October 22, 2014

- Wrap-up on Formal Models, Time, and the HW/SW Interface, with exercices
- If time allows: introduction to WCET
  Two problems: analysis of the control flow + HW modeling
- Before Nov 12: read the survey paper on WCET

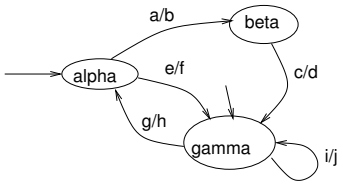
Simple Mealy machines

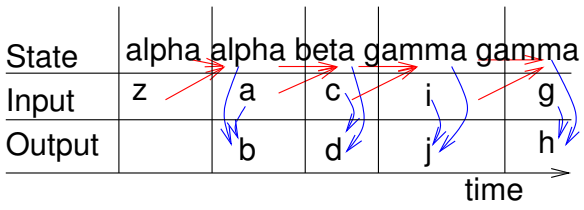
$M = (Q, Q_0 \subseteq Q, T \subseteq Q \times I \times O \times Q)$ where

- I (resp. O) is the set of inputs (resp. outputs)
- Q is the set of states
- Q_0 is the set of initial states
- T is the set of transitions, labeled by tuples (input,output)

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

Trace Semantics





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

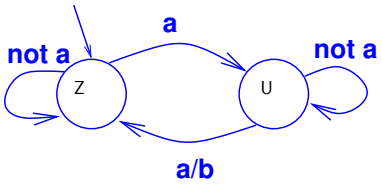
Same definition, but with $I = 2^{BI}$ and $O = 2^{BO}$.

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

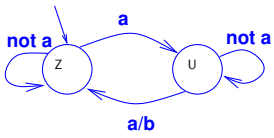
On the "2 lights" example: a.-b/LightA_on

(read: if a is pressed and not b, then switch lightA on)

Example Mealy machine : one 'b' every two 'a's



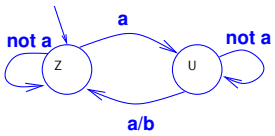
Encoding into Boolean Equations



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-1))
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



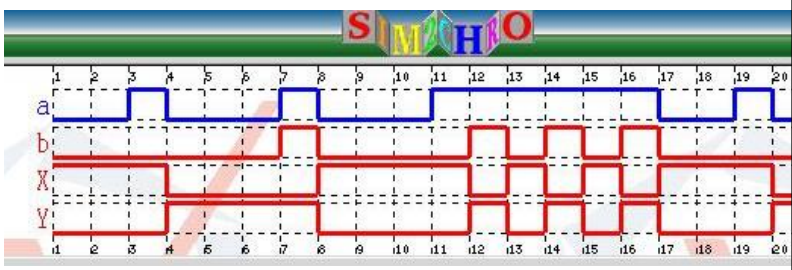

```
-- Lustre program
node Bit
  (a : bool) returns
  (b : bool ; Z, U : bool ) ;
let
 Z = true ->
       pre (Z and not a  or
            U and a) ;

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

 b = U and a ;
tel.
```


Lustre Restrictions vs Recurrence Equations

What's the essential difference between:
`z(0) = true ;`
`z(n) = (z(n-1) and not a(n-1)) or (u(n-1) and a(n-1))`
and
`Z = true -> pre (Z and not a or U and a) ;` ?

In Lustre: no explicit reference to the index `n`. Impossible to write something like `a(n-i)`, `i` being an (not statically known) input. This is how Lustre forbids non statically-bounded memory in programs.

Encoding into Boolean Equations: example behavior



Synchronous Product of Boolean Mealy Machines (without communication)

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

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

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

Lustre, Mealy machines, Dataflow Connections and Products (1. No Communication)

Mealy1 (i1)(o1) → Lustre Node
N1 (i1:...) returns (o1: ...)

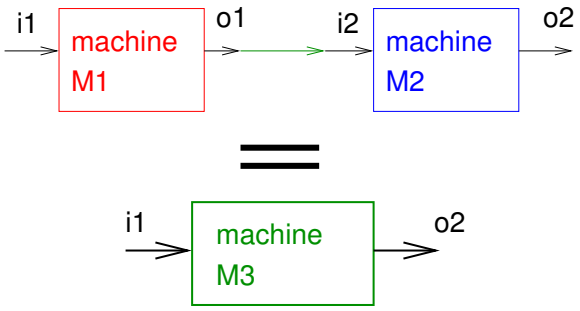
×

Mealy 2 (i2)(o2) → Lustre Node
N2 (i2:...) returns (o2: ...)

||

Mealy3 (i1,i2)(o2,o2) → Lustre Node
N(i1,i2:...) returns (o1,o2:...)
let
  o1 = N1(i1);
  o2 = N2 (i2);
tel.

Communication: idea



The output of a machine can be connected to the input of another one (as in synchronous circuits, or Lustre programs made of several nodes).

Communication: 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; they become "invisible" in the product.

$\backslash x$ is the encapsulation operation

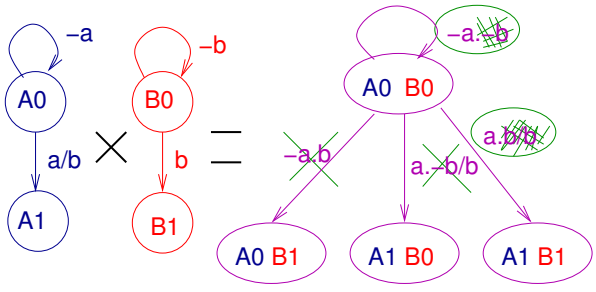
Communication and Encapsulation: $(M_1 \times M_2)\backslash A$

Consider each transition of the product:

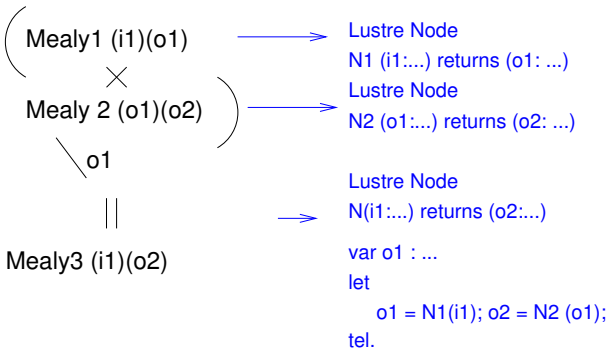
- A label of the form `...-A/A...` is inconsistent:
  Machine M2 reacts as if A was absent, but machine M1 sets A to true
- A label of the form `A.../` is inconsistent too:
  Machine M2 reacts as if A was present, but machine M1 does not set A to true (and nobody else can do that, hence it is false)

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

The Encapsulation Operation



Lustre, Mealy machines, Dataflow Connections and Products (2. With Communication)



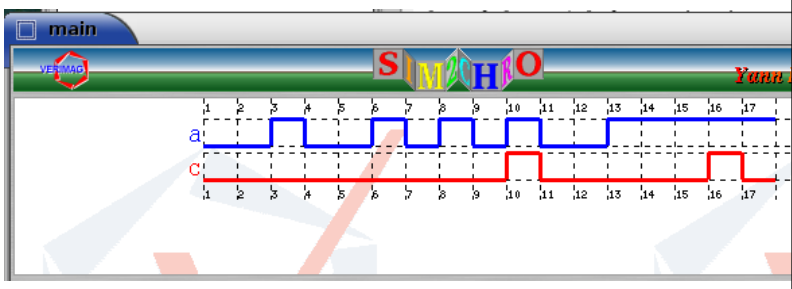
Example: Composing n 1-bit Counters to Obtain n-bit Counters

```
node Bit (x: bool) returns (y: bool) ;
-- one bit counter, i.e., one y every two x's
...

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


Composing n 1-bit Counters to Obtain n-bit Counters

... The automaton view...

The Behaviour



c is emitted exactly when the 4th a occurs.

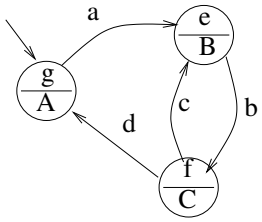
Simple Moore Machines

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

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

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

Trace Semantics



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

Imperative Program

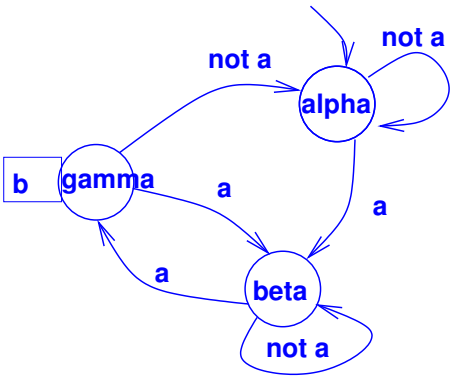
```
State := A  -- take one of the initial states
while (true) loop
    get (I) ; O := null ;
    case State :
        when A : O := g ;
                 if I = a then State:=B ;
        when B : O := e ;
                 if I = b then State:=C;
        when C : O := f ;
                 if I = c then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


Boolean Moore Machines

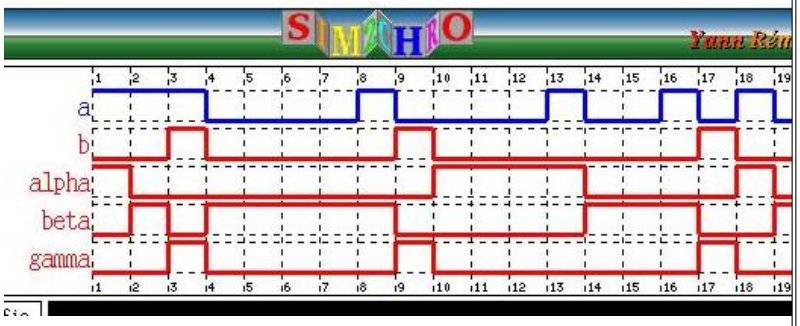
Same definition, but with $I = 2^{BI}$ and $O = 2^{BO}$.

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

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

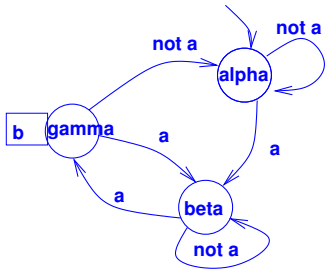


Behaviour



b is emitted one instant *after* the second occurrence of a.

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
```

Synchronous Product (without communication)

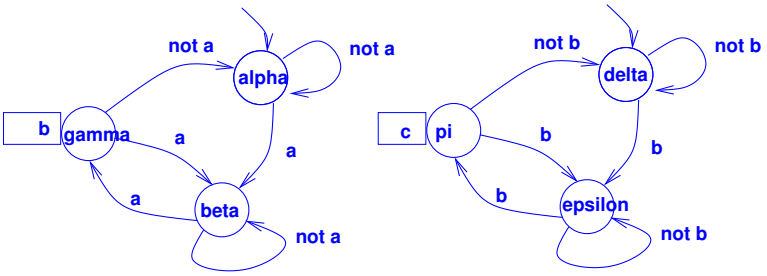
$M_i = (Q_i, Q_{0_i}, I_i, O_i, T_i, F_i)$, for $i = 1, 2$.

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

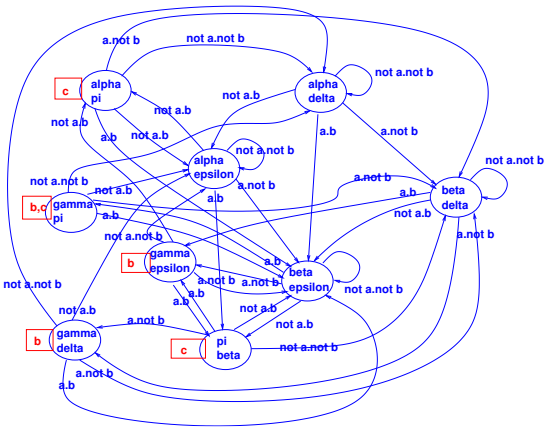
$$\frac{(q_1, m_1, q_1') \in T_1, \quad (q_2, m_2, q_2') \in T_2}{((q_1, q_2), m_1 \wedge m_2, (q_1', q_2')) \in T_{\text{prod}}}$$

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

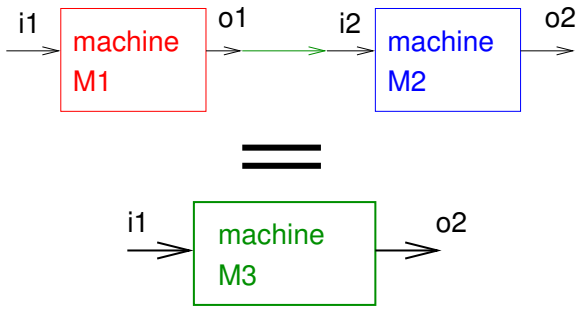
Example



Example, result



Communication: idea



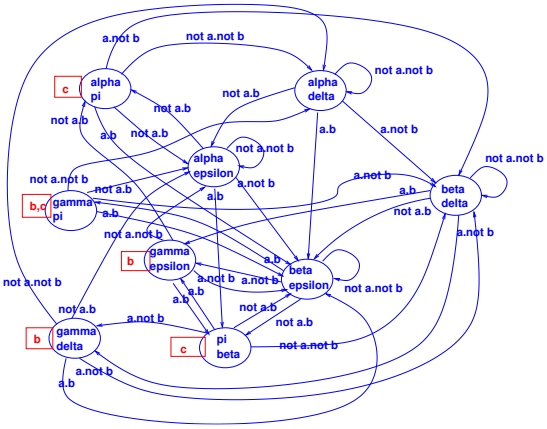
The output of a machine can be connected to the input of another one (as in synchronous circuits).

Communication: encapsulation

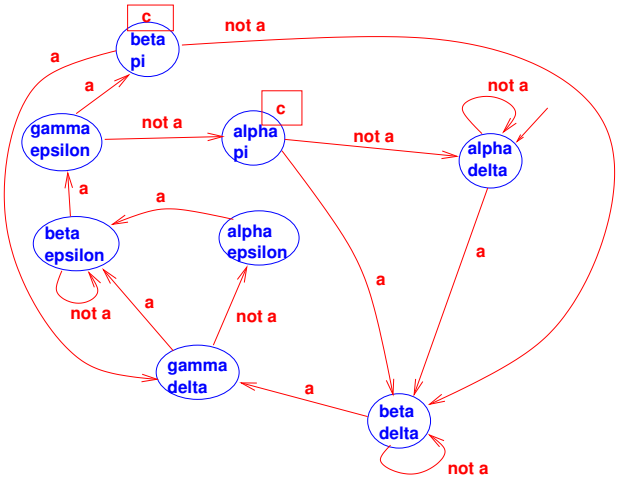
For all the encapsulated signals:

- The transitions that need a signal to be present can be taken only from a state that emits it.
- The transitions that need a signal to be absent can be taken only from a state that does not emit it.

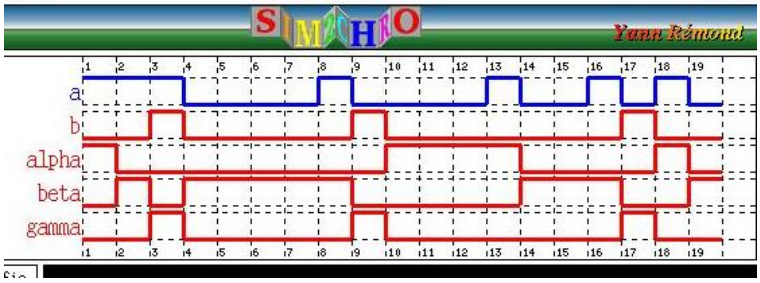
Encapsulation : example (before...)



Encapsulation : example (...after)



Example behavior

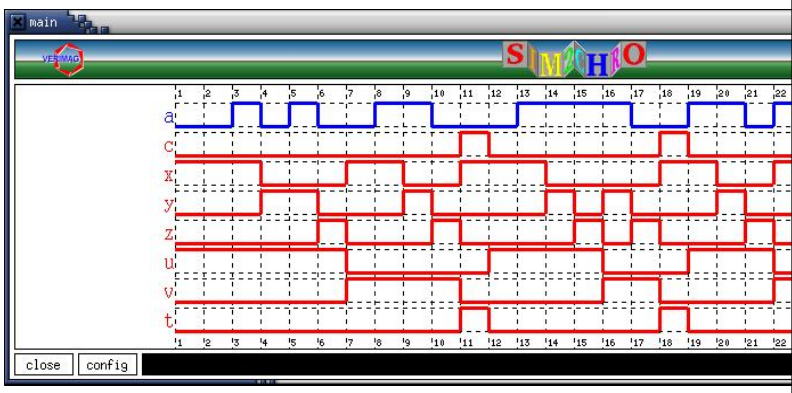


The output 'b' is delayed by one tick, with respect to the second occurrence of the input 'a'.

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'.

Conclusion (1)

The synchronous product of Mealy machines (not Moore Machines) is a perfect design parallelism operator:

- The parallel version of the 2-bit counter has the same number of states as a handwritten counter
- The latency of the output is the same

When *modeling* systems, it's better to use Mealy machines than Moore ones, because you don't "pay" for a convenient decomposition into parallel activities.

Conclusion (2)

Lustre can be used to encode Moore or Mealy machines, or any combination of them.
Each dependency cycle should contain at least one Moore machine (the elementary Moore machine is a PRE operator).

From the Introduction...

How to specify the instructions for use of a HW device, so that the incorrect uses of the device by the SW can be detected?

Motivations: similar to SLAM (Microsoft). See:
SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. Thomas Ball, Byron Cook, Vladimir Levin and Sriram K. Rajamani. January 28, 2004. Technical Report MSR-TR-2004-08.
`ftp://131.107.65.22/pub/TR/TR-2004-08.pdf`

Motivations (c'td)

Android's power management API can lead to 'no-sleep energy bugs'
`www.theverge.com/2012/6/18/3094840/android-power-management-api-no-sleep-energy-bugs`

Forget to remove wakelocks for the devices in a phone (GPS, WIFI, ...)

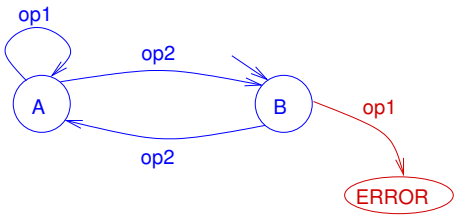
A Simple HW Device

Two states `A`, `B` (`B` is the initial state), two operations `op1`, `op2`, and `op1` should not be used when the device is in state `B`. `op2` switches states of the device. There's no way to "ask" the device in which state it is.

Example: a radio device, `B` *means radio off,* `op2` *switches the radio on or off,* `op1` *means "transmit something via the radio". In other words: one should not try to send something via the radio when it is off; and you have to use* `op2` *to switch it on or off, but you cannot ask it in which state it is.*

A Simple HW Device (as an automaton)

No outputs for the moment, so we don't have to choose between Mealy and Moore yet.



A Simple SW Using the HW Device

A reactive software layer, whose functions can be called from upper layers, and which send commands to the HW (`op1`, `op2`). Needs to "remember" the state of the device, in order to avoid sending `op1` when it should not.

Example: a MAC protocol that uses the radio device: it should make sure that the radio is on when it uses operation `op1` *to send a message.*

A Simple SW Using the HW Device (as C code)

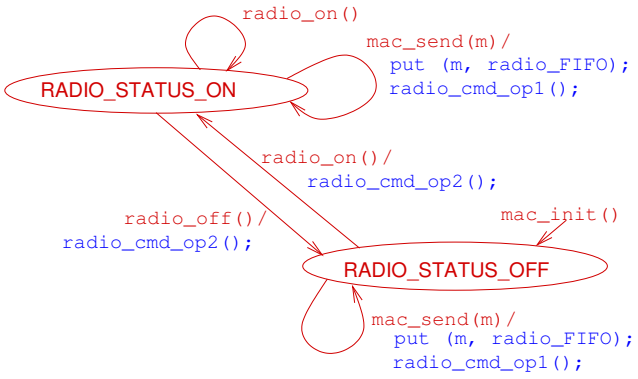
```
#define RADIO_STATUS_ON  0x00
#define RADIO_STATUS_OFF 0x01
int radio_status ;
void mac_init ()
{ ... radio_cmd_init(); radio_status = RADIO_STATUS_OFF; ... }

void radio_on () {
     if (radio_status != RADIO_STATUS_ON) radio_cmd_op2();
     radio_status =  RADIO_STATUS_ON; }
void radio_off () { ... // similar ... }

void mac_send (Message m) {
  put (m, radio_out_FIFO) ;
  // the radio should be in the right state, but no check here
  radio_cmd_op1 ();
  // means "send the content of the output FIFO"
}
```


A Simple SW Using the HW Device (as C code) - Typical Use By An Upper Layer

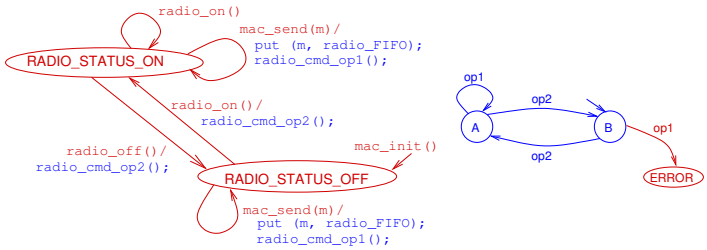
```
mac_init ();
// ...
// prepare a message m to send
// ...
radio_on ();
mac_send (m);
```

Typical bug: forget `radio_on ()` before `mac_send ()`.

A Simple SW Using the HW Device (as an Automaton)

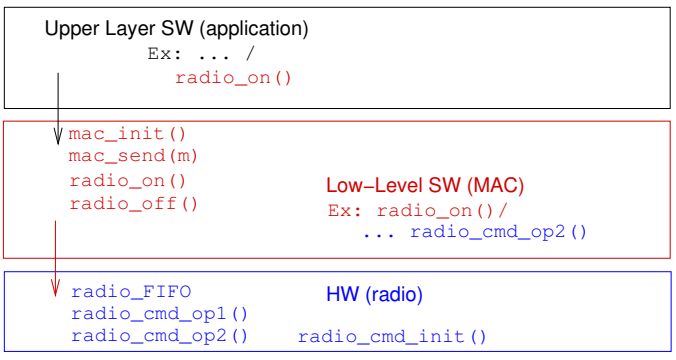


SW Using HW = Synchronous Product of Mealy Machines

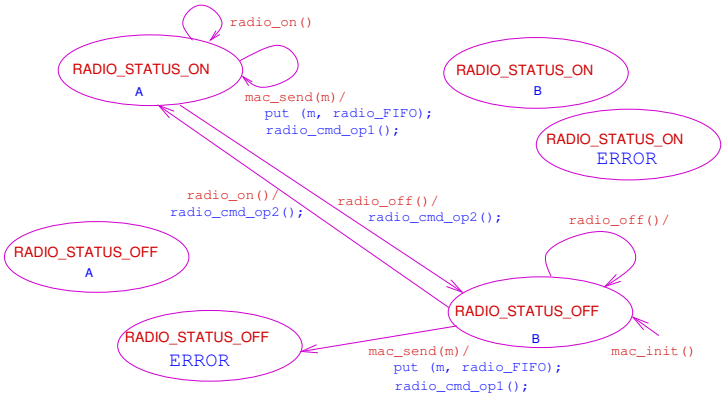


Notation: `radio_cmd_opi = opi`

SW Using HW = A "Dataflow" View of the HW/SW Architecture



SW Using HW = Synchronous Product of Mealy Machines



Potential Errors and How to Detect Them

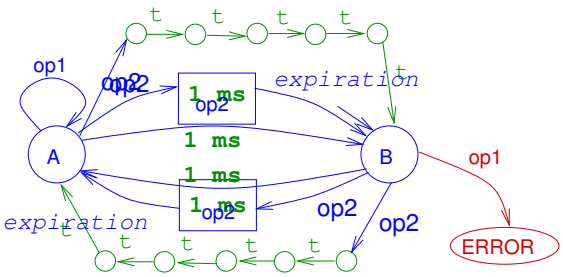
- In the basic software (the MAC layer here), forget to update the variable `radio_status` when using the command `radio_cmd_op2()`; this results in a desynchonization between the real state of the HW, and the vision of it by the SW. The states `RADIO_STATUS_ON` $\times$ `B` and `RADIO_STATUS_OFF` $\times$ `A` could become reachable from the initial state.
- In the application layer (on top of the MAC layer), forget to turn the radio on before sending; this results in an incorrect use of the radio device; the state `RADIO_STATUS_OFF` $\times$ `ERROR` becomes reachable.

Extension 1: Time

What if the transition between states in the HW device takes (significant) time?
Example: Two states `A`, `B` *(*`B` *is the initial state), two operations* `op1`, `op2`, *and* `op1` *should not be used when the device is in state* `B`. `op2` *switches states of the device. There's no way to "ask" the device in which state it is. Switching between states takes 1 ms.*

- Example code using the device: `init(); op2(); op1();` . Should we *wait* 1ms between `op2()` and `op1()`?
- How to model time in the model of the HW?
- How to model the HW/SW interface (do we have to change something)?

Modeling Time (1)



See also later, timed graphs and clocks

Modeling Time (2)

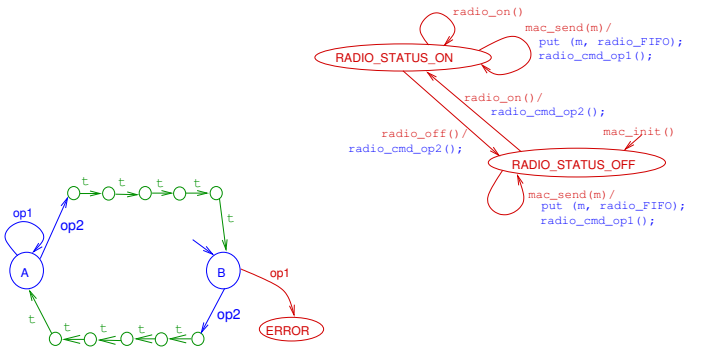
Discrete time in synchronous models and programs: just an additional input `t`, which is the tick of an (external) clock.

The example: 5 ticks is "more or less" 1 ms. Can never be perfect. Better approximation id 1 ms corresponds to 500 ticks (the base clock is more precise).

More on timed models later (timed graphs, ...)

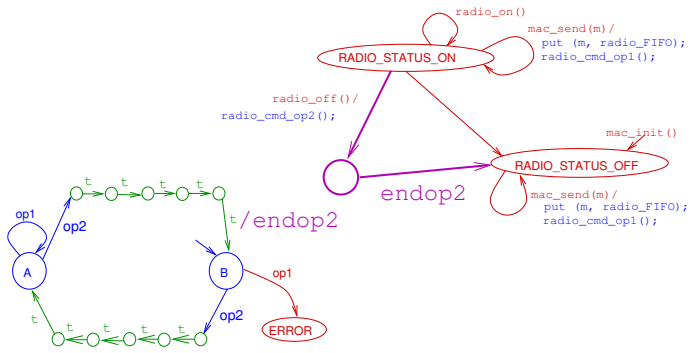
Modeling the HW/SW interface (1)

How to combine a transition of the SW model that emits op2 with the HW model?



Modeling the HW/SW interface (2)

(synchronous calls).
As long as the HW model is in a "counting" state (not yet in the normal destination state), the SW model should not be in its destination state either.



Modeling the HW/SW interface (3)

Depending on how it works in the real system:

- Synchronous calls (see previous slides), with or without time: a single operation (function call) to request the state change and wait until it is done.
- Asynchronous calls (the SW requests the state change — with a synchronous call — but some time will elapse before it is actually done by the HW):
  - The HW is equipped with an "answer" mechanism (can be an interrupt): the SW requests the state change, and then waits for the interrupt;
  - The HW is not equipped with such a mechanism: the SW has to wait long enough, counting time:
    - The SW uses a timer (start it, and wait for the expiration IT)
    - The SW uses "busy wait", executing effectless instructions (problem: with complex processor architectures, it may be very hard to relate a sequence of instructions with physical time).

Extension 2: Failed Operations

When an operation is requested by the SW, it sometimes fails. How to model this ? How to do that together with the modeling of time?

Hint: could use nondeterministic machines.

Conclusions (1)

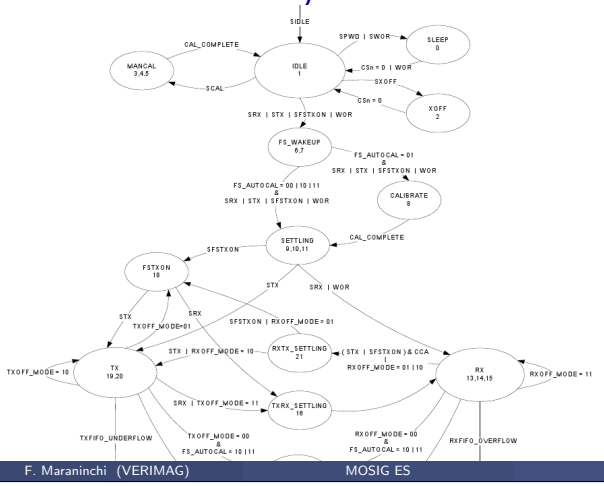
- A SW layer S can be modeled by a Mealy machine (inputs are the calls to functions offered by S to upper layers, outputs are the calls to functions offered to S by lower layers (and ultimately the HW).
- Choosing the "*granularity*" of the model, i.e. the *states*, can be done in several ways, depending on the use of the model.
- The composition of SW layers then corresponds to the synchronous product of their Mealy models. This may reveal incorrect combinations of states.

Conclusions (2)

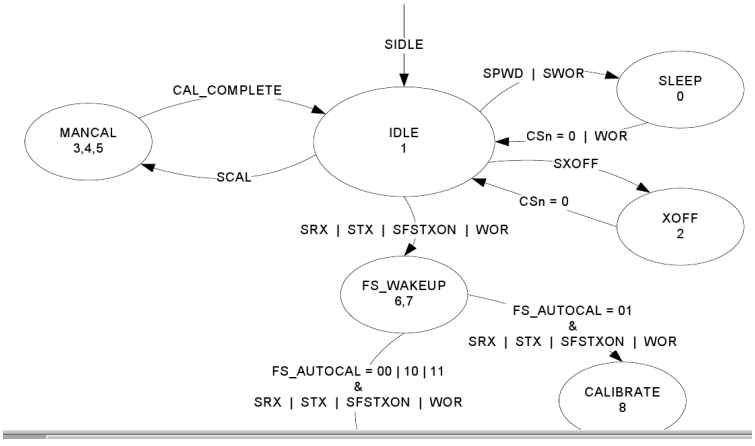
How to Use such an Idea?

- From a piece of handwritten C code to its Mealy model: can sometimes be automated; it's a *model extraction* problem, well known in the domain of formal verification.
- From a piece of HW to its Mealy model: could perhaps be automated from the VHDL/Verilog description of the HW device; has to be "invented" and validated most of the time. Information available in HW documentation.
- The product: easy to automate.
- The analysis of the result: a formal verification problem (see later).

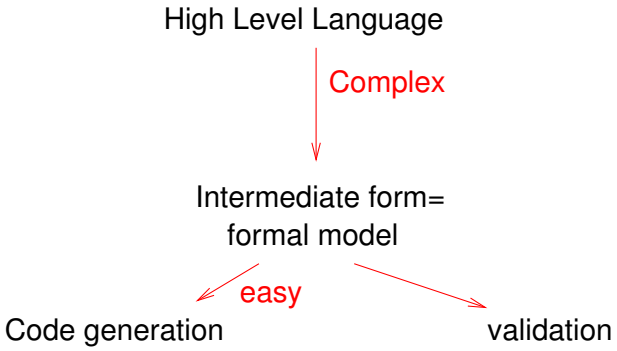
Mealy Models for the HW (Texas Instruments CC1100 radio device)



Mealy Models for the HW (Texas Instruments CC1100 radio device)



Conclusions (3)
Programs, models, validation, ...



Conclusions (4)
system-level validation

- (unfortunately) Not all systems are entirely synchronous and implemented in a centralized way
- For asynchronous systems and/or distributed implementations, there's a need to reason about the *system-level* properties of the design, in particular *timing* properties.
- How to model time so that system design can rely on rigorous analysis of the timing properties (even for asynchronous systems)?

Principles

The idea is to build the set (or a superset) of all the possible behaviors of a concurrent system:

- The individual sequential behaviors are modeled by automata
- Concurrency is modeled by some kind of Cartesian product *(either synchronous or asynchronous)*
- Concurrency without Communication/synchronization is modeled by the complete Cartesian product
- Communication/synchronization is modeled by removing transitions (and therefore reachable states) in the complete Cartesian product.

Several Cases

- Synchronous Product of Boolean Mealy (or Moore) machines + Mealy (or Moore) encapsulation
- Asynchronous Product of Timed Automata in Uppaal + rendez-vous
- Asynchronous Products + several variants of *shared memory* encapsulation. See below.

Again: Synchronous Product of Boolean Mealy Machines (without communication)

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

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

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

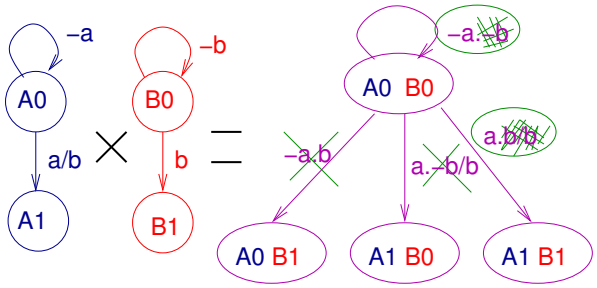
Again: Communication with Signal A The Encapsulation Operation: $(M_1 \times M_2)\backslash A$

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.

The Encapsulation Operation



Summary on Products for Modeling Concurrency (1)

- Synchronous products (the two automata move together): for systems in which there is an intrinsic synchronization, so that it's realistic to model concurrency in such a way. Ex: synchronous circuits, synchronous programs (e.g., Lustre), all kinds of computer systems that are explicitly synchronized.
- Asynchronous products (the two automata never move together, their executions are interleaved): all other systems.

Summary on Products for Modeling Concurrency (2)

For asynchronous systems with some known constraints on asynchrony (e.g., quasi-synchronous systems, N-synchronous systems, systems with a clock-synchronization protocol, ...):

- The synchronous operations can be used to build constrained-asynchronous models (using several clocks and a clock generator, see previous lecture)
- We can design an ad hoc asynchronous product and encapsulation operations (see example below).

Simple Automata for Modeling Asynchronous Systems

Automata with abstract transition labels in a set $\mathscr{A}$ (neither inputs nor outputs):

$$(Q, Q_0 \subseteq Q, \mathscr{A}, T \subseteq Q \times \mathscr{A} \times Q)$$

"Pure" Asynchonous Product: the Interleaving Principle

$A_i = (Q^i, Q_0^i, \mathscr{A}^i, T^i)$ for $i = 1, 2$

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

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

Asynchronous Communication/Synchronization

Since the two participants in an asynchronous system (and also the two operands of an asynchronous product) cannot be guaranteed to be active together (at the same "time"), the only way to exchange information is via some form of *"shared memory"*.

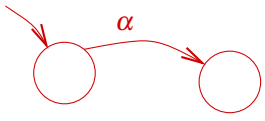
Very similar to the way synchronous products of Moore machines are built: one automaton *"writes"* to the shared memory, and the other can *"read"* from it, later.

Example: Monoprocessor Systems with Shared Memory

A monoprocessor computer, running multiple processes or threads thanks to a time-sharing scheduler. All processes (or threads) access the same memory (directly, without caches!).

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

Models: Simple Automata with 3 Types of Transition Labels



$\alpha, \beta, \ldots$ 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 $A = (Q, Q_0 \subseteq Q, L, T \subseteq Q \times L \times Q)$

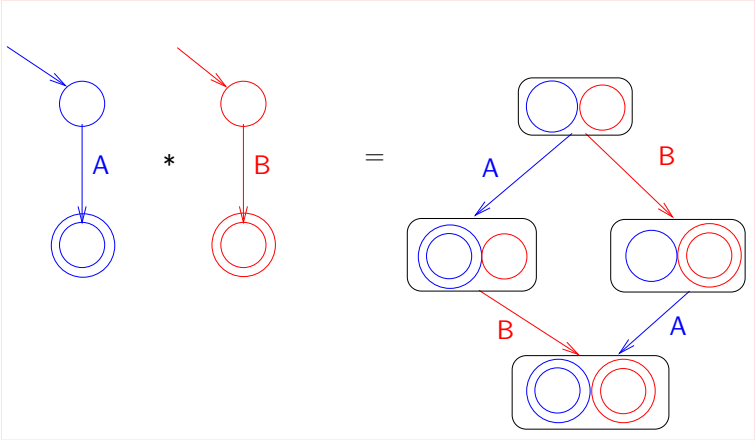
(see also the notion of "process Graphs" p 874 in "Computer Systems, a programmer's perspective", Bryant and O'Hallaron 2003)

A Note on "Granularity" and Atomicity

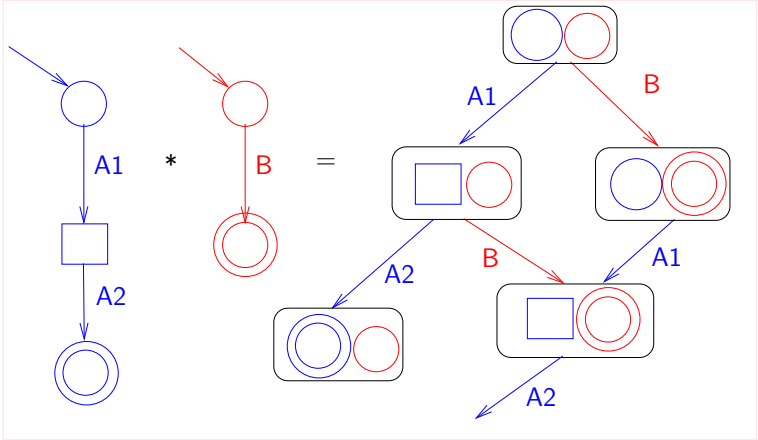
One transition = one atomic action (with respect to the time-sharing scheduler, i.e., something that cannot be interrupted in the middle of its execution).

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" - A is atomic



A note on "granularity" - A is A1;A2

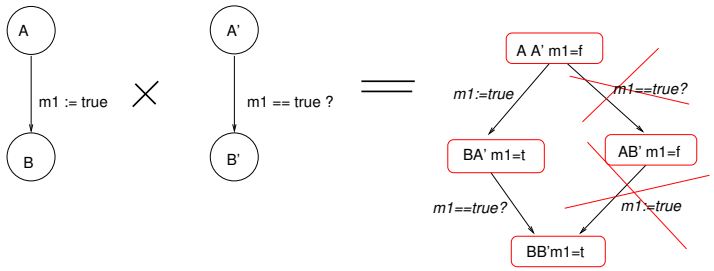


Modeling the Communication via Shared Memory (1)

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 (for our simple models): no memory allocation, no process creation.

Modeling the Communication via Shared Memory (2)



CCS by R. Milner (80's): a Calculus of Communicating Systems[1]

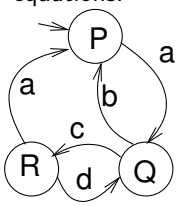
P	::=	$\emptyset$	empty process
	|	$a.P$	prefix: do a, then P
	|	$P+P$	non-deterministic choice
	|	$P \mid P$	parallel composition
	|	$P\backslash a$	encapsulation: force synchro by a

Where:
a is the name of an *atomic action* (in a vocabulary $\mathscr{A}$).
P is the name of a *process*.
Use of *recursive equations* of the form: $Q = a.P + R + c.Q$.

[1] http://en.wikipedia.org/wiki/Calculus_of_communicating_systems

Automata in CCS

A non-deterministic automaton can be described by a set of CCS equations:



$$P = a.Q$$
$$Q = b.P + c.R$$
$$R = a.P + d.Q$$

Not all CCS sets of equations correspond to non-deterministic finite-state and finite-transition automata.
Example: $P = P + a.Q$ (see also the notion of *well-guardedness*).

Encoding an Automaton into CCS Equations...

Is the same as encoding a non-deterministic regognizer for a regular language, into a set of equations whose solution is a corresponding regular expression:

$$P = a.Q$$
$$Q = b.P + c.R$$
$$R = a.P + d.Q$$

$$\begin{aligned} Q &= b.P + c.(a.P + d.Q) \\ &= (b + c.a).P + c.d.Q = \\ &= (c.d)^*.(b + c.a).P \\ P &= a.(c.d)^*.(b + c.a).P \\ &= [a.(c.d)^*.(b + c.a)]^* \end{aligned}$$

Axiom:
$X = a.X + b$ implies
$X = a^*.b$

Synchronization in CCS

The synchronization primitive is the *binary rendez-vous*.
Use of special actions: $\mathscr{V} = \mathscr{A} \cup \overline{\mathscr{A}}$

Idea: action a in a process P is synchronized with action $\overline{a}$ in another process P', with a rendez-vous mechanism:

- P (resp. P') cannot do an a (resp. $\overline{a}$) without P' (resp. P) doing an $\overline{a}$ (resp. a) "at the same time".
- When P and P' do an a and an $\overline{a}$ together, the result is unobservable for further synchronizations. The transition has a new label τ representing an internal action.

Operational Semantics of CCS

Notation: $P -\!\!- a \longrightarrow P'$ means that P evolves into P' with action a.
Inference rules:

$$\frac{P -\!\!- a \longrightarrow P'}{P+Q -\!\!- a \longrightarrow P'} \qquad \frac{Q -\!\!- a \longrightarrow Q'}{P+Q -\!\!- a \longrightarrow Q'}$$

$$\frac{P -\!\!- a \longrightarrow P'}{P|Q -\!\!- a \longrightarrow P'|Q} \qquad \frac{Q -\!\!- a \longrightarrow Q'}{P|Q -\!\!- a \longrightarrow P|Q'}$$

$$a.P -\!\!- a \longrightarrow P$$

$$\frac{P -\!\!- a \longrightarrow P', \quad Q -\!\!- \overline{a} \longrightarrow Q'}{P|Q -\!\!- \tau \longrightarrow P'|Q'}$$

$$\frac{P -\!\!- a \longrightarrow P', \quad a \notin \{b, \overline{b}\}}{P \backslash b -\!\!- a \longrightarrow P' \backslash b}$$

Parallelism in CCS (without synchronization)

$$P = P_1 \mid P_2$$
$$P_1 = a.Q_1$$
$$Q_1 = b.P_1$$
$$P_2 = c.Q_2$$
$$Q_2 = d.P_2$$

Expansion theorem:

$a.P \mid b.Q$ is the same as
$a.(P|b.Q) + b.(a.P|Q)$

This is exactly the same as the asynchronous product of the two automata, representing all possible interleavings of P and Q actions.

$$\begin{aligned} P_1 \mid P_2 &= a.Q_1 \mid c.Q_2 \\ &= a.(Q_1 \mid c.Q_2) + c.(a.Q_1 \mid Q_2) \\ &= a.(b.P_1 \mid c.Q_2) + c.(a.b.P_1 \mid d.P_2) \\ &= \ldots \end{aligned}$$

Comments on the Synchronization in CCS

$(P \mid Q)\backslash a$ represents the rendez-vous synchronization, on the labels $a, \overline{a}$, of P and Q.

exercise

Redefine the CCS *rendez-vous* mechanism in terms of a dedicated asynchronous product + an encapsulation operation, on explicit automata.

Note that the semantics of the rendez-vous is expressed in two places:
– the special rule for the operator $|$, for $a, \overline{a}$
– the restriction rule, to remove the transitions labeled with a or $\overline{a}$, taken alone.

Other Process Algebras
Other Formalisms/Languages with rendez-vous

SCCS (Synchronous CCS, Milner)
CSP (Communicating Sequential Processes) by T. Hoare,
ACP (Bersgtra and Klop),
LOTOS (norm)
Meije (R. de Simone et al., 1985)

LOTOS (Asynchronous, n-ary rendez-vous with data exchange)
Ada,
...

Comments

- Process Algebras are most of the time used as a language for *implicit automata*, products, and restrictions.
- There are *process algebras* for asynchronous or for synchronous processes, or for both (SCCS, Meije)
- Rendez-vous is only one possible choice for a synchronization mechanism between parallel processes. It is very different from the *synchronous broadcast* used in synchronous models.

Choices

- Explicit shared memory, semaphores, ...
- Synchronous Broadcast
- (binary) Rendez-vous
- ...

Shared Memory

- This is an asymmetric mechanism
- Together with the (pure) asynchronous product (one process writes to the memory, others read the memory, *later*).
- Together with the synchronous product of Moore machines (essentially the same)

– Represents exactly what happens in some *implementation synchronization mechanisms* (e.g., threads on a mono-processor with semaphores, synchronous circuits connected via their memory)

Synchronous Broadcast

The mechanism defined by the synchronous product of Mealy machines:

- This is an asymmetric mechanism: the emitter can always proceed, even if nobody listens
- Broadcast: any number of "listeners" (including 0)...
- Synchronous: ... that react at the same time

— Allows the use of *observers*, i.e. additional programs that observe a program P without changing its behavior; is the basis of *free design parallelism*.
— Represents exactly what happens in some *implementation synchronization mechanisms* (e.g., circuits, compiled synchronous programs).

(binary) Rendez-vous

- Rendez-vous is a *symmetric* operation (both processes are blocked)
- Together with a special asynchronous product: it represents a very high-level synchronzation mechanism
- Can be described by the synchronous broadcast (a principle called "instantaneous dialogue")
- Cannot be extended easily to *n-ary* rendez-vous

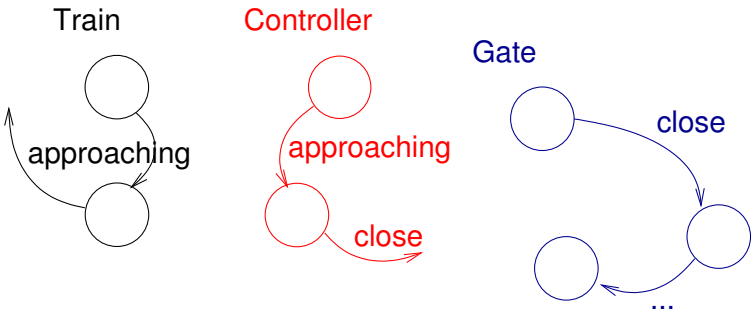
— Does not represent exactly an *implementation synchronization mechanism* (in fact, rendez-vous is quite hard to implement in general, except in synchronous systems!).

An Example of misusing the rendez-vous mechanism in models: "The gate stops the train"

Problem: a process representing the gate, another process representing the controller, another process representing the train(s).

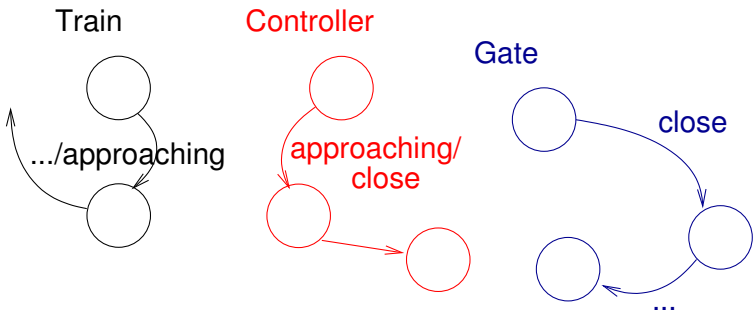
- Modeling with explicit inputs/outputs and the synchronous broadcast: ok
- Modeling with abstract labels, the rendez-vous, and asynchronous products: there might be a bias in the model: the gate stops the train!

"The gate stops the train"



If the gate is in a state that does not accept "close", it blocks the controller... which won't be able to do a correct cycle until it reaches a state in which it can accept "approaching"... hence the train cannot do "approaching" either: it is stopped!

With inputs and outputs



The notion of "Input-Enabledness"

Intuitively: a process P cannot "*refuse*" an action a that another process wants to execute, hence there are transitions with label a from each state of P. This is what you need to represent "inputs" in modeling formalisms where the synchronization is expressed by the rendez-vous mechanism.

Action transducers and timed automata
Frits Vaandrager and Nancy Lynch
Lecture Notes in Computer Science Springer Berlin / Heidelberg
Volume 630, 1992
Proceedings of CONCUR '92

Design Parallelism vs Implementation Parallelism

The only possible "design" parallelism is the synchronous product of Mealy machines.

The synchronous product of Moore machines has a cost (no instantaneous communication) ;
All asynchronous products have a cost (same remark).