Functional Synchronous Programming of Reactive Systems

Marc Pouzet
LRI

Marc.Pouzet@lri.fr

Seminar Tao, July 4, 2006
Embedded Software

Evolution since the 70’s

• Moore law: the computing power double every 18 months
• 2006: 500 000 000 transistors (Pentium), 4Ghz
• Especially: small dedicated processors everywhere!
  – consumer electronics: TV, video, mobile phones
  – automotive: electronic systems, ABS, braking systems, electronic key
• dedicated circuits (ASIC), dedicated or general purpose computers executing software
• now: dynamic reconfiguration, cold (ADSL modem) or hot (industrial systems, mobile phones)
A progressive transition

- first in **industrial fields**
  - replacing mechanical, electro-mechanical, electronic systems
  - from relay systems (train) to software systems
  - flight commands (from direct mechanics, electro-mechanic support (Caravelle), analog systems (Concorde) to numerical and software (Airbus 320, 1988))
  - regulation systems, control/command of industrial processes

- almost everywhere now
  - heater controller, washing machine, TV boxes, etc.

- and it is possible to **simulate** the whole before building it
Embedded real-time systems: characteristics

Hard Real-time Constraints

- the system is submitted to hard real-time constraints
- **imposed by the environment** (physics does not wait!)

→ (statically) bounded response time and memory

Heterogeneous environment

- **continuous** physical environment: temperature sensors, activators
- and/or **discrete**: button, threshold, other computers
- a huge number of input/outputs

→ the formalism must be able to express this heterogeneity
Concurrent and Deterministic Systems

- **closed loop** systems: a heater controller and the heater itself run in parallel
- concurrency is the natural way for **composing systems**: *control at the same time rolling and pitching*
- the whole system must stay **deterministic**
- also **simpler**: reproducibility, easier debugging, simulation

→ the formalism must be able to compose sub-systems in parallel
→ the model must conciliate concurrency and determinism

Safety is important

- **critical** systems: fly-by-wire, braking systems, airbags, medical systems
- some systems do not have a stable (safe) position (plane?)

→ properties must be guaranteed statically: “dynamic” = “too late”
→ languages with a well founded semantics, formal validation
Design Domain Specific Languages

• the general-purpose software model is in-adapted: Turing complete, too expressive, too hard to verify

• complexity is not where it is needed: pointer arithmetics, dynamic allocation, etc.

• do we need all that?

• concurrency and determinism are absent but there are fundamental!

• far from the mathematical model of the engineer

• design specific languages with a limited expressive power, a formal semantics, well adapted to the culture of the field

• this culture is rich:
  – continuous control (control theory, sampling, etc.)
  – discrete control (automata theory, etc.)
Continuous control (control theory, signal processing...)

- A signal/event is represented by a discrete sequence (a *stream*).
  - Stream equations, generating functions, z-transform
  - Graphical formalism (block-diagrams)

- Manual transcription of these equations into imperative programs
- Hard and error-prone

\[
Y_0 = bX_0, \quad \forall n \quad Y_{n+1} = aY_n + bX_{n+1}
\]

The idea of Lustre (and Signal) (1984):

- Program directly with these stream equations
- Provide a compiler and tool analysis
Discrete Control

- transition systems (automata), Mealy/Moore machines
- synchronous composition of automata + hierarchy, etc.
- process calculi (Milner’s SCCS)

Esterel (82):

- propose a high-level language for concurrent systems
- based on the synchronous composition
- preserving determinism (causality problems are solved)
The Synchronous Model of Time

- these languages are based on the **zero delay** model
  - time is **logical** as the sequence of atomic reaction of the system to input events
  - the system is the **parallel composition** of sub-systems which (virtually) execute in parallel
  - check afterwoods the **correspondence** between logical time and physical time: is the machine fast enough?

maximum response time $\max_{n \in \mathbb{N}}(t_n - t_{n-1}) \leq \text{bound}$
Synchronous Languages

• based on a common model but with different programming styles

• imperative (automata): Esterel, SyncCharts, Argos

• declarative (dataflow): Lustre, Signal

• industrial compiler (and environment) SCADE/Lustre, Esterel-studio (Esterel-Technologies), Signal/Sildex (TNI)

• formal semantics, hardware and software compilation of the same description, test tools and automatic verification

• automatic distribution (distributed architectures), (real-time) multi-tasking, etc.

• industrial success: avionics (Airbus, Dassault, Eurocopter), ground transportation (Matra, Audi), circuits (Xilinx, TI, Intel)
But also...

- simulation tools: Simulink/StateFlow (The MathWorks), Catia (Dassault-Systèmes), etc.
- very rich platform to simulate the whole system and its environment
- based on numerical analysis techniques, simulation techniques
- (partial) code generation, verification tools, etc.
- these tools are not that far from synchronous tools
- block-diagram description à la SCADE with Simulink; state-transition systems à la SyncCharts with StateFlow...
- but they have not been designed with a programming discipline in mind (where what is executed is exactly what is modeled and simulated)
- informal semantics (code certification, good code quality?), formal proof/verification tools?
- what is simulated and what is executed must be the same!
Software Factory with Catia + LCM (Dassault-Systèmes)
Needs for Synchronous Tools and Models

- master the complexity and large scale systems
  - what to do with all these transistors?
  - critical systems become big: 500 000 lines of code for the fly-by-wire command of the A380
  - some companies only specify the system and assemble the code made by others

- modularity (libraries), abstraction mechanism

- "langage" tools (vs verification) which give guaranty at compile time: automatic type and clock inference (mandatory in a graphical environment), deadlock freedom, etc.

- how to combine dataflow (e.g., Lustre) and control-flow (e.g., Esterel) in a uniform way?

- links with tools for formal certification (code certification is mandatory in civil avionics, DO 178B norm)

- code certification, link with proof assistant
The origin of Lucid Synchrone

In 95, with Paul Caspi (VERIMAG)

What are the relationships between:

- Kahn process networks
- synchronous data-flow programming (e.g., Lustre)
- tools and models of control theory/signal processing
- lazy functional programming (e.g., Haskell)
- types and clocks
- state machines and stream functions

What can we learn from bringing together synchronous programming and functional programming?
Lucid Synchrone

Build a “laboratory” language

• study the extensions of Lustre (synchronous and functional)
• experiment things and write programs!
Semantics

• Synchronous Kahn networks [ICFP’96]
• Clocks as dependent types [ICFP’96]
• synchronous stream functions and transition systems (co-induction vs co-itération) [CMCS’98]
• ML-like clock calculus [Emsoft’03]
• causality analysis [ESOP’01]
• initialization analysis [SLAP’03, STTT’04]
• higher-order and typing [Emsoft’04]
• data-flow and state machines [Emsoft’05]
• N-Synchronous Kahn Networks [Emsoft’05, POPL’06]
Some examples (V3)

- **int** denote the type of streams of integers,
- **1** denotes an (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>if c then x else y</td>
<td>x₀</td>
<td>y₁</td>
<td>x₂</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>x₀</td>
<td>x₁</td>
<td>x₂</td>
<td>...</td>
</tr>
<tr>
<td>y</td>
<td>y₀</td>
<td>y₁</td>
<td>y₂</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 computes 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”

let half_add(a,b) = (s, co)
    where
        s = a xor b
        and co = a & b

Instanciate twice

let full_add(a,b,c) = (s, co)
    where
    rec (s1, c1) = half_add(a,b)
    and (s, c2) = half_add(c, s1)
    and co = c1 or c2
Sequential Functions

Operators $\texttt{fby}$, $\rightarrow$, $\texttt{pre}$

- $\texttt{fby}$: unitary (initialized) delay
- $\rightarrow$: initialization
- $\texttt{pre}$: un-initialized delay (register in circuits)

\[
\begin{array}{|c|c|c|c|}
\hline
x & x_0 & x_1 & x_2 & \cdots \\
\hline
y & y_0 & y_1 & y_2 & \cdots \\
\hline
x \texttt{fby} y & x_0 & y_0 & y_1 & \cdots \\
\hline
\texttt{pre} x & \texttt{nil} & x_0 & x_1 & \cdots \\
\hline
x \rightarrow y & x_0 & y_1 & y_2 & \cdots \\
\hline
\end{array}
\]
Sequential Functions

• Stream functions may depend on the past (statefull systems)

• Example: edge front detector

  let node edge x = x -> not (pre x) & x

  val sum : int => int
  val sum :: 'a -> 'a

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

In V3, we distinguish combinatorial function (->) from sequential functions (=>)
Polymorphism (code reuse)

let node delay x = x -> pre x

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

let node edge x = false -> x <> pre x

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

In Lustre, polymorphism is limited to a set of predefined operators (e.g., \texttt{if/then/else}, \texttt{when}) and does not pass abstraction.
Library and Curryfication

(* module Numerical *)

let node integr dt x0 dx = x where
  rec x = x0 -> pre x +. dx *. dt

val integr : float -> float -> float => float
val integr :: 'a -> 'a -> 'a -> 'a

(* module Main *)

let static dt = 0.001

let integr = integr dt

val integr : float -> float => float
val integr :: 'a -> 'a -> 'a -> 'a
Example: the inverted pendulum

Specification: control an inverted pendulum

\[ l \frac{d^2 \theta}{dt^2} = \sin(\theta) \left( \frac{d^2 y_0}{dt^2} + g \right) - \left( \cos(\theta) \frac{d^2 x_0}{dt^2} \right) \]

\[ x = x_0 + l \sin(\theta) \]

\[ y = y_0 + l \cos(\theta) \]
Main module:

Constants:

let static dt = 0.001 (* sampling step *)
and static l = 10.0 (* length *)
and static g = 9.81 (* acceleration *)

(* partial application with fixed step *)
let integr = Numerical.integr dt
let deriv = Numerical.deriv dt

The equation of the pendulum

let node pendulum d2x0 d2y0 = theta where
  rec theta = integr (integr ((sin thetap)*.(d2y0 +. g)
     -. (cos thetap)*.d2x0)/.l)
  and thetap = 0.0 -> pre theta
Reject programs

Reject program which cannot be executed sequentially

let node pendul d2x0 d2y0 = theta
  where rec theta =
      integr (integr ((sin theta) *. (d2y0 +. g)
        --
        -. (cos theta) *. d2x0) /. l)

thetap depends instantaneously on itself

- a “syntactical” criteria: a recursion must cross a delay
- a type system, with Pascal Cuoq [ESOP’01]
- thus, with type signatures (interfaces)
- modular and higher-order
Reject programs

Reject program for which the result depend on the initial value of some delays

let node pendul d2x0 d2y0 = theta
where rec theta =
  integr (integr ((sin (pre theta))*.d2y0 +. g)
  -.(cos (pre theta))*.d2x0)/.1)
this expression may not be initialized

• 1-bit abstraction

• a type system (with sub-typing rules), with JL-Colaço from Esterel-Technologies [SLAP’02, STTT’04]

• works well for SCADE

• tested on real-size examples (75000 lines) at Esterel-Tech.
Clocks: mix several time-scale

- mix slow and fast processes in a safe way?
- multi-sampled systems (software), multi-clock (hardware)
- introduced in Lustre and Signal at the very beginning
- also present in Simulink (periodic systems)

In **Lucid Synchrone**, a clock is a type and is automatically inferred
Two operators

when (under-sampling) and merge (over-sampling)

<table>
<thead>
<tr>
<th></th>
<th>t</th>
<th>t</th>
<th>f</th>
<th>f</th>
<th>t</th>
<th>f</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>$x_5$</td>
<td>...</td>
</tr>
<tr>
<td>x when c</td>
<td>$x_0$</td>
<td>$x_1$</td>
<td></td>
<td></td>
<td>$x_4$</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>x when not c</td>
<td></td>
<td></td>
<td>$x_2$</td>
<td>$x_3$</td>
<td></td>
<td>$x_5$</td>
<td>...</td>
</tr>
<tr>
<td>y</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td></td>
<td></td>
<td>$y_2$</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>merge c y (x when not c)</td>
<td>$y_0$</td>
<td>$y_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$y_2$</td>
<td>$x_5$</td>
<td>...</td>
</tr>
</tbody>
</table>
Example

let node sum x = s where rec s = x -> pre s + x
let node sampled_sum x c = sum (x when c)

val sampled_sum : int -> bool => int
val sampled_sum :: 'a -> (_c0:'a) -> 'a on _c0

let clock ten = count 10 true
let node sum_ten x = sampled_sum x ten

val ten : bool
val ten :: 'a
val sum_ten : int => int
val sum_ten :: 'a -> 'a on ten
Over-sampling

• Define systems whose internal rate is faster that the rate of their inputs?
• express temporal constraints, scheduling, resources

Example: Computation of $x^5$

let node power x = x * x * x * x * x

let clock four = count 4 true
let node spower x = y where
  rec i = merge four x ((1 fby i) whenot four)
  and o = 1 fby (i * merge four x (o whenot four))
  and y = o when four

val power :: 'a -> 'a
val spower :: 'a on four -> 'a on four
Property: 1 fby (power x) and spower x are observationally equivalent
The computation of \((x_n \& x_{2n})_{n \in \mathbb{IN}}\) is not real-time

\[
\begin{align*}
\text{let } \text{odd } x &= x \text{ when half} \\
\text{let } \text{non\_synchronous } x &= x \& (\text{odd } x)
\end{align*}
\]

This expression has clock ’a on half, but is used with clock ’a.

**Execution with unbounded FIFOs!!!**

- clocks = an information about the behavior of streams
- clocks = types
- the **merge** and type based clock calculus is reused in the ReLuC compiler of SCADE
Higher-order

Iteration:

\[
\text{let node it } f z x = y \\
\text{where rec } y = f x (\text{init fby } y)
\]

\[
\text{val it : ('b -> 'a -> 'a) -> 'a -> 'b => 'a}
\]

\[
\text{val it :: ('b -> 'a -> 'a) -> 'a -> 'b -> 'a}
\]

Example:

\[
\text{let node sum x = it (+) 0 x}
\]

\[
\text{let node mult x = it (*) 1 x}
\]
Mixed systems (data-flow + automata)

Data-dominated systems: sampled systems, block-diagram formalisms
  ← Simulation tools: Simulink, etc.
  ← Programming languages: Scade/Lustre, Signal, etc.

Control dominated systems: transition systems, event systems, automata formalisms
  ← StateFlow, StateCharts
  ← SyncCharts, Argos, Esterel, etc.

What about mixed designs?

- real systems are a mix of both styles: systems have running modes
- each mode is defined by a control law, naturally written with data-flow equations
- a transition system for switching between these modes
Extend SCADE/Lustre with state machines

Existing solutions

- two (or more) specific languages: one for the data part, one for the control part
- “linking” mechanisms: a sequential system is always more or less of the form
  - a transition function $f : S \times I \rightarrow O \times S$
  - an initial memory $M_0 : S$
- agree on a common representation + glue
- exist in most academic or industrial tools
- PtolemyII, Simulink + StateFlow, Lustre + Esterel Studio SSM, etc.
An example: the cruise control (SCADE V4.2)
Observations

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

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

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

• code certification?

• efficiency/simplicity of the code?

• how to exploit this information in static analysis and verification tools?
The Approach

- extend a synchronous data-flow language (Lustre) with automata constructs
- base it on a unified theory of synchronous systems
- produce efficient code (which compete with ad-hoc techniques)
- efficient compilation techniques, conservative (accept all SCADE/Lustre)

Two implementations

- ReLuC compiler of SCADE at Esterel-Tech.
- Lucid Synchrone V3
A simple example: the Franc/Euro converter

In *Lucid Synchrone* syntax:

```plaintext
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 eu = v
             until c then Franc
end
```
The Cruise Control (SCADE V6)
Other Examples

• the cruise control
• the heater
• the (Milner) coffee machine
• approximation methods (Euler, Runge-Kutta)
Laboratory language

Collaboration with the SCADE team since 1999

- the ReLuC compiler of SCADE is based (and improves) techniques introduced in Lucid Synchrone
- typing, clock calculus
- some constructions (e.g., merge)
- static analysis (initialization)
- design/semantics of SCADE V6

Collaboration with Athys (Dassault-Systèmes) for the integration of a programming environment into the Catia suite for industrial systems (LCM)

- automatic type synthesis (with polymorphism)
- other type-based analysis
Conclusion and Future Works

Compilation, semantics

- other extensions, program analysis, etc.
- certified compilation (for software and hardware), proof assistant tools

Relaxed Synchrony for Video Systems

- relax (a little) the clock calculus in order to compose non strictly synchronous systems but which can be synchronized through the insertion of buffers
- model of N-Synchronous Kahn Networks [Emsoft’05, POPL’06]
- with the Alchémy project (INRIA) and Philips NatLabs

Take Physical Resources into Account

- how to model real (physical) time, resources?
- how to compile synchrony on a pipelined machine (or a compiled parallel machine)?