

LUTIN Reference manual
Version Trilby-1.54([sha:cd27afd](#))

Pascal Raymond
Erwan Jahier

May 8, 2012

Contents

1	An overview of the language	4
1.1	Symbolic state/transition systems	4
1.2	Synchronous relations	4
1.3	Weights	5
1.4	Static weights versus dynamic weights	5
1.5	Global concurrency	5
2	The LUTIN language	6
2.1	Data types	6
2.2	Nodes	6
2.2.1	Support variables	6
2.2.2	Local variables	6
2.2.3	Memory variables	6
2.3	Trace Statements	7
2.3.1	Atomic Trace Statements (Constraints)	7
2.3.2	Sequence	7
2.3.3	Choice	7
2.3.4	Loops	8
2.4	Exceptions	8
2.4.1	Defining and Raising Exceptions	8
2.4.2	Catching exceptions	9
2.4.3	The predefined Deadlock exception	9
2.4.4	Non determinism and deadlocks	9
2.5	Parallel composition	9
2.5.1	Parallelism and exceptions	10
2.6	Some sugared shortcuts	10
2.6.1	Propagating a constraint into a scope	10
2.6.2	Random loops	11
2.6.3	Define and catch an exception	11
2.6.4	Catching deadlocks	11
2.7	Combinators	11
2.7.1	Reference declarations	12
2.8	Calling external code	12
3	Syntax	14
3.1	Lexical conventions	14
3.2	Syntax notation (EBNF)	14
3.3	Syntax rules	14
3.4	Priorities	16
4	Semantics	17
4.1	Abstract syntax	17
4.2	The run function	17
4.3	The recursive run function	18
4.3.1	Basic traces.	18

4.3.2	Sequence.	18
4.3.3	Priority choice.	18
4.3.4	Empty filter.	18
4.3.5	Priority loop.	19
4.3.6	Catch.	19
4.3.7	Parallel composition.	19
4.3.8	Weighted choice.	19
4.3.9	Random loop.	19
4.4	The execution environment	20
4.4.1	Random sort of weighted choices	20
4.5	Predefined loop profiles	20
5	Executing LUTIN programs	21
5.1	The toplevel interpreter	21
5.2	The C and the OCAML API	22
5.3	Tools that can be used in conjunction with LUTIN	22
5.3.1	LUSTRE	22
5.3.2	LUCIOLE	22
5.3.3	LURETTE	22
5.3.4	CHECK-RIF	23
5.3.5	SIM2CHRO	23
5.3.6	GNU PLOT-RIF	23
6	Known bugs and issues	25
6.1	Numeric solver issues	25
6.1.1	Solving integer constraints in dimension $n \geq 2$	25
6.1.2	Fairness versus efficiency	25
6.1.3	Fair mode and precision and the computations	26
6.2	Last breath	26
7	Examples	27
7.1	Up and down	27
7.2	The crazy rabbit	27
7.3	Calling external code	30

Abstract.

A reactive system indefinitely responds to its environment. We are particularly interested here in control and embedded applications, where the environment is often the physical world. During the development of such systems, non-determinism is often useful, for describing a partially designed system and/or its environment.

LUTIN is a language designed to describe and simulate such non deterministic reactive systems. Executing a LUTIN program consists in randomly generating a particular behaviour consistent with its definition. In order to guide the generation, the language provides some constructs for controlling the random choices.

1 An overview of the language

Synchronous programs [?, ?, ?] deterministically produce outputs from input values. To be able to compile, synchronous programs need to be fully deterministic. However, sometimes, we want to be able to describe synchronous systems in a non deterministic manner.

- If one wants to describe (and simulate) an intrinsically non-deterministic system. A typical example is when one want to describe the environment of a reactive program; it can be very useful for testing and simulation purposes.
- Another potential use of the animation of non-deterministic code is when one wants to simulate partially written reactive programs (some components are missing). The idea is then to take advantage of program signatures, pre/post conditions, or code chunks to simulate those programs the more realistically as possible, taking into account the available constraints, and drawing the non-deterministic parts. This can be very useful to simulate and test applications at every stage of the development process.

We call an *non-deterministic program* such pieces of code that produce their outputs non-deterministically. LUTIN is a language to describe such non-deterministic programs. LUTIN program describes a set of data-flow constraints over Booleans and numeric values, that are combined with an explicit control-structure based on regular expressions. LUTIN can be seen as a language to program stochastic processes (Markov chains).

1.1 Symbolic state/transition systems

The basic qualitative model consists in a very general state/transition system, characterised by:

- a memory: a finite set of variables with no special restrictions on their domains (to simplify, we will consider here just boolean, integer and floating values);
- an interface: variables are declared as inputs, outputs, or locals;
- a finite control structure based on regular expressions, whose atoms represent reactions of the machine.

A global state of the system is then a pair made of the current control point (the *control-state*), and a current valuation of its memory (the *data-state*).

1.2 Synchronous relations

We adopt the synchronous approach for the reactions: all values in the memory are changing simultaneously when a reaction is performed. The previous value of the memory corresponds to the source data-state, and the current value to the next data-state. The program statements denote what are the possible values of the current memory depending on the current data-state. This information is quite general: it is a *relation* between the past and current values of the variables. In particular, no syntactic distinction is made between uncontrollable (inputs and past values) and controllable (locals and outputs) variables. Performing a reaction will consist in finding solutions to such a formula. This problem induces a restriction: we suppose that, once reduced according to the past and input values, the constraints are solvable by some actual procedure¹.

¹concretely, we have developed a constraint solver for mixed boolean/linear constraints.

1.3 Weights

Since we have to deal with uncontrollable variables, defining a sound notion of distribution must be done carefully: depending on its variables, a formula may be infeasible, and thus its actual probability is zero. In other terms, if we want to use probabilistic distributions, we would have to define a reaction as a map from the tuple $\langle \text{source state, past values, input values} \rangle$ to a distribution over the pairs $\langle \text{controllable values, next state} \rangle$. Expressing and exploiting this kind of model would be too complex. We prefer a pragmatic approach where probabilities are introduced in a more symbolic way.

The main idea is to keep the distinction between the probabilistic information and the constraint information. Since constraints are influencing probabilities (zero or non-zero), this information does not express the probability to be drawn, but the probability to be *tried*. Therefore, we do not use distributions (i.e., set of positive values the sum of which is 1) but *weights*. A weight is a positive integer: if two possible reactions (i.e., the corresponding constraints are both satisfiable) are labelled respectively with the weights w and w' , then the probability to perform the former is w/w' times the probability to perform the latter.

1.4 Static weights versus dynamic weights

The simplest solution is to define weights as constants, but in this case, the expressive power can be too weak. With such static weights, the uncontrollable variables qualitatively influence the probabilities (zero or not, depending on the constraints) but not quantitatively: the idea is then to define *dynamic weights* as numerical functions of the inputs and the past-values. Taking numerical past-values into account can be particularly useful. A good example is when simulating an *alive process* where the system has a known average life expectancy before breaking down; at each reaction, the probability to work properly depends *numerically* on an internal counter of the process age.

1.5 Global concurrency

Concurrency (i.e., parallel execution) is a central issue for reactive systems. The problem of merging sequential and parallel constructs has been largely studied: classical solutions are hierarchical automata “à la StateCharts” [?, ?], or statement-based languages like Esterel [?]. Our opinion is that deeply merging sequence and parallelism is a problem of high-level language, and that it is sufficient to have a notion of global parallelism: intuitively, local parallelism can always be made global by adding extra idle states. As a consequence, concurrency is a top level notion in our model: a complete system is a set of concurrent program, each one producing its own constraints on the resulting global behaviour.

2 The LUTIN language

2.1 Data types

There exists 3 pre-defined data types: **bool**, **int**, and **real**.

Structured data-types (arrays, enums, structures) and user-defined abstract types are not yet implemented (coming soon hopefully).

2.2 Nodes

Nodes are entry points for LUTIN programs. Nodes are made of an interface declaration and a body. LUTIN nodes can be reused in other LUTIN nodes (as LUSTRE nodes); they can also be top-level programs.

2.2.1 Support variables

The **node**² interface declares the set of input and output variables; they are called the *support variables*. During the node execution, actual input values are provided by the program environment. Output variable ranges can specified (or not) in the declaration. By default, numeric value ranges from -10000 to 10000.

▷ **Example:**

```
node foo(x:bool) returns (t:real [0.0;100.0]; i:int) =
  NodeBodyStatement
```

The node body is made of statements that we describe below.

2.2.2 Local variables

Node body statements can be made of a local variable declarations. Such variables are declared with the **exist** keyword.

▷ **Example:**

```
exist y : real in st
exist z : int [-100000; 100000] in st
```

In their scope, local variables are similar to outputs; we call them the *controllable variables*.

2.2.3 Memory variables

Any expression may refer to the previous value of a variable using the **pre** keyword. The value of **pre x** is inherited from the past and cannot be modified. Memories are therefore similar to inputs; we sometimes call them the *uncontrollable variables*.

A memory variable **pre x** doesn't need to be declared, as long as the variable **x** is declared.

▷ **Example:**

```
if x then (t > pre t) else (t <= pre t )
```

describes any valuation of the support where **t** is higher than its value at the previous instant when **x** is true, and lower otherwise.

²used to be called **system** in the earlier versions of LUTIN, to highlight the difference with LUSTRE nodes.

Note that **pre** can only operate over variables. For example, **pre (t+10.0)** is forbidden.

2.3 Trace Statements

A node body can be made of statements that describes how support and local variables evolve at each instant. Such statements are called *trace expressions* (or a *trace statements*). Trace expressions are either atomic, or composed of other trace expressions using operators inspired by regular expressions, and described below.

2.3.1 Atomic Trace Statements (Constraints)

An atomic trace is simply a relation (or a constraint) between the program variables. A constraint is a trace of length 1.

▷ **Example:**

x and (0.0 < t) and (t <= 10.0)

The constraint above describes any valuation of the support variables where **x** is true and **t** is between **0.0** and **10.0**.

Note that, during the execution, if **x** is an input of the current node, and if **x** is false at the current instant, then the constraint is unsatisfiable.

Atomic statements can be combined to describe longer traces using the trace operators described below.

An atomic statement is said to be *startable* if it is made of a satisfiable constraint. When an atomic statement is not satisfiable, we say that it *deadlocks*.

2.3.2 Sequence

If *st1* and *st2* are 2 trace expressions, *st1 fby st2* is a trace expression that

- behaves as *st1*, and when it terminates, behaves as *st2*;
- deadlocks as soon as *t1* or *t2* deadlocks.

The sequence *st1 fby st2* is *startable* if and only if *st1* is startable.

2.3.3 Choice

If *st1*, ..., *stn* are n trace expressions, $\{|st1 | \dots | stn \}$ (the first | is optional) behaves as follows: randomly choose one of the *startable* statements from *st1*, ..., *stn*. If none of them are startable, the whole statement deadlocks. $\{|st1 | \dots | stn \}$ is startable if and only if one of the *sti* is startable.

Weighted choice. In a choice, the random selection of a particular startable statement is uniform. For instance, if *k* of *n* statements are startable, each of them is chosen with a probability of $1/k$.

This is the reason why the choice is not a binary, associative statement:

$\{|st1 | \{|st2 | st3 \} \}$

is not *stochastically* equivalent to

$\{\{|st1 | st2 \} | st3 \}$

In order to influence the probabilities, the user may assign *weights* to the branches of a choice:

```
{ |w1: st1 ... |wn: stn }
```

Weights (*w_i*) may be any *integer expression* made of constants and uncontrollable variables. In other terms, only the environment and the past may influence the probabilities. If not specified, the weight is equal to 1, and the first bar is optional (e.g., '{st1 |st2 }' is equivalent to '{|1: st1 |1: st2 }'). Weights do not define the probability to be chosen among the choices, but the probability to be chosen among the *possible* choices, (i.e., among startable statements).

Priority Choice. { |>st1 |>... |>stn } behaves as *st1* if *st1* is startable, otherwise behaves as *st2* if *st2* is startable, etc. If none of them are startable, the whole statement deadlocks. The first |> is optionnal.

2.3.4 Loops

loop st terminates *normally* if *st* deadlocks; otherwise, it behaves as *st fby loop st*. This can be read as “repeat the behavior of *st* as long as possible”.

Nested loops. The execution of nested loops may results on infinite, instantaneous loops.

▷ **Example:** If *c* is a non satisfiable constraint, the statement

```
loop loop c
```

keeps the control but do nothing.

We consider programs that generates such instantaneous loops as *incorrect* (this is quite similar to infinite recursion in classical languages).

Statically checking if a program is free of instantaneous loops is undecidable. One solution is to adopt a statical criterium rejecting all incorrect programs, but also some correct ones.

Typically, a program is certainly free of instantaneous loop if each control branch whitin a loop contains a statement that “takes time” (i.e., a constraint).

▷ **Example:** The (potentially) incorrect program:

```
loop loop c
```

can be safely replaced by:

```
loop { c fby loop c }
```

The opposite solution is to accept a priori any programs and generate a runtime error if an instantaneous loops arises during the execution. This is the solution adopted in the operational semantics (Section 4).

2.4 Exceptions

2.4.1 Defining and Raising 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`

2.4.2 Catching exceptions

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.

2.4.3 The predefined `Deadlock` exception

When a trace expression deadlocks, the `Deadlock` exception is raised. In fact, this exception is internal and cannot be redefined nor raised by the user. The only possible use of the `Deadlock` in programs is one try to catch it:

▷ **Example:**

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

Cf Section 2.6.4.

2.4.4 Non determinism and deadlocks

The general rule is that, if a statement can start, then it must start; this is the *reactivity principle*.

2.5 Parallel composition

In order to put in parallel several statements, one can write:

```
{&>st1 &>... &>stn }
```

where the first `&>` can be omitted.

This statement executes in parallel all the statements *st1* ... *stn*. All along the parallel execution each branch produces its own constraint; the conjunction of these local constraints gives the global constraint.

If one branch terminates normally, the other branches continue. The whole statement terminates when the last branches terminates.

If (at least) one branch raises an exception, the whole statement raises the exception.

Parallelism vs stockastic directives It is impossible to define a parallel composition which is fair according to the stockastic directive.

▷ **Example:** Consider the statement:

```
{ { |1000: X | Y } &> { |1000: A | B } }
```

where *X*, *A*, *X*∧*B*, *A*∧*Y* are all satisfiable, but not *X*∧*A*:

- the priority can be given to *X*∧*B*, which does not respect the stockastic directive of the second branch,
- or to *A*∧*Y*, which does not respect the stockastic directive of the first branch.

In order to solve the problem, the stockastic directives are not treated in parallel, but in *sequence*, from left to right:

- the first branch makes its choice, according its local stockastic directives,
- the second one branch makes its choice, according to what has been chosen by the first one etc.

In the example, the priority is then given to $X \wedge B$.

Finally, the treatment is:

- parallel w.r.t. constraints (it's a conjunction),
- but sequential w.r.t. weights directives (left to right).

Note that the concrete syntax ($\&>$) reflects the fact the operation is not commutative.

2.5.1 Parallelism and exceptions

There is no notion of “muti-raising”, even when several statements are executed in parallel. In a parallel composition, exception raising are, like stockastic directives, treated in sequence from left to right.

2.6 Some sugared shortcuts

In this section, we present a set of operators that do not add any expressing power to the language, but that ought to make the LUTIN programmer's life more harmonious.

2.6.1 Propagating a constraint into a scope

Very often, one wants to define some constraints that should hold in all (or most) of the program statements. One way to do this is to create a dummy Boolean variable that carry the constraint, and to put it in parallel with the statement.

```

▷ Example:
  exist b : bool in
  let b = exp in
    b  $\&>$  st

```

Because this is very useful, we defined a dedicated construct (**assert**) that has exactly the same semantics:

```

▷ Example:
  assert exp in st

```

In other words, the constraint *exp* (a Boolean expression) is distributed (propagated) in all the constraints of the statement *st*.

2.6.2 Random loops

Random loops are defined by constraining the number of iterations. There are actually two pre-defined kinds of random loops:

- Interval: `loop [min, max]`
the number of iteration should be comprized between the integer constants *min* and *max* (which must satisfy $0 \leq min \leq max$).
- Average: `loop~ av: sd`
the average number of iteration should be *av*, with a standard deviation *sd*. The behavior is defined if and only if $4 * sd < av$.

Note. Random loops are following the *reactivity principle*, which means that the actual number of loops may significantly differ from the “expected” one since looping may sometimes be required or impossible, according to the satisfiability of constraints. The precise semantics is given in §4.

2.6.3 Define and catch an exception

The following statement:

`trap x in st1 do st2`

is a shortcut for: `exception x in catch x in st1 do st2`

2.6.4 Catching deadlocks

`catch Deadlock in st1 do st2`

can be written:

`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.

2.7 Combinators

Combinator were introduced in the language to allow code reuse. It’s a kind of well-typed macro. One can define a combinator with the `let` statement:

`let id (Params) : Type = St1 in St2`

- Such a definition may appear at top-level, outside a node, in which case the “`in St2`” is absent.
- Classical scoping rules apply for *St1*: free variables are first binded to the *Params* declaration, otherwise they are binded to the scope in which the whole statement appears.
- The “(*Params*)” part is optional; with no parameters, the declaration simply means that *id* is an alias for the expression *St1* within *St2*.

- The ": *Type*" is optional; when absent, the type is deduced from the expression *StI*.
- The type is either a data-type (`bool`, `int`, `real`) or the type `trace`, meaning that the expression "*StI*" (and thus the identifier "*id*") denotes a behaviour. `trace` is an abstract type. It does not say anything about the support variables of the denoted behaviour.

Here is an example of (global) Boolean combinator over data expressions:

▷ **Example:**

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

Here is an example of trace combinator. It takes two traces and returns a trace that:

- runs the two trace arguments in parallel,
- terminates when the second one terminates.

▷ **Example:**

```
let as_long_as(X, Y : trace) : trace =
  trap Stop in
    X &> {Y fby raise Stop}
  }
```

2.7.1 Reference declarations

If one wants to access to the previous value of a variable, one has to declare in the combinator profile that it is a reference using the `ref` keyword.

▷ **Example:**

```
let foo(pt: real ref, t: real) : bool =
  if pre pt < pt then pt < t else t < pt
```

Another example of the use of reference variables is given in Section 7.1.

2.8 Calling external code

In order to use external code from LUTIN, we provide a mechanism based on dynamic (shared) libraries (a.k.a. `.so` or `.dll` files). Such dynamic libraries should be built and used according to certain conventions that we describe in this section.

Moreover, the type of imported functions should be declared in the LUTIN file, and of course, the declared types should match their definitions in the library. For example, in order to be able to call the `sin` and the `cos` extern functions in a lutin file, one have to declare them like that:

▷ **Example:** A LUTIN program calling 2 extern functions `sin()` and `cos()`.

```
extern sin(x: real) : real
extern cos(x: real) : real

node cartesian(r, alpha: real) returns (x, y: real) =
  loop {
    x = r * cos (alpha) and
    y = r * sin (alpha)
  }
```

Another example, as well as the extern library compilation process and the LUTIN interpreter options, is provided in Section [7.3](#)

BEWARE: if the types you declare in the Lutin file does not match their definitions, it might run silently returning wrong values!

3 Syntax

3.1 Lexical conventions

- One-line comments start with `--` and stop at the the end of the line.
- Multi-line comments start with `(*` and end at the next following `*)`. Multi-line comments cannot be nested.
- *Ident* stands for identifier, following the C standard (`[_a-zA-Z] [_a-zA-Z0-9]*`),
- *Floating* and *Integer* stands for decimal floating point and integer notations, following C standard,

3.2 Syntax notation (EBNF)

- Keywords are displayed like that: `keyword`.
- Grammatical symbols like that: *GramaticalSymbol*.
- Optional parts like that: `[something]`.
- List (0 or more) parts like that: `{ something }`.
- Grouped parts like that: `(something)`.

3.3 Syntax rules

Those syntax rules are automatically extracted from the yacc.

LUTIN files. A Lutin file (`.lut`) is a list of declarations. Top-level declarations can be combinator, exception, or node declarations.

```

File           ::= { OneDecl }
Include        ::= include <string>
OneDecl        ::= [ Include | LetDecl | ExceptDecl | ExternNodeDecl | NodeDecl ]
ExceptDecl     ::= exception IdentList
LetDecl        ::= let Ident [ ( [ TypedParamList ] ) ] [ : Type ] = TraceExp
ExternNodeDecl ::= extern Ident [ ( [ TypedParamList ] ) ] [ : Type ]
NodeStart      ::= node | system
NodeDecl       ::= NodeStart Ident ( TypedIdentListOpt ) returns ( TypedIdentList )
                = TraceExp

```

Variable and combinator Parameter Declaration. Both are declared with their type. The `ref` type flag may only appear in combinator parameter declaration. A default value (`=Exp`) may only appear in variable declaration. Range annotations are only meaningful for numeric variables.

```

IdentList      ::= Ident { , Ident }
IdentTuple     ::= IdentList
                | ( IdentList )

```

```

ERunVars ::= ( ERunVarList )
           | ERunVarList
ERunVarList ::= ERunVar { , ERunVar }
TypedIdent ::= IdentList : Type [ [ Exp ; Exp ] ] [ = Exp ]
TypedIdentListOpt ::= [ TypedIdentList ]
TypedIdentList ::= TypedIdent { ; TypedIdent } [ ; ]
TypedParamList ::= TypedParam { ; TypedParam }
TypedParam ::= IdentList : ParamType
ERunVar ::= Ident [ : Type ] [ = Exp ]

```

```

Type ::= PredefType | trace
PredefType ::= bool | int | real
ParamType ::= Type | PredefType ref

```

Trace expressions. A Trace expression is a statement of type **trace**.

```

TraceExp ::= Exp
           | raise Ident
           | TraceExp fby TraceExp
           | LoopExp
           | LoopStatExp
           | BraceExp
           | LetDecl in TraceExp
           | [ strong | weak ] assert Exp in TraceExp
           | erun ERunVars := Exp in TraceExp
           | run IdentTuple := Exp in TraceExp
           | run IdentTuple := Exp
           | exist TypedIdentList in TraceExp
           | exception IdentList in TraceExp
           | try TraceExp [ do TraceExp ]
           | catch Ident in TraceExp [ do TraceExp ]
           | trap Ident in TraceExp [ do TraceExp ]
LoopExp ::= [ strong | weak ] loop TraceExp
LoopStatExp ::= loop ( Average | Gaussian ) TraceExp
Average ::= [ Exp [ , Exp ] ]
Gaussian ::= ~ Exp [ : Exp ]
Choice ::= | [ Exp : ] TraceExp { | [ Exp : ] TraceExp }
Prio ::= | > TraceExp { | > TraceExp }
Para ::= &> TraceExp { &> TraceExp }
BraceExp ::= { ( TraceExp | Prio | TraceExp Prio | Choice | TraceExp Choice | Para |
                TraceExp Para ) }

```

Trace expressions are surrounded by braces, and data expressions by parenthesis.

Data Expressions. A data expression is a statement of type **bool**, **int**, or **real**. They are almost classical algebraic expressions, except for the special "operator" **pre** which requires a

variable identifier.

```

Exp ::= Const
      | IdentRef
      | pre Ident
      | ( Exp )
      | UnExp
      | BinExp
      | if Exp then Exp else Exp
UnExp ::= ( - | not ) Exp
BinExp ::= Exp ( = | <> | or | xor | and | => | + | - | * | / | div | mod | < | <= | > | >= )
Const ::= true | false | Integer | Floating

```

Ident references, with or without arguments, appear in both trace or data expressions. Arguments can be any expressions.

```

IdentRef ::= Ident [ ( [ ArgList ] ) ]
ArgList ::= Arg { , Arg }
Arg ::= TraceExp

```

3.4 Priorities

Priorities are the following, from lower precedence to higher precedence. In the same level, the default is to group binary operators left-to-right (note that it may result in type errors).

- **else**,
- =>, logical implication, group **right-to-left**,
- **or**,
- **xor**,
- **and**,
- =, <>,
- >, <, >=, <=,
- +, - (binary),
- *, /, **div**, **mod**,
- **not**,
- - (unary).

4 Semantics

4.1 Abstract syntax

The semantics is defined according to the following abstract syntax, where:

- we only consider binary priority choice and parallel composition, since they are left-associative,
- we define the empty-behaviour (ε) and the empty-behaviour filter ($t \setminus \varepsilon$), which are not available in the concrete syntax, but useful for defining the semantics,
- random loops are *normalized* by expliciting their weight functions:
 - the stop function ω_s takes the number of iteration already performed and returns the relative weight of the “stop” choice,
 - the continue function ω_c takes the number of iteration already performed and returns the relative weight of the “continue” choice.

These functions are completely determined by the “profile” of the loop in the concrete syntax (interval or average, together with the corresponding static arguments). See §4.5 for a precise definition of these weight functions.

- the actual number of (already) performed iterations is syntactically attached to the loop; this is convenient to define the semantics in terms of rewriting. In the main statement, this flag is obviously set to 0.

empty behaviour: ε	empty filter: $t \setminus \varepsilon$
atomic constraint: c	catch: $[t \xrightarrow{x} t']$
raise: \uparrow^x	choice: $\prod_{i=1}^n t_i / w_i$
sequence: $t \cdot t'$	random loop: $t_i^{(\omega_c, \omega_s)}$
priority: $t \succ t'$	priority loop: t^*
parallel: $t \& t'$	

\mathcal{T} denotes the set of trace expressions, and \mathcal{C} the set of constraints.

4.2 The run function

The semantics of an execution step is given by a function taking an environment e and a (trace) expression t : $Run(e, t)$.

This function returns an *action* which is either:

- a transition $\xrightarrow{c} n$, which means that t produces a constraint c and rewrite itself in the (next) trace n ,
- a termination \uparrow^x , where x is a termination flag which is either ε (normal termination), δ (deadlock) or some user-defined exception.

\mathcal{A} denotes the set of actions, and \mathcal{X} denotes the set of termination flags.

The run function is inductively defined using a recursive function $\mathcal{R}_e(t, g, s)$ where the parameters g and s are continuation functions returning actions.

- $g : \mathcal{C} \times \mathcal{T} \mapsto \mathcal{A}$ is the *goto* function, defining how a local transition should be treated according to the calling context.

- $s : \mathcal{X} \mapsto \mathcal{A}$ is the *stop* function, defining how a local termination should be treated according to the calling context.

At the top level, \mathcal{R}_e is simply called with the trivial continuations:

$$\text{Run}(e, t) = \mathcal{R}_e(t, \lambda(c, v). \overset{c}{\rightarrow} v, \lambda x. \uparrow^x) \quad (1)$$

4.3 The recursive run function

4.3.1 Basic traces.

The empty behavior raises the termination flag in the current context:

$$\mathcal{R}_e(\varepsilon, g, s) = s(\varepsilon)$$

A raise statement terminates with the corresponding flag:

$$\mathcal{R}_e(\uparrow^x, g, s) = s(x)$$

A constraint generates a goto or raises a deadlock, depending on its satisfiability in the environment:

$$\mathcal{R}_e(c, g, s) = (e \models c)? g(c, \varepsilon) : s(\delta)$$

4.3.2 Sequence.

$$\mathcal{R}_e(t \cdot t', g, s) = \mathcal{R}_e(t, g', s')$$

where:

$$\begin{aligned} g'(c, n) &= g(c, n \cdot t') \\ s'(x) &= (x = \varepsilon)? \mathcal{R}_e(t', g, s) : s(x) \end{aligned}$$

4.3.3 Priority choice.

There is no continuation here: just a deterministic choice between the two branches. The second branch is taken if and only if the first branch deadlocks in the current context.

$$\mathcal{R}_e(t \succ t', g, s) = (r \neq \uparrow^\delta)? r : \mathcal{R}_e(t', g, s) \quad \text{where } r = \mathcal{R}_e(t, g, s)$$

4.3.4 Empty filter.

This internal construct is introduced to ease the definition of the loops. Intuitively, it forbids the core t to terminate immediately.

$$\mathcal{R}_e(t \setminus \varepsilon, g, s) = \mathcal{R}_e(t, g, s')$$

where:

$$s'(x) = (x = \varepsilon)? \uparrow^{\uparrow^\delta} : s(x)$$

4.3.5 Priority loop.

The semantics is defined according to the equivalence:

$$t^* \Leftrightarrow (t \setminus \varepsilon) \cdot t^* \succ \varepsilon$$

4.3.6 Catch.

Note that z is a catchable exception (either δ or a user-defined exception).

$$\mathcal{R}_e([t \xrightarrow{z} t'], g, s) = \mathcal{R}_e(t, g', s')$$

where:

$$\begin{aligned} g'(c, n) &= g(c, [n \xrightarrow{z} t']) \\ s'(x) &= (x = z)? \mathcal{R}_e(t', g, s) : s(x) \end{aligned}$$

4.3.7 Parallel composition.

$$\mathcal{R}_e(t \& t', g, s) = \mathcal{R}_e(t, g', s')$$

where:

$$\begin{aligned} s'(x) &= (x = \varepsilon)? \mathcal{R}_e(t', g, s) : s(x) \\ g'(c, n) &= \mathcal{R}_e(t', g'', s'') \text{ with:} \\ s''(x) &= (x = \varepsilon)? g(c, n) : s(x) \\ g''(c', n') &= g(c \wedge c', n \& n') \end{aligned}$$

4.3.8 Weighted choice.

The evaluation of the weights, and the (random) total ordering of the branches according those actual weights are both performed by the environment:

$Sort_e(t_1/w_1, \dots, t_n/w_n)$ returns:

- a priority expression $t_{\sigma(1)} \succ \dots \succ t_{\sigma(k)}$ reflecting the priorities that have been (randomly) assigned to the branches; note that k may be less than n , since some branches may have an actual weight of 0.
- the deadlock expression \uparrow^δ if all weights are evaluated to 0.

See §4.4.1 for the precise definition of $Sort_e$.

$$\mathcal{R}_e(\prod_{i=1}^n t_i/w_i, g, s) = \mathcal{R}_e(Sort_e(t_1/w_1, \dots, t_n/w_n), g, s)$$

4.3.9 Random loop.

The semantics is defined according to the equivalence:

$$t_i^{(\omega_c, \omega_s)} \Leftrightarrow (t \setminus \varepsilon) \cdot t_{i+1}^{(\omega_c, \omega_s)} / \omega_c(i) \mid \varepsilon / \omega_s(i)$$

4.4 The execution environment

4.4.1 Random sort of weighted choices

4.5 Predefined loop profiles

5 Executing LUTIN programs

5.1 The toplevel interpreter

Here is the output of `lutin --help`:

5.2 The C and the OCAML API

It is possible to call the LUTIN interpreter from C or from OCAML programs.

Calling the LUTIN interpreter from C. In order to do that from C, one can use the functions provided in the `luc4c_stubs.h` header file (that should be in the distribution). A complete example can be found in `examples/lutin/C/`. It contains, a C file, a LUTIN file that is called in the C file, and a Makefile that illustrates the different compilers and options that should be used to generate a stand-alone executable.

Calling the LUTIN interpreter from OCAML. In order call LUTIN from OCAML, one can use the functions provided in the `luc4ocaml.mli` interface file (or cf the `ocamlc` [generated html files](#)). A complete example can be found in `examples/lutin/ocaml/`.

5.3 Tools that can be used in conjunction with LUTIN

Some tools developed in the Verimag lab might be useful in you write LUTIN programs. In this section, we list the tools and describe briefly how they can be used in conjunction with LUTIN.

5.3.1 LUSTRE

Using the `lutin --2c-4lustre <string>` option and the C API described in Section 5.2, one can call the LUTIN interpreter from a lustre node. A complete example can be found in `examples/lutin/lustre/`.

5.3.2 LUCIOLE

LUCIOLE is GUI that provides buttons and slide bars to ease the execution of LUSTRE programs. Using the `lutin --2c-4luciole` option, one can use the LUTIN interpreter in conjunction with Luciole. This can be very handy when writing LUTIN programs. A complete example can be found in `examples/lutin/luciole/`.

todo : *Faire une copie d'écran illustrant une simu luciole/lutin.*

5.3.3 LURETTE

LURETTE is a tool that automates the testing of reactive programs, for example written LUSTRE. The LUTIN program interpreter is embedded into LURETTE; it is mainly used to program the environment of the System Under Test (a.k.a. SUT). Hence, LURETTE is able to test the program into a simulated environment. The SUT inputs are the LUTIN outputs, and vice versa.

Therefore, LUTIN is used to close the reactive programs by providing inputs. From a lutin-centric point of view, a LUTIN program could use LURETTE and LUSTRE to close the LUTIN program. A complete example can be found in `examples/lutin/xlurette`.

5.3.4 CHECK-RIF

A tool that performs post-mortem oracle checking using the Lustre expanded code (.ec) interpreter ECEXE.

Here is the output of `check-rif --help`:

5.3.5 SIM2CHRO

SIM2CHRO is a program written par Yann Rémond that displays data files that follows the RIF convention. For example, to display RIF file, one can launch the command : `sim2chrogtk -ecran -in data.rif`

5.3.6 GNUPLOT-RIF

GNUPLOT-RIF is another tool that displays RIF files. Sometimes it performs a better job than SIM2CHRO, sometimes not.

Here is the output of `gnuplot-rif --help`:

An example is provided in Figure 1 of Section 7.1.

6 Known bugs and issues

6.1 Numeric solver issues

Since we target the test of real-time software, we put the emphasis on the efficiency of the solver. In order to solve numeric linear constraints, we use the library of convex polyhedron POLKA [?] which is reasonably efficient, at least for small dimension of manipulated polyhedra – the algorithms complexity is exponential in the dimension of the polyhedron. Polyhedron of dimension bigger that 15 generally leads to unreasonable response time.

Note however that independent variables – namely, variables that do not appear in the same constraint – are handled in different polyhedra. This means that the limitation of 15 dimensions does not lead to a limitation of 15 variables. Fortunately, having more than 15 variables that are truly interdependent in the same cycle ought to be quite rare.

6.1.1 Solving integer constraints in dimension $n \geq 2$

When the dimension is greater than 2, for the sake of efficiency, we do not use classical methods such as linear logic for solving integer constraints: we solve those constraints in the domain of rational numbers and then we truncate. The problem is of course that the result may not be a solution of the constraints.

In such a case, we chose to pretend that the constraint is unsatisfiable (after a few more tries according to various heuristics), which can be wrong, but which is safe in some sense. The right solution there would be to call an integer solver, which is very expensive, and yet to be done.

6.1.2 Fairness versus efficiency

A LUTIN program can be interpreted in two different modes; one that emphasises the fairness of the draw; the other one that emphasises the efficiency. Indeed, suppose we want to solve the following constraint:

$$((b \wedge \alpha_1) \vee (\bar{b} \wedge \alpha_2)) \wedge \alpha_3 \wedge (\alpha_4 \vee \alpha_5)$$

where b is a Boolean, and where α_i are atomic numeric constraints of the form: $\sum_i a_i x_i < cst$. The first step is to find solution from the Boolean point of view. This leads to the four solutions:

$$b\alpha_1\bar{\alpha}_2\alpha_3\bar{\alpha}_4\alpha_5, \quad b\alpha_1\bar{\alpha}_2\alpha_3\alpha_4\bar{\alpha}_5, \quad \bar{b}\bar{\alpha}_1\alpha_2\alpha_3\bar{\alpha}_4\alpha_5, \quad \bar{b}\bar{\alpha}_1\alpha_2\alpha_3\alpha_4\bar{\alpha}_5$$

Now, suppose that:

$$\alpha_1 = 100 > x, \quad \alpha_2 = 200 > x, \quad \alpha_3 = x > 0, \quad \alpha_4 = x > x, \quad \alpha_5 = x > 1$$

where x is an integer variable that has to be generated by LUTIN. We use the convex polyhedron library to solve the numeric constraints, which lead respectively to the following sets of solutions:

$$S1 = b \wedge x \in [2; 100]; \quad S2 = b \wedge x = 0; \quad S3 = b \wedge \bar{x} \in [2; 200]; \quad S4 = b \wedge \bar{x} = 0$$

In order to perform a fair draw among the set of all solutions, we need to compute the number of solutions in each of the set S_i . But this computation is very very expensive for polyhedron of big dimension. Moreover, as we use Binary Decision Diagrams [?] to solve the Boolean part, associating a volume to each numeric part results in a lost of sharing in BDDs.

Therefore, we have adopted a pragmatic approach:

- implement an efficient mode that is fair with respect to the Boolean part only;
- implement a fair mode that performs an approximation of the polyhedron volume.

The polyhedron volume is approximated by the smallest hypercube containing the polyhedron. Note that this leads to no approximation for polyhedron of dimension 1 (intervals), and reasonable approximation in dimension 2. But the error made increases exponentially in the dimension. Therefore, for polyhedron of big dimension, it is better to use the efficient mode, and to rely only the probability defined by transition weights.

Note that when there are only Boolean variables as output or local variables, the two modes are completely equivalent.

6.1.3 Fair mode and precision and the computations

In the fair mode, we compute an approximation of polyhedron volume. But how to mix set of solutions that involves both integers and floats (which are necessarily computed by distinct polyhedra)?

The solution we have adopted is the following: relate both domain via the precision of the computations, which is a parameter of the LUTIN programs interpreter. For example, with a precision of 2 digits after the dot, we consider that the set $x \in [0; 3]$ contains 300 solutions.

6.2 Last breath

Before stopping (Vanish exception), the LUTIN interpreter generates one dummy vector of values that should be ignored.

7 Examples

7.1 Up and down

The `examples/lutin/up_and_down` directory of the LUTIN distribution contains a complete running (via the Makefile) example.

▷ **Example:** The `ud.lut` file.

```

let between(x, min, max : real) : bool = ((min < x) and (x < max))

node up(init, delta:real) returns( x : real) =
  x = init fby loop { between(x, pre x, pre x + delta) }

node down(init, delta:real) returns( x : real) =
  x = init fby loop { between(x, pre x - delta, pre x) }

node up_and_down(min, max, delta : real) returns (x : real) =
  between(x, min, max)
  fby
  loop {
    | run x := up(pre x, delta) in loop { x < max }
    | run x := down(pre x, delta) in loop { x > min }
  }

node main () returns (x : real) =
  run x:= up_and_down(0.0, 100.0, 5.0)

```

This program first defines 3 combinators: `between`, `up`, and `down`. `between` is used to constraint a variable between a min and a max. Is is used by the `up` combinator, that constraint a controllable variable to be between its previous value and its previous value plus a constant (`delta`). The parameter of `up` needs to be declared as reference, so that it possible to use its previous value (cf 2.7.1).

Then comes the definition of the main node. At the first instant, the output `x` is chosen between the minimum and the maximum. Then, either it goes up or it goes down. If it goes up (resp down), it does so until the maximum (resp minimum) value is exceeded, and then it goes down (resp up), and so on forever.

7.2 The crazy rabbit

The `examples/lutin/crazy_rabbit` directory of the LUTIN distribution contains a bigger program.

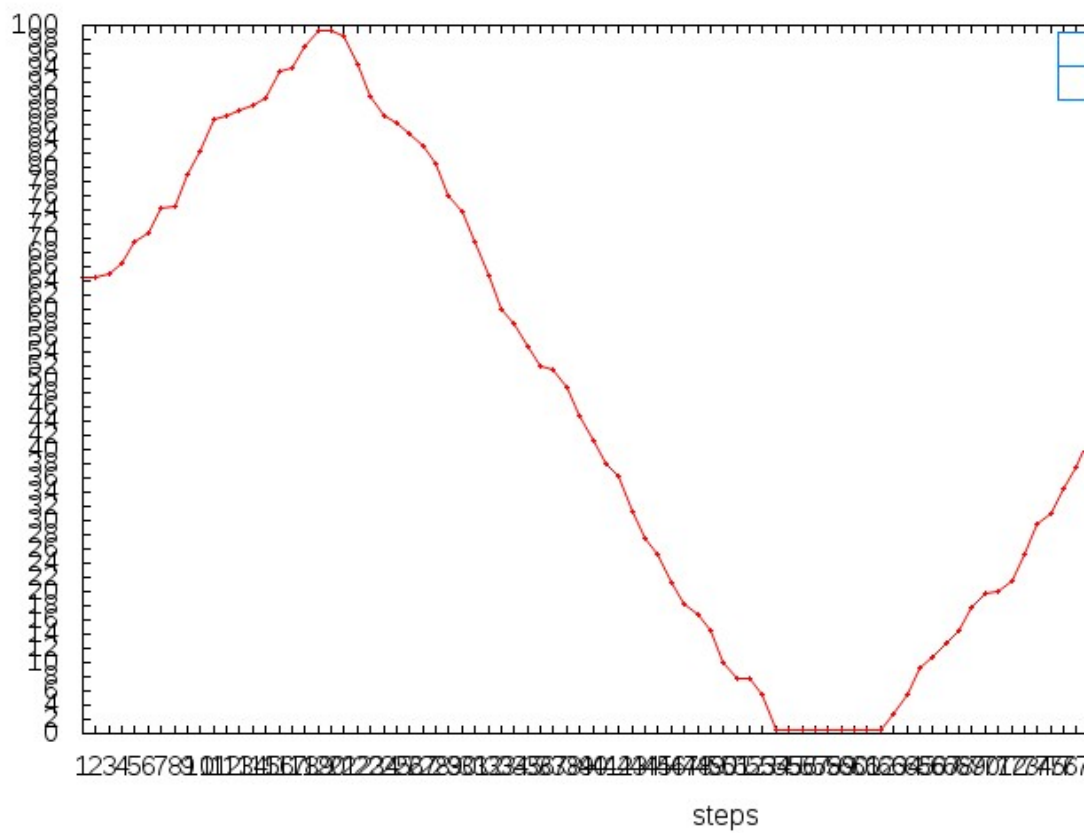


Figure 1: This image has been obtained with the command `lutin -l 100 ud.lut -main main > ud.rif ; gnuplot-rif -jpg ud.rif`

▷ **Example:** The rabbit.lut file.

```

include "ud.lut"
include "moving-obstacle.lut"

node rabbit_speed (low, high:real) returns (Speed: real) =
  exist Delta, SpeedLow, SpeedHigh: real in
  let draw_params() =
    between(Delta, 0.5, 1.0) and
    between(SpeedLow, 0.0, low) and between(SpeedHigh, 1.0, high)
  in
  let keep_params() =
    Delta = pre Delta and SpeedLow = pre SpeedLow and
    SpeedHigh = pre SpeedHigh
  in
  {
    &> loop { draw_params() fby loop ~100: 10 { keep_params() } }
    &> Speed = 1.0 fby
    run Speed := up_and_down(pre SpeedLow, pre SpeedHigh, pre Delta)
  }
extern sin(x: real) : real
extern cos(x: real) : real

-- extern printint(i:int):unit

exception Pb

node rabbit (x_min, x_max, y_min, y_max : real)
returns(x, y, p1x, p1y, p2x, p2y, p3x, p3y, p4x, p4y: real ; freeze:bool) =
  exist Speed, Alpha, Beta : real in
  let keep_position() = (x = pre x and y = pre y) in
  let draw_params() = between(Alpha, -3.14, 3.14) and between(Beta, -0.3, 0.3)
  in
  -- The beginning
  run Speed := rabbit_speed(5.0, 50.0) in
  run p1x,p1y, p2x,p2y, p3x,p3y, p4x,p4y := obstacle(x_min, x_max, y_min, y_max) in
  let line() =
    x = (pre x + Speed * cos(pre Alpha)) and
    y = (pre y + Speed * sin(pre Alpha)) and
    Alpha = pre Alpha and
    -- And he always avoids the obstacle
    not is_inside(x,y,p1x,p1y,p2x,p2y,p3x,p3y,p4x,p4y)
  in
  let escape () =
    try
      between(x, pre x - 21.0, pre x + 21.) and
      between(y, pre y - 21.0, pre y + 21.) and
      not is_inside(x,y,p1x,p1y,p2x,p2y,p3x,p3y,p4x,p4y)
    do raise Pb
  in
  let curve() =
    x = (pre x + Speed * cos(pre Alpha)) and
    y = (pre y + Speed * sin(pre Alpha)) and
    Alpha = pre Alpha - Beta and Beta = pre Beta and
    not is_inside(x,y,p1x,p1y,p2x,p2y,p3x,p3y,p4x,p4y)
  in
  let spiral() =
    x = (pre x + Speed * cos(pre Alpha)) and
    y = (pre y + Speed * sin(pre Alpha)) and

```

7.3 Calling external code

The `examples/lutin/external_code` directory of the LUTIN distribution contains a complete running (via the Makefile) example of calling extern code from LUTIN.

This directory contains a C file `foo.c` that defines a C function `rand_up_to`.

▷ **Example:** The `foo.c` file.

```
#include <stdlib.h>
#include <math.h>

#ifdef WIN32
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif

// Uniformly draws an integer between 0 and max.
// Not the most useful function for Lutin...
EXPORT int rand_up_to(int min, int max)
    double r = ((double) random ());
    int res = min + ((int) ((r * (((double) (max-min+1)) / ((double) RAND_MAX)))));
    return res;
```

This C function, as well as two other function that are part of the standard C math library is are used in the LUTIN program `call_external_c_code.lut`.

▷ **Example:** The `call_external_c_code.lut` file.

```
extern sqrt(x: real): real
extern sin(x: real) : real
extern rand_up_to(min, max: int): int

node Fun_Call() returns (f1: real = 1.0; f2: real; i: int) =
  loop {
    0.0 < f1 and f1 < 100.0
    and f2 = sin(sqrt(pre f1))
    and i = rand_up_to(0, 10)
  }
```

One needs to generate a shared lib from this C file (`foo.so` under unix, or `foo.dll` under windows), and to pass this shared library to the LUTIN interpreter via the `-L foo.so` option. Since the LUTIN file also uses the `sin` and the `sqrt` functions that are part of the standard math library, one also need to pass the `-L libm.so` option. For instance

```
lutin call_external_c_code.lut -m Fun_Call -L libm.so -L obj/foo.so
```

All this compilation process is illustrated in the `Makefile` contained in the directory.