Lutin

Semantics and compilation

Pascal Raymond

Erwan Jahier

Yvan Roux

VERIMAG, Grenoble

Contents

The language

Reactive systems

- **Lutin allows the description of non-deterministic reactive systems.**

  A system is declared with its parameters:

  ```
  system toto (x:  int; y:  bool)
  returns (a,b:bool; c:int) =  trace-exp
  ```

- **The body** *trace-exp* **describes the possible behaviours as a "language" whose words are *contraints* on the parameters (said *support variables*).**

Reactions

- An atomic behaviour (i.e. a non-deterministic reaction) is described as a constraint on the current and previous values of the support variables (`X` and `pre X`):

  alg-exp $::=$ algebraic Boolean expression

- Thus, variables may be:
  - ⋆ controlables (outputs),
  - ⋆ uncontrolable (inputs, `pre`'s).

- Performing an atomic reaction, for a *given* value of the uncontrolable variables, consists in generating *randomly* a value for the controlable variable that satisfies the constraint.

- N.B. if no solution exists, the system deadlocks

- **Example.** Let `x` be a Boolean input, `c` a real output, the execution of the following atomic reaction:

  `(x and (c <= 10.0) and (c > pre c))`

  - ⋆ produces, if `x` is true and `pre c` ≤ 10, some value `c` in the interval $]$`pre c`$, 10.0]$,
  - ⋆ otherwise deadlocks.

Sequencing reactions

- **Atomic reactions are combined with operators inspired by regular expressions:**

trace-exp ::= *trace-exp* **`fby`** *trace-exp*
| **`loop`** *trace-exp*
| **`{`***trace-exp* **`|`** ... **`|`***trace-exp* **`}`**

- **And some specific constructs:**
  - **`assert`** *alg-exp* **`in`** *trace-exp*

    distributes the constraint *alg-exp* **(Boolean expression) all along the behaviour** *trace-exp*
  - **`exist`** *ident* **`:`** *type* **`in`** *trace-exp*

    declares a local support variables (hiden outpout)
  - **`try`** *trace-exp* **`do`** *trace-exp*

    if the left *trace-exp* **deadlocks, behaves as the right one.**

Controling non-determinism

- **relative weights on choices (default is `1`):**

  `{` *trace-exp* `weight` *w1* `|`**...** `|` *trace-exp* `weight` *wn* `}`

  where each *wi* **is a *uncontrolable* integer expresssion**

- **iterations constrained by an interval:**

  `loop [`*min*`,`*max*`]`*trace-exp*

  where *min* **and** *max* **are *static* integer expressions**

- **iterations constrained by average and standard deviation:**

  `loop ~`*av*`:`*sd* *trace-exp*

  where *av* **and** *sd* **are *static* integer expressions**

Instantaneous loops

- An iteration loop may be instantaneous:
- Worst: `loop loop` c infinitely loops without doing anything if c is not satisfiable

Well founded loop principle

A loop may stop of continue, but if it continues it must generate something not empty.

In terms of regular languages: `loop` $t \ = \ (t \setminus \varepsilon)^*$

Non-determinism, deadlock and probabilities

- **Reactivity principle: a choice should not deadlock unless *all possibilities* deadlock**

- **N.B. reactivity is prior than weights:**

  { *t1* **`weight` 1000000 |** *t2* **`weight` 1 }**

  if *t1* **deadlocks while** *t2* **do not,** *t2* **is chosen.**

- **Example :** `{`X `weight 3 |` Y `weight 5 |` Z `}`

  Actual probabilities are:

deadlocking set	X	Y	Z	deadlock
$\emptyset$	3/9	5/9	1/9	0
$\{$X$\}$	0	5/6	1/6	0
$\{$Y$\}$	3/4	0	1/4	0
...	...	...	...	...
$\{$X,Y$\}$	0	0	1	0
...	...	...	...	...
$\{$X,Y,Z$\}$	0	0	0	1

Non-determinism, deadlock and priority

Even with a tiny weight, a non-deadlocking branch has some probability to be chosen. We need some well defined *priority choice*.

- **priority choice (aka "or else") :**

  {*t1* **|>***t2* **|>... |>***tn* **}**

- **Typical example: {***optimal* **|>***degraded* **|>***rescue* **|>***lost* **}**

Concurrency

- **Syntax :**

  { *trace-exp* **>... >** *trace-exp* **}**

- **All along the execution, each branch produces its own constraint, whose conjunction gives the global one.**

- **The statement terminates if and when all branches have terminated (cf. Esterel).**

- **If (at least) one branch deadlocks, the whole statement deadlocks.**

Concurrency versus probabilities

They do not live in harmony ...

`{ {`*X* `weight` 1000 `|`*Y* `}` `&>` `{`*A* `weight` 1000 `|`*B* `} }`

If *X* and *A* do not deadlock separately, while their conjunction do:

- **the most probable behaviour can be *Y*&>*A*, which is unfair for the first branch,**
- **or it can be *X*&>*B*, which is unfair for the second one.**

Design choice:

- **the first branch “plays” first, the second tries to do with it, etc.**
- **i.e., weights are treated in sequence.**
- **N.B. The syntax outlines the fact that the statement is not commutative.**

Exceptions

They allow to bypass the *normal* control-flow. They ressemble classical exceptions (caml, Java etc.) and also Esterel trap signals.

- **Declaration/scope:**

  `exception` *ident* **-- global**

  `exception` *ident* `in` *trace-exp* **-- local**

- **Raise statement:** `raise` *ident*

- **Catching point:**

  `catch` *ident* `in` *t1* `do` *t2*

  if *ident* **is raised within** *t1***, the control passes immediately to** *t2***.**

- **Shortcut:** `trap` *x* `in` *t1* `do` *t2*

  for : `exception` *x* `in catch` *x* `in` *t1* `do` *t2*

- **Deadlock: is equivalent to the raise of a *predefined* exception.**

  `catch DeadLock in` *t1* `do` *t2*

  is equivalent to: `try` *t1* `do` *t2*

- **Exception and concurrency:**
  - ⋆ **there is no “multiple” raise,**
  - ⋆ **just like weights, `raise` statements are treated in sequence, from left to right.**
  - ⋆ **e.g.** `{raise` **E** `&>`*X* `}` $\Leftrightarrow$ `raise` **E**

Modularity

The language provides a “functional” layer in order to:

- share definitions,
- define and re-use new operators, for both data and behaviours.
- An (abstract) type `trace` is defined, in order to characterize behaviour operators and parameters.
- The semantics is simply defined in terms of substitution (macros rather than functions).

- A macro can be global (outside a particular `system`) :

  `let` *ident* `(`*params* `)` `:` *type* `=` *exp*

  exp is either a *trace-exp* or a *data-exp*, according to its *type*.

- or it can be local to a *trace-exp*:

  `let` *ident* `(`*params* `)` `:` *type* `=` *exp* `in` *exp*

  in which case, classical scope rules hold.

- Input *params*, and output *type* are optional.

- Beware to not mistake support variables with input-free macros (aliases)

Examples

Of data combinator: the "interval" relation

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

Of trace combinator: the initial constraint

```
let assert_init(init: bool; t: trace): trace =
trap Stop in {
    -- implicit cast bool ⟶ (1-length) trace
    init
&>
    t fby raise Stop
}
```

Examples

Concurrent execution that terminates as soon as the second branch terminates:

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

Or as soon as one branch terminates:

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

Parameters and support variables

- **The type `trace` is rather abstract: what about the support ?**
- **Actually, it does not matter:**
  trace operators (pre- or user-defined) are in general polymorphic.
- **If a support variable is specifically expected as argument, the type must be over-specified:** *x*`:` *type* `ref`
  In this case, the type-checking will reject any call where the actual argument is not a support variable.
- **N.B. the flag `ref` is not really necessary unless some `pre` operator is used within the macro:**

```
let foo (x: bool)= ... pre x ... -- TYPE ERROR
let foo (x: bool ref)= ... pre x ... -- OK
```

Example

The “first-order-filter” relation:

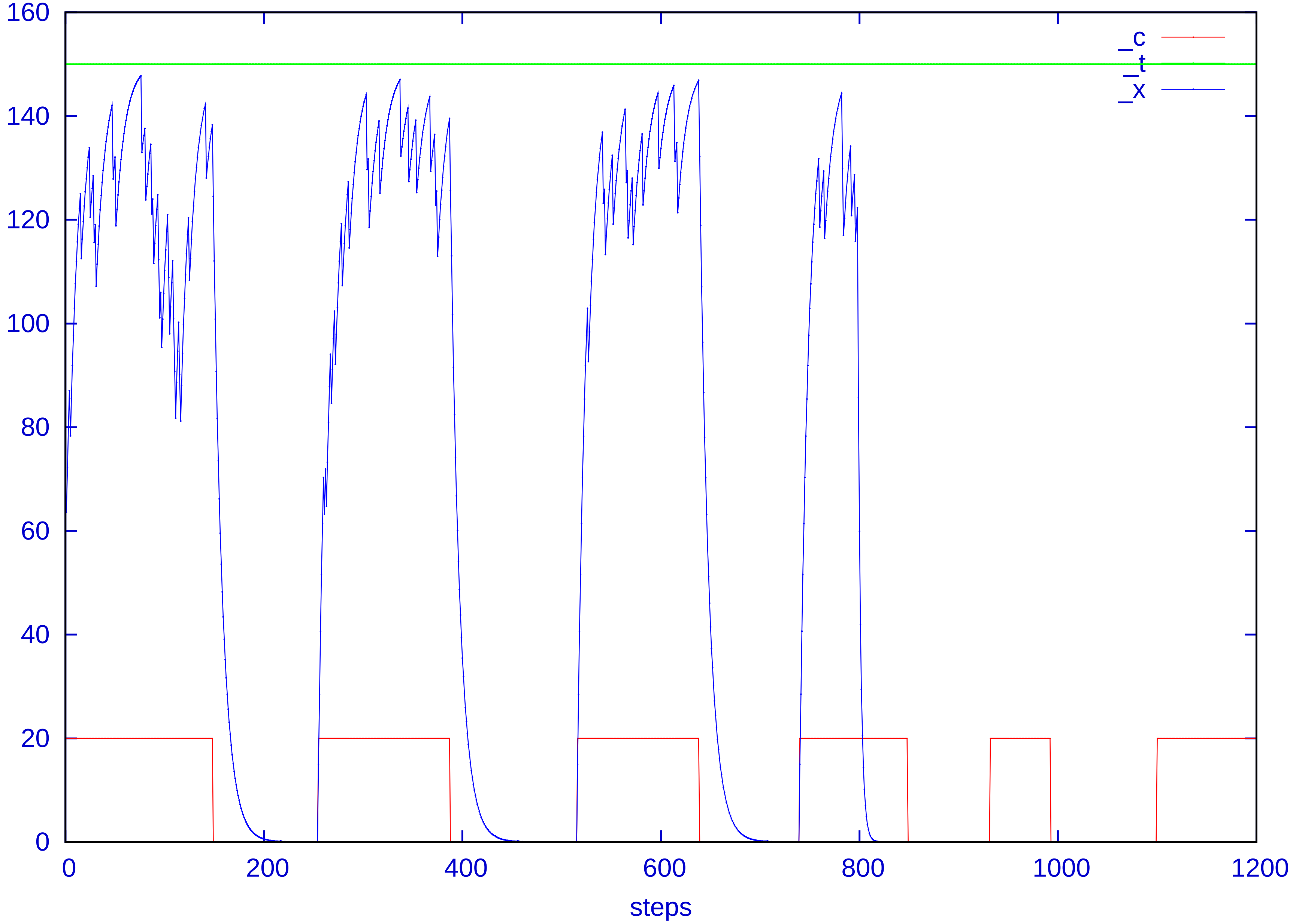
```
let fof (y: real ref; x, gain: real): bool =
        (y = gain*(pre y) + (1.0-gain)*x)
```

Example

A whole system:

- **Output `x` tends to input `t` when input `c` is true, otherwise tends to 0.**
- **The system works almost properly for about 1000 reactions: it may "miss" some `c` commands (1 times out of 10).**
- **Then it breaks down, and `x` quickly tends to 0.**

```
system foo(c: bool; t:real) returns (x: real) =
       within(x, -100.0, 100.0) fby
       local a:  real in
       let gen_gain(): trace = loop {
          within(a, 0.8, 0.9)
          fby loop[30,40] (a = pre a)
       } in
       as_long_as (
           gen_gain(),
           loop~1000:100 {
               (c and fof(x, a, t)) weight 9
           |   fof(x, a, 0.0)
           }
       ) fby loop fof(x, 0.7, 0.0)
```



160
140
120
100
80
60
40
20
0
0
200
400
600
800
1000
1200
steps
_c
_t
_x

Compiler front-end

Type/binding check

- rather classical

Expansion

- To an internal “core” language.
- Not really necessary, but modular compilation is far more complex (related to higher-order implementation).

Operational semantics is defined for the core language.

Abstract syntax

Trace (i.e. behaviour) expressions are:

empty:	ε	**empty filter:**	$t \setminus \varepsilon$
contraint:	c	**catch:**	$[t \xhookrightarrow{x} t']$
raise:	$\uparrow^{x}$	**choice:**	$t/w \mid t'/w'$
sequence:	$t \cdot t'$	**random loop:**	$t_i^{(\omega_c, \omega_s)}$
priority:	$t \succ t'$	**priority loop:**	$t^\star$
concurrency:	$t \,\&\, t'$		

- ε **and** $t \setminus \varepsilon$ **do not exist in the concrete syntax, but are helful for the semantics.**
- **The random loop syntax is explained in the sequel.**

Semantics

Execution environement

Constraint solving, weights evaluation and random selection are all devoted to the execution environment. We suppose that this environment provides:

- a predicate $e \models c$, true if c is satisfiable
- a "function" $Sort_e(t_1/w_1, \cdots, t_n/w_n) = [t_{i_1}, \cdots, t_{i_k}]$ that:
  - ⋆ evaluates the weights w_j,
  - ⋆ randomly computes priority range according to those relative weights,
  - ⋆ sorts the k traces with non-null weights according to those priorities. Note that, the result is empty if all weights are evaluated to 0.

Atomic action

- **The execution of a trace t within an environment e ($Run(e, t) = \alpha$), produces an action α which is either:**
  - ⋆ **a normal transition $\xrightarrow{c}$n where c is a satisfiable constraint and t rewrites into the (next) trace n.**
  - ⋆ **a termination $\rfloor^{x}$ where the flag x is either:**
    - ∗ **ε in case of normal termination,**
    - ∗ **δ in case of deadlock,**
    - ∗ **some user-defined exception.**
- **Let c be a satisfiable constraint in e, e rewrites itself (after random selection, memorization etc) in a next environment e': $e \xrightarrow{c} e'$.**

A complete run

The execution of a trace t_0 within an initial environment e_0 is defined as a sequence of environments: $(e_0, e_1, \cdots, e_n)$

where:

- $\exists c_0, \cdots, c_{n-1}\ \exists t_1, \cdots, t_n$ **such that**
- $\forall i = 0 \cdots n-1$
  $$Run(e_i, t_i) = \xrightarrow{c_i} t_{i+1} \text{ and } e_i \xrightarrow{c_i} e_{i+1}$$
- **and** $\exists x\ \ Run(e_n, t_n) = \ \Lsh^{x}$

The semantics function

- **Now that all "dirty stuff" is hiden within the environment, the semantics can be formally defined as a deterministic Run function**
- **Run is defined via an inductive function $\mathcal{R}_e$ (*run in e*) whose parameters are:**
  - **the trace t,**
  - **the *goto* continuation function $g(c, n)$ called whenever a transition is up to be fired whitin t,**
  - **the *stop* continuation function $s(x)$ called whenever t is up to terminates with the flag x.**

Top-level semantics

The main call is $Run(e,t) = \mathcal{R}_e(t,g,s)$ where the continuations are "trivial":

- $g(c,n) = \xrightarrow{c} n$
- $s(x) = \ \Lsh^{x}$

Empty behaviour

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

Exception raise

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

Constraint

This is where satisfiability matters:

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

Sequence

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

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

Priority choice

$$\begin{array}{rll} \mathcal{R}_e(t \succ t', g, s) = & \textbf{let} & \alpha = \mathcal{R}_e(t, g, s) \\ & \textbf{in} & (\alpha \neq \,\,\Lsh^{\delta})?\ \alpha\ :\ \mathcal{R}_e(t', g, s) \end{array}$$

Priority loop

- **empty filter replaces normal terminations by deadlocks:**
  $\mathcal{R}_e(t \setminus \varepsilon, g, s) = \mathcal{R}_e(t, g, s')$ **where:**
  $\star\ s'(x) = (x = \varepsilon)?\ \Lsh^{\delta}\ :\ s(x)$

- **Semantics is defined by syntactic equivalence:**

$$t^\star \;\Leftrightarrow\; (t \setminus \varepsilon) \cdot t^\star \quad \succ \quad \varepsilon$$

Catch

N.B. by construction, it only concerns δ and user-defined exception, (not the normal termination ε):

$\mathcal{R}_e([t \xhookrightarrow{z} t'], g, s) = \mathcal{R}_e(t, g', s')$ **where:**

- $g'(c, n) = g(c, [n \xhookrightarrow{z} t'])$
- $s'(x) = (x = z)?\ \mathcal{R}_e(t', g, s)\ :\ s(x)$

Concurrency

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

- $s'(x) = (x = \varepsilon)?\ \mathcal{R}_e(t', g, s)\ :\ s(x)$
- $g'(c, n) = \mathcal{R}_e(t', g'', s'')$ **where:**
  - ⋆ $s''(x) = (x = \varepsilon)?\ g(c, n)\ :\ s(x)$
  - ⋆ $g''(c', n') = (e \models c \wedge c')?\ g(c \wedge c', n \ \& \ n')\ :\ \ \Uparrow^{\delta}$

Weighted choice

In the current environment, weights are evaluated, and a random sort is performed according to the weights:

$$\mathcal{R}_e(t_1/w_1|\cdots|t_n/w_n,\ g,\ s) =$$

- $s(\delta)$ **if** $Sort(t_1/w_1, \cdots, t_n/w_n) = []$
  (i.e. all weights are actually null).

- $\mathcal{R}_e(t_{j_1} \succ \cdots \succ t_{j_k},\ g,\ s)$
  if $Sort(t_1/w_1, \cdots, t_n/w_n) = [t_{j_1}, \cdots t_{j_k}]$

Random loops

- The abstract syntax is: $t_i^{(\omega_c,\omega_s)}$
- i is an integer constant giving the the number of already performed iterations,
- The other labels are weight functions depending on i:
  - ⋆ the weight of "continue", $\omega_c(i)$
  - ⋆ the weight of "stop" $\omega_s(i)$
- Those functions are statically determined by the nature (interval, random) and the parameters of the concrete loop.
- The semantics follows:

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

Compiler back-end

Interpreter

- Constraints generation strictly follows the operational semantics.
- Constraints solving is performed by a module inherited from Lucky/Lurette (Lustre testing tool). The solver mixes BDDs and convex polyhedra.

Compilation into automata

- The target language is Lucky (explicit automata labelled with constraints and weights).
- The generation almost follows the semantics:
  - ⋆ states are derivations of the initial program
  - ⋆ termination is guaranted because the number of (different) derivations is finite (cf. regular expression to automata).
  - ⋆ deadlock management is simplified, because it is “built-in” in the target language.

Further work

- Compilation into flat automata is not satisfactory (combinational explosion). We plan to compile Lutin into hierarchical, concurrent automata (*à la SynchCharts*).
- The forthcomming version of the language will allow to define *mutually tail-recursive traces*; in other terms *explicit automata*.
- Data types should be extented (arrays, records ...)
- Some notion of signal and clock whould also be helpful.