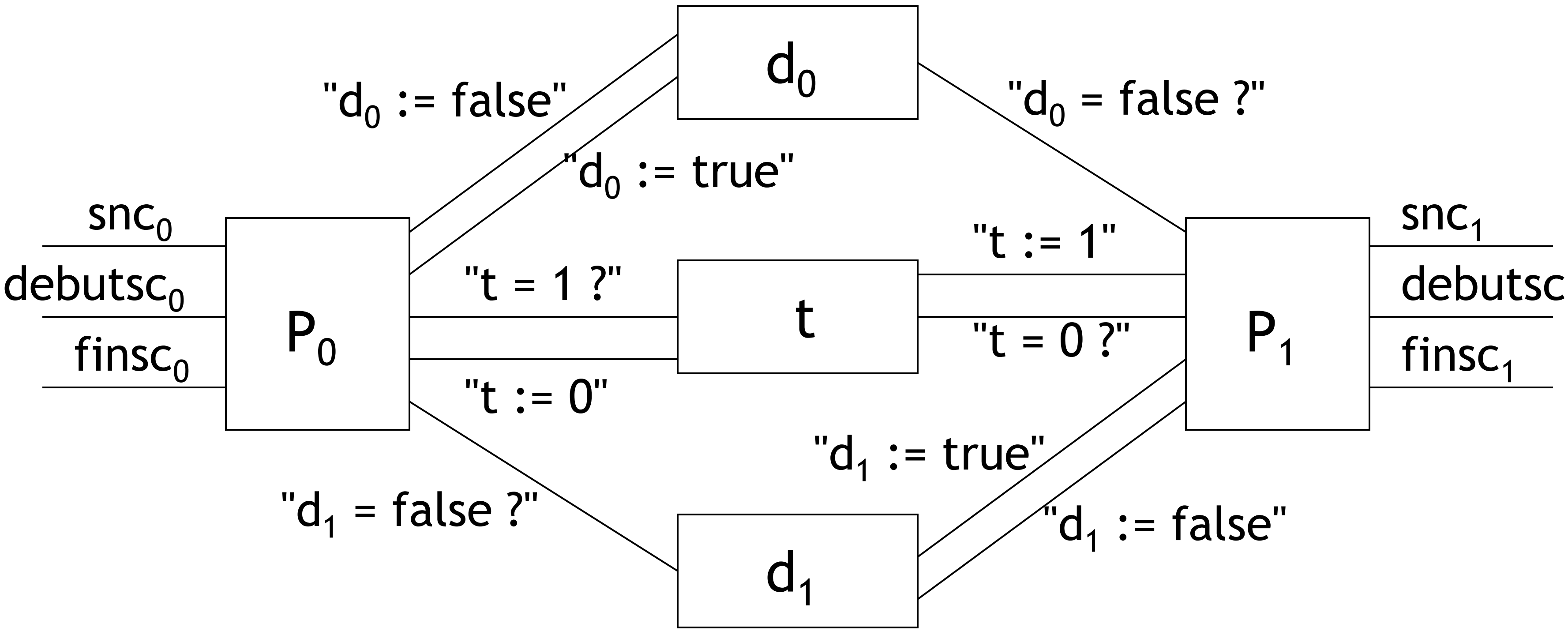




Peterson: Automata for d_0, d_1, and t



Peterson: System architecture (1/2)



- synchronized actions: "d_0 := false", "d_0 := true", ..., "t = 0 ?", ...
- non-synchronized actions: snc_0, snc_1, $debutsc_0$, ...

Peterson: System architecture (2/2)

The architecture can be expressed in several ways

$$(P_0 \otimes_{\varnothing} P_1) \otimes_{D0 \cup D1 \cup T} (d_0 \otimes_{\varnothing} d_1 \otimes_{\varnothing} t)$$

$$\big((P_0 \otimes_{\varnothing} P_1) \otimes_{D0 \cup D1} (d_0 \otimes_{\varnothing} d_1)\big) \otimes_{T} t$$

with $D0 = \{$ "d_0 := false", "d_0 := true", "d_0 = false ?" $\}$
$D1 = \{$ "d_1 := false", "d_1 := true", "d_1 = false ?" $\}$
$T = \{$ "t := 0", "t := 1", "t = 0 ?", "t = 1 ?" $\}$

Beware that $\otimes_L$ is not an associative operator:

if $L \neq L'$ then $(\exists P_1, P_2, P_3)\ (P_1 \otimes_L P_2) \otimes_{L'} P_3 \neq P_1 \otimes_L (P_2 \otimes_{L'} P_3)$

Building the product automaton

Adopted method: exhaustive enumeration of states

- Construction of the state space by exploring the transition relation forward from the initial state (forward reachability)
- Transitions are generated using R_1, R_2, and R_3
- When a new state is reached, one must verify whether it was already met ; in this case, one must loop back to the existing state
- Various exploration strategies exist: breadth-first, depth-first, guided by a criterion, ...

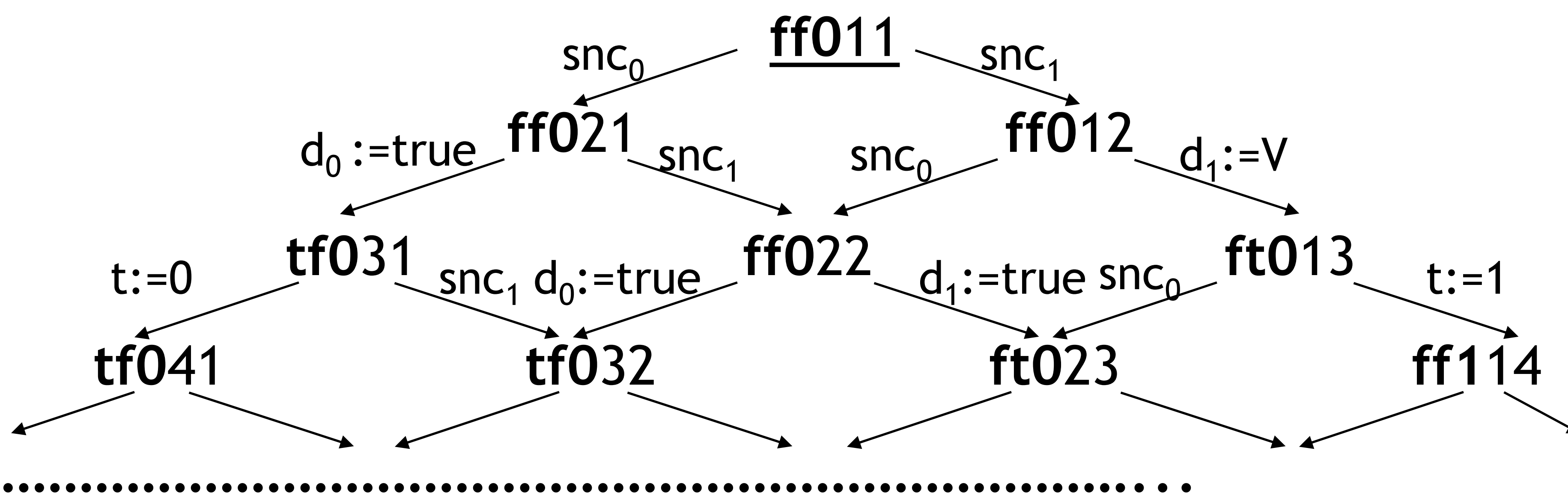
Peterson: Product automaton

$S = \{ \mathbf{f}, \mathbf{t} \} \times \{ \mathbf{f}, \mathbf{t} \} \times \{ \mathbf{0}, \mathbf{1} \} \times \{ 1..7 \} \times \{ 1..7 \}$

$A = \{ snc_0, snc_1, \ldots, \text{"}d_0 := true\text{"}, \ldots \}$

$s_0 = \langle \mathbf{f}, \mathbf{f}, \mathbf{0}, 1, 1 \rangle = \mathbf{ff0}11$

T =

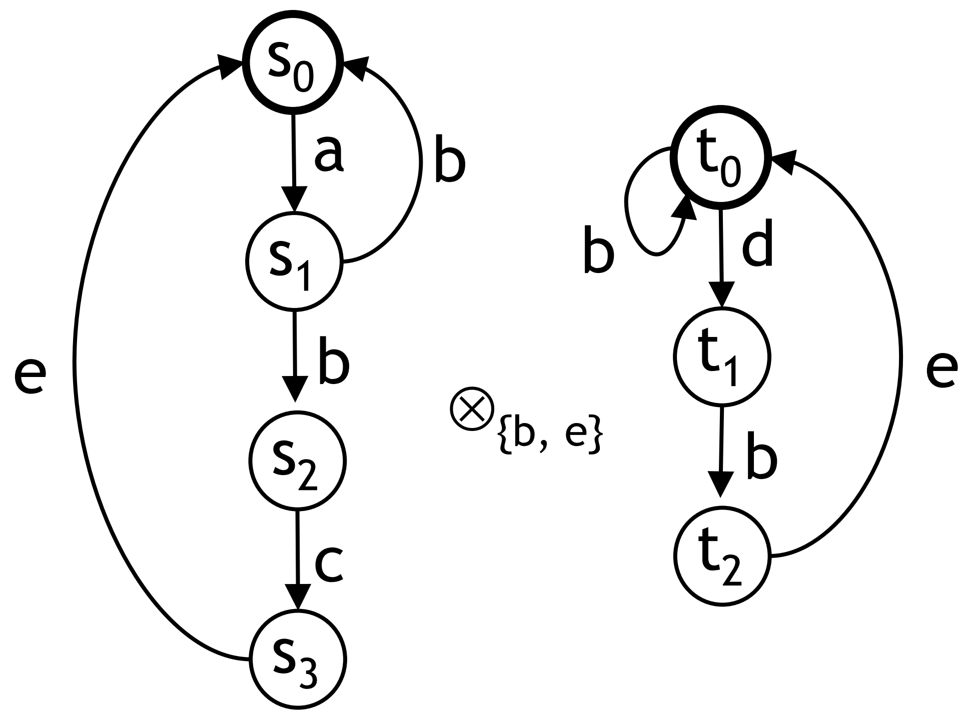


Remarks

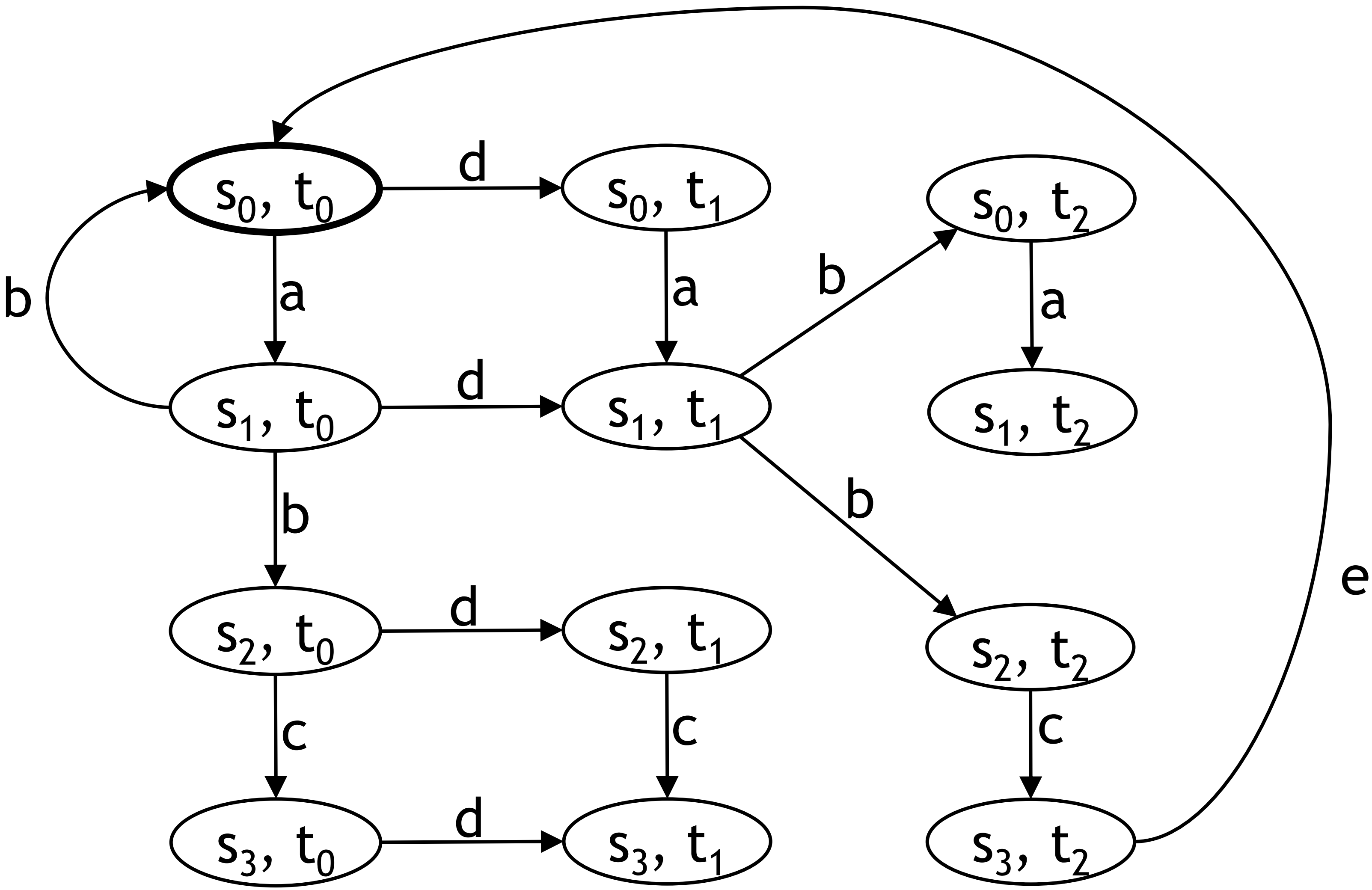
- The product automaton of the system is finite:
  $$|S| \leq 2 \times 2 \times 2 \times 7 \times 7 = 392$$
- Often, the set of states that are reachable from the initial state is much smaller than the cartesian product of variable values (forbidden transitions due to synchronization constraints)
  Peterson: ~50 states, ~110 transitions
- There are tools to build the product automaton and/or explore the transition relation automatically

Exercise

Compute the following product:



Solution



s_0, t_0
s_0, t_1
s_0, t_2
s_1, t_0
s_1, t_1
s_1, t_2
s_2, t_0
s_2, t_1
s_2, t_2
s_3, t_0
s_3, t_1
s_3, t_2
a
b
c
d
e

Verification

Once the product automaton is generated, various properties of the system can be verified automatically (model checking).

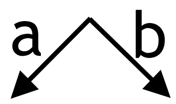
For the Peterson algorithm:

- deadlock freeness: every state has (at least) one sucessor
- mutual exclusion: for $i, j \in \{ 0, 1 \}$
  every sequence from $debutsc_i$ to $debutsc_j$ contains $finsc_i$
- starvation freeness: no process can monopolize the critical section indefinitely
- independent progress: each process can access the critical section if the other processes « do nothing »

Various forms of choice

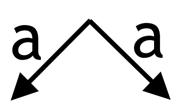
Classically, one distinguishes between:

- ***external choice*** (the environnement decides which branch will be taken)



the branch proposed by the environment will be chosen (if a and b are proposed: ND)

- ***internal choice*** (the system decides)



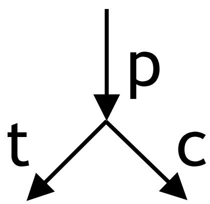
if the environment proposes action a, the system chooses a branch in a ND way

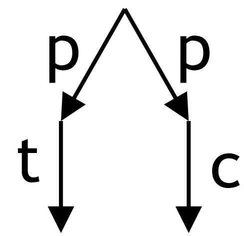
Comparison of LTS

Due to ND, beware when comparing two LTS

The coffee machine example

3 actions: p = the custommer introduces a coin
t / c = the custommer chooses tea / coffee

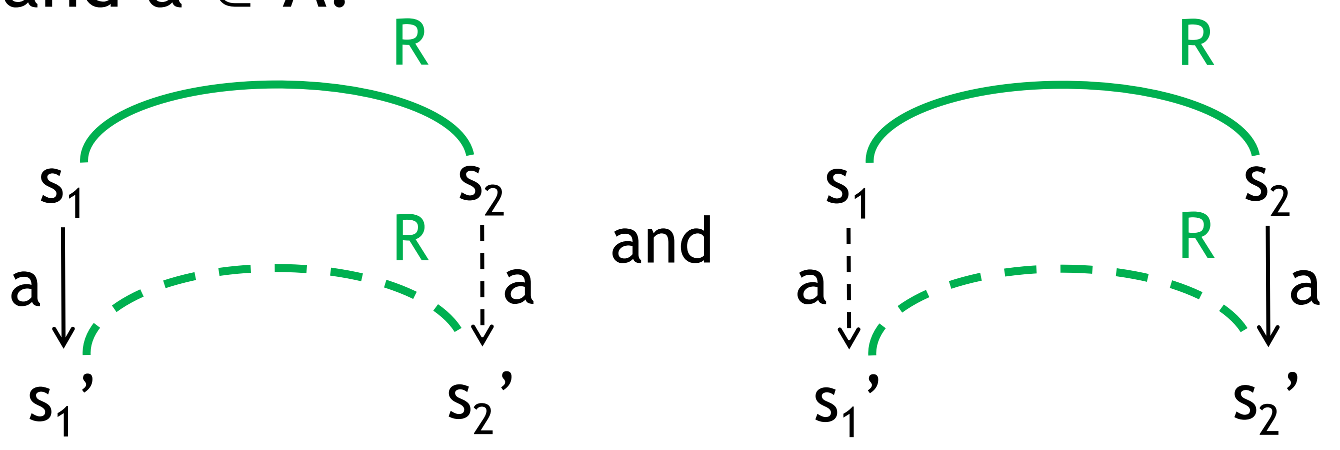
M_1 = 

M_2 = 

- M_1 and M_2 define the same language { p.t, p.c }
- Only M_1 is correct: possibility to choose t or c
- Equivalences stronger than language equivalence are necessary: bisimulations

Strong bisimulation

- **Goal**: build a relation between states that have « the same behaviour »
- Strong bisimulation is the strongest of such relations
- A relation R is a strong bisimulation if for all s_1, s_2 $\in$ S and a $\in$ A:



Solid line: universal quantification, dashed line: existential quantification

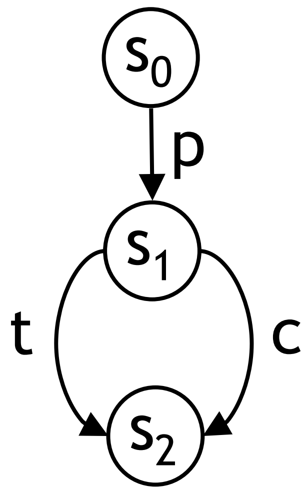
Strong bisimulation: formal definition

- R is a strong bisimulation if $\forall$ $(s_1, s_2) \in R$:
  1. $(\forall\ s_1\text{-}a\text{->}s_1')$ $(\exists\ s_2')$ $s_2\text{-}a\text{->}s_2'$ and $(s_1', s_2') \in R$
  2. $(\forall\ s_2\text{-}a\text{->}s_2')$ $(\exists\ s_1')$ $s_1\text{-}a\text{->}s_1'$ and $(s_1', s_2') \in R$
- Two states s_1 and s_2 are strong bisimilar ($s_1 \approx s_2$) iff there exists a strong bisimulation R st. $(s_1, s_2) \in R$
- Two LTS are strong bisimilar iff their initial states are strong bisimilar
- Without condition 2, the relation R is a strong simulation: s_1 is simulated by s_2 (s_2 simulates s_1), written $s_1 \leq s_2$

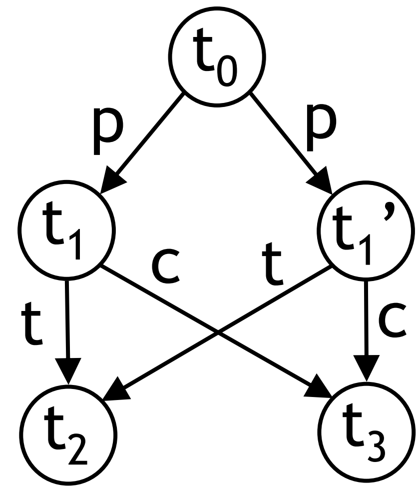
Remark

- To prove bisimilarity: build a relation R and show that each of its elements satisfies conditions 1 and 2
- To prove non-bisimilarity: find states that should verify the conditions but do not

Example



and



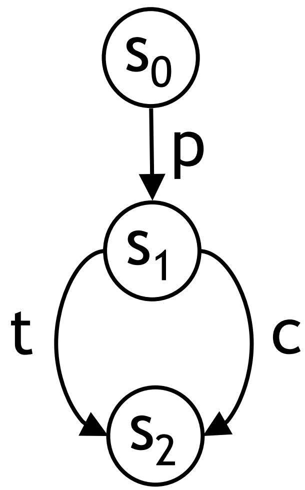
are bisimilar

Proof:

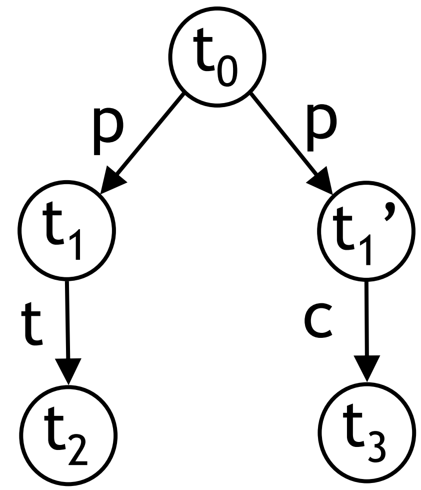
R = { (s_0, t_0), (s_1, t_1), (s_1, t_1'), (s_2, t_2), (s_2, t_3) } is a bisimulation because:

- For the pair (s_0, t_0) :
  - s_0-p-> s_1 and there exists t_0 -p-> t_1 st. $(s_1, t_1) \in R$
  - t_0 -p-> t_1 and there exists s_0 -p-> s_1 st. $(s_1, t_1) \in R$
  - t_0 -p-> t_1' and there exists s_0 -p-> s_1 st. $(s_1, t_1') \in R$
- For the pair (s_1, t_1): similar check…
- etc.

Example



and



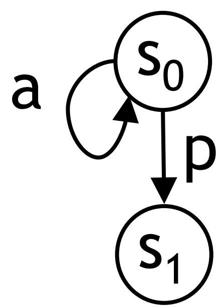
are not bisimilar

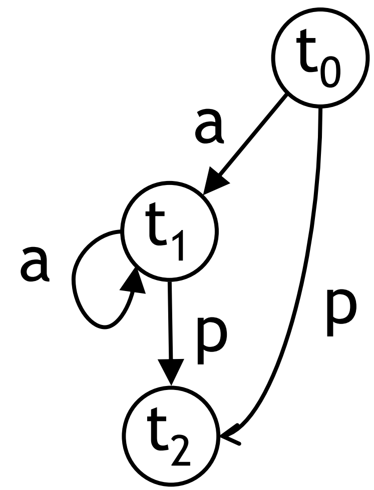
Proof:

- Assume $s_0 \approx t_0$
- Then, following action p, we must also have $s_1 \approx t_1$ and $s_1 \approx t_1'$
- To have $s_1 \approx t_1$, t_1 should have an outgoing transition labeled by c, which is not the case
- (a similar argument with t allows $s_1 \approx t_1'$ to be disproved)
- This disproves $s_0 \approx t_0$ – QED

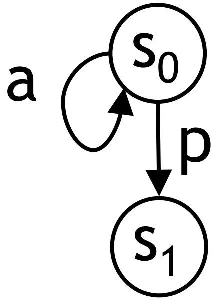
Exercise

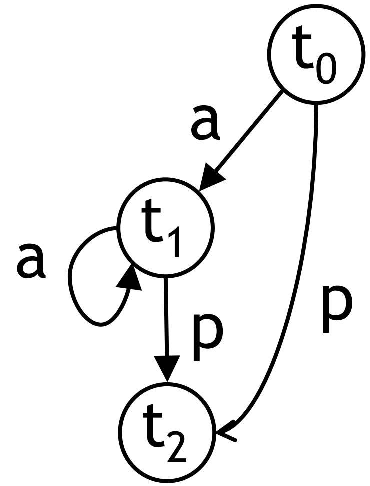
Are the following LTS bisimilar?





Solution





The answer is: yes

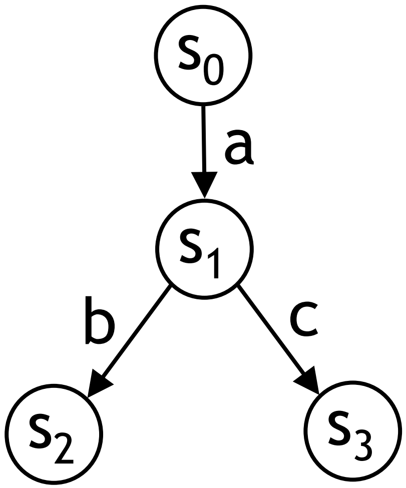
R = { (s_0, t_0), (s_0, t_1), (s_1, t_2) } is a strong bisimulation (homework: prove it)

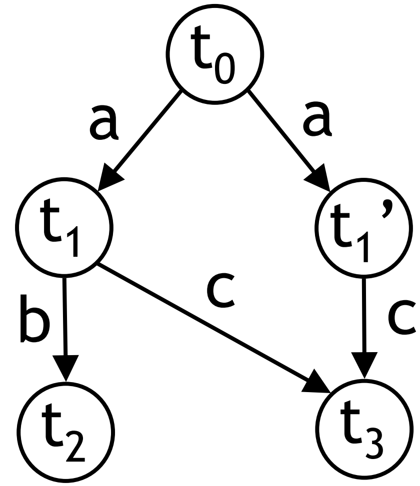
Exercise

1. Are the following LTS strong bisimilar?

2. Does the leftmost LTS simulate the rightmost LTS, and vice-versa ?

Solution





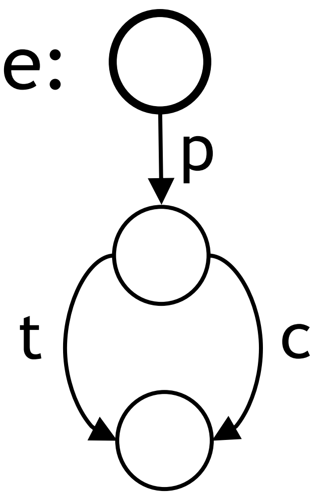
1. The answer is no: s_1 and t_1’ are not bisimilar, so neither are s_0 and t_0
2. The answer is yes in both cases

If two LTS are bisimilar then each one simulates the other

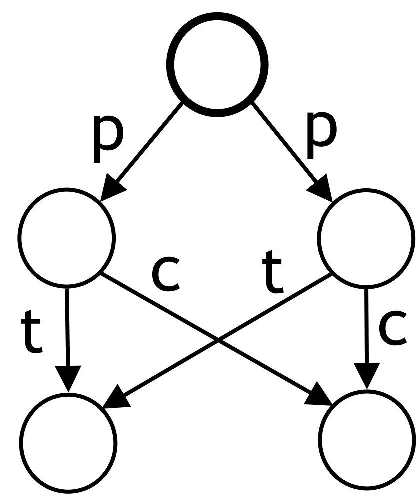
This exercise shows that the converse is false

Minimization

- To every LTS corresponds a unique LTS that is equivalent for strong bisimulation and whose number of states and transitions is minimal
- There exists an automated procedure allowing this minimal representative to be computed: minimization
- Example:



is the minimal representative of

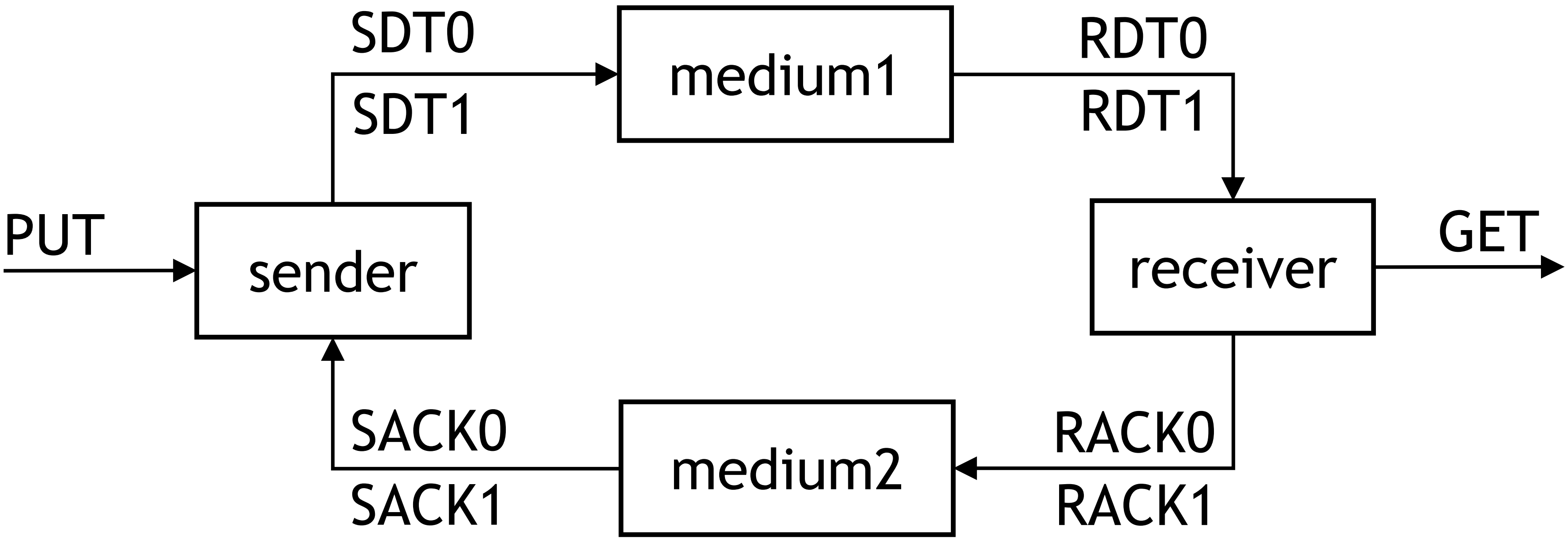


Internal action

- Special action, traditionally written τ
- Occurs in the LTS but « non observable » or « hidden » (non synchronizable)
- Possibility to abstract away (= rename into τ) the actions of the system that we do not need to observe, to fight against state space explosion
- Equivalences
  - Strong bisimulation: τ is handled as any other action
  - Weaker equivalences (e.g., branching bisimulation): transition s -τ-> s' can be compressed if s and s' lead to the same choices of visible actions

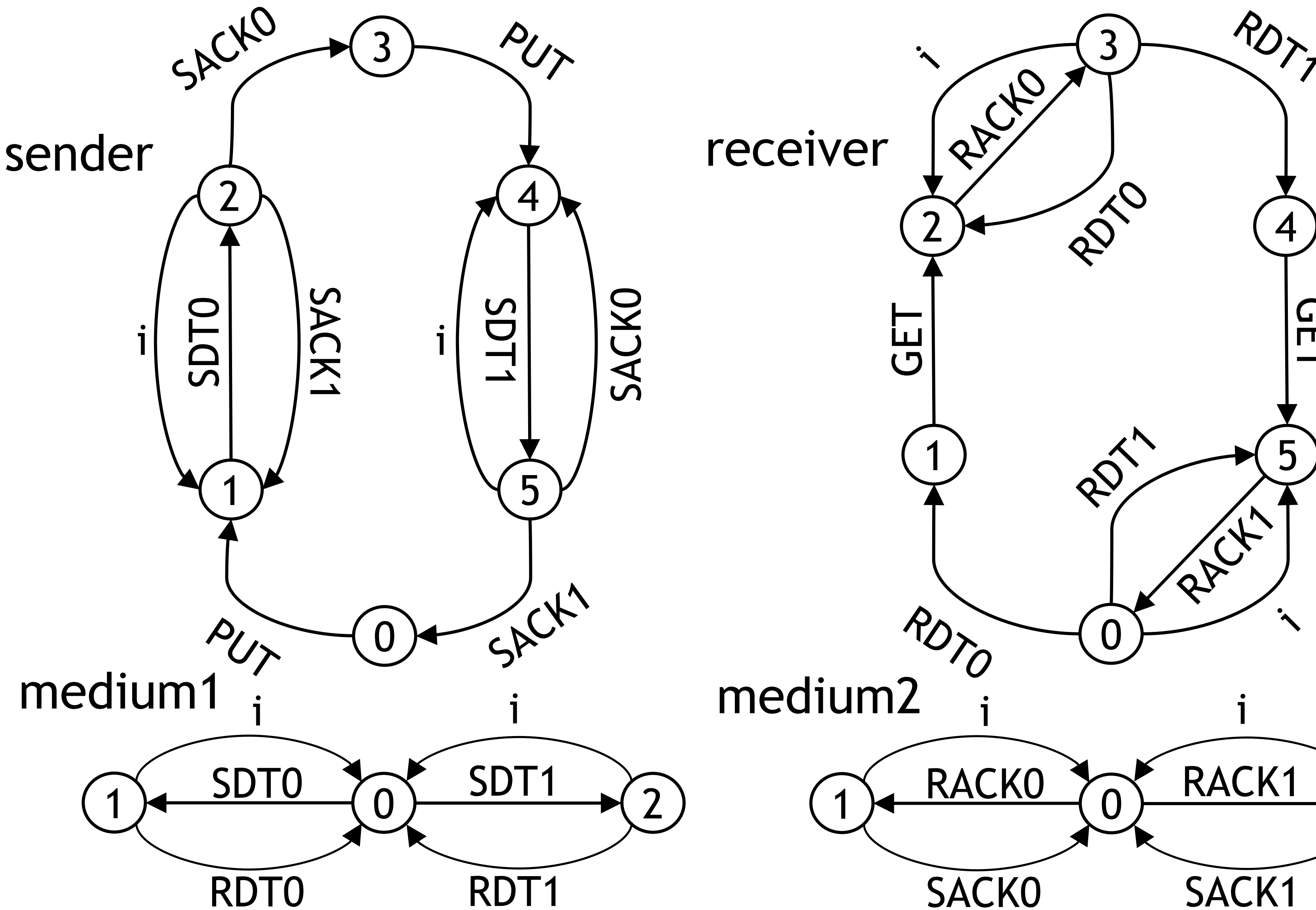
Example: The alternating-bit protocol

The alternating-bit protocol

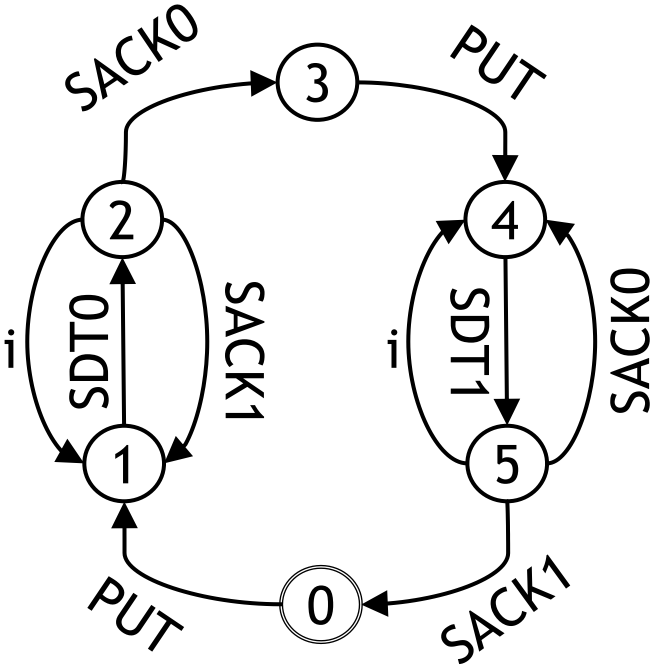


(sender $\otimes_{\emptyset}$ receiver)
$\otimes_{\{\text{ SDT0, SDT1, RDT0, RDT1, RACK0, RACK1, SACK0, SACK1 }\}}$
(medium1 $\otimes_{\emptyset}$ medium2)

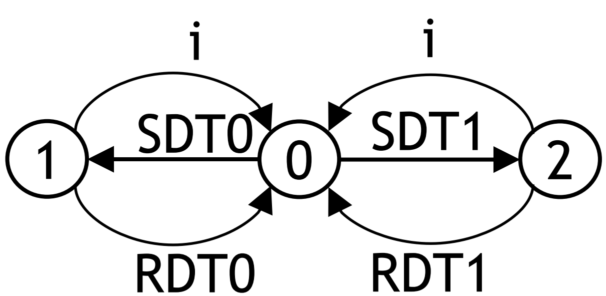
Automata


sender
SACK0
3
PUT
2
4
i
SDT0
SACK1
i
SDT1
SACK0
1
5
PUT
0
SACK1
receiver
i
3
RDT1
RACK0
2
RDT0
4
GET
GET
1
RDT1
5
RACK1
RDT0
0
i
medium1
i
i
SDT0
SDT1
1
0
2
RDT0
RDT1
medium2
i
i
RACK0
RACK1
1
0
2
SACK0
SACK1

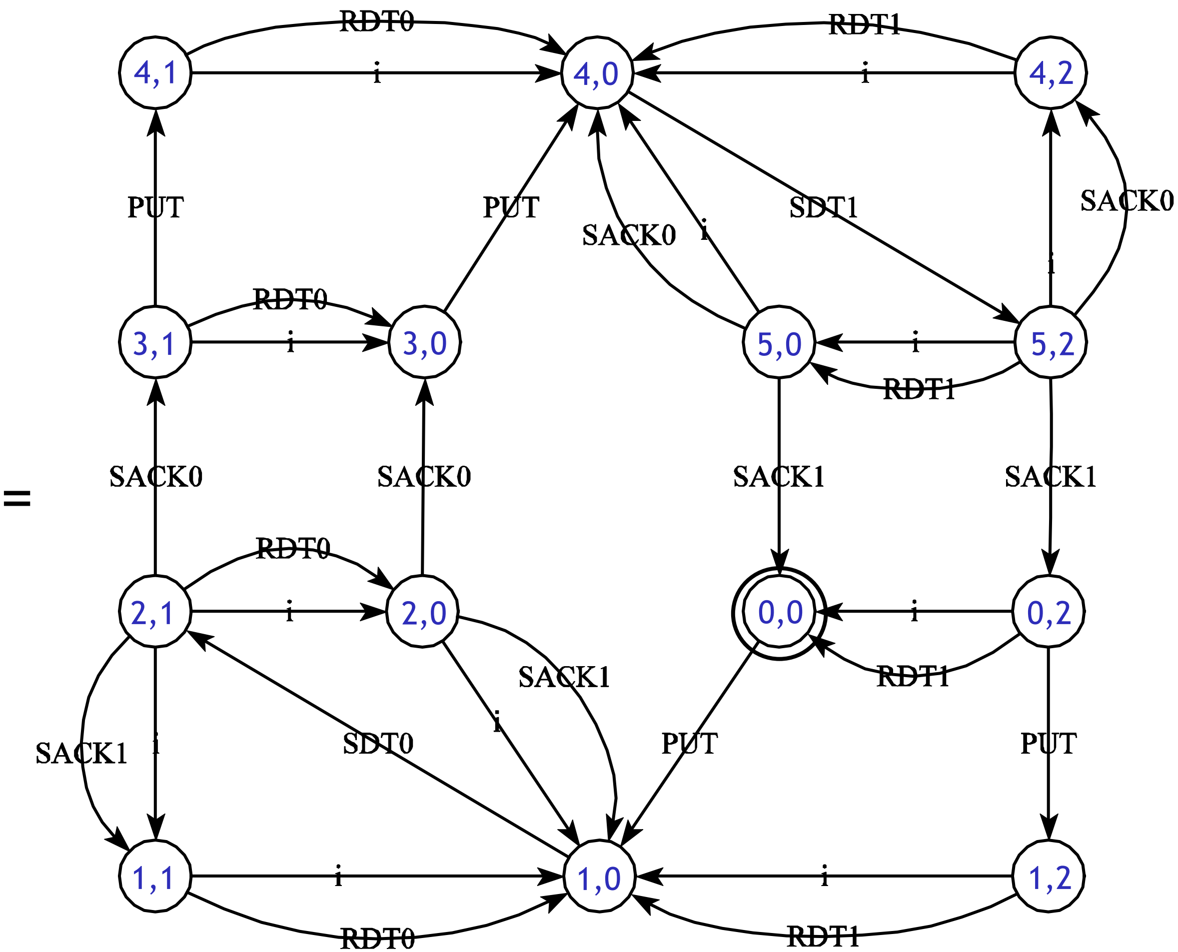
Product of sender and medium1



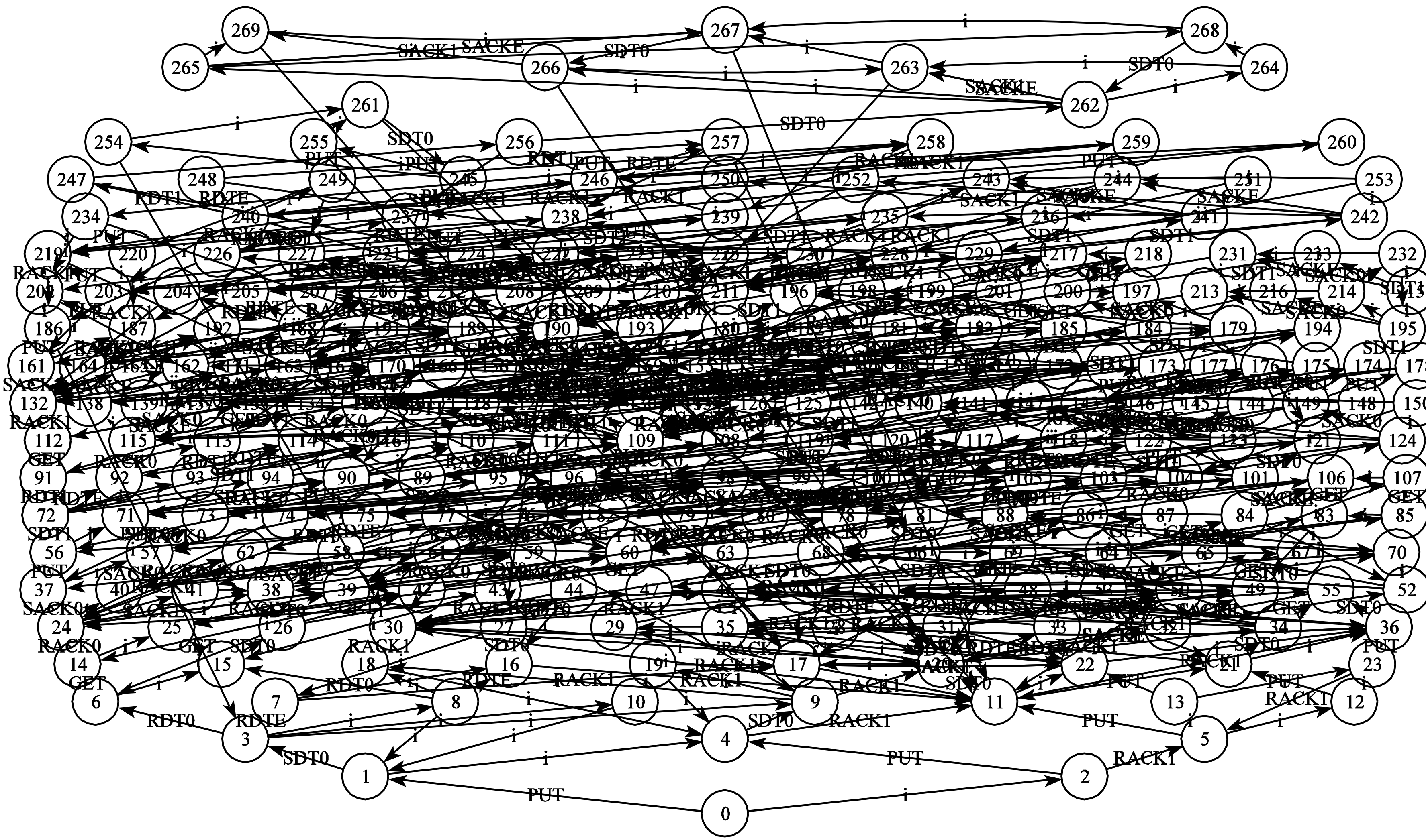
⊗{SDT0, SDT1}



=

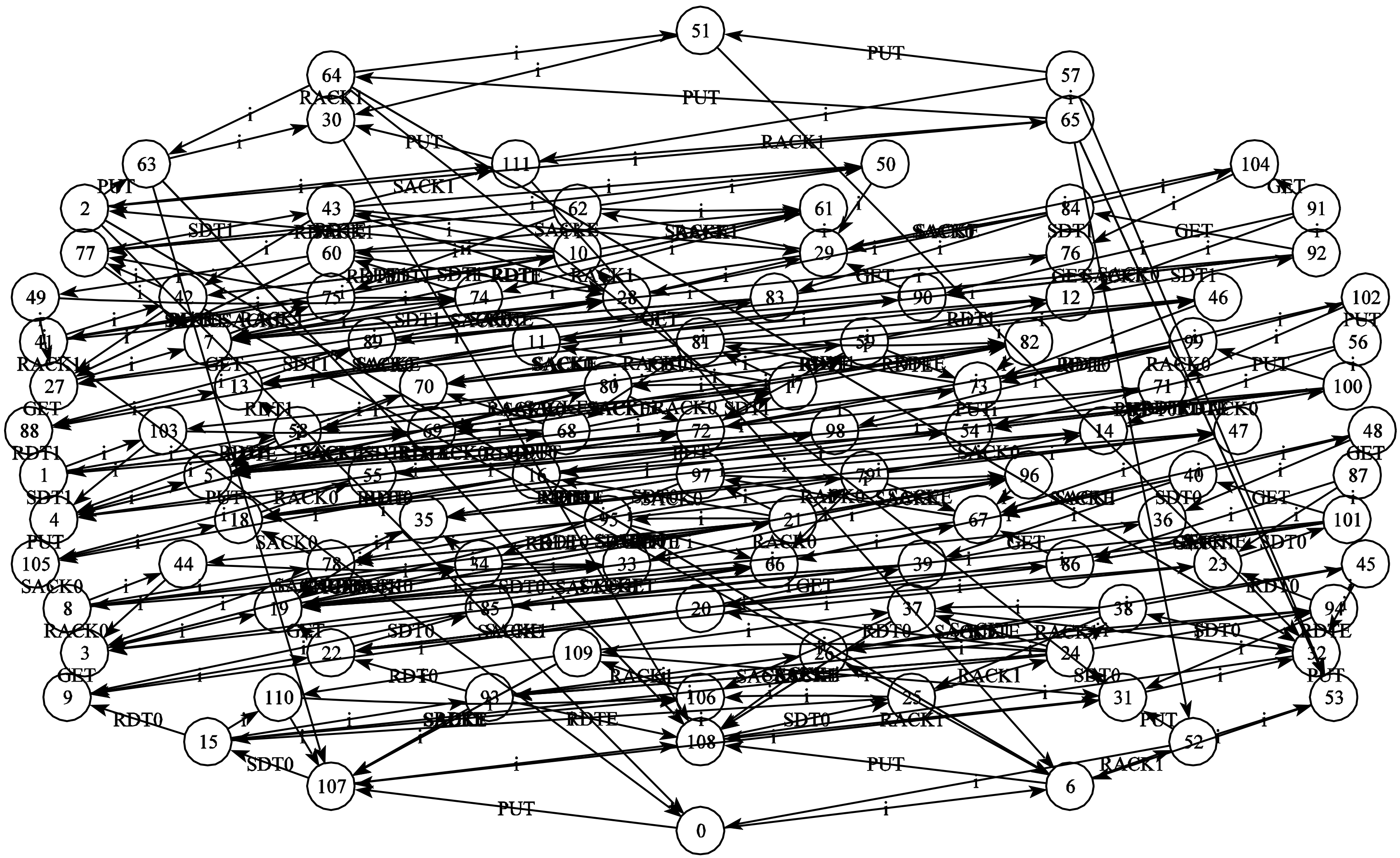


LTS of the full system



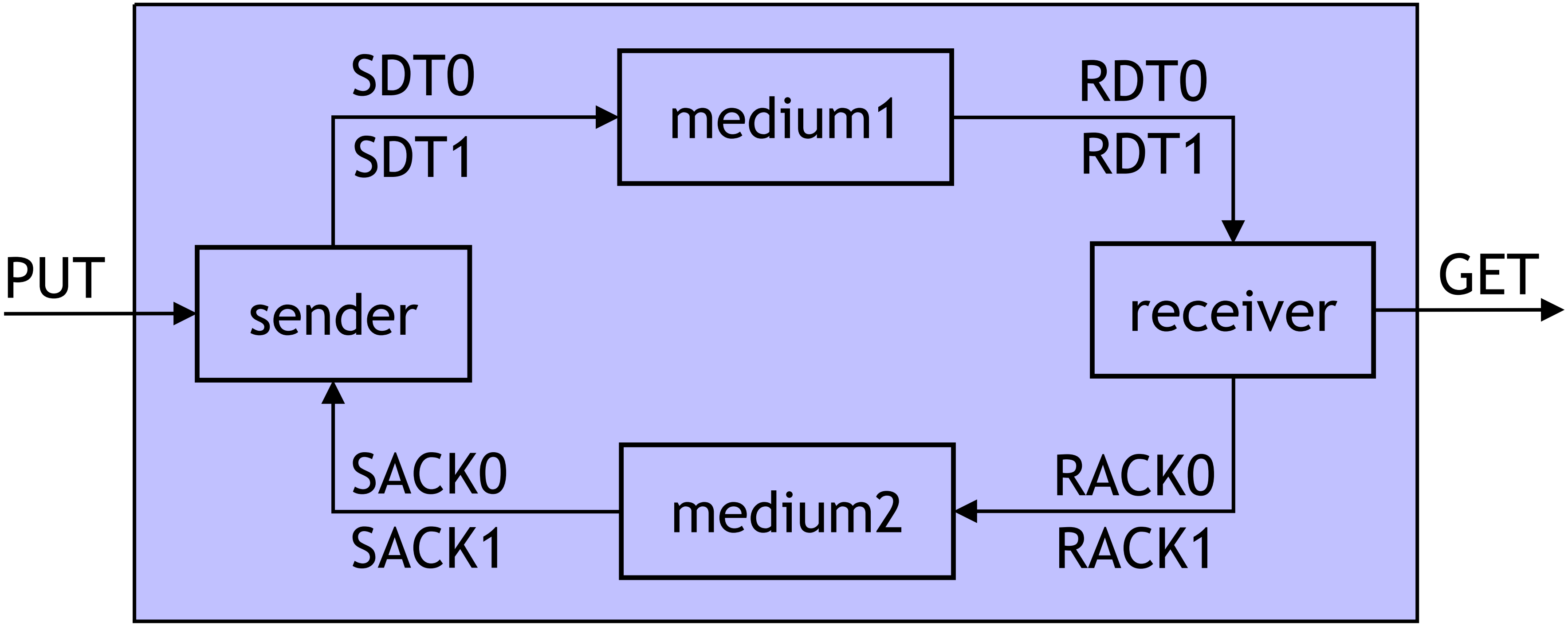
270 states, 773 transitions

LTS minimized for strong bisimulation



112 states, 380 transitions

Action hiding



hide SDT0, SDT1, RDT0, RDT1, RACK0, RACK1, SACK0, SACK1 in

(sender $\otimes_{\varnothing}$ receiver)

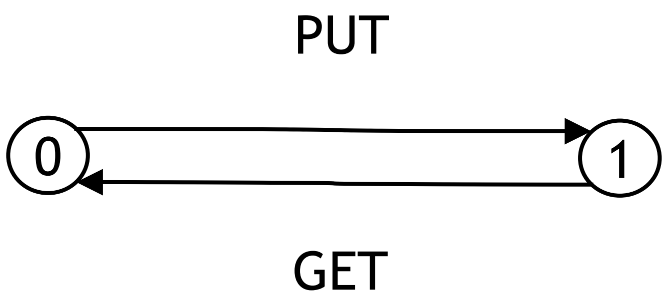
$\otimes_{\{\text{ SDT0, SDT1, RDT0, RDT1, RACK0, RACK1, SACK0, SACK1 }\}}$

(medium1 $\otimes_{\varnothing}$ medium2)

Action hiding and branching bisimulation

- « hide » renames a *visible action* « a » into the *internal action* « i » (or « τ »)

- After hiding and minimization for branching bisimulation, we get the following LTS for the alternating-bit protocol:



Synthesis on CA (1/2)

Advantages :

- Simple model to describe concurrency
- Introduce many concepts that we will use later on in this course
- Available CA handling tools:
  - Altarica (Université de Bordeaux, Labri)
  - CADP (http://cadp.inria.fr)
- Many industrial applications

Synthesis on CA (2/2)

Limitations :

- Risk of state space explosion when generating the product automaton (minimisation can help)
- Low-level model, difficult to read and maintain
- Limited modeling expressiveness
  - Static architecture: no dynamic creation or destruction of automata
  - Difficult to express: A then (B || C) then D
  - No modeling of data (one variable = one automaton): not acceptable for complex data types (int, list, struct,…)