Falsification of Hybrid Systems using Automatic Differentiation

Journée scientifique conjointe Chaire ISC et projet DGA AID

Ismail Bennani

PhD Student, PARKAS, ENS/INRIA

supervised by

Marc Pouzet	Goran Frehse	Timothy Bourke
PARKAS, ENS/INRIA	U2IS, ENSTA Paris	PARKAS, ENS/INRIA

March 13, 2020

Summary

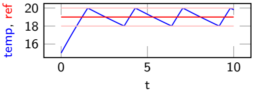
Context: Hybrid systems

Example written in Zélus [Bourke and Pouzet, 2013], a synchronous language with ODEs

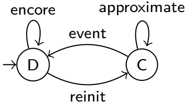
The heater

(Nicolas Halbwachs, Collège de France, 2010)

```
let low = 1.0
let high = 1.0
let c = 50.0
let α = 0.1
let β = 0.05
let tempext = 0.0
let temp0 = 15.0

let hybrid heater(u) = temp where
  rec der temp =
        if u then α *. (c -. temp)
             else β *. (tempext -. temp)
      init temp0

let hybrid relay (low, high, temp) = u where rec
  automaton
  | Low  → do u = true until up(temp -. high) then High
  | High → do u = false until up(low -. temp) then Low

let hybrid system(ref) = temp where
  rec u = relay(ref -. low, ref +. high, temp)
  and temp = heater(u)
```



Simulation algorithm of a hybrid program



- discrete phase **D**: execution of discrete reactions
- continuous phase **C**: integration of ODEs by numeric solver
- **event**: zero-crossings defined by up
- **encore**: if additional discrete step needed
- **approximate**: no event found yet

A static analysis ensures that continuous and discrete contexts are enforced

---

Note: during the continuous phase, all values are expected to be continuous. In particular, booleans should be constant.

Context: The falsification problem

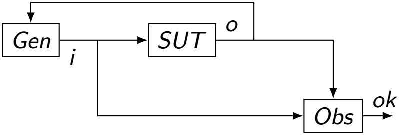
Notations
T, time domain $\quad$ $\mathbf{S_V} = T \to V$, signals
$\mathbf{I} = S_{V_{i_1}} * \ldots * S_{V_{i_n}}$, inputs set
$\mathbf{O} = S_{V_{j_1}} * \ldots * S_{V_{j_m}}$, outputs set
$\mathbf{SUT} : I \to O$
$\mathbf{Obs} : I * O \to S_{\mathbb{B}}$ $\quad$ $\mathbf{Assert} : I * O \to S_{\mathbb{B}}$
$\forall t \in T,\ T\vert_t = \{t' \vert t' \leq t\}$
$\forall s \in S_V, t \in T, s\vert_t : T\vert_t \to V$ prefix of s

Falsification problem: Given a system **SUT**, a property **Assert** over its inputs and a property **Obs** over its outputs,
find an input $i \in I$ and a time $t \in T$ such that the prefix $s|_t$ produces an output $o|_t = \mathbf{SUT}(i|_t)$ such that they verifies **Assert** but not **Obs**.

$$\exists i \in I, t \in T, o|_t = \mathbf{SUT}(i|_t),$$
$$[\forall t' \in T|_t, \mathbf{Assert}(i|_t, o|_t)(t')] \wedge$$
$$\neg \mathbf{Obs}(i|_t, o|_t)(t)$$

Common approach: write an input generator **Gen** : $O \to I$ that generates correct inputs, that is,

$$\forall o \in S^m, i = \mathbf{Gen}(o) \Rightarrow$$
$$\forall t \in T, \mathbf{Assert}(i, \mathbf{SUT}(i))(t)$$



How to find a falsifying input ?

Several tools:
Breach [Donzé, 2010]
S-TaLiRo [Annpureddy et al., 2011]
Lurette [Jahier et al., 2013]
FALSTAR [Ernst et al., 2018]

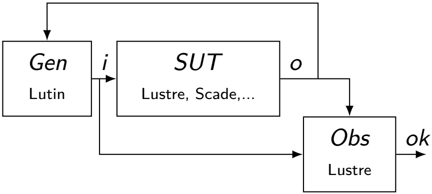
Summary

State-of-the-art: Discrete systems

Lurette [Jahier et al., 2013] [Jahier et al., 2004]

Lurette: random testing of discrete systems

- $T = \mathbb{N}$
- signals are streams
- **Assert**: linear constraints over I and O, can depend on time



Gen (Lutin)

```
node gen_x_v2() returns (x:real) =
    loop {
        (0.0<x and x<42.0) fby loop[20] x = pre x
      }

node gen_x_v3() returns (target:real; x:real=0.0) =
    run target := gen_x_v2() in
    loop { x = (pre x + target) / 2.0 }
```

Obs (Lustre observer [Halbwachs et al., 1994])

```
node true_since_n_seconds(n: real; signal: bool)
returns (res: bool)
var timer : real;
let
    timer = n →
    if not signal then n else
    max(0.0, pre(timer)-1000.0 * cycle_time);
    res = (timer = 0.0);
tel
```

State-of-the-art: Discrete systems

Lurette [Jahier et al., 2013] [Jahier et al., 2004]: Lutin [Raymond et al., 2008]

Variables of a Lutin node are booleans or numerical values, the node defines:

- constraints over booleans
- linear numerical constraints

From a partial valuation of the variables (induced by inputs and memory), the node compute a valuation that satisfies all the constraints, or fails.

Execution of the program: at each step

1. represent the constraints as a BDD with polyhedra at its leaves:
   - 1.1 gather all boolean constraints for the current step
   - 1.2 construct a BDD with them
   - 1.3 for each path in the BDD, gather the corresponding numerical constraints
   - 1.4 construct a polyhedron from those and put it as a leaf of the BDD path
2. choose a satisfiable path in the BDD with a non-empty polyhedron leaf
3. pick a random value in the polyhedron (increased probability near the borders)
4. the BDD path and the value from the polyhedron are the new valuation

Summary

State-of-the-art: Hybrid systems

S-TaLiRo [Annpureddy et al., 2011], Breach [Donzé, 2010]: optimization-based testing of hybrid systems

- $\mathbf{T} \subset \mathbb{R}$ a finite time domain, that is, a signal is an array of timed values.
- signals are computed and sampled by Simulink
- **Assert**: linear constraints over I, does not depend on time

---

New idea: quantitative semantic (called *robustness*) over properties, that is,

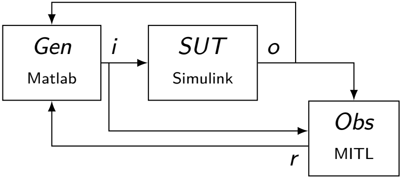
$$\mathbf{Obs} : I * O \to \mathbb{R}$$

Meaning: sign of $\mathbf{Obs}(i, o)$ is the truth value and its absolute value is a score
$\hookrightarrow$ problem becomes:

$$\exists i \in I, t \in T, o|_t = \mathbf{SUT}(i|_t), [\forall t' \in T|_t, \mathbf{Assert}(i|_t, o|_t)(t')] \wedge \mathbf{Obs}(i|_t, o|_t) < 0$$

This can be solved as a minimization problem.

We feed that robustness back to the generator: $\mathbf{Gen} : O * \mathbb{R} \to I$



State-of-the-art: Hybrid systems

S-TaLiRo [Annpureddy et al., 2011], Breach [Donzé, 2010]

Observers: Metric Interval Temporal Logic (MITL) formulas [Alur et al., 1996]

Syntax of a formula:

$$\varphi_1, \varphi_2 \; := \; \top \mid p \mid \neg\, \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \; U_{[a,b]} \; \varphi_2$$

(with v a variable, p a predicate (e.g. $(x > 0)$))

Derived operators:

$$\bot = \neg\top \qquad \Diamond_{[a,b]}\varphi = \top \; U_{[a,b]} \; \varphi \text{ (eventually)}$$

$$\varphi \vee \varphi' = \neg(\neg\varphi \wedge \neg\varphi') \qquad \Box_{[a,b]}\varphi = \neg\Diamond_{[a,b]}(\neg\varphi) \text{ (always)}$$

The robust semantic [Fainekos and Pappas, 2009] of φ is a function **Obs** $= \rho(\varphi)$:

$$\rho(\top)(i,o)(t) \; = +\infty \qquad \rho(v < f)(i,o)(t) \; = f - v(t)$$

$$\rho(\neg\,\varphi)(i,o)(t) \; = -\,\rho(\varphi)(i,o)(t) \qquad \rho(v > f)(i,o)(t) \; = v(t) - f$$

$$\rho(\varphi_1 \wedge \varphi_2)(i,o)(t) = \min(\rho(\varphi_1)(i,o)(t), \; \rho(\varphi_2)(i,o)(t))$$

$$\rho(\varphi_1 \; U_I \; \varphi_2)(i,o)(t) = \min_{t' \in (t +_R I)} \left(\max\left(\rho(\varphi_2)(i,o)(t'), \max_{t < t'' < t'} \rho(\varphi_1)(i,o)(t'') \right) \right)$$

(with v a variable in i or in o)

State-of-the-art: Hybrid systems

S-TaLiRo [Annpureddy et al., 2011]

Generator: optimization algorithm

in S-TaLiRo, by default: Simulated Annealing with Monte-Carlo sampling

```
function pick_neighbor(radius : float, point : float array) : float array;
function bernoulli(prob : float) : bool;
function update_displace_acceptance(keep_new : bool) : void;

global last_r, acceptance, displace : float;
global i, last_i : float array;

function Gen(o : float array, r : float) {
    p ← if (r < last_r) then 1. else exp((last_r − r) ∗ acceptance);
    keep_new ← bernoulli(p);

    actual_i ← if keep_new then i else last_i;
    actual_r ← if keep_new then r else last_r;
    new_i ← pick_neighbor (displace, actual_i);

    last_i ← actual_i;
    last_r ← actual_r;
    i ← new_i;

    update_displace_acceptance(keep_new);

    return new_i;
  }
```

This generator does not use the output vector of the SUT.

In Breach, by default: Nelder-Mead algorithm.

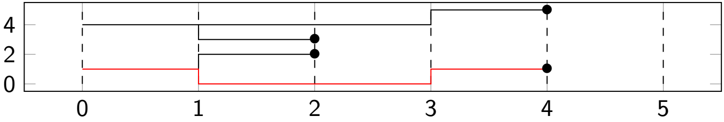
State-of-the-art: Hybrid systems

FALSTAR [Ernst et al., 2018]: tree-search based testing of hybrid systems

- $\mathbf{T} \subset \mathbb{R}$ a finite time domain
- signals are integrated by Simulink and then sampled
- **Assert**: linear constraints over I, does not depend on time
- **Obs**: interval robustness of MITL formulas over partial traces

---

Gen: explore a tree of all possible piecewise-constant inputs



For each partial input u, they compute an interval $[\underline{\rho_{\mathbf{SUT}}}(\varphi)(u), \overline{\rho_{\mathbf{SUT}}}(\varphi)(u)]$

$$\underline{\rho_{\mathbf{SUT}}}(\varphi)(u) = \min_{u' / uu' \in I} \rho(\varphi)(uu', \mathbf{SUT}(uu'))$$

$$\overline{\rho_{\mathbf{SUT}}}(\varphi)(u) = \max_{u' / uu' \in I} \rho(\varphi)(uu', \mathbf{SUT}(uu'))$$

bounding the expected robustness of the subtree starting at u. They then use that to choose whether to keep exploring the subtree or to explore another one.

Questions

Writing a specification

MITL:

- **Not intuitive** easy to make mistakes, not easy to understand.
  From [Hoxha et al., 2015]: ϕ_8^{AT}: A gear increase from first to fourth in under 10secs, ending in an RPM above `max_rpm` within 2 seconds of that, should result in a vehicle speed above `max_speed`.

  Paper formula:

$$\Big((g_1 \; U \; g_2 \; U \; g_3 \; U \; g_4) \wedge \Diamond_{[0,10]}\big(g_4 \wedge \Diamond_{[0,2]}(\omega \geq \bar{\omega})\big)\Big) \Rightarrow \Diamond_{[0,10]}(g_4 \Rightarrow \Diamond_{[0,\epsilon]}(g_4 \; U_{[0,1]} \; (v \geq \bar{v})))$$

  According to me:

$$\Big(g_1 \wedge \Diamond_{[0,10]}\big(g_4 \wedge \Diamond_{[0,2]}(\omega \geq \bar{\omega})\big)\Big) \Rightarrow \Box_{[0,10]}\big((g_4 \wedge \omega \geq \bar{\omega}) \Rightarrow v \geq \bar{v}\big)$$

- The robust semantic is computed on sampled signals by the tools.
  Can we compute it **online** ? No [Maler et al., 2006]
  that is, some formulas require non-deterministic automata (e.g. $\Box(y \Rightarrow (\Diamond_{[a,b]} x)))$

Do we really need all the expressive power of MITL to write our specifications ?

Questions

Generating inputs

The dimension of the set of (sampled) inputs of a hybrid system, is too big. How can we reduce it ?

Possible answers:

- **S-TaLiRo** and **Breach**: inputs of the system are made using statically generated parameters at the beginning of the run.
  They generate an array of timed values (user-specified length) and interpolate it.
- **FALSTAR**: compute piecewise-constant inputs with a user-specified period.

Idea: use gradient-based optimization techniques to explore the input space more efficiently.
Problem: how to compute the gradient of the robustness wrt. the inputs of the system?

Summary

Specification: Pattern templates [Gamma et al., 1995]

Do we really need all the expressive power of MITL ?

Pattern templates: enough to express most of our specifications [Frehse et al., 2018]

Pattern name	Description
Absence	After q, it is never the case that p holds $\square(q \Rightarrow \square(\neg p))$
Absence (timed)	T seconds after q is first satisfied, it is never the case that p holds $\square(q \Rightarrow \square_{[0,T]}(\neg p))$
Minimum duration	After q, it is always the case that once p becomes satisfied, it holds for at least T seconds $\square(q \Rightarrow (\square(p \; FS \; q) \Rightarrow \square_{[\epsilon,T]} p)))$
Maximum duration	After q, it is always the case that once p becomes satisfied, it holds for at most T seconds $\square(q \Rightarrow (\square(p^{\uparrow \epsilon} \Rightarrow \diamond_{[\epsilon,T]} \neg p)))$
Bounded recurrence	After q, it is always the case that p holds at least every T seconds $\square(q \Rightarrow \square(\diamond_{[0,T]} p))$
Bounded response	After q, it is always the case that if p holds, then s holds after at most T seconds $\square(q \Rightarrow \square(p \Rightarrow \diamond_{[0,T]} s))$
Bounded invariance	After q, it is always the case that if p holds then s holds for at least T seconds $\square(q \Rightarrow \square(p \Rightarrow \square_{[0,T]} s))$

with p, q predicates, that is, boolean signals
with $\varphi^{\uparrow \epsilon} = \neg\varphi \wedge \diamond_{[0,\epsilon]} \varphi$ (pronounced "up φ")
with $\varphi \; FS \; \psi = \varphi \wedge ((\neg\varphi) \; S \; \psi)$ (pronounced "first φ since ψ"")

Idea: write these these templates as hybrid observers and give them a quantitative semantics

Synchronous observers [Halbwachs et al., 1994]

Synchronous observer: discrete boolean node that encodes a specification.

Example: B has been true at least once between A and C

```
node never p = neverp where
    rec neverp = not p → (pre neverp && not p)

node since(a, b) = asinceb where
    asinceb = if b then a
    else (true → a or pre(asinceb))

node onceBfromAtoC(a, b, c) =
    c ⇒ (never a || since(b, a))
```

⚠ "since(a, b)" here stands for "a has been true **at least once** since **last time** b has been true" which is not the same as the usual meaning of the operator S in MITL.

Synchronous observers with quantitative semantics

Discrete time

```
let qnot p = -. p              let qand (a, b) = min a b
let qor (a, b) = max a b       let qimply (a, b) = qor (qnot a, b)

node never p = neverp where
rec neverp = qnot p → qand (pre neverp, qnot p)

node since(a, b) = ... (* next slide *)

node onceBfromAtoC(a, b, c) =
qimply(c, qor (never a, since(b, a)))
```

Synchronous observers with quantitative semantics

Discrete time

```
node once p = oncep where rec oncep = p → qor (pre oncep, p)
node since(a, b) = asinceb where
automaton
    | INIT →
        do asinceb = qor (never b, a)
        until (qistrue b) () then BTRUE
    | BTRUE →
        do asinceb = a
        until (qisfalse b) () then BFALSE(a)
    | BFALSE(lastres) →
        do asinceb = qor (lastres, once a)
        until (qistrue b) () then BTRUE
```

Hybrid observers

Continuous time: always and once

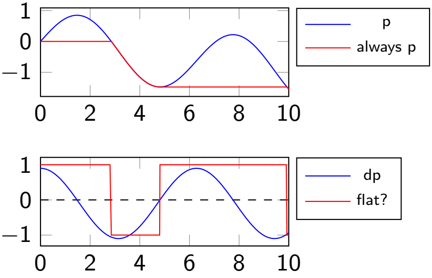
Discrete time:

```
node always p = alwaysp where rec alwaysp = p → min p (pre alwaysp)
```

Continuous time ? Possible solution:

- if we have access to the time-derivative dp of p wrt. time:

```
hybrid always (p, dp) = alwaysp where
    (* flat : if true then alwaysp should not change *)
    rec flat =
        present
            (* follow the derivative if *)
            (* p becomes < alwaysp *) | xup(alwaysp -. p)
            | (disc dp) on (p < alwaysp && dp < 0.)
            → false
            (* stop following it if *)
            (* dp becomes > 0 *)        | xup(dp)
            | (disc p) on (p > alwaysp)
            → true
        init dp ≥ 0.
    and der alwaysp = if flat then 0. else dp
        init p
        reset (disc p) → min p (last alwaysp)
```



- we can compute dp automatically by using automatic differentiation (here, with dual numbers for example)
- problem: requires to use dual numbers instead of normal floats

Hybrid observers

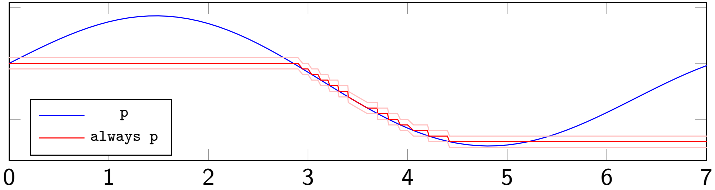
Continuous time: always and once

Discrete time:

```
node always p = alwaysp where rec alwaysp = p → min p (pre alwaysp)
```

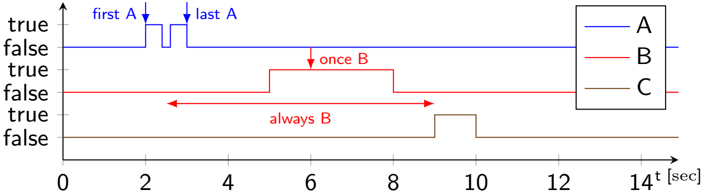
Continuous time ? Simpler solution: sample the signal

```
let static eps = 0.1
    hybrid always p = alwaysp where
    rec init alwaysp = p
    and der t = 1. init 0.
    and present
        (* compute new min when *)
        (* | current min p - p | > eps *)
        | xup((alwaysp -. eps) -. p) | xup(p -. (alwaypp +. eps))
        (* p crosses 0 *)
        | xup(p) | xup(-.p) →
            do alwaysp = min ((last alwaysp), p) done
```



Specification: A library of hybrid observers

How to specify the outputs



after	first last	A holds,	once always	B holds until C holds

↪ these 4 constructions are enough to express all the properties in these benchmarks: [Ernst et al., 2019] [Dokhanchi et al., 2018] [Hoxha et al., 2014]
↪ they are enough to express all the pattern templates of [Frehse et al., 2018]
↪ they express a subset of MITL

My contribution on this: quantitative semantics to synchronous observers + continuous version with quantitative semantics

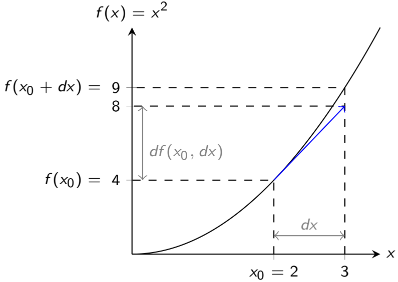
Summary

Preliminary: Differentiation

Definition of a differential

Let $f : \mathbb{R}^n \to \mathbb{R}^m$ be a function differentiable in $\mathbb{R}^n$ and
$\forall i \in [1, m], f_i(x_1, ..., x_n) = (f(x_1, ..., x_n))_i$.
The differential function $df : \mathbb{R}^{2n} \to \mathbb{R}^m$ is defined by

$$
\begin{aligned}
df(x_1, dx_1, ..., x_n, dx_n) = (&\frac{\partial f_1}{\partial x_1} dx_1 + ... + \frac{\partial f_1}{\partial x_n} dx_n, \\
&..., \\
&\frac{\partial f_m}{\partial x_1} dx_1 + ... + \frac{\partial f_m}{\partial x_n} dx_n)
\end{aligned}
$$



$f(x_1, ..., x_n) + df(x_1, dx_1, ..., x_n, dx_n)$
is a first order approximation of
$f(x_1 + dx_1, ..., x_n + dx_n)$

FADBADml

Automatic Differentiation

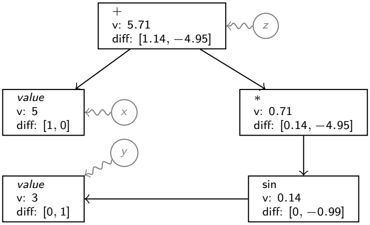
FADBADml:

- OCaml porting of FADBAD++ [Stauning, 1997] (written by Ole Stauning)
- written by François Bidet and myself
- available on github: https://github.com/fadbadml-dev/FADBADml

---

Example:

```
let x = make 5 in
let y = make 3 in
diff x 0 2;
diff y 1 2;
let
z = x + sin y * x
in ...
(* access dz/dx
and dz/dy
with ‘d z 0‘
and ‘d z 1‘ *)
```



- Can be used for discrete nodes to compute differentials without source-to-source transformation.
- For hybrid nodes, we would need to provide an API to the external ODE solver.

A toy language (WIP)

Syntax (to be extended)

- This is a subset of Zélus, it will be compiled into Zélus
- All variables are of type float, booleans are given a quantitative semantic (same idea as robustness of MITL). I am working on extending this kernel.
- Primitive combinatorial functions ($+$, sin, ...) are expected to be differentiable wrt. their input. They also come with their differential (e.g. `let sin_d (x, dx) = (sin x, dx*cos(x)), ...`).
- Other combinatorial functions such as *abs* (not differentiable in 0) can be used if implemented as a node:

```
let hybrid abs(x) = (y) where
present xup x → sign = 1.
else xup (-.x) → sign = -1.
init sign = (if x ≥ 0. then 1. else -1.)
and y = sign *. x
```

How to compute the differential of a node ?

$$impl ::= \texttt{let}\ kind\ id\ (id(,\ id)^*) = (id(,\ id)^*)\ \texttt{where}\ eq$$

$$kind ::= \texttt{fun} \mid \texttt{node} \mid \texttt{hybrid}$$

$$\begin{aligned} eq ::=\ & id(,\ id)^* = exp \\ & \mid \texttt{der}\ id = exp\ \texttt{init}\ exp \\ & \mid \texttt{if}\ exp\ \texttt{then}\ eq\ \texttt{else}\ eq \\ & \mid eq\ \texttt{and}\ eq \\ & \mid \texttt{present}\ (\mid cond \to eq)^{+} \texttt{else}\ eq \\ & \mid \texttt{automaton}\ (\mid state \to \texttt{do}\ eq\ transition)^{+} \end{aligned}$$

$$transition ::= \texttt{done} \mid \texttt{until}\ exp\ \texttt{then}\ state$$

$$\begin{aligned} exp ::=\ & \texttt{true} \mid const \mid x \mid (exp) \\ & \mid \texttt{let}\ eq\ \texttt{in}\ exp \\ & \mid id(exp(,\ exp)^*) \\ & \mid \texttt{pre}\ exp \mid exp \to exp \\ & \mid exp > 0 \mid \texttt{up}\ exp \\ & \mid \texttt{not}\ exp \mid exp\ b\_op\ exp \end{aligned}$$

$$b\_op ::= \&\& \mid || \mid \texttt{on}$$

Differentiation operator

$$D(\text{let } kind\ f\ (x_1, \ldots, x_n) = (y_1, \ldots, y_m) \text{ where } eq) \stackrel{def}{=} \text{let } kind\ df(x_1, dx_1, \ldots, x_n, dx_n) = (y_1, dy_1, \ldots, y_m, dy_m) \text{ where } D^{eq}(eq)$$

$$D^{eq}((x_1, \ldots, x_n) = e) \stackrel{def}{=} (x_1, dx_1, \ldots, x_n, dx_n) = D^{eq}(e)$$

$$D^{eq}(\text{if } e \text{ then } eq_1 \text{ else } eq_2) \stackrel{def}{=} \text{if } istrue(e) \text{ then } D^{eq}(eq_1) \text{ else } D^{eq}(eq_2)$$

$$D^{eq}(\text{der } x = e_1 \text{ init } e_2) \stackrel{def}{=} \left[\begin{array}{l} (x_d, dx_d) = D^{eq}(e_1) \\ \text{and } (x_0, dx_0) = D^{eq}(e_2) \\ \text{and der } x = x_d \text{ init } x_0 \\ \text{and der } dx = dx_d \text{ init } dx_0 \end{array}\right.$$

$$D^{eq}(eq_1 \text{ and } eq_2) \stackrel{def}{=} D^{eq}(eq_1) \text{ and } D^{eq}(eq_2)$$

$$D^{eq}(\text{present } (\mid e_i \rightarrow eq_i)_{0 \leq i < n} \text{ else } eq_n) \stackrel{def}{=} \text{present } (\mid e_i \rightarrow D^{eq}(eq_i))_{0 \leq i < n} \text{ else } D^{eq}(eq_n)$$

$$D^{exp}(const) \stackrel{def}{=} (const, 0)$$

$$D^{exp}(true) \stackrel{def}{=} (+\infty, 0)$$

$$D^{exp}(\text{not } e) \stackrel{def}{=} -D^{exp}(e)$$

$$D^{exp}(e > 0) \stackrel{def}{=} D^{exp}(e)$$

$$D^{exp}((x_1, \ldots, x_n)) \stackrel{def}{=} (x_1, dx_1, \ldots, x_n, dx_n)$$

$$D^{exp}(\text{up } e) \stackrel{def}{=} \left[\begin{array}{l} \text{let } (v, dv) = D^{exp}(e) \\ \text{in } (\text{up } v, dv) \end{array}\right.$$

$$D^{exp}(\text{pre } e) \stackrel{def}{=} (\text{pre } e, 0)$$

$$D^{exp}(\text{let } eq \text{ in } exp) \stackrel{def}{=} \text{let } D^{eq}(eq) \text{ in } D^{exp}(exp)$$

$$D^{exp}(x_1, \ldots, x_n = e)) \stackrel{def}{=} x_1, dx_1, \ldots, x_n, dx_n = D^{exp}(e)$$

$$D^{exp}(f(e_1, \ldots e_n))) \stackrel{def}{=} \left[\begin{array}{l} \text{let } (v_1, dv_1) = D^{exp}(e_1) \\ \text{and } \ldots \\ \text{and } (v_n, dv_n) = D^{exp}(e_n) \\ \text{in } df(v_1, dv_1, \ldots, v_n, dv_n) \end{array}\right.$$

$$D^{exp}(e_1 \ \&\& \ e_2) \stackrel{def}{=} \left[\begin{array}{l} \text{let } (v_1, dv_1) = D^{exp}(e_1) \\ \text{and } (v_2, dv_2) = D^{exp}(e_2) \\ \text{and if } v_1 - v_2 > 0 \\ \qquad \text{then } v = v_2 \text{ and } dv = dv_2 \\ \qquad \text{else } v = v_1 \text{ and } dv = dv_1 \\ \text{in}(v, dv) \end{array}\right.$$

$$D^{exp}(e_1 \text{ on } e_2) \stackrel{def}{=} D^{exp}(e_1 \mid\mid (\text{not } e_2))$$

$$D^{exp}(e_1 \rightarrow e_2) \stackrel{def}{=} D^{exp}(e_1) \rightarrow D^{exp}(e_2)$$

$$D^{exp}(e_1 \mid\mid e_2) \stackrel{def}{=} D^{exp}(\text{not } ((\text{not } e_1) \ \&\& \ (\text{not } e_2)))$$

Differentiation operator

Important cases

Assumption: All available combinatorial functions ($+$, sin, ...) are differentiable wrt. their inputs.
$\Rightarrow$ during a **continuous phase**, all signals are differentiable.

$$D^{eq}(\texttt{der}\ x = e_1\ \texttt{init}\ e_2) := \begin{bmatrix} (x_d, dx_d) = D^{eq}(e_1) \\ \texttt{and}\ (x_0, dx_0) = D^{eq}(e_2) \\ \texttt{and der}\ x = x_d\ \texttt{init}\ x_0 \\ \texttt{and der}\ dx = dx_d\ \texttt{init}\ dx_0 \end{bmatrix}$$

Leibniz integral rule

Let f be such that $f(x, t)$ and $\frac{\partial f}{\partial x}(x, t)$ are continuous in $([0, t] * \mathcal{X})$, then

$\forall x \in \mathcal{X}$

$$F(x) = \int_0^t f(x, h)dh \;\Rightarrow\; \frac{\partial F}{\partial x}(x) = \int_0^t \frac{\partial f}{\partial x}(x, h)dh$$

In our case, the program `der x = f(x,y) init x0` encodes the fixpoint equation

$$x(y, t) = x_0 + \int_0^t f(x, y, h)dh$$

so, following the Leibniz rule, its partial derivative is

$$\frac{\partial x}{\partial y}(y, t) = \int_0^t \frac{\partial f}{\partial y}(x, y, h)dh$$

that we encode as

```
der dx = df(x, dx, y, dy) init 0.
```

Differentiation operator

Important cases

Assumption: All available combinatorial functions ($+$, sin, ...) are differentiable wrt. their inputs.
$\Rightarrow$ during a **continuous phase**, all the signals are differentiable.

$$D^{eq}(\texttt{der}\ x = e_1\ \texttt{init}\ e_2) := \left[\begin{array}{l} (x_d, dx_d) = D^{eq}(e_1) \\ \texttt{and}\ (x_0, dx_0) = D^{eq}(e_2) \\ \texttt{and der}\ x = x_d\ \texttt{init}\ x_0 \\ \texttt{and der}\ dx = dx_d\ \texttt{init}\ dx_0 \end{array}\right.$$

Leibniz integral rule

Let f be such that $f(x, t)$ and $\frac{\partial f}{\partial x}(x, t)$ are continuous in $([0, t] * \mathcal{X})$, then

$\forall x \in \mathcal{X}$
$$F(x) = \int_0^t f(x, h)dh \Rightarrow \frac{\partial F}{\partial x}(x) = \int_0^t \frac{\partial f}{\partial x}(x, h)dh$$

In our case, the program `der x = f(x,y) init` x_0 encodes the fixpoint equation
$$x(y, t) = x_0 + \int_0^t f(x, y, h)dh$$
so, following the Leibniz rule, its partial derivative is
$$\frac{\partial x}{\partial y}(y, t) = \int_0^t \frac{\partial f}{\partial y}(x, y, h)dh$$
that we encode as
```
der dx = df(x, dx, y, dy) init 0.
```

$$D^{exp}(\texttt{pre}\ e) := (\texttt{pre}\ e, 0)$$

we want the differential of pre e wrt. the current inputs of the node. pre e does not depend on the current inputs, so its differential is 0.

$$D^{exp}(\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2) := \texttt{if}\ istrue(e)\ \texttt{then}\ D^{exp}(e_1)\ \texttt{else}\ D^{exp}(e_2)$$

- *istrue* is a Zelus node that listens for the events up(e) and up($-.e$) to compute a boolean from the value of e.
  In particular, its output is constant during a continuous phase.
- if e does not depend on the values of the variables wrt. which we differentiate, this differential is correct
  otherwise, weird things can happen:

```
let node mySqr(x) = y where
    if x = 5. then y = 25.
    else y = x *. x
```

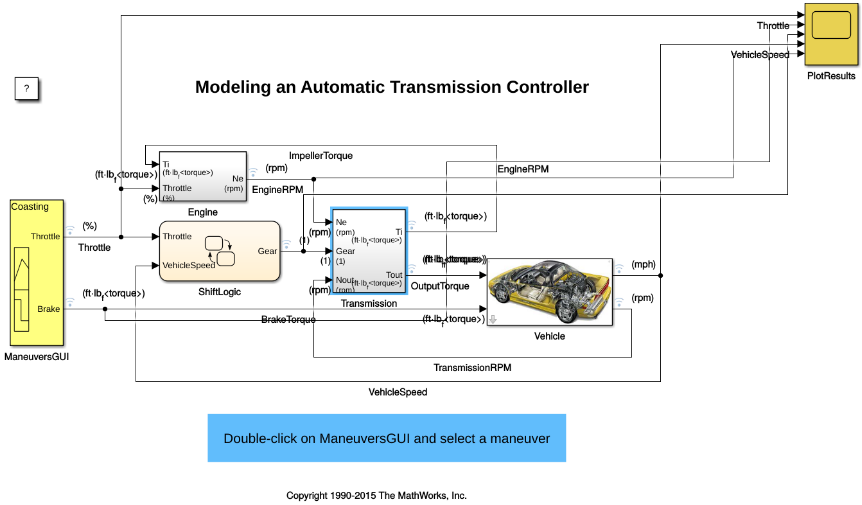
would become

```
let node dmySqr(x, dx) = y, dy where
    if x = 5.
        then y = 25. and dy = 0.
        else y = x *. x
    and dy = 2. *. dx *. x
```

Summary

Falsifying a specification using differentials (WIP)

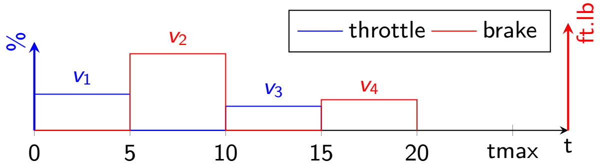
The automatic transmission benchmark [Hoxha et al., 2015]



Input: `throttle`, `brake` **Output**: `gear`, `speed`, `rpm`

Falsifying a specification using differentials (WIP)

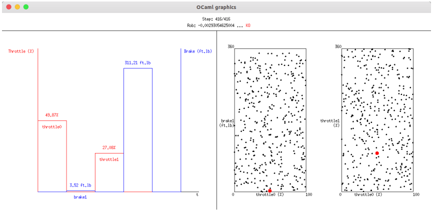
The automatic transmission benchmark [Hoxha et al., 2015]

Input shape (dim. 4)	
Specification	If the engine speed never reaches $\bar{w}$ in the first 30 seconds, then the vehicle speed never reaches $\bar{v}$ in the first d seconds. $(\square_{[0,30]}(w < \bar{w})) \Rightarrow (\square_{[0,d]}(v < \bar{v}))$
Robustness	$\rho(w, v)(t) = \max(-\min_{t\in[0,30]}(\bar{w} - w(t)), \min_{t\in[0,d]}(\bar{v} - v(t)))$
Observer	`qimplies(always_timed 30 (qle (w, w_bar)), always_timed d (qle (v, v_bar)))`

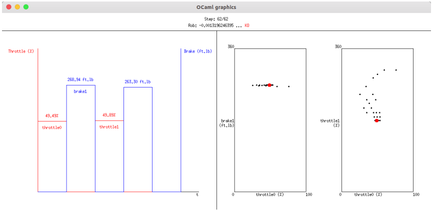
Falsifying a specification using differentials (WIP)

The automatic transmission: **discrete version**

Uniform Random



Gradient descent



Summary

Falsifying a specification using differentials (WIP)

The heater (transformed by hand)

```
let low = 1.0
let high = 1.0
let c = 50.0
let α = 0.1
let β = 0.05
let tempext = 0.0
let temp0 = 15.0

let hybrid heater(u) = temp where
  rec der temp =
        if u then α *. (c -. temp)
              else β *. (tempext -. temp)
      init temp0

let hybrid relay (low, high, temp) = u where rec
  automaton
  | Low  → do u = true until up(temp -. high) then High
  | High → do u = false until up(low -. temp) then Low

let hybrid system(ref) = temp where
  rec u = relay(ref -. low, ref +. high, temp)
  and temp = heater(u)
```

```
let low_d = 0.000000
let high_d = 0.000000
let c_d = 0.000000
let α_d = 0.000000
let β_d = 0.000000
let tempext_d = 0.000000
let temp0_d = 0.000000
let h_d = 0.000000

let hybrid heater_d (u) = temp, temp_d where
  rec der temp =
        if u then (α *. (c -. temp))
             else (β *. (tempext -. temp))
      init temp0
  and der temp_d =
        if u then
          ((α_d *. (c -. temp)) +.
           (α *. (c_d -. temp_d)))
        else
          ((β_d *. (tempext -. temp)) +.
           (β *. (tempext_d -. temp_d)))
      init temp0_d

let hybrid relay (low, high, v) = u where rec
  automaton
  | Low  → do u = true until up(v -. high) then High
  | High → do u = false until up( low -. v) then Low

let hybrid qsystem_d (ref, ref_d) = temp, temp_d where
  rec u = relay(ref -. low, ref +. high, temp)
  and temp, temp_d = heater_d(u)
```

Falsifying a specification using differentials (WIP)

The heater (transformed by hand)

```
let low = 1.0
let high = 1.0
let c = 50.0
let α = 0.1
let β = 0.05
let tempext = 0.0
let temp0 = 15.0

let hybrid heater(u) = temp where
  rec der temp =
        if u then α *. (c -. temp)
             else β *. (tempext -. temp)
      init temp0

let hybrid relay (low, high, temp) = u where rec
  automaton
  | Low  → do u = true until up(temp -. high) then High
  | High → do u = false until up(low -. temp) then Low

let hybrid system(ref) = temp where
  rec u = relay(ref -. low, ref +. high, temp)
  and temp = heater(u)
```

```
let low_d = 0.000000
let high_d = 0.000000
let c_d = 0.000000
let α_d = 0.000000
let β_d = 0.000000
let tempext_d = 0.000000
let temp0_d = 0.000000
let h_d = 0.000000

let hybrid heater_d (u) = temp, temp_d where
  rec der temp =
        if u then (α *. (c -. temp))
             else (β *. (tempext -. temp))
      init temp0
  and der temp_d =
        if u then
          ((α_d *. (c -. temp)) +.
           (α *. (c_d -. temp_d)))
        else
          ((β_d *. (tempext -. temp)) +.
           (β *. (tempext_d -. temp_d)))
      init temp0_d

let hybrid relay (low, high, v) = u where rec
  automaton
  | Low  → do u = true until up(v -. high) then High
  | High → do u = false until up( low -. v) then Low

let hybrid qsystem_d (ref, ref_d) = temp, temp_d where
  rec u = relay(ref -. low, ref +. high, temp)
  and temp, temp_d = heater_d(u)
```

Problem:

$$d\dot{temp}(t) = \begin{cases} 0 & \text{if } t = 0 \\ d\alpha * (c - temp(t)) + \alpha * (dc - dtemp(t)) & \text{else if u} \\ d\beta * (tempext - temp(t)) + \beta * (dtempext - dtemp(t)) & \text{else} \end{cases}$$

$$= \begin{cases} 0 & \text{if } t = 0 \\ -\alpha * dtemp(t) & \text{else if u} \\ -\beta * dtemp(t) & \text{else} \end{cases}$$

has a unique solution $\forall t, dtemp(t) = 0$. If $dtemp = 0$, no matter what *spec* you define, *dspec* will also be 0.
$\Rightarrow$ differentials do not guide us

Switching modes using differentials (WIP)

The quantitative heater (transformed by my compiler)

Point of interest: discontinuities of `u`
We can compute a quantitative semantic `qu` for the boolean `u`. `qu` satisfies the sufficient condition because there is no `if` (or `present`) between `ref` and `qu`.

```
let low = 1.0
let high = 1.0
let c = 50.0
let α = 0.1
let β = 0.05
let tempext = 0.0
let temp0 = 15.0

let hybrid heater(qu) = temp where
  rec der temp =
        if qu
             then α *. (c -. temp)
             else β *. (tempext -. temp)
      init temp0

let hybrid relay (low, high, v) = qu where rec
  automaton
  | Low  → do qu = (high -. v) until up(-.qu) then High
  | High → do qu = (low -. v) until up(qu) then Low

let hybrid system(ref) = qu, temp where
  rec qu = relay(ref -. low, ref +. high, temp)
  and temp = heater(qu)
```

```
let low_d = 0.000000
let high_d = 0.000000
let c_d = 0.000000
let α_d = 0.000000
let β_d = 0.000000
let tempext_d = 0.000000
let temp0_d = 0.000000
let h_d = 0.000000

let hybrid heater_d (qu, qu_d) = temp, temp_d where
  rec der temp =
        if istrue(qu)
             then (α *. (c -. temp))
             else (β *. (tempext -. temp))
      init temp0
  and der temp_d =
        if istrue(qu) then
          ((α_d *. (c -. temp)) +.
           (α *. (c_d -. temp_d)))
        else
          ((β_d *. (tempext -. temp)) +.
           (β *. (tempext_d -. temp_d)))
      init temp0_d

let hybrid qrelay_d (low, low_d, high, high_d, v, v_d) = qu, qu_d where rec
  automaton
  | Low → do
                qu = (high -. v)
            and qu_d = (high_d -. v_d)
          until up(-.qu) then High
  | High → do
                qu = (low -. v)
            and qu_d = (low_d -. v_d)
           until up(qu) then Low

let hybrid qsystem_d (ref, ref_d) = qu, qu_d, temp, temp_d where
  rec qu, qu_d =
      qrelay_d ((ref -. low), (ref_d -. low_d),
                (ref +. high), (ref_d +. high_d),
                temp, temp_d)
  and temp, temp_d = heater_d(qu, qu_d)
```

Now that we have `qu_d`, we can use gradient descent to make `qrelay` go from one state to the other and try to find a bug this way.

Switching modes using differentials (WIP)

The main loop

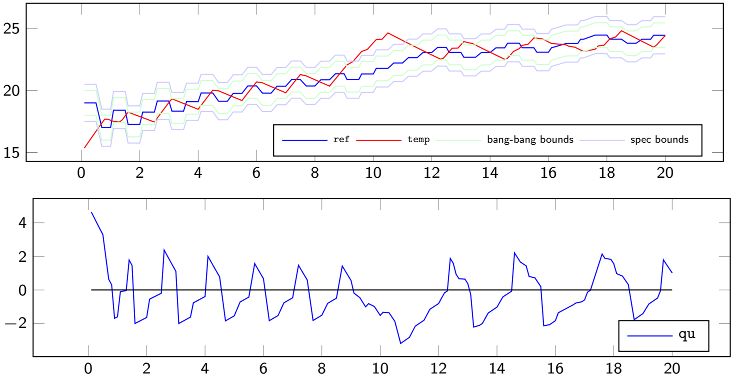
```
let ref0 = 19.
let α = 2.
let inp_step = 0.5 (* period of sampling of the input *)
let plot_step = 0.1 (* period of sampling of the plot *)

let node grad_descent (ref, grad) = new_ref where
    rec acc_grad = grad *. grad → (pre acc_grad +. grad *. grad)
    and new_ref = ref +. α *. grad /. (Pervasives.sqrt acc_grad)

let hybrid main () =
    let der t = 1. init 0. in

    let rec qu, qu_d, temp, temp_d = qsystem_d (ref, 1.)

    and init ref = ref0
    and present (period(inp_step)) →
        do
            next ref = grad_descent (ref, if qu > 0. then -.qu_d else qu_d)
        done
    in

    present (period(plot_step)) →
        plot (ref, t, (qu, qu_d, temp, temp_d))
    else ()
```

Remark: we are not trying to falsify a property here, we are trying to trigger mode switches.

Switching modes using differentials (WIP)

The heater

Spec: Between 4s and 20s, the temperature stays between $ref - 1.5$ and $ref + 1.5$. FALSIFIED



References I

Alur, R., Feder, T., and Henzinger, T. A. (1996).
The benefits of relaxing punctuality.
J. ACM, 43(1):116–146.

Annpureddy, Y., Liu, C., Fainekos, G., and Sankaranarayanan, S. (2011).
S-taliro: A tool for temporal logic falsification for hybrid systems.
In Abdulla, P. A. and Leino, K. R. M., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 254–257, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bourke, T. and Pouzet, M. (2013).
Zélus: A Synchronous Language with ODEs.
In Belta, C. and Ivančić, F., editors, *HSCC - 16th International Conference on Hybrid systems: computation and control*, Proceedings of the 16th International Conference on Hybrid systems: computation and control, pages 113–118, Philadelphia, United States. Calin Belta and Franjo Ivančić, ACM.

Dokhanchi, A., Yaghoubi, S., Hoxha, B., Fainekos, G., Ernst, G., Zhang, Z., Arcaini, P., Hasuo, I., and Sedwards, S. (2018).
Arch-comp18 category report: Results on the falsification benchmarks.
In Frehse, G., editor, *ARCH18. 5th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 54 of *EPiC Series in Computing*, pages 104–109. EasyChair.

Donzé, A. (2010).
Breach, a toolbox for verification and parameter synthesis of hybrid systems.
In Touili, T., Cook, B., and Jackson, P., editors, *Computer Aided Verification*, pages 167–170, Berlin, Heidelberg. Springer Berlin Heidelberg.

References II

Ernst, G., Arcaini, P., Donz\'e, A., Fainekos, G., Mathesen, L., Pedrielli, G., Yaghoubi, S., Yamagata, Y., and Zhang, Z. (2019).
Arch-comp 2019 category report: Falsification.
In Frehse, G. and Althoff, M., editors, *ARCH19. 6th International Workshop on Applied Verification of Continuous and Hybrid Systems*, volume 61 of *EPiC Series in Computing*, pages 129–140. EasyChair.

Ernst, G., Sedwards, S., Zhang, Z., and Hasuo, I. (2018).
Fast falsification of hybrid systems using probabilistically adaptive input.
CoRR, abs/1812.04159.

Fainekos, G. E. and Pappas, G. J. (2009).
Robustness of temporal logic specifications for continuous-time signals.
Theoretical Computer Science, 410(42):4262 – 4291.

Frehse, G., Kekatos, N., Nickovic, D., Oehlerking, J., Schuler, S., Walsch, A., and Woehrle, M. (2018).
A toolchain for verifying safety properties of hybrid automata via pattern templates.
In *2018 Annual American Control Conference (ACC)*, pages 2384–2391.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995).
Design Patterns: Elements of Reusable Object-Oriented Software, pages 360–365.
Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, MA.

Halbwachs, N., Lagnier, F., and Raymond, P. (1994).
Synchronous observers and the verification of reactive systems.
In *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology*, AMAST '93, pages 83–96, Berlin, Heidelberg. Springer-Verlag.

References III

Hoxha, B., Abbas, H., and Fainekos, G. (2015).
Benchmarks for temporal logic requirements for automotive systems.
In Frehse, G. and Althoff, M., editors, *ARCH14-15. 1st and 2nd International Workshop on Applied veRification for Continuous and Hybrid Systems*, volume 34 of *EPiC Series in Computing*, pages 25–30. EasyChair.

Hoxha, B., Abbas, H., and Fainekos, G. E. (2014).
Benchmarks for temporal logic requirements for automotive systems.
In *ARCH@CPSWeek*.

Jahier, E., Halbwachs, N., and Raymond, P. (2013).
Engineering functional requirements of reactive systems using synchronous languages.
In *International Symposium on Industrial Embedded Systems, 2013. SIES'13.*, Porto, Portugal.

Jahier, E., Raymond, P., and Baufreton, P. (2004).
Case studies with lurette v2.
In *1st International Symposium on Leveraging Applications of Formal Methods, ISoLA 2004*, Paphos, Cyprus.

Maler, O., Nickovic, D., and Pnueli, A. (2006).
From mitl to timed automata.
In Asarin, E. and Bouyer, P., editors, *Formal Modeling and Analysis of Timed Systems*, pages 274–289, Berlin, Heidelberg. Springer Berlin Heidelberg.

Raymond, P., Roux, Y., and Jahier, E. (2008).
Lutin: a language for specifying and executing reactive scenarios.
EURASIP Journal on Embedded Systems, 2008.
http://jes.eurasipjournals.com/content/2008/1/753821.

References IV

Stauning, O. (1997).
Automatic validation of numerical solutions.
PhD thesis.