Hierarchical State Machines as Modular Horn Clauses

Pierre-Loïc Garoche[*], Temesghen Kahsai[♮], Xavier Thirioux[†]

[*]: DTIM – UFT – ONERA - The French aerospace Lab, Toulouse
[♮]: NASA Ames / CMU
[†]: IRIT – UFT – CNRS

Journées CAFEIN 2016 – Fev 2016

Motivation / Background

- Toolchain from Simulink to C via Lustre
  - CoCoSim from Simulink to Lustre, with traceability information
  - LustreC compiler performing modular compilation[1]
- Extended language to specify node contracts: assume-guarantee
  - Observer blocks in Simulink
  - Extended annotation language for Lustre
  - ACSL at C level
- Formal verification of contracts
  - SMT-based Model-checking of Lustre models, eg. Kind2, Tuff
  - Numerical invariants synthesis
  - Frama-C Weakest Precondition engine on C code
  - Proof preservation along compilation

[1] Biernacki et al., "Clock-directed modular code generation for synchronous data-flow languages".

This talk's contribution

- Extension of LustreC to handle hierarchical states automaton[2]
  - Compile automaton into clocked Lustre expressions and node reset
  - Handle until/unless transitions
  - Provide a node definition of each state: support computation of state local invariants
- Compile Lustre into modular Horn Clauses
  - first SMT encoding that preserves the modular structure of the Lustre model[3]
  - support advances constructs: enumerated clocks, when, merge, conditional reset of node (every)
  - enable the use of Z3-PDR or Spacer algorithms
  - enable the computation of node local / state local invariants using a safety property driven static analysis.

---

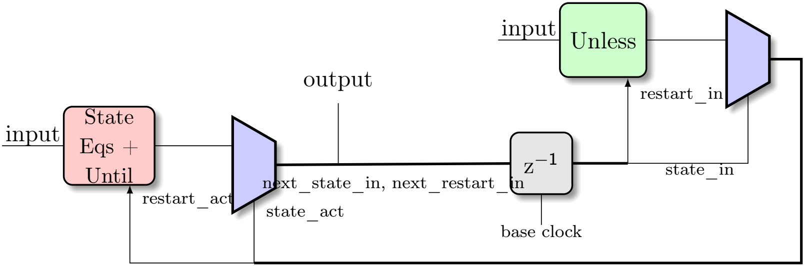
[2] Colaço, Pagano, and Pouzet, "A conservative extension of synchronous data-flow with state machines".

[3] Garoche, Gurfinkel, and Kahsai, "Synthesizing Modular Invariants for Synchronous Code".

Contents

Automaton transitions and semantics

- Strong/Weak transitions: Unless/Until
- at each step: putative state_in and an actual state state_act
- actual state: at most one unless transition from state_in
- next putative state: at most one until trans. from state_act
- actual state equations executed
- Resume/Restart flags to reset of state in its initial setting



Automaton semantics: design choices

- Enforce structural modularity:
  - Computation of weak/strong transition, and state equations in independent nodes (one for each state)
  - Enable local scheduling and optimizing of state equations
  - State local variables
  - State invariants can be computed without complex data analysis
- Loss in expressivity,because of possible causality issues.
- Could be recover by inlining until/unless nodes.

Causality issues

The non valid lustre node

```
node failure (i:int) returns (o1, o2:int);
let
  (o1, o2) = if i = 0
             then (o2, i)
             else (i, o1);
tel
```

can be reformulated as a valid automaton

```
node solution (i:int) returns (o1, o2:int);
let
  automaton condition
  unless i <> 0 resume KO
  state OK:
  let
    (o1, o2) = (o2, i);
  tel
  state KO:
  unless i = 0 resume OK
  let
    (o1, o2) = (i, o1);
  tel
tel
```

Causality issues

This node non causal: o is not defined when evaluating the unless condition

```
node triangle_invalid (r:bool) returns (o:int);
let
  automaton trivial
  state One:
  unless r || o = 100
  let
    o = 0 -> 1 + pre o;
  tel
tel
```

But this one is accepted by KCG while our compiler rejects it

```
node triangle (r:bool) returns (o:int);
let
  automaton trivial
  state One:
  unless r || pre o = 100
  let
    o = 0 -> 1 + pre o;
  tel
tel
```

We do not authorize unless condition on putative state memories

Contents

Automaton skeleton

```
node nd (inputs) returns (outputs);
var locals
let
  other_equations
  automaton aut
  ...
  state S_i :
  ...
  unless (sc_j, sr_j, SS_j)
  ...
  var locals_i
  let
    equations_i
  tel
  ...
  until (wc_j, wr_j, WS_j)
  ...
tel
```

Expression as pure dataflow equations

- States are encoded as fresh enumerated datatype

```
type aut_type = enum { S_1 , ... , S_n };
```

- Unless node

```
node S_i_unless (ReadUnless_i) returns (restart_act : bool,
                                        state_act : aut_type clock)
let
  (restart_act, state_act) =
    if sc_1 then (sr_1, SS_1) else
    if sc_2 then (sr_2, SS_2) else ...
    (false, S_i);
tel
```

- Handler (output computation) and until nodes

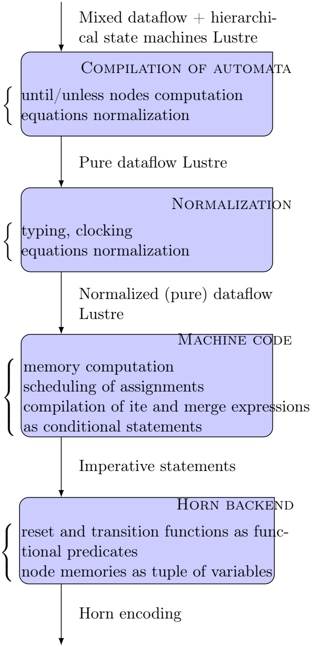
```
node S_i_handler_until (ReadEqs_i ∪ ReadUntil_i)
returns (restart_in : bool, state_in : aut_type clock,
         WriteEqs);
var locals_i -- used for output equations
let
  (restart_in, state_in) =
    if wc_1 then (wr_1, WS_1) else
    if wc_2 then (wr_2, WS_2) else ...
    (false, S_i);
  equations_i                          tel
```

Main node

```
node nd (inputs) returns (outputs);
var locals;
    aut_restart_in, aut_next_restart_in, aut_restart_act : bool;
    aut_state_in, aut_next_state_in, aut_state_act : aut_type clock;
let
  other_equations
  (aut_restart_in, aut_state_in) =
     (false, S1) -> pre (aut_next_restart_in, aut_next_state_in);
  (aut_restart_act, aut_state_act) =
     merge aut_state_in
     ...
     (Si -> Si_unless((ReadUnless_i) when Si(aut_state_in))
                                          every aut_restart_in)
     ...
  (aut_next_restart_in, aut_state_next_in, WriteEqs) =
     merge aut_state_act
     ...
     (Si -> Si_handler_until((ReadEqs_i ∪ ReadUntil_i)
                                    when Si(aut_state_act))
                                         every aut_restart_act)
     ...
tel
```

Contents

LustreC Compiler stages



Compiler stages

- Normalization: introduce fresh variables for each
  - function call with a fresh uid
  - pre construct
  - follow-by (arrow) $->$
- Machine code
  - computes memories
  - schedule node equations
  - substitute functional ite into imperative ones

Definition (Node memories and instances)

Let f be a Luste node with normalized equations eqs. Then we define its set of memories and node callee instances as:

$$\begin{aligned} Mems(f) = & \ \{x \mid x = pre\ \_ \in eqs\} \\ Insts(f) = & \ \{(foo, uid) \mid \_ = foo^{uid}(\_) \in eqs\} \end{aligned}$$

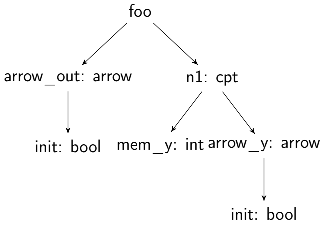
Modular Node Memories

The *follow by* operator is interpreted as a node instance of a generic polymorphic node `arrow`:

```
node arrow (e1, e2: 'a) returns (out: 'a)
var init: bool;
let
  init = true -> false;
  out = if init then e1 else e2;
tel
```

Example of node and associated hierarchical memory

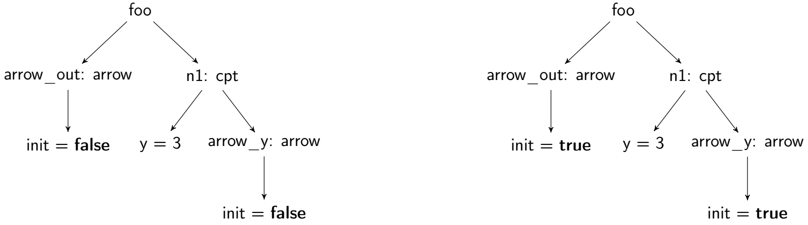
```
node cpt (z: bool) returns (y: int);
let
  y = 0 -> if z then 0 else pre y + 1;
tel

node foo (z: bool) returns (out: int)
let
  out = 1 -> cpt(z);
tel
```



Node reset

A node reset can be defined conditionally with the every feature
e = foo (...) every ClockValue(clock_var};

- preserves the memories value
- resets arrows init to true



Horn encoding

- State tree as l-labeled tuple $\mathtt{state}^{l}(\mathtt{f,uid})$
- Node modular reset predicates

```
rule (=> (= initⁿ true)
         (arrow_reset (initᶜ, initⁿ)))

rule (=>
  (    ⋀          (= memⁿ memᶜ) -- local memories are preserved
   mem ∈ Mems(f)
  -- child nodes (callee) are recursively reset,
  -- eventually arrow nodes
        ⋀            g_reset (stateᶜ (g,guid), stateⁿ (g,guid))
  (g, uid) ∈ Inst(f)
  )
     (f_reset (stateᶜ(f,uid), stateⁿ(f,uid))))
```

- Node modular step predicates

```
rule (=>
       (⋀ node equations) -- derived from machine code
     (f_step (inputs, outputs, stateᶜ(f,uid), stateⁿ(f,uid))))
```

Collecting semantics and property verification in Horn

- Collecting semantics inductively defined
  - initial states are reachable

```
rule (=> (f_reset (state^c(f,uid), state^n(f,uid)))
         (Reach (state^n(f,uid))))
```

  - and so do states reachable in one transition

```
rule (=>
  (and
    (f_step (inputs, outputs, state^c(f,uid), state^n(f,uid)
    (Reach (state^c(f,uid))))
  (Reach (state^n(f,uid))))
```

- Safety property

```
(declare-rel ERR ())
(rule (=>
  (and (not (property over state values)
       (Reach (state^n(f,uid))))
  ERR))
(query ERR)
```

If ERR is sat(isfiable) we have a trace leading to the violation of the property

Clocks in Horn clauses

- Clock values defined as regular enumerated type

```
type clock_type = enum {Value1, Value2 };
```

becomes

```
(declare-datatypes () ((clock_type Value1 Value2)));
```

- Merge are compiled, in Machine code, as ite statements

```
e = merge ck (Value1 -> x when Value1(ck))
             (Value2 -> y when Value2(ck))
```

become the imperative switch-case expression.

```
switch (ck) {
  case Value1 : e = x; break;
  case Value2 : e = y; break;
}
```

and the Horn predicate

```
(and (=> (= ck Value1) (= e x))
     (=> (= ck Value2) (= e y)))
```

- **when** expressions are used to type the program but do not impact code generation

Node reset in Horn clauses: e = foo (x) every condition ;

In machine code, it becomes a conditional side effect instruction:

```
if (condition) { Reset(foo, uid) };
```

In Horn, all equations have to be functional. We have to relate new states labeled *n* to current (old) states labeled *c*. We introduce an intermediate label *i*:

- A regular call is defined as

```
(and (= (state^c(f,uid)) (state^i(f,uid)))
     (f_step (inputs, outputs, state^i(f,uid), state^n(f,uid))))
```

- A reset call becomes

```
(and (f_reset (state^c(f,uid), state^i(f,uid)))
     (f_step (inputs, outputs, state^i(f,uid), state^n(f,uid))))
```

To ease the generation of these instructions, we compile
e = foo(x) every cond; as

```
if (cond) Reset(foo, uid) else NoReset(foo, uid); Step(foo, uid);
```

or, for regular (un-restarted) calls e = foo(x);

```
NoReset(foo, uid); Step(foo, uid);
```

NoReset instruction denotes the equality between *i*- and *c*- labeled

Contents

The two counters example

Classical example in our Kind benchmarks: prove that both nodes compute the same output.

```
node greycounter (x:bool) returns (out:bool);
var a,b:bool;
let
  a = false -> not pre(b);
  b = false -> pre(a);
  out = a and b;
tel

node intloopcounter (x:bool) returns (out:bool);
var time: int;
let
  time = 0 -> if pre(time) = 3 then 0
              else pre time + 1;
  out = (time = 2);
tel
```

An automaton based version

```
node auto (x:bool) returns (out:bool);
let
  automaton four_states
  state One :
  let
   out = false;
  tel until true restart Two
  state Two :
  let
   out = false;
  tel until true restart Three
  state Three :
  let
   out = true;
  tel until true restart Four
  state Four :
  let
   out = false;
  tel until true restart One
tel
```

Compilation of automaton as clocked expressions

- Enumerated clock type to represent automaton states

```
-- Datatype encoding states as enumerated clocks
type auto_ck = enum {One, Two, Three, Four };
```

- Nodes for each state behavior: handler, until transitions and unless transitions

```
-- unless transitions (none in this example)
function Four_unless (restart_in: bool; state_in: auto_ck)
returns (restart_act: bool; state_act: auto_ck)
let
    restart_act, state_act = (restart_in,state_in);
tel

-- state handler and until transitions
function Four_handler_until (restart_act: bool;
                             state_act: auto_ck)
returns (restart_in: bool; state_in: auto_ck; out: bool)
let -- encodes the next state, here One
    restart_in, state_in = (true,One);
    out = false; -- returns true in the handler
                 -- for state Three
tel
```

Main node

```
node auto (x: bool) returns (out: bool)
-- Local variables
var -- variables capturing states and restart status
    mem_restart: bool; mem_state: auto_ck;
    next_restart_in: bool; restart_in: bool;
    restart_act: bool; next_state_in: auto_ck;
    state_in: auto_ck clock; state_act: auto_ck clock;

    -- variables for each state
    ---- output of the node (handler)
    four_out: bool ;
    ---- until behavior
    four_restart_in: bool; four_state_in: auto_ck;
    ---- unless behavior
    four_restart_act: bool; four_restart_in: auto_ck;

let
-- restart status is false initial, initial state is One
restart_in, state_in = ((false,One) -> (mem_restart,mem_state));
-- next values are determined by next_restart_in, next_state_in
mem_restart, mem_state = pre (next_restart_in,next_state_in);
```

Main node (continued)

```
restart_act, state_act =
 merge state_in
    (One -> (one_restart_act, one_state_act))
    (Two -> (two_restart_act, two_state_act))
    (Three -> (three_restart_act, three_state_act))
    (Four -> (four_restart_act, four_state_act));

next_restart_in, next_state_in, out =
 merge state_act -- merging on
          (One -> (one_restart_in, one_state_in, one_out))
          (Two -> (two_restart_in, two_state_in, two_out))
          (Three -> (three_restart_in, three_state_in, three_out))
          (Four -> (four_restart_in, four_state_in, four_out));
four_restart_in, four_state_in, four_out = Four_handler_until
  (restart_act when Four(state_act),
   state_act   when Four(state_act)) every (restart_act);
... -- similar definitions for other states
four_restart_act, four_state_act = Four_unless
   (restart_in when Four(state_in),
    state_in when Four(state_in)) every (restart_in);
... -- similar definitions for other states
tel
```

Automaton clock, and transition nodes

The Horn encoding can now be produced. Enumerated type enable the declaration of clock's values:

```
(declare-data auto_ck () ((auto_ck One Two Three Four)));
```

Until and unless functions are defined as Horn predicates.

```
; Four_handler_until
(declare-rel Four_handler_until (Bool auto_ck Bool auto_ck Bool))
(rule (=> (and (= out false)
               (= state_in One)
               (= restart_in true))
      (Four_handler_until restart_act state_act
                          restart_in  state_in out)))

; Four_unless
(declare-rel Four_unless (Bool auto_ck Bool auto_ck))
(rule (=> (and (= state_act state_in)
               (= restart_act restart_in))
      (Four_unless restart_in state_in restart_act state_act)))
```

Reset and step predicates

Finally the reset and step predictates are defined:

```
(rule (=> (and (= mem_restart_m mem_restart_c)
               (= mem_state_m mem_state_c)
               (= arrow.init_m true))
     (auto_reset mem_restart_c mem_state_c arrow.init_c
                 mem_restart_m mem_state_m arrow.init_m)))
```

```
(rule (=>
 (and (= arrow.init_m arrow.init_c)
      (= arrow.init_x false) ; update of arrow state
 (and (=> (= arrow.init_m true) ; current arrow is first
          (and (= state_in One)
               (= restart_in false)))
      (=> (= arrow.init_m false) ; current arrow is not first
          (and (= state_in mem_state_c)
               (= restart_in mem_restart_c))))
 (and (=> (= state_in Four) ; unless block for aut. state Four
    (and (Four_unless restart_in state_in four_restart_act four_st
         (= state_act four_state_act)
         (= restart_act four_restart_act)))
    ...) ; similar definition for other states
 (and (=> (= state_act Four) ; handler + until for state Four
          (and (Four_handler_until restart_act state_act
                    four_restart_in four_state_in four_out)
               (= out four_out)
               (= next_state_in four_state_in)
               (= next_restart_in four_restart_in)))
      ...) ; similar definition for other states
 (= mem_state_x next_state_in) ;next val. for mem_state
 (= mem_restart_x next_restart_in)) ; next val. for mem_restart
(auto_step x     ; inputs
          out   ; outputs
          mem_restart_c mem_state_c arrow.init_c     ; old state
          mem_restart_x mem_state_x arrow.init_x))) ; new state
```

Results

Generated predicates, collecting semantics and safety properties are analyzed Z3/PDR or Spacer.

Our tool Zustre synthesize the following node local invariants:

```
contract intloopcounter (x:bool) returns (out:bool);
var time:int;
let
  guarantee (
     true ->
     (pre (top.ni_2.intloopcounter.__intloopcounter_2) >= 3 =>
        not (top.ni_2.intloopcounter.__intloopcounter_2 >= 2)
       or
        pre top.ni_2.intloopcounter.__intloopcounter_2 >= 4
     ));
tel

contract auto (x:bool) returns (out:bool);
var (four_states__next_restart_in,four_states__next_state_in:bool;
    (four_states__next_restart_in,four_states__next_state_in:four_
let
   guarantee (
      true -> (One = pre auto.automato_state)
                => (Two = auto.automato_state)
   );
tel
```

Invariants for the automaton

```
automaton_contract four_states__Three_handler_until;
let
  four_states__Three_handler_until.out_out and
  four_states__Three_handler_until.four_states__state_in =Four
tel

automaton_contract four_states__Four_handler_until;
let
  not four_states__Four_handler_until.out_out and
  four_states__Four_handler_until.four_states__state_in = One
tel

automaton_contract four_states__Four_unless;
let
  four_states__Four_unless.four_states__state_in = One or
  four_states__Four_unless.four_states__state_act = Four or
  four_states__Four_unless.four_states__state_in = Three or
  four_states__Four_unless.four_states__state_in = Two
tel
```

Conclusion

Results:

- Modular compilation of Simulink/Lustre into Horn clauses
- Handle clocks, enumerated types, node reset and ... automaton
- Enable SMT-based model-checking and node/state local invariant compuation

Perspectives:

- Formalize the expression (a subset of) Stateflow in our automaton semantics
- Apply it to more serious examples
  - Microwave
  - NASA Docking station example
  - NASA next Lunar Rover (under dev)