

A Lutin Tutorial

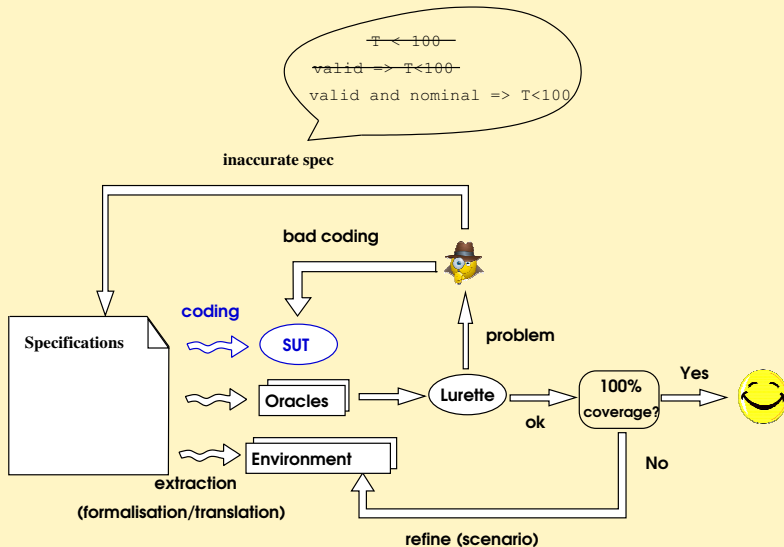
Erwan Jahier

8 décembre 2011

Plan

- 1 Forewords
- 2 Execute Lutin programs
- 3 The Language
- 4 A new operator `run/in`
- 5 Advanced examples

Motivations : testing reactive programs



Lutin in one slide

- Lustre-like : Dataflow, parallelism, modular, logic time, pre.
- But not exactly Lustre though
 - Plus
 - Control structure operators (regular expressions)
 - Stochastic (controlled and pseudo-Aleatory)
 - Minus
 - No implicit top-level loop
 - No topological sort of equations

In order to run this tutorial

- You will need to install
 - Lutin
 - Tcl/tk
 - Gnuplot
- you'll find on the Verimag website
 - The tools
- An html wordier version of those slides

nb : in order to run the demo from the pdf slides you can edit your .xpdfrc resource file modify the urlCommand rule :

```
urlCommand      "browserhook.sh '%s' "
```

and make the bash script browserhook.sh executable and available from your path

Stimulate Lutin programs

- A program with no input

Consider the one.lut program :

```
node one() returns (y:int) =  
  loop y = 1
```

```
<prompt> lutin -l 5 one.lut
```

- Be quiet

```
<prompt> lutin -l 5 -quiet one.lut
```

- Stimulate Lutin programs graphically with `luciole`

```
<prompt> lutin -luciole incr.lut
```

Store and Display the produced data : `sim2chro` and `gnuplot-rif`

```
node N() returns (y:int) =
  y = 0 fby loop y = pre y+1
```

- Generate a RIF file

It is possible to store the lutin RIF output into a file using the `-rif` option.

```
<prompt> lutin -l 10 -rif ten.rif N.lut; ls -lh ten.rif
```

- Visualize a RIF file

```
<prompt> cat ten.rif
```

- Visualize a RIF file (bis)

```
<prompt> cat ten.rif | sim2chrogtk -ecran > /dev/null
```

- Visualize a RIF file (ter)

```
<prompt> gnuplot-rif ten.rif
```

Automatic stimulation of Lutin programs

```
node incr(x:int) returns (y:int) =
  loop y = x+1
node decr(y:int) returns (x:int) =
  x = 42 fby loop x = y-1
```

```
<prompt> lurettetop -l 20 -rp "sut:lutin:incr.lut:incr" -rp
"env:lutin:decr.lut:decr"
```



- I've bought 2 electronic chess games
- connected one to another
- And now I have peace

Back to programs of Section 1

- Let's come back to the Lutin programs mentioned so far.

```
node incr(x:int) returns (y:int) =  
  loop [10] y = x+1
```

- Those programs illustrate the 2 kinds of expressions we have in Lutin.
 - **constraint expressions** ($y = x+1$) that asserts facts outputs variables.
 - **trace expression** (`loop <te>`) that allows one to combine constraint expressions.

Non deterministic programs

- A First non-deterministic program

```
node trivial() returns(x:int; f:real ; b:bool) =
  loop true
```

```
<prompt> lutin -l 10 -q trivial.lut
```

- It is possible to set the variable range at declaration time, as done in trivial2.lut :

```
node trivial() returns(x:int [-100;100];
                      f:real [-100.0;100.0];b:bool)=
  loop true
```

```
<prompt> lutin -l 10 -q trivial2.lut
```

Non deterministic programs (cont)

Now consider the range.lut program :

```
node range() returns (y:int) = loop 0 <= y and y <= 42
```

```
<prompt> lutin range.lut -l 10 -q
```

- Linear constraints → union of convex polyhedra
- Several heuristics are defined to perform the solution draw
- `--step-inside (-si)` : draw inside the polyhedra (the default)
- `--step-vertices (-sv)` draw among the polyhedra vertices
- `--step-edges (-se)` : promote edges

```
<prompt> lutin range.lut -l 10 -q -step-vertices
```

- Question : write a simpler program than range.lut that behaves the same.

Non deterministic programs (cont)

- A 3D non-deterministic example

```
node polyhedron() returns(a,b,c:real) =
  loop (0.0 < c and c < 4.0 and
        a + 1.0 * b > 0.0 and
        a + 1.0 * b - 3.0 < 0.0 and
        a - 1.0 * b < 0.0 and
        a - 1.0 * b + 3.0 > 0.0)
```

```
<prompt> lutin polyhedron.lut -l 1000 -q > poly.data;
echo 'set point 0.2;set term wxt persist;plot "poly.data"
using 1:2:3'| gnuplot
```

- One can observe the effect of `-step-edges` and `-step-vertices` options on the repartition of generated points

Non deterministic programs (cont)

Constraint may also depend on inputs. Try to play the range-bis.lut program :

```
node range_bis(i:int) returns (y:int) =  
  loop 0 <= y and y <= i
```

```
<prompt> lutin range-bis.lut -luciole
```

The major difference with range.lut is that the constraint expression $0 \leq y$ and $y \leq i$ is not always satisfiable. If one enters a negative value, that program will stop.

Controlled non-determinism : the choice operator

```
node choice() returns(x:int) =
  loop {
    | x = 42
    | x = 1
  }
```

<prompt> `lutin -l 10 -q choice.lut`

It is possible to favor one branch over the other using weight directives (:3):

```
node choice() returns(x:int) =
  loop {
    |3: x = 42
    |1: x = 1
  }
```

In `choice2.lut`, `x=42` is chosen with a probability of 3/4.

<prompt> `lutin -l 10000 -q choice2.lut | grep 42 | wc -l`

Controlled non-determinism : the choice operator

```
node choice3(b:bool) returns(x:int) =  
  loop {  
    | x = 1 and b  
    | x = 2 and b  
    | x = 3 and not b  
  }
```

```
<prompt> lutin -luciole choice3.lut
```

Local variables

Sometimes, it is useful to use auxiliary variables that are not output variables. Such variables can be declared using the `exist/in` construct. Its use is illustrated in the `true_since_n_instants.lut` program :

```
let n = 3
node ok_since_n_instants(b:bool) returns (res:bool) =
  exist cpt: int = n in
    loop {
      cpt = (if b then (pre cpt-1) else n) and
      res = (b and (cpt <= 0))
    }
```

```
<prompt> lutin true_since_n_instants.lut -luciole
```

Local variables again

Local variables can also plain random variables, as illustrated the local.lut program :

```
node local() returns(x:real = 0.0) =
  exist target : real in
  loop {
    0.0 < target and target < 42.0 and x = pre x
    fby
    loop [20] { x = (pre x + target) / 2.0 and
               target = pre target }
  }
```

```
<prompt> lutin local.lut -l 100 -rif local.rif; gnuplot-rif
local.rif
```

- Question : modify the previous program so that x reaches the target after a damped oscillation

Answer

Distribute a constraint into a scope : assert

Consider for instance the true_since_n_instants2.lut program :

```
node ok_since_n_instants(b:bool;n:int)returns(res:bool)=
  exist cpt: int in
  cpt = n and res = (b and (cpt <= 0))
  fby
    loop {
      cpt = (if b then (pre cpt-1) else n) and
      res = (b and (cpt <= 0))
    }
```

- One flaw is that $res = (b \text{ and } (cpt \leq 0))$ is duplicated.
- $\text{assert } \langle ce \rangle \text{ in } \langle te \rangle \equiv \langle te' \rangle$,
where $\langle te' \rangle = \langle te \rangle [c/c \text{ and } ce]_{\forall c \in \mathcal{C} \text{ onstraints}(te)}$
- Question : Rewrite the true_since_n_instants2.lut using the assert/in construct and avoid code duplication.

Answer

External code

- Lutin program can call any function defined in a shared library

```
extern sin(x: real) : real
let abs(x:real) : real = if x < 0.0 then -x else x
node bizzare() returns (res: real) =
  exist x,eps: real in
  res = 0.0 and eps = 0.0 fby
  loop      abs(eps - pre eps) < .02
            and x = pre x + 0.5 + pre eps
            and res = sin(pre x)
```

```
<prompt> lutin -L libm.so -l 200 ext-call.lut -rif
f.rif;gnuplot-rif f.rif
```

Combinators

- A *combinator* is a well-typed macro that eases code reuse. One can define a combinator with the `let/in` statement, or just `let` for top-level combinators.
- A simple combinator
The `combinator.lut` program illustrates the use of combinators :

```
let within(x, min, max: int): bool =
  (min <= x) and (x <= max)

node random_walk() returns (y:int) =
  within(y,0,100) fby loop within(y,pre y-1,pre y+1)
```

```
<prompt> lutin -l 100 combinator.lut -rif walk.rif ;
gnuplot-rif walk.rif
```

Combinators (cont)

- A combinator that needs memory

```

let within(x, min, max: real): bool =
  (min <= x) and (x <= max)
let up  (delta:real; x : real ref) : bool =
  within(x, pre x, pre x + delta)
let down(delta:real; x : real ref) : bool =
  within(x, pre x - delta, pre x)
node up_and_down(min,max,d:real) returns (x:real) =
  within(x, min, max) fby
  loop {
    | loop { up  (d, x) and pre x < max }
    | loop { down(d, x) and pre x > min }
  }

```

<prompt> lutin -luciole up-and-down.lut

Question : what happens if you guard the up combinator by $x < \max$ instead of $\text{pre } x < \max$?

Combinators (cont)

- Trace Combinators

```
let myloop(t:trace) : trace = loop try loop t
```

Here we restart the loop from the beginning whenever we are blocked somewhere inside `t`. (`myloop.lut`)

```
let myloop(t:trace) : trace = loop try loop t
node use_myloop(reset:bool) returns(x:int) =
  myloop(
    x = 0 fby
    assert not reset in
    x = 1 fby
    x = 2 fby
    x = 3 fby
    x = 4
  )
```

```
<prompt> lutin -luciole myloop.lut
```

Exceptions

- Global exceptions can be declared outside the main node :

```
exception ident
```

- or locally within a trace statement :

```
exception ident in st
```

- An existing exception ident can be raised with the statement :

```
raise ident
```

- An exception can be caught with the statement :

```
catch ident in st1 do st2
```

If the exception is raised in st1, the control immediately passes to st2. If the “do” part is omitted, the statement terminates normally.

Exceptions (cont)

- The predefined `Deadlock` exception can only be caught

```
catch Deadlock in st1 do st2
```

≡

```
try st1 do st2
```

- If a deadlock is raised during the execution of `st1`, the control passes immediately to `st2`. If `st1` terminates normally, the whole statement terminates and the control passes to the sequel.

Exceptions (cont)

```
node toto(i:int) returns (x:int)=
  loop {
    exception Stop in
    catch Stop in
      loop [1,10] x = i fby raise Stop fby x = 43
    do x=42
  }
```

<prompt> lutin except.lut -luciole

Note that the 43 value is never generated (if $i < 43$ of course).

About exceptions

- Very (too ?) powerful mechanism
- Can be used to build complex trace operators
- But should they used to program ?

Parallelism : &>

```
node n() returns(x,y:int) = {
  loop { -10 < x and x < 10 }
  &> y = 0 fby loop { y = pre x } }
```

<prompt> lutin -luciole paralel.lut

nota bene : this construct can be expensive because of :

- **the control structure** : such a product is equivalent to an automata product, which, in the worst case, can be quadratic ;
- **the data** : the polyhedron resolution is exponential in the dimension of the polyhedron.

Use the `run/in` construct instead if performance is a problem.

Cheap parallelism : Calling Lutin nodes `run/in`

The idea is the following : when one writes :

```
run (x,y) := foo(a,b) in
```

in order to be accepted the following rules must hold :

- `a` and `b` be uncontrollable variables (e.g., inputs or memories)
- `x` and `y` should be controllable variables (e.g., outputs or locals)
- in the scope of such a `run/in`, `x` and `y` becomes uncontrollable.

nb : it is exactly the parallelism of Lustre, with an heavier syntax. In Lustre, one would simply write

```
(x,y)=foo(a,b);
```

Moreover in Lutin, the order of equations matters.

Cheap parallelism : Calling Lutin nodes `run/in`

- The `run/in` construct is another (cheaper) way of executing code in parallel
- The only way of calling Lutin nodes.
- Less powerful : constraints are not merged, but solved in sequence

```
include "N.lut"  
include "incr.lut"  
node use_run() returns(x:int) =  
  exist a,b : int in  
  run a := N() in  
  run b := incr(a) in  
  run x := incr(b) in  
  loop true
```

```
<prompt> lutin -l 5 -q run.lut -m use_run
```

Why does the `run/in` statement makes Lutin usable ?

Using combinators and `&>`, it was already possible to reuse code, but `run/in` is

- Much more efficient : polyhedra dimension is smaller
- Mode-free (args can be in or out) combinators are error-prone

Wearing sensors

The `sensors.lut` program that makes extensive use of the `run` statements.

```
<prompt> lutin sensors.lut -m main -luciole
```

Waiting for the stability of a signal

- Defining and checking the stability of a variable (in particular in presence of noise) is not that easy.
 - One definition could be that a variable is stable if it remains within an interval during a certain amount of time. More precisely :

A variable V is (s,n) -stable if there exists an interval I of size s , and an instant i such that, for all x in $[i,i+n]$, $V(x)$ is in I .

$$\exists i, \exists I, \text{st } |I| = s, \forall x \in [i, i+n] x(n) \in I$$

The Lutin version

```
<prompt> lutin -luciole is_stable.lut -m is_stable
```

The Crazy rabbit

- The rabbit serves as an environment for a caml program that displays its position in a graphical windows
- The rabbit remains in its field, and avoids a moving obstacle
- The rabbit changes its speed and trajectory from time to time

```
<prompt> luretteop -rp "sut:ocaml:rabbit.cmxs:" -rp  
'env:lutin:rabbit.lut:-main:rabbit:-L:libm.so:-loc'
```

- crazy-rabbit/ud.lut
- crazy-rabbit/moving-obstacle.lut
- crazy-rabbit/rabbit.lut
- crazy-rabbit/rabbit.ml