Embedded and Real-Time Applications
A) Models for Time and Concurrency

Florence Maraninchi
www-verimag.imag.fr/~maraninx

Verimag/Grenoble INP

2007-2008

Part I

General Introduction

Outline

1. Objectives and Contents of the Course
2. Classifying Computer Systems
3. Constraints
4. Industrial State of the Art - examples
5. Summary

1 Objectives and Contents of the Course

2 Classifying Computer Systems

3 Constraints

4 Industrial State of the Art - examples

5 Summary

Objectives

- Understand what is an embedded application
- Learn the main mathematical models that help understanding time and concurrency

Prerequisites

- One Imperative Programming Language
- Operating Systems
- Automata and Languages
- Basics of circuits and machine architectures

1 Objectives and Contents of the Course

2 Classifying Computer Systems

3 Constraints

4 Industrial State of the Art - examples

5 Summary

Transformational Systems

Typical example : a compiler

Inputs at the beginning, then some finite time computation, outputs at the end.

A transformational system has to terminate.

Interactive Systems

Typical example: a man-machine interface

loop-based behavior (does not necessarily terminate), where inputs come all the time (human actions on buttons, mouse, keyboard) and outputs are produced all the time also (changes of the interface, effects on the underlying computer system).

Reactive Systems

Typical example: a heater controller.

The same as interactive systems, but the speed of the interaction is driven by the (physical) environment. The computer system should be sufficiently fast in order not to miss relevant evolutions of the environment.

1 Objectives and Contents of the Course

2 Classifying Computer Systems

3 Constraints

4 Industrial State of the Art - examples

5 Summary

Reactivity to an Environment, Real-Time

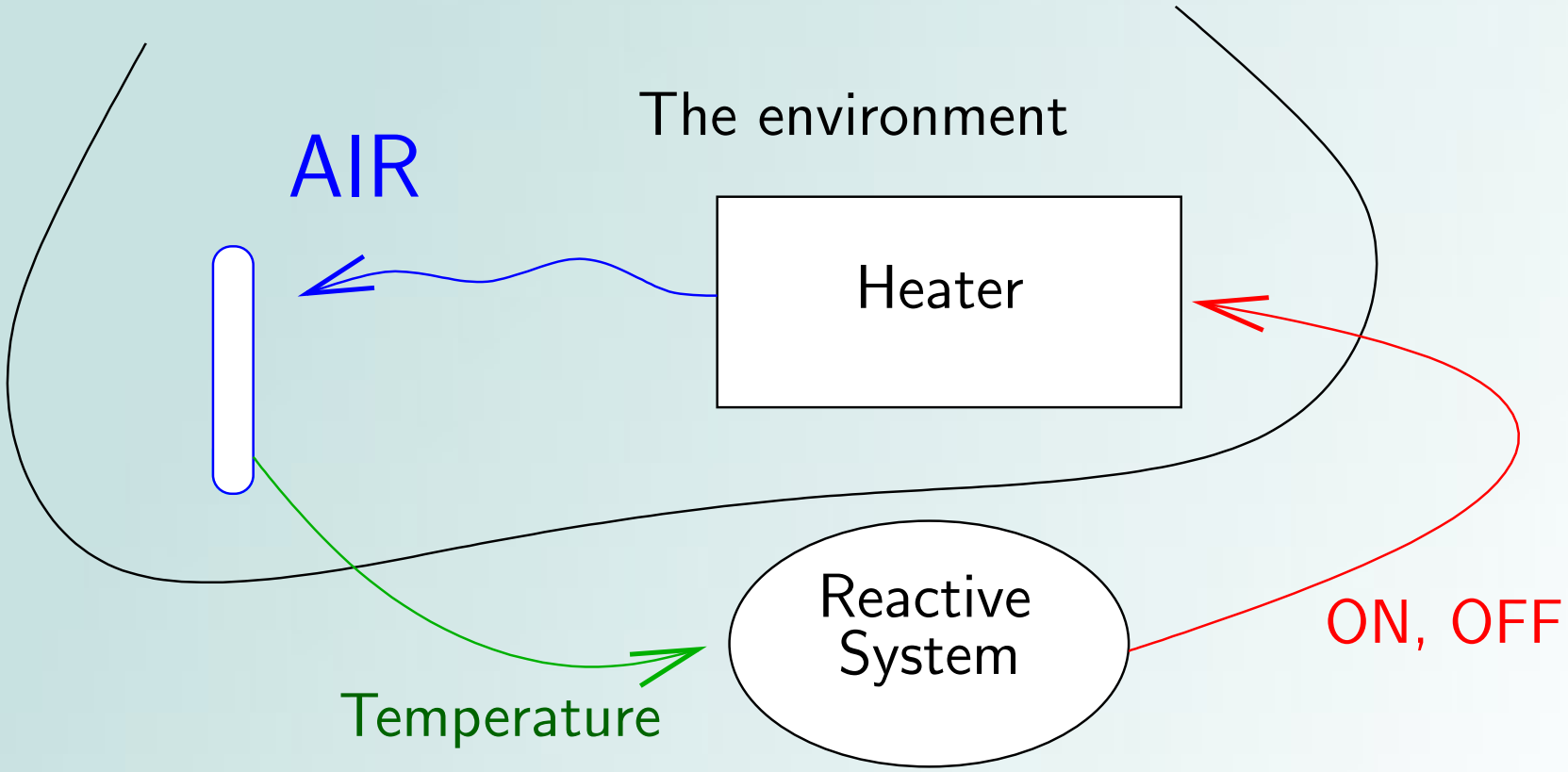
A typical real-time program:

```
initializations
while (true) {
      --- point (1)
      get inputs
            (read sensors)
      compute outputs
            and update memory
      write outputs
            on the actuators
      --- point (2)
}
```

The time it takes to execute the code between points (1) and (2) defines the sampling period of the program. This is real time.

The outputs to the environment may have some influence on future inputs. This is reactivity.

Reactivity of a heater controller



Real-Time Programming Problems

- Write code that is sufficiently fast (you're not always allowed to answer: "*try a bigger machine*")
- Be able to tell how fast your program is, *in advance* (Worst-Case-Execution-Time static evaluation)
- It's not always possible to write single-loop code, because of the *intrinsic parallelism* of a reactive system.
  e.g., between two independent sensor-computing-actuator lines

Criticity and Fault-tolerance

Criticity:
A fault is very "expensive" (human lives, damage to the environment, ...).

Try and verify properties of the computer system before it is used.

Criticity and Fault-tolerance

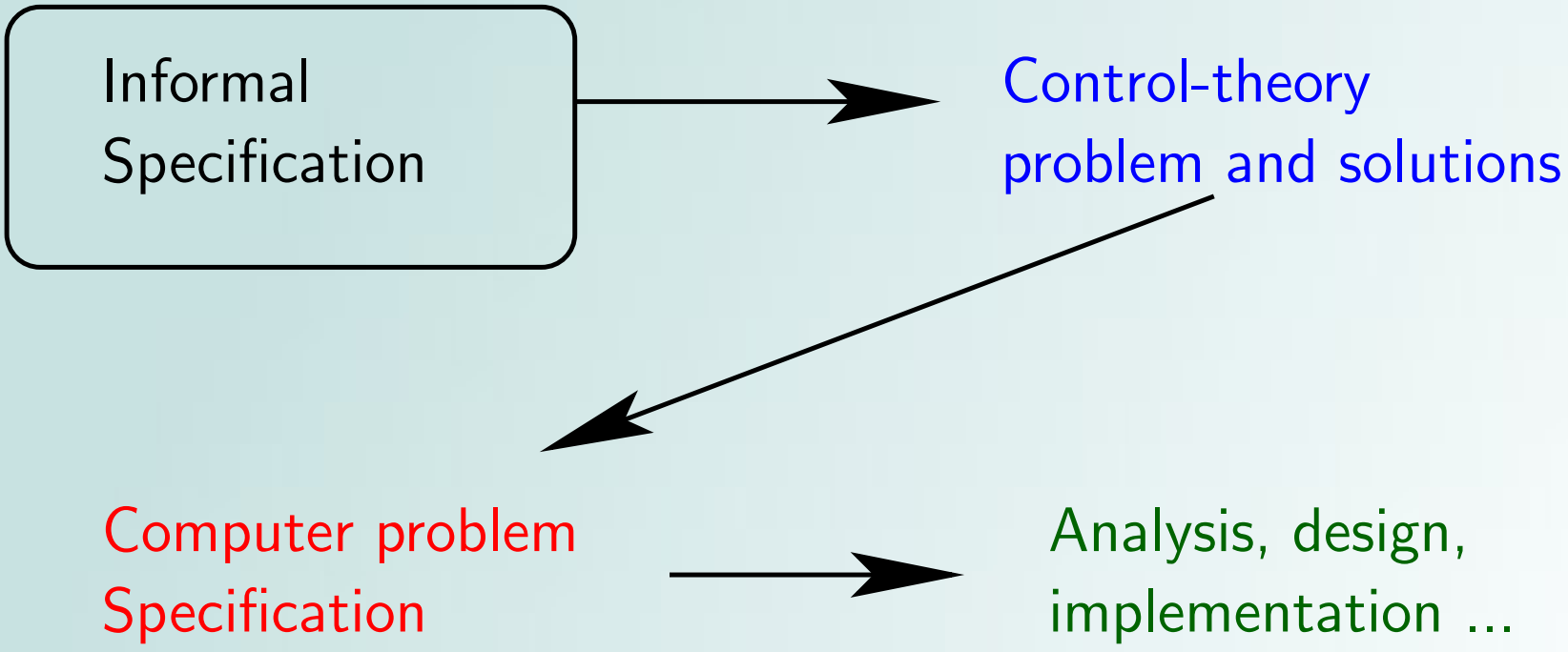
Criticity:
A fault is very "expensive" (human lives, damage to the environment, ...).

Try and verify properties of the computer system before it is used.

HW Fault-tolerance:
Example: several sensors of the same kind, plus a voter

SW Fault-tolerance:
Example: several copies of the same code, running on several processors, or several distinct versions developed independently from the same specification (example: the flight-commands in the Airbus aircraft).

Relationship with control engineering



Hardware, software or mixed solutions? Discussion

Pure hardware:

Hardware, software or mixed solutions? Discussion

Pure hardware: more efficient, more difficult to develop, more "static" hence easier to analyse...

Pure software:

Hardware, software or mixed solutions? Discussion

Pure hardware: more efficient, more difficult to develop, more "static" hence easier to analyse...

Pure software: less efficient, more flexible, more "dynamic", less easy to analyse...

Mixed solutions:

Hardware, software or mixed solutions? Discussion

Pure hardware: more efficient, more difficult to develop, more "static" hence easier to analyse...

Pure software: less efficient, more flexible, more "dynamic", less easy to analyse...

Mixed solutions: try and get the advantages of boh HW and SW, and not the drawbacks...

Centralized or Distributed Systems?

Fact: centralized systems are far simpler than distributed ones.

But:

- Fault-tolerance forces distribution
- Physical position of sensors and computers also forces distribution
- Efficiency sometimes requires distribution

Distributed real-time programming is very hard.

Validation and Certification

Validation:
Simulation, automatic testing, formal verification, ... are methods that help in analysing (functional) properties of a computer system before it is deployed.

Validation and Certification

Validation:
Simulation, automatic testing, formal verification, ... are methods that help in analysing (functional) properties of a computer system before it is deployed.

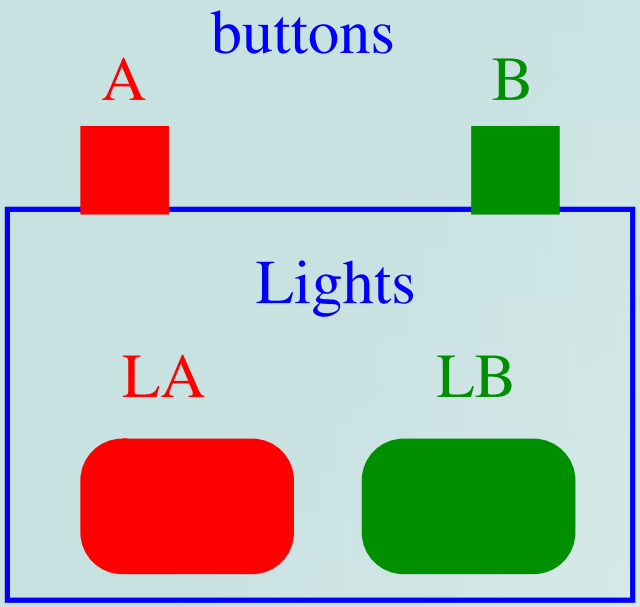
Certification:
Examples: the DO178B norm for civil avionics, common criteria for smart cards, ...

General-Purpose or Domain-Specific Languages?

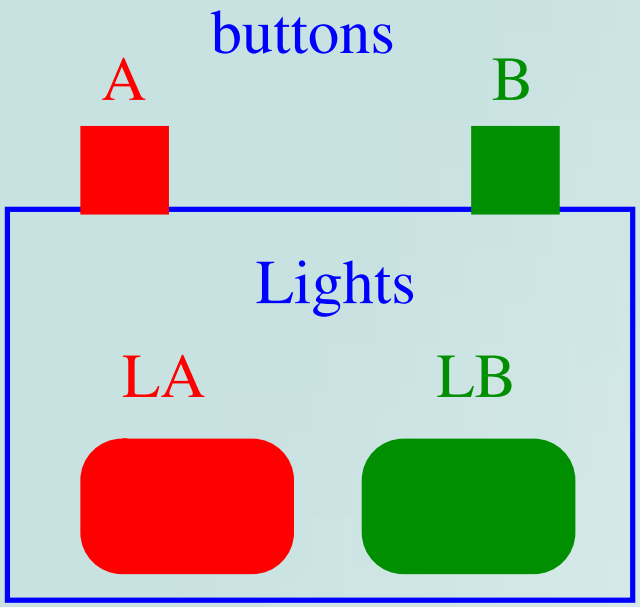
Ada, Java for real-time programming?

A DSL (Domain-Specific Languages) may have specific operations dedicated to reading sensors, writing actuators, synchronizing on time, etc., but it also has less than general-purpose languages.
Example: a programming language for embedded systems should not allow to write programs for which the memory is not bounded (implies: no recursion, no dynamic allocation. See Lustre later)

A Special Note on Parallel Systems (1)



A Special Note on Parallel Systems (1)



How to program such a system?

Behavior to be programmed:
— Each time A is pressed, toggle light A
— Each time B is pressed, toggle light B

A Special Note on Parallel Systems (2)

```
StateA := OFF ;
while (true) {
    read button A
    if (buttonA) {
        StateA := not StateA ;
    }
    if (StateA) {
        LightA.ON
    }
    else {
        LightA.OFF
    }}
```

```
StateB := OFF ;
while (true) {
    read button B
    if (buttonB) {
        StateB := not StateB ;
    }
    if (StateB) {
        LightB.ON
    }
    else {
        LightB.OFF
    }}
```

This is a solution for (one button, one light). How to describe two of them in parallel?

A Special Note on Parallel Systems (3)

```
StateB := OFF ; StateA := OFF ;
while (true) {
    read button B ; read button A ;
    if (buttonB) { StateB := not StateB ; }
    if (buttonA) { StateA := not StateA ; }
    if (StateB)  { LightB.ON } else { LightB.OFF }
    if (StateA)  { LightA.ON } else { LightA.OFF }
}
```

A solution with static scheduling: the code produced is sequential, but the high-level language may be parallel.

1 Objectives and Contents of the Course

2 Classifying Computer Systems

3 Constraints

4 Industrial State of the Art - examples
- Simulink in the automotive industry
- SCADE in the avionics industry
- SystemC for Systems-on-a-Chip

5 Summary

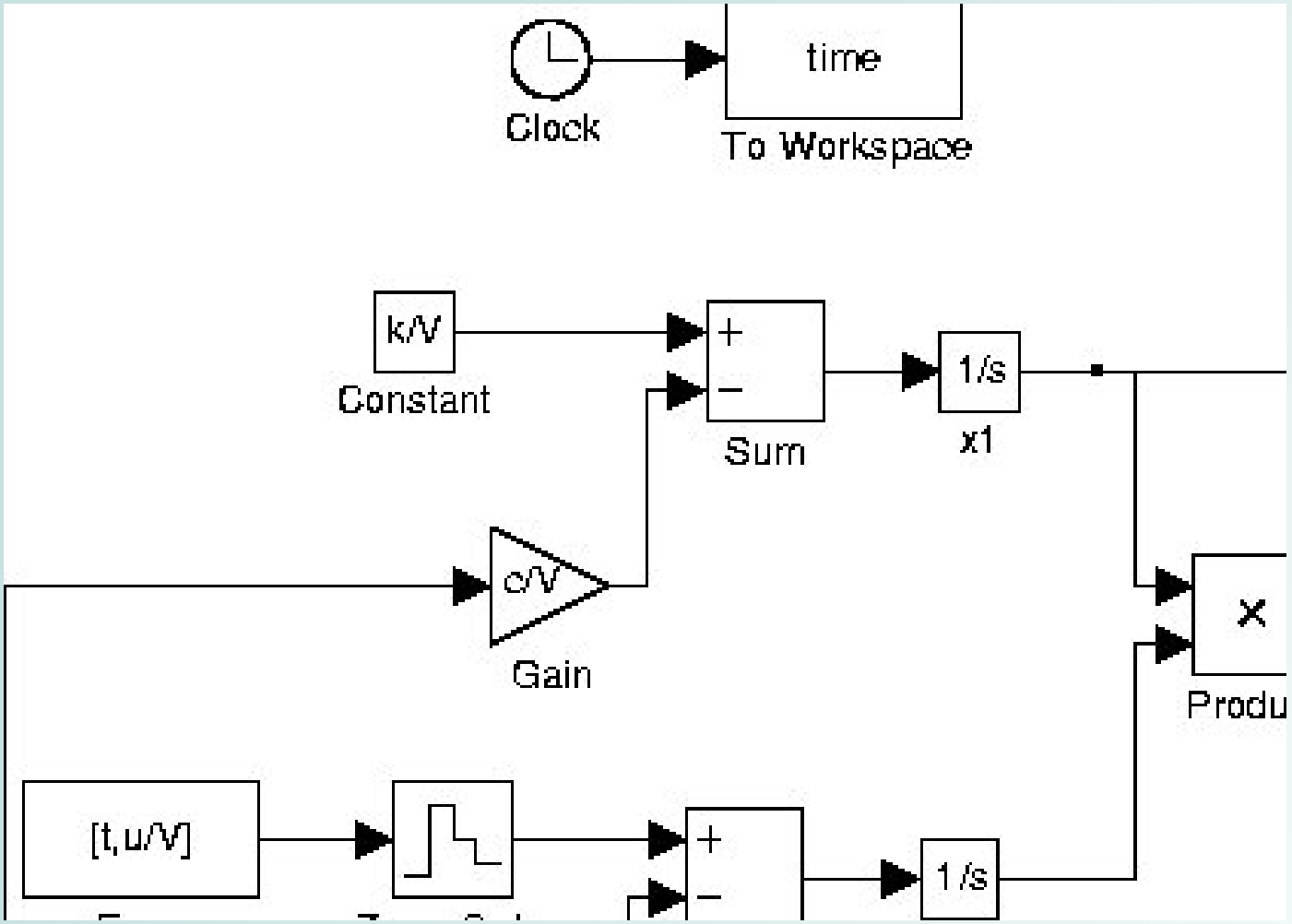
4 Industrial State of the Art - examples

- Simulink in the automotive industry
- SCADE in the avionics industry
- SystemC for Systems-on-a-Chip

Recommended readings

http://www.dspace.de/ww/fr/pub/systems.htm
http://www.interfacebus.com/Design_Connector_CAN.html
http://www.tttech.com/

A Simulink Diagram



Development from Simulink

- A continuous control problem and solution, including a model of the environment
- A discrete solution for the controller part
- An implementation. *Automatic code generation from Simulink?* or manual encoding, considering the diagrams as a detailed specification?

A complete chain from Simulink to embedded code is an instance of the general model-driven approaches.

4 Industrial State of the Art - examples

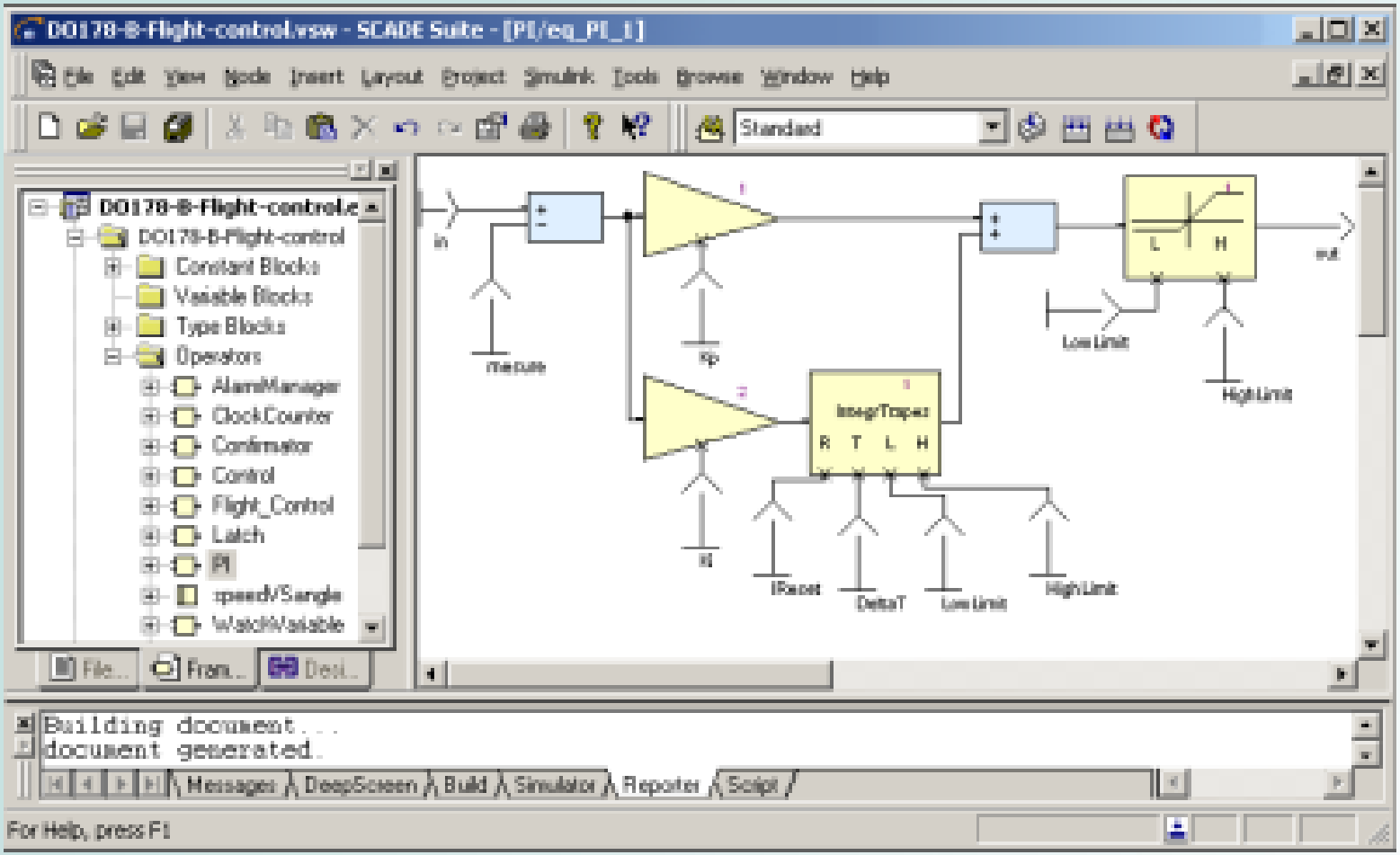
- Simulink in the automotive industry
- SCADE in the avionics industry
- SystemC for Systems-on-a-Chip

Recommended readings - Lustre and SCADE

```
@article{ halbwachs91synchronous,
    author = "N. Halbwachs and P. Caspi and
              P. Raymond and D. Pilaud",
    title = "The synchronous data-flow
             programming language {LUSTRE}",
    journal = "Proceedings of the IEEE",
    volume = "79",
    number = "9",
    month = "September",
    year = "1991"}
```

www.esterel-technologies.com/products/scade-suite/overview.html

A Scade diagram



Main features

- A formal language
- Powerful bi-directional interface to requirements management tools like DOORS.
- KCG Code Generator qualification eliminates the need for low level testing.
- Verification tools
- *Import and reuse of Simulink block diagrams and Stateflow diagrams into SCADE.*

4 Industrial State of the Art - examples

- Simulink in the automotive industry
- SCADE in the avionics industry
- SystemC for Systems-on-a-Chip

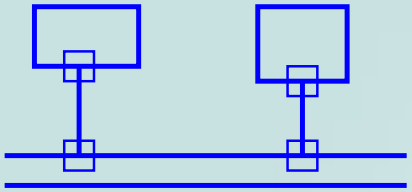
Recommended readings

http://www.us.design-reuse.com/ipsoc2004/

A new abstraction level: TLM, and SystemC

RTL :
+ precise (time/data)
- slow simulations
+ synthesis tools

A new abstraction level: TLM, and SystemC



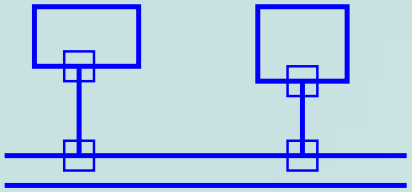
TLM :

+ allows to run SW
+ fast simulations
+ "easy" to write
- not precise (time/data)
- no synthesis tools

RTL :

+ precise (time/data)
- slow simulations
+ synthesis tools

A new abstraction level: TLM, and SystemC



TLM :

+ allows to run SW
+ fast simulations
+ "easy" to write

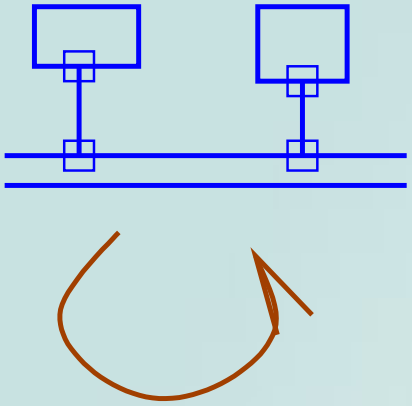
- not precise (time/data)
- no synthesis tools

Reference models

RTL :

+ precise (time/data)
- slow simulations
+ synthesis tools

A new abstraction level: TLM, and SystemC



validation
(func., non-func.)

TLM :

+ allows to run SW
+ fast simulations
+ "easy" to write

- not precise (time/data)
- no synthesis tools

Reference models

RTL :

+ precise (time/data)
- slow simulations
+ synthesis tools

A new abstraction level: TLM, and SystemC

TLM :

+ allows to run SW
+ fast simulations
+ "easy" to write

- not precise (time/data)
- no synthesis tools

Reference models

validation
(func., non-func.)

Comparison

RTL :

+ precise (time/data)
- slow simulations
+ synthesis tools

1 Objectives and Contents of the Course

2 Classifying Computer Systems

3 Constraints

4 Industrial State of the Art - examples

5 Summary

Characteristics of Embedded Applications

- Hardware / software / OS / middleware that are designed together.
- continuous / discrete, digital / analog
- synchronous / asynchronous / Globally-Asynchronous-Locally-Synchronous / ...
- several timing granularities (e.g., from RTL to TLM for SoCs)
- several data granularities
- several degrees of dynamicity (from simple reconfigurations to mobile code...)

Difficulties, Constraints

- Real-Time
- Functional correctness
- Power Consumption and other extra-functional properties
- Fault-Tolerance
- A wide variety of practises, languages, cultures, ...

Beyond Embedded Systems...

Autonomous, intelligent, ambient...

References

www.college-de-france.fr/default/EN/all/inn_tec/index.htm

leçon inaugurale : Pourquoi et comment le monde devient numérique

15/02/2008
Cours, 10h30-11h30 : Les systèmes embarqués et l'informatisation des objets.
Séminaire, 11h30-12h30 : La certification, ou comment faire confiance au logiciel pour l'avionique critique, par Gérard Ladier (Airbus).

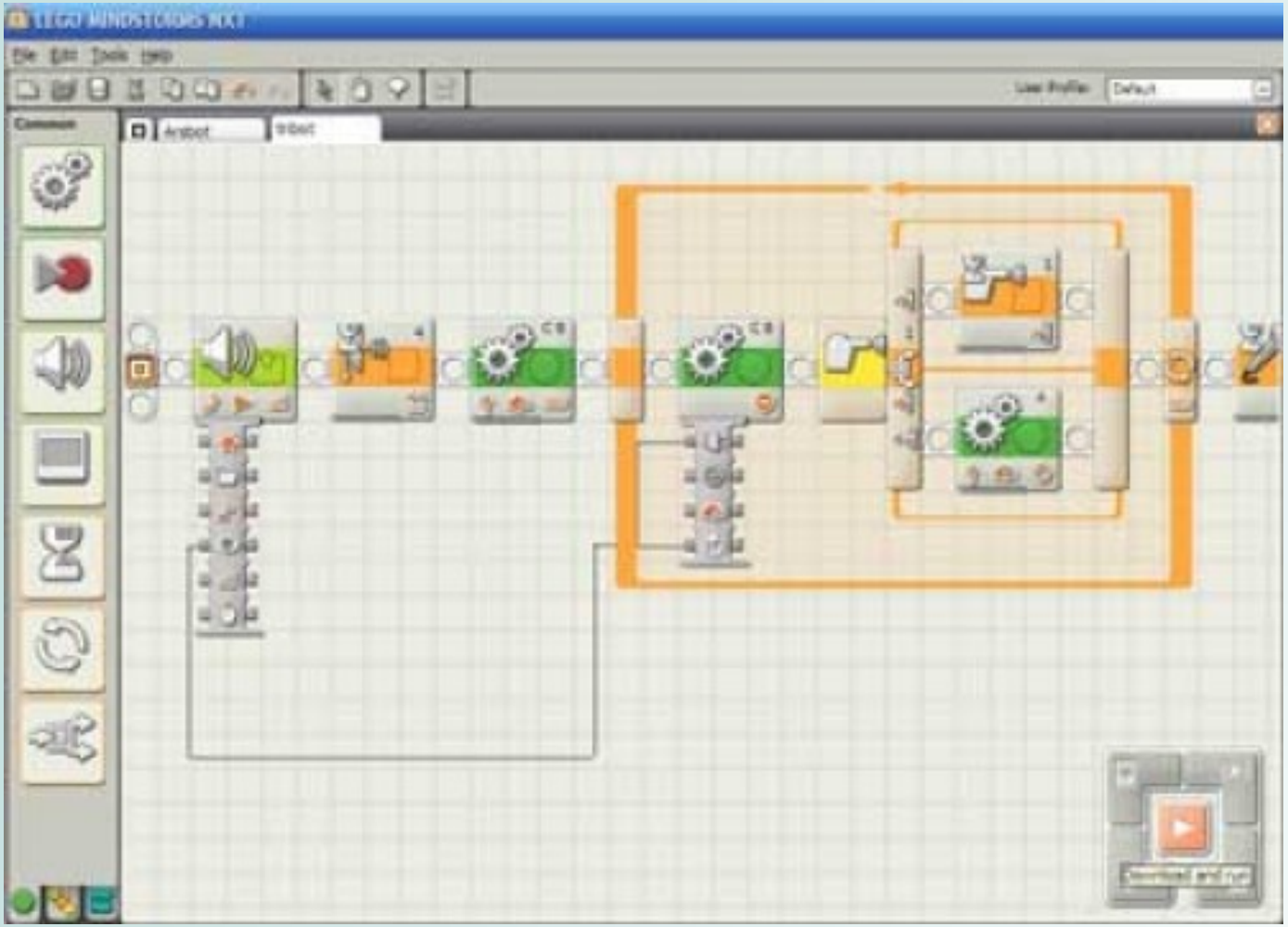
Video et mp3 à récupérer sur le Web.

Course Synopsys

- Models
- Using Models (defining languages, verification, ...)
- Real-Time Programming (RTOS)
- NEW: Programming an Embedded Application: Experiments with the Lego Mindstorms

Lego Mindstorms

Lego Mindstorms
“programming” language for kids



Experiments with the Lego Mindstorms

- Study an Embedded Execution Platform (ARM + "middleware")
- Program a Real-Time Application in Lustre/SCADE
- Understand Concurrency and Timing Problems

Part II

Models

Outline

6 Why Models?

7 Synchronous Models

8 Asynchronous Models

6 Why Models?

7 Synchronous Models

8 Asynchronous Models

Motivations

- Industrial practice for embedded applications : an ever growing wide variety of languages and models

Motivations

- Industrial practice for embedded applications : an ever growing wide variety of languages and models
- Computer science : a relative stable and small set of well understood models of computation

Motivations

- Industrial practice for embedded applications : an ever growing wide variety of languages and models
- How to understand the relationship between the two ?
- Computer science : a relative stable and small set of well understood models of computation

Classifying models

- Synchronous models are good at: modeling parallelism between processes that share a common clock; implementing a "logical" parallelism that will be compiled into sequential code

Classifying models

- Synchronous models are good at: modeling parallelism between processes that share a common clock; implementing a "logical" parallelism that will be compiled into sequential code
- Asynchronous models are good at: modeling parallelism when no common clock exists;

Classifying models

- Synchronous models are good at: modeling parallelism between processes that share a common clock; implementing a "logical" parallelism that will be compiled into sequential code
- Asynchronous models are good at: modeling parallelism when no common clock exists;
- GALS (Globally Asyncnhronous Locally Synchronous) models mix the two

What to do with models?

- Understand the behavior of parallel and timed systems

What to do with models?

- Understand the behavior of parallel and timed systems
- Build a language with a well defined semantics

What to do with models?

- Understand the behavior of parallel and timed systems
- Build a language with a well defined semantics
- Perform formal validation (verification, test, ...)

6 Why Models?

7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

8 Asynchronous Models

7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

Simple Mealy machines

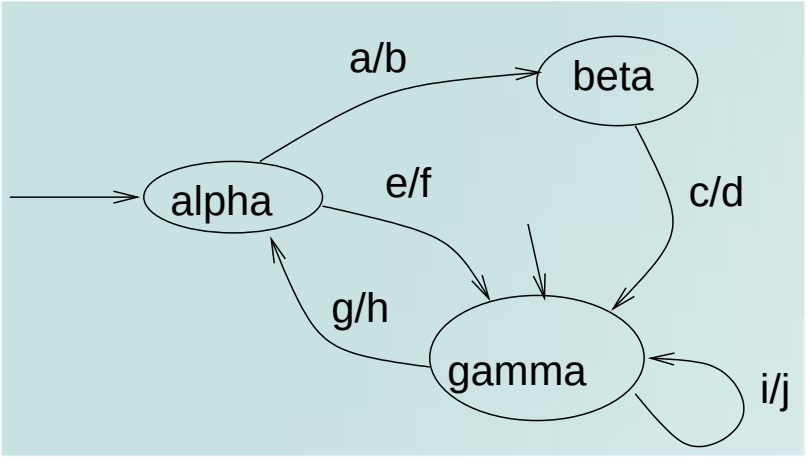
(cf. 1A course)

$M = (Q, Q_0 \subseteq Q, T \subseteq Q \times I \times O \times Q)$ where

- I (resp. O) is the set of inputs (resp. outputs)
- Q_0 is the set of initial states

Drawings: $q \xrightarrow{\text{input/output}} q'$

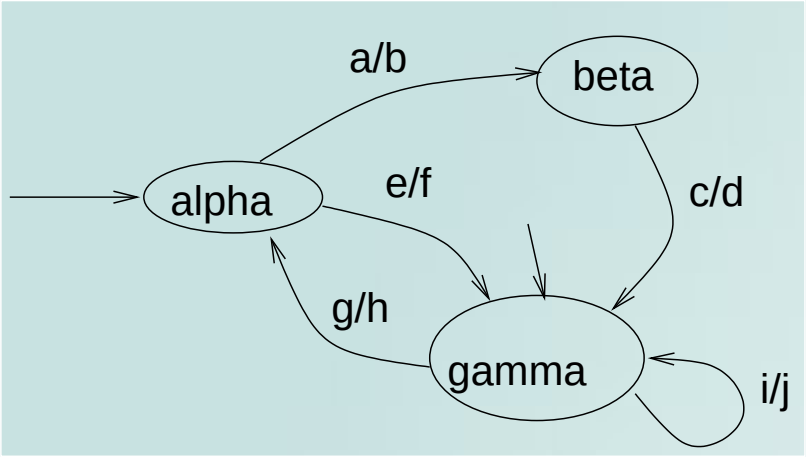
Trace Semantics



State	alpha	alpha	beta	gamma	gamma
Input	z	a	c	i	g
Output		b	d	j	h

time

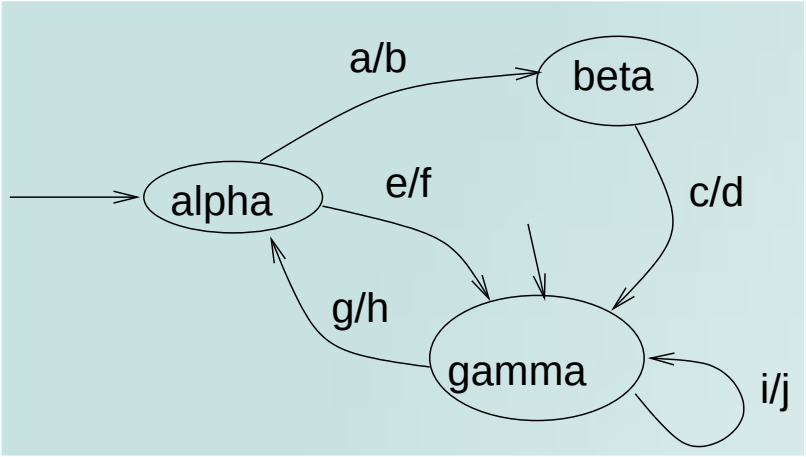
Trace Semantics



State	alpha	alpha	beta	gamma	gamma
Input	z	a	c	i	g
Output		b	d	j	h

time

Trace Semantics



State	alpha	alpha	beta	gamma	gamma
Input	z	a	c	i	g
Output		b	d	j	h

time

Imperative Program

```
State := alpha  -- take one of the initial states
while (true) loop
  get (I) ; O := null ;
  case State :
     when alpha  =>  if I = a then State := beta ; O := b ;
                     elsif I = e then ...
     when beta   =>  if I = c then State := gamma ; O := d ;
     when gamma  =>  if I = i then O := j
                     elsif I = g then O:=h ; State := alpha
  end case
  put (O)
end loop
```

Boolean Mealy Machines

Same definition, but with $I = 2^{\mathsf{BI}}$ and $O = 2^{\mathsf{BO}}$.

BI : input signals, any configuration is possible.
BO : output signals, any configuration is possible.

Determinism

$$\mid Q_0 \mid = 1 \quad \wedge$$

$$\forall q \in Q, \left\{ \begin{array}{l} \exists (q, m_1/o_1, q_1) \in T \wedge \\ \exists (q, m_2/o_2, q_2) \in T \end{array} \right\} \quad \Longrightarrow \; m_1 \wedge m_2 = \text{false}$$

Reactivity

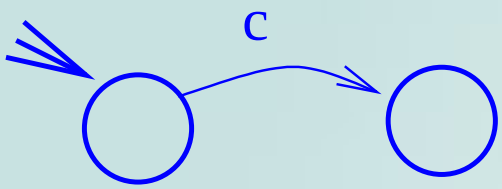
$$\forall q \in Q, \quad \bigvee_{(q,m_i/o_i,q_i)\in T} m_i \quad = \text{true}$$

Examples of BMMs - exercices

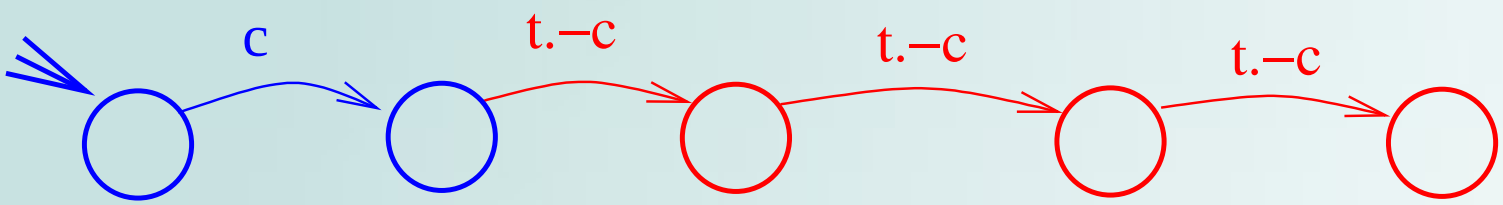
- Double-click (if there are two clicks in less than 4 tops (of a clock) it is a double click, otherwise it is a simple click
- Machine for recognizing `a* b a` continuously
- Decoder for a 4-digit door code (e.g.: the last 4 digits are : 1234)
- ...

Mouse Double Click

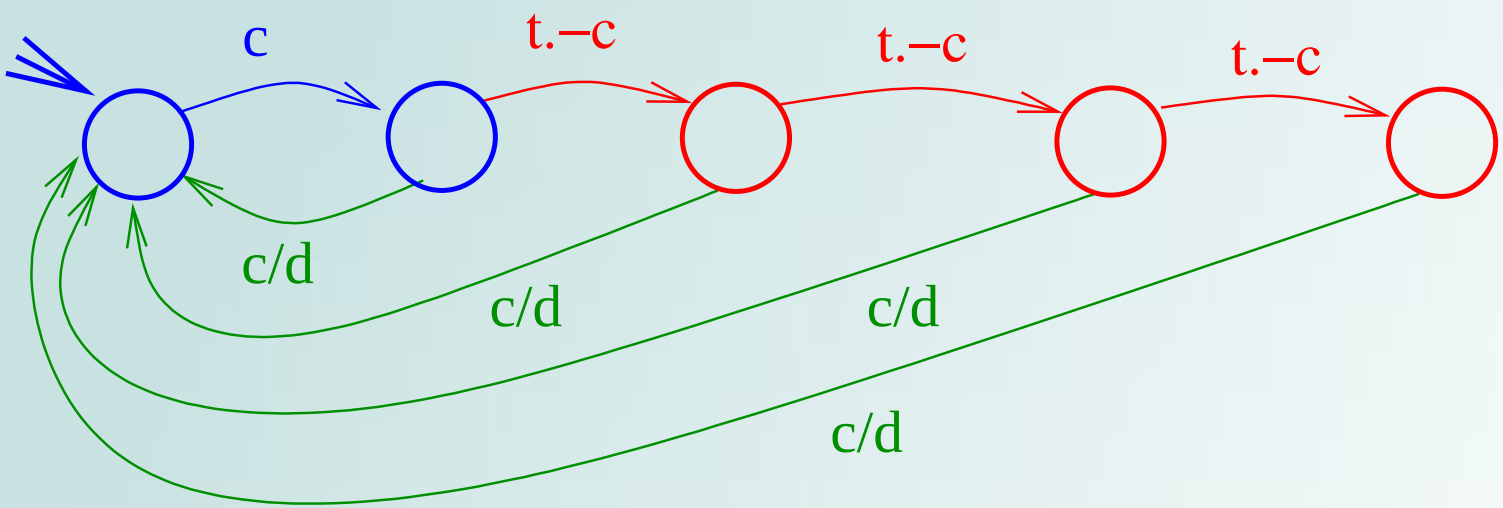
Mouse Double Click



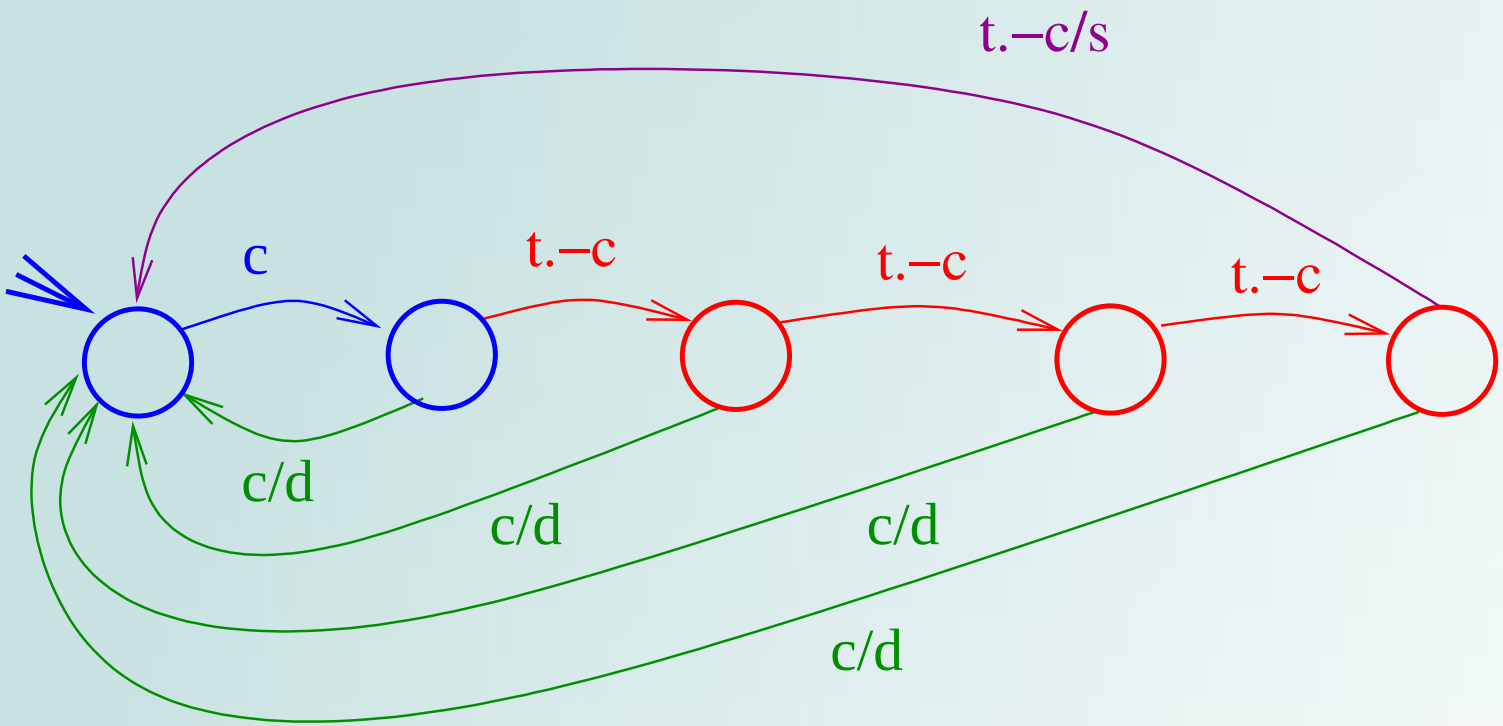
Mouse Double Click



Mouse Double Click

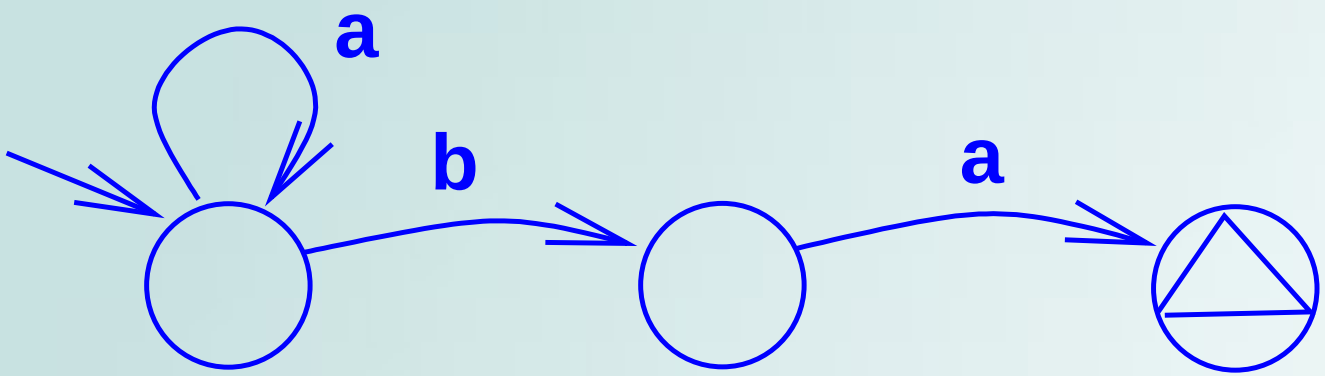


Mouse Double Click

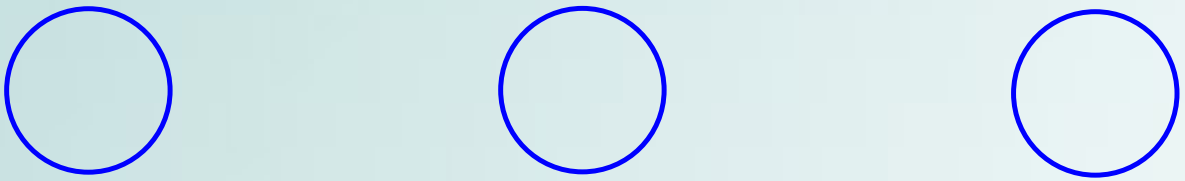


Machine for recognizing `a* b a`

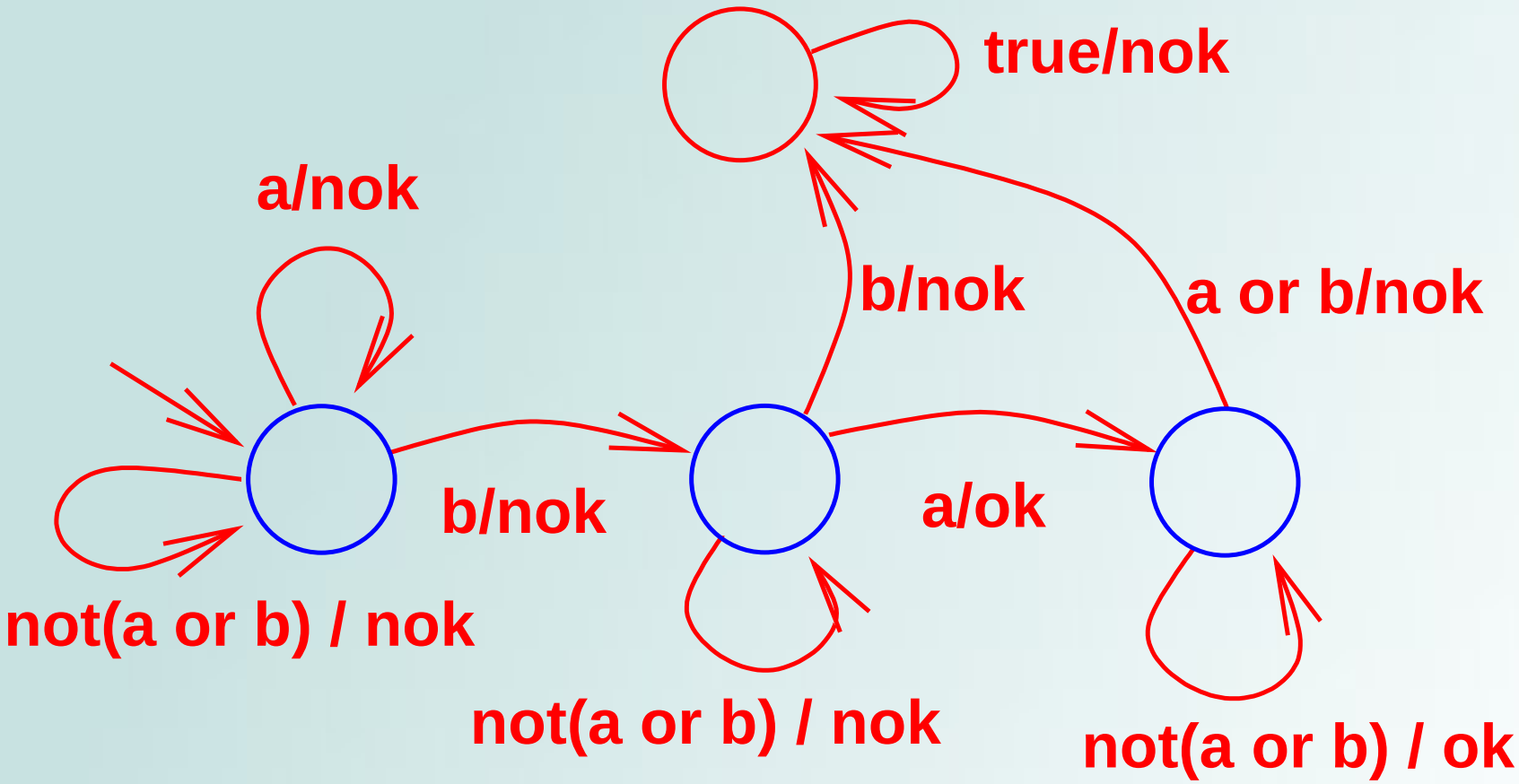
Machine for recognizing a* b a



Machine for recognizing `a* b a`



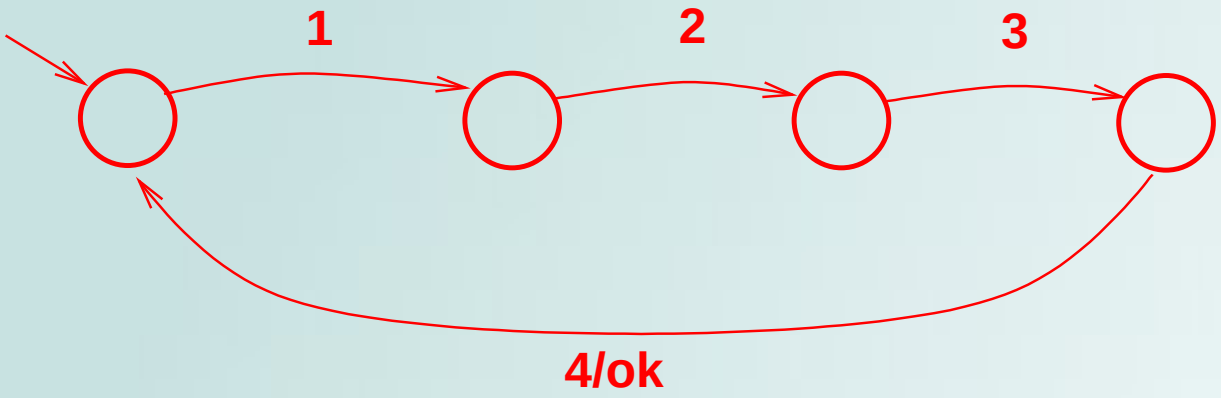
Machine for recognizing `a* b a`



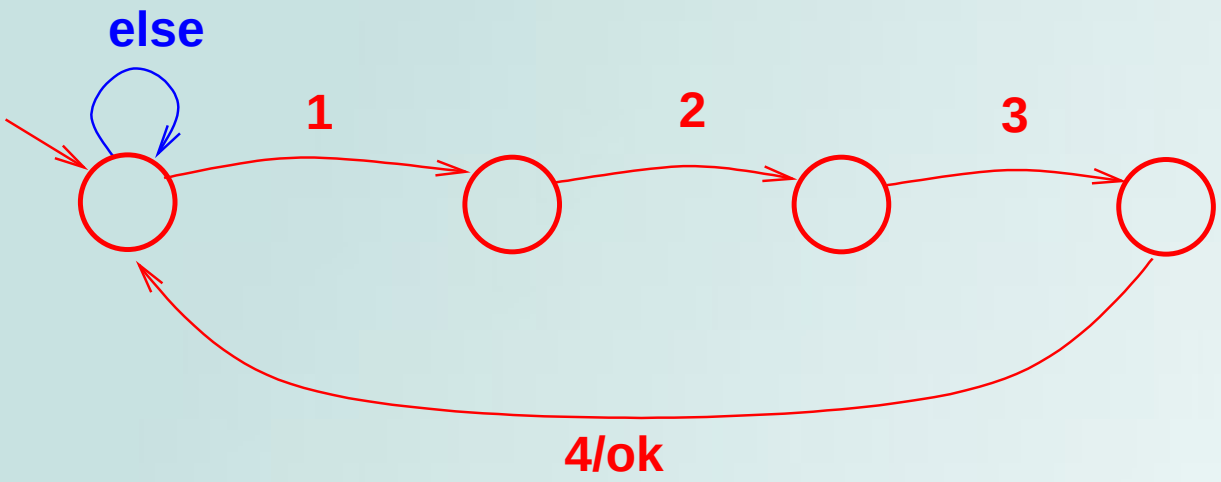
Hypothesis : always not (a and b)
outputs : ok, nok

Decoder for a 4-digit code (output ok)

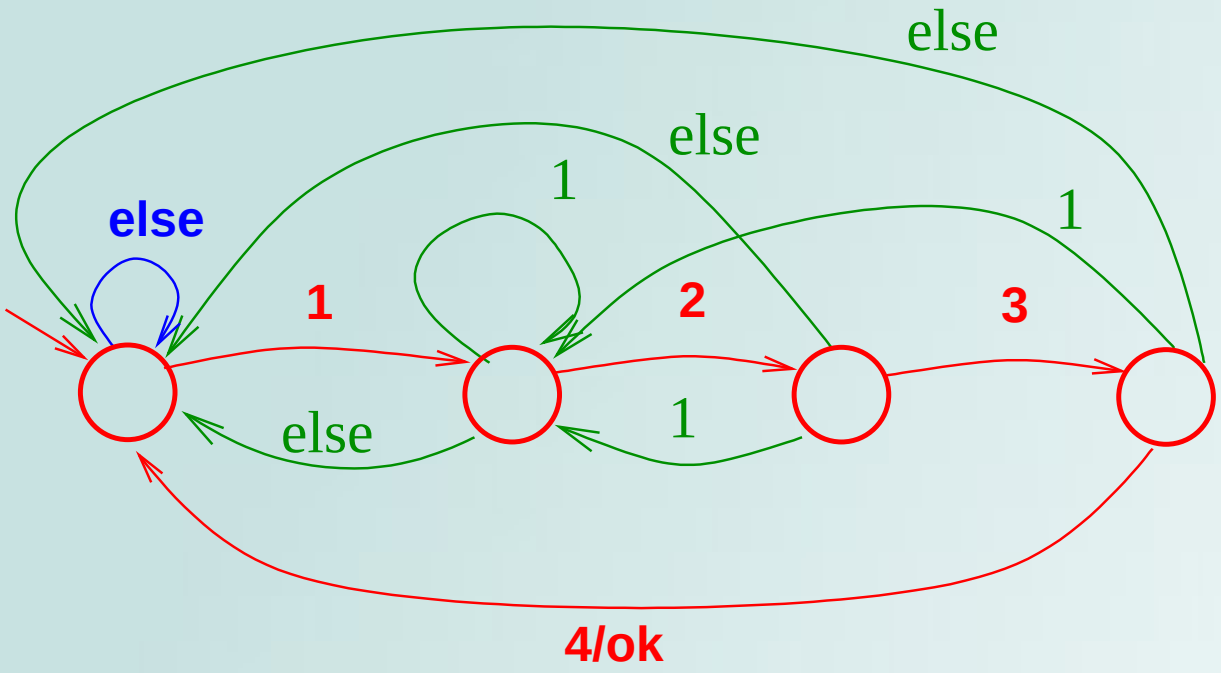
Decoder for a 4-digit code (output ok)



Decoder for a 4-digit code (output ok)



Decoder for a 4-digit code (output ok)



Discussion: if the code is 1212 ?

Equivalences

Two Mealy machines are equivalent if they produce the same sequence of outputs when fed with the same sequence of inputs.

(adapted from the trace-equivalence of regular language recognizers).

Exercice

Find two (small) equivalent Boolean Mealy Machines.

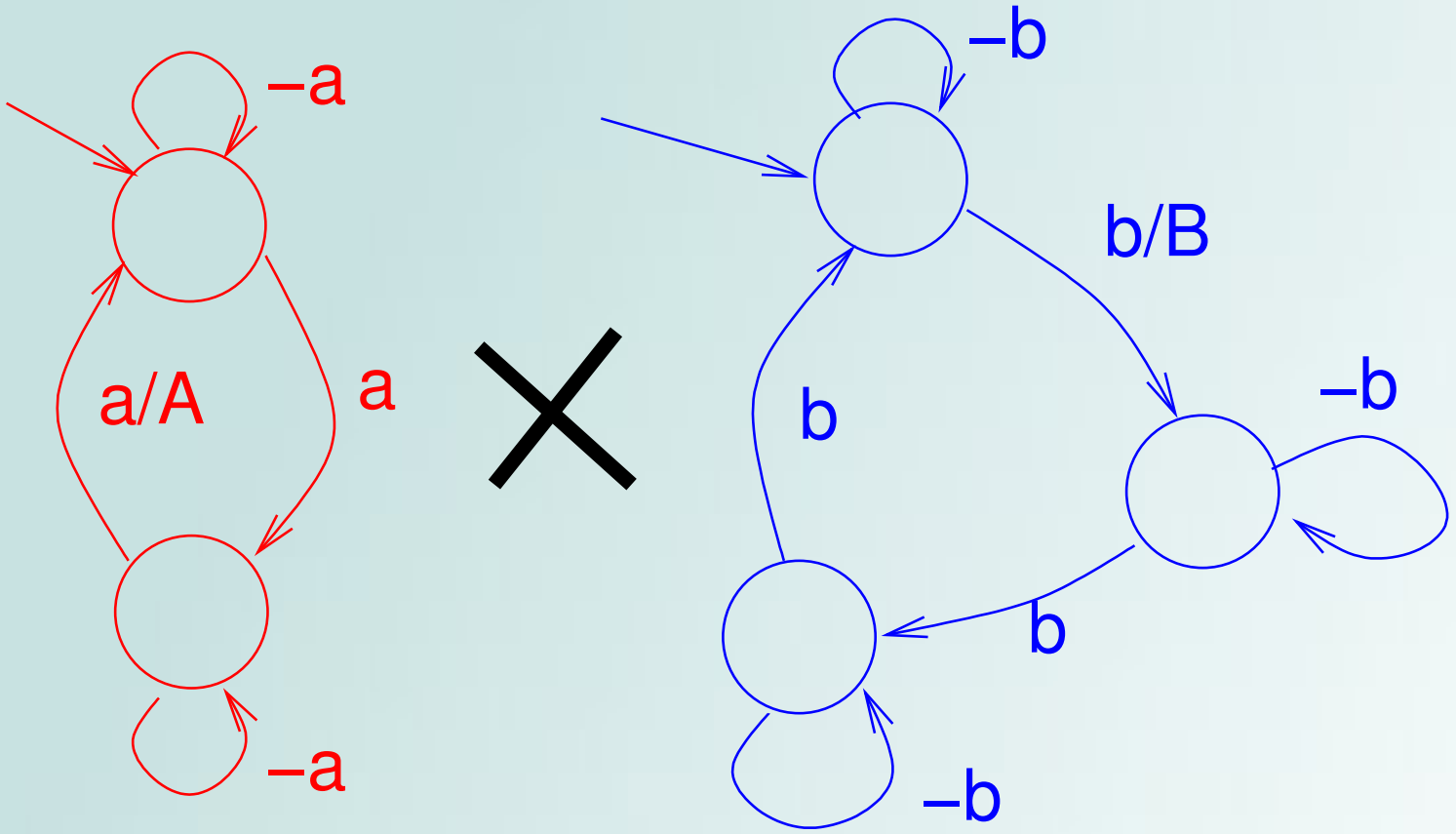
Synchronous Product (without communication)

$M_i = (Q_i, Q_{0_i}, I_i, O_i, T_i)$, for $i = 1, 2$.

$M_1 \times M_2 = (Q_1 \times Q_2, Q_{0_1} \times Q_{0_2}, I_1 \cup I_2, O_1 \cup O_2, T_{\text{prod}})$

$$\frac{(q_1, m_1/o_1, q_1') \in T_1, \quad (q_2, m_2/o_2, q_2') \in T_2}{((q_1, q_2), m_1 \wedge m_2/o_1 \cup o_2, (q_1', q_2')) \in T_{\text{prod}}}$$

Example (-b denotes the negation of b)

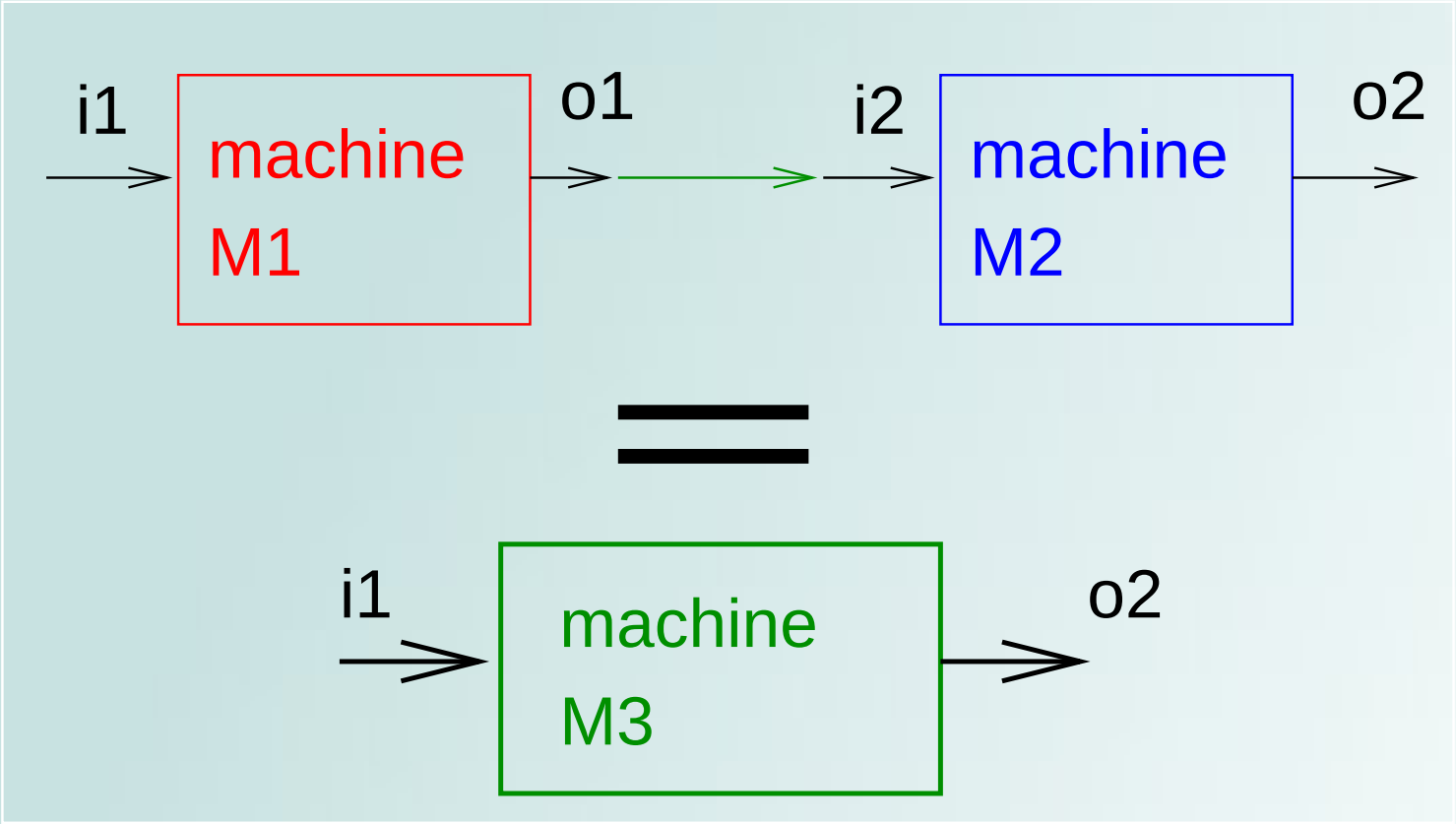


Exercice

Prove that:

- the synchronous product preserves both determinism and reactivity
- the synchronous product is commutative
- the synchronous product is associative
- the synchronous product is idempotent (i.e. : $P \times P = P$)

Communication without loops: idea



The output of a machine can be connected to the input of another one. (As in synchronous circuits).

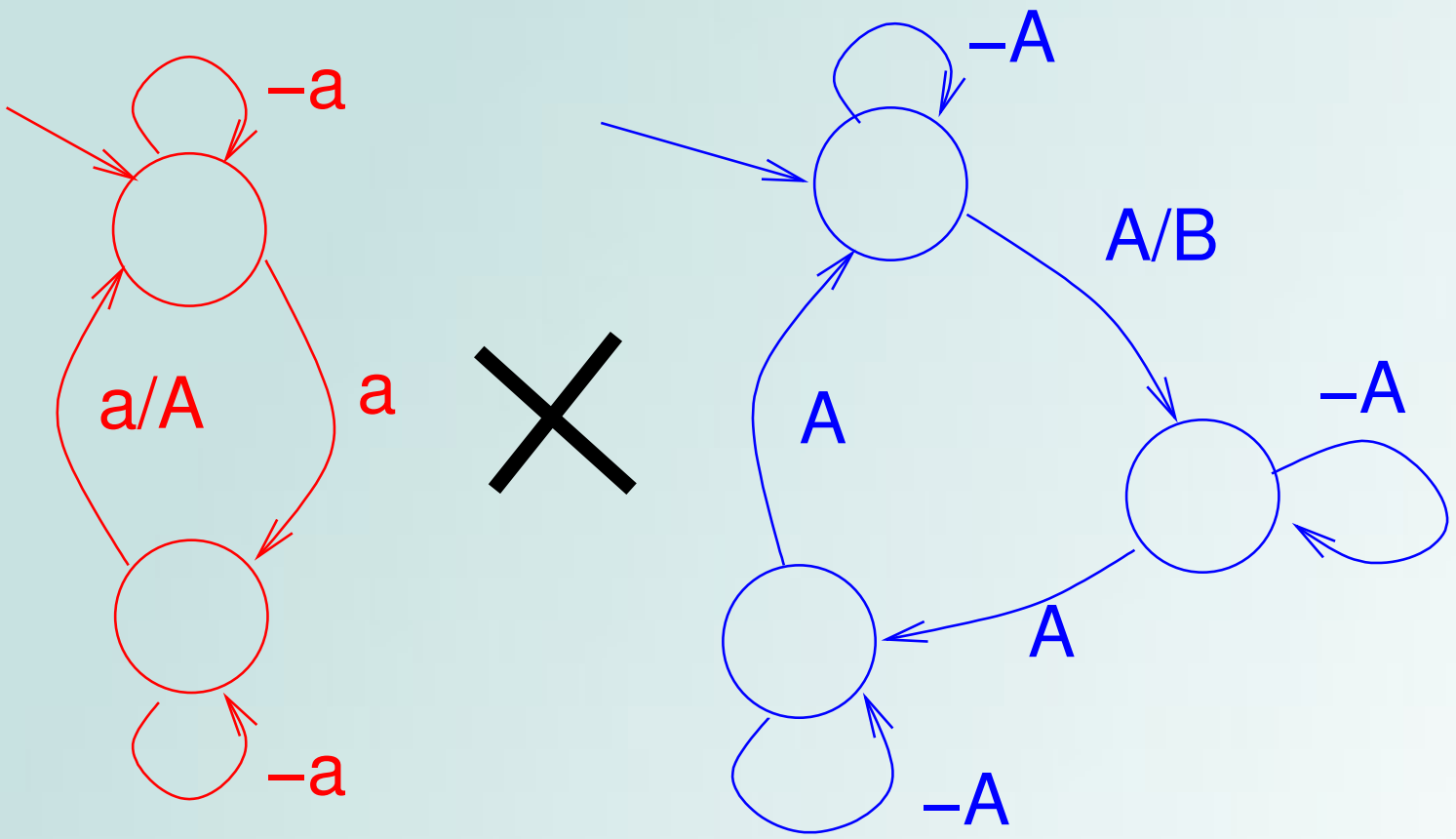
Communication without loops
Computing M3

The machine M3 is defined as: $M_3 = (M_1 \times M_2)\backslash x$
where x are the wires that connect the two machines.

$\backslash x$ is the encapsulation operation

Communication without loops

Example: $x = A$



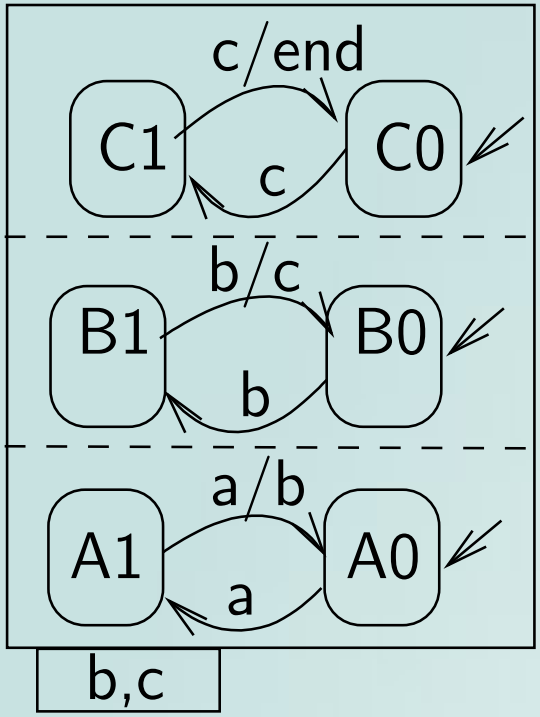
Communication without loops
Encapsulation: $(M_1 \times M_2)\backslash A$

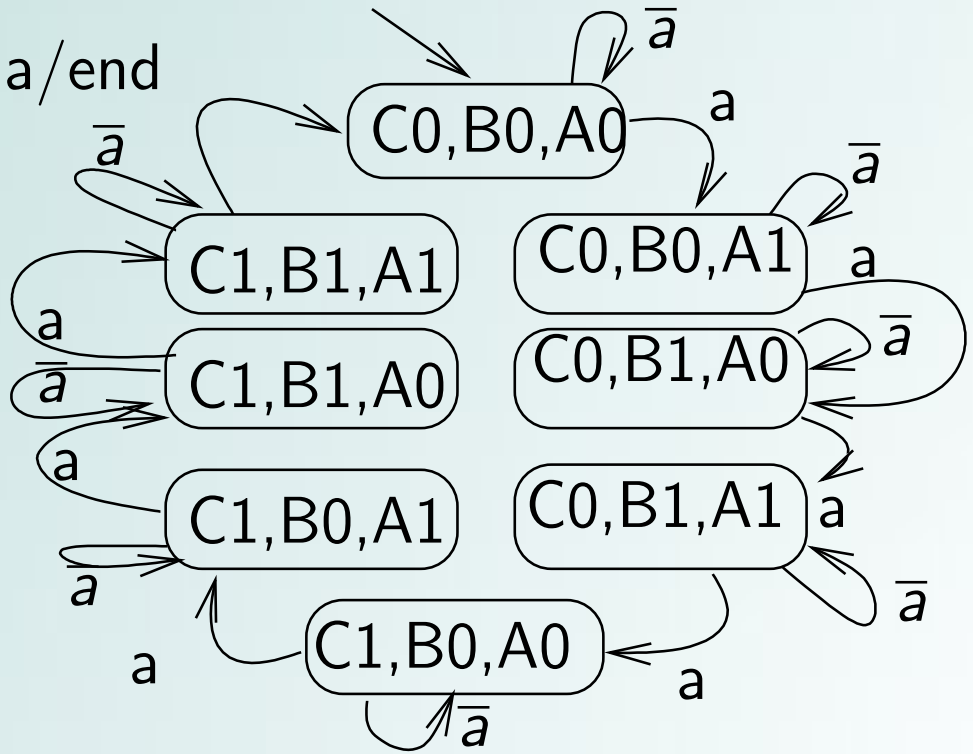
Consider each transition of the product:

- A label of the form `...-A/A...` is inconsistent:
  Machine M2 reacts as if A was absent, but machine M1 sets A to true
- A label of the form `A.../` is inconsistent too:
  Machine M2 reacts as if A was present, but machine M1 does not set A to true (and nobody else can do that, hence it is false)

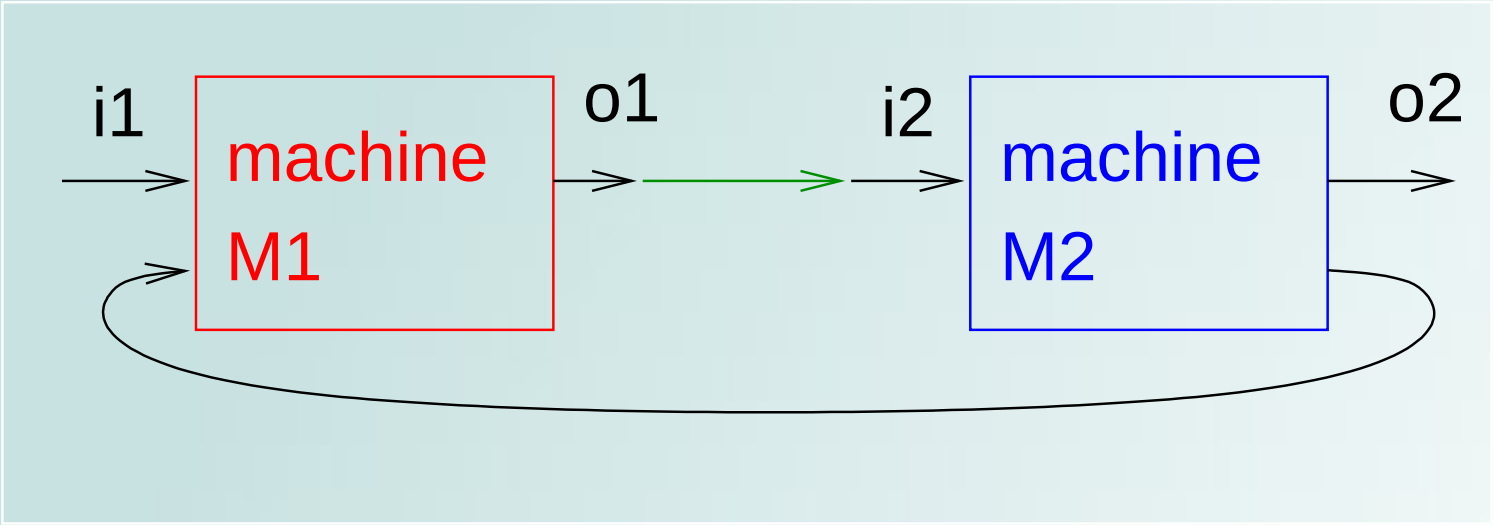
Inconsistent transitions are removed, and `A` is hidden in the labels of the remaining transitions.

Example: the modulo-8 counter





Loops



Two Problems

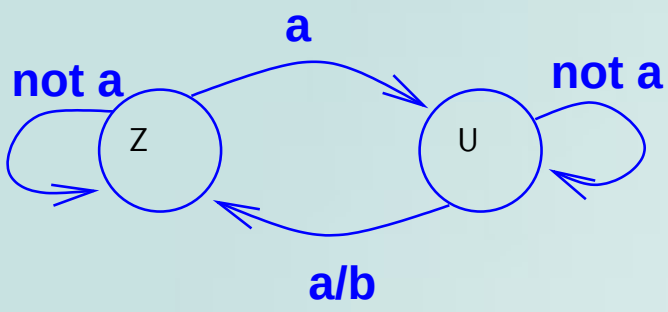
- Producing non-determinism from two deterministic machines: $i.a/b||i.b/a$
- Producing non-reactivity from two reactive machines: $i.a/b||i.\overline{b}/a$

A mini-language

- basic objects: deterministic and reactive Boolean Mealy machines
- Operators: synchronous product + encapsulation
- Incorrect programs = cyclic circuits
  the programs for which one of the encapsulation operations produces non-determinism or non-reactivity.
- Semantics by flattening a program into a single automaton applying the product and the encapsulation recursively.

Example program: see exam 05

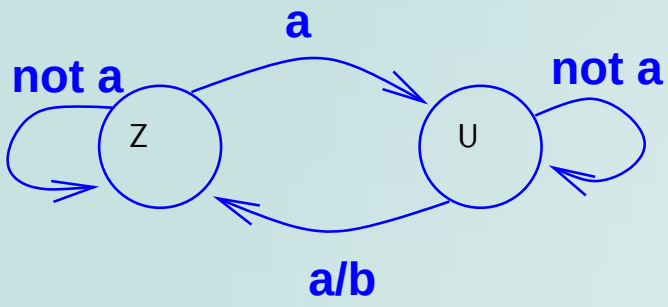
Encoding into Boolean Equations



One Bool variable per state: `z, u`
One Bool variable per signal: `a, b`.

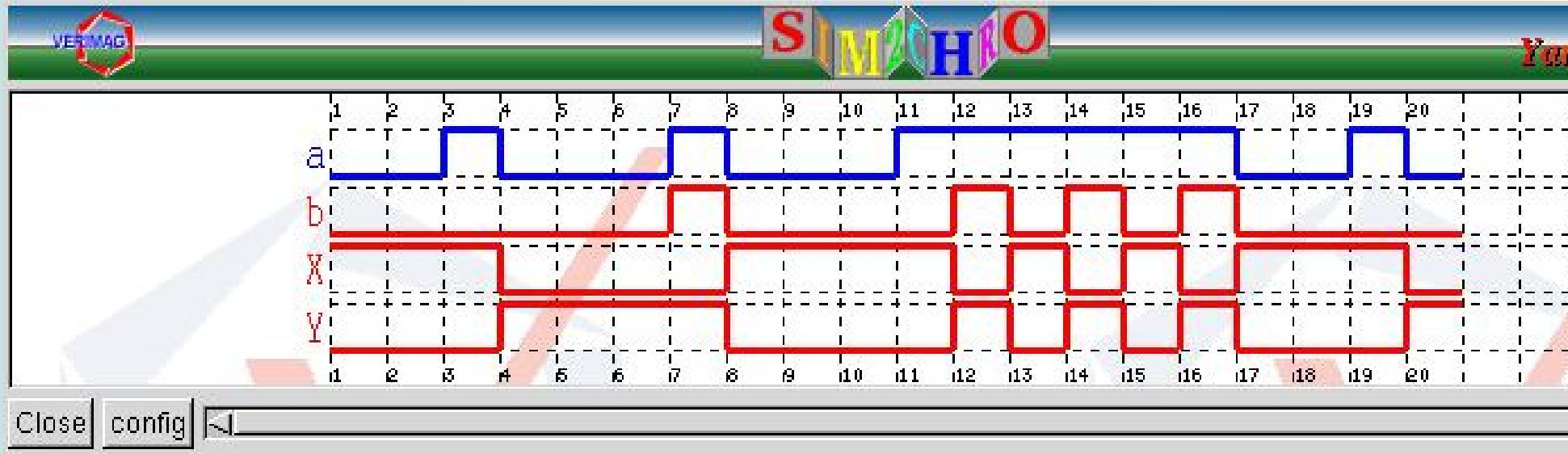
```
z(0) = true ;
u(0) = false ;
z(n) = (z(n-1) and not a(n-1)) or
           (u(n-1) and a(n))
u(n) = (u(n-1)  and not a(n-1)) or
           (z(n-1) and a(n-1))
b(n)  = u(n) and a(n)
```

Encoding into Boolean Equations: Lustre




```
-- Lustre program
node Bit
  (a : bool) returns
  (b : bool ; Z, U : bool ) ;
let
 Z = true ->
       pre (Z and not a) or
       pre (U and a) ;

 U = false ->
       pre (U and not a) or
       pre (Z and a) ;

 b = U and a ;
tel.
```

Encoding into Boolean Equations: example behavior



Mealy machine for an imperative program

Any imperative sequential program can be translated into an interpreted automata.

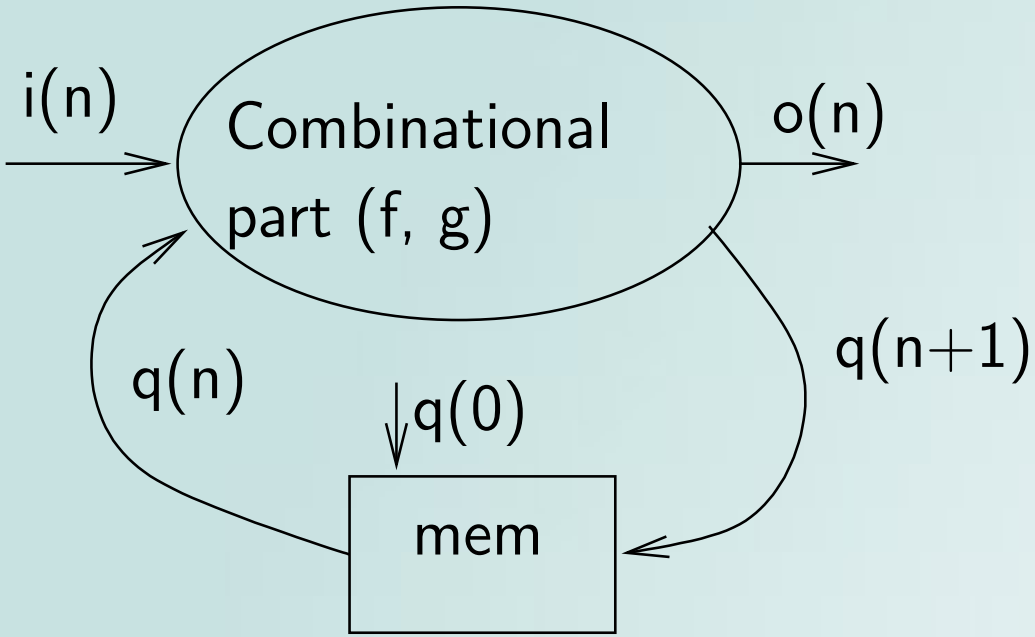
States are control-points in the code;
The transitions are labeled by: a condition on variables, a set of assignments to variables.

Mealy machine for a synchronous circuit

Any synchronous circuit can be translated into a Mealy machine.

The states are the configurations of all the Boolean memories (flip-flops)
Transition are labeled by a condition on the inputs, the set of true outputs.

Mealy machine for a synchronous circuit



$o_n = f(i_n, q_n)$
$q_{n+1} = g(i_n, q_n)$

For k flip-flops, there are 2^k potential states.

Start with q_0, compute all possible successor states (examining all combinations of the inputs) and the associated outputs... and so on until this produces no new state.

7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

Simple Moore Machines

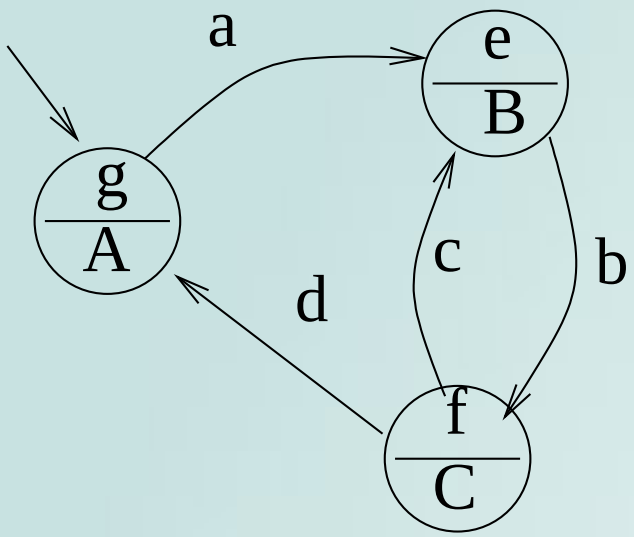
(cf. 1A course)

$M = (Q, Q_0 \subseteq Q, T \subseteq Q \times I \times Q, F : Q \longrightarrow O)$ where

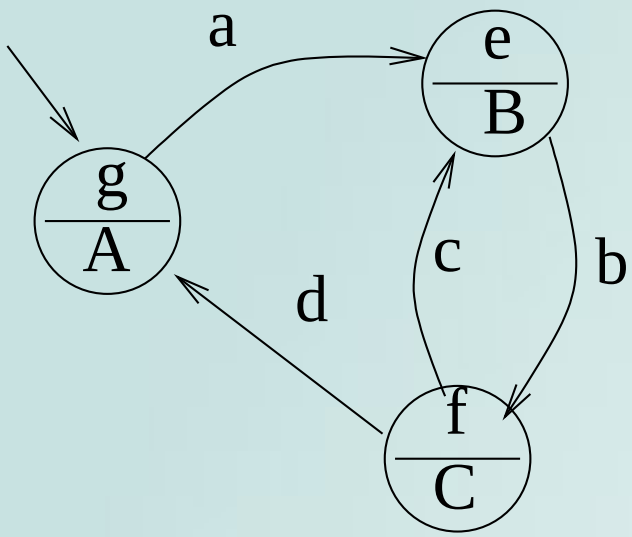
- I (resp. O) is the set of inputs (resp. outputs)
- Q_0 is the set of initial states
- $F : Q \longrightarrow O$ associates an output with each state

Drawings: $q(\text{output}) \xrightarrow{\text{input}} q'(\text{output})$

Trace Semantics

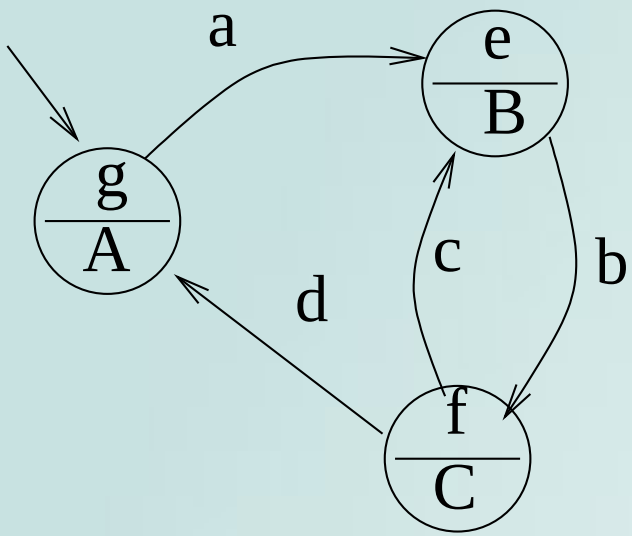


Trace Semantics



state	A	B	C	B	C	A
input	a	b	c	b	d	
output	g	e	f	e	f	g

Trace Semantics



state	A	B	C	B	C	A
input	a	b	c	b	d	
output	g	e	f	e	f	g

Imperative Program

```
State := A  -- take one of the initial states
while (true) loop
    get (I) ; O := null ;
    case State :
        when A : O := g ;
                 if I = a then State:=B ;
        when B : O := e ;
                 if I = b then State:=C;
        when C : O := f ;
                 if I = c then State:=B;
                 elsif I = d then State:=A ;
    end case
    put (O)
end loop
```

Boolean Moore Machines

Same definition, but with $I = 2^{\mathsf{BI}}$ and $O = 2^{\mathsf{BO}}$.

BI : input signals, any configuration is possible.
BO : output signals, any configuration is possible.

Determinism and reactivity

Determinism:

$$\mid Q_0 \mid = 1 \quad \wedge$$

$$\forall q \in Q, \left\{ \begin{array}{l} \exists (q, m_1, q_1) \in T \wedge \\ \exists (q, m_2, q_2) \in T \end{array} \right\} \quad \Longrightarrow \; m_1 \wedge m_2 = \text{false}$$

Reactivity:

$$\forall q \in Q, \quad \bigvee_{(q, m_i, q_i) \in T} m_i \quad = \text{true}$$

Equivalences

Two Moore machines are equivalent if they produce the same sequence of outputs when fed with the same sequence of inputs.

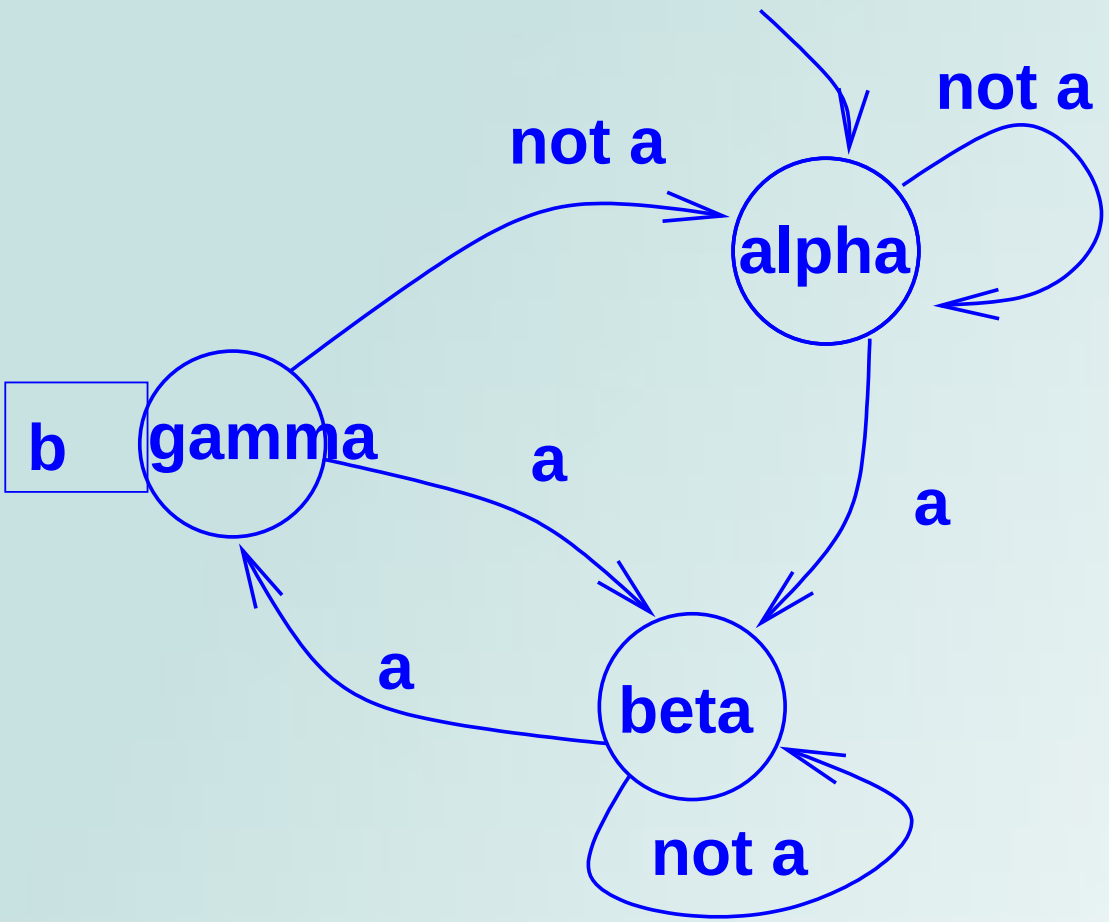
(adapted from the trace-equivalence of regular language recognizers).

Moore machines, exercice

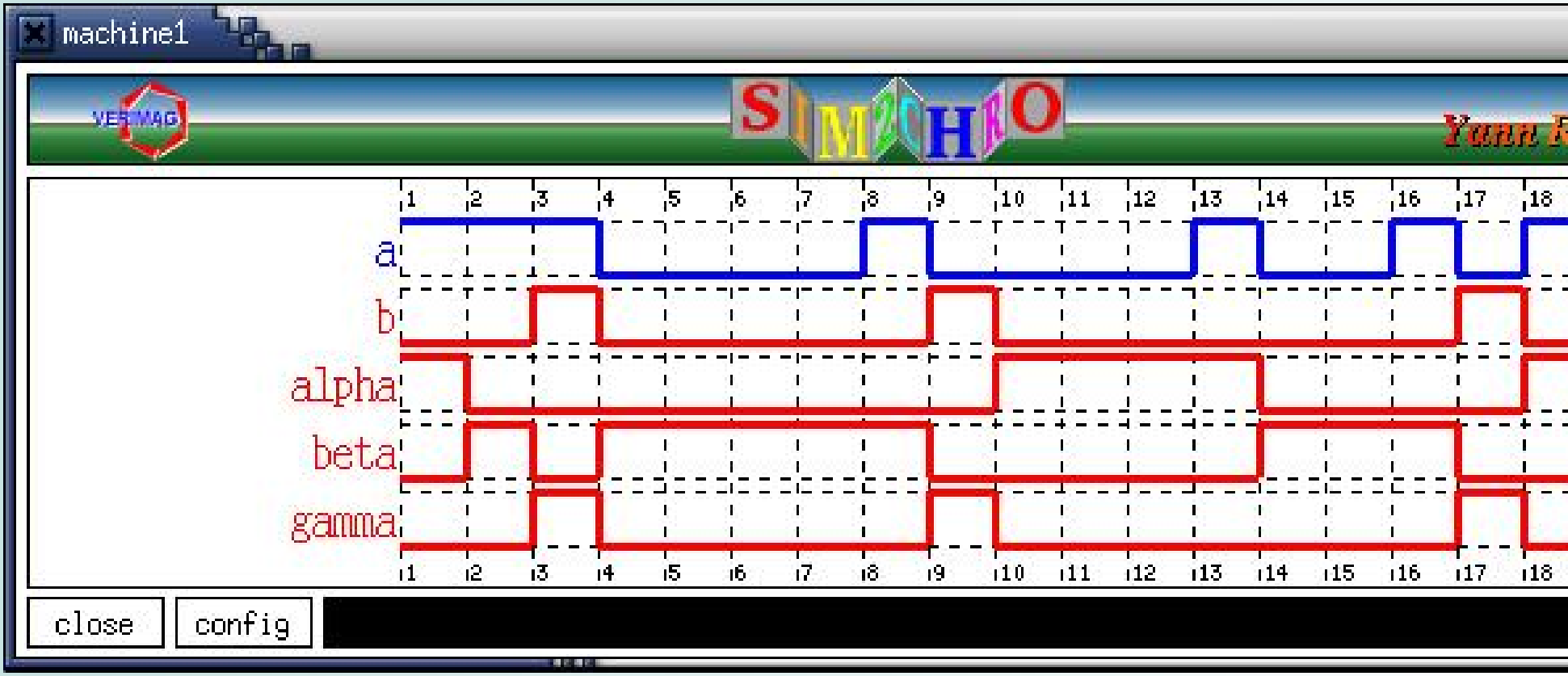
Write a Moore machine that emits one 'b' every two 'a's (during one cycle).

Give some trace examples.

Moore machine : one 'b' every two 'a's



Traces


machine1
VERIMAG
SIMCHO
a
b
alpha
beta
gamma
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
close
config

Synchronous Product (without communication)

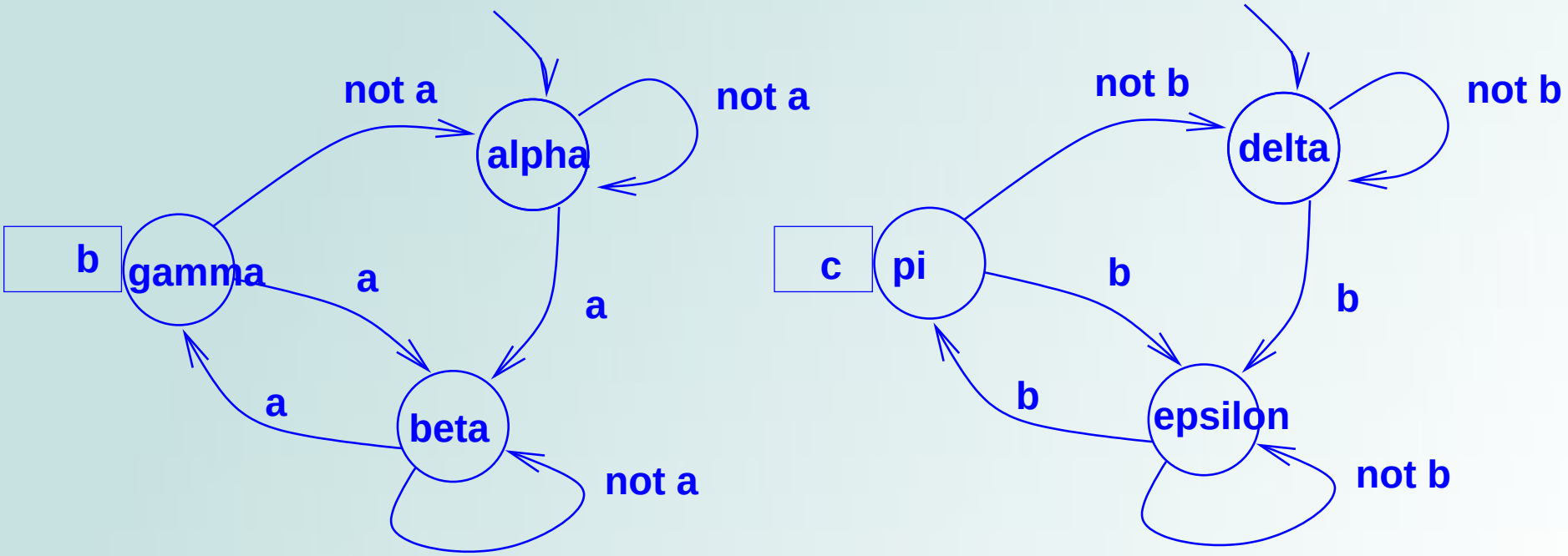
$M_i = (Q_i, Q_{0_i}, I_i, O_i, T_i, F_i)$, for $i = 1, 2$.

$M_1 \times M_2 = (Q_1 \times Q_2, Q_{0_1} \times Q_{0_2}, I_1 \cup I_2, O_1 \cup O_2, T_{\text{prod}}, F_{\text{prod}})$

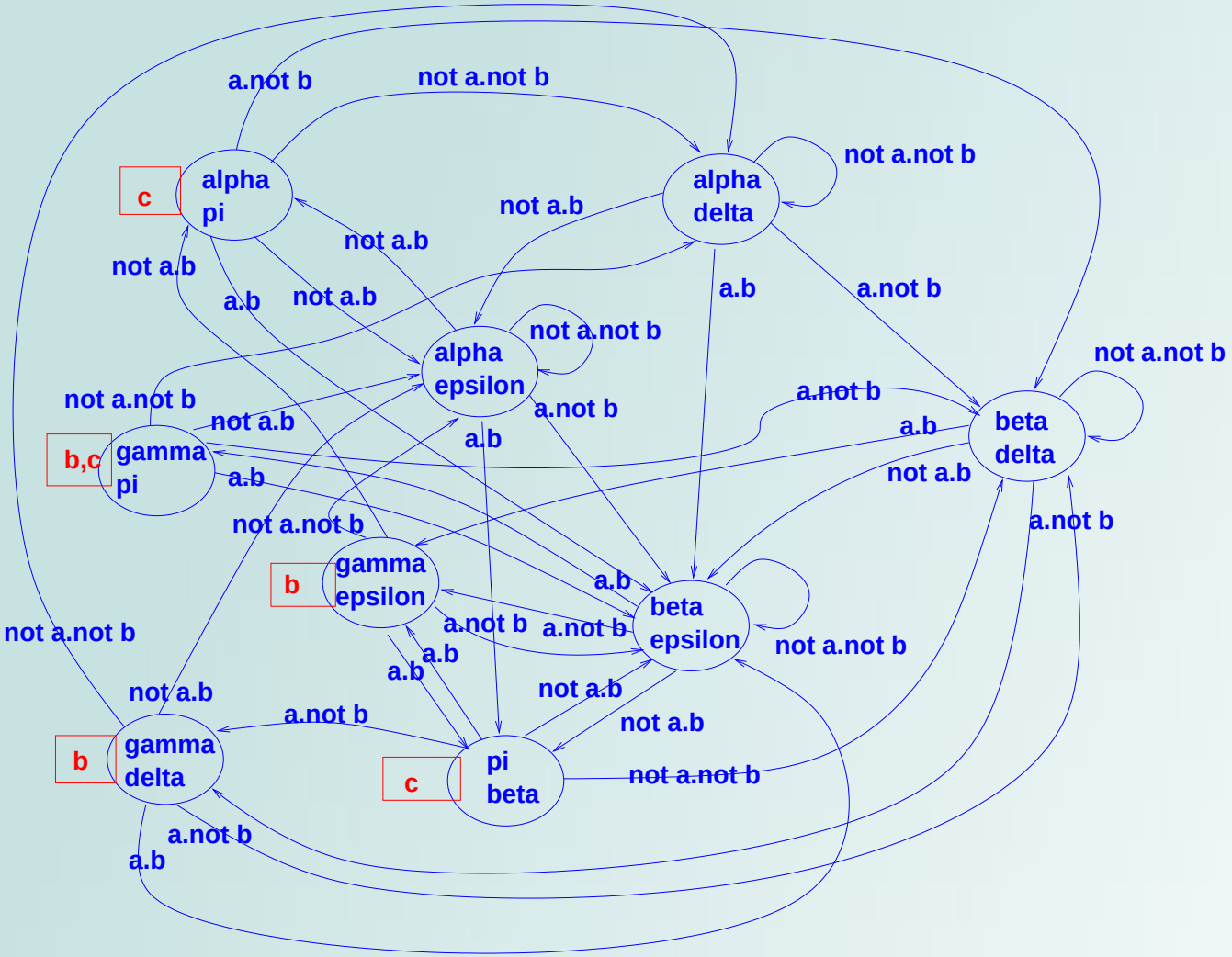
$$\frac{(q_1, m_1, q_1') \in T_1, \quad (q_2, m_2, q_2') \in T_2}{((q_1, q_2), m_1 \wedge m_2, (q_1', q_2')) \in T_{\text{prod}}}$$

and $F_{\text{prod}}((q_1, q_2)) = F_1(q_1) \cup F_2(q_2)$.

Example


not a
not a
alpha
b
gamma
a
a
a
beta
not a
not b
not b
delta
c
pi
b
b
b
epsilon
not b

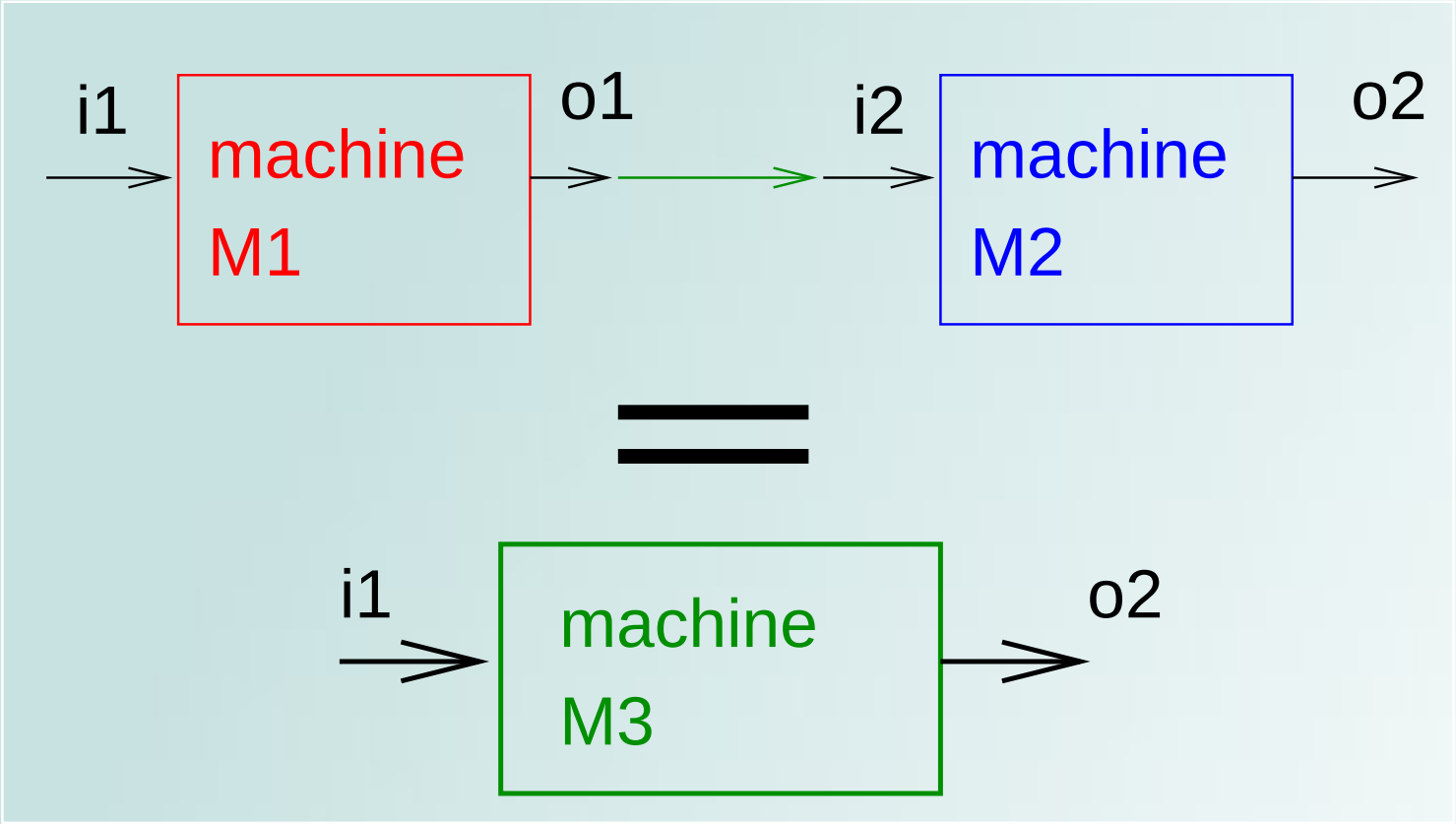
Example, result



Properties

- the synchronous product preserves both determinism and reactivity
- the synchronous product is commutative
- the synchronous product is associative
- the synchronous product is idempotent

Communication: idea

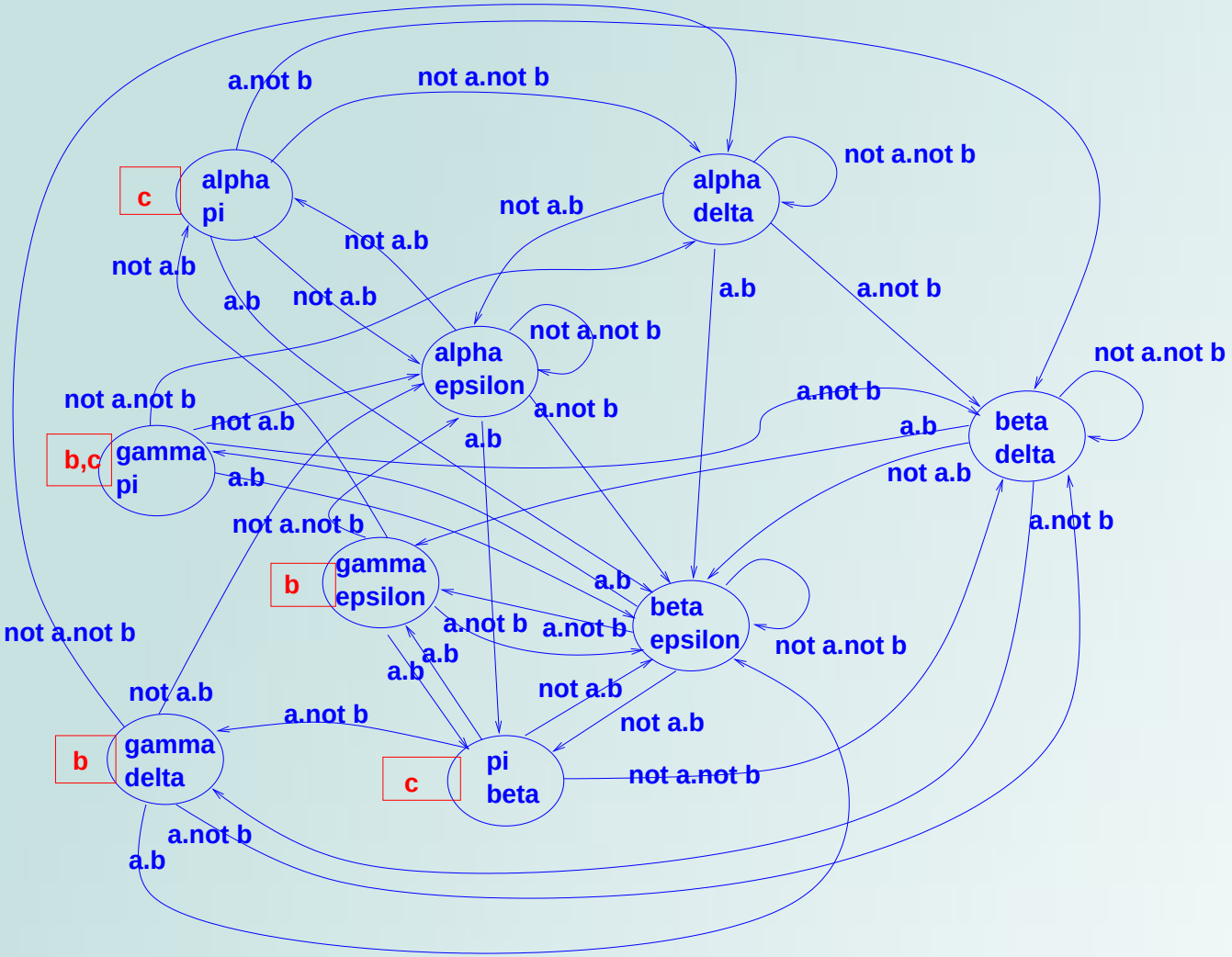


The output of a machine can be connected to the input of another one. (As in synchronous circuits).

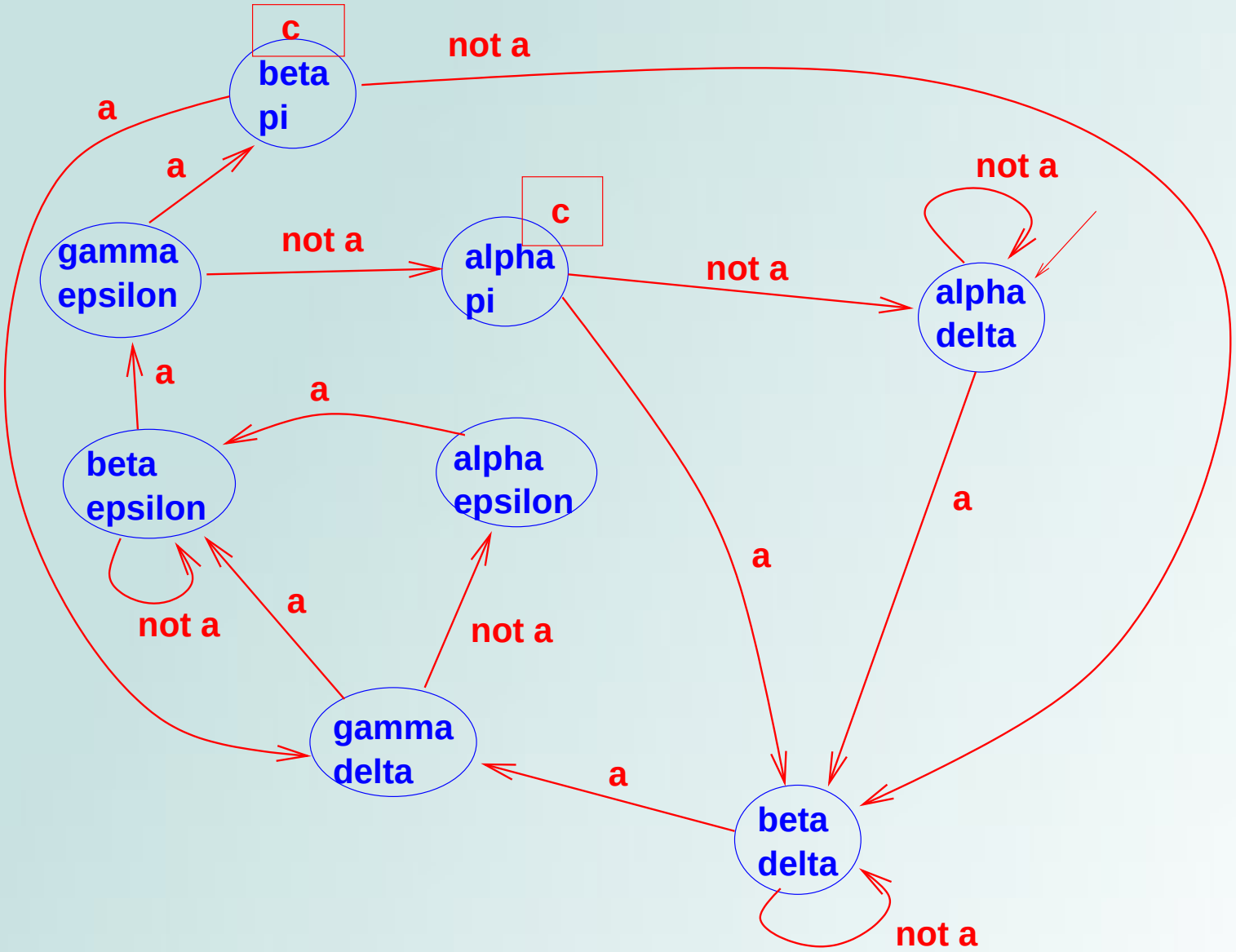
Communication: encapsulation

For all the encapsulated signals, the transitions that need a signal to be present can be taken only from a state that emits it.

Encapsulation : example (before...)



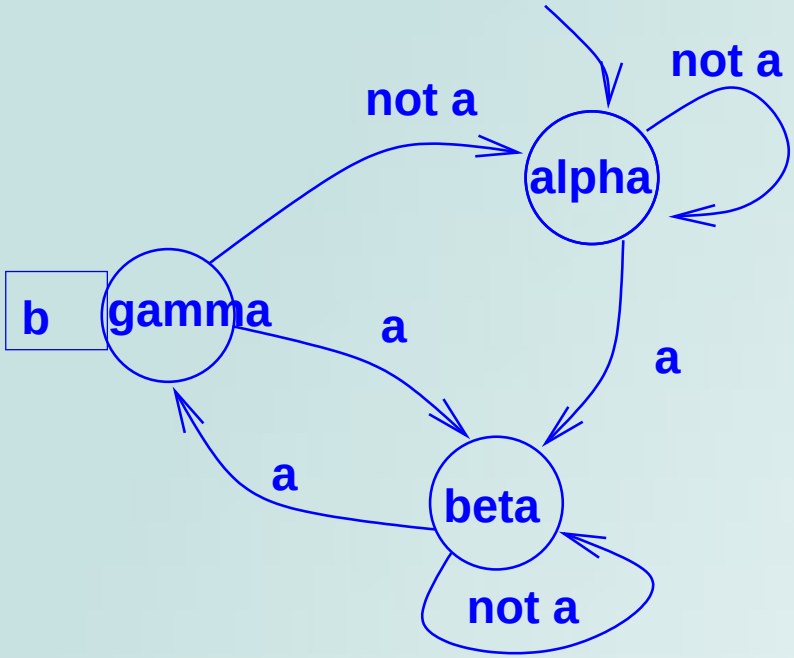
Encapsulation : example (...after)



Encapsulation : Loops problems

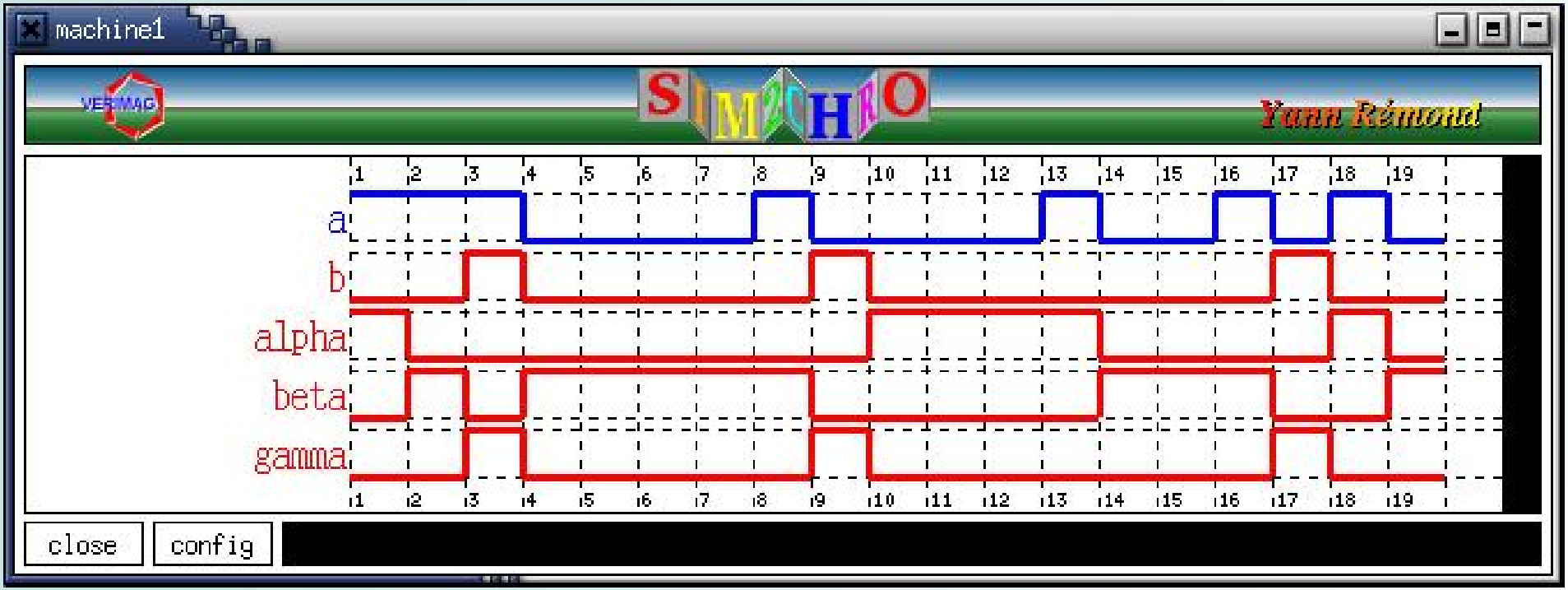
There are no loops problems in the encapsulation of the product of two Moore machines.
(no instantaneous communication).

Encoding into Boolean Equations: Lustre




```
node machine1 (a : bool)
returns (b : bool) ;
var
   alpha, beta, gamma : bool ;
let
alpha = true ->
        pre (alpha and not a) or
        pre (gamma and not a) ;
beta = false ->
        pre (beta and not a) or
        pre (alpha and a) or
        pre (gamma and a) ;
gamma = false ->
        pre (beta and a) ;
b = gamma ;
tel.
```

Example behavior

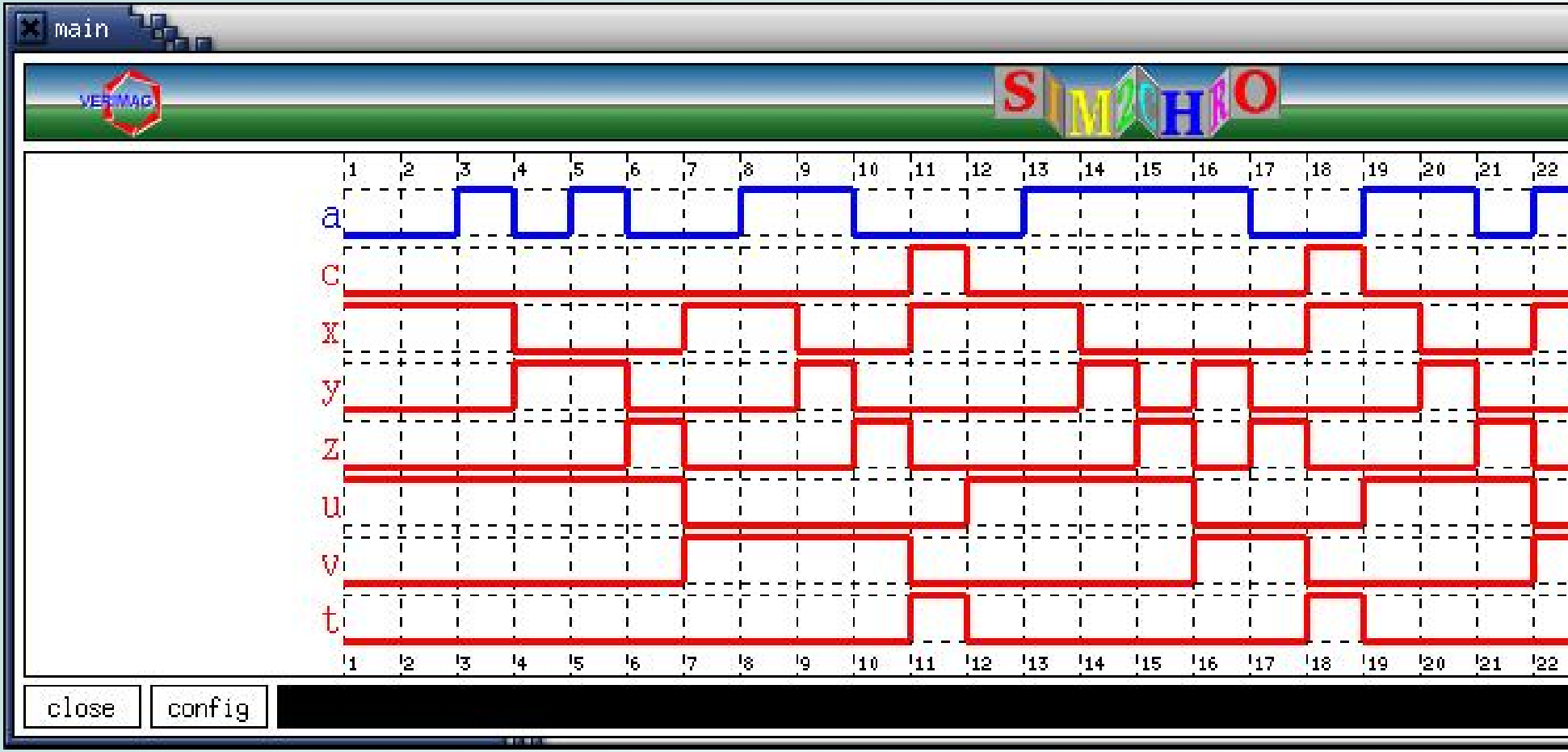


The output 'b' is delayed by one tick, with respect to the second occurrence of the input 'a'.

The Product

```
node main (a : bool)
returns   (c : bool) ;
var
   b : bool ;
let
    b = machine1 (a) ;
    c = machine1 (b) ;
tel.
```

Example behavior



The output 'c' is delayed by two ticks, with respect to the fourth occurrence of the input 'a'.

7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

General i/o behavior

- General case : $\forall n,\ o_n = f(i_0, i_1, ..., i_n)$

  Remarks :

  $$\begin{cases} o_n \text{ may not depend on the } \textit{future } i_{n+...} \ (\textit{causality}) \\ o_n \text{ may depend on } i_n \ (\textit{synchrony hypothesis}) \\ \text{The history of inputs is not bounded} \end{cases}$$

General i/o behavior

- General case : $\forall n,\ o_n = f(i_0, i_1, ..., i_n)$
  Remarks :
  $$\begin{cases} o_n \text{ may not depend on the } \textit{future } i_{n+...} \ (\textit{causality}) \\ o_n \text{ may depend on } i_n \ (\textit{synchrony hypothesis}) \\ \text{The history of inputs is not bounded} \end{cases}$$
- Combinational circuits : $\forall n,\ o_n = f(i_n)$
  No memory

General i/o behavior

- General case : $\forall n,\ o_n = f(i_0, i_1, ..., i_n)$
  Remarks :
  $$\begin{cases} o_n \text{ may not depend on the } \textit{future } i_{n+...} \ (\textit{causality}) \\ o_n \text{ may depend on } i_n \ (\textit{synchrony hypothesis}) \\ \text{The history of inputs is not bounded} \end{cases}$$
- Combinational circuits : $\forall n,\ o_n = f(i_n)$
  No memory
- Bounded-Memory Systems (Sequential Circuits) :
  $\forall n,\ o_n = f(Abs(i_0, i_1, ..., i_n))$
  Where *Abs* is an *abstraction* function,
  such that $\exists B \mid Image(Abs) \mid < B$

Bounded-Memory Systems
Program Scheme

Initialization of the *memory* M
while true

Invariant :	$M = Abs(i_0, i_1, ..., i_{n-1})$
Get input	i_n
Compute output	$o_n = f(M, i_n)$
Update memory	$M = g(M, i_n)$
Property here :	$M = Abs(i_0, i_1, ..., i_n)$
Emit output	o_n

The abstraction function must be such that:
$\exists g,\ Abs(i_0, i_1, ..., i_n) = g(Abs(i_0, i_1, ..., i_{n-1}), i_n)$

7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

The Picture

A mini-language for composing Mealy or Moore machines ...

The Picture

A mini-language for composing Mealy or Moore machines ...
can be compiled into a single flat machine ...

The Picture

A mini-language for composing Mealy or Moore machines ...
can be compiled into a single flat machine ...
that can then be implemented in both hardware and software.

A good basis for a language dedicated to embedded systems:

- Easy expression of synchronous parallelism
- Easy compilation into software, easy synthesis of hardware

7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

The Synchronous Dataflow Paradigm

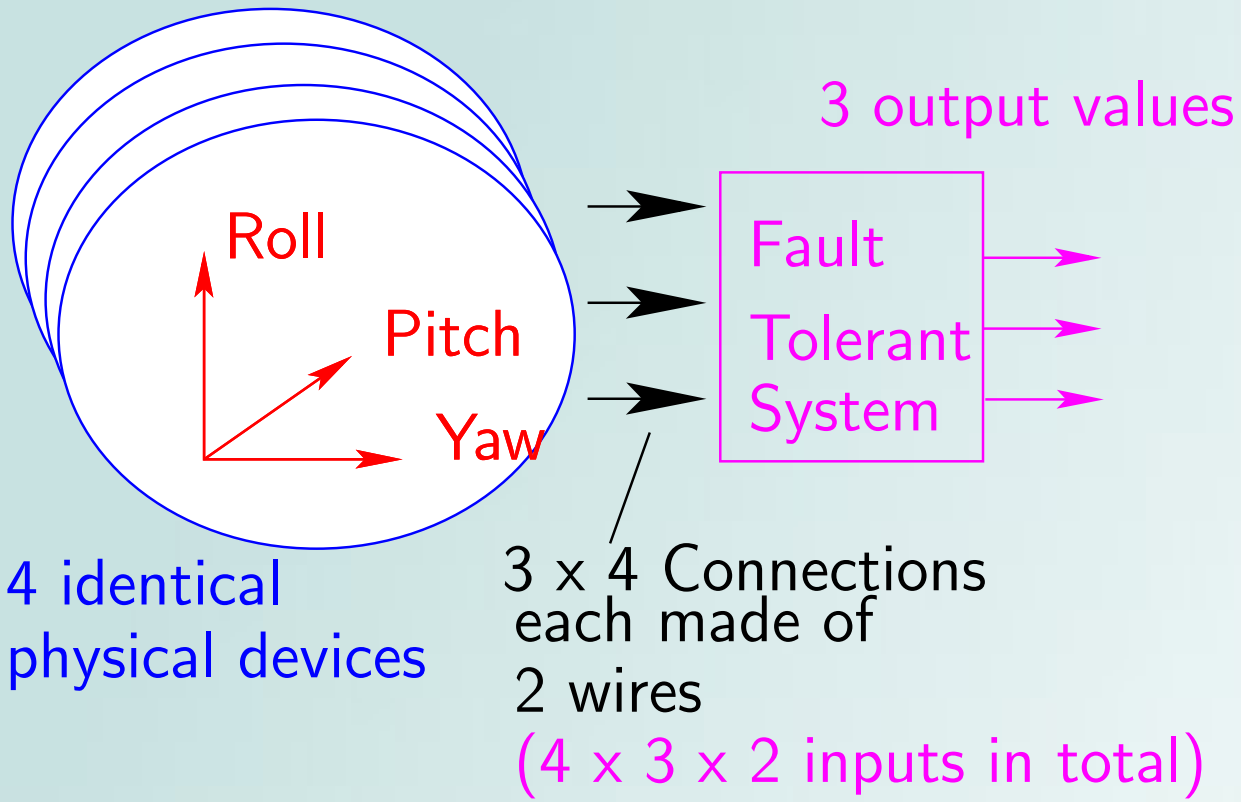
Systems are made of boxes and oriented wires, considered to behave in parallel.
There is a notion of global discrete clock on which all the boxes evolve: they take one input on their input wires, and produce one output on their output wires.

Very similar to the diagrammatic view of synchronous circuits.

Lustre

- Lustre (1984, VERIMAG, Grenoble) is a language for describing synchronous dataflow networks.
- It can be compiled into hardware or in software.
- It can be used for formal verification (model-checking, cf. the Turing Award in 2007...)
- It can also be used for expressing specifications (safety properties)

The System


3 output values
Roll
Pitch
Yaw
Fault
Tolerant
System
4 identical
physical devices
3 x 4 Connections
each made of
2 wires
(4 x 3 x 2 inputs in total)

Structure of the program

- 4 identical "channels". Each one receives all the values from one sensor, and communicates with the three other ones to decide whether the values it computes are relevant
- A "main" program that connects the 4 channels together, plus a voter

In the sequel, we look at one channel, written for one axis only (roll).

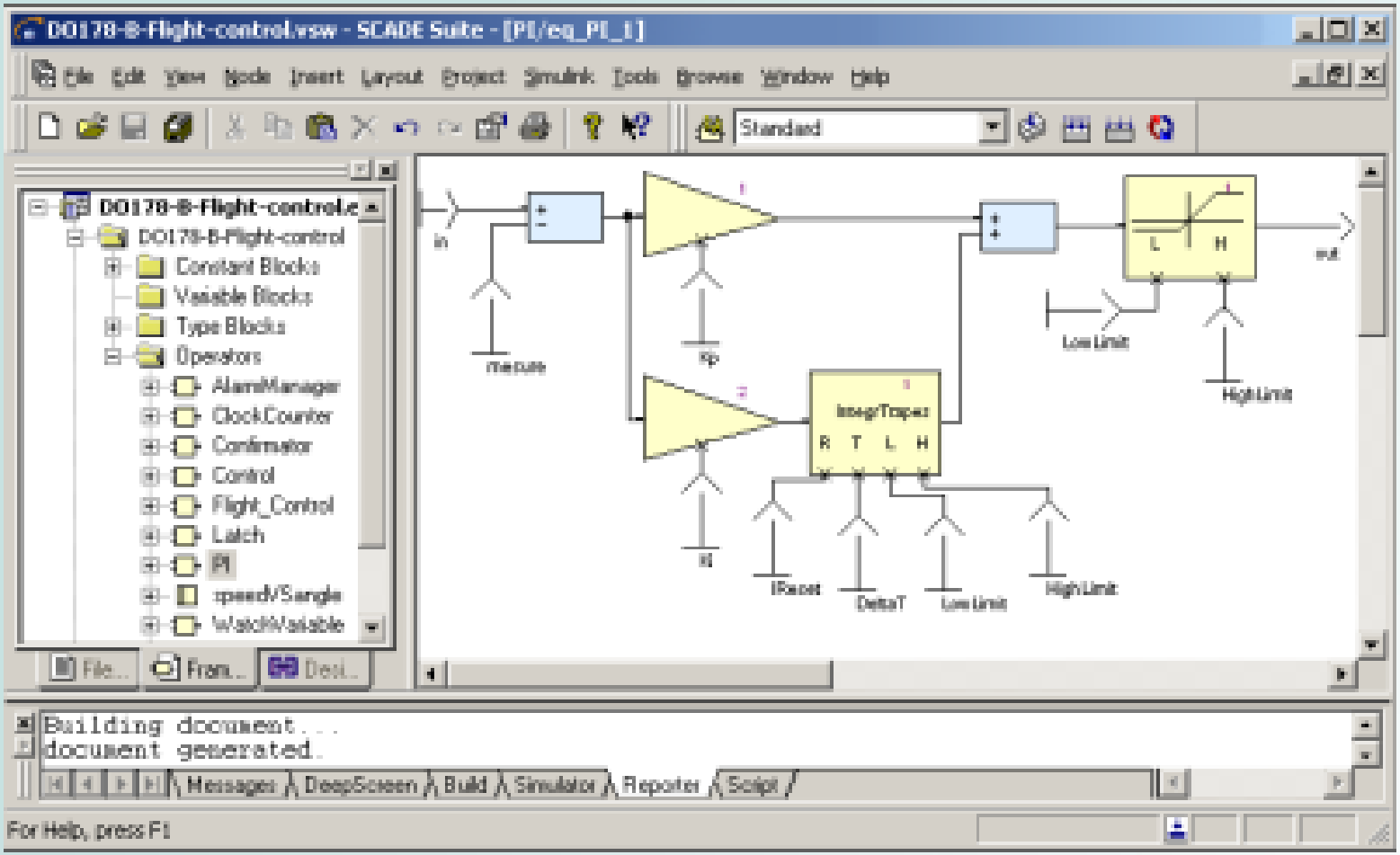
Lustre Example: the Channel (1)

```
node Channel (
    ongroundreset, inairreset : bool ; -- the global inputs
    choffi : bool ; -- channel offline
    xai, xbi : real ;  -- two inputs
    disci : bool ; -- inputs disconnected
    pxother1, pxother2, pxother3 :  real ; -- values computed by th
    pfother1, pfother2, pfother3 : bool ; -- failure statuses of th
    allowedi : bool ; -- allowed to become reset inhibited
)
returns (
    xi : real ; -- the value computed by this channel
    fi : bool ; -- the failure status of this channel
    aski : bool ; -- a request to become reset inhibited
    debug_localfailure : bool ; -- for observation purposes
    debug_cross_failure : bool ; -- for observation purposes
    debug_st : int ; -- for observation purposes
)
```

Lustre Example: the Channel (2)

```
var
    local_failure : bool ;
let

    (xi, local_failure)  = Monitor (xai, xbi, disci) ;

    (fi, aski, debug_cross_failure, debug_st) = FailDetect (
        local_failure, xi, allowedi,
        ongroundreset, inairreset,
        choffi,
        pxother1, pxother2, pxother3,
        pfother1, pfother2, pfother3) ;

    debug_localfailure = local_failure ;
tel
```

SCADE Graphical Syntax



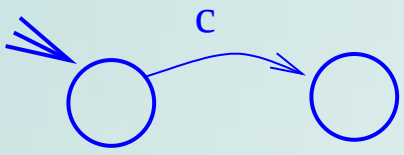
Mouse Double Click

Double-click: if there are two clicks in less than 10 tops (of a clock) it is a double click, otherwise it is a simple click

Exercice: the same in Lustre.

Mouse Double Click

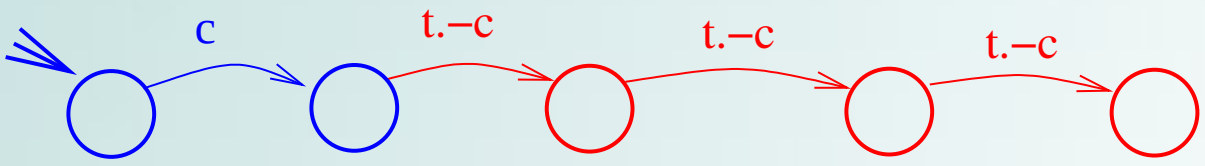
Double-click: if there are two clicks in less than 10 tops (of a clock) it is a double click, otherwise it is a simple click



Exercice: the same in Lustre.

Mouse Double Click

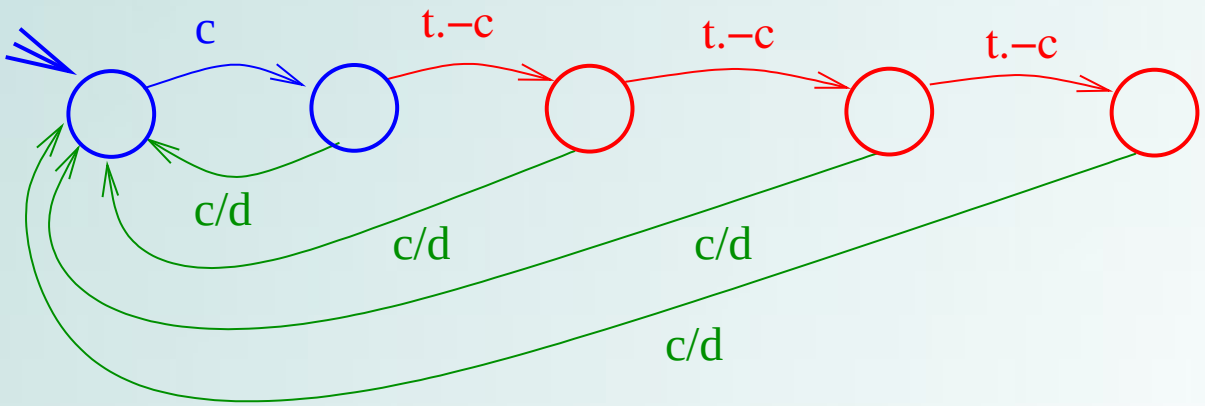
Double-click: if there are two clicks in less than 10 tops (of a clock) it is a double click, otherwise it is a simple click



Exercice: the same in Lustre.

Mouse Double Click

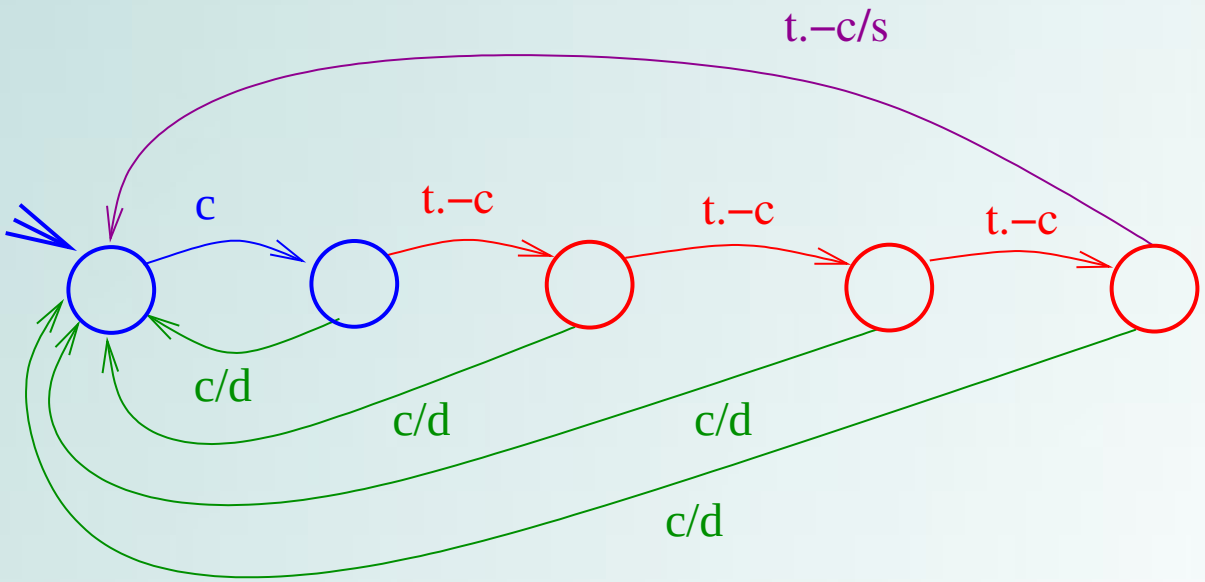
Double-click: if there are two clicks in less than 10 tops (of a clock) it is a double click, otherwise it is a simple click



Exercice: the same in Lustre.

Mouse Double Click

Double-click: if there are two clicks in less than 10 tops (of a clock) it is a double click, otherwise it is a simple click



Exercice: the same in Lustre.

Sub-question: a two-state automaton

```
node twostates (a, b : bool) returns (un, deux : bool) ;
let
        un = true -> pre ((un and not a) or (deux and b)) ;
        deux = false -> pre ((deux and not b) or (un and a))
tel.
```

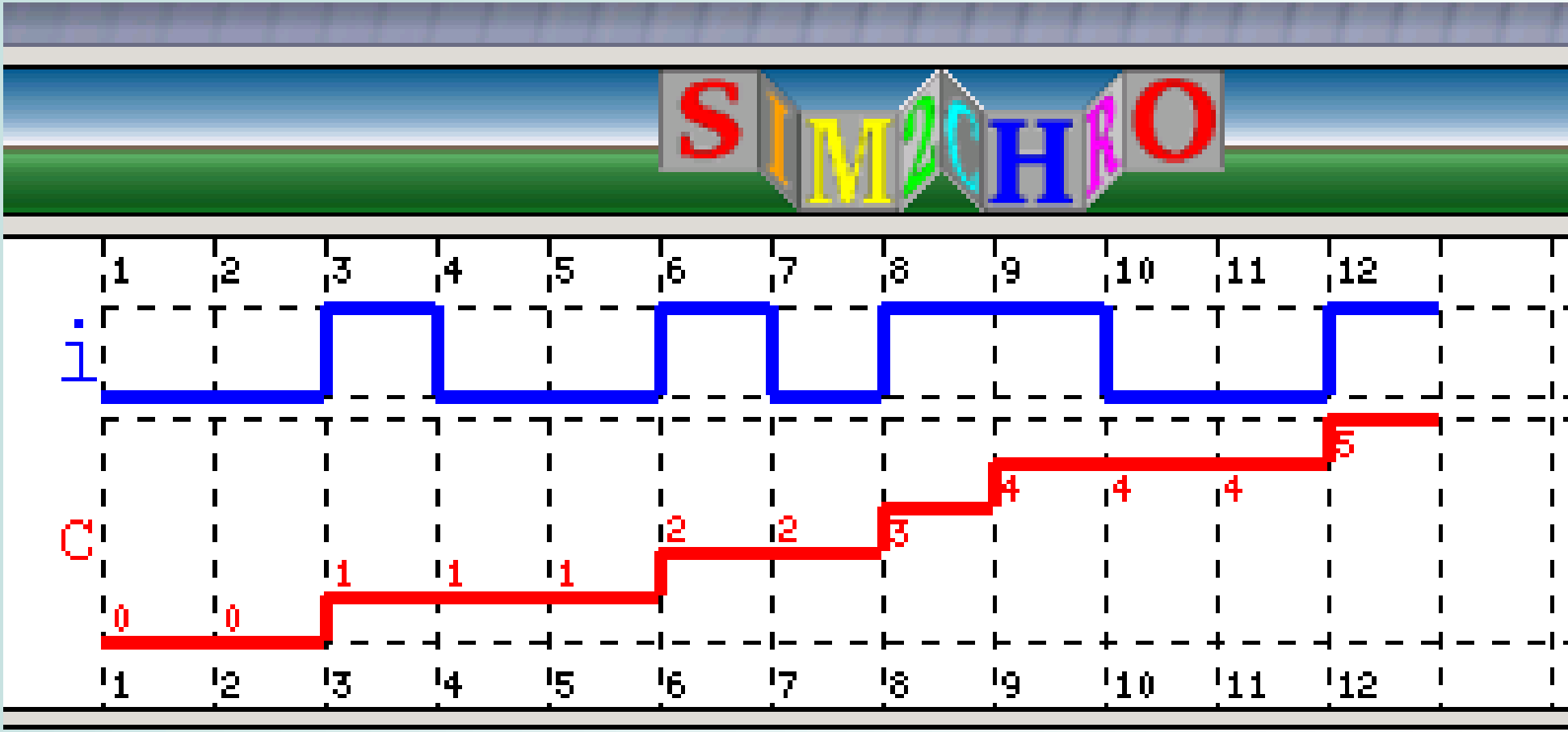
Exercice: draw the corresponding Moore machine.
Then use this automaton to program the double-click.

Sub-question: a counter

```
node Counter (i : bool) returns (c : int) ;
let
   c = 0 -> (pre c + (if i then 1 else 0)) ;
tel.
```

Exercice: draw timing diagrams of this program.

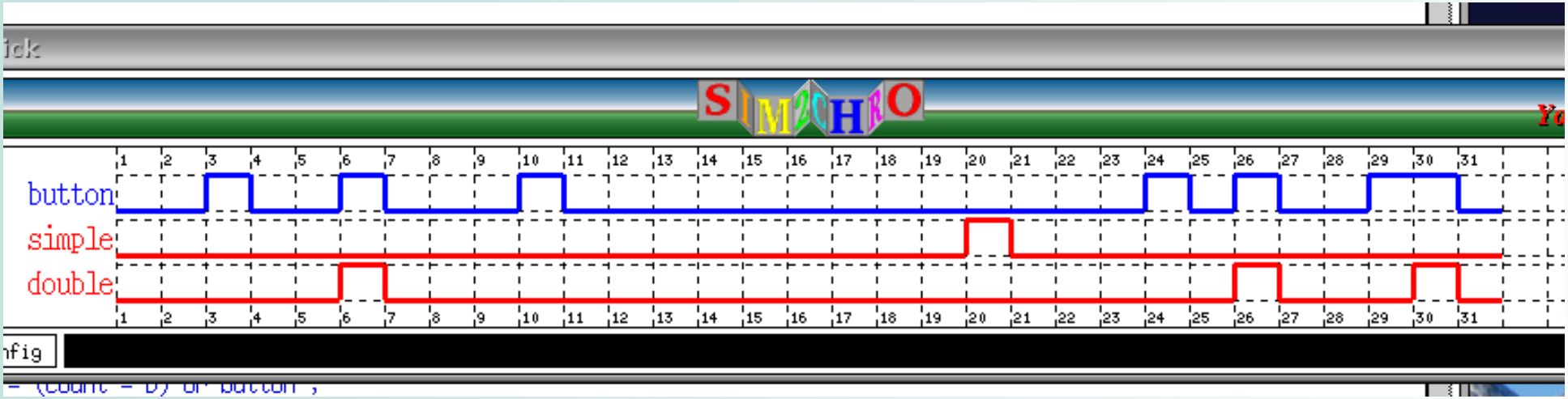
Counter: Simulation



Lustre Example: Double-Click

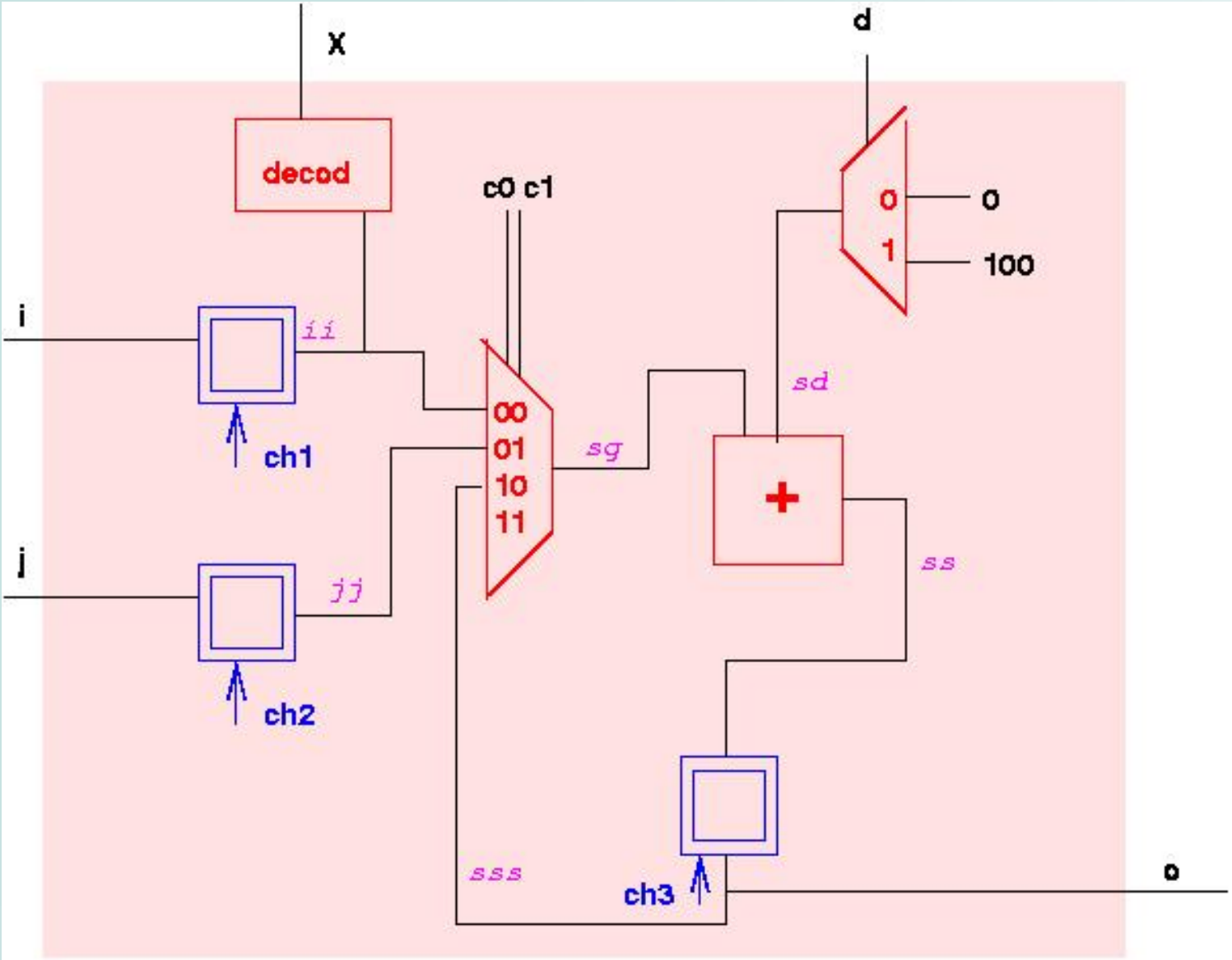
```
const D : int = 10 ;

node doubleclick (button : bool) returns
                 (simple : bool ; double : bool) ;
var  foo, yaclick, reset : bool ;
     count : int   ;
let
        (foo, yaclick) = twostates (button, reset) ;
        count = if foo then 0 else pre(count) + 1 ;
        reset = (count = D) or button ;
        simple = (count = D) ;
        double = yaclick and button ;
tel.

node twostates (a, b : bool) returns (un, deux : bool) ;
let     un = true -> pre ((un and not a) or (deux and b)) ;
        deux = false -> pre ((deux and not b) or (un and a)) ;
tel.
```

Double-Click: Simulation

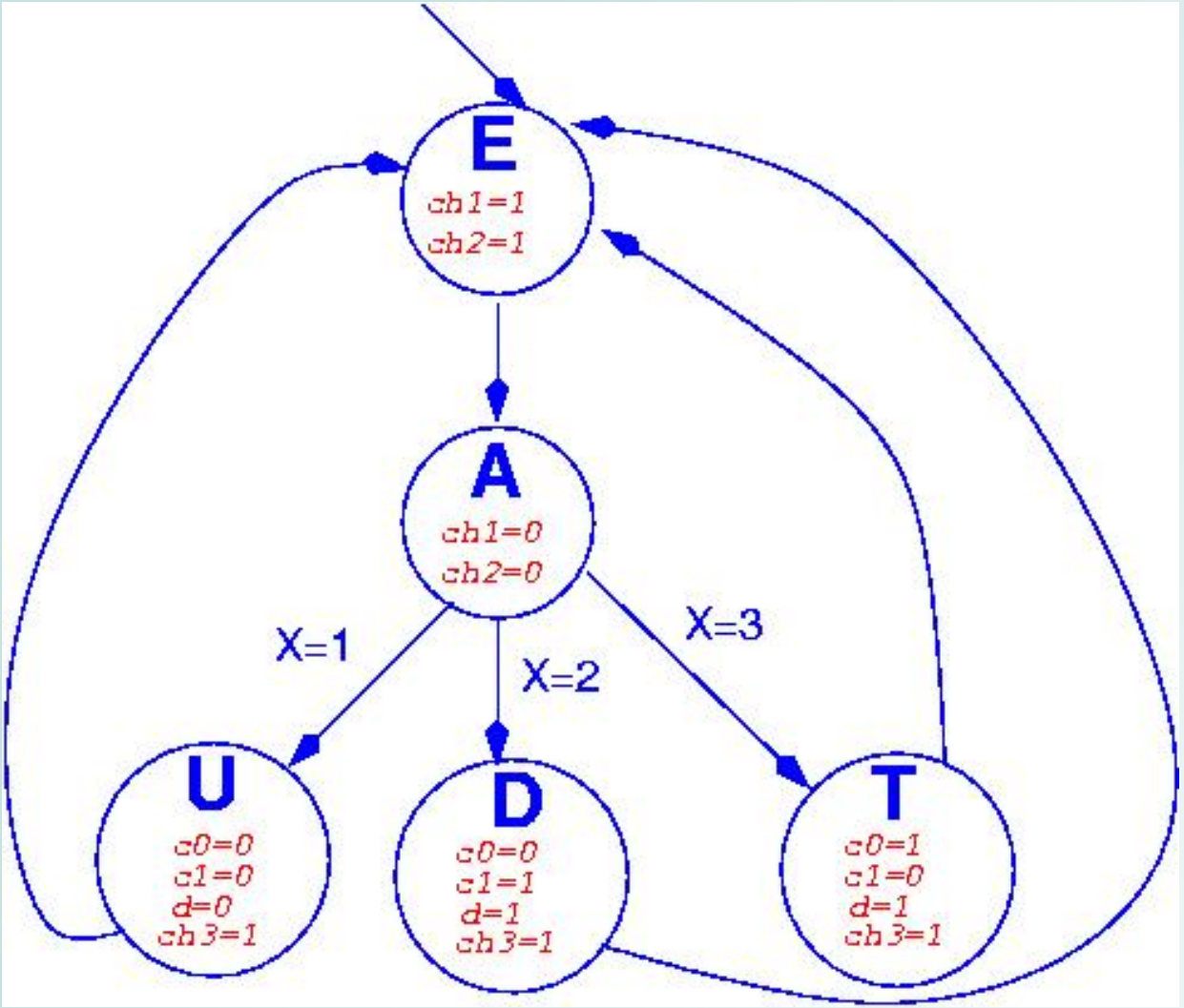


7 Synchronous Models
- Mealy Machines
- Moore Machines
- Expressivity of the Models
- Summary on automata models
- Synchronous Dataflow Models and Lustre/SCADE
- A new look at: Control/Operative Parts

An Operative Part



A Control Part



Encoding into Lustre: the operative part

```
node po (i, j : int  ; c0, c1, d, ch1, ch2, ch3 : bool)
returns (o, X : int) ;
var
    ii, jj, sg, sd, ss, sss : int ;
let
    ii = 0 -> pre (if ch1 then i else ii) ;
    jj = 0 -> pre (if ch2 then j else jj) ;
    sss = 0 -> pre (if ch3 then ss else sss) ;
    ss = sg + sd ;
    sg = if not c0 and not c1 then ii
         else if not c0 and c1 then jj else sss ;
    sd = if not d then 0 else 1 ;
    o = sss ;
    X = decod (ii) ;
tel.
```

Encoding into Lustre: the decoder

```
node decod (ii : int) returns (x : int) ;
let
    x = if ii < 10 then 1
        else if  ii < 100 then 2
        else 3 ;
tel.
```

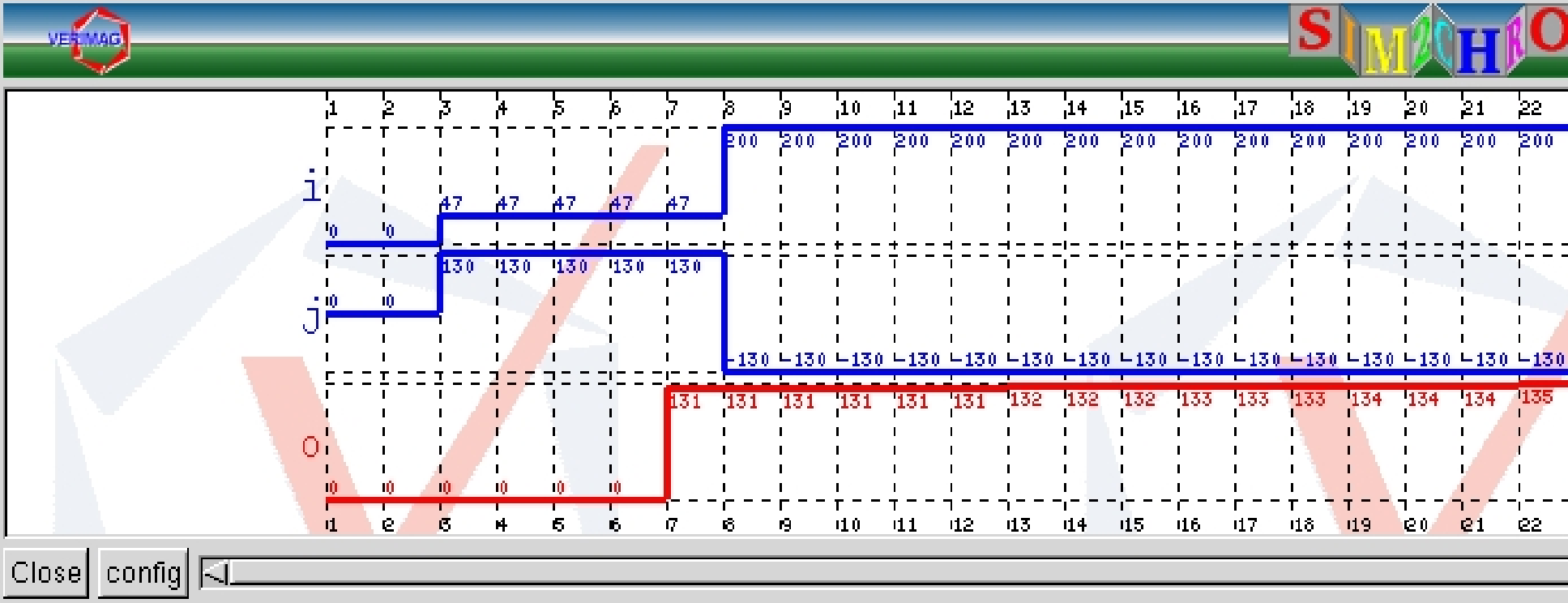
Encoding into Lustre: the control part

```
node pc (x : int) returns (ch1, ch2, ch3, c0, c1, d : bool) ;
var E, A, U, D, T : bool ;
let
    E = true  -> pre (U or D or T) ;
    A = false -> pre (E) ;
    U = false -> pre (A and x=1);
    D = false -> pre (A and x=2) ;
    T = false -> pre (A and x=3) ;
    ch1 = E ;
    ch2 = E ;
    ch3 = U or D or T ;
    c0 = T ;
    c1 = D ;
    d = D or T ;
tel.
```

Encoding into Lustre: putting it all together

```
node main (i, j : int ) returns (o : int) ;
    x : int ;
    ch1, ch2, ch3, c0, c1, d : bool ;
let

    (o, x) = po (i, j, c0, c1, d, ch1, ch2, ch3) ;
    (ch1, ch2, ch3, c0, c1, d) = pc (x) ;
tel.
```

Encoding into Lustre: Example behaviors



Recommended Reading and Listening Complements on Synchronous Models and Languages

6 Why Models?

7 Synchronous Models

8 Asynchronous Models
- Introduction
- Principles of "Interleaving" Semantics
- Monoprocessor systems with shared memory
- Multiprocessor systems with shared memory
- Multicomputer systems

8 Asynchronous Models

- Introduction
- Principles of "Interleaving" Semantics
- Monoprocessor systems with shared memory
- Multiprocessor systems with shared memory
- Multicomputer systems

Why do we need asynchronous models?

a-synchronous = without a common clock

These systems involve asynchronous parallel activities:

- Asynchronous circuits
  (the gates are not synhronized by physical clocks, or there may exist several clock domains)
- Processes on a monoprocessor system with shared memory
  (two processes are never alive at the same time)
- Multiprocessor systems with shared memory
  (the various processors may have unsynchronized clocks)
- Multicomputer systems
  (internet... even worse)

Asynchronous Parallelism vs. Asynchronous Communication

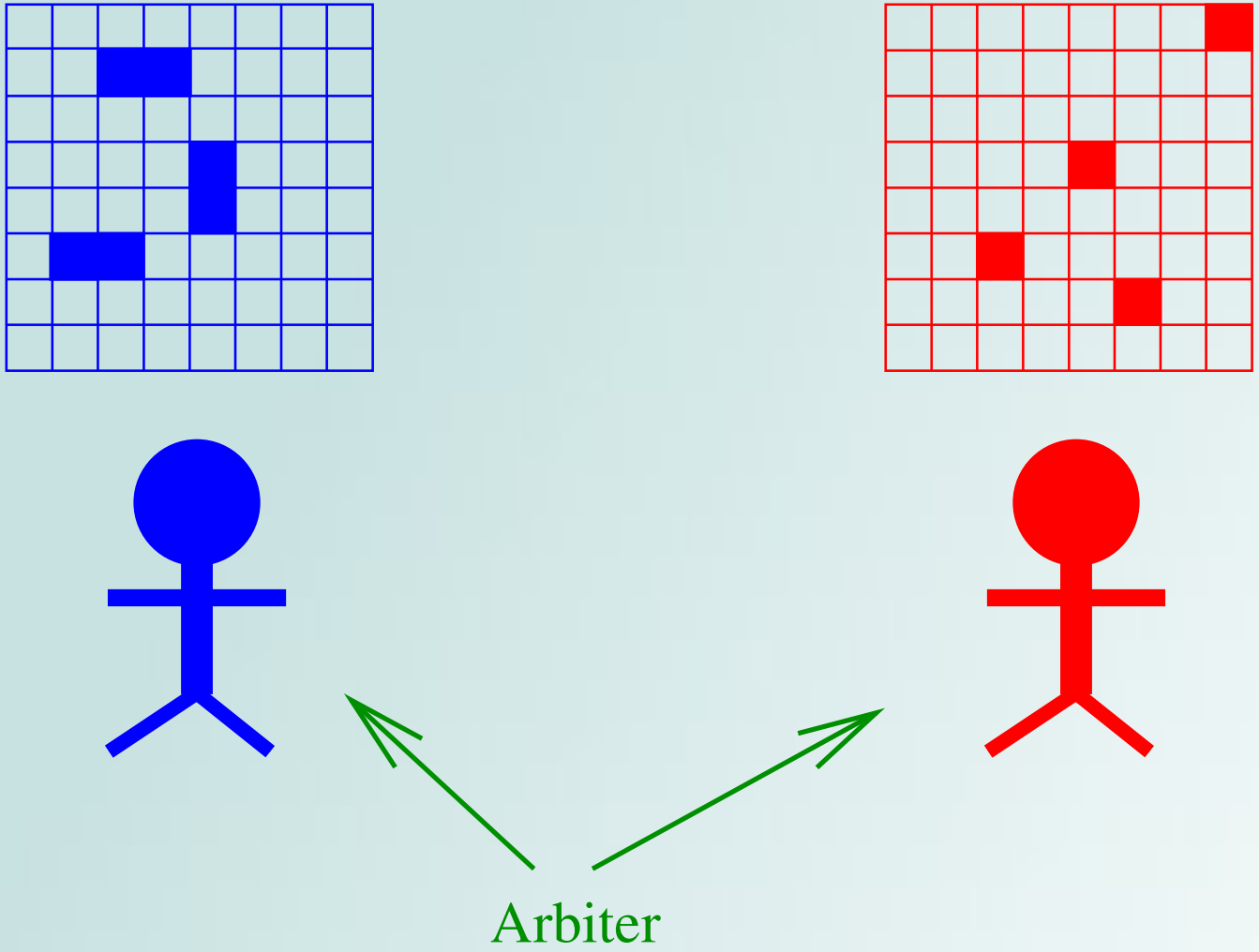
Asynchronous parallelism:
The systems do not share a common clock, they can evolve independently
Asynchronous Communication:
The system that sends a message and the system that receives it, are not alive "at the same time"; sending is non-blocking.
Typical image: mail.
as opposed to "synchronous communication", e.g.: rendez-vous, telephone calls, ...

Asynchronous Parallelism vs. Asynchronous Communication

Asynchronous parallelism:
The systems do not share a common clock, they can evolve independently
Asynchronous Communication:
The system that sends a message and the system that receives it, are not alive "at the same time"; sending is non-blocking.
Typical image: mail.
as opposed to "synchronous communication", e.g.: rendez-vous, telephone calls, ...

8 Asynchronous Models
- Introduction
- Principles of "Interleaving" Semantics
- Monoprocessor systems with shared memory
- Multiprocessor systems with shared memory
- Multicomputer systems

Two actors + an arbiter

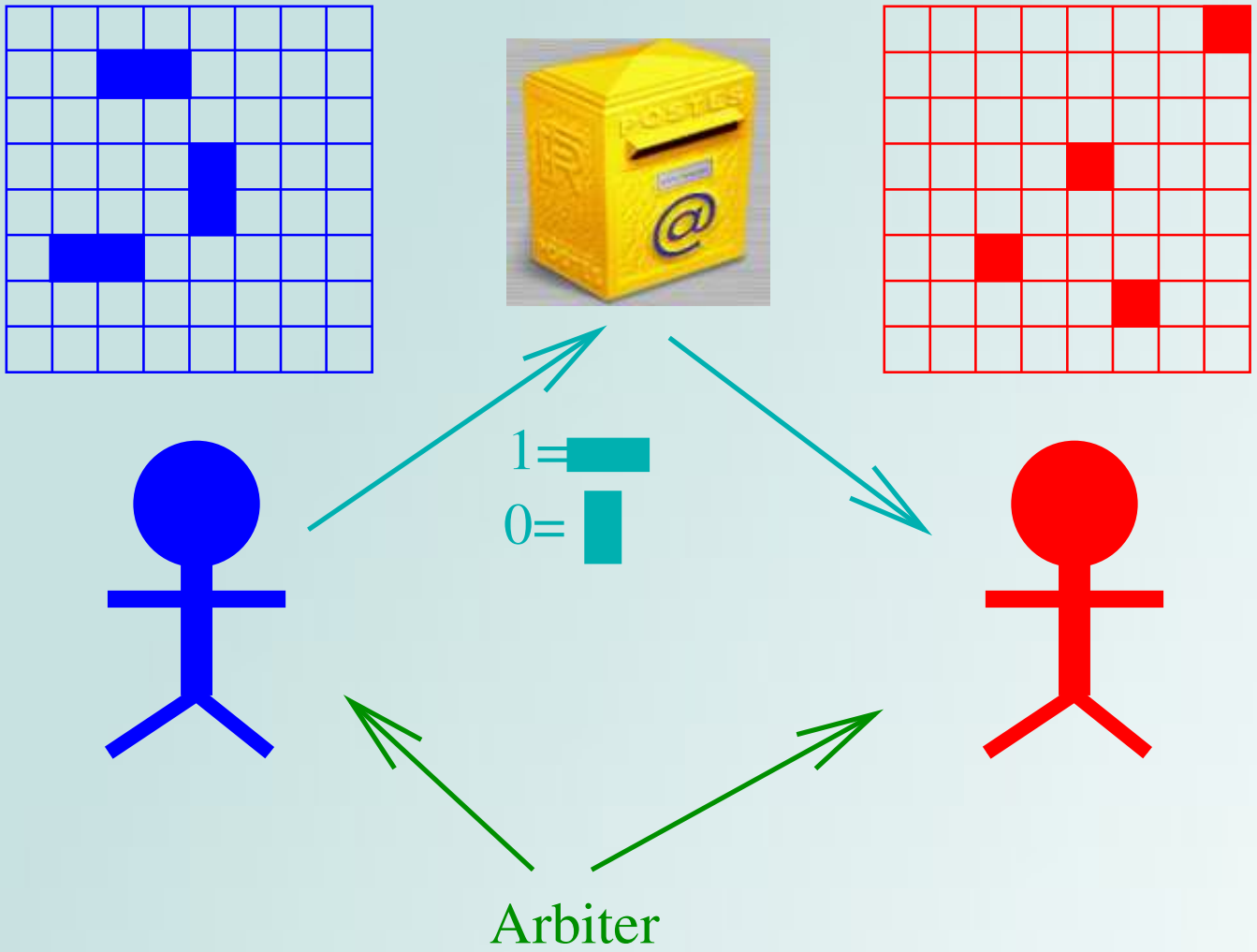


Intuition: "interleaving" semantics

- The player on the left may put two adjacent tokens at the same time, on his board
- The player on the right may put one token at a time, on his board
- The arbiter repeatedly selects one of the players by tossing a coin, and gives him the right to play for one action

What are the possible global states of the two boards?
How to represent the behavior of each player by an automaton?

Communication: adding a mailbox



Communication: adding a mailbox

- The player on the left may send a one bit letter to the other one, telling him whether he's just put a vertical block, or an horizontal block
- The player on the right does not put a block on one border when the other one has just put a vertical block
- The arbiter repeatedly selects one of the players by tossing a coin, and gives him the right to play for one action and one READ or WRITE operation on the mailbox.

What are the possible global states of the two boards (+ the mailbox)?
How to represent the behavior of each player by an automaton?

Communication: More questions

- Can the two players agree on their next moves?
  Example:
  - either the player on the left puts a vertical block, and the player on the right puts a block on a border
  - or, the player on the left puts an horizontal block, and the player on the right puts a block not on a border

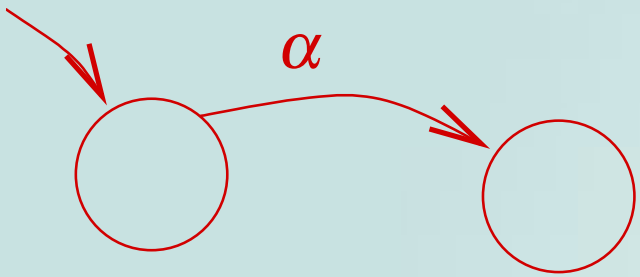
8 Asynchronous Models
- Introduction
- Principles of "Interleaving" Semantics
- Monoprocessor systems with shared memory
- Multiprocessor systems with shared memory
- Multicomputer systems

Monoprocessor systems with shared mem. Hypothesis

A monoprocessor computer, running multiple processes or threads thanks to a time-sharing scheduler. All processes (or threads) access the same memory.

Reading or writing a word from/to memory is made atomic by the HW.

Simple automata



$\alpha, \beta, \ldots$ are atomic. They can be :
— internal instructions of the processor
— memory reads
— memory writes

Formally: let L be the set of labels, an automaton is a tuple
$A = (Q, Q_0 \subseteq Q, T \subseteq Q \times L \times Q)$

(see also the notion of "process Graphs" p 874 in "Computer Systems, a programmer's perspective", Bryant and O'Hallaron 2003)

Determinism and reactivity

Determinism still makes sense. Same notion as before:

$$\forall q \in Q.(\exists(q, \ell, q_1) \in T \wedge \exists(q, \ell, q_2) \in T) \implies q_1 = q_2$$

Reactivity is less obvious because there is no clear distinction between inputs and outputs.

"Pure" asynchonous product

$A_i = (Q^i, Q_0^i, T^i)$ for $i = 1, 2$

$A_1 || A_2 = (Q^1 \times Q^2, Q_0^1 \times Q_0^2, T_{\text{Prod}})$ where:

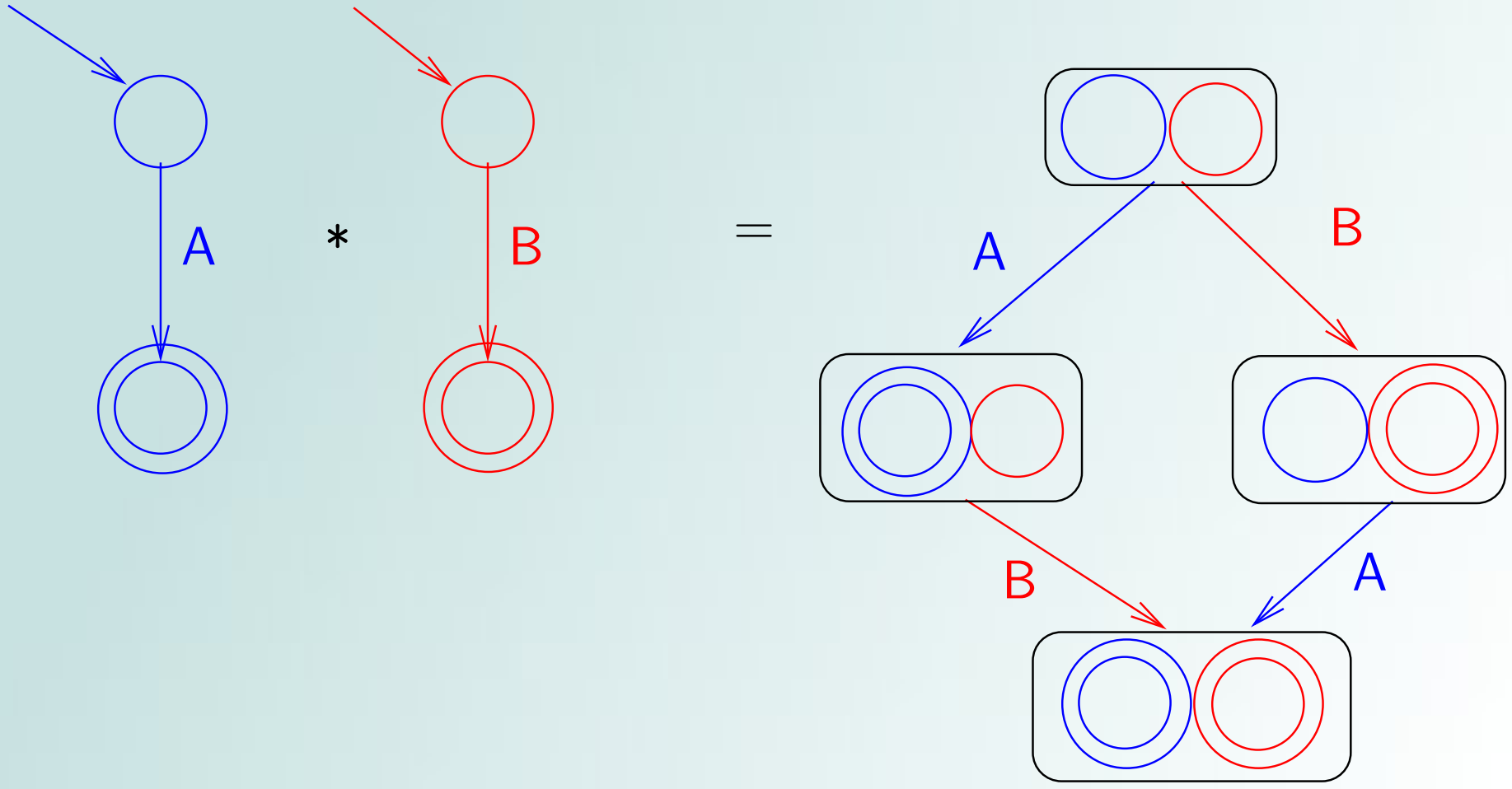
$$\frac{(q_1, \ell_1, q_1') \in T_1}{((q_1, q_2), \ell_1, (q_1', q_2)) \in T_{\text{prod}}} \qquad \frac{(q_2, \ell_2, q_2') \in T_2}{((q_1, q_2), \ell_2, (q_1, q_2')) \in T_{\text{prod}}}$$

A note on "granularity"

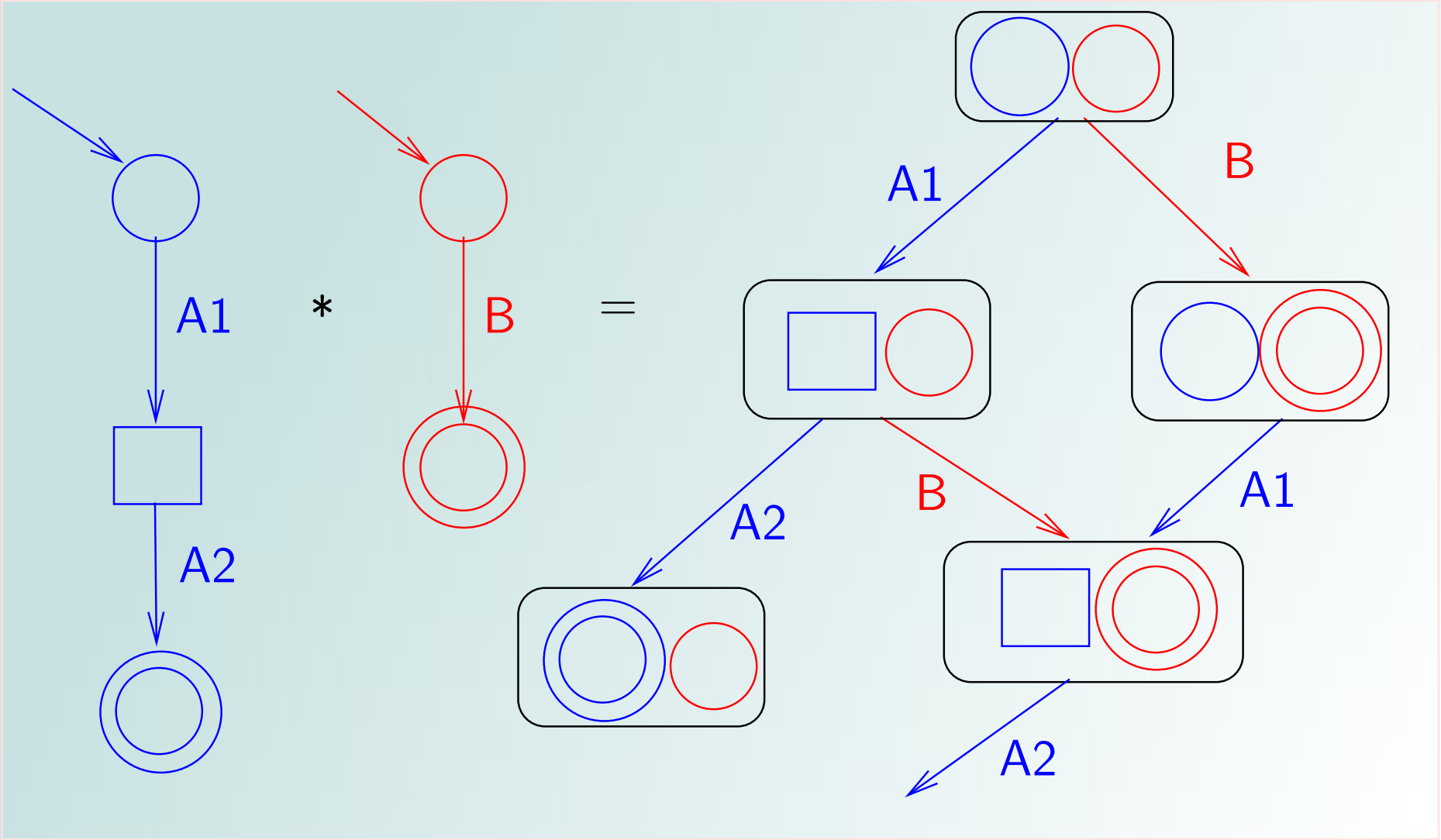
One transition = one atomic action.

The pure asynchronous product produces the whole set of global states that are potentially reachable when the two systems are executed with a scheduler that may stop them at any time, but only between two atomic actions (i.e., in a "state" of the model).

A note on "granularity" - example



A note on "granularity" - example

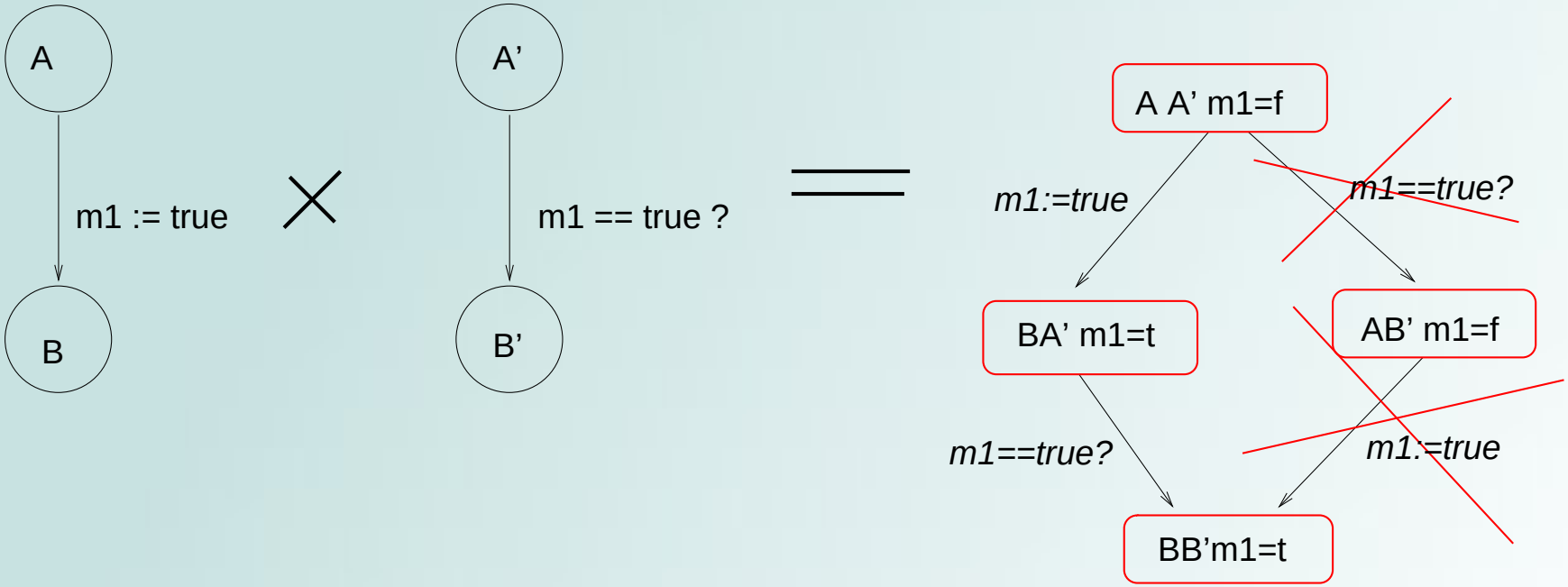


Communicating via the shared memory (1)

Since the two machines are never active *at the same time*, they can exchange information only by using a shared memory.

The states of the global process now include the state of the memory locations that are relevant for the processes (this set has to be known statically: no memory allocation, no process creation).

Communicating via the shared memory (2)



Modeling Mutual Exclusion Problems and Solutions

In terms of the automaton representing the global states, mutual exclusion means that a set of states $S = S_1 \times S_2$ (where S_i is a set of states of P_i), is unreachable.

Mutual exclusion: solution 1
avoiding global states by masking interrupts

Masking interrupts is a technical solution that can be modeled by transition fusion: a sequence of transitions is compacted into a single transition, to be considered as *atomic*.

An atomic sequence should not be too long: this mechanism is used to build synchronization primitives like semaphores, not to ensure mutual exclusion of large portions of code.

Semaphores (Dijkstra 1965)

Semaphores are a higher-order construct, built on the atomicity of a sequence of actions (obtained by masking interrupts, for instance).

A semaphore is a tuple made of an integer (a counter) and a FIFO: `s = (s.c, s.f)`

With two operations:

- `P(s) : s.c-- ;` if `s.c` < 0 then process is blocked in `s.f`
- `V(s) : s.c++ ;` if `s.c` ≤ 0 then activate one process blocked in `s.f`

Modeling Semaphores

Simple automata in which the atomic actions are:

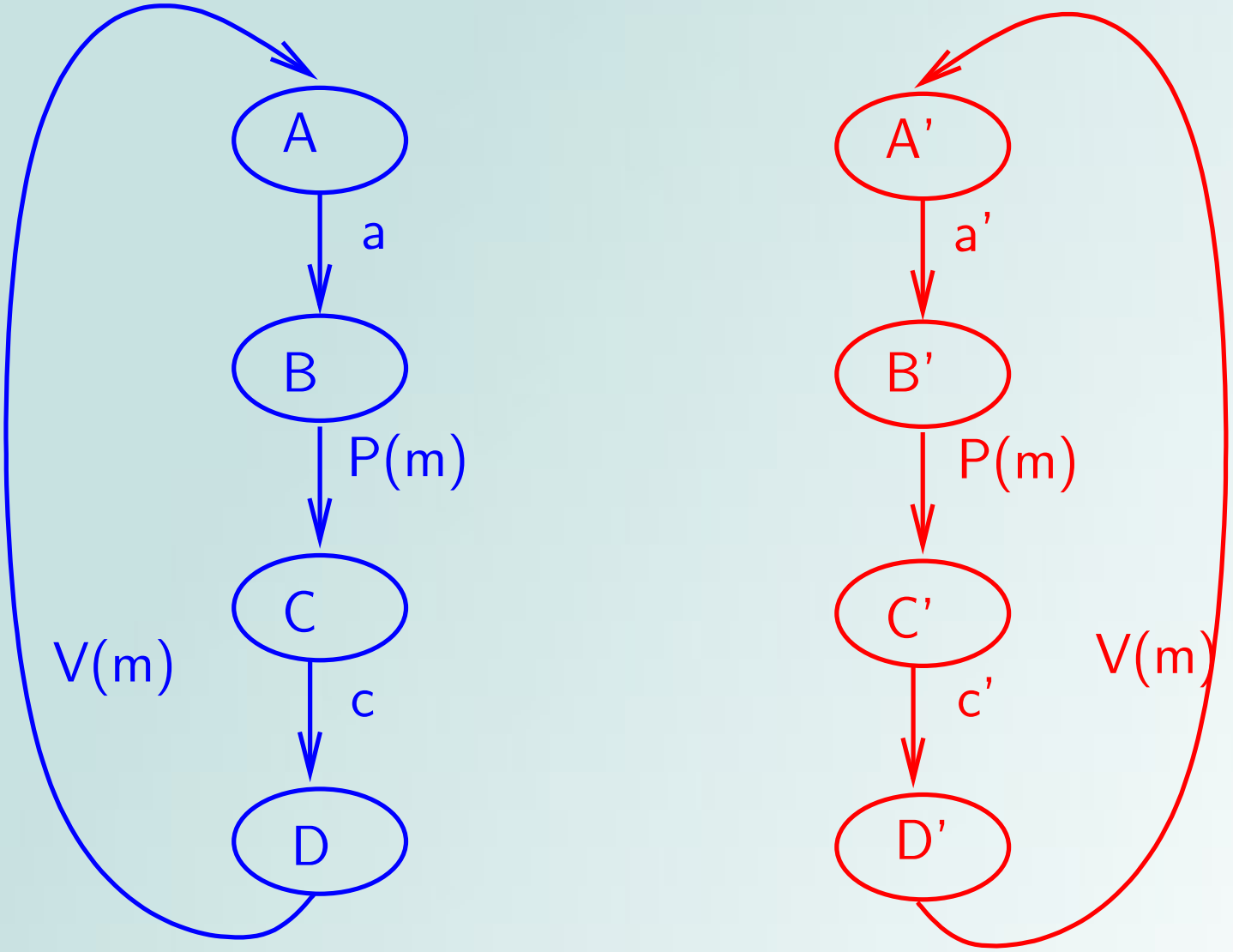
- Internal actions
- Memory accesses (read, write)
- `V(s)` or `P(s)` : actions on a semaphore `s`, considered as atomic

Products: the global state includes the state of the semaphores involved. The transitions labeled by `P(s)` or `V(s)` have an effect on this part of the global state, according to the usual rules (previous slide).

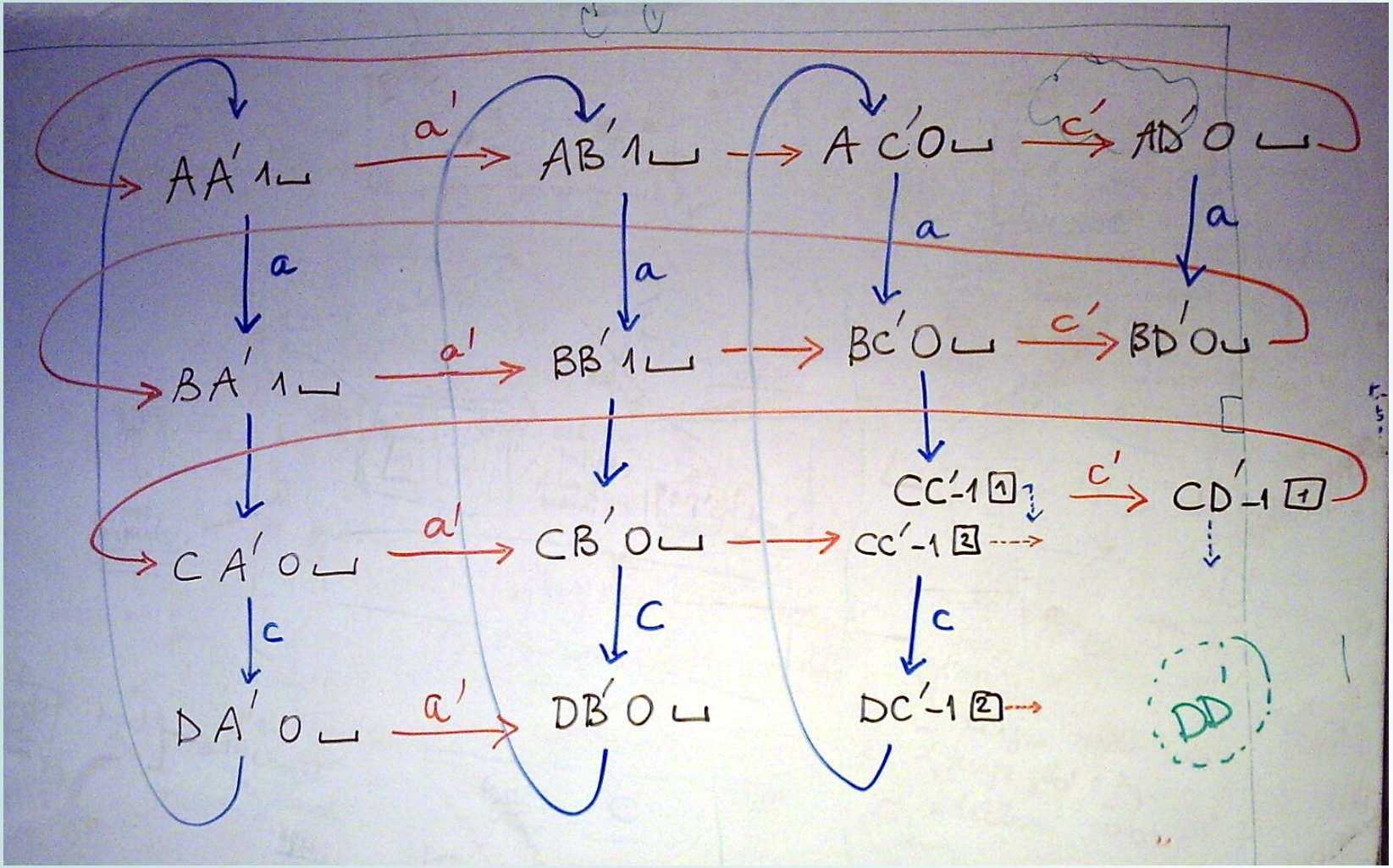
Building mutual exclusion with semaphores (1)

```
semaphore mutex;     // variable's value is
                     // restricted to 0 or 1.

  P(mutex);          // execute a P on mutex before
    ...              // entering critical section
    ...


  V(mutex);          // execute a V on mutex when
                     // leaving critical section
```

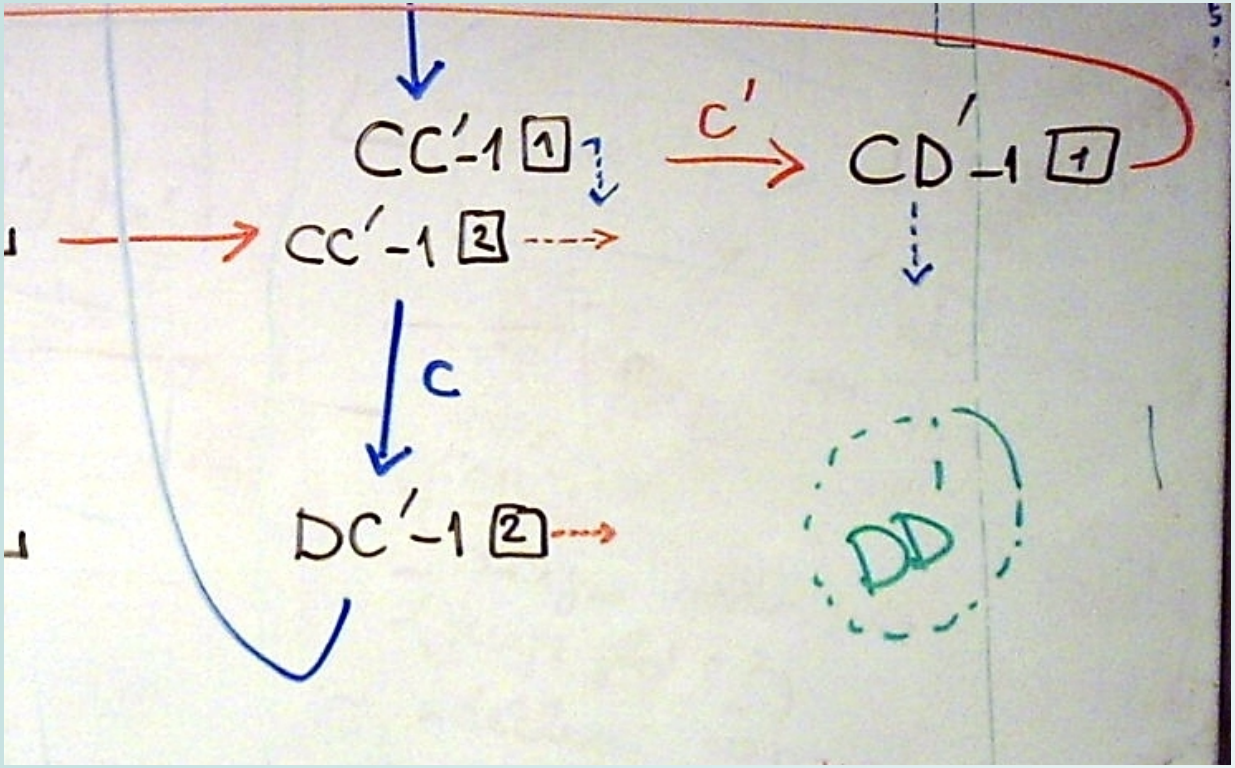
Mutual Exclusion: the Processes



Mutual Exclusion: the Product



Mutual Exclusion: the Product



DD' is unreachable.

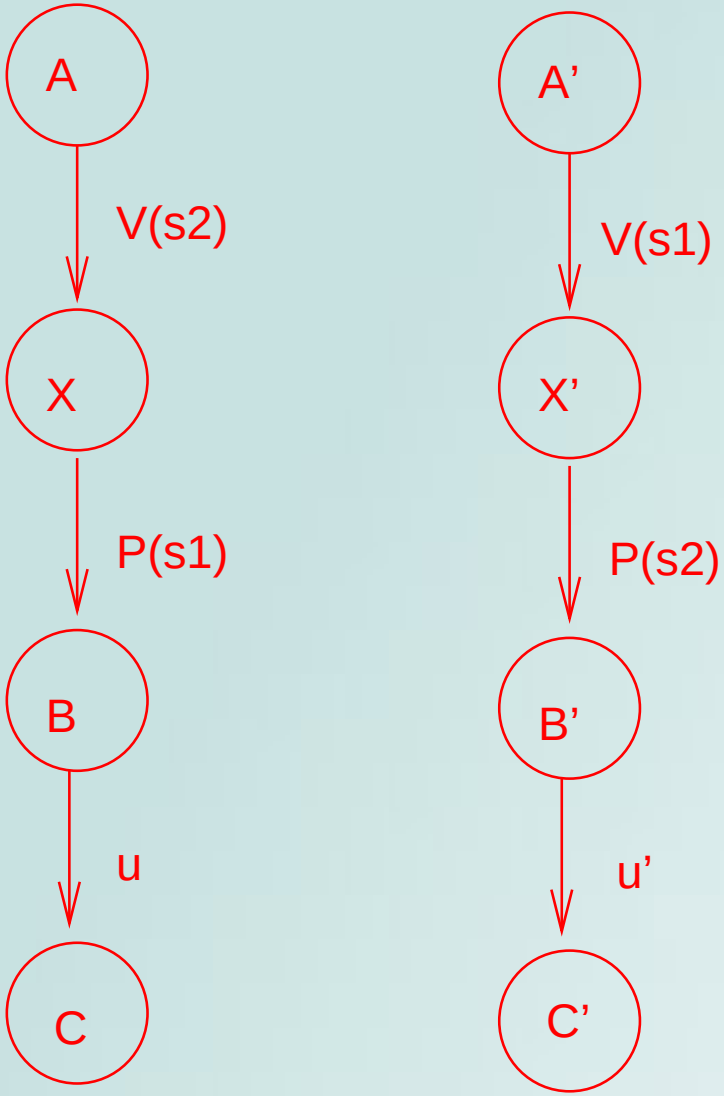
Building rendez-vous with semaphores (1)

Two processes 1 and 2 have to wait for each other before entering a section of their code (u for 1, u' for 2). Solution in the OS course:

Process 1 : `V(s2) ; P (s1) ; u ; ...`
Process 2 : `V(s1) ; P (s2) ; u' ; ...`

Each of the actions `V(s2)`, `P(s1)`, `u`, `V(s1)`, `P (s2)`, `u'` is atomic.

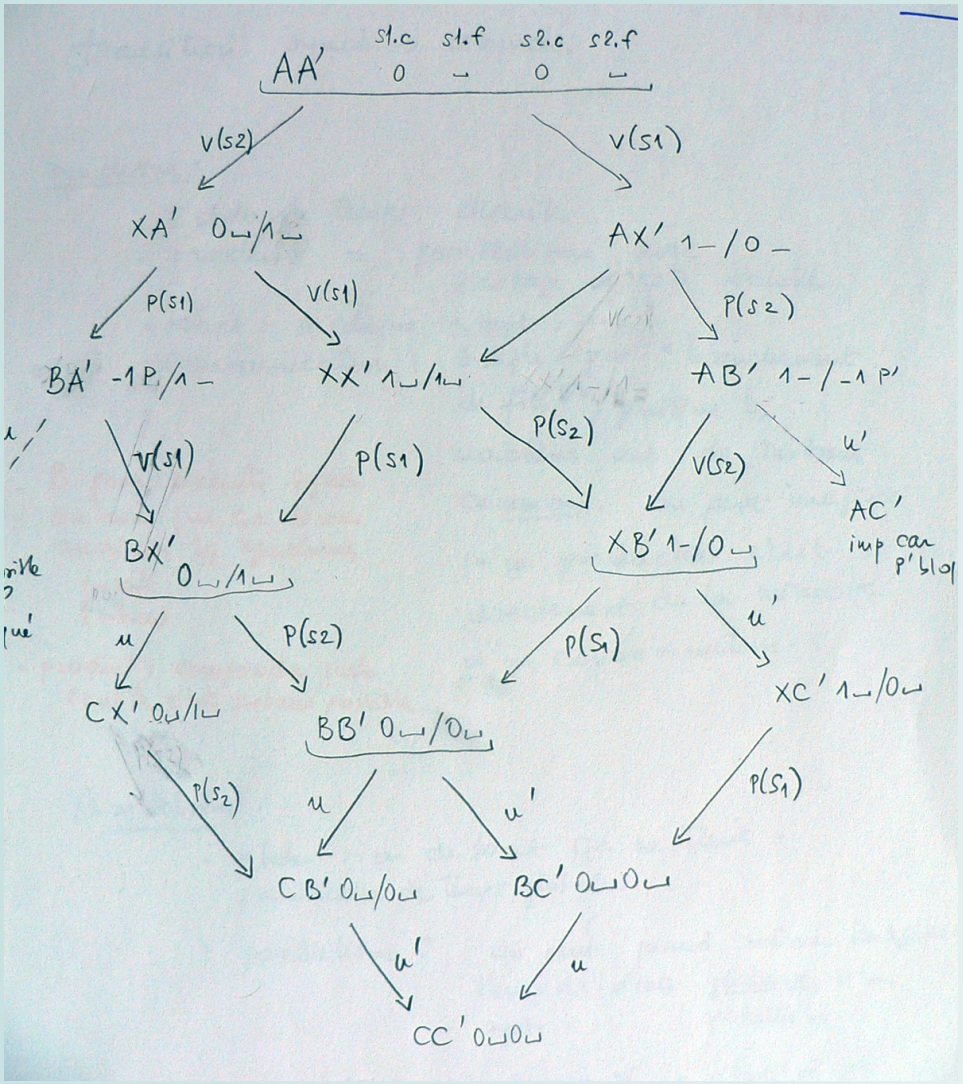
Building rendez-vous with semaphores (2)



A state of the product contains:

- the state of process 1 among A, X, B, C
- the state of process 2 among A', X', B', C'
- the state of `s1` : `s1.c, s1.f`
- the state of `s2` : `s2.c, s2.f`

Building rendez-vous with semaphores (3)

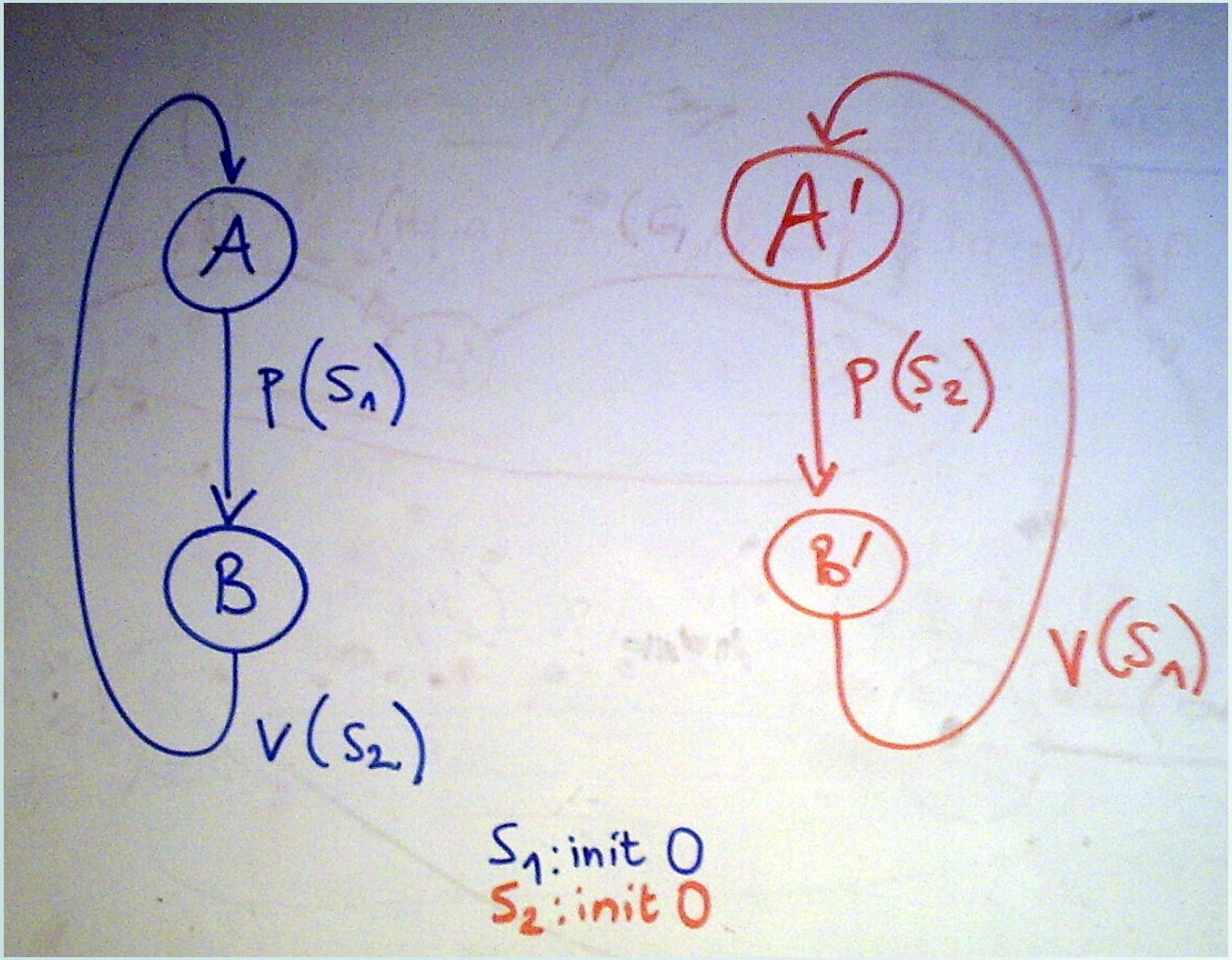


(Well Known) Potential problems of semaphore-based synchronizations

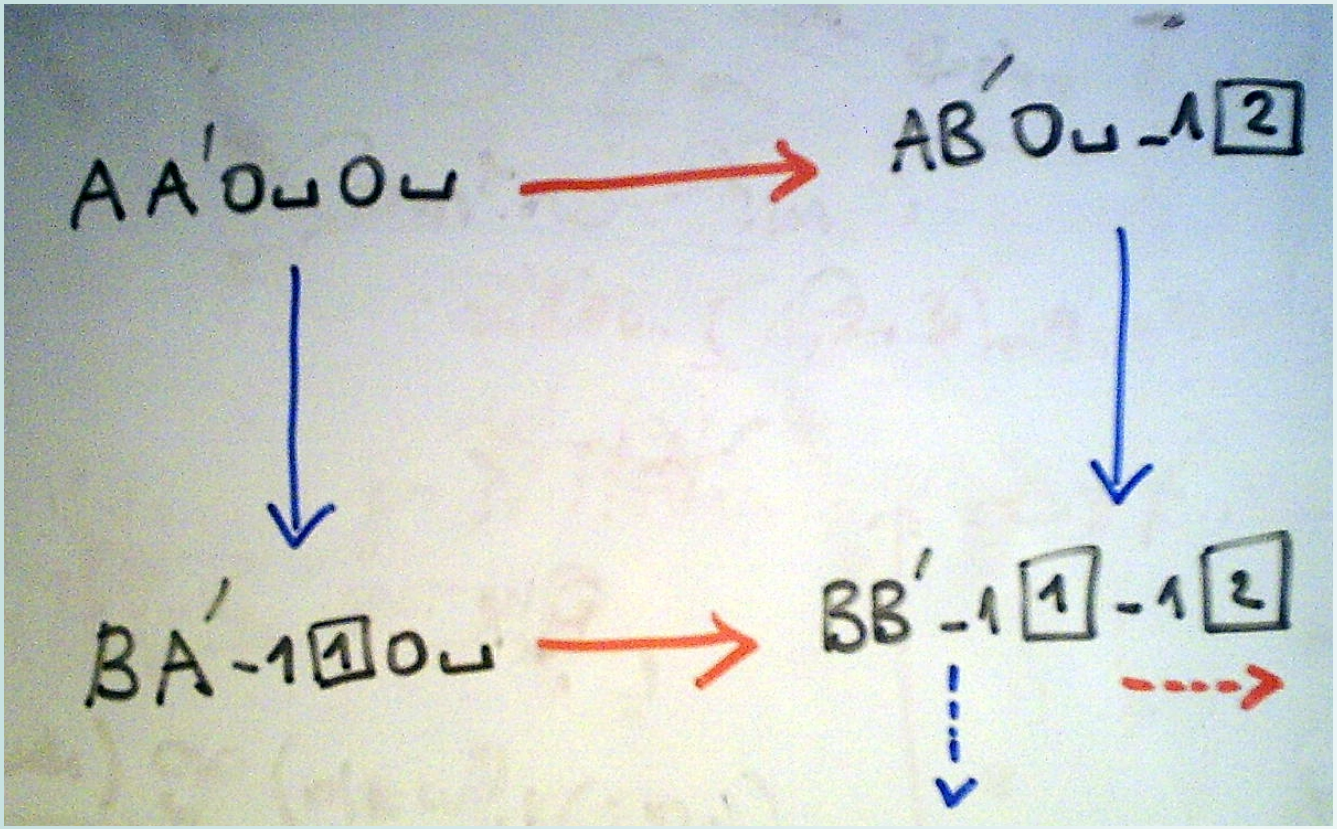
- Deadlocks (= sink states in the product) Example p 894 in [Bryant,O'Hallaron]
- Fairness/Starvation (related to the behavior of the scheduler. What happens the scheduler always chooses the same process?)

Exercice: invent a system that deadlocks.

Deadlocks (1)



Deadlocks (2)



Monoprocessor systems with shared memory: summary

- Masking interrupts allows to build semaphores
- Semaphores allow to program large critical sections
- *Other mechanisms: monitors*
- Synchronization problems: deadlocks

Automata models and products allow to answer all questions on deadlocks, unreachable states, etc.

Exercice 1 : 3-parties rendez-vous

A "counter" semaphore initialized with 3.
Atomic operations :
— `P(S)` decrements the counter
— `Z(S)` tests whether the counter is 0

Process number i executes: $\texttt{Bef}_i$; `P(S)` ; `Z(S)` ; $\texttt{Aft}_i$

Property: a process cannot execute its $\texttt{Bef}_i$ before all processes have executed $\texttt{Aft}_i$.

Develop the product and check the property.

8 Asynchronous Models

- Introduction
- Principles of "Interleaving" Semantics
- Monoprocessor systems with shared memory
- Multiprocessor systems with shared memory
- Multicomputer systems

Multiprocessor systems with shared mem. Hypothesis

A multiprocessor computer, each processor running multiple processes or threads thanks to a time-sharing scheduler.

All processors access the same memory (without caches).

Reading or writing a word from/to memory is atomic
Access conflicts are forbidden by hardware mechanisms.

Extended asynchronous product

$A_i = (Q^i, Q_0^i, T^i)$ for $i = 1, 2$, $A_1 || A_2 = (Q^1 \times Q^2, Q_0^1 \times Q_0^2, T_{\mathsf{Prod}})$

$$\frac{(q_1, \ell_1, q_1') \in T_1}{((q_1, q_2), \ell_1, (q_1', q_2)) \in T_{\mathsf{prod}}} \qquad \frac{(q_2, \ell_2, q_2') \in T_2}{((q_1, q_2), \ell_2, (q_1, q_2')) \in T_{\mathsf{prod}}}$$

$$\frac{(q_1, \ell_1, q_1') \in T_1, (q_2, \ell_2, q_2') \in T_2, \ell_1 \oplus \ell_2 \neq \bot}{((q_1, q_2), \ell_1 \oplus \ell_2, (q_1', q_2')) \in T_{\mathsf{prod}}}$$

Extended asynchronous product (cont'd)

Example Definition of the $\oplus$ operation on labels:

- (internal action i_1)$\oplus$ (internal action i_2) $= i_1 \cup i_2$
- (memory access address a_1)$\oplus$ (memory access address a_2) $\neq \perp$ only if $a_1 \neq a_2$
- (internal action i) $\oplus$ (memory access address a) $= i \cup a$

The hardware atomicity mechanisms are entirely expressed here.

Mutual exclusion: masking interrupts is no longer sufficient

Solutions:

- The Peterson algorithm uses the atomicity of Read and Write instructions, and busy-wait
- The Test-and-Set instruction guarantees the atomicity of a combined Read-Write instruction

The Peterson algorithm

Process 1

```
d0 := false;
loop
      <non critical section>
   a: d0 := true;
   b: tour := 0;
   c: wait(d1=false or tour=1);
   d: <critical section>
   e: d0 := false;
```

Process 2

```
d1 := false;
loop
      <non critical section>
   A: d1 := true;
   B: tour := 1;
   C: wait(d0=false or tour=0);
   D: <critical section>
   E: d1 := false;
```

The Peterson algorithm

Process 1

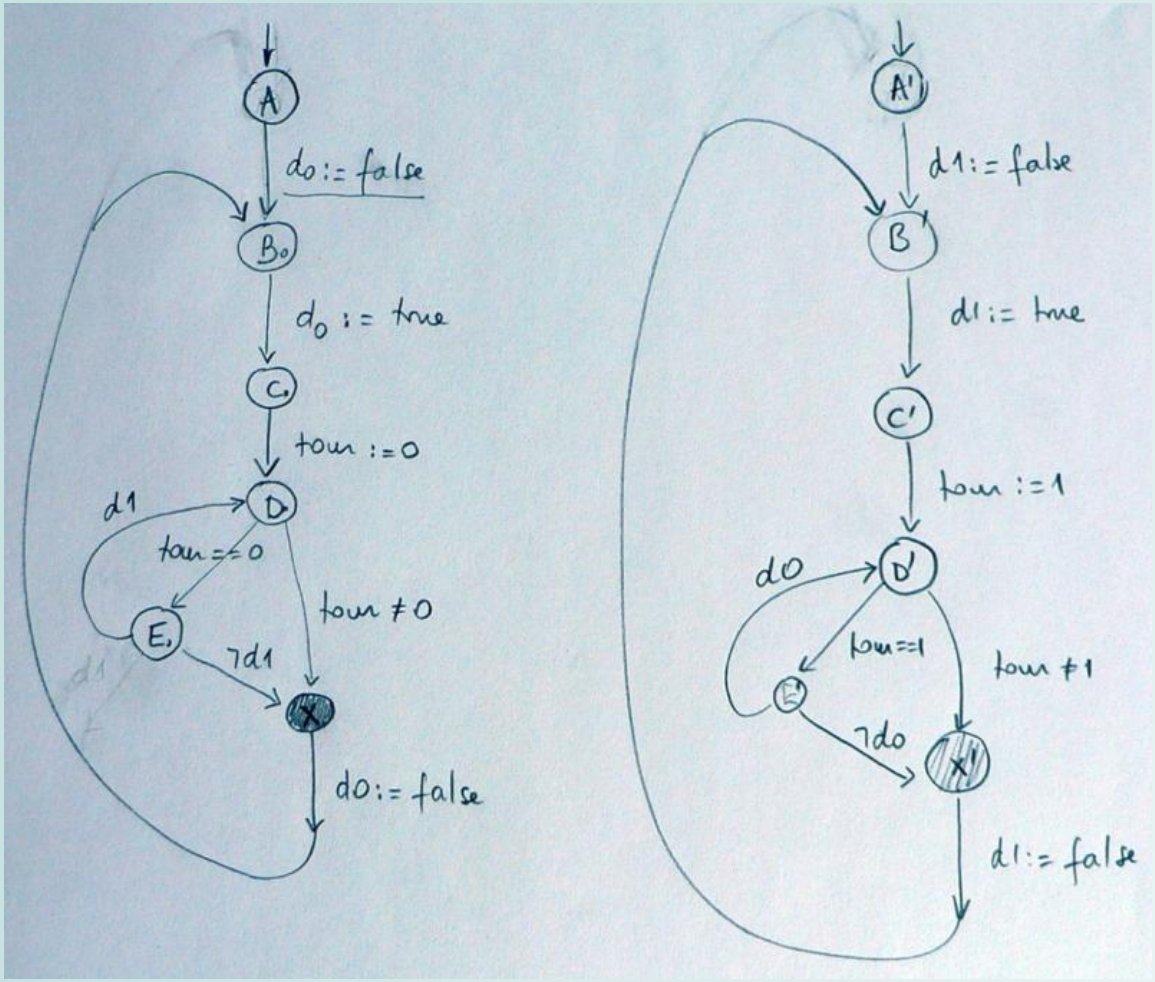
```
d0 := false;
loop
      <non critical section>
   a: d0 := true;
   b: tour := 0;
   c: wait(d1=false or tour=1);
   d: <critical section>
   e: d0 := false;
```

Process 2

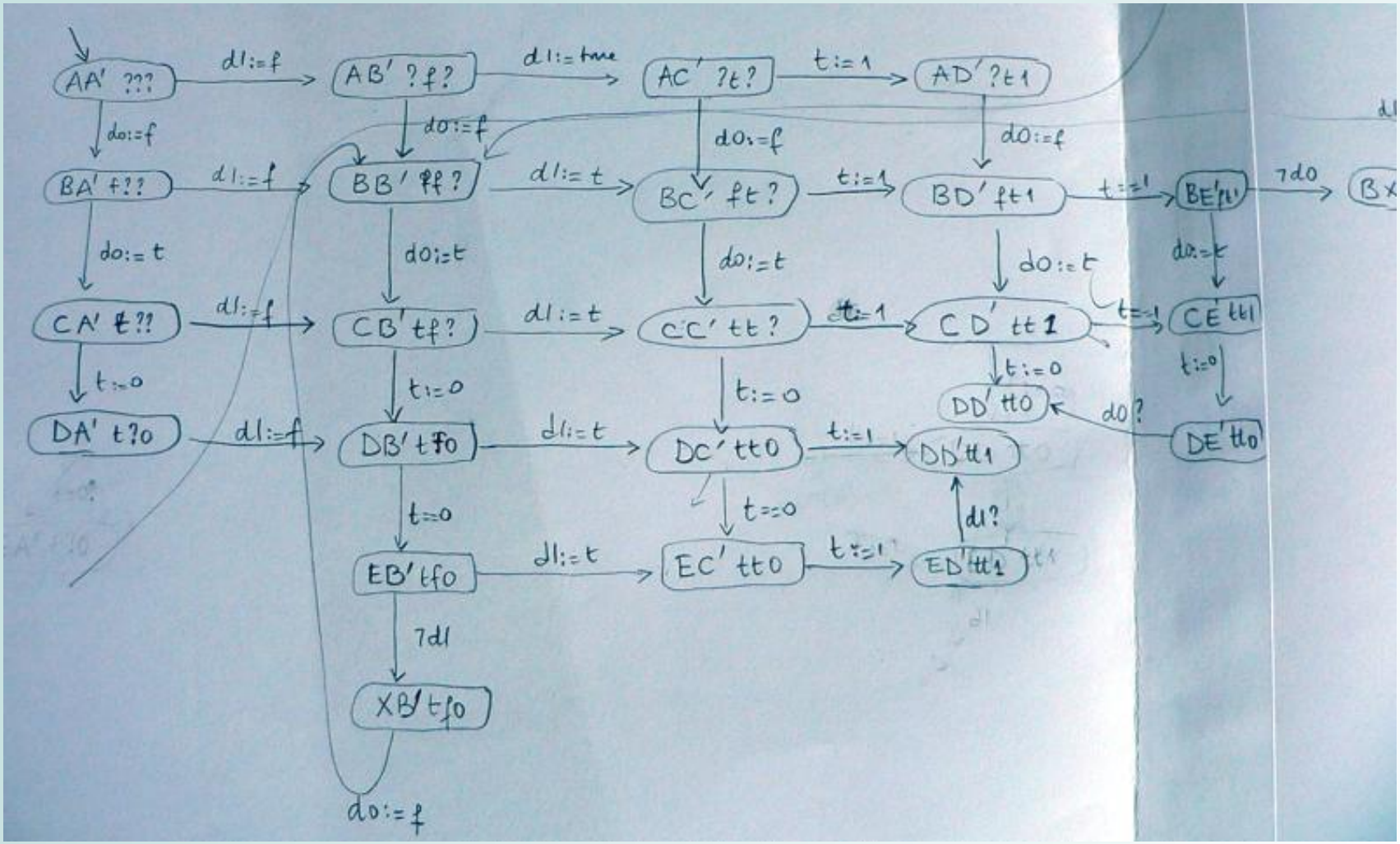
```
d1 := false;
loop
      <non critical section>
   A: d1 := true;
   B: tour := 1;
   C: wait(d0=false or tour=0);
   D: <critical section>
   E: d1 := false;
```

state = (
program counters a..e, A..E +
values of the variables d0, d1, tour
).

2 Automata for the Peterson Algorithm



Complete graph for the Peterson algorithm



Remarks:

In the previous slide, we suppose that testing that a variable is 0 or 1 can be made atomic. In fact, in assembly-line languages, we normally need an operation that sets the flags, and then a branch operation depending on the flags. The flags themselves are local variables of a process (since they are saved when commuting from one process to the other).
The complete automaton has more states, but the product is still ok.

The Test-and-Set (atomic) instruction

A simultaneous read/write operation that is guaranteed to be atomic by the HW: it is a single instruction, and the bus is locked during this operation.

Uses of the TAS instruction: `TAS (a) ; test of flags...`

`TAS (a)` is equivalent to: `Flags := according to MEM[a] ; MEM[a] := 1`

Example for SPARC

SPARC Instruction Set

MHS electronic

LDSTUB — Atomic Load/Store Unsigned Byte

LDSTUB

Operation: r[rd] ← zero extnd[r[rs1] + (r[rs2] or sign extnd(simm13))]
[r[rs1] + (r[rs2] or sign extnd(simm13))] ← FFFFFFFF H

Assembler Syntax: ldstub [*address*], reg_{rd}

Description: The LDSTUB instruction moves an unsigned byte from memory into the destination register, r[rd], and rewrites the same byte in memory to all ones, while preventing asynchronous trap interruptions. In a multiprocessor system, two or more processors executing atomic load/store instructions which address the same byte simultaneously are guaranteed to execute them serially, in some order.

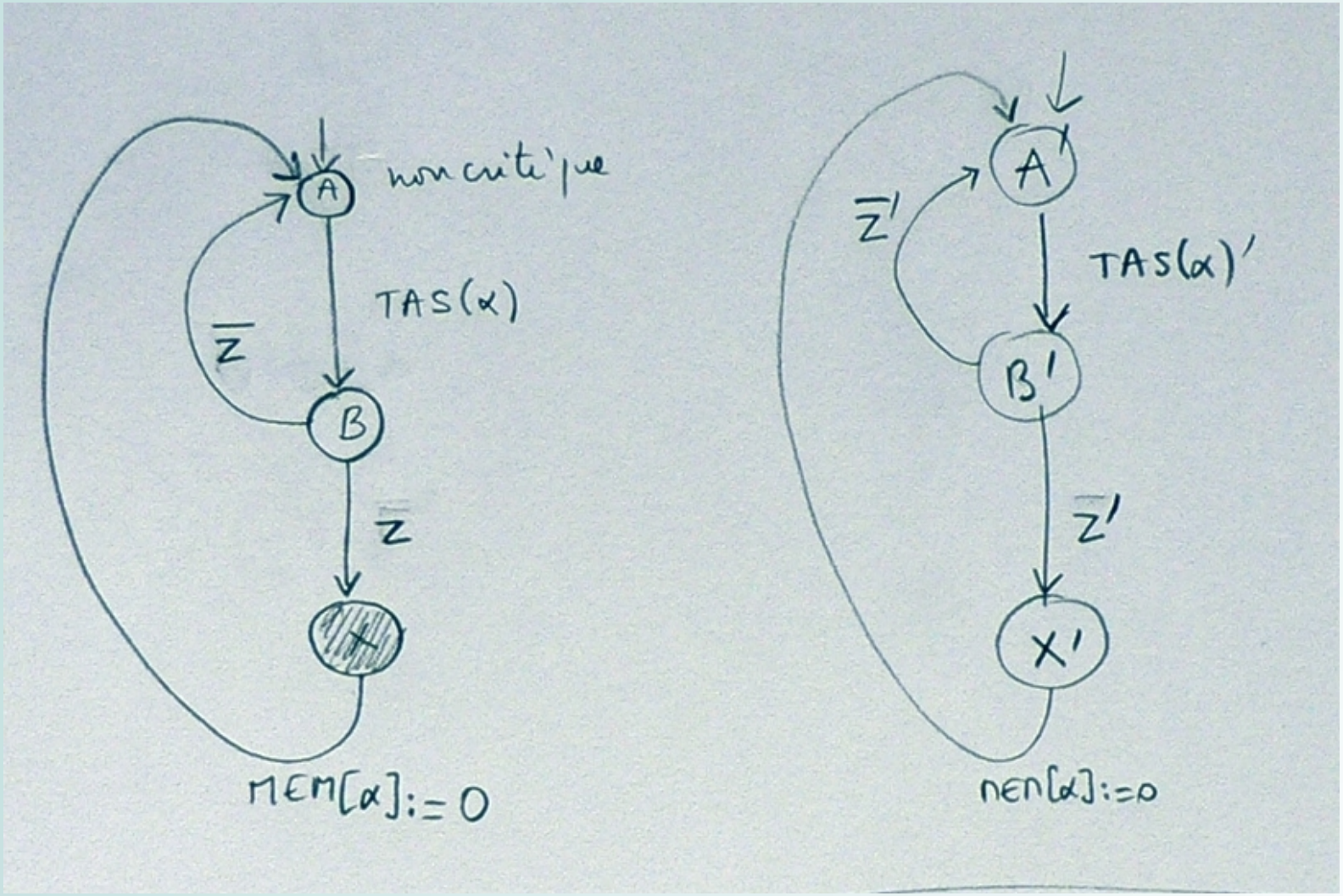
The effective memory address is derived by summing the contents of r[rs1] and either the contents of r[rs2] if the instruction's *i* bit equals zero, or the 13-bit, sign-extended immediate operand contained in the instruction if *i* equals one. The fetched byte is right-justified and zero-extended in r[rd].

If the instruction following an integer load uses the load's r[rd] register as a source operand, hardware interlocks add one or more delay cycles depending upon the memory subsystem.

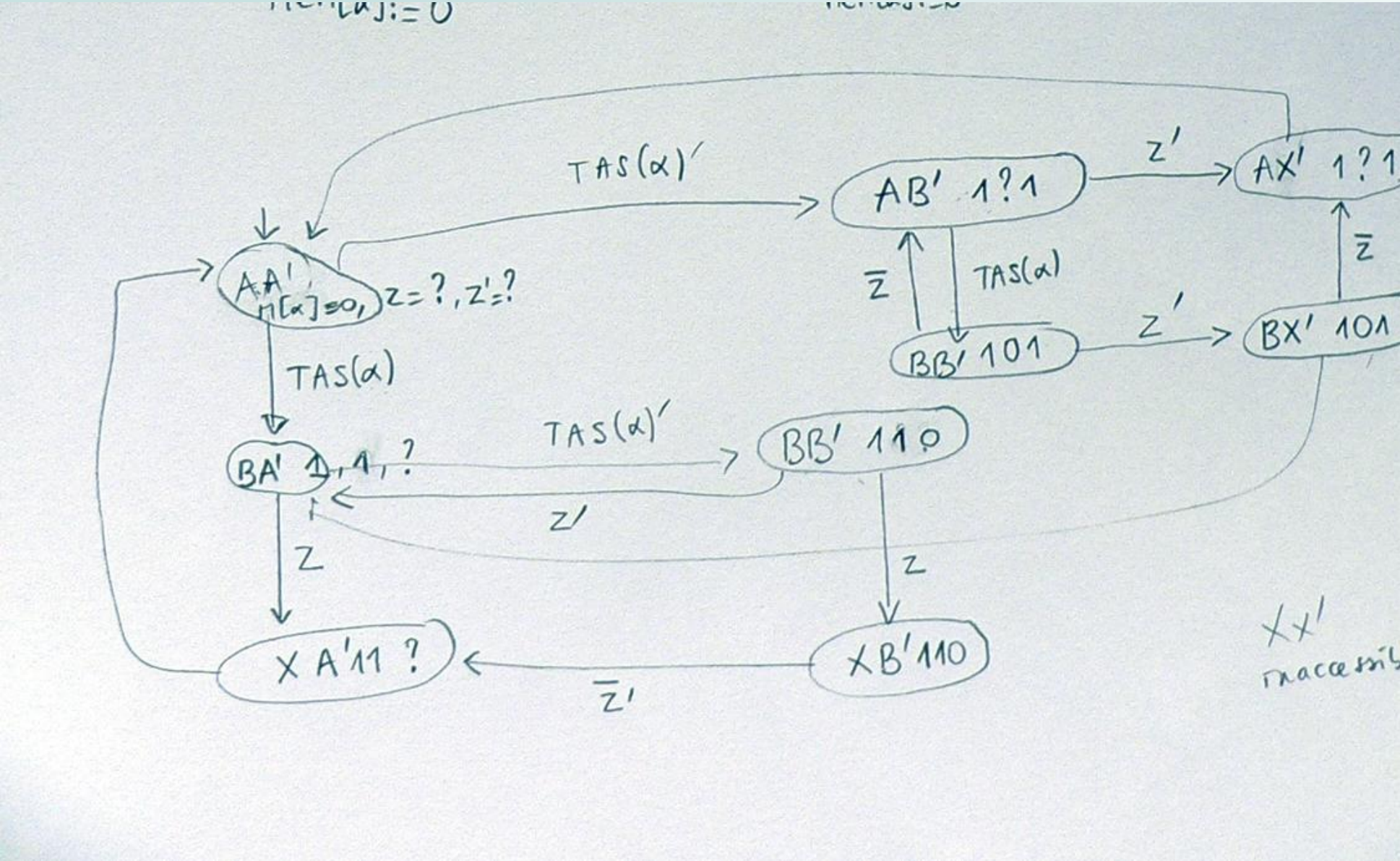
If LDSTUB takes a trap, the contents of the memory address remain unchanged.

Programming note: If *rs1* is set to 0 and *i* is set to 1, any location in the lowest or highest 4 kbytes of an

Mutual Exclusion with Test-And-Set (TAS)



Mutual Exclusion with Test-And-Set (TAS)



8 Asynchronous Models
- Introduction
- Principles of "Interleaving" Semantics
- Monoprocessor systems with shared memory
- Multiprocessor systems with shared memory
- Multicomputer systems

Multicomputer systems: Hypothesis

- No shared memory.
- Computers can exchange information and synchronize with each other via messages, that travel on links.
- The links may not be reliable, which means that protocols have to be used

Modeling protocols

We consider an established connection between two machines. We have to model:

- The behaviour of the sender process, trying to send a message
- The behaviour of the receiver process, listening to messages
- The behaviour of the link that may transmit or lose messages

A protocol is used to guarantee that messages are received in the order they are sent.

The Alternating-Bit protocol (wikipedia) - 1

Alternating bit protocol (ABP) means a simple data link layer network protocol that retransmits lost or corrupted messages.

Messages are sent from transmitter A to receiver B. Assume that the channel from A to B is initialised and that there are no messages in transit. Each message contains a data part, a checksum, and a one-bit sequence number, i.e. a value that is 0 or 1.

The Alternating-Bit protocol (wikipedia) - 2

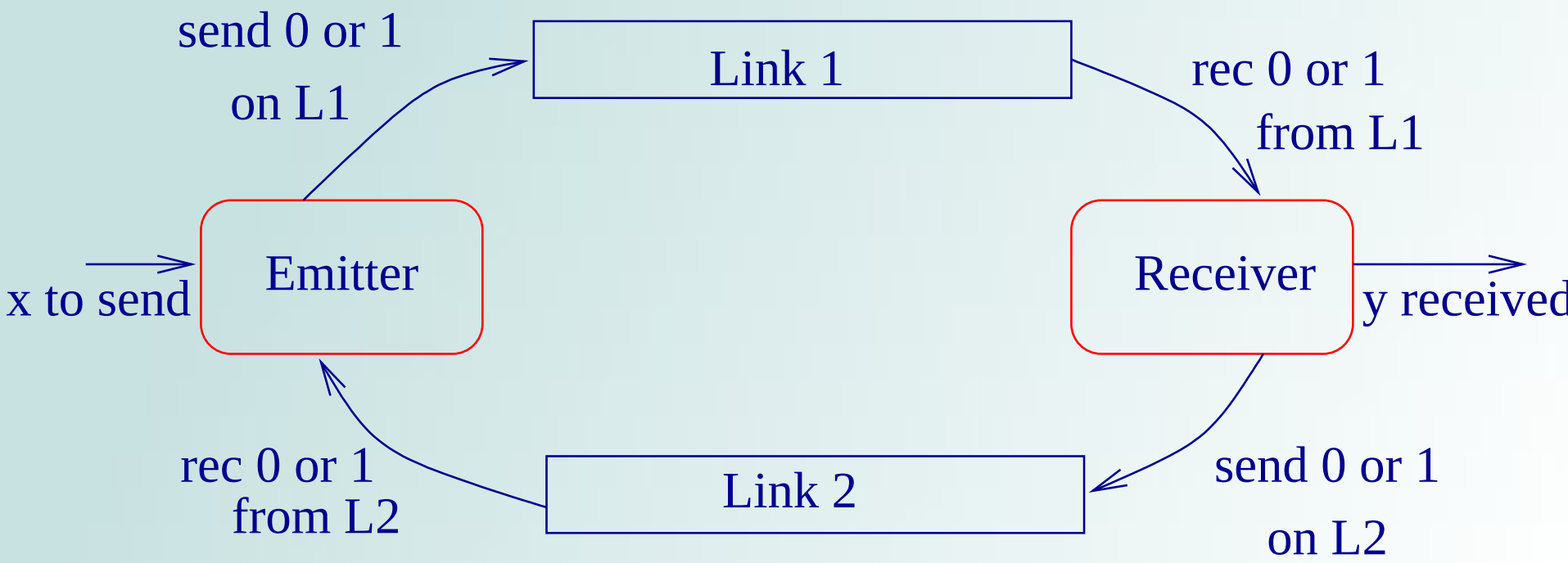
When A sends a message, it sends it continuously, with the same sequence number, until it receives an acknowledgment (ACK) from B that contains the same sequence number. When that happens, A complements (flips) the sequence number and starts transmitting the next message.

When B receives a message from A, it checks the checksum. If the message is not corrupted B sends back an ACK with the same sequence number. If it is the first message with that sequence number then it is sent for processing. Subsequent messages with the same sequence bit are simply acknowledged. If the message is corrupted B sends back an negative-acknowledge character (NAK). This is optional, as A will continue transmitting until it receives the correct ACK.

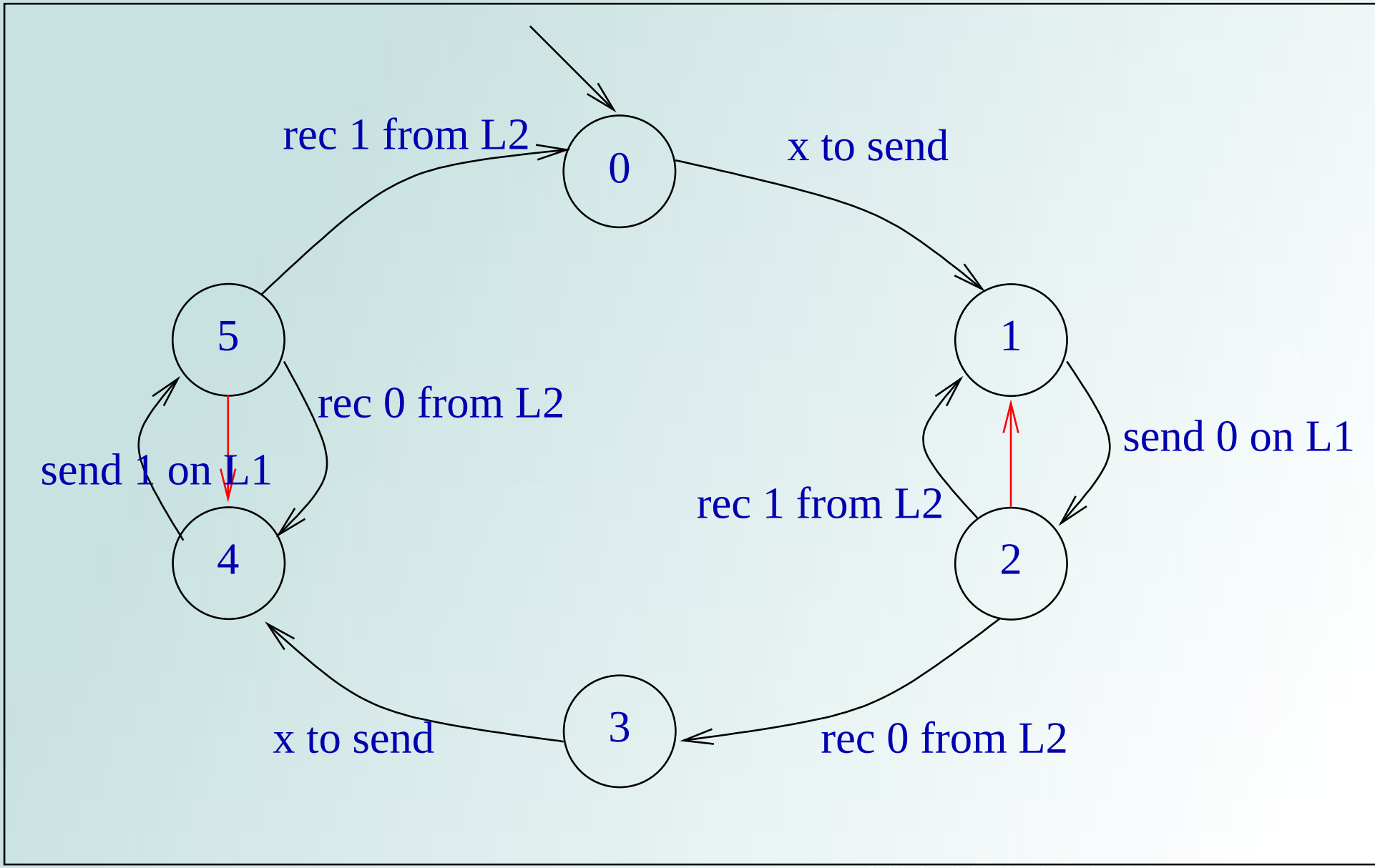
Modeling Abstractions

- The contents of the messages is irrelevant for checking the protocol
- We ignore the checksum
- How to model a lossy channel?

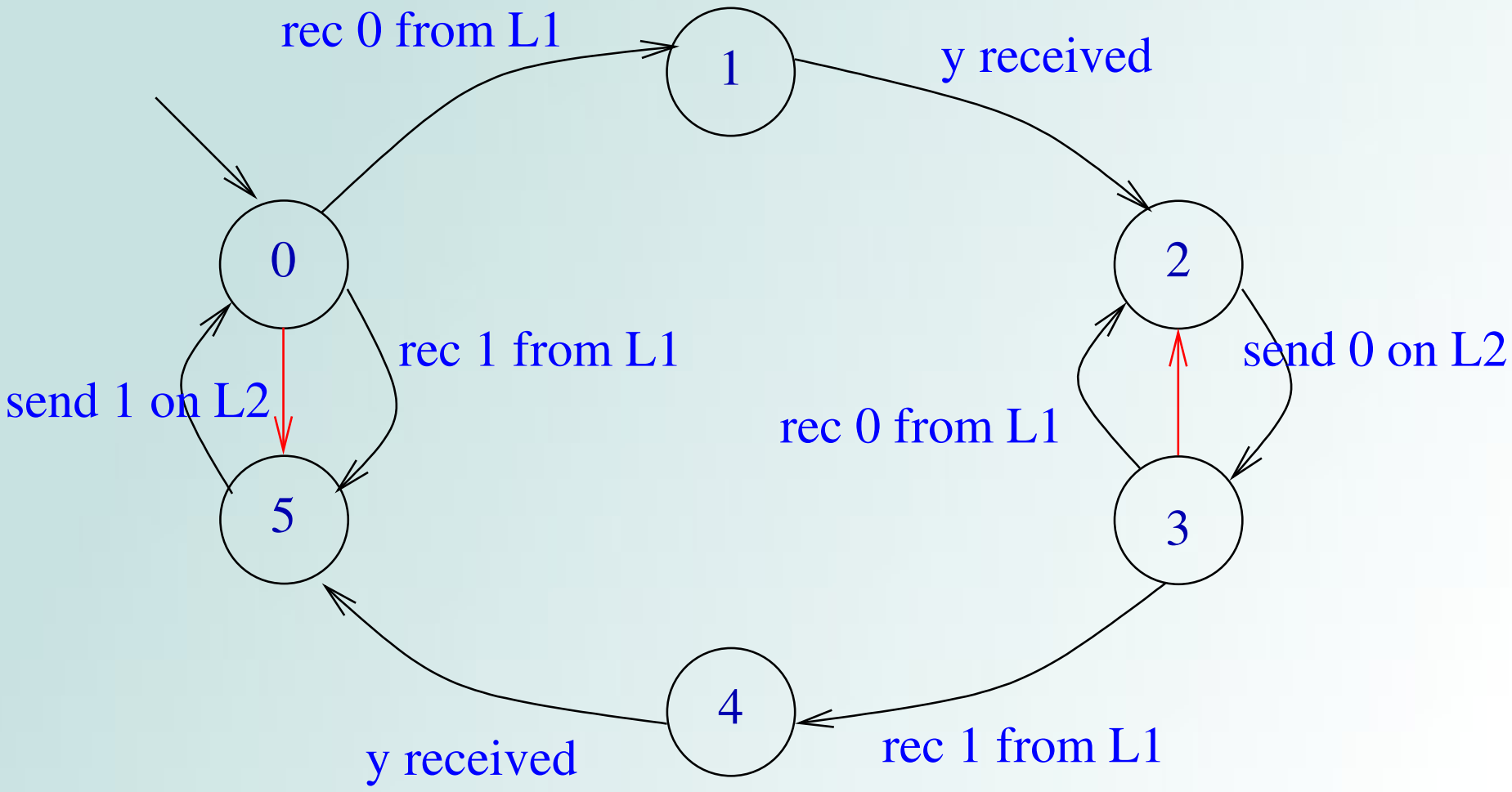
The global structure (it's not an automaton!)



The emitter automaton

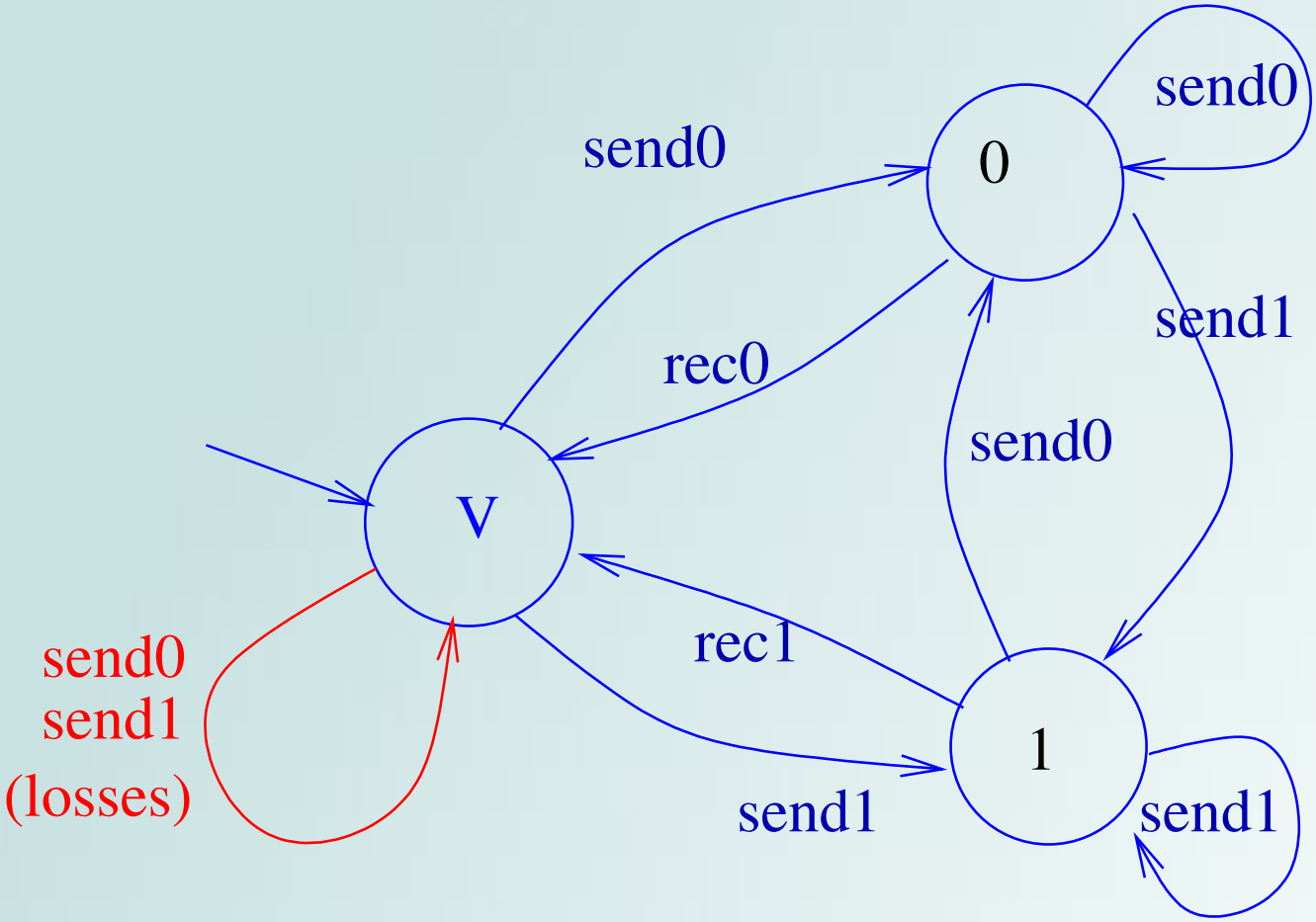


The receiver automaton



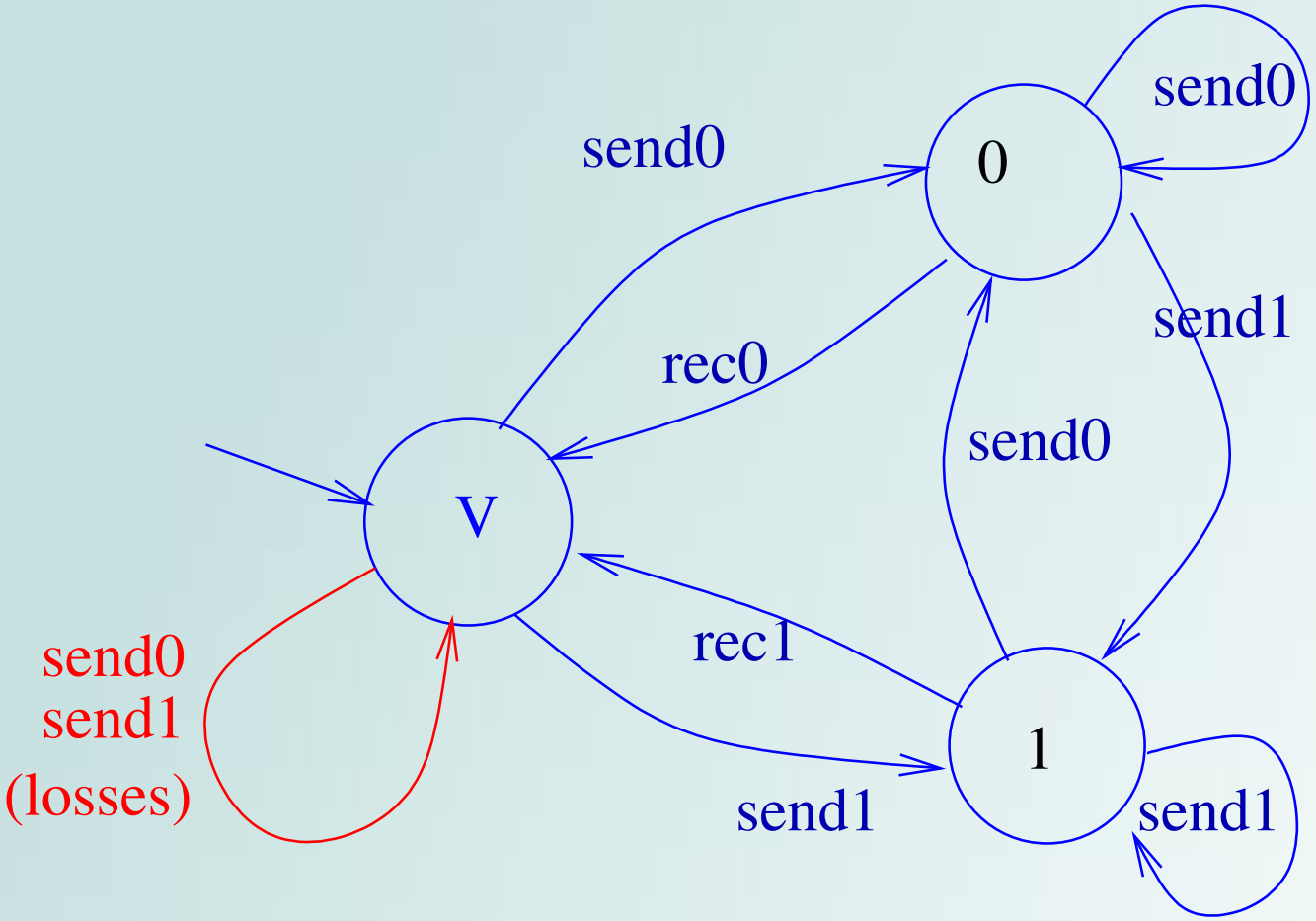
Link1 from emitter to receiver

sends are to L1, and receives are from L1



Link2 from receiver to emitter

sends are to L2, and receives are from L2

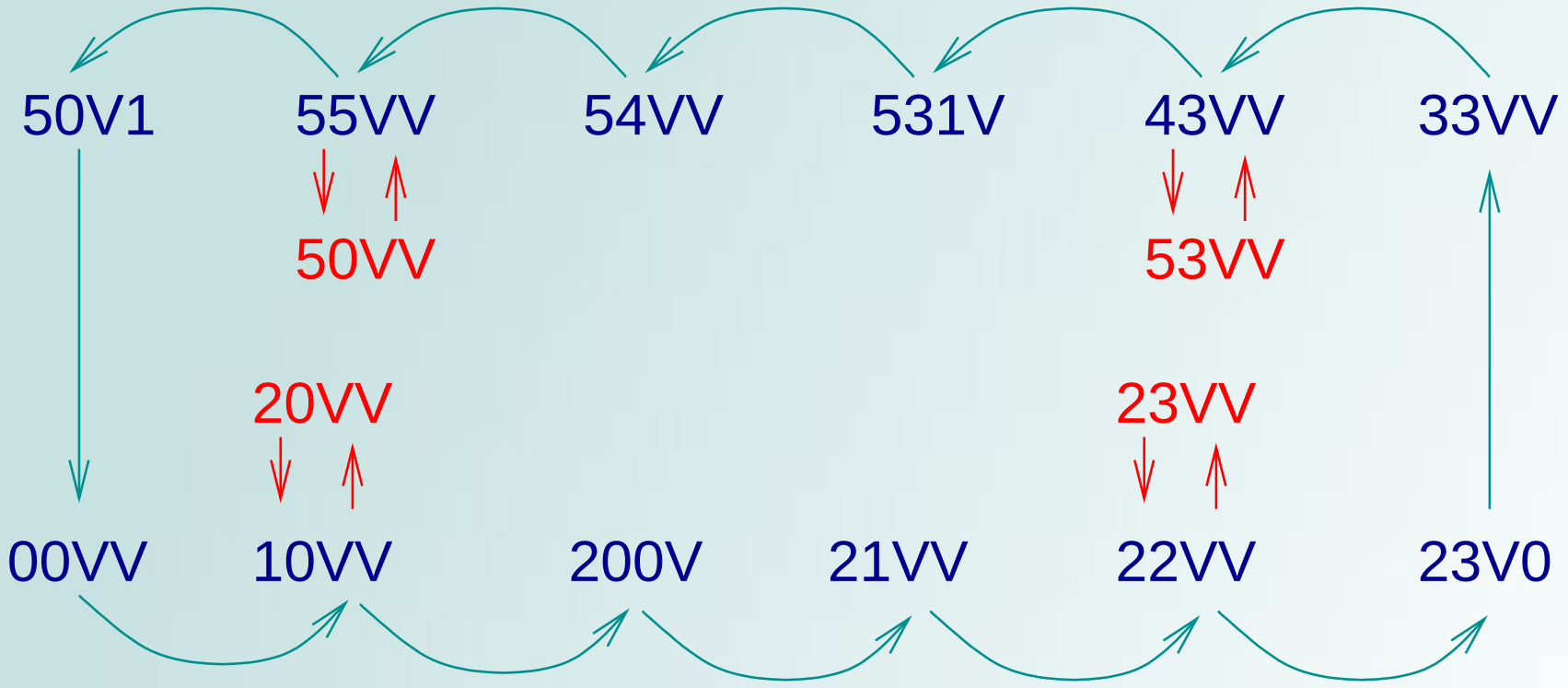


Building the product

The links behave as "memory": send=write, receive=read, except that the read operation destroys the element just read.
In the product `emitter * receiver * link1 * link2`, the transitions `send j on Li` (resp. `rec j from Li`) should be synchronized with the corresponding transitions `send j` in `link i` (resp. `rec j` in `Link i`).

The product of the 4 elements is built on the same principle as a product of parallel processes with a shared memory.

(Part of) The product



Exploring the States After a Message Loss

What about the transitions from states `50VV`, `53VV`, `23VV`, `20VV` ?

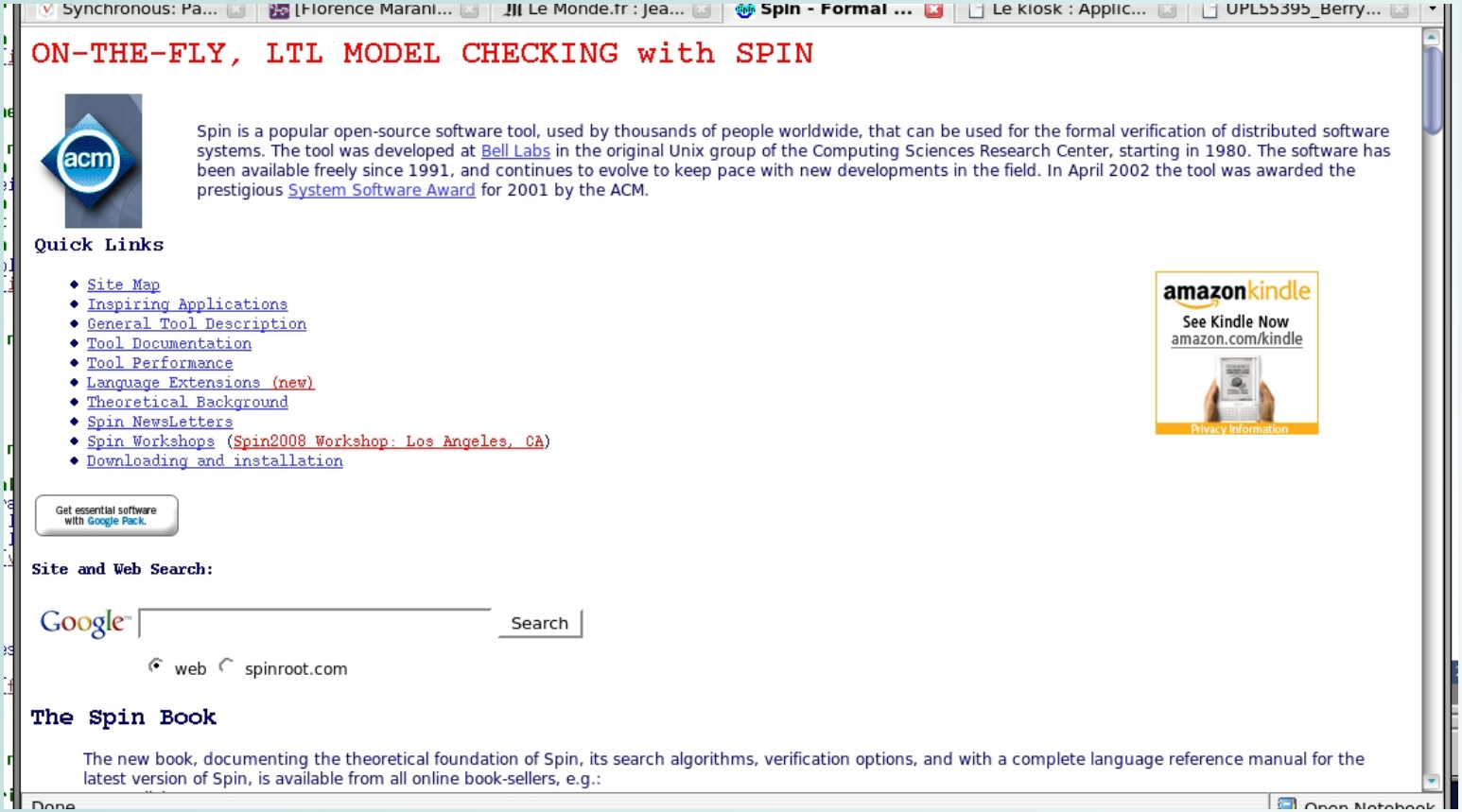
Reasoning on the Global Model
Faithfulness of the Model

- What does the synchronization between the emitter (or the receiver) and one link represent? Is it realistic?
- Could we model things with an explicit synchronization between the emitter and the receiver?

Reasoning on the Global Model
Deducing Properties

- Deadlocks: what about the global model if there are no (red) unlabeled transitions in the emitter and the receiver?
- Correct transmissions (safety part): do an occurrence of `x to send` and the following occurrence of `y received` correspond to the same (alternating) bit?
- Repeated losses and the notion of fairness: is it true that *after* `x to send`, *all possible paths lead to* `y received` *?*

SPIN



http://spinroot.com/spin/whatispin.html

http://compilers.cs.ucla.edu/spin08/ International SPIN Workshop on Model Checking of Software

SPIN version of the alternating-bit protocol

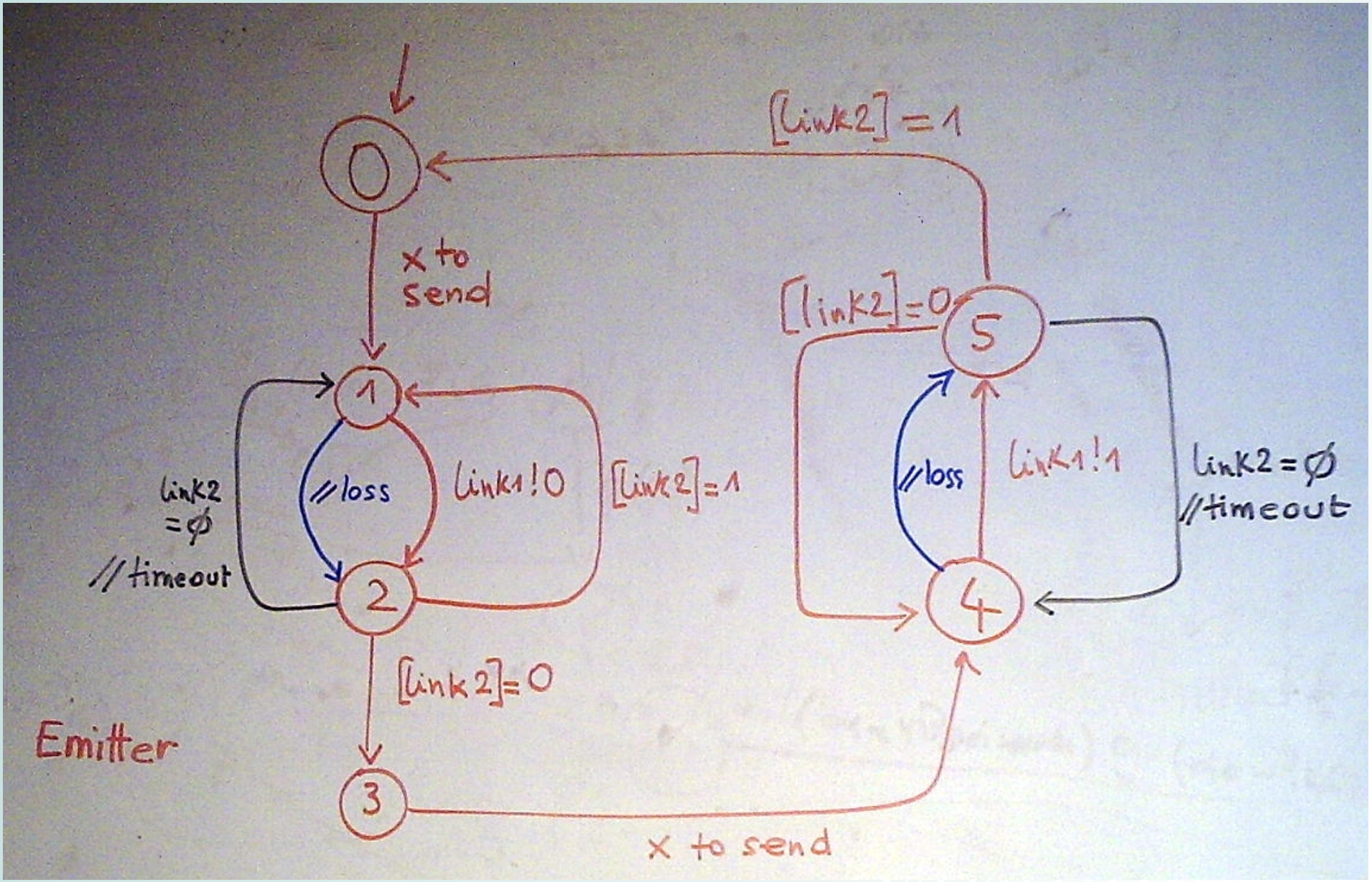
```
/* transmission channels, no loss */
chan link1 = [1] of {bool} ;
chan link2 = [1] of {bool} ;
```

Losses will be modeled in the automata of the receiver and the transmitter.

SPIN version: The Emitter

```
active proctype Emitter () {
   byte state = 0 ; byte x = 0 ;
   do
     :: atomic { state == 0 -> printf ("x to send %d\n",0) ; state = 1  ;}

     :: atomic { state == 1 -> printf("put 0 on link1\n") ; link1!0 ; state = 2 ; }
     :: atomic { state == 1 -> printf ("link1 loses\n") ; state = 2 ; }  /* perte */

     :: atomic { state == 2 && !link2?[x] -> printf ("timeout Emitter\n") ; state = 1 ; }  /* timeout */
     :: atomic { state == 2 && link2?[0] -> printf ("get 0 from link2\n") ; link2?x ; state = 3 ; }
     :: atomic { state == 2 && link2?[1] -> printf ("get 1 from link2\n") ;link2?x ; state = 1 ; }

     :: atomic { state == 3 -> printf ("x to send %d\n",1) ; state = 4  ;}

     :: atomic { state == 4 -> printf("put 1 on link1\n") ; link1!1 ; state = 5 ; }
     :: atomic { state == 4 -> printf ("link1 loses\n") ; state = 5 ; }  /* perte */

     :: atomic { state == 5 && !link2?[x] -> printf ("timeout Emitter\n") ; state = 4 ; }  /* timeout */
     :: atomic { state == 5 && link2?[0] -> link2?x ; printf ("get 0 from link2\n") ; state = 4 ; }
     :: atomic { state == 5 && link2?[1] -> link2?x ; printf ("get 1 from link2\n") ;state = 0 ; }
   od
}
```

The Emitter


0
1
2
3
4
5
[link2] = 1
x to send
[link2] = 0
link2 = ∅
//timeout
//loss
link1!0
[link2] = 1
//loss
link1!1
link2 = ∅
//timeout
[link2] = 0
x to send
Emitter

SPIN version: The receiver

```
active proctype Receiver () {
    byte state = 0 ; byte x     = 0 ;
    do
     :: atomic { state == 1 -> printf ("y received %d\n",0) ; state = 2  ;}

     :: atomic { state == 2 -> link2!0 ; printf("put 0 on link2\n") ; state = 3 ; }
     :: atomic { state == 2 -> printf ("link2 loses\n") ; state = 3 ; }  /* perte */

     :: atomic { state == 3 && !link1?[x] -> printf ("timeout Receiver\n") ; state = 2 ; }   /* timeout */
     :: atomic { state == 3 && link1?[0] -> link1?x ; printf("get 0 from link1\n") ; state = 2 ; }
     :: atomic { state == 3 && link1?[1] -> link1?x ; printf("get 1 from link1\n") ; state = 4  ; }

     :: atomic { state == 4 -> printf ("y received %d\n",1) ; state = 5  ;}

     :: atomic { state == 5 -> link2!1 ; printf("put 1 on link2\n") ; state = 0 ; }
     :: atomic { state == 5 -> printf ("link2 loses\n") ; state = 0 ; }  /* perte */

     :: atomic { state == 0 && !link1?[x] -> printf ("timeout Receiver\n") ; state = 5 ; }   /* timeout */
     :: atomic { state == 0 && link1?[0] -> link1?x  ; printf("get 0 from link1\n") ; state = 1 ; }
     :: atomic { state == 0 && link1?[1] -> link1?x  ;  printf("get 1 from link1\n") ; state = 5 ; }
   od
}
```

SPIN version: example runs

```
        timeout Receiver
        put 1 on link2
    x to send 0
    put 0 on link1
        get 0 from link1
        y received 0
    get 1 from link2
        put 0 on link2
        timeout Receiver
    put 0 on link1          -- re-emit same bit
    get 0 from link2        -- acknowledged

    x to send 1
        put 0 on link2
    put 1 on link1
        get 0 from link1
    get 0 from link2
        put 0 on link2
        get 1 from link1
    put 1 on link1          -- re-emit same bit
        y received 1
        link2 loses
....
```

Part III

Using Models

Outline

9 High Level Languages for concurrent systems

10 Test and Verification

9 High Level Languages for concurrent systems
- Concurrency
- Java
- Lustre
- Time-related primitives

10 Test and Verification

Categories of Languages

- Programming or Specification languages?

Categories of Languages

- Programming or Specification languages?
- Deterministic behaviours or not?

Categories of Languages

- Programming or Specification languages?
- Deterministic behaviours or not?
- Logical or physical parallelism (or both) ?

Categories of Languages

- Programming or Specification languages?
- Deterministic behaviours or not?
- Logical or physical parallelism (or both) ?
- Abstraction level (Ada or assembly-line languages?)

Categories of Languages

- Programming or Specification languages?
- Deterministic behaviours or not?
- Logical or physical parallelism (or both) ?
- Abstraction level (Ada or assembly-line languages?)
- Formal semantics, or not?

Categories of Languages

- Programming or Specification languages?
- Deterministic behaviours or not?
- Logical or physical parallelism (or both) ?
- Abstraction level (Ada or assembly-line languages?)
- Formal semantics, or not?
- Automatic code generation, or not?

Examples

Lustre/SCADE, Simulink, UML, SDL, Ada95, Real-time Java, VHDL or Verilog, SystemC, ...

- Lustre, VHDL, Verilog:

Examples

Lustre/SCADE, Simulink, UML, SDL, Ada95, Real-time Java, VHDL or Verilog, SystemC, ...

- Lustre, VHDL, Verilog: synchronous parallelism between blocks
- Java:

Examples

Lustre/SCADE, Simulink, UML, SDL, Ada95, Real-time Java, VHDL or Verilog, SystemC, ...

- Lustre, VHDL, Verilog: synchronous parallelism between blocks
- Java: asynchronous parallelism between threads + preemptive scheduler
- SystemC:

Examples

Lustre/SCADE, Simulink, UML, SDL, Ada95, Real-time Java, VHDL or Verilog, SystemC, ...

- Lustre, VHDL, Verilog: synchronous parallelism between blocks
- Java: asynchronous parallelism between threads + preemptive scheduler
- SystemC: asynchronous parallelism between threads + non-preemptive scheduler (see dedicated course, 3A)
- ...

9 High Level Languages for concurrent systems

- Concurrency
- Java
- Lustre
- Time-related primitives

Synchronous / Asynchronous Compositions

Important notions:

- The notion of parallelism+synchronization in the programming language (tasks, ...)
- The available support for parallelism at execution time (real parallelism in HW, simulated parallelism in SW+OS)

Two examples:

- Java
- Lustre

9 High Level Languages for concurrent systems

- Concurrency
- Java
- Lustre
- Time-related primitives

Main Ideas

- A thread is an object
- A shared resource is also an object, of a class `C`; the methods of class `C` may be declared `synchronized`, in which case a thread that is executing one of these methods for an object `O` of class `C` cannot be interrupted to give the CPU to another thread executing one of the synchronized methods on the *same* object `O`. Example:

```
class ObjetPartage {
    private int i ;
    synchronized  public void set (int ii) { i = ii ; }
    synchronized  public int  get ()       { return i ; }
}
```

`O.set` and `O.get` are atomic, for any `O`.

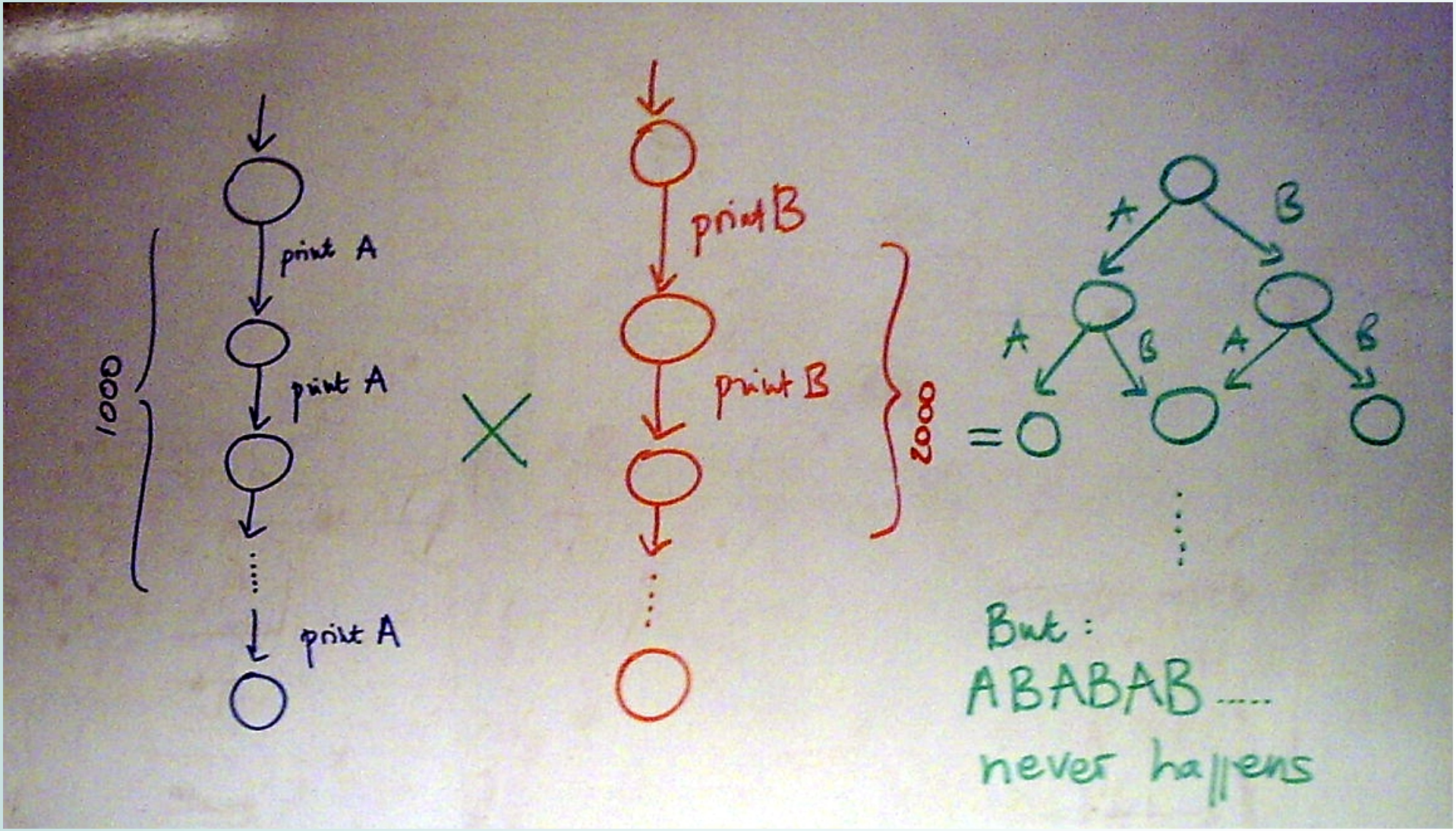
Example Code, no Synchronization

```
class BasicThread extends Thread {
    char c;
    int  l;
    BasicThread(char c, int l) { this.c = c; this.l = l;}
    public void run() {
        for(int i=0; i<l; i++) {
            System.out.print(c);
        }
    }
}
...
BasicThread bt  = new BasicThread('A', 1000);
BasicThread bt1 = new BasicThread('B', 2000);
bt.start();
bt1.start();
```

Example Run, and Comments

```
AAAA
```

The Underlying Formal Model



Comments

The atomicity of instructions is that of a
`put` operation (or even less)...
Hence the model has *states* between any two such instructions.

But the scheduler does not change threads as soon as possible. Only long sequences of the same letter do occur.

However, there is a need for synchronization.

Building an Example that Shows the need for Synchronization...

Two threads that access the same shared.
Writing to the shared object is quite long, hence the probability that the writing thread is interrupted while performing the write operation is not too small to be observed.

Example Code, Need for Synchronization (1)

```
class TestThreadsComm {
    public static void main (String [] args) {
        ObjetPartage comm = new ObjetPartage ();
        Thread t1 = new MonThread ("producteur",   false, comm);
        Thread t2 = new MonThread ("consommateur", true,  comm);

        t1.start (); t2.start ();

        while (true) {
            try {Thread.sleep(1000);} catch (InterruptedException e) { }
            System.out.print ("(main thread) ");
        }
    }
}
```

Example Code, Need for Synchronization (2)

```
class ObjetPartage {
    private int t[] = new int[10000];
    public void set (int ii) {
        for (int i = 0 ; i<10000 ;i++) {
            try {Thread.sleep(1);}
            catch (InterruptedException e) { }
            t[i] = ii ;
        }
    }
    public int  get ()       { return t[0] ; }
    synchronized public boolean obs ()       {
        for (int i = 0 ; i<10000 ;i++)
            if (t[i] != t[0]) return false ;
        return true ;
    }
}
```

Example Code, Synchronization (3)

```
class  MonThread extends Thread {
    int i ; String label ;
    ObjetPartage comm ;
    boolean consoProd ;  // true : cons, false : prod

    MonThread (String l, boolean b, ObjetPartage o) {
        super () ; label = l ; comm = o ; consoProd = b ;
    }
    public void run () {
        while (true) {
            try {Thread.sleep(100);} catch (InterruptedException e) { }
            System.out.print ("("+ label + " : ");
            if (consoProd) // consommateur
                System.out.print ("comm ? " + comm.obs());
            else {         // producteur
                int v = (comm.get()+1);
                System.out.print ("comm := " + v);
                comm.set (v) ;
            }
            System.out.print (")  ");
        }
    }
```

Example Run, and Comments

With the following "write" code in the shared object:

```
public void set (int ii) {
 for (int i = 0 ; i<10000 ;i++) t[i] = ii ;
}
```

```
(producteur : comm := 1)  (consommateur : comm ? true)  (producteur : comm := 2(consommateur : )  comm ? true)
(producteur : comm := 3)  (consommateur : comm ? true)  (producteur : comm := 4)  (consommateur : comm ? true)
(producteur : comm := 5)  (consommateur : comm ? true)  (producteur : comm := 6)  (consommateur : comm ? true)
(producteur : comm := 7)  (consommateur : comm ? true)  (producteur : comm := 8)  (consommateur : comm ? true)
(producteur : comm := 9)  (consommateur : comm ? true)  (main thread) (producteur : comm := 10)
(consommateur : comm ? true)  (producteur : comm := 11)  (consommateur : comm ? true)  (producteur : comm := 12
(consommateur : comm ? true)  (producteur : comm := 13)  (consommateur : comm ? true)  (producteur : comm := 14
(consommateur : comm ? true)  (producteur : comm := 15)  (consommateur : comm ? true)  (producteur : comm := 16
(consommateur : comm ? true)  (producteur : comm := 17)  (consommateur : comm ? true)  (producteur : comm := 18
(consommateur : comm ? true)  (producteur : comm := 19)  (consommateur : comm ? true)  (main thread)
(producteur : comm := 20)  (consommateur : comm ? true)  (producteur : comm := 21)  (consommateur : comm ? true
(producteur : comm := 22)  (consommateur : comm ? true)  (producteur : comm := 23)  (consommateur : comm ? true
(producteur : comm := 24)  (consommateur : comm ? true)  (producteur : comm := 25)  (consommateur : comm ? true
(producteur : comm := 26)  (consommateur : comm ? true)  (producteur : comm := 27)  (consommateur : comm ? true
(producteur : comm := 28)  (consommateur : comm ? true)  (main thread) (producteur : comm := 29)
(consommateur : comm ? true)  (producteur : comm := 30)  (consommateur : comm ? true)  (producteur :
```

Example Run, and Comments

With the following "write" code in the shared object:

```
public void set (int ii) {
   try {Thread.sleep(1);} catch (InterruptedException e) { }
   for (int i = 0 ; i<10000 ;i++) t[i] = ii ;
}
```

```
(producteur : comm := 1(consommateur : comm ? true)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false) (consommateur : comm ? false)
(main thread) (consommateur : comm ? false)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false)  (consommateur : comm ? false)
(consommateur : comm ? false)  (main thread)
```

1 ms is enough to observe the fact that the "write" method has been interrupted before completion.

Example Code, Synchronization

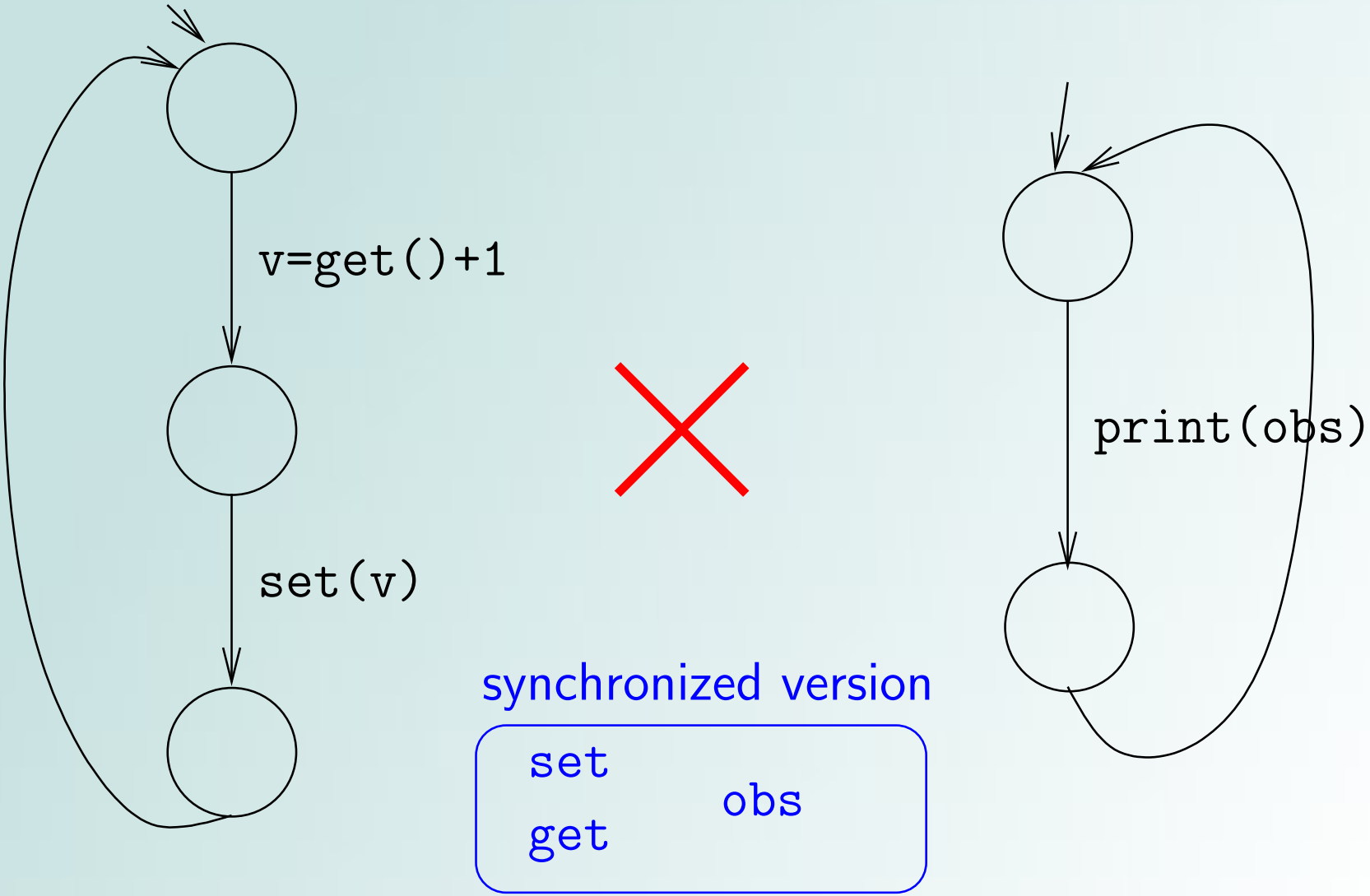
```
class ObjetPartage {
    private int t[] = new int[10000];
    synchronized public void set (int ii) {
        for (int i = 0 ; i<10000 ;i++) {
            try {Thread.sleep(1);}
            catch (InterruptedException e) { }
            t[i] = ii ;
        }
    }
    synchronized public int  get ()       { return t[0] ; }
    synchronized public boolean obs ()       {
        for (int i = 0 ; i<10000 ;i++) {
            if (t[i] != t[0]) {
                return false ;
            }
        }
        return true ;
    }
}
```

Example Run, and Comments

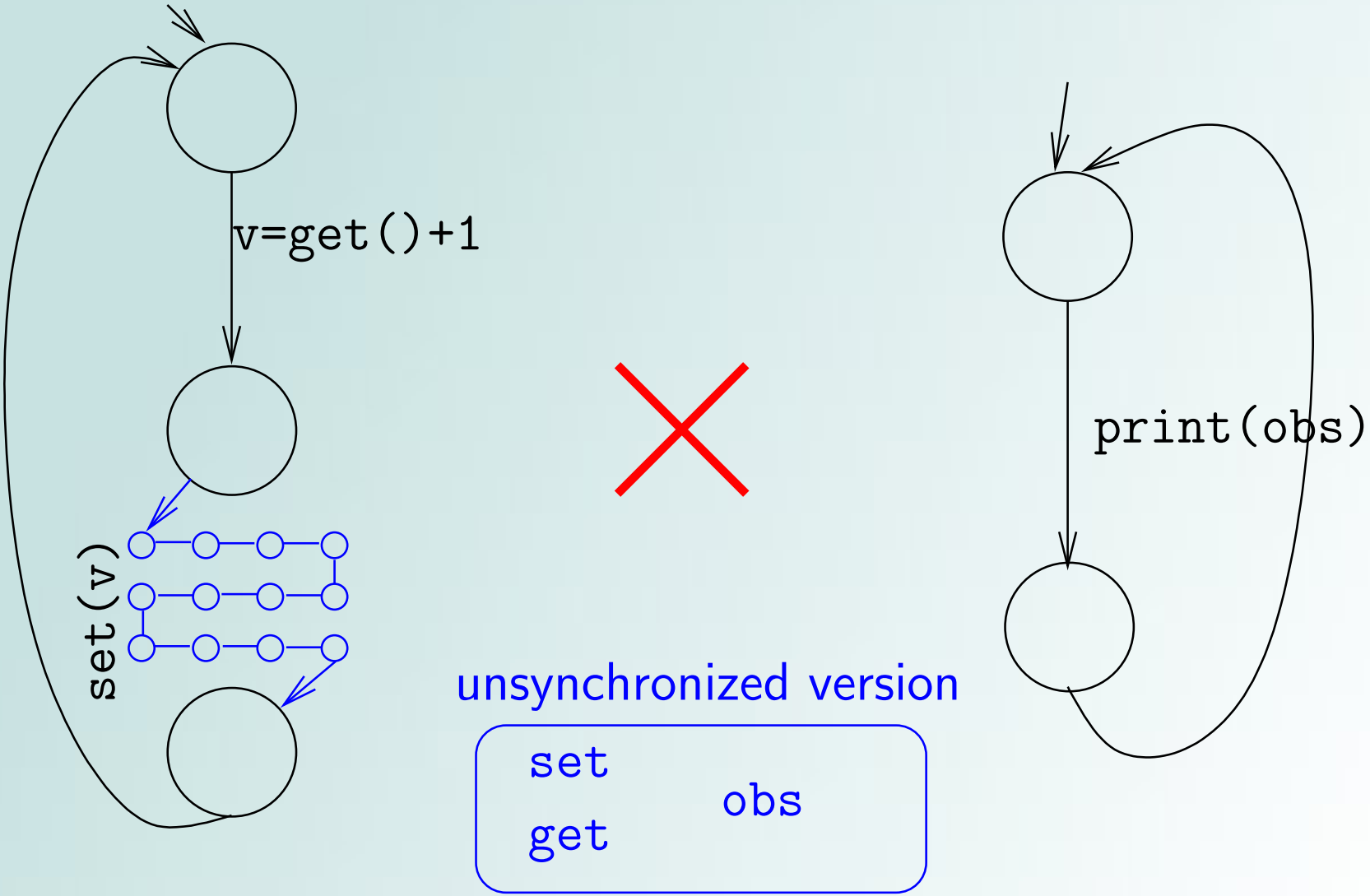
main now prints "()"

```
(producteur : comm := 1(consommateur : () () () () () () ()
() () () () () () () () () () () () () () () () () () ()
() () () () () () () () () () () () () () () () () () ()
() () () () () () () () () () () () )  comm ? true)
(producteur : comm := 2(consommateur : () () () () () () ()
() () () () () () () () () () () () () () () () () () ()
() () () () () () () () () () () () () () () () () () ()
() () () () () () () () () () () () () ) comm ? true)
(producteur : comm := 3(consommateur : () () () () () () ()
() () () () () () () () () () () () () () () () () () ()
() () () () () () () () () () () () () () () () () () ()
() () () () () () () )  comm ? true)  (producteur : comm := 4
(consommateur : () () () () () () () () () () () () () ()
() () () () () () () () () () () () () () (
```

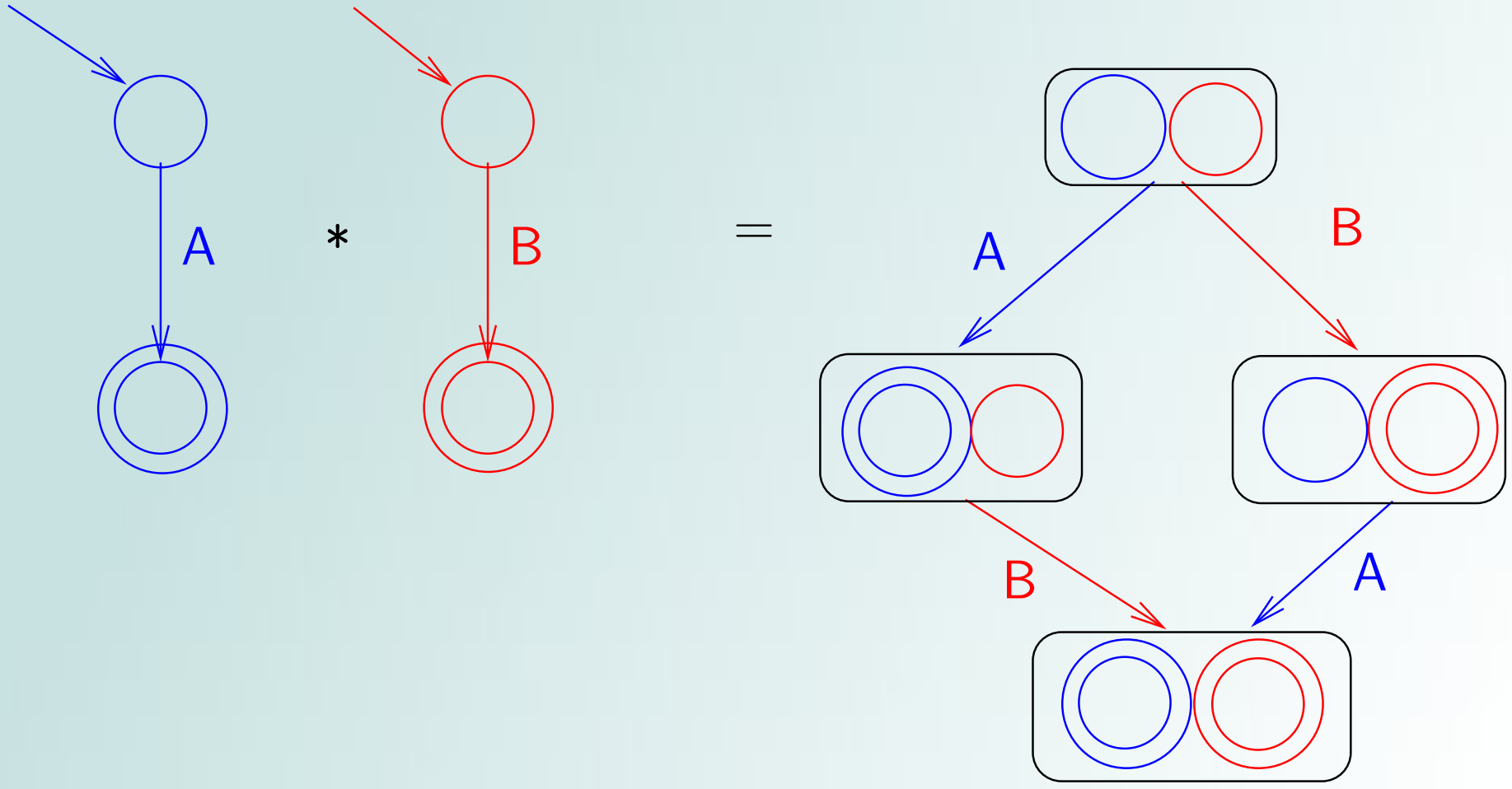
The Underlying Formal Model (1)



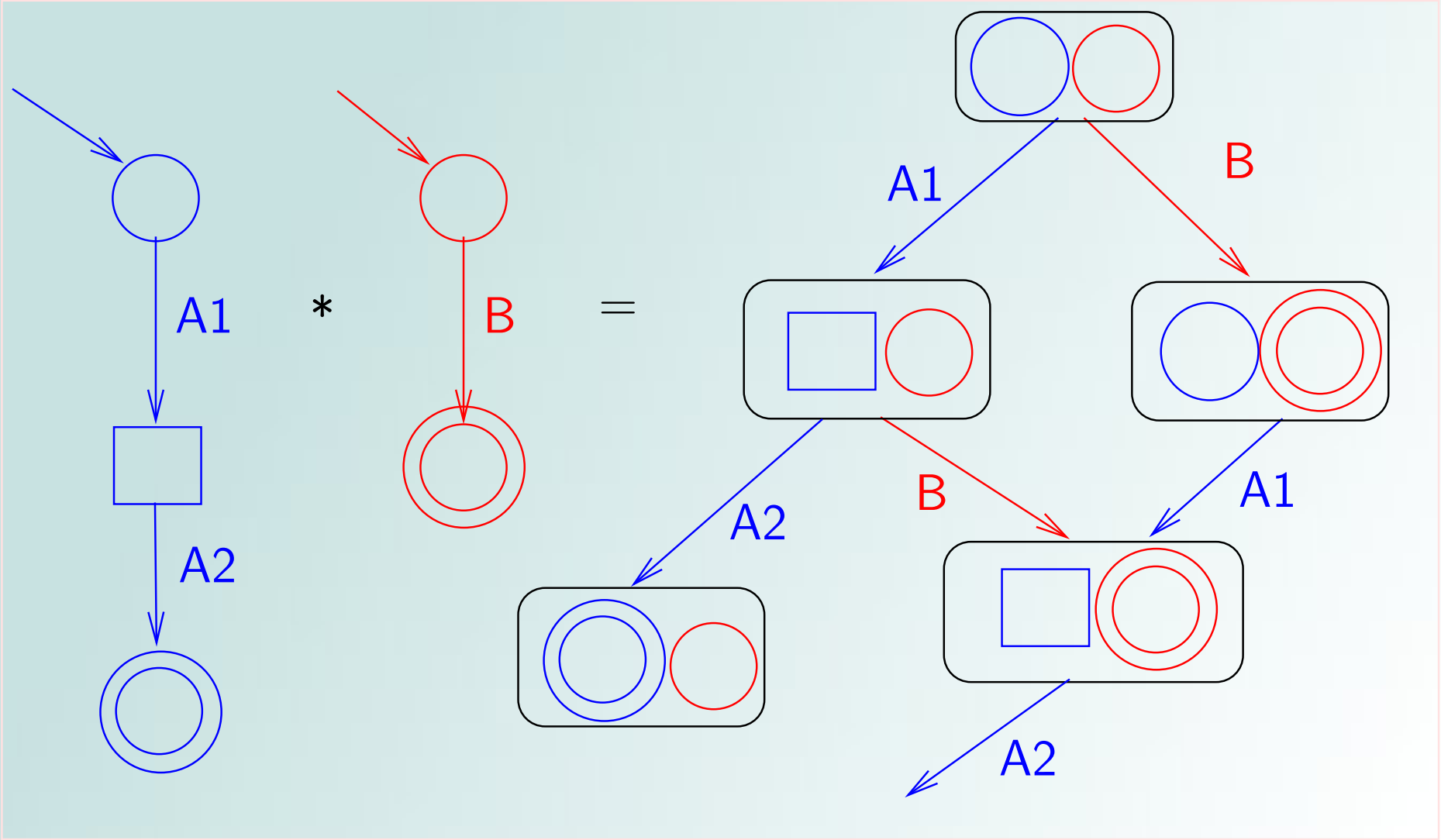
The Underlying Formal Model (2)



A note on “granularity” - example



A note on "granularity" - example



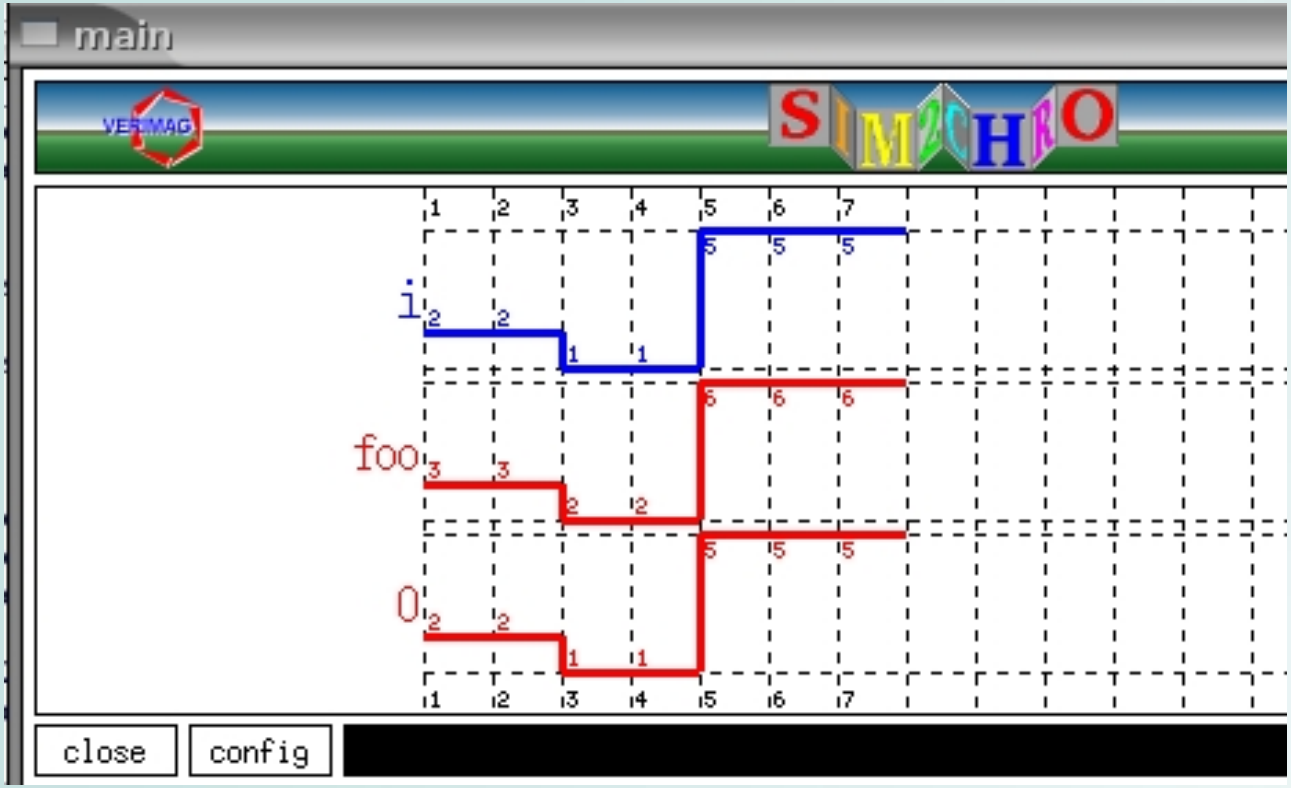
9 High Level Languages for concurrent systems

- Concurrency
- Java
- Lustre
- Time-related primitives

Example Parallel Program, no Synchronization

```
node Prod (i : int) returns (ii : int) ;
let
    ii = i+1;
tel.

node Cons (x : int) returns (O : int) ;
let
    O = x ;
tel.

node main (i : int) returns (foo : int ; O : int) ;
let
    foo = Prod (i) ;
    O   = Cons (i) ;
tel.
```

Example Run



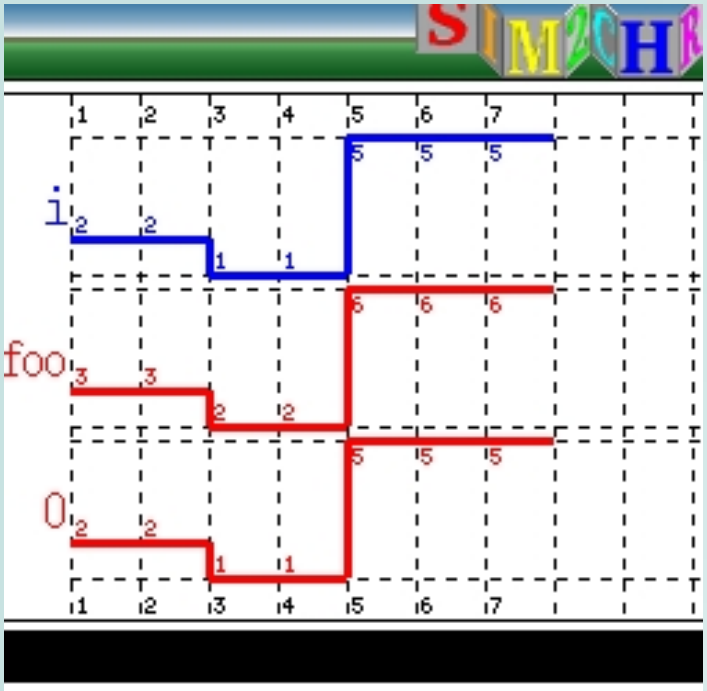
Structure of the Sequential Generated Code + a Typical main Program

```
void main_step(main_ctx* ctx){
//LOCAL VARIABLES
   _integer L1;
//CODE
   L1 = (ctx->_i + 1);
   main_O_foo(ctx->client_data, L1);
   main_O_O(ctx->client_data, ctx->_i);
}
main () {
    main_reset () ;
    while (1) {
       // read inputs into ctx->_i
       main_step (ctx) ;
    }
}
```

Example Parallel Program, Synchronization

```
node Prod (i : int) returns (ii : int) ;
let ii = i+1;  tel.

node Cons (x : int) returns (O : int) ;
let  O = x ; tel.

node main (i : int) returns (O : int) ;
var foo : int ;
let
    foo = Prod (i) ;
    O   = Cons (foo) ;
tel.
```

Example Run



Structure of the Sequential Generated Code

```
void main_step(main_ctx* ctx){
   _integer L1, L6, T5, L8, T7;

   if (ctx->M2) { L1 = 0; } else { L1 = ctx->M5; }
   main_O_O(ctx->client_data, L1);
   if (ctx->M2) { L6 = 0; } else { L6 = ctx->M7; }
   T5 = L6;
   L8 = (ctx->_i + 1);
   T7 = L8;
   ctx->M5 = T5;
   ctx->M5_nil = _false;
   ctx->M7 = T7;
   ctx->M7_nil = _false;
   ctx->M2 = ctx->M2 && !(_true);
}
```

Implementing a Lustre Program by a Synchronous Circuit

For the example above, trivial: `pre` is a flip-flop.
Just draw the Lustre program and you get the circuit.

The logical parallelism of the Lustre program is mapped one-for-one on the physical parallelism offered by the synchronous hardware.

Conclusion on Software Implementations: Dynamic/Static Schedulings (1)

Java: dynamic (= happens at runtime) scheduling between tasks, the scheduler may interrupt the tasks at any point (between two machine instructions).
The `synchronized` keyword is used to specify atomic portions of code.
Communication is via protected shared memory.

Lustre: static (= happens at compile-time) scheduling between tasks, there is no scheduler at runtime.
an instant is atomic
Communication is via signal exchanges, two parallel blocks may react "at the same time" (= in the same instant = in the same execution of the main loop)

Conclusion on Dynamic/Static Schedulings (2)

Static is safer (you know what is time)

Dynamic is more flexible (you may create and kill tasks at runtime)

9 High Level Languages for concurrent systems

- Concurrency
- Java
- Lustre
- Time-related primitives

Real-Time requirements

- Provide both periodic services and responses to asynchronous demands within strictly enforced deadlines. A real time system is said to be deterministic, meaning it can be relied upon to execute a given task in a predictable amount of time, every time it is called.
- Hard real-time systems never miss a delay
- Soft real-time systems tolerate some degree of delay and missed deadlines

See the following part of the course.

Example: Real-time Java

www.particle.kth.se/~lindsey/JavaCourse/Book/Part3/Chapter24/realTime.html

Main problems:

- The garbage collection is hardly predictable
- The scheduling has to be made more predictable also

Example: Lustre on the LEGO

There is a way of establishing a corresponsdance between the logical instant, and its physical duration, which can be made regular.

There is nothing like garbage collection, since there is no dynamic memory.

9 High Level Languages for concurrent systems

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

Model Extraction

Definition: extraction of a formal model (for instance a set of communicating parallel automata) from a piece of code or from a hardware description.

Examples: can be done for Java, SystemC, VHDL, Lustre, ...

Some of the languages have a formal semantics (e.g., Lustre) and this is simply a compiling technique. For other languages (Java, SystemC, ...), it is also a formalization of their semantics.

Modeling

definition: design of a formal model (for instance a set of communicating parallel automata) to represent the behaviour of a complex system.

Example: the model of the alternating bit protocol contains a model of the links, which is not extracted from any operational description of these links.

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

Definitions: test

For hardware : testing is needed for the physical device produced from a logical (functional) description.

For software and hardware : functional testing makes sense.
Functional testing =
— find "relevant" input sequences
(among the infinity of possible input sequences)
— run the system for these input sequences
— observe the "result" and decide whether the test passes.

— Can only be partial
— Needs a notion of coverage

Definitions: Verification

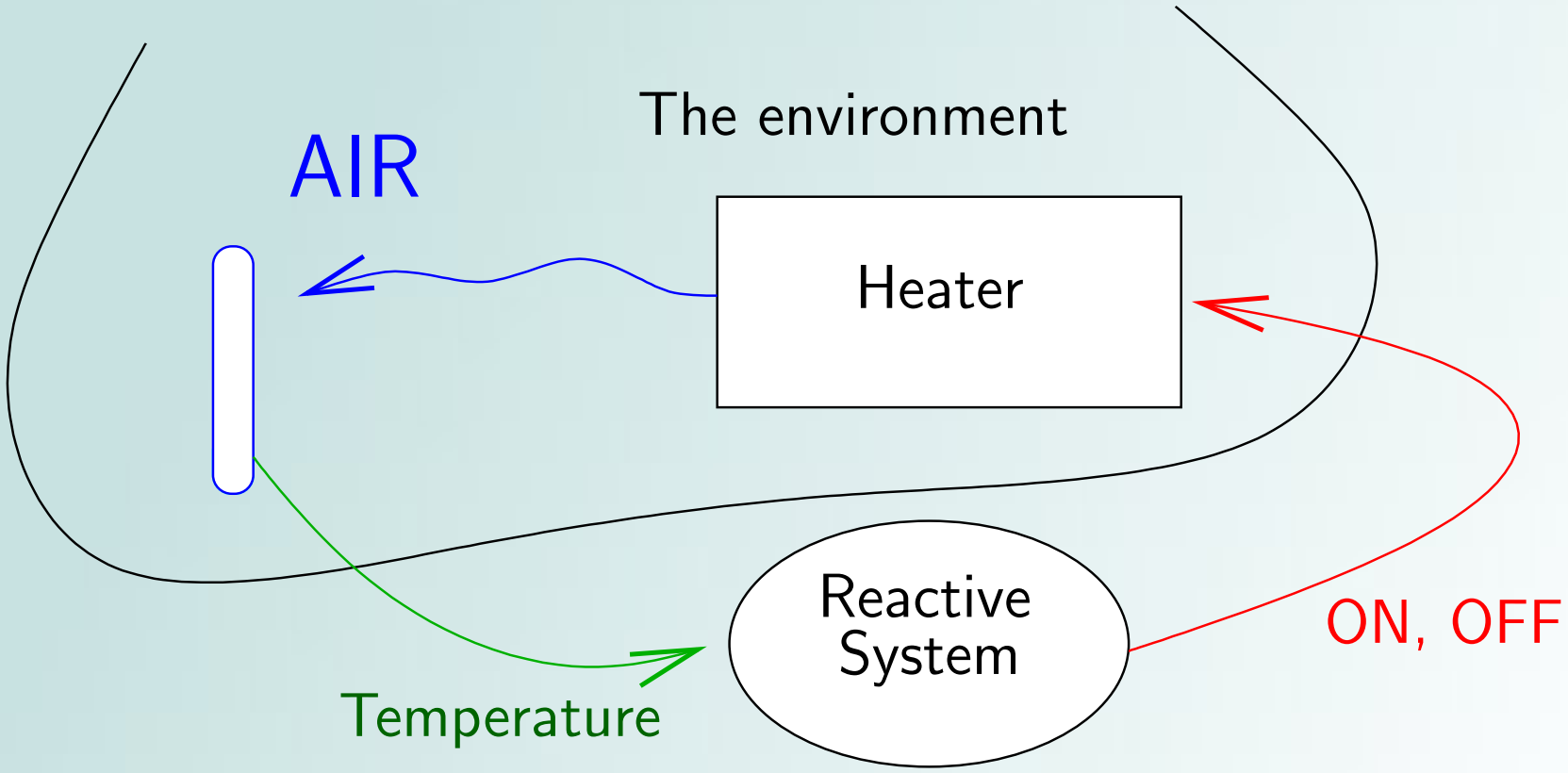
For software and hardware : say something about the behaviour of the system for all possible input sequences.

Since the set of input sequences is infinite, only approximate verification methods exist.

Interesting ones : conservative methods.
(reject correct programs, but do not accept incorrect ones)

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

The example: a heater controller



Testing a heater controller

The controller C should maintain the temperature u between 17° and 20° by turning a heater ON and OFF.
When the heater is running, the temperature u increases with $0.2 \leq du/dt \leq 0.5$; otherwise it decreases with $-0.3 \leq du/dt \leq -0.1$.

Initially, the heater is turned off, and $18^\circ \leq u \leq 19^\circ$.

Model of the environment

```
node Assumption(on, off: bool; u: real) returns (relevant: bool);
var dudt: real; heating: bool;
let
   relevant = (u>=18 and u<=19) ->
              if heating then (dudt>=0.2 and dudt<=0.5)
              else (dudt>=–0.3 and dudt<=–0.1);
   dudt = (u – pre(u));
   heating = false ->
              if pre(on) then true
                 else if pre(off) then false
                 else pre(heating);
tel
```

Oracle (the property to be checked)

```
node Property(on, off: bool; u: real) returns (ok: bool);
let
   ok = (u>=17 and u<=20);
tel
```

Generation of test sequences

We have to generate test sequences (i.e., sequences of values for the temperature input) realistic w.r.t. the physical environment of the system.

How to do that?

- Build some kind of non-deterministic automaton from the description of the environment
- Run the program (may be a black box) ...
- ... and the environment in turn. Running the environment means going through the automaton and solving constraints.

Example

- The environment plays first: choose a value according to: (u>=18 and u<=19), e.g., 18
- The program plays: given the input 18, it outputs some commands on, off
- *The oracle observes (u=18, on, off) and checks (u>=17 and u<=20)*
- The environment plays: given the input on, off, it outputs some new value for the temperature. This is done by solving a numerical constraint.
- and so on...

Constraint to solve - 1

First, determine whether `heating` is true, depending on the values just emitted by the program for `on, off`.

```
node Assumption(on, off: bool; u: real) returns (relevant: bool);
var dudt: real; heating: bool;
let
   relevant = (u>=18 and u<=19) ->
              if heating then (dudt>=0.2 and dudt<=0.5)
              else (dudt>=–0.3 and dudt<=–0.1);
   dudt = (u – pre(u));
   heating = false ->
              if pre(on) then true
                 else if pre(off) then false
                 else pre(heating);
tel
```

Constraint to solve - 2

Then, select the appropriate branch of the IF:

```
node Assumption(on, off: bool; u: real) returns (relevant: bool);
var dudt: real; heating: bool;
let
   relevant = (u>=18 and u<=19) ->
              if heating then (dudt>=0.2 and dudt<=0.5)
              else (dudt>=–0.3 and dudt<=–0.1);
   dudt = (u – pre(u)); – knowing that pre(u) = 18
... tel
```

and extract the constraint, e.g. (dudt>=0.2 and dudt<=0.5),
or (u - 18 >=0.2 and u - 18 <=0.5)

Applications

- Testing control systems (ABS, ...)
- Testing ad hoc sensor networks
- ...

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

The example
feature interactions (from ICFI'05)

- Feature interactions in software systems: networks, appliances, components, security systems, medical systems, transport systems, instant messaging, and mobile code.
- Feature interactions in emerging telecom architectures: Internet, Internet telephony (e.g. SIP, H.323, Megaco), agent architectures, policy-based services and architectures, Web services, 3G, Parlay, .NET, GRID networks, active networks.
- Current industrial practice and experience...
- ...

The example
feature interactions (from ICFI'05)

- ...
- Mechanisms for interaction detection and resolution : off-line methods, including service modelling, formal validation, testing techniques, software tool support, scalability and efficiency, filtering methods. on-line management solutions, runtime mechanisms, signalling capabilities, inter-working of domains and platforms.
- User-centric view on the interaction problem service personalization, usage scenarios, behaviour modelling, user-specified features, statistical usage profiles, service reliability aspects, presence/location/contextual services.

Easy-to-understand cases from www.research.att.com/ pamela/faq.html

- Bob has Call Forwarding, and is forwarding all calls to Carol. Carol has Do Not Disturb enabled. Alice calls Bob, the call is forwarded to Carol, and Carol's phone rings, because Do Not Disturb is not applied to a forwarded call.

Easy-to-understand cases from www.research.att.com/ pamela/faq.html

- Bob has Call Forwarding, and is forwarding all calls to Carol. Alice calls Bob, the feature forwards the call to Carol, and also changes the source address of the call to Bob's address (on the grounds that Bob is responsible for this leg of the call).
  One scenario: Carol has Call Blocking, and is blocking all calls from Alice. This call is not blocked, however, because it appears to be from Bob.
  Another scenario: Carol misses the call, but later uses a Call Return feature to return it. The feature places a call to Bob's address, which is forwarded back to Carol.

What to do with feature interactions?

Testing :
— obtain a formal model of the possible behaviours :
one feature = one automaton ; interactions = product

— find relevant scenarii
— run the scenarii on the model, observe the behaviour

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

Safety and liveness properties

Definition :
— a safety property expresses that something wrong will never happen.
— a liveness property expresses that something good will eventually happen.

Examples :

- Mutual exclusion is a safety property
- *"if the message is sent, then it will eventually be received"* is a liveness property

Observers for safety properties

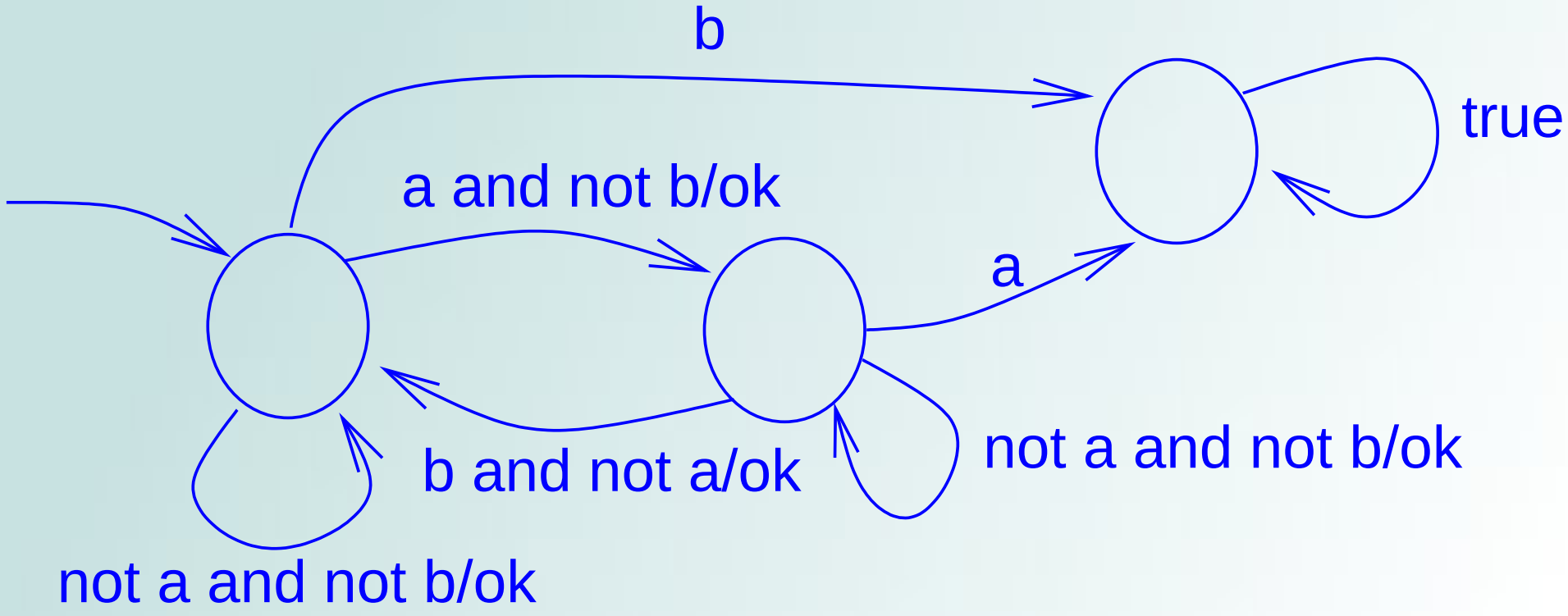
- In the synchronous world : very easy. Just add a component that reads the inputs and outputs of the program under observation, and has no effect on it.
- In the asynchronous world : not so easy.

Synchronous Observers: an example

How to specify that the Boolean outputs a and b of the program alternate?

Synchronous Observers: an example

How to specify that the Boolean outputs *a* and *b* of the program alternate?



Safety = state reachability

Independently of the way the properties are expressed, a safety property is true on a system if and only if a given set of (error) states is unreachable.

Safety properties and abstractions

A safety property ϕ is true $=$ a given set X of states is unreachable from the initial states.

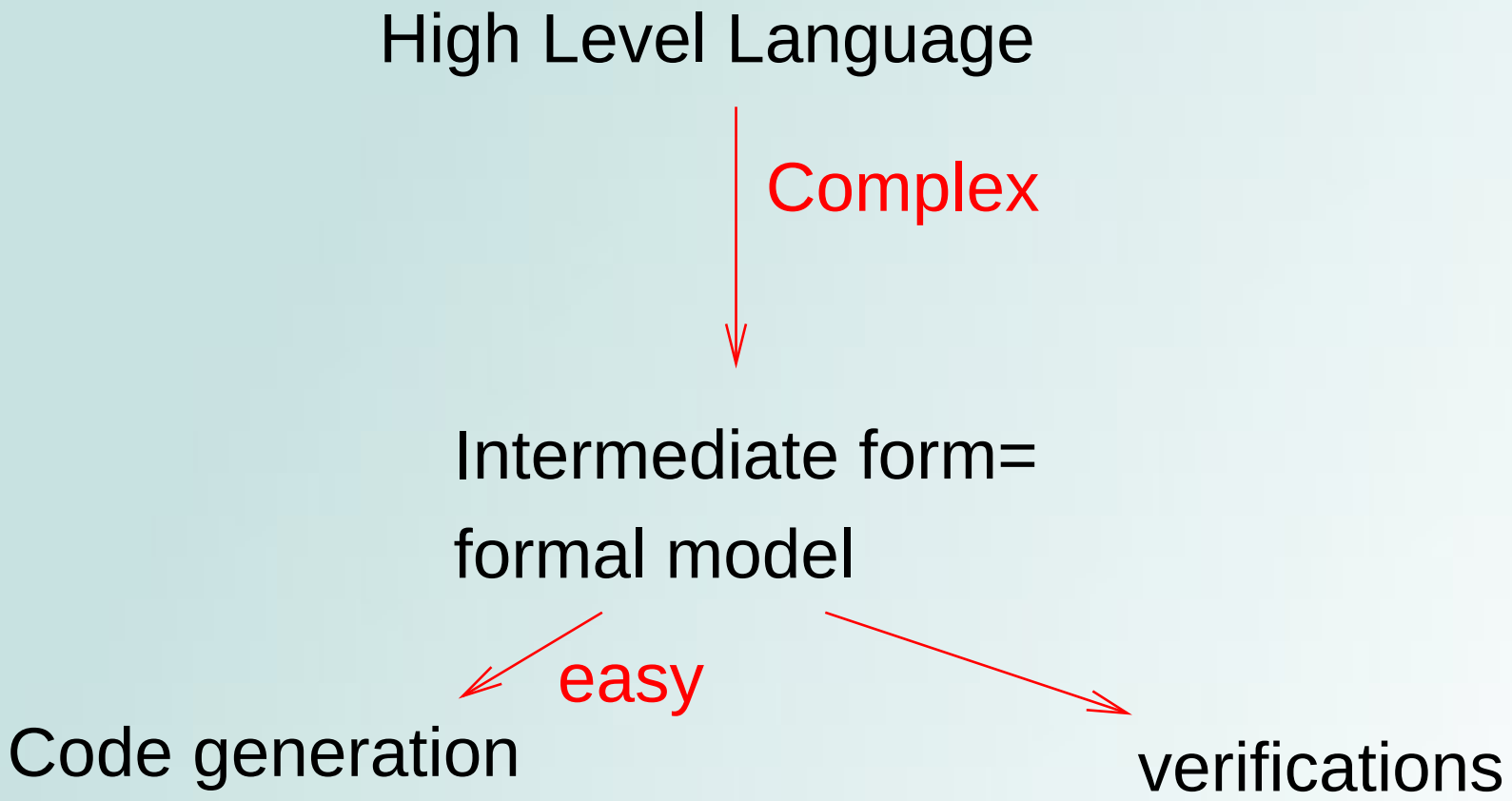
Abstracting an automaton : considering a super-set of states and transitions, hence believe that a set X is reachable, while it is not in the possible executions.

Abstraction means we may believe a safety property is false, while it is true. But we may never believe a safety property is true, while it is false.

conservative abstractions may reject correct programs, but will never accept incorrect programs.

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

The WYPIWYE principle: What You Prove Is What You Execute

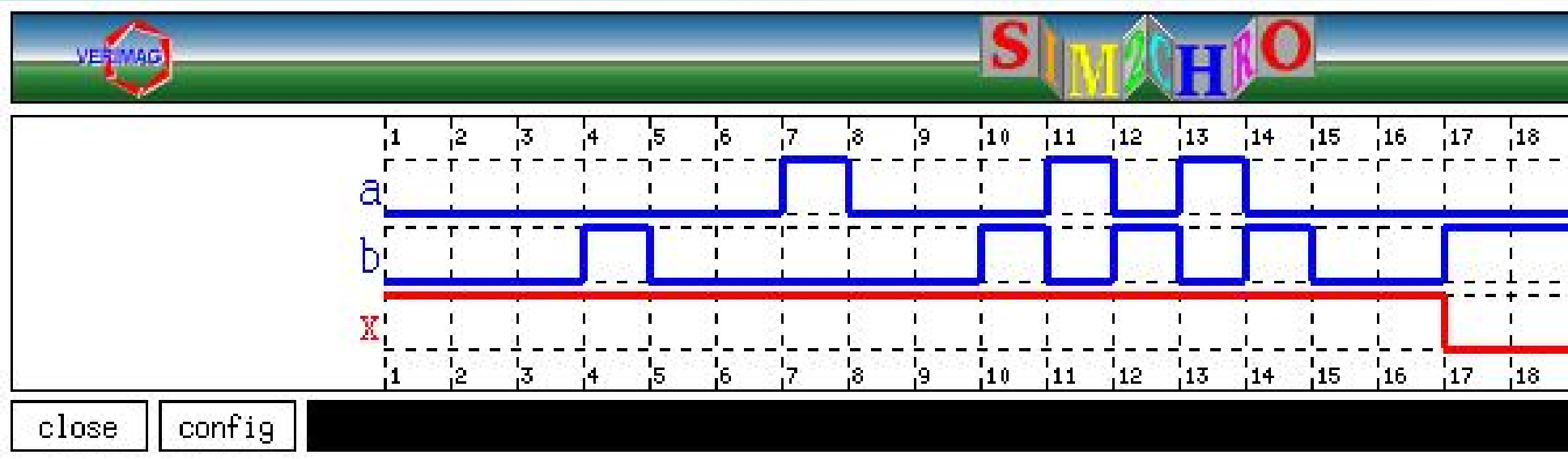


The example: two versions of "alternate"

The informal specification :
A system with two Boolean inputs a and b, and one Boolean output OK, expressing that the inputs alternate, starting with b.

It is possible to have $\neg a \wedge \neg b$ from time to time, but not to have two successive a's, or two successive b's.

Example of expected traces



Example program, version 1

```
node version1 (a, b : bool) returns (x : bool) ;
var
    lastab : bool ;
    -- true if last seen (strict) is a, false otherwise
let
    lastab = true -> pre (if a then true
                          else if b then false
                          else lastab) ;
    x = (not a) -> pre x and
                   not (a and lastab)  and
                   not (b and not lastab) ;
tel.
```

Example program, version 2

```
node version2 (a, b : bool) returns (x : bool) ;
var
    waita, waitb, error : bool ;
let
    waita = false -> pre (waita and (not a and not b) or
                          waitb and b) ;
    waitb = true ->  pre (waitb and (not b and not b) or
                          waita and a) ;
    error = false -> pre (waita and b   or
                          waitb and a   or
                          error) ;
    x = not  (error   or  waita and b  or  waitb and a) ;
tel.
```

Expressing that they behave the same

```
node verif (a, b : bool) returns (ok : bool) ;
var
    x1, x2 : bool ;

let
    x1 = version1 (a, b) ;
    x2 = version2 (a, b) ;
    ok = (x1 = x2) ;
tel.
```

ok should always be true, for all possible sequences of a and b valuations.

Using a model-checker : Lesar (P. Raymond)

```
etale(288) lesar verif.lus verif -diag
--Pollux Version 2.2

TRUE PROPERTY
```

Version 1'

```
node version1 (a, b : bool) returns (x : bool) ;
var
    lastab : bool ;
    -- true if last seen (strict) is a, false otherwise
let
    lastab = FALSE -> pre (if a then true
                           else if b then false
                           else lastab) ;
    x = (not a) -> pre x and
                   not (a and lastab)  and
                   not (b and not lastab) ;
tel.
```

Running the model-checker again...

```
etale(289) lesar verif.lus verif -diag
--Pollux Version 2.2

DIAGNOSIS:

--- TRANSITION 1 ---
not a and not b
--- TRANSITION 2 ---
a and not b or
not a and b
FALSE PROPERTY
```

10 Test and Verification
- Modeling versus Model Extraction
- Test and Verification: Definitions
- Test : a complete example for reactive systems
- Test : an example from telecom systems
- Expressing properties, and the notion of an "observer"
- Verification : an example in the Lustre toolbox
- Verification : beyond Boolean programs

Lesar for Boolean programs: exact verification

In the previous example, Lesar builds the graph of all possible STATES. It is finite.

It "compiles" the `verif` node into a Moore machine, where `not ok` is the ERROR state.

Then it searches the automaton, looking for a path from the initial state to the ERROR state.

If it exists : FALSE property, and diagnosis
If it does not exist : TRUE property.

Lesar (and other model-checkers) for non-Boolean programs

The problem :

A program and its property (or two versions of the same program) manipulating integers (or real numbers, or even more complex data structures).

Example program, v1 and v2

```
node nonboolv1 (a, b : int) returns (x : int) ;
let
     x = 0 -> a + pre (b+1) ;
tel.

node nonboolv2 (a, b : int) returns (x : int) ;
let
     x = 0 -> a + pre(b+1) ;
tel.
```

Running the model-checker

```
etale(313) lesar nonbool-verif.lus nonboolverif -diag

DIAGNOSIS:
LOCAL VARIABLE C0 IS
(V5_x1)=(V5_x1)

--- TRANSITION 1 ---
not C0
FALSE PROPERTY
```

Arithmetic decision : `not ((V5_x1)=(V5_x1))` is not satisfiable in the set of integers.

Running the model-checker with a bit of arithmetic decision

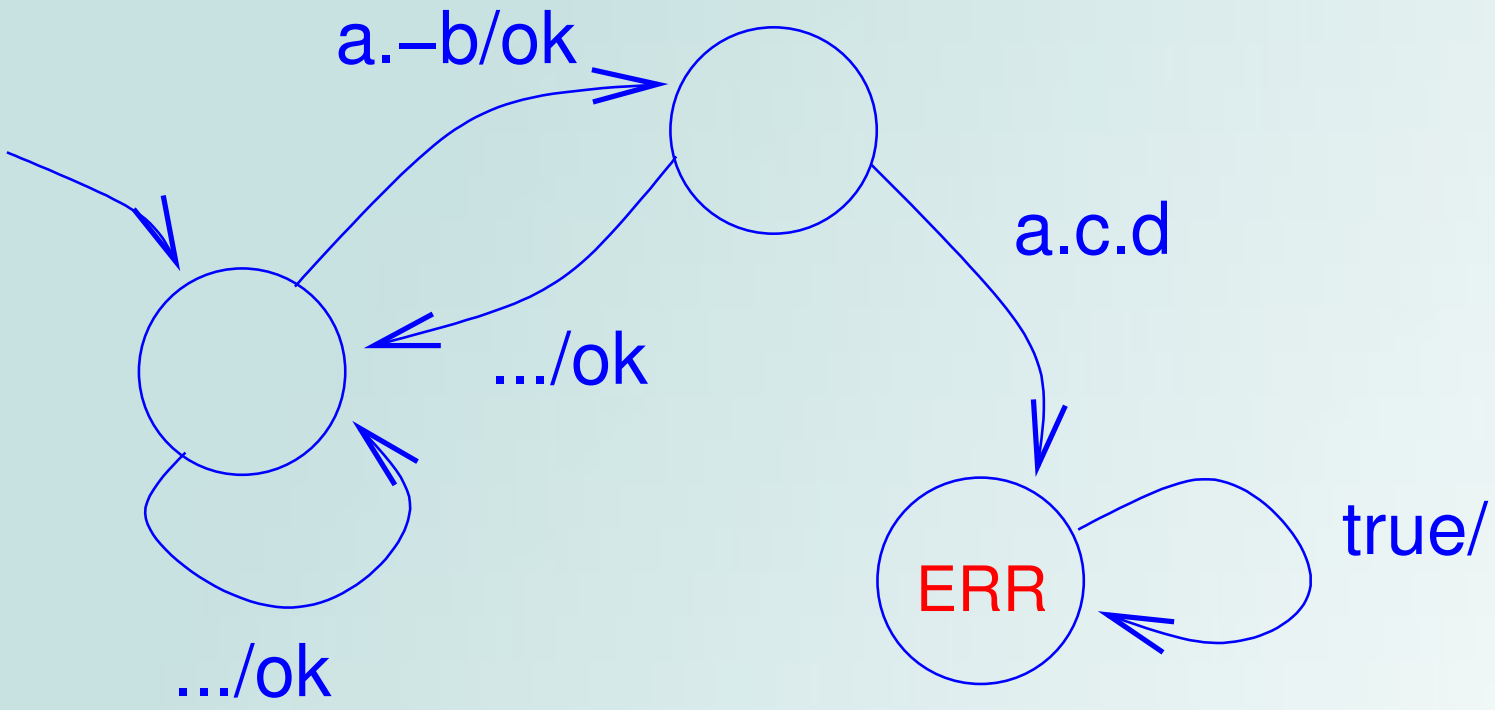
```
(314) lesar nonbool-verif.lus nonboolverif -poly -diag
--Pollux Version 2.2

TRUE PROPERTY
```

Summary on models, their expressivity, and verification methods

Types of models, on which reachability of states is examined:

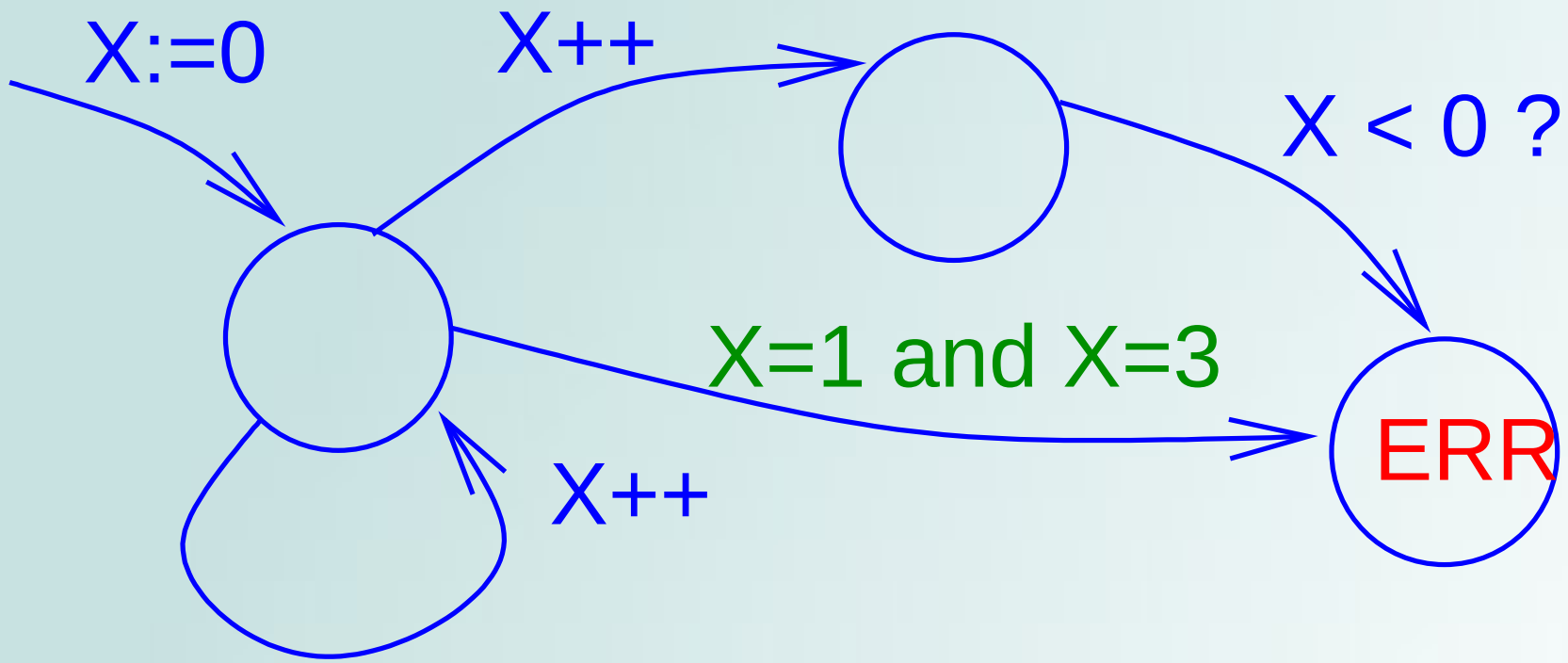
- Pure Boolean programs and models
- Programs (or systems) whose set of states is finite and small (via expansion of variables : small enumerated types, list of blocked processes in a semaphore, ...)
- Programs that manipulate BIG integers, or unbounded data structures

Pure Boolean Models and reachability



Is the error state reachable from the initial one ?
Needs only a bit of Boolean calculus.

General automata



Is the error state reachable from the initial one ?
complex question.

Local arithmetic decisions on automata

```
lesar -poly ...
```

For the transition labeled by `X=1 and X=3`
It's easy to see that the condition is locally (i.e., independently of the structure of the automaton) unsatisfiable.

Contextual arithmetic decisions on automata

For the transition labeled by `X < 0 ?`

This transition cannot be taken because of the possible histories that lead to its source state :

`X := 0` followed by a non-empty sequence of `X++` cannot lead to `X < 0`.

Very hard to compute with an automatic tool...

Fundamental result

The reachability of a pair `(control state, valuation of the variables)` from the initial state of a general interpreted automaton is :

undecidable

i.e., there exist no algorithm that can answer the question in every possible case.

Abstractions, and approximate results

- Forget about the labels !
  `lesar` on non Boolean programs

Abstractions, and approximate results

- Forget about the labels !
  `lesar` on non Boolean programs
- Look at local satisfiability
  `lesar -poly` on non Boolean programs

Abstractions, and approximate results

- Forget about the labels !
  `lesar` on non Boolean programs
- Look at local satisfiability
  `lesar -poly` on non Boolean programs
- Abstract the sets of possible values of the variables
  Abstract-interpretation tools (Polyspace.com, absint.com, the example of Ariane 5)

Abstractions, and approximate results

- Forget about the labels !
  `lesar` on non Boolean programs
- Look at local satisfiability
  `lesar -poly` on non Boolean programs
- Abstract the sets of possible values of the variables
  Abstract-interpretation tools (Polyspace.com, absint.com, the example of Ariane 5)
- active research in the field of : *automatic verification methods for complex systems*

Part IV

Real-Time Programming

Outline