Synchronous circuits, Automata, Parallel composition

Marc Pouzet
École normale supérieure
Marc.Pouzet@ens.fr

MPRI, October 13, 2014
In this course

- Equivalence between an explicit and implicit representation of a transition system (synchronous circuit *versus* an automaton).
- *One-hot* coding.
- Synchronous parallel and hierarchical composition of boolean automata.
- Translation of Esterel into data-flow equations.
- Some hints about the way data-flow and hierarchical automata are expressed and compiled into clocked data-flow equations in SCADE 6.
We focus on the model of synchronous programs.

- We restrict to boolean Lustre programs and pure Esterel.
- Two classical models of explicit automata:
  - Moore automata: the output is associated to a state.
  - Mealy automata: the output is associated to a transition.
- Explicit versus implicit representation of an automaton.
- Boolean automata and interpreted automata.
- Parallel composition, hiding, hierarchy.

Hierarchical automata are widely used: StateCharts [5], StateFlow \(^a\) but with semantics pitfalls.

Historically, Argos [6] was the first definition of hierarchical boolean automata with a precise semantics for composition. It is based on a synchronous semantics. SyncCharts [1], Esterel [2], SCADE 6 are also based on a synchronous semantics.

\(^a\)www.mathworks.com/products/stateflow
Moore Machines

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.
- $\Sigma$ is the finite input alphabet, $\Delta$ the output alphabet.
- $\delta$ is an application from $Q \times \Sigma$ to $Q$.
- $\lambda$ is an application from $Q$ to $\Delta$, that gives the output associated to every state.

The answer of $M$ to input $a_1a_2...a_n$, $n \geq 0$ 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 $\epsilon$. 
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 $\epsilon$, returns 0; for 1, returns 1; for 2 returns 2; for 5 returns 2 and for 10, returns 1.
Mealy Machines

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.
- $\Sigma$ is a input alphabet, $\Delta$ the output alphabet.
- $\delta$ is an application from $Q \times \Sigma$ to $Q$
- $\lambda$ is an application from $Q \times \Sigma$ to $\Delta$

$\lambda(q, a)$ 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 $\epsilon$, a Mealy automaton returns the output $\epsilon$. 
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 1 (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 1 (Equivalence)**

- If $M_1$ is a Moore automaton 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 $\Delta$. 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:

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

- An automaton can be represented by a set of boolean functions.
- That is, a circuit with logical operators (e.g., and, or gates) and registers.
- A set of inputs $I$, outputs $O$, memories $S$ (state variables).
- Initial state: $\vec{init} \in IB^{\lvert S \rvert}$
- Output functions: $o_j = f_j(\vec{s}, \vec{i}) \in IB$
- Transition functions: $s'_k = g_k(\vec{s}, \vec{i}) \in IB$

Particular case of a synchronous observer: a single boolean output.
Explicit automaton

• Set of inputs $I$, outputs $O$, states $Q$ (finite)
• Initial state: $q_{init} \in Q$
• Transition relation: $T \subseteq Q \times I \rightarrow O \times Q$

NB: Deterministic iff the relation is a function.

• Notation: $T(q, \vec{i}) = (\vec{o}, q')$ written $q \vec{i} / \vec{o} \rightarrow q'$

Mealy automaton:

• 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 $Q = IB^{|S|}$, $q_{init} = \vec{i}_{init}$

• $q \xrightarrow{\vec{i}/\vec{o}} q'$ if $\vec{o} = (f_1(q, \vec{i}), \ldots, f_{|O|}(q, \vec{i}))$
  
  $q' = (g_1(q, \vec{i}), \ldots, g_{|S|}(q, \vec{i}))$
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 $T : Q \times \Sigma \rightarrow \Delta \times Q$
- For any state $q$, write:
  - $Prec(q) = \{(p, i) \mid p \xrightarrow{i/o} q\}$
  - $Succ(q) = \{(i, r) \mid q \xrightarrow{i/o} r\}$
- Write $Output(o) = \{(p, q, i) \mid p \xrightarrow{i/o} q\}$
One hot-coding:

- A boolean state variable $s_{qj}$ per explicit state;
- a boolean variable $i_k$ per element of $\Sigma$;
- a boolean variable $o_k$ per element of $\Delta$.

- 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 $(p_k, i_k) \in \text{Prec}(q)$, $k \in \{1, \ldots, n\}$
  - Let $j_1, \ldots, j_m$ when there exists $r$ such that $(j_k, r) \in \text{Succ}(q)$, $k \in \{1, \ldots, m\}$

  \[
  s'_q = \begin{cases} 
  \text{if } s_q \text{ then } \neg j_1 \land \ldots \land \neg j_m \\
  \text{else } s_{p_1} \land i_1 \lor \ldots \lor s_{p_n} \land i_n 
  \end{cases}
  \]

  \[
  o = \bigvee_{(p, q, i) \in \text{Output}(o)} (p \land i)
  \]

- 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 a regular language

When an automaton is not deterministic, it may not be made deterministic first. A synchronous circuit is an excellent recogniser of a regular language!

![Automaton Diagram]

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;
    ok = q4;
  tel;

Several states can be active at the same instant. The code is deterministic and linear size without building the exponential power set.
Boolean automata and interpreted 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:

  \[ p^{f/o_1,\ldots,o_n} \rightarrow q \]

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

  \[ f ::= x \mid f \lor f \mid f \land f \mid \overline{f} \text{ où } x \in I \]

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

**Remark:** synchronous circuit composition (here a single global clock) = composition of boolean functions.

Boolean Mealy automata $M = (S, s_o, I, O, T)$ 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 $s \xrightarrow{b_i/\ldots} s'$ and $s \xrightarrow{b_j/\ldots} s''$, $b_i \land b_j = false$

**Reactivity:** For all state $s$, the set of transitions $s \xrightarrow{b_{i_1}/\ldots} s_{i_1}, 0 \leq i \leq k$ from $s$ verifies $\lor_{0 \leq i \leq k} b_i = true$

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

$$(p, q) \xrightarrow{c_1 \land c_2 / e_1, e_2} (p', q') \text{ if } (p \xrightarrow{c_1/e_1} p') \land (q \xrightarrow{c_2/e_2} q')$$

- Cartesian product of states with conjunction of guards 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 instantaneous (same instant).
Some conditions on transitions are unsound.

- $\overline{A}/A \rightarrow$: if $A$ is absent, is $A$ emitted? (all reaction must be logically sound)
- $A/A \rightarrow$: when $A$ is a local signal? (non determinacy)
- $A/\ldots \rightarrow$: where $A$ is local and not emitted? (a signal is present if it is emitted)
No communication/synchronisation

Communication and hiding

- 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.
**Hiding:** hide $bP$ ($b$ is a local signal in $P$)

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

\[(p, q)^{c_1 \land c_2/e_1, e_2} \rightarrow (p', q') \text{ if } (p \xrightarrow{c_1/e_1} p') \land (q \xrightarrow{c_2/e_2} q')\]

- some transition are logically unsound: keep transition $\xrightarrow{c/e}$ iff:

\[(b \in e \Rightarrow c \land b \neq false) \land (b \notin e \Rightarrow c \land \text{not} \ b \neq false)\]

- no logical contradiction during a reaction;
- then remove $b$ from transitions.

**Conclusion:**

- Iff $P$ and $Q$ are causal, $P \parallel Q$ is not necessarily causal.
- A static analysis, called **causality analysis** is used to ensure that the overall program is causal.
Automata and circuits:

What happens if we build the circuit corresponding to 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 produit(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
Causality

If the system has a instantaneous loop (a variable depends instantaneously on itself), it is statically rejected by the Lustre compiler.

Yet, some programs do have such loops but make sense mathematically. They are constructively causal: feed with a constant input, their output stabilizes in bounded time. An example is:

\[
y = \text{if } c \text{ then } x \text{ else } i2 \\
x = \text{if } c \text{ then } i1 \text{ else } y
\]

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:

Herarchical automata

• Introduced by David Harel in StateCharts.
• A state is itself an automata.

But the semantics of StateCharts is unclear (almost 40 different ones).

We consider here:

• Weak preemption: the current reaction terminates.
• Synchronous semantics following the one proposed by Florence Maraninchi [6] and implemented in Argos)
Synchronous semantics

Parallel composition Synchronous composition, eliminating unsound transitions.

Hierarchical composition

Weak preemption: is the source state $s_k$ makes an internal transition $s_k \overset{f_k/\vec{o}_k}{\rightarrow} s_{k2}$ 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 $f_k \land f(i)$ 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:

- $s_{k1} \overset{f_k \land \lnot f(i)/\vec{o}_k}{\rightarrow} s'_{k2}$ (if the transition is logically sound)
- $s_{k1} \overset{f_k \land f(i)/\vec{o}_k, \vec{o}}{\rightarrow} s'_o$ (if the transition is logically sound)
Programs with non boolean values: interpreted automata

Equationnal model.

- \( O = T_{o1} \times \ldots \times T_{o|O|} \)
- \( I = T_{i1} \times \ldots \times T_{i|I|} \)
- \( S = T_{s1} \times \ldots \times T_{s|S|} \) with \( T_x = IB, IN, \ldots \)
- Initial state \( Init = (v_1, \ldots, v_{|S|}) \)
- A transition function \( T : S \times I \to O \times S \)
- 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).

Essentially the result of a Lustre compiler.

**Example:**

```plaintext
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;
```

```
init
\[\text{i:=if click then 1 else 0; cpt:=if top then i else cpt}\]
```

```
\text{i:=if click then 1 else 0; cpt:=if top then i else i}
```
Programming Language Questions

In practice, systems are mixed.

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

The two descriptions are equivalent (one can be used to express/translate the other) mais the written code is not necessarily efficient nor easy to read.

**Questions** Find a unique language allowing to mix both kinds of descriptions. It is possible in all existing industrial tools: SCADE + SSM (before SCADE 6 was introduced; Simulink + StateFlow; etc.

- Mode automata (Maraninchi & al. [7]): automata whose states may contain automata or Lustre equations.
- An extension of this idea but with a source-to-source translation into a clocked data-flow kernel (Colaço et al. [4, 3]). Introduced first in Lucid Synchrone
- The basis of SCADE 6 \(^a\).

\(^a\)www.esterel-technologies/scade
Mode automata (Lucid Synchrone)

let node weak(up, down) = ok where
  rec automaton
    | Up -> do ok = true until down then Down
    | Down -> do ok = false until up then Up
  end

let node strong(up, down) = ok where
  rec automaton
    | Up -> do ok = true unless down then Down
    | Down -> do ok = false unless up then Up
  end

Compilation:

- Every automaton can be translated into a set of clocked data-flow equations with merge and when.
- Then, use the existing code generation method to produce sequential code.
**Application: compilation of Esterel into boolean circuits**

**Principle:** the control point is encoded by a boolean circuit.

**Example:** ABRO.

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

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

```plaintext
await R;
loop
    abort
        [ await A || await B ];
    emit 0; halt
    when R
end
```
The Kernel Language

\[ p ::= \text{emit } s \mid p; p \mid P || P \mid \text{loop } p \]
\[ \mid \text{abort } p \text{ when } s \mid \text{present } s \text{ then } p \text{ else } p \]
\[ \mid \text{suspend } P \text{ when } s \mid \text{nothing} \mid \text{signal } s \text{ in } p \mid \text{halt} \]

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

**Remark:** The two elementary construct of the kernel should be exception and suspension.

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.
await R; -- 1
loop
    abort
        [ await A -- 2
        || await B -- 3
    ];
    emit 0;
    halt -- 4
when R
end

Two different compilation methods for Esterel

- Explicit automaton obtained by symbolic evaluation of the operational semantics. Code may explode in size.
- Implicit automaton, i.e., translation into boolean equations (circuits).
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:

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

\[
\begin{align*}
  \text{term} &= \text{go}; \\
  \text{halt} &= \text{false}; \\
  S &= \text{go}
\end{align*}
\]

\text{pause}

\[
\begin{align*}
  \text{term} &= \text{false} \rightarrow \text{pre go and enable}; \\
  \text{halt} &= \text{go};
\end{align*}
\]

\text{await } S

\[
\begin{align*}
  \text{term} &= \text{enable and pwait and } S; \\
  \text{halt} &= \text{enable and wait and not}(S); \\
  \text{pwait} &= \text{false} \rightarrow \text{pre}(\text{go and hal}t)
\end{align*}
\]

$p_1 \parallel p_2$

\[
\begin{align*}
  (\text{term}_1, \text{halt}_1, S'_1, ...) &= p_1(\text{go, enable, }...); \\
  (\text{term}_2, \text{halt}_2, S'_2, ...) &= p_2(\text{go, enable, }...); \\
  \text{halt} &= \text{halt}_1 \lor \text{halt}_2; \\
  \text{term} &= \text{term}_1 \land \text{term}_2; \\
  S' &= S'_1 \lor S'_2...
\end{align*}
\]
Translation rules

\[ p_1; p_2 \]

\[
\begin{align*}
(\text{term}_1, \text{halt}_1, S'_1, ...) &= p_1(\text{go}, \text{enable}, S_1, ...); \\
(\text{term}_2, \text{halt}_2, S'_2, ...) &= p_2(\text{term}_2, \text{enable \ and \ not}(\text{halt}_1), ...); \\
\text{halt} &= \text{halt}_1 \ \text{or} \ \text{halt}_2; \\
\text{term} &= \text{term}_2; \\
S' &= S'_1 \ \text{or} \ S'_2
\end{align*}
\]

\textbf{abort} \ p \ \textbf{when} \ S

\[
\begin{align*}
(\text{term}_1, \text{halt}_1, S'_1, ...) &= p_1(\text{go}, \text{enable \ and} \ (\text{not}(S) \ \text{or} \ \text{go}), ...); \\
\text{halt} &= \text{halt}_1 \ \text{and} \ \text{not}(S); \\
\text{term} &= \text{term}_1 \ \text{or} \ \text{halt}_1 \ \text{and} \ S
\end{align*}
\]

\textbf{loop} \ p

\[
\begin{align*}
(\text{term}_1, \text{halt}_1, S'_1) &= p_1(\text{go \ or} \ \text{term}_1, \text{enable}, ...); \\
\text{term} &= \text{false}; \\
\text{halt} &= \text{halt}_1
\end{align*}
\]
Translation rules

suspend \( p \) when \( S \)

\[(term_1, halt_1, S'_1, ...) = p_1(go, enable \ and \ (\text{not}(S) \ or \ go), S_1, ...); \]

\[\text{halt} = \text{halt}_1 \ or \ (S \ and \ \text{not}(go)); \]

\[\text{term} = \text{term}_1 \]

Reincarnation:

loop
    signal \( S \) in
        [await \( T \); emit \( S \)
        | |
        present \( S \) then emit \( 0 \)]
    end
end

Two different instances of \( S \) at the same time.
Three solutions:

**Solution 1:**

- Code duplication:

  ```
  loop
    signal S1 in [await T; emit S1 || present S1 then emit 0] end;
    signal S2 in [await T; emit S2 || present S2 then emit 0] end;
  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.

  \[
  \text{surface}(\text{await } T) = \text{pause} \\
  \text{profondeur}(\text{await } T) = \text{await immediate } T \\
  \text{surface}(\text{present } S \text{ then emit } O) = \text{present } S \text{ then emit } O \\
  \text{profondeur}(\text{present } S \text{ then emit } O) = \text{nothing}
  \]
loop
    signal S1 in [pause || present S1 then emit 0] 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”.

Solution 3 Introduce an intermediate language with gotopause constructs.

Read “De la sémantique opérationnelle à la spécification formelle de compilateurs:

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, with no loss in expressiveness.


