A Conservative Extension of Synchronous Data-flow with State Machines

Marc Pouzet
LRI
Marc.Pouzet@lri.fr

Journées FAC
15 – 16 mars 2007
Toulouse

Joint work with Jean-Louis Colaço, Grégoire Hamon and Bruno Pagano
A Bit of History

Arround 1984, several groups introduced domain-specific languages to program/design control embedded systems.

- **Lustre** (Caspi & Halbwachs, Grenoble): data-flow (block diagram) formalisms with functional (deterministic) semantics;

- **Signal** (Benveniste & Le Guernic, Rennes): data-flow formalisms with relational (non-deterministic) semantics to model also under-specified systems;

- **Esterel** (Berry & Gonthier, Sophia): hierarchical automata and process algebra (and SCCS flavor)

All these languages were recognised to belong to the same family, sharing the same *synchronous model of time.*
The Synchronous Model of Time

- a global logical time scale shared by all the processes;
- every event can be tagged according to this global time scale;
- parallel processes all agree on the presence/absence of events during those instants;
- parallel process do not fight for resources (as opposed to time-sharing concurrency): $P \parallel Q$ means that $P$ and $Q$ (virtually) run in parallel;
- this reconcile parallelism and determinism

maximal reaction time $\max_{n \in \mathbb{N}}(t_n - t_{n-1}) \leq \text{bound}$
Extension Needs for Synchronous Tools

Around 1995, with Paul Caspi, we identified several “language” needs in synchronous tools

• modularity (libraries), abstraction mechanisms

• how to mix dataflow (e.g., Lustre) and control-flow (e.g., Esterel) in a unified way?

• language-based approach (vs verification) in order to statically guaranty some properties at compile time: type and clock inference (mandatory in a graphical tool), absence of deadlocks, etc.

• links with classical techniques from type theory (e.g., mathematical proof of programs, certification of a compiler)
The origins of Lucid Synchrone

What are the relationships between:

• Kahn Process Networks
• Synchronous Data-flow Programming (e.g., Lustre)
• (Lazy) Functional Programming (e.g., Haskell)
• Types and Clocks
• State machines and stream functions

What can we learn from the relationships between synchronous and functional programming?
Lucid Synchrone

Build a laboratory language to investigate those questions

- study extensions for SCADE/Lustre
- experiment things and write programs!
Milestones

- Synchronous Kahn Networks [ICFP’96]
- Clocks as types [ICFP’96]
- Compilation (co-induction \textit{vs} co-iteration) [CMCS’98]
- Clock calculus à la ML [Emsoft’03]
- Causality analysis [ESOP’01]
- Initialization analysis [SLAP’03, STTT’04]
- Higher-order and typing [Emsoft’04]
- Mixing data-flow and state machines [EMSOFT’05, EMSOFT’06]]
- N-Synchronous Kahn Networks [EMSOFT’05, POPL’06]
Some examples (V3)

- `int` denotes the type of integer streams,
- `1` denotes the (infinite) constant stream of 1,
- usual primitives apply point-wise

<table>
<thead>
<tr>
<th>c</th>
<th>t</th>
<th>f</th>
<th>t</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>...</td>
</tr>
<tr>
<td>y</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>...</td>
</tr>
<tr>
<td>if c then x else y</td>
<td>$x_0$</td>
<td>$y_1$</td>
<td>$x_2$</td>
<td>...</td>
</tr>
</tbody>
</table>
Combinatorial functions

Example: 1-bit adder

let xor x y = (x & not (y)) or (not x & y)

let full_add(a, b, c) = (s, co)
  where
    s = (a xor b) xor c
    and co = (a & b) or (b & c) or (a & c)

The compiler automatically infer the type and clock signature.

val full_add : bool * bool * bool -> bool * bool
val full_add :: 'a * 'a * 'a -> 'a * 'a
Full Adder (hierarchical)

Compose two “half adder”

\[ \text{let half_add}(a, b) = (s, co) \]

where

\[ s = a \text{ xor } b \]
and \[ co = a \text{ \& } b \]

Instantiate it twice

\[ \text{let full_add}(a, b, c) = (s, co) \]

where

\[ (s_1, c_1) = \text{half_add}(a, b) \]
and \[ (s, c_2) = \text{half_add}(c, s_1) \]
and \[ co = c_1 \text{ or } c_2 \]
Temporal operators

Operators $\text{fby, }\text{->, pre}$

- $\text{fby}$: unit initialized delay
- $\text{->}$: stream initialization operator
- $\text{pre}$: non initialized delay (register)

<table>
<thead>
<tr>
<th></th>
<th>$x$</th>
<th>$x_0$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$\ldots$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td></td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$\ldots$</td>
</tr>
<tr>
<td>$x$</td>
<td>$x_0$</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td>$\text{pre}$ $x$</td>
<td>$\text{nil}$</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$\ldots$</td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td>$x_0$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$\ldots$</td>
<td></td>
</tr>
</tbody>
</table>
Sequential functions

- Functions may depend on the past (the system has a state)
- Example: edge front detector

\[
\text{let node edge } x = x \rightarrow \text{not (pre } x) \& x
\]

\[
\begin{align*}
\text{val edge : bool } \Rightarrow \text{bool} \\
\text{val edge :: } 'a \rightarrow 'a
\end{align*}
\]

<table>
<thead>
<tr>
<th>x</th>
<th>t f t t</th>
<th>t f t f t f ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>edge x</td>
<td>t f t f</td>
<td>f f f f f f ...</td>
</tr>
</tbody>
</table>

In the V3, we distinguish combinatorial functions (\(\rightarrow\)) from sequential ones (\(\Rightarrow\))
Polymorphism (code reuse)

let node delay x = x \to\ pre x

val delay : 'a => 'a
val delay :: 'a -> 'a

let node edge x = false \to\ x <> pre x

val edge : 'a => 'a
val edge :: 'a -> 'a

In Lustre, polymorphism is limited to a set of predefined operators (e.g, if/then/else, when) and do not pass abstraction barriers.

Other features: higher-order, data-types, etc.

Question: How to mix data-flow and control-flow in an arbitrary way?
Designing Mixed Systems

Data dominated Systems: continuous and sampled systems, block-diagram formalisms
→ Simulation tools: Simulink, etc.
→ Programming languages: SCADE/Lustre, Signal, etc.

Control dominated systems: transition systems, event-driven systems, Finite State Machine formalisms
→ StateFlow, StateCharts
→ SyncCharts, Argos, Esterel, etc.

What about mixed systems?

• most system are a mix of the two kinds: systems have “modes”
• each mode is a big control law, naturally described as data-flow equations
• a control part switching these modes and naturally described by a FSM
Extending SCADE/Lustre with State Machines

SCADE/Lustre:

- data-flow style with synchronous semantics
- certified code generator

Motivations

- activation conditions between several “modes”
- arbitrary nesting of automata and equations
- well integrated, inside the same language (tool)
- in a uniform formalism (code certification, code quality, readability)
- be conservative: accept all Scade/Lustre and keep the semantics of the kernel
- which can be formally certified (to meet avionic constraints)
- efficient code, keep (if possible) the existing certified code generator
First approach: linking mechanisms

• two (or more) specific languages: one for data-flow and one for control-flow

• “linking” mechanism. A sequential system is more or less represented as a pair:
  – a transition function $f : S \times I \rightarrow O \times S$
  – an initial memory $M_0 : S$

• agree on a common representation and add some glue code

• this is provided in most academic and industrial tools

• PtolemyII, Simulink + StateFlow, Lustre + Esterel Studio SSM, etc.
An example: the Cruise Control (SCADE V5.1)
Observations

• automata can only appear at the leaves of the data-flow model: we need a finer integration

• forces the programmer to make decisions at the very beginning of the design (what is the good methodology?)

• the control structure is not explicit and hidden in boolean values: nothing indicate that modes are exclusive

• code certification?

• efficiency/simplicity of the code?

• how to exploit this information for program analysis and verification tools?

   Can we provide a finer integration of both styles inside a unique language?
Extending Synchronous Data-flow with Automata
[EMSOFT05]

Basis

- Mode-Automata by Maraninchi & Rémond [ESOP98, SCP03]
- SignalGTI (Rutten [EuroMicro95] and Lucid Synchrone V2 (Hamon & Pouzet [PPDP00])

Proposal

- extend a basic clocked calculus with automata constructions
- base it on a translation semantics into well clocked programs; gives both the semantics and the compilation method

Two implementations

- Lucid Synchrone language and compiler
- ReLuC compiler of SCADE at Esterel-Technologies; the basis of SCADE V6 (released in summer 2007)
Semantic principles

- only one set of equations is executed during a reaction
- two kinds of transitions: Weak delayed (“until”) or Strong (“unless”)
- both can be “by history” (H* in UML) or not (if not, both the SSM and the data-flow in the target state are reseted)
- at most one strong transition followed by a weak transition can be fired during a reaction
- at every instant:
  - what is the current active state?
  - execute the corresponding set of equations
  - what is the next state?
- forbids arbitrary long state traversal, simplifies program analysis, better generated code
Translation semantics into well-clocked programs

- use clocks to give a precise semantics: we know how to compile clocked data-flow programs efficiently
- give a translation semantics into the basic clocked data-flow language;
- clocks are fundamental here: classical one-hot (clock-less) coding (as done for circuits) does not allow to generate good sequential code afterwards
- type and clock preserving source-to-source transformation
  - \( T : \text{ClockedBasicCalculus} + \text{Automata} \rightarrow \text{ClockedBasicCalculus} \)
  - \( H \vdash e : ty \iff H \vdash T(e) : ty \)
  - \( H \vdash e : cl \iff H \vdash T(e) : cl \)
A clocked data-flow basic calculus

Expressions:

\[ e ::= C \mid x \mid e \text{ fby } e \mid (e, e) \mid x(e) \]
\[ \mid x(e) \text{ every } e \]
\[ \mid e \text{ when } C(e) \]
\[ \mid \text{merge } e (C \rightarrow e) \ldots (C \rightarrow e) \]

Equations:

\[ D ::= D \text{ and } D \mid x = e \]

Enumerated types:

\[ td ::= \text{type } t \mid \text{type } t = C_1 + \ldots + C_n \mid td; td \]

Basics:

- synchronous data-flow semantics, type system, clock calculus, etc.
- efficient compilation into sequential imperative code
N-ary Merge

merge combines two complementary flows (flows on complementary clocks) to produce a faster one:

```
.. a3 a2 a1
.. b7 b6 b5 b4 b3 b2 b1
```

introduced in Lucid Synchrone V1 (1996), input language of ReLuC

Example: merge c (a when c) (b when not c)

Generalization:

- can be generalized to $n$ inputs with a specific extension of clocks with enumerated types
- the sampling $e$ when $c$ is now written $e$ when $\text{True}(c)$
- the semantics extends naturally and we know how to compile it efficiently
- thus, a good basic for compilation
Reseting a behavior

- in Scade/Lustre, the "reset" behavior of an operator must be explicitly designed with a specific reset input

```plaintext
let node count () = s where
  rec s = 0 -> pre s + 1

let node resetable_counter r = s where
  rec s = if r then 0 else 0 -> pre s + 1
```

- painful to apply on large model

- propose a primitive that applies on node instance and allow to reset any node (no specific design condition)
Modularity and reset

Specific notation in the basic calculus: \( x(e) \)\ every c

- all the node instances used in the definition of node \( x \) are reseted when the boolean \( c \) is true

- the reset is “asynchronous”: no clock constraint between the condition \( c \) and the clock of the node instance

is-it a primitive construct? yes and no

- modular translation of the basic language with reset into the basic language without reset [PPDP00]

- essentially a translation of the initialization operator \( \rightarrow \)

- \( e_1 \rightarrow e_2 \) becomes if true \( \rightarrow c \) then \( e_1 \) else \( e_2 \)

- very demanding to the code generator whereas it is trivial to compile!

- useful translation for verification tools, basic for compilation

- thus, a good basic for compilation
Automata extension

- Scade/Lustre implicit parallelism of data-flow diagrams
- automata can be composed in parallel with these diagrams
- hierarchy: a state can contain a parallel composition of automata and data-flow
- each hierarchy level introduces a new lexical scope for variables
An example: the Franc/Euro converter

In concrete (Lucid Synchrone) syntax:

let node converter v c = (euro, fr) where
  automaton
    Franc -> do fr = v and eur = v / 6.55957
                 until c then Euro
    | Euro -> do fr = v * 6.55957 and eur = v
                until c then Franc
  end

Remark: fr and eur are shared flow but with only one definition at a time
**Strong vs Weak pre-emption**

Two types of transitions can be considered

let node converter v c = (euro, fr) where

automaton

  Franc -> do fr = v and eur = v / 6.55957
  unless c then Euro

  | Euro -> do fr = v * 6.55957 and eu = v
  unless c then Franc

end

- **until** means that the escape condition is executed after the body has been executed
- **unless** means that the escape condition is executed before and determines the active state of the reaction
Equations and Expressions in States

• every state defines the current value of a *shared flow*

• a flow must be defined only once per cycle

• the Lustre “pre” is local to its upper state (*pre* \( e \) gives the previous value of \( e \), the last time \( e \) was alive)

• the substitution principle of Lustre is still true at a given hierarchy \( \Rightarrow \) data-flow diagrams make sense!

• the notation \( \text{last } x \) gives access to the latest value of \( x \) in its scope (Mode Automata in the Maraninchi & Rémont sense)

• an absent definition for a shared flow \( x \) is implicitly complemented (i.e., \( x = \text{last } x \))
Mode Automata, a simple example

let node two_modes () = x where
rec automaton
  Up -> do x = 0 -> last x + 1
       until x = 5 continue Down
  | Down -> do x = last x - 1
        until x = -5 continue Up
end

Remark: replacing until by unless would lead to a causality error!
The Cruise Control with Scade 6
The extended language

\[ e ::= \cdots | \text{last } x \]

\[ D ::= D \text{ and } D | x = e \]

| match e with \( C \to D \) ... \( C \to D \)  
| reset \( D \) every e  
| automaton \( S \to u \ s \) ... \( S \to u \ s \)  

\[ u ::= \text{let } D \text{ in } u | \text{do } D \ w \]

\[ s ::= \text{unless } e \text{ then } S \ s | \text{unless } e \text{ continue } S \ s | \epsilon \]

\[ w ::= \text{until } e \text{ then } S \ w | \text{until } e \text{ continue } S \ w | \epsilon \]
Translation semantics

- several steps in the compiler, each of them eliminating one new construction
- must be preserve type (in the general sense)

Several steps

- compilation of the automaton construction into the control structures (match/with)
- compilation of the reset construction between equations into the basic reset
- elimination of shared memory last x
Translation

\[ T(\text{reset } D \text{ every } e) = \text{let } x = T(e) \text{ in } C\text{Reset}_{x} \ T(D) \]

\[ \text{where } x \notin \text{fv}(D) \cup \text{fv}(e) \]

\[ T(\text{match } e \text{ with } C_1 \rightarrow D_1 \ldots C_n \rightarrow D_n) = C\text{Match}(T(e)) \]

\[ (C_1 \rightarrow (T(D_1), \text{Def}(D_1))) \]

\[ \ldots \]

\[ (C_n \rightarrow (T(D_n), \text{Def}(D_n))) \]

\[ T(\text{automaton } S_1 \rightarrow u_1 \ s_1 \ldots S_n \rightarrow u_n \ s_n) = C\text{Automaton} \]

\[ (S_1 \rightarrow (T_{S_1}(u_1), T_{S_1}(s_1))) \]

\[ \ldots \]

\[ (S_n \rightarrow (T_{S_n}(u_n), T_{S_n}(s_n))) \]
Static analysis

- they should mimic what the translation does
- well typed source programs must be translated into well typed basic programs

Typing: easy

- check unicity of definition (SSA form)
- can we write \texttt{last }x\texttt{ for any variable?}
- No (in Lucid Synchrone): only shared variables can be accessed through a \texttt{last}
- otherwise, possible confusion with the regular \texttt{pre}

Clock calculus: easy under the following conditions

- free variables inside a state are all on the same clock
- the same for shared variables
- corresponds exactly to the translation semantics into \texttt{merge}
Initialization analysis

More subtle: must take into account the semantics of automata

let node two x = o where
  automaton
  S1 -> do o = 0 -> last o + 1
  until x continue S2
  | S2 -> do o = last o - 1 until x continue S1
end

o is clearly well defined. This information is hidden in the translated program.

let node two x = o where
  o = merge s (S1 -> 0 -> (pre o) when S1(s) + 1)
  (S2 -> (pre o) when S2(s) - 1)
  and
  ns = merge s (S1 -> if x when S1(s) then S2 else S1)
  (S2 -> if x when S2(s) then S1 else S2)
  and
  clock s = S1 -> pre ns
This program is not well initialized:

let node two x = o where
automaton
  S1 -> do o = 0 -> last o + 1
      unless x continue S2
  | S2 -> do o = last o - 1
      until x continue S1 end

• we can make a local reasoning
• because at most two transitions are fired during a reaction (strong to weak)
• compute shared variables which are necessarily defined during the initial reaction
• intersection of variables defined in the initial state and variables defined in the successors by a \textit{strong} transition
• implemented in Lucid Synchrone (soon in ReLuC)
New questions and extensions

A more direct semantics

- the translation semantics is good for compilation but...
- can we define a more “direct” semantics which expresses how the program reacts?
- we introduce a logical reaction semantics

Further extensions

- can we go further in closing the gap between synchronous data-flow and imperative formalisms?
- Parameterized State Machines: this provides a way to pass local information between two states without interfering with the rest of the code
- Valued Signals: these are events tagged with values as found in Esterel and provide an alternative to regular flows when programming control-dominated systems
Parameterized State Machines

• it is often necessary to communicate values between two states upon taking a transition

• e.g., a setup state communicate initialization values to a run state

  Setup \[\xrightarrow{\text{cond/x<-...}}\] Run

• can we provide a safe mechanism to communicate values between two states?

• without interfering with the rest of the automaton, i.e.,

• without relying on global shared variables (and imperative modifications) in states nor transitions?

Parameterized states:

• states can be Parameterized by initial values which can be used in turn in the target automaton

• preserves all the properties of the basic automata
A typical example

several modes of normal execution and a failure mode which needs some contextual information

let node controller in1 in2 = out where

automaton

| State1 ->
  do out = f (in1, in2)
  until (out > 10) then State2
  until (in2 = 0) then Fail_safe(1, 0)

| State2 ->
  let rec x = 0 -> (pre x) + 1 in
  do out = g (in1, x)
  until (out > 1000) then Fail_safe(2, x)

| Fail_safe(error_code, resume_after) ->
  let rec
    resume = resume_after -> (pre resume) - 1 in
  do out = if (error_code = 1) then 0
     else 1000
  until (resume <= 0) then State2
end
Parameterized states vs global modifications on transitions

Is all that useful?

- **expressiveness?** every parameterized state machine can be programmed with regular state machines using global shared flows

- **efficiency?** depends on the program and code-generator (though parameters only need local memory and are not all alive at the same time)

But this is bad!

- who is still using global shared variables to pass parameters to a function in a general-purpose language?

- passing this information through shared memory would mean having global shared variables to hold it

- they would receive meaningless values during normal execution and be set on the transition itself

- this breaks locality, modularity principles and is error-prone

- making sure that all such variables are set correctly before being use is not trivial
Parameterized states

• we want the language to provide a safer way to pass local information

• complementary to global shared variables and do not replace them

• keep the communication between two states local without interfering with the rest of the automaton

• do not raise initialization problems

• reminiscent to continuation passing style (in functional programming)

• yet, we provide the same compilation techniques (and properties) as in the case of unparameterized state machines (initialization analysis, causality, type and clocks)
Example (encoding Mealy machines)

- reduces the need to have equations on transitions
- adding equations on transitions is feasible but make the model awfully complicated

\[ | S(v) \rightarrow \text{do } o = v \text{ unless } c_1 \text{ then } T_1(o_1) \]
\[ \ldots \]
\[ \text{unless } c_n \text{ then } T_n(o_n) \]
\[ \ldots \]
end
Valued Signals and Signal Pattern Matching

- in a control structure (e.g., automaton), every shared flow must have a value at every instant
- if an equation for $x$ is missing, it keeps implicitly its last value (i.e., $x = \text{last } x$ is added)
- how to talk about absent value? If $x$ is not produced, we want it to be absent
- in imperative formalisms (e.g., Esterel), an event is present if it is explicitly emitted and considered absent otherwise
- can we provide a simple way to achieve the same in the context of data-flow programming?
An example

let node vend drink cost v = (o1, o2) where
  match v >= cost with
  true ->
    do emit o1 = drink
    and o2 = v - cost
    done
  | false ->
    do o2 = v done
  end

• o2 is a regular flow which has a value in every branch
• o1 is only emitted when (v >= cost) and is supposed to be absent otherwise
Accessing the value of a valued signal

• the value of a signal is the one which is emitted during the reaction

• what is the value in case where no value is emitted?

• Esterel: keeps the last computed value (i.e., implicitly complement the value with a register)
  
  \[
  \text{emit } S(\ ?A + 1) \\
  \]

  this is unsafe and raises initialization problems: what is the value if it has never been emitted?

• need extra methodology development rules to guard every access by a test for presence

  \[
  \text{present } A \text{ then } \ldots \text{ emit } S(?A + 1) \ldots \\
  \]

  provide a programming construct which forbid the access to a signal which is not emitted
Signal pattern matching

- a pattern-matching construct testing the presence of valued signals and accessing their content
- a block structure and only present value can be accessed

```latex
let node sum x y = o where
  present
  | x(v) & y(w) -> do emit o = v + w done
  | x(v1) -> do emit o = v1 done
  | y(v2) -> do emit o = v2 done
  | _ -> do done
end
```
Signals as existential clock types

let node sum x y = o where
  present
  | x(v) & y(w) -> do emit o = v + w done
  | x(v1) -> do emit o = v1 done
  | y(v2) -> do emit o = v2 done
  | _ -> do done
end

• o is partially defined and should have clock \( ck \) on \( (?x \land ?y) \lor ?x \lor ?y \) if \( x \) and \( y \) are themselves on clock \( ck \)

• giving it the existential type \( \Sigma (c : ck).ck \) on \( c \), that is, "exists \( c \) on clock \( ck \) such that the result is on clock \( ck \) on \( c \) is a correct abstraction"
Clock type of a signal: a dependent pair \( ck \text{ sig} = \Sigma (c : ck).ck \text{ on } c \) made of:

- a boolean sequence \( c \) which is itself on clock type \( ck \)
- a sequence sampled on \( c \), that is, with clock type \( ck \) on \( c \)

The flow is boxed with its presence information

- this is a restriction compared to what can provide a synchronous data-flow language equipped with a powerful clock calculus
- but this is the way Esterel valued signal are implemented
- reminiscent to the constraints in Lustre to return the clock of a sampled stream

Clock verification (and inference) only need modest techniques

- box/unbox mechanisms of a Milner type system + extension by Laufer & Odersky for abstract data-types

\[ H \vdash e : ck \text{ on } c \]

\[ \frac{} {H \vdash \text{emit } x = e : [x : ck \text{ sig}]} \]
Translation Semantics

- parameterized state machines and signals can be combined in an arbitrary way
- a translation semantics of the extension into a basic language

Example

let node sum (a, b, r) = o where
  automaton
  | Await -> do unless \(a(x) \& b(y)\) then Emit \((x + y)\)
  | Emit \((v)\) -> do emit \(o = v\) unless \(r\) then Await
• a signal of type $t$ is represented by a pair of type $\text{bool} \times t$

• $\text{nil}$ stands for any value with the right type (think of a local stack allocated variable)

let node sum $(a, b, r) = o$ where

match $\text{pnextstate}$ with
| $\text{Await} \to$ match $(a, b)$ with
  | $((\text{True}, x), (\text{True}, x)) \to$ $\text{state} = \text{Emit}(x + y)$
  | _ $\to$ $\text{state} = \text{Await}$
| $\text{Emit}(v) \to$ match $r$ with
  | $\text{true} \to$ $\text{state} = \text{Await}$
  | $\text{false} \to$ $\text{state} = \text{Emit}(v)$

and

match $\text{state}$ with
| $\text{Await} \to$ $o = (\text{False}, \text{nil})$ and $\text{nextstate} = \text{Await}$
| $\text{Emit}(v) \to$ $o = (\text{True}, \text{nil})$ and $\text{nextstate} = \text{Emit}(v)$

and

$p\text{nextstate} = \text{Await} \to$ pre $\text{nextstate}$
Conclusion

• An extension of a data-flow language with automata constructs
• various kinds of transitions, yet quite simple
• translation semantics relying on the clock mechanism which give a good discipline
• the existing code generator has not been modified and the code is (at least as) efficient than direct ad-hoc techniques
• fully implemented in Lucid Synchrone; integration in Scade 6 is under way
• distribution and documentation: www.lri.fr/~pouzet/lucid-synchrone
References

