A Lutin Tutorial

Erwan Jahier

Verimag
Plan

1. Forewords
2. Execute Lutin programs
3. The Lutin Language
4. The `run` operator
5. Advanced examples
Motivations : testing reactive programs

T < 100
valid -> T<100
valid and nominal => T<100

Wrong or imprecise spec
bug
design coding
SUT
Specifications
Oracles
Coverage
Environnements
extraction
(formalisation/translation)

Lurette
100% cov?
ok
ko

problem

Erwan Jahier
A Lutin Tutorial
Verimag
Lutin in one slide

- Lustre-like: Dataflow, parallelism, modular, logic time, \( \text{pre} \).
- 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 first need to install opam. For instance, on debian-like boxes do

```
sudo apt-get install opam
opam init ; opam switch 4.04.0 ; eval 'opam config env'
```

and then do:

```
sudo apt-get install gnuplot tcl
opam repo add verimag-sync-repo
        "http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/opam-repository"
opam update
opam install lutin
```

and also the Lustre V4 distribution (for luciole and sim2chro)

<prompt> echo "go!"
Plan

1. Forewords
2. Execute Lutin programs
3. The Lutin Language
4. The `run` operator
5. Advanced examples
Stimulate Lutin programs

• A program that increments its input
  Let’s consider the following Lutin program named incr.lut.

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

```prompt> lutin incr.lut```

• A program with no input

```lutin
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*.

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

<prompt> luciole-rif lutin incr.lut
```
Store and Display the produced data: **sim2chro** and **gnuplot-rif**

- Generate a RIF file
  It is possible to store the lutin RIF output into a file using the `-o` option.
  
  `<prompt> lutin -l 10 -o 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> lurette -sut "lutin decr.lut -n incr" -env "lutin decr.lut -n decr" <prompt> cat rdbg.rif | sim2chrogtk -ecran > /dev/null
```

- I’ve bought 2 electronic chess games
- connected one to another
- And now I’m at peace
Plan

1. Forewords
2. Execute Lutin programs
3. The Lutin Language
4. The run operator
5. Advanced examples
• 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** \((\text{loop} \ <\text{te}>\)) that allows one to combine constraint expressions.
Non deterministic programs

• A First non-deterministic program

```plaintext
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:

```plaintext
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
Non deterministic programs (cont)

- A 3D non-deterministic example

```plaintext
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; splot "poly.data" using 1:2:3;pause mouse close'| gnuplot
```

- One can observe the effect of `-step-edges` and `-step-vertices` options on the repartition of generated points
Constraint may also depend on inputs.

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

<prompt> luciole-rif lutin range-bis.lut
```
Controlled non-determinism: the choice operator

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

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

```plaintext
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.

```bash
$ lutin -l 10 -q choice.lut
$ lutin -l 10000 -q choice2.lut | grep 42 | wc -l
```
Controlled non-determinism: the choice operator

```
node choice(b:bool) returns(x:int) =
    x=0 fby
    x=-15 fby
    loop {
        |1: x = 1 and b
        |9: x = 2 and b
        | x = 3 and not b
    }
```

<prompt> luciole-rif lutin choice3.lut
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

```lutin
let n = 3

node foo() returns (i:int) =
    loop [3] 0<= i and i < n fby
    let s=10 in
    loop [3] s<= i and i < s+n
```

<prompt> lutin -quiet letdef.lut

Erwan Jahier
A Lutin Tutorial
Verimag
A parametric combinator

The `combinator.lut` program illustrates the use of parametric combinators:

```plaintext
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 -o walk.rif ; gnuplot-rif walk.rif
A combinator that needs memory (ref)

```luciole-rif
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 }
}
```

Question: what happens if you guard the up combinator by x<max instead of pre x < max?
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:

```lut
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> luciole-rif lutin true-since-n-instants.lut
Local variables again

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

```plaintext
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 }
  }
```

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:

```plaintext
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 and (cpt<=0))` is duplicated.

- `assert <ce> in <te> ≡ <te'>,`
  where `<te'>= <te>[c/c and ce]∀c∈Constraints(te)`

**Question**: Rewrite the `true-since-n-instants2.lut` using the `assert/in` construct and avoid code duplication.

**Answer**: 

Erwan Jahier

A Lutin Tutorial

Verimag
Lutin program can call any function defined in a shared library (.so)

```lisp
extern sin(x: real) : real
let between(x, min, max : real) : bool = ((min < x) and (x < max))
node bizzare() returns (x,res: real) =
  exist noise: real in
  assert between(noise,-0.1, 0.1) in
  res = 0.0 and x = 0.0 fby
  loop x = pre x + 0.1 + noise
    and res = sin(pre x)
```

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

- Global exceptions can be declared outside the main node:
  \[
  \text{exception ident}
  \]

- or locally within a trace statement:
  \[
  \text{exception ident in st}
  \]

- An existing exception ident can be raised with the statement:
  \[
  \text{raise ident}
  \]

- An exception can be caught with the statement:
  \[
  \text{catch ident in st1 do st2}
  \]

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

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

≡

```lutin
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.
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> luciole-rif lutin except.lut
Note that the 43 value is generated iff i=43.
About exceptions

- Very powerful mechanism
- Can be used to build complex trace operators
- But should they used to program?
Parallelism: \&>

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

<prompt> luciole-rif lutin 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.
Combinators (again)

• Trace Combinators

```ml
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)

```ml
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> luciole-rif lutin myloop.lut
Plan

1. Forewords
2. Execute Lutin programs
3. The Lutin Language
4. The run operator
5. Advanced examples
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 \texttt{run/in}

- The \texttt{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.

```plaintext
#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 is important?

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
Plan

1. Forewords
2. Execute Lutin programs
3. The Lutin Language
4. The `run` operator
5. Advanced examples
Wearing sensors

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

<prompt> luciole-rif lutin sensors.lut -m main
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:

\[
\exists \; I, \; st \; |I| = \varepsilon, \; \forall \; n \in [i-d, i], \; V(n) \in I
\]

A variable \( V \) is \((d,\varepsilon)\)-stable at instant \( i \) if there exists an interval \( I \) of size \( \varepsilon \), such that, for all \( n \) in \([i-d,i]\), \( V(n) \) is in \( I \).

The Lutin version

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