Mixing Signals and Modes in Synchronous Data-flow Systems

Marc Pouzet
LRI

Joint work with Jean-Louis Colaço and Grégoire Hamon (see [Emsoft 06])

Synchron, 29/11/2006
Designing Mixed Systems

**Data dominated Systems:** continuous and sampled systems, block-diagram formalisms, data-flow equations

→ Simulation tools: Simulink, etc.

→ Programming languages: SCADE/Lustre, Signal, etc.

**Control dominated systems:** transition systems, event-driven systems, Finite State Machine formalisms, signal emission and testing

→ StateFlow, StateCharts

→ SyncCharts, Argos, Esterel, etc.

**What about mixed systems?**

- most systems 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
Traditional Approaches: 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, SCADE + Esterel Studio SSM, etc.
An example: the Cruise Control (SCADE V4.2)
Observations

• automata can only appear at the leaves of the data-flow model

• 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

• what is the semantics of the whole?

• code certification (to meet avionic constraints)?

• 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, SLAP04]))

Proposal

- extend a basic clocked calculus (SCADE/Lustre) 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)
The Cruise Control with SCADE 6
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
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

\[
\text{Setup} \quad \text{Run}
\]

\[
\text{cond/x<\ldots}
\]

- 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

\[
\text{automaton}
\]

\[
\begin{align*}
&\text{...}
\end{align*}
\]

\[
| S(v) \rightarrow \text{do } o = v \text{ unless } c_1 \text{ then } T_1(o_1) \\
&\text{...}
\end{align*}
\]

\[
\begin{align*}
&\text{unless } c_n \text{ then } T_n(o_n) \\
&\text{...}
\end{align*}
\]

\[
\text{end}
\]
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 $\cdot$ is missing, it keeps implicitly its last value (i.e., $\cdot = \cdot$ is added)

• how to talk about absent value? If $\cdot$ 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

A part of the Milner coffee machine...

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 \((v \geq cost)\) and is supposed to be absent otherwise; we call it a signal
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 may be **unsafe** and raise **initialization problems**: what is the value if it has never been emitted?

• need extra methodology development rules (e.g., guarding every access by a test for presence)

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

Propose a programming construct reminiscent to pattern matching and 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

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
The Recursive Buffer

type 'a option = None | Some of 'a

let node count n = ok where
  rec o = 0 -> (pre o + 1) mod n
  and ok = false -> o = 0

(* the 1-buffer with bypass *)
let node buffer1 push pop = o where
  rec last memo = None
  and match last memo with
    None ->
      do present
        do present
          push(v) & pop() -> do emit o = v done
        | push(v) -> do memo = Some(v) done
      end done
    | Some(v) ->
      do present
        push(w) -> do emit o = v and memo = Some(w) done
        | pop() -> do emit o = v and memo = None done
      end done

end
A $n$-buffer can be build by putting $n$ buffers of size one in parallel

(* the recursive buffer *)

let rec node buffer n push pop = o where
  match n with
  | 0 ->
    do o = push done
  | n ->
    let pusho = buffer1 push pop in
    do
      o = buffer (n-1) pusho pop
    done
  end
Signals vs clocked streams

- in control structures, an absent definition for $x$ is implicitly completed with an equation $x = \text{last } x$

- this means that we need a memory to keep the value of $\text{last } x$

- signals are thus intrinsically more efficient: no memory is needed. $x$ is absent if nothing defines $x$

Is all that useful?

- signals already exist in synchronous data-flow: we have clocks!

- a signal is a flow which is present from time to time with a particular clock

- ask a lot for a compiler (and even the user).

- we need full dependent types here (the clock of $x$ must keep the control information defining the instant where $x$ is emitted)

- can we rely on more modest (but safe) mechanism while keeping the philosophy of the basic language?
Signals as existential 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 \((?x \wedge ?y) \vee ?x \vee ?y\) 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
Signals as Existential Types

Clock type of a signal: a pair \( ck \; \text{sig} = \Sigma(c : ck).ck \; \text{on} \; c \) made of:

- a (hidden) boolean sequence \( c \) which is itself on clock type \( ck \)
- a sequence sampled on \( c \), that is, with clock type \( ck \; \text{on} \; c \)

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
- mimics 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

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

\[
H \vdash \text{emit} \; x = e : [x : ck \; \text{sig}]
\]
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 $p\text{nextstate}$ with

| Await -> match $(a, b)$ with

  | $((\text{True}, x), (\text{True}, x))$ -> $\text{state} = \text{Emit}(x + y)$

  | _ -> $\text{state} = \text{Await}$

| $\text{Emit}(v)$ -> match $r$ with

  | true -> $\text{state} = \text{Await}$

  | false -> $\text{state} = \text{Emit}(v)$

and

match $\text{state}$ with

| Await -> $o = (\text{False}, \text{nil})$ and $\text{nextstate} = \text{Await}$

| $\text{Emit}(v)$ -> $o = (\text{True}, \text{nil})$ and $\text{nextstate} = \text{Emit}(v)$

and

$p\text{nextstate} = \text{Await} -> \text{pre} \text{nextstate}$
Conclusion

Automata and control structures

• an extension of a data-flow language with control structures

• various kinds of transitions, yet quite simple

• two semantics: a translation semantics and a logical semantics

Extensions: parameterised states and signals

• transmit local information between states

• signals as a light way to abstract the clock of a flow

• both features combine well

• light to implement in a translation-based compiler

• try it! (www.lri.fr/~pouzet/lucid-synchrone)