Zélus to DynIbex: compilation toward an interval CSP framework for contracts verification

François Pessaux

U2IS, ENSTA Paris, Institut Polytechnique de Paris

Synchron, 22 Nov. 2021

`firstname.lastname@ensta-paris.fr`

What – Why

What

- Write hybrid systems in a high-level programming language.
- Compile them to a simulation executable.
- Check contracts compliance during the simulation.
- Use a guaranteed integration framework.

Why

- Rely on high-level constructs.
- Rely on analyses performed by the language (causality, initialization).
- Avoid manual encoding in C++.

The Programming Language : Zélus[1]

Paradigm

- Synchronous language :
  - dataflow equation, hierarchical automata, signals. . .
- With Ordinary Differential Equations.
- Allows modeling hybrid systems.
- Generates OCaml simulation code.

Structure of Programs

- Hierarchy of (parameterized) nodes returning value(s).
- Nodes contain dataflow and differential equations.
- Operations lifted on stream of data.
- Nodes can be (non-recursively) instantiated.

---

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

Shape of Addressed Programs

Restrictions on Zélus programs

- Hierarchy of `hybrid` nodes.
- One unique explicit return value per node.
- No discrete computation.
- Contain only ODEs, dataflow equations (and opt. 1 automaton).
- No nested automata (along the hierarchy or in a same node).

Several Dynamics

- Use automaton, transitions between automaton states on conditions.
- Some dynamics common to all the automaton states (outside the automaton).
- Some dynamics particular to some automaton states.

⇒ Automaton state change may imply dynamics change.

- New dynamics may trigger a continuous state "jump" (reset).

Example : Rocket

```
let hybrid main () = zpos where
  rec init zpos = 0.0
  and init speed = 0.0
  and der power = -. 2.0 *. power init 100.0
  and automaton
    | EngineOn ->
        do
        der speed = -9.81 +. power
        and der zpos = speed
        until up (-. (power -. 0.001)) then EngineOff
    | EngineOff ->
        do
        der speed = -. 9.81
        and der zpos = speed
        until up (-. zpos) then Crashed
    | Crashed ->
        do
        der speed = 0.0
        and der zpos = 0.0
        done
   end
```

The Guaranteed Integration Framework : DynIbex[2]

Features

- Plug-in of the Ibex library written in C++.
- Provides validated numerical integration methods, using intervals.
- Can be used to simulate differential equations.

Example of Initial Value Problem (decreasing exponential)

```
int main () {
  const int n = 1 ;
  Variable y (n) ;
  IntervalVector yinit (n) ;
  yinit[0] = Interval (1.0, 1.0) ;                          /* y_0 = [1, 1] */
  Array<const ExprNode> eq_body (n) ;
  eq_body.set_ref (0, -y[0]) ;                              /* der (y) = -y */
  const ExprVector& eq_return = ExprVector::new_ (eq_body, true) ;
  Function ydot = Function (y, eq_return) ;
  ivp_ode problem = ivp_ode (ydot, 0.0, yinit) ;
  simulation simu = simulation (&problem, __DURATION__, __METH__, __PREC__) ;
  simu.run_simulation () ;
  return 0 ;
}
```

---

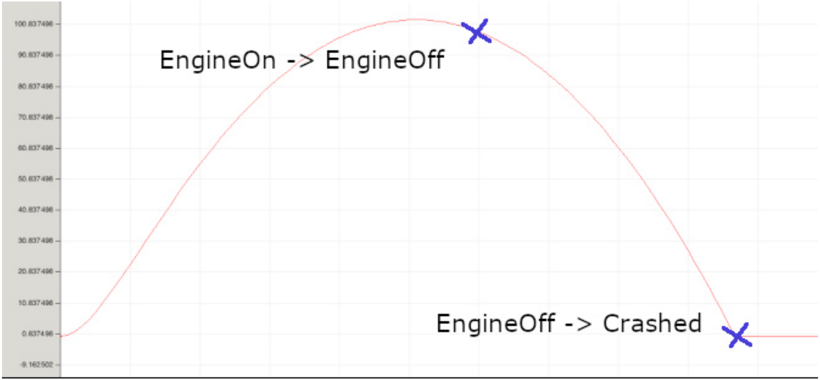
[2] http://perso.ensta-paristech.fr/~chapoutot/dynibex/

Need for a Dedicated Compilation

Why not a simple C++ translation of Zélus output?

- Generated code tightly dependent on the ODE solver.
- Zélus' solving runtime very different from DynIbex's one.
- Intervals strongly incompatible with point-wise simulation.
- Runtime simulation code deeply mixed with the physics code.
- Automaton mode switches no more deterministic.

Point-wise Automaton Simulation

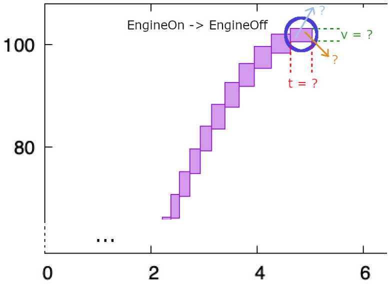


Point-wise simulation

- New dynamics at a precise time.
- New dynamics with precise initial conditions.

⇒ New evolution at precise time and from precise state.

Interval-Based Automaton Simulation



Interval-based simulation

- Old dynamics possibly still active in a part of the box.
- New dynamics possibly starts at all the instants in the box.
- New dynamics possibly starts with all the initial values in the box.

⇒ New evolution at imprecise time from imprecise state.

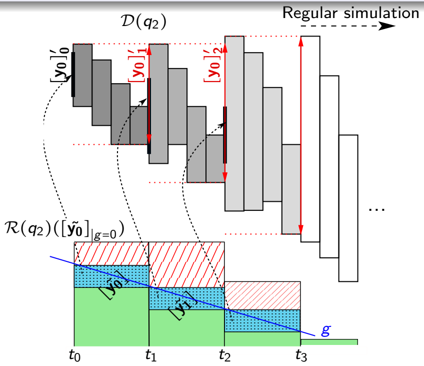
- Several automaton states possibly reachable.

⇒ Tree of simulations.

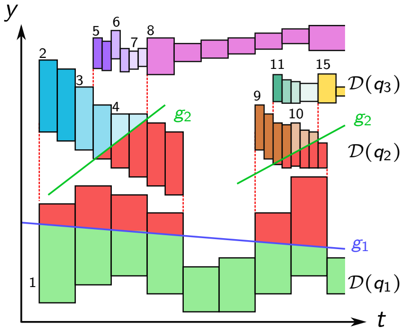
Time Uncertainty to Space Uncertainty

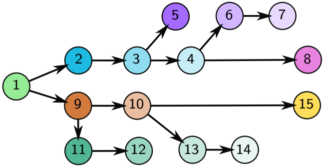
Principle

- Iteratively perform a “sub-simulation” of the new dynamics,
- …on each box where the guard crosses 0,
- …applying the “jump”,
- …flattening the obtained intervals.



Tree of Simulations





Chaining Simulations

- "Sub-simulations" chained together.
- First one is child of the simulation of the previous dynamics.
- Last one is the parent of the standard one with new dynamics.
- Recursive process (during "sub-simulations").

Internal Representation

C++ Implementation

- Runtime written once for all.
- For each system, automaton data generated from the Zélus program:
  - DynIbex `Function` and `ExprNode` constructs.
  - Static values generated at compile-time (arrays of structs).
  - Final call the runtime main function.

Data Interpreted by the Simulation Runtime

Automaton = (state → dynamics) × (state → transition) × (state → jump)
Transition = (state × guard × jump)
Transition condition, jump = arithmetic expression

Code Generation Principle: from Zélus to Pre-Automaton

First step

- Nodes without Zélus automaton transformed to a trivial automata.
- Toplevel ODEs injected in all the states, init removed.
- Toplevel dataflow equations injected in all the states.
- For each state
  - Eliminate syntactic constructs not handled.
  - Compute state's jump as union of inits of ODEs having some (identity otherwise).
- Compute inits of the pre-automaton as union of toplevel ODEs inits.

. . . and some omitted other gory details.

From Zélus to Pre-Automaton (Example)

```
let hybrid time () = t where
    der t = 1.0 init 0.0

let hybrid main () = zpos where
  rec init zpos = 0.0
  and g = -9.81
  and init speed = 0.0
  and der power =
    -. 2.0 *. power init 100.0
  and t = time ()
  and automaton
    | EngineOn ->
        do
        der speed = g +. power
        and der zpos = speed
        until up (-. (power -. 0.001))
        then EngineOff
    | EngineOff ->
        do
        der speed = g
        and der zpos = speed
        done
   end
```

```
Node: time
  Toplevel inits:
    init t = 0.0
  State: _St0
    der t = 1.0
    Jumps:
      t <- t
Node: rocket
  Toplevel inits:
    init power = 100.0
    init speed = 0.0
    init zpos = 0.0
  State: EngineOn
    der zpos = speed
    der speed = g + power
    t = time ()
    der power = -2.0 * power
    g = -9.81
    Transitions:
      -> EngineOff on -(power - 0.001)
    Jumps:
      zpos <- zpos
      speed <- speed
      power <- power
```

Code Generation Principle: from Pre-Automaton to Automaton

Second step

- Sort and inline toplevel inits together.
- For each pre-automaton state
  - Inline node instantiations in equations (ODEs & dataflow).
  - Transform non-redefined toplevel inits into dataflow equations.
  - Inline dataflow equations.
  - Compute the jump of the state :
    - use nodes instantiations result (inits of automata)
    - use jump of the "pre-automaton" state.
  - Sort the final equations in a canonical order.
- Sort the toplevel inits in a canonical order.
- Convert to vector-valued representation.

. . . and some omitted other gory details.

From Pre-Automaton to Automaton (Example)

```
Node: time
  Toplevel inits:
    init t = 0.0
  State: _St0
    der t = 1.0
    Jumps:
      t <- t
Node: rocket
  Toplevel inits:
    init power = 100.0
    init speed = 0.0
    init zpos = 0.0
  State: EngineOn
    der zpos = speed
    der speed = g + power
    t = time ()
    der power = -2.0 * power
    g = -9.81
    Transitions:
      -> EngineOff on -(power - 0.001)
    Jumps:
      zpos <- zpos
      speed <- speed
      power <- power
```

```
Automaton: rocket
  Toplevel inits:
    init 1 = 100.0
    init 2 = 0.0
    init 3 = 0.0
  State: EngineOn
    der [0] = 1.0
    der [1] = -2.0 * [1]
    der [2] = -9.81 + [1]
    der [3] = [2]
    Transitions:
      -> EngineOff on - (1 - 0.001)
    Jumps:
      [0] <- 0.0
      [3] <- [3]
      [2] <- [2]
      [1] <- [1]
...
```

From Automaton to C++(Example)

```
enum StateId { EngOn, EngOff, Crashed };

int main () {
  const int dim = 3 ;
  Variable y (dim) ;

  Function EngOn_dynamics = Function (y, Return (-2 * y[0], -9.81 + y[0], y[1])) ;
  (... Idem with dynamics of states EngOff, Crashed)

  Function dyn_of_state[] = { EngOn_dynamics, EngOff_dynamics, Crashed_dynamics };

  struct tr tra_EngOn[] = { { EngOff, Function (y, - (y[0] - 0.001)) } };
  struct tr tra_EngOff[] = { { Crashed, Function (y, -y[2]) } };
  struct tr tra_Crashed[] = { };
  struct trs_set trs_EngOn = { 1, tra_EngOn } ;
  struct trs_set trs_EngOff = { 1, tra_EngOff } ;
  struct trs_set trs_Crashed = { 0, NULL } ;
  struct trs_set* trs_by_state[] = { &trs_EngOn, &trs_EngOff, &trs_Crashed };

  Function reset_EngOn = Function (y, Function (y[0], y[1], y[2])) ;
  (... Idem with resets of states EngOff, Crashed)

  Function *reset_of_state[] = { &reset_EngOn, &reset_EngOff, &reset_Crashed };
  struct automaton automaton = { dyn_of_state, trs_by_state, reset_of_state  };

  IntervalVector yinit (dim) ; yinit[0] = Interval (100.) ;
  yinit[1] = Interval (0.) ; yinit[2] = Interval (0.) ;
  if (reset_of_state[EngOn])
    yinit = (autom->reset_of_state[state])->eval_vector (yinit) ;
  SimuNode *root = run_state (&automaton, EngOn, dim, yinit, 0., GLOBAL_T_END) ;
  return 0 ;
}
```

Contracts Verification

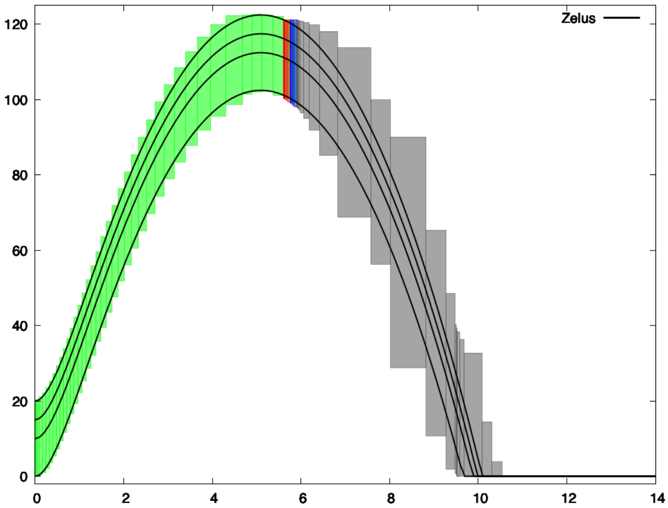
Syntax

```
{| safe x1 in [0.0, +oo] x2 in [0.0, +oo] ;  (* Interval belonging. *)
   safe x4 in [0.5, 100.0] ;
   constraint x2 -. x3 -. 1. ; |}            (* Constraint < 0. *)
```

Compilation

- Same principle than compilation of expressions.
- Generates an extra C++ `check` function :
  - Recursive traversal of the tree of simulations.
- Check each box to ensure the constraints are satisfied.
- Variables absent in a `safe` clause implicitly `in [-oo, +oo]`.

Experimental Results : Rocket (c.f. slide 5)



Other Stuff and Future Work

Remains to do

- Extend the shape of verified contracts.
- Relax syntax restrictions on Zélus accepted programs.
- Address nested (hierarchical) automata?
- Address discrete computation.

Implementation

- Compilation implemented in the Zélus compiler.
- Takes place after typing, causality check, initialization check.
- Does not break Zélus standard compilation.

Some questions ?