Synchronous circuits, Boolean Automata and their Synchronous Parallel Commposition

Marc Pouzet

École normale supérieure
Marc.Pouzet@ens.fr

M1, Calcul parallèle et réactif
Novembre 2018

In this course

- A minimal model to describe a reactive system with inputs/outputs.
- Understand what is the parallel composition of two reactive systems.
- Under what circonstances/constraint, the composition of two deterministic reactive systems is still a deterministic system.
- Equivalence between an explicit and implicit representation of a reactive system: synchronous circuit *vs* boolean automata.
- *One-hot* coding.
- Synchronous parallel and hierarchical composition of boolean automata.
- Application: translation of pure Esterel into synchronous circuits.

Automata with inputs/outputs

A reactive system is a system that continuously reads inputs and produce outputs.

A minimal model is an *automaton* [1] with inputs and outputs.

It can encode the control structure of a sequential program that run in bounded time and space and cyclically read inputs and produce outputs.

Two classical models of automata with input/output has been considered.

[1] In the field of circuits and embedded system design, the name "Finite State Machines" is used more often.

Moore Machines

An automaton with inputs and outputs. In a Moore machine, the output depends on the state only.

Definition

A Moore automaton is a tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$

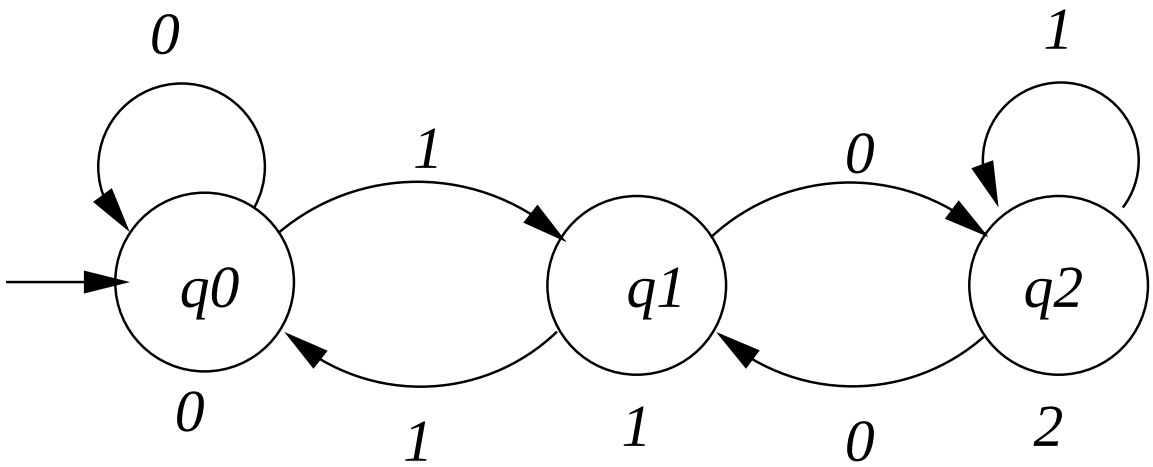
- Q is a finite set of states, q_0 is the initial state.
- Σ is the finite input alphabet, Δ the output alphabet.
- δ is an application from $Q \times \Sigma$ to Q.
- λ is an application from Q to Δ, that gives the output associated to every state.

The answer of M to input $a_1 a_2 ... a_n$, $n \geq 0$ is $\lambda(q_0)\lambda(q_1)...\lambda(q_n)$ where $q_0, ..., q_n$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$.

Remark: A Moore automaton returns the output $\lambda(q_0)$ for input ϵ.

Example

Counter modulo 3 from a binary word.



On input 1010, the sequence of states is q_0, q_1, q_2, q_2, q_1 producing output 01221. For ϵ, returns 0; for 1, returns 1; for 2 returns 2; for 5 returns 2 and for 10, returns 1.

Mealy Machines

Conversely, in a Mealy machine, the output depends on the current state and current input.

Definition

A Mealy automaton is a tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$

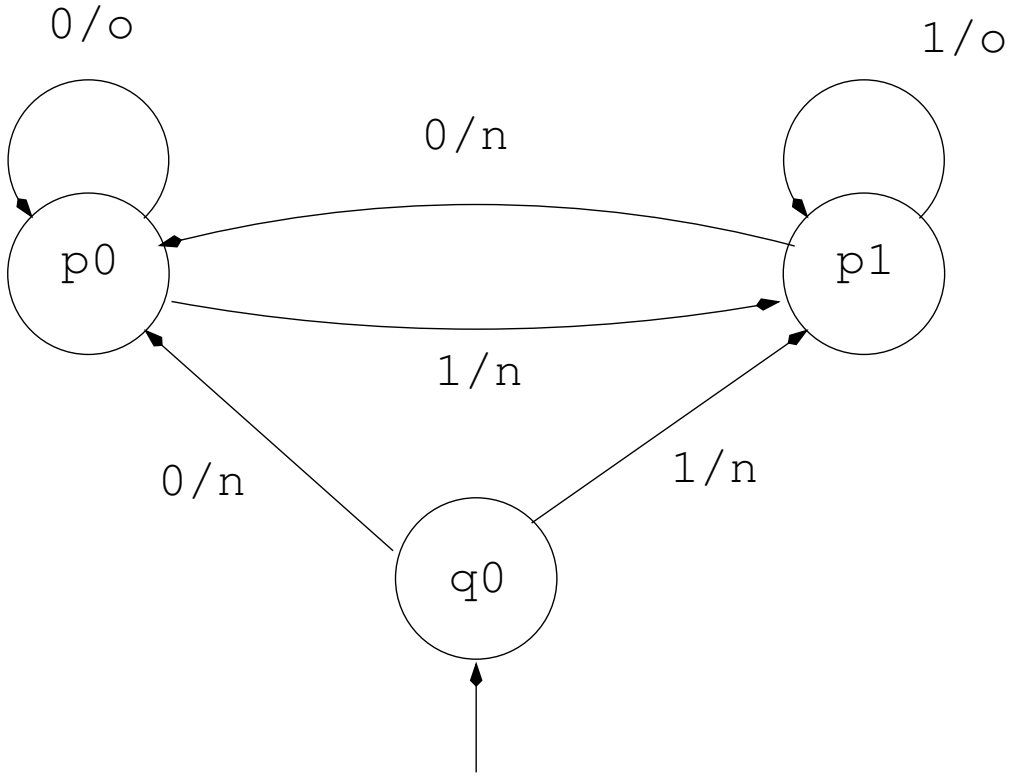
- Q is a finite set of stages, q_0 the initial state.
- Σ is a input alphabet, Δ the output alphabet.
- δ is an application from $Q \times \Sigma$ to Q
- λ is an application from $Q \times \Sigma$ to Δ

$\lambda(q, a)$ returns the output associated to a transition from state q with input a.
The output of M for input sequence $a_1 ... a_n$ is $\lambda(q_0, a_1)\lambda(q_1, a_2)...\lambda(q_{n-1}, a_n)$ where $q_0, q_1, ..., q_n$ is the sequence of states such that $\delta(q_{i-1}, a_i) = q_i$ for $1 \leq i \leq n$.

Remark: This sequence is of length n whereas it was of length $n + 1$ for a Moore automaton. On input ϵ, a Mealy automaton returns the output ϵ.

Example

Recognize words from $\{0, 1\}$ which terminate either with 00 or 11. A Mealy automaton with 3 states which returns *o* (for ok), when the input is valid and *n* (for not ok) otherwise.



The answer of M to input 01100 is *nnono*.

Equivalence

$T_M(w)$ is the output produced by M on input w.

Definition (Equivalence between automata)

A Moore automaton M' is equivalent to a Mealy automaton M if for any input w, $bT_M(w) = T_{M'}(w)$ where b is the output of M' in the initial state.

Theorem (Equivalence)

- *If M_1 is a Moore atomaton it exists a Mealy automaton M_2 equivalent to M_1.*
- *If M_1 is a Mealy automaton it exists a Moore automaton M_2 equivalent to M_1.*

Remark: Mealy automata are more concise than Moore automata. Encoding a Mealy automaton into an equivalent Moore automaton may need an number of states at worst equal to $|Q'| \times |\Delta|$

From Moore to Mealy

Let M_1 a Moore automaton. Build a Mealy automaton $M_2 = (Q, \Sigma, \Delta, \delta, \lambda', q_0)$ such that $\lambda'(q, a) = \lambda(\delta(q, a))$

From Mealy to Moore

Let M_1 a Mealy automaton. Build $M_2 = (Q', \Sigma, \Delta, \delta', \lambda', [q_0, b_0])$ where b_0 is any element from Δ. States in M_2 are pairs $[q, b]$ made of a state from M_1 and an output symbol ($Q' = Q \times \Delta$). We define:

$$\delta'([q, b], a) = [\delta(q, a), \lambda(q, a)]$$

and:

$$\lambda'([q, b]) = b.$$

Representing an automaton by a synchronous circuit

An automaton can be represented by a synchronous circuit.

A sequential synchronous circuit is a set of logical operators (e.g., and, or, not gates) and synchronous registers.

Definition

- A finite set of input variables I, outputs variables O, memories (state variables or registers) S.
- An initial state: $\vec{init} \in IB^{|S|}$ defines the initial value of the registers.
- Output functions: $o_j = f_j(\vec{s}, \vec{i}) \in IB$
- Transition functions: $s'_k = g_k(\vec{s}, \vec{i}) \in IB$

A **synchronous observer** is a circuit with a single boolean output.

From an implicit to an explicit automaton

A synchronous circuit is an implicit representation of a Mealy machine.

- Set of inputs I, outputs O, states S (finite)
- Initial state $\vec{init} \in \mathbb{B}^{|S|}$.
- Output function $(f_j)_{j \in [1..|O|]}$, transition function $(g_k)_{k \in [1..|S|]}$

Mealy machine:

- The input alphabet is the set of all possible tuple values for inputs.
- The output alphabet is the set of all possible tuple values for outputs.

From implicit to explicit

By enumeration of boolean values.

- Input alphabet $IB^{|I|}$, output alphabet $IB^{|O|}$
- Let $Q = IB^{|S|}$, $q_{init} = \vec{init}$
- $q \xrightarrow{\vec{i}/\vec{o}} q'$ iff $\begin{array}{rcl} \vec{o} & = & (f_1(q, \vec{i}), \ldots, f_{|O|}(q, \vec{i})) \\ q' & = & (g_1(q, \vec{i}), \ldots, g_{|S|}(q, \vec{i})) \end{array}$

Exponential gain in size between an implicit and explicit automaton.
(n input variables encode up to 2^n inputs; m state variables encode up to 2^m states).

From explicit to implicit

Various solutions, more or less efficient. The simplest is *one-hot* coding.

- Input alphabet $\Sigma = \{a_1, \ldots, a_n\}$
- Output alphabet $\Delta = \{b_1, \ldots, b_m\}$
- Finite set of states Q and initial state $Init$
- Transition function $T : Q \times \Sigma \to \Delta \times Q$
- For any state q, write:
  - $Prec(q) = \{(p, i) \ / \ p \stackrel{i/o}{\to} q\}$
  - $Succ(q) = \{(i, r) \ / \ q \stackrel{i/o}{\to} r\}$
- Write $Output(o) = \{(p, q, i) \ / \ p \stackrel{i/o}{\to} q\}$

One hot-coding:

- A boolean state variable s_{q_j} per explicit state;
- a boolean variable i_k per element of Σ;
- a boolean variable o_k per element of Δ.
- Every state variable s_q and output variable o is defined by:
  - Let $i_1, \ldots, i_n$ and $p_1, \ldots, p_n$ such that $(p_k, i_k) \in Prec(q)$, $k \in \{1, \ldots, n\}$
  - Let $j_1, \ldots, j_m$ when there exists r such that $(j_k, r) \in Succ(q)$, $k \in \{1, \ldots, m\}$

$$\begin{array}{rcl} s'_q & = & \texttt{if } s_q \texttt{ then } \texttt{not}(j_1) \wedge \cdots \wedge \texttt{not}(j_m) \\ & & \texttt{else } s_{p_1} \wedge i_1 \vee \cdots \vee s_{p_n} \wedge i_n \\ o & = & \bigvee_{(p,q,i) \in Output(o)} (p \wedge i) \end{array}$$

- Initial state $Init = (Init_1, \ldots, Init_{|Q|})$ such that $Init_k = 1$ and $Init_j = 0$ for all $j \neq k$ if q_k is the initial state.

Remarks:

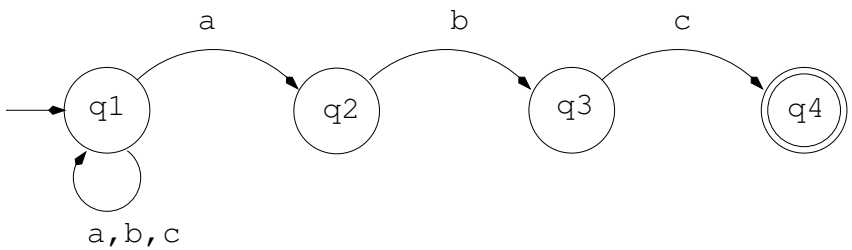
- n boolean variables to encode n states whereas $\log n$ is enough.
- Same thing for the input and output alphabet.
- Other encoding exist.

Recognising regular expressions

A synchronous circuit is an efficient way to implement a recogniser of regular expressions or a non determinitic automaton without making it deterministic first.

Read the (wonderful) paper by P. Raymond (ICALP'96). [2]
It explains how to build a synchronous circuit from a regular expression, with a size that is proportional.

Example: consider the non deterministic automaton which describe words over the alphabet $\{a, b, c\}$ that contain the sequence abc.
The corresponding regular expression is
$(a + b + c)^* \, abc \, (a + b + c)^*$



[2] P. Raymond, "Recognizing Regular Expressions by means of Dataflow Networks", ICALP'96. Paper available on the web page of the course.

Build a one-hot coding.

- Three inputs a, b and c which must be exclusive with always one true. E.g., $a = 1$ when the current input is 'a'.
- Represent every state by a boolean variable. E.g., $q = 1$ means that the active state is q.
- Output $ok = 1$ at instant $n \in \mathbb{N}$ when the input prefix of length n belongs to the language.

Remark:

Several state variable can be true at the same time.

Transition

$$\begin{array}{rlcl} \forall n \in \mathbb{N}^*. & q1_n & = & q1_{n-1} \\ & q2_n & = & q1_{n-1} \wedge a \\ & q3_n & = & (q2_{n-1} \wedge b) \\ & q4_n & = & (q3_{n-1} \wedge c) \vee q4_{n-1} \\ & ok_n & = & q4_n \end{array}$$

$$q1_0 = 1 \qquad q2_0 = q3_0 = q4_0 = 0$$

Implementation as a Lustre program

This set of data-flow equations can be written in Lustre.
`pre` . is the synchronous register. . `->` . is the initialization.

- $\forall n \in \mathbb{N}^*, (\texttt{pre}\, x)_n = x_{n-1}$ and $\texttt{pre}\, x_0 = \mathit{nil}$
- $\forall n \in \mathbb{N}^*, (x \texttt{->} y)_n = y_n$ and $(x \texttt{->} y)_0 = x_0$

```
node grep_abc(a, b, c: bool) returns (ok: bool);
  var q1, q2, q3, q4: bool;
  let
    q1 = true -> pre q1;
    q2 = false -> pre q1 and a;
    q3 = false -> pre q2 and b;
    q4 = false -> (pre q3 and c) or pre q4;
    ok = q4;
  tel;
```

The program is deterministic and linear in size whereas determinising an non deterministic automata may result in an exponential blow up. The program can be compiled into sequential code (of proportional size).

Boolean automata

- Transitions can be made with boolean expressions.
- Equivalence with an implicit automaton is trivial, e.g., using one-hot coding.
- Transitions are of the form:

$$p \overset{f/o_1,\ldots,o_n}{\rightarrow} q$$

where f is a boolean formula on input variables and o_i are output variables.

$$f \quad ::= \quad x \mid f \vee f \mid f \wedge f \mid \overline{f} \text{ où } x \in I$$

factorizes transitions (exponential gain on the number of transitions w.r.t using an alphabet). E.g., transitions $p \overset{f_1/o}{\rightarrow} q$ and $p \overset{f_2/o}{\rightarrow} q$ are represented by $p \overset{(f_1 \vee f_2)/o}{\rightarrow} q$.

From implicit to explicit

Compilers manage both representations (explicit and implicit).

Implicit

- Reasonnable size thus good model for code generation: corresponds to the compilation in "single loop code" for Lustre.
- More compact; boolean simplification algorithms.

Explicit

- (potentially) exponential size.
- Simple model for analysis and verification: an infinite number of equivalent automata but a unique minimal one.
- In practice, it is impossible to build an explicit automaton from an implicit one.

Synchronous Composition of Boolean Automata

The composition of synchronous circuits is the composition of functions. Take two synchronous circuits $C^c = (I^c, O^c, S^c, init^c, f^c, g^c)$ with $c \in \{0, 1\}$.

- Set of inputs I^c, outputs O^c, states S^c (finite)
- Initial state $init^c \in \mathbb{B}^{|S^c|}$
- Output function $(f_j^c)_{j \in [1..|O^c|]}$
- Transition function $(g_k^c)_{k \in [1..|S^c|]}$

Compose them in parallel, with some inputs of C^1 be outputs of C^2 and some inputs of C^2 be outputs of C^1. Let $L = L^1 \cup L^2$ be the set of local variables such that:

$$\begin{array}{rclcrcl} O^1 & = & OO^1 \cup L^2 & & O^2 & = & OO^2 \cup L^1 \\ I^1 & = & L^1 \cup II^1 & & I^2 & = & L^2 \cup II^2 \end{array}$$

L^1 are the inputs of C^1 which are outputs of C^2. L^2 are the inputs of C^2 which are outputs of C^1.

The parallel composition is a circuit $C = (II, OO, S, init, f, g)$ where:

- $S = S^1 \cup S^2$
- $Init = (Init^1, Init^2)$
- $OO = OO^1 \cup OO^2$ is the set of outputs with $\vec{oo} = (\vec{oo}^1, \vec{oo}^2)$.
- $II = II^1 \cup II^2$ is the set of inputs with $\vec{ii} = (\vec{ii}^1, \vec{ii}^2)$.

Let $\vec{o^1} = (\vec{oo}^1, \vec{l^2})$, $\vec{o^2} = (\vec{oo}^2, \vec{l^1})$, $\vec{i^1} = (\vec{ii}^1, \vec{l^1})$, $\vec{i^2} = (\vec{ii}^2, \vec{l^2})$.

If $\vec{s^1} \in \mathbb{B}^{S^1}$ and $\vec{s^2} \in \mathbb{B}^{S^2}$, the parallel composition of C^1 and C^2 must verify:

$$
\begin{array}{lcl}
oo_j^1 = f_j^1(\vec{s^1}, \vec{i^1}) \text{ for all } j \in OO^1 & \wedge & II_j^1 = f_j^1(\vec{s^1}, \vec{i^1}) \text{ for all } j \in LL^1 \\
o_j^2 = f_j^2(\vec{s^2}, \vec{i^2}) \text{ for all } j \in OO^2 & \wedge & II_j^2 = f_j^2(\vec{s^2}, \vec{i^2}) \text{ for all } j \in LL^2 \\
s'^1_k = g_k(\vec{s^1}, \vec{i^1}) \text{ for all } k \in S^1 & \wedge & s'^2_k = g_k(\vec{s^2}, \vec{i^2}) \text{ for all } k \in S^2
\end{array}
$$

Parallel composition: be careful of cycles!

By connecting some of the output of C^2 to inputs of C^1 and conversely, it is possible to get a cyclic circuit that contain a *combinatorial loop*.

This means that the set of boolean equations has no solution or several. The result is no more a synchronous circuit.

Examples

1. Compose a synchronous circuit $\mathit{Indentity}(x, y)$ with input x and output y such that $y = x$, with $\mathit{Identity}(y, x)$, and add in parallel the equation $z = x$, with z an output, the result must verify:

   $$x = y \wedge y = x \wedge z = x$$

   which has two solutions for z ($z = 0$ or $z = 1$ are two valid solutions).

Examples

1. Compose $Identity(x, y)$ with the logical inverter $Not(x, y)$ such that $y = \texttt{not}(x)$ and $z = x$, the result is:

$$y = x \wedge x = \texttt{not}(y) \wedge z = x$$

   which has no solution (it is not possible to give either 0 or 1 to the output z and get a logically correct set of equations).

2. Some boolean equations with a combinatorial loop may have a unique solution. E.g.,

$$tobe = tobe \vee \texttt{not}(tobe)$$

3. Some boolean equations may have a unique solution. E.g.,

$$x = mux(c, a, y) \wedge y = mux(c, x, b)$$

Sufficient condition for f and g: no combinatorial loop, that is, an output does not depend instantaneously on itself.

Boolean Mealy machine

$M = (S, s_o, I, O, T)$ where:

- I: input variables, O: output variables with $I, O \subseteq A$
- $T \subseteq S \times f(I) \times 2^O \times S$
- $f(I)$ is a boolean formula over I

Determinism: For all state s and for all pair of transitions $s \overset{b_i/\dots}{\rightarrow} s'$ and $s \overset{b_j/\dots}{\rightarrow} s''$, $b_i \wedge b_j = \mathit{false}$

Reactivity: For all state s, the set of transitions $s \overset{b_i/\dots}{\rightarrow} s_i$, $0 \leq i \leq k$ from s verifies $\vee_{0 \leq i \leq k} b_i = \mathit{true}$

We say that an automaton is **causal** when it is reactive and deterministic.

Summary

The composition of Boolean Mealy machines (or synchronous circuits) may not define a Boolean Meany machine because of cycles.

On the contrary:

The composition of Boolean Moore machines, that is, where the output is a function of the state variables only, always result in a Boolean Moore machine: all cycles cross a synchronous register. E.g.:

```
x = x0 -> pre(f(y));
y = y0 -> pre(g(x))
```

Synchronous Parallel Composition

What is the meaning of $P||Q$ where P and Q are two transition systems? If both P and Q are causal, is $P||Q$ causal?
Synchronous Product:

$$(p, q) \overset{c_1 \wedge c_2 / e_1, e_2}{\rightarrow} (p', q') \text{ if } (p \overset{c_1/e_1}{\rightarrow} p') \wedge (q \overset{c_2/e_2}{\rightarrow} q')$$

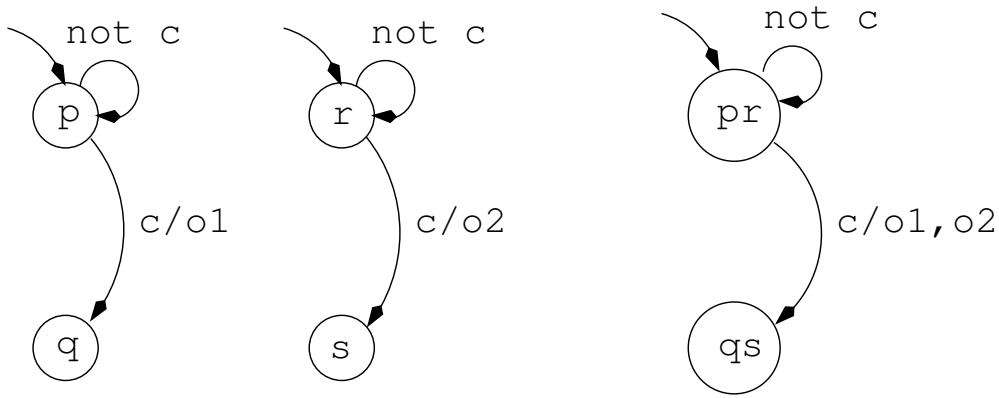
- Cartesian product of states with conjunction of gards and union of outputs.
- Synchronous broadcast: a signal is broadcast to all other signals:
- sending is non blocking;
- an arbitrary number of processes can receive a signal (broadcast)
- reaction to a broadcast is instantanous (same instant).

Some conditions on transitions are unsound

- $\xrightarrow{\overline{A}/A}$: if A is absent, is A emitted? (all reaction must be logically sound)
- $\xrightarrow{A/A}$ when A is a local signal? (non determinacy)
- $\xrightarrow{A/\ldots}$ where A is local and not emitted? (a signal is present if it is emitted)

Examples

No communication/synchronisation



The states qr and ps are unreachable. They would mean that if is possible to have the transitions:

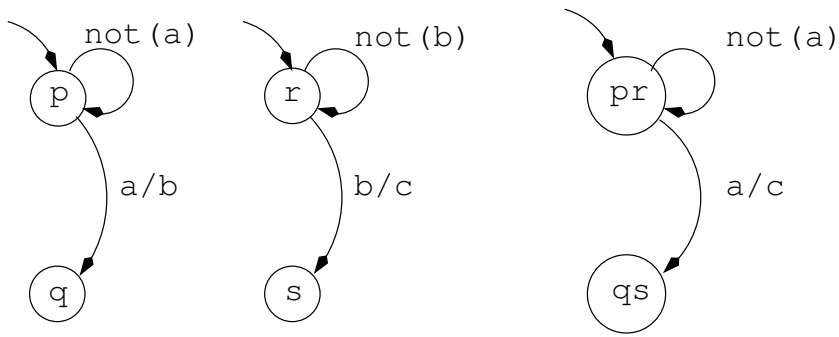
$$pr \overset{c \wedge \mathtt{not}(c)}{\rightarrow} qr \qquad pr \overset{\mathtt{not}(c) \wedge c}{\rightarrow} ps$$

which are logically unsound.

Communication and hiding

Suppose that b is a local signal.

- Reaction to a broadcast is instantaneous: when b is emitted, it is immediately seen present.
- Add a hiding operation to eliminate certain transitions from the cartesian product.



In the composition of the two, the other transitions to consider are:

$$pr \xrightarrow{\texttt{not}(a)\wedge b/c} ps \quad pr \xrightarrow{a\wedge\texttt{not}(b)/b} qr \quad qr \xrightarrow{b/c} qs \quad ps \xrightarrow{a/b} qs$$

Hiding

hide $b\,P$ (b is a local signal in P)

- Makes b local;
- synchronous product of the two automata:

$$(p,q) \overset{c_1 \wedge c_2 / e_1, e_2}{\rightarrow} (p',q') \text{ if } (p \overset{c_1/e_1}{\rightarrow} p') \wedge (q \overset{c_2/e_2}{\rightarrow} q')$$

- some transition are logically unsound: keep transition $\overset{c/e}{\rightarrow}$ iff: $(b \in e \Rightarrow c \wedge b \neq \mathit{false}) \wedge (b \notin e \Rightarrow c \wedge \mathtt{not}(b) \neq \mathit{false})$
- no logical contradiction during a reaction;
- then remove b from transitions.

As a result, only the transitions:

$$pr \overset{\mathtt{not}(a)}{\rightarrow} pr \quad pr \overset{a/c}{\rightarrow} qs$$

are kept.

Causality

- If P and Q are **causal**, $P||Q$ is not necessarily **causal**.
- A static analysis, called **causality analysis** is used to ensure that the overall program is causal.
- This analysis should be modular, i.e., compute the causality from P and from Q, then compose the two causalities.

Automata and circuits:

This definition of parallel composition coincides with that of synchronous circuits.
Indeed, what happens if we build the one-hot encoding of every automaton?

```
node left(a: bool) returns (b: bool);
  var p: bool;
  let b = a and (true -> pre p);
      p = (not(a)) and (true -> pre p);
  tel;

node right(b: bool) returns (c: bool);
  var r: bool;
  let c = b and (true -> pre r);
      r = (not b) and (true -> pre r);
  tel;
```

```
node product(a: bool) returns (c: bool);
  var b: bool;
  let b = left(a); c = right(b);
  tel;

node simple(a: bool) returns (c: bool);
  var pr: bool;
  let c = a and (true -> pre pr);
      pr = (not a) and (true -> pre pr);
  tel;

node observe(a: bool) returns (ok: bool);
  let ok = simple(a) = produit(a);
  tel;
```

```
% lesar auto.lus observe
--Pollux Version 2.3
TRUE PROPERTY
```

The tool lesar proves that ok is always true.

A final remark on causality

If the system has a instantaneous loop (a variable depends instantaneously on itself), the Lustre compiler rejects it.
Yet, some programs do have such loops but make sense as synchronous circuits, they are **constructively causal**. Feed with a constant input, their output stabilizes in bounded time. An example is:

$$x = mux(c, a, y) \wedge y = mux(c, x, b)$$

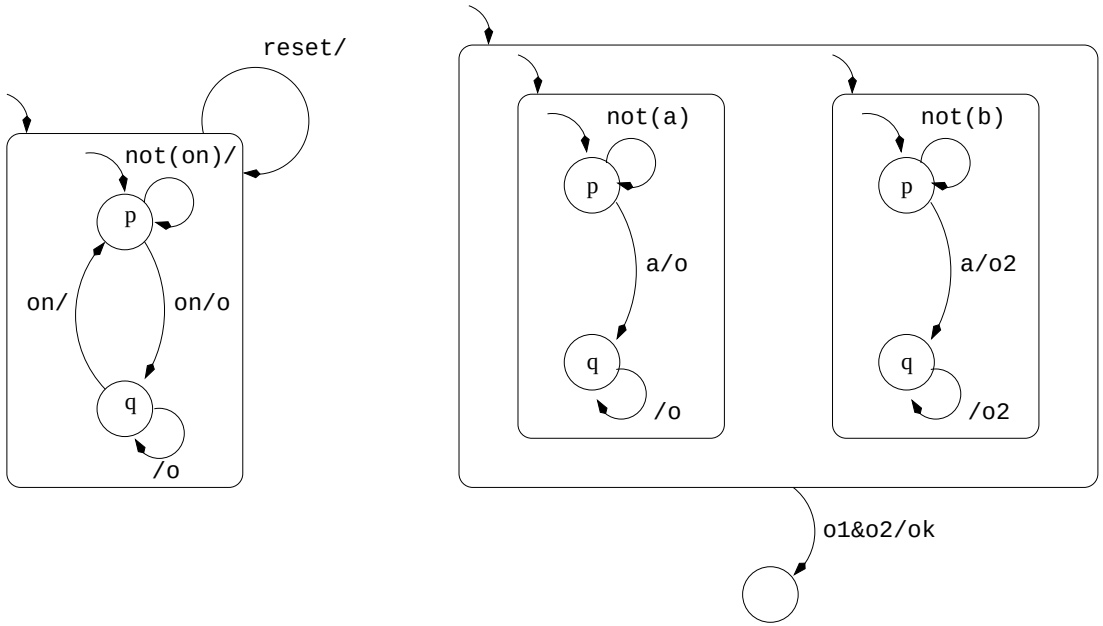
It is not valid as a Lustre program but constructively correct.
Constructive causality is a very interesting question (and would deserve a full extra course!).

Read

Michael Mendler, Tom Shiple, Gérard Berry. Constructive Boolean circuits and the exactness of timed ternary simulation Formal Methods in System Design, 2012; 40(3).

Hierarchical automata

- Introduced by David Harel in StateCharts.
- A state is itself an automaton.



But the semantics of StateCharts is difficult (more than 40 different).
We give here a synchronous interpretation to the hierarchical composition of boolean automata

Hierachical synchronous composition of boolean automata

- Weak preemption: the current reaction terminates.
- Synchronous semantics following the one proposed by Florence Maraninchi [**?**]

Hierarchical composition

Weak preemption = "la dernière cigarette du condamné".

If the source state s_k makes an internal transition $s_{k_1} \overset{f_k/\vec{o_k}}{\rightarrow} s_{k_2}$ at the same time with an external transition $s_k \overset{f(i)/\vec{o}}{\rightarrow} s_{k'}$, then signals $\vec{o_k}$ are emitted when $f_k \wedge f(i)$ is true.

Let two hierarchical states $M_1 = (S_1, s_o, I, O, T_1)$ and $M_2 = (S_2, s'_o, I, O, T_2)$ and a transition $s_k \overset{f(i)/\vec{o}}{\rightarrow} s_{k'}$.
Build an automaton $M = (S_1 + S_2, s_o, I, O, T)$ such that:

- $s_{k_1} \overset{f_k \wedge \mathtt{not}(f(i))/\vec{o_k}}{\rightarrow} s_{k_2}$ (if the transition is logically sound)
- $s_{k_1} \overset{f_k \wedge f(i)/\vec{o_k},\vec{o}}{\rightarrow} s'_o$ (if the transition is logically sound)

This 'flattened' semantics gives a precise meaning to parallel and hierarchical composition. Yet, it does not mean the composition must be computed explictly, e.g., for generating code.

Programs with non boolean values: interpreted automata

Equationnal model.

- $O = T_{o_1} \times \cdots \times T_{o_{|O|}}$
- $I = T_{i_1} \times \cdots \times T_{i_{|I|}}$
- $S = T_{s_1} \times \cdots \times T_{s_{|S|}}$ with $T_x = IB, IN, \ldots$
- Initial state $Init = (v_1, ..., v_{|S|})$
- A transition function $T : S \times I \rightarrow O \times S$

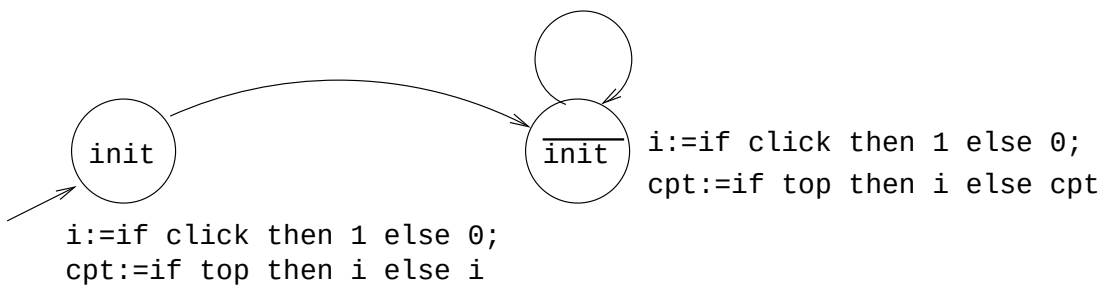
- Essentially Lustre.
- Explicit automaton: impossible to build (infinite set of states and transitions)

Interpreted automaton

- Finite control structure (boolean);
- transitions are labelled by conditions;
- equationnal model for the rest (integers, reals).

Example:

```
node compter(top, click: bool) returns (cpt: int);
  let
    cpt = if top then i else i + 0 -> pre cpt;
    i = if click then 1 else 0;
  tel;
```



Application: compilation of Esterel into boolean circuits

Consider the following specification:

ABRO

1. "every time A and B has been emitted, emit instantaneously O"
2. "reset the above behavior every time R is emitted"

This specification can be written in the following way, in the language Esterel.

```
every R do
  [ await A || await B];
  emit O;
end every
```

After simplification (i.e., translation into a kernel language), we get:

```
await R;
loop
  abort
   [ await A || await B ];
   emit O; halt
  when R
end
```

The Kernel Language

$$
\begin{array}{rcl}
p & ::= & \texttt{emit}\, s \mid p; p \mid P \| P \mid \texttt{loop}\, p \\
& & \mid \texttt{abort}\, p\, \texttt{when}\, s \mid \texttt{present}\, s\, \texttt{then}\, p\, \texttt{else}\, p \\
& & \mid \texttt{suspend}\, P\, \texttt{when}\, s \mid \texttt{nothing} \mid \texttt{signal}\, s\, \texttt{in}\, p \mid \texttt{halt}
\end{array}
$$

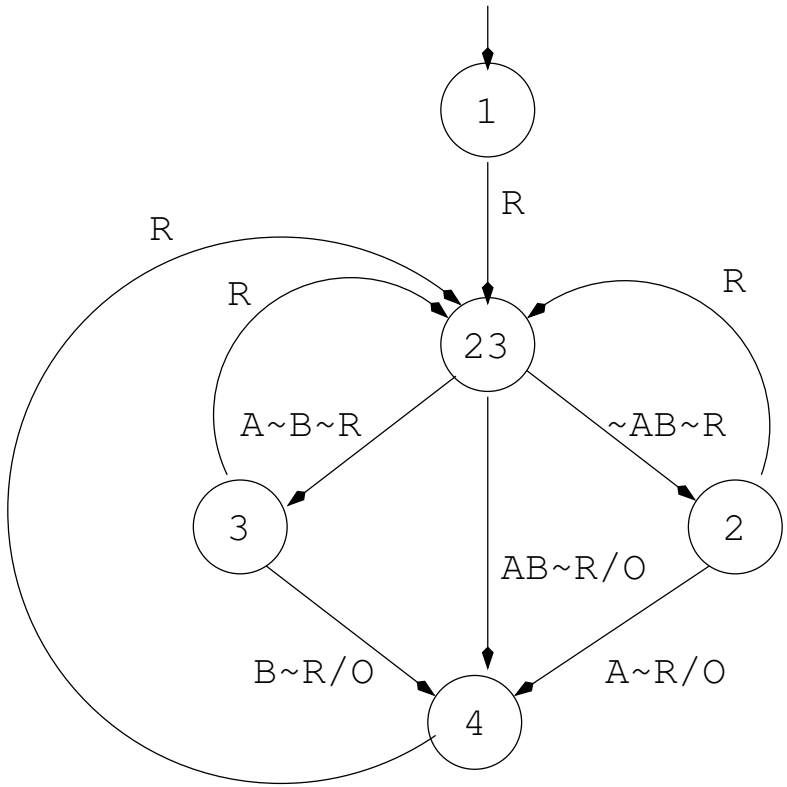
- $\texttt{pause} \equiv \texttt{await tick}$
- $\texttt{await S} \equiv \texttt{abort half when S}$
- $\texttt{halt} \equiv \texttt{loop pause end}$

An other minimal kernel only need `suspend` and `trap` (exception).
Cf. [G. Berry, *Preemption in Concurrent Systems*, FSTTCS'96].
We consider here only preemption which corresponds to a simple form of exception, together with suspension.

ABRO

The principle of the translation into a synchronous circuit is to represent the active control point in the program with a synchronous register.

```
await R; -- 1
loop
  abort
   [ await A -- 2
  || await B -- 3
  ];
   emit O;
   halt -- 4
  when R
end
```



Two different compilation methods for Esterel

- Explicit automaton obtained by symbolic evaluation of the operational semantics. Code size may be too big.
  Read: [G. Gonthier et G. Berry. The Esterel Synchronous Programming Language: Design, Semantics, Implementation, 1992]
- Implicit automaton, i.e., translation into boolean equations (circuits).
  Read: [F. Mignard. Compilation du langage Esterel en systèmes d'équations booléennes. Thèse de doctorat, 1994].

Compilation into circuits

Principe Every construct is translated into a system of boolean equations, i.e., a Lustre program.

- inputs $S_1, ..., S_n$; outputs $S'_1, ..., S'_k$.
- control inputs: *go* and *enable*
- control outputs: *term* and *halt*

Corresponds to a Lustre signature:

```
node f(go, enable: bool; S1,...,Sn:bool)
  returns (term, halt: bool; S'1,...,S'k: bool)
```

Translation rules

`emit S`

$$
\begin{aligned}
&term = go;\\
&halt = false;\\
&S = go
\end{aligned}
$$

`pause`

$$
\begin{aligned}
&term = false \texttt{ -> } \texttt{pre}\, go \texttt{ and } enable;\\
&halt = go;
\end{aligned}
$$

`await S`

$$
\begin{aligned}
&term = enable \texttt{ and } pwait \texttt{ and } S;\\
&halt = enable \texttt{ and } wait \texttt{ and not}(S);\\
&pwait = false \texttt{ -> pre}\,(go \texttt{ and } halt)
\end{aligned}
$$

p_1 `||` p_2

$$
\begin{aligned}
&(term_1, halt_1, S_1', ...) = p_1(go, enable, ...);\\
&(term_2, halt_2, S_2', ...) = p_2(go, enable, ...);\\
&halt = halt_1 \texttt{ or } halt_2;\\
&term = term_1 \texttt{ and } term_2;\\
&S' = S_1' \texttt{ or } S_2' ...
\end{aligned}
$$

Translation rules

$p_1 \,;\, p_2$

$$
\begin{aligned}
&(term_1, halt_1, S_1', \ldots) = p_1(go, enable, S_1, \ldots);\\
&(term_2, halt_2, S_2', \ldots) = p_2(term_2, enable \texttt{ and } \texttt{not}(halt_1), \ldots);\\
&halt = halt_1 \texttt{ or } halt_2;\\
&term = term_2;\\
&S' = S_1' \texttt{ or } S_2'
\end{aligned}
$$

$\texttt{abort}\ p\ \texttt{when}\ S$

$$
\begin{aligned}
&(term_1, halt_1, S_1', \ldots) = p_1(go, enable \texttt{ and } (\texttt{not}(S) \texttt{ or } go), \ldots);\\
&halt = halt_1 \texttt{ and } \texttt{not}(S);\\
&term = term_1 \texttt{ or } halt_1 \texttt{ and } S
\end{aligned}
$$

$\texttt{loop}\ p$

$$
\begin{aligned}
&(term_1, halt_1, S_1') = p_1(go \texttt{ or } term_1, enable, \ldots);\\
&term = false;\\
&halt = halt_1
\end{aligned}
$$

Translation rules

suspend p when S

$(term_1, halt_1, S_1', ...) = p_1(go, enable \texttt{ and } (\texttt{not}(S) \texttt{ or } go), S_1, ...);$
$halt = halt_1 \texttt{ or } (S \texttt{ and not}(go));$
$term = term_1$

Reincarnation:

```
loop
  signal S in
    [await T; emit S
  ||
    present S then emit O]
  end
end
```

Two different instances of S at the same time.

Reincarnation

Three solutions:

Solution 1

- Code duplication:

```
loop
  signal S1 in [await T; emit S1 || present S1 then emit O]
  signal S2 in [await T; emit S2 || present S2 then emit O]
end
```

  Expensive in size and efficiency.

Solution 2

- Do better by distinguishing “surface” and “depth”.
  - The surface of a programme is the part to be executed at the very first instant.
  - The depth is the complementary part.

$surface(\texttt{await}\ T) = \texttt{pause}$

$profondeur(\texttt{await}\ T) = \texttt{await immediate}\ T$

$surface(\texttt{present}\ S\ \texttt{then emit}\ O) = \texttt{present}\ S\ \texttt{then emit}\ O$

$profondeur(\texttt{present}\ S\ \texttt{then emit}\ O) = \texttt{nothing}$

```
loop
  signal S1 in [pause || present S1 then emit O] end;
  signal S2 in [await immediate T; emit S2 || nothing ] end;
end
```

To go further, read "Constructive semantics of Esterel" of G. Berry or (better), the book "Compiling Esterel, by Berry et al.".

Solution 3

Introduce an intermediate language with `gotopause` constructs.
Read "De la sémantique opérationnelle à la spécification formelle de compilateurs: l'exemple des boucles en Esterel". Thèse de doctorat, 2004. Olivier Tardieu.

The reincarnation problem is specific to Esterel.
It does not exists with mode automata and the solution adopted in SCADE 6 to mix data-flow and hierarchical automata.

Some PL Questions

In practice, real systems are mixed:

- some parts are purely data-flow: regulation systems, filters, etc.
- some are control-oriented: drivers, protocols, systems with modes, etc.

Whereas one can be encoded into the other, the result is not necessarily efficient. It is also necessary to be able to mix the two so that programs are easy to understand, read, modify.

The industrial language SCADE 6 mix the two. To go further, you can read:
"JL. Colaco, B. Pagano, M. Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In TASE, 2017"