Hybrid Systems Simulation with Transparent Observers

Francois Bidet and Marc Pouzet

DIENS
PSL/ENS/INRIA
Marc.Pouzet@ens.fr

Workshop SYNCHRON
Fréjus
December 1, 2022

Context

- A language to express executable hybrid systems models.
- E.g., a discrete-time model of a controller or a plant;
- e.g., a continuous-time model of a controller or a plant with possible discontinuities.
- + their parallel composition.

Take your favorite one, e.g.: Simulink, Ptolemy or Zelus [Bourke and Pouzet, 2013] [1].

Here, Zelus:

- A synchronous language; parallel composition is ideal (no concurrency).
- discrete-time signals and systems: streams and functions;
- continuous-time signals and systems: ODEs/zero-crossings, functions;
- and a type discipline to ensure the composition is mathematically sound.

[1] `http://zelus.di.ens.fr`

Programming a Hybrid System Model

- A Hybrid system model is simulated with an ODE/zero-crossing solver;
- The solver is global (a single one). Hence:
  - adding/removing an ODE may change what is observed;
  - adding/removing an event may change what is observed.
- this make hybrid systems models extremely fragile and hardly portable.

```
let hybrid sin_cos(freq)() = (sin, cos) where
  rec der sin = freq *. cos init 0.0
  and der cos = -. freq *. sin init 1.0

let hybrid main1 () = sin_cos(1.0)

(* below, the computation of [o2] changes that of [o1] *)
let hybrid main2 () =
  let o1 = sin_cos(1.0) in
  let o2 = sin_cos(100.0) in
  o1, o2
```

What to do?

- A model is an approximation or simplification of the reality;
- Simulation is essential as well as compile-time checks: is the programmed model correctly written?
- It can be buggy, contain stupid errors, etc. exactly like programs.
- How to be convinced that a model is correct?
- Can we provide some of the very basic tools a programmer is using to test/debug hybrid system models as if they were regular programs?

Dynamic assertions in programming languages is one such tool.

Why Assertions are great

- Introduced first by Turing, rediscovered by Naur and Floyd, studied by Hoare as a logic on program variables [Hoare, 2000];
- provided by many general purpose languages

```
...
assert (x != 0)
z = y / x;
...
```

- when assertion arguments are language expressions, they can be used defensively (at run-time);
- or to specify logical (not necessarilly executable) properties (e.g., Ada);
- possibly split into an assume/guaranty contract.

In a reactive language, e.g., *synchronous observers* [Halbwachs et al., 1993].

To retain: removing/adding assertions does not change outputs!

Solvers find a compromise between precision and speed

- slow when the dynamics is stiff, many intermediate points are computed;
- fast when the dynamics is smooth; smaller intermediate points.
- Adding an event (time or state) stops the simulation.

Hence:

- adding/removing an ODE changes the computed approximation.
- adding/removing an event changes the computed approximation.

If I add/remove `assert(e)`, is what is observed changed?

Can I look without touching? [2]

[2] Timothy Bourke

Examples

Two independent oscillators: one fast; one slow.

The Water tank.

The heater model [3]

Examples in Ocaml, using the SundialsML binding [Bourke et al., 2018] [4]

[3] https://fr.mathworks.com/help/simulink/slref/thermal-model-of-a-house.html

[4] https://github.com/inria-parkas/sundialsml

The example in Zelus

```
(* The Brusselator *)
let hybrid brusselator(a,b) = (x,y) where
    rec der x = a +. x *. x *. y -. b *. x -. x init 1.0
    and der y = b *. x -. x *. x *. y init 1.0

let pi = 3.141592653589793
(* add an other oscillator *)
let hybrid harmonic(p) = x where
    rec der x = v init 1.0
    and der v = -2.0 *. pi *. x /. p init 0.0

(* Putting the harmonic besides the brusselator *)
(* changes the output of the first *)
let hybrid simu() =
    let der t = 1.0 init 0.0 in
    let x,y = brusselator(1.0,2.001) in
    let z = 0.0 (* harmonic(1e-5) *) in
    print(t, x)
```

Examples of uses of an assertion

The water tank. Checks that none of the water tank is empty.

```
let hybrid tank(h0,vo)(vi) = h where
    rec der h = vi -. vo init h0
    (* and assert (h >= 0.0) *)

(* fill a tank as soon as its level is below 0.5 *)
(* if input vi < 2.5, then cumulative level decreases and a *)
(* tank will eventually have a negative level *)
let hybrid tanks(vi) = (h1,h2) where
    rec h1 = tank(2.0, 1.0)(vi1)
    and h2 = tank(1.0, 1.5)(vi2)
    and automaton
    | First ->
        do  vi1 = vi and vi2 = 0.0
        until up(0.5 -. h2) then Second
    | Second ->
        do  vi1 = 0.0 and vi2 = vi
        until up(0.5 -. h1) then First
```

Proposal

We consider a functional interface of a hybrid system model. E.g., generated by the compiler of a hybrid system language.

Purpose

- We would like to be able to write assertions in a model;
- anywhere, e.g., in a mode of an automaton like an invariant to be checked during simulation.
- that does not influence numerically what is observed.

Proposal

- Formalize what is the concrete semantics of a hybrid system model.
- It is abstract, the solvers are parameters of the semantics.
- Extend the functional interface to incorporate assertions.
- Each assertions will use its own solver.

Synchronous Systems in OCaml

```
(* A synchronous model. *)
(*- node model(u:'a) returns (o:'b)
     rec  o, s = step (last s) u
     and init s = state *)

type ('p, 'a, 'b) node =
  Node : { s : 's; (* the internal state *)
           step : 's -> 'a -> 'b * 's; (* the transition function *)
           reset : 'p -> 's -> 's; (* reset *)
         } -> ('p, 'a, 'b) node

(* auxiliary function *)
let mapfold f acc l =
  let rec maprec acc = function
    | [] -> [], acc
    | x :: l ->
       let y, acc = f acc x in
       let l, acc = maprec acc l in
       y :: l, acc in
  maprec acc l

(* Simulation of a synchronous stream function - a node *)
(* The run function returns a function from lists to lists *)
let run: ('p, 'a, 'b) node -> 'a list -> 'd list =
  fun (Node { s; step }) u_list ->
  let o_list, s = Aux.mapfold step s u_list in o_list
```

A Functional Interface in OCaml

```
type ('x, 'xder, 'zin, 'zout) solver =
  { csolve : ((time -> 'x -> 'xder) -> 'x -> time
             -> time * (time -> 'x));
    zsolve : ((float -> 'x -> 'zout) -> (time -> 'x)
             -> time -> time * 'zin option) }
```

- Given `f` and horizon to reach `h`, `csolve f h = t, dky` returns the actual horizon `h` and `dky: [0,h] -> 'x` a dense solution.
- Given `g`, dense solution `dky` and horizon to reach `h`, `zsolve g dky h = h', zin_opt` returns the actual horizon `h'` and optional zero-crossing `zin_opt`.

Hybrid Systems in OCaml

```
(* A hybrid functional system model - *)
(* ODEs + zero-crossing events + states + sequential function *)
type ('a, 'b, 's, 'x, 'xder, 'zin, 'zout) hnode =
  Hnode :
    { s : 's; (* state *)
      fder : 's -> 'a -> 'x -> 'xder; (* derivative *)
      fzero : 's -> 'a -> 'x -> 'zout; (* zero-crossing *)
      fstep : 's -> 'a -> 'b * 's; (* step function *)
      fout : 's -> 'x -> 'a -> 'b; (* output function *)
      horizon : 's -> float; (* the new horizon *)
      cset : 's -> 'x -> 's; (* sets the continuous state into [s] *)
      cget : 's -> 'x; (* gets the continuous state from [s] *)
      zset : 's -> 'zin -> 's; (* sets the zero-crossing into [s] *)
    } -> ('a, 'b, 's, 'x, 'xder, 'zin, 'zout) hnode

(* The same except that it can contains 0 or more hybrid assertions *)
type ('a, 'b, 'x, 'xder, 'zin, 'zout) hybrid =
  Hybrid : { body : ('a, 'b, 's, 'x, 'xder, 'zin, 'zout) hnode;
             assertions : ('s, bool, 'x, 'xder, 'zin, 'zout) hybrid list } ->
           ('a, 'b, 'x, 'xder, 'zin, 'zout) hybrid
```

Super dense time

```
(* a super dense time signal is a list of pairs { length; u } where
 *- [length] is a positive (possibly null) floatting point number and
 *- [u : [0,length] -> t] *)
type time = float

type 'a value = { length : time; u : time -> 'a }

type parameters = { period : float; }


let dot x = { length = 0.0; u = fun _ -> x }
let dense u h = { length = h; u }

(* Check that a property is true at sampled instants *)
(* given a value { u; horizon } and a period p, check that *)
(* [u(k.p)] for all k in Nat such that 0 <= k.p <= horizon is true *)
let assert_cont p { u; horizon } =
  let rec sample t =
    if t <= horizon then
      begin
        assert (u t);
        sample (t +. p)
      end in
  sample 0.0
```

The Simulation Loop

```
let run { csolve; zsolve }
      (Hnode({ s; fder; fzero; fout; fstep; cset; cget; zset; encore } as hm))
      { u; length } =
  let fout s dky u t0 t = fout s (dky t) (u (t +. t0)) in
  (* discrete mode *)
  let rec discrete s t0 k_list =
    let o, s = fstep s (u t0) in
    let k_list = (dot o) :: k_list in
    let tmax = min (horizon s) length in
    if t0 >= length then k_list, s
    else if tmax <= 0.0 then discrete s t0 k_list
         else continuous s t0 tmax k_list
  (* continuous mode *)
  and continuous s t0 k_list =
    let f t x = fder s (u (t +. t0)) x in
    let g t x = fzero s (u (t +. t0)) x in
    let h, dky = csolve f (cget s) (horizon -. t0) in
    let h, zin_opt = zsolve g dky h in
    let s = cset s (dky h) in
    let s = match zin_opt with
      (* when no zero-crossing was detected -
         we do a blank discrete step *)
      | None -> s | Some(zin) -> zset s zin in
    discrete s (t0 +. h) ((dense (fout s dky u t0) h) :: k_list) in
  let k_list, s = discrete s 0.0 [] in
  List.rev k_list, Hnode { hm with s }
```

The Simulation Loop (check)

```
let scheck { csolve; zsolve } { period }
      (Hnode({ s; fder; fzero; fout; fstep; cset; cget; zset; encore } as hm))
      { u; horizon } =
  let sout s dky u t0 t = cset s (dky (t +. t0)) in
  (* discrete mode *)
  let rec discrete s t0 k_list =
    let o, s = fstep s (u t0) in
    assert o;
    let k_list = (dot s) :: k_list in
    let tmax = min (horizon s) length in
    if t0 >= length then k_list, s
    else if tmax <= 0.0 then discrete s t0 k_list
         else continuous s t0 tmax k_list
  (* continuous mode *)
  and continuous s t0 k_list =
    let f t x = fder s (u (t +. t0)) x in
    let g t x = fzero s (u (t +. t0)) x in
    let h, dky = csolve f (cget s) (horizon -. t0) in
    let h, zin_opt = zsolve g dky h in
    assert_cont period
      { horizon; u = fun t -> fout s (dky t) (u (t +. t0)) };
    let s = cset s (dky h) in
    let s = match zin_opt with
      (* when no zero-crossing was detected - we do a blank discrete step *)
      | None -> s | Some(zin) -> zset s zin in
    discrete s (t0 +. h)
      ((dense (sout s dky u t0) h) :: k_list) in
```

The Main Loop - Hybrid Systems with Nested Assertions

```
let check solver period hm u_list =
  let rec check : 'c. ('c, bool, 'x, 'xder, 'zin, 'zout) hybrid ->
                'c value -> ('c, bool, 'x, 'xder, 'zin, 'zout) hybrid =
    fun (Hybrid { body; assertions }) { horizon; u } ->
    let s_list, body =
      Sim.scheck solver period body { horizon; u } in
    let assertions =
      List.map (fun hm -> List.fold_left check hm s_list) assertions in
    Hybrid { body; assertions } in
  List.fold_left check hm u_list

let run solver period hm u_list =
  let run (Hybrid { body; assertions }) { horizon; u } =
    let o_s_list, body =
      Sim.srun solver period body { horizon; u } in
    let o_list, s_list = List.split o_s_list in
    let assertions =
      List.map (fun hm -> check solver period hm s_list) assertions in
    o_list, Hybrid { body; assertions } in
  let o_list, hm = Aux.mapfold run hm u_list in
  o_list
```

Why it does not work.

An other idea: define the solver, the zero-crossing solver and the simulation itself as a Mealy-machine.

It works!

Conclusion

- This is on going work.
- For the moment, the definition in OCaml of the different elements — interface of a hybrid system, simulation functions.
- I have shown the simulation for memoryless solvers (e.g., Runge Kutta).
- The definition for statefull solvers is done.
- A new version of Zelus based on a reference executable semantics on development (branch `work` in the GitHub repo).

References I

Bourke, T., Inoue, J., and Pouzet, M. (2018).
Sundials/ML: connecting OCaml to the Sundials numeric solvers.
Electronic Proceedings in Theoretical Computer Science, 285:101–130.

Bourke, T. and Pouzet, M. (2013).
Zélus, a Synchronous Language with ODEs.
In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, Philadelphia, USA. ACM.

Halbwachs, N., Lagnier, F., and Raymond, P. (1993).
Synchronous observers and the verification of reactive systems.
In Nivat, M., Rattray, C., Rus, T., and Scollo, G., editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente. Workshops in Computing, Springer Verlag.

Hoare, T. (2000).
Assertions.
In Grieskamp, W., Santen, T., and Stoddart, B., editors, *Integrated Formal Methods*, pages 1–2, Berlin, Heidelberg. Springer Berlin Heidelberg.