A Conservative Extension of Synchronous Data-flow with State Machines

Jean-Louis Colaço, Bruno Pagano
Esterel-Technologies (France)

Marc Pouzet
Université Paris-Sud (France)

Workshop SYNCHRON, Malta
Nov. 24th, 2005
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 Sudio SSM, etc.
An example: the Cruise Control (SCADE V4.2)
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?
Second approach: designing a “language” extension

Mode automata (Lustre): Maraninchi & Rémond [ESOP98, SCP03]

- Lustre + automata: states are made of Lustre equations
- specific compilation method, generates good code
- restriction on the Lustre language, on the type of transitions

SignalGTI (Signal): Eric Rutten [EuroMicro95]

- control structures encoded with clocks
- all Signal, uniformity with the basic language, no specific compilation
- too demanding for the compiler, hard to program with

Lucid Synchrone V2: Hamon & Pouzet [PPDP00,SLAP04]

- extend Lustre with a modular reset, no restriction
- rely on the clock mechanism to express control structures in a safe way
- no particular syntax (manual encoding of automata), hard to program with
Our Proposal

- extend a basic clocked calculus (Lustre) with automata constructions
- founded on the use of *clocks*
- provide an ad-hoc syntax and follow a *clock directed approach*
- essentially a source-to-source compilation

Two implementations

- ReLuC compiler of Scade at Esterel-Technologies
- Lucid Synchrone language and compiler
**Principles**

- do not ask too much to a compiler: only provide automata constructs which compile well
- keep things simple: one definition of a flow during a reaction, one active state, substitution principle
- use clocks to give a precise semantics: we know how to compile clocked data-flow programs efficiently
- give a translation semantics into the basic data-flow language
- type and clock preserving source-to-source transformation
  - \( T : \text{ClockedBasicCalculus + Automata} \rightarrow \text{ClockedBasicCalculus} \)
  - \( H \vdash e : \text{ty} \) iff \( H \vdash T(e) : \text{ty} \)
  - \( H \vdash e : \text{cl} \) iff \( H \vdash T(e) : \text{cl} \)
  - the same for causality, initialization analysis
A clocked data-flow basic calculus

Expressions:

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

Equations:

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

Enumerated types:

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

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 whennot c)`

**Generalization:**

- can be generalized to *n* inputs with a specific extension of clocks with enumerated types
- the sampling *e when c* is written *e when 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 explicitely designed with a specific reset input

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: \( x(e) \text{ every } c \)

- 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 \( \text{if } c \text{ then } e_1 \text{ else } e_2 \)

- 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

\[
\begin{align*}
\text{Franc} & \quad \text{Euro} \\
fr &= v; \\
eu &= v/6.77957; \\
\text{fr} &= v*6.55957; \\
eu &= v;
\end{align*}
\]

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
Features

Semantic principles:

- only one transition can be fired per cycle
- only one active state per automaton, hierarchical level and cycle

Transitions and states

- two kinds: Strong or Weak delayed

- 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)
Strong vs Weak Preemption

let node weak_switch on_off = o where
  automaton
    False -> do o = false until on_off then True
  | True -> do o = true until on_off then False
end

let node strong_switch on_off = o where
  automaton
    False -> do o = false unless on_off then True
  | True -> do o = true unless on_off then False
end
Equations and expressions in states

• flows are defined in the states (state actions)

• a flow must be defined only once per cycle

• the “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 last \( x \) gives access to the latest value of \( x \) in its scope (Mode Automata in the Maraninchi & Rémond sense)
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
The Cruise Control with Scade 6
The extended language

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

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

\[
| \text{match } e \text{ with } C \to D \ldots C \to D
\]

\[
| \text{reset } D \text{ every } e
\]

\[
| \text{automaton } S \to u s \ldots S \to u s
\]

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

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

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

• several steps in the compiler, each of them eliminating one new construction
• must preserve types (in the general sense)

Several steps

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

\[ T(\text{reset } D \text{ every } e) = \begin{array}{c} \text{let } x = T(e) \text{ in } CReset_x \ T(D) \\ \text{where } x \notin \text{fv}(D) \cup \text{fv}(e) \end{array} \]

\[ T(\text{match } e \text{ with } C_1 \to D_1 \ldots C_n \to D_n) = CMatch \ (T(e)) \]
\[ (C_1 \to (T(D_1), \text{Def}(D_1))) \]
\[ \ldots \]
\[ (C_n \to (T(D_n), \text{Def}(D_n))) \]

\[ T(\text{automaton } S_1 \to u_1 \ s_1 \ldots S_n \to u_n \ s_n) = CAutomaton \]
\[ (S_1 \to (T_{S_1}(u_1), T_{S_1}(s_1))) \]
\[ \ldots \]
\[ (S_n \to (T_{S_n}(u_n), T_{S_n}(s_n))) \]
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 `last x` for any variable?
- No (in Lucid Synchrone): only shared variables can be accessed through a `last`
- otherwise, possible confusion with the regular `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 `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 reasonning

• 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)
Conclusion and Future work

• 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 (surprisingly) efficient

• fully implemented in Lucid Synchrone (next release V3)

• integration in Scade 6 is under way

• adding pure and valued signals, final states, etc.

• formal certification of the translation inside a proof assistant