Specifying and executing reactive scenarios with Lutin

Pascal Raymond, Yvan Roux, Erwan Jahier [1]

VERIMAG (CNRS, UJF, INPG)
Grenoble, France [2]

Abstract

This paper presents Lutin, a language for describing and generating random reactive behaviours. This language specifically targets the domain of reactive systems, where an execution is a (virtually) infinite sequence of input/output reactions. More precisely, it is dedicated to the description and the execution of constrained random scenarios. Its first use is for test sequence specification and generation. It can also be useful for early simulation of huge systems, where Lutin programs can be used to describe and simulate modules that are not yet fully developed.
The programming style mixes relational and imperative features. Basic statements are input/output relations, expressing constraints on a single reaction. Those constraints are then combined to describe non deterministic sequences of reactions. The language constructs are inspired by regular expressions, and process algebra (sequence, choice, loop, concurrency). Moreover, the set of statements can be enriched with user defined operators. A notion of stochastic directive is also provided, in order to finely influence the selection of a particular class of scenarios.

Keywords: Reactive systems, synchronous programming, language design, test, simulation.

1 Introduction

This paper presents an overview of the Lutin language, and its informal semantics. The complete formal semantics can be found in the extended version [11].

The targeted domain is the one of reactive systems, where an execution is a (virtually) infinite sequence of input/output reactions. Examples of such systems are control/command in industrial process, embedded computing systems in transportation. Testing reactive software raises specific problems. First of all, a single execution may require thousands of atomic reactions, and thus as many input vector values. It is almost impossible to write input test sequences by hand: they must be automatically generated according to some concise description. More specifically, the relevance of input values may depend on the behavior of the program itself: the program influences the environment which in turn influences the program. As a

[1] Emails: Pascal.Raymond@imag.fr, Yvan.Roux@imag.fr, Erwan.Jahier@imag.fr.
[2] URL: http://www-verimag.imag.fr

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

matter of fact, the environment behaves itself as a reactive system, whose environment is the program under test. This feed-back aspect makes off-line test generation impossible: testing a reactive system requires to run it in a simulated environment.

All these remarks have led to the idea of defining a language for describing random reactive systems. Since testing is the main goal, the programming style should be close to the intuitive notion of test scenarios, which means that the language is mainly control-flow oriented.

Note that, even if testing is the main goal, such a language can be useful for other purposes. In particular, for early prototyping and simulation, where constrained random programs can implement missing modules.

For programming random systems, one solution is to use a classical (deterministic) language together with a random procedure. In some sense, non-determinism is achieved by relaxing deterministic behaviors. We have adopted an opposite solution where non-determinism is achieved by constraining chaotic behaviors; in other terms, the proposed language is mainly relational, not functional.

In the language Lutin, non-predictable atomic reactions are expressed as input/output relations. Those atomic reactions are combined using statements like sequence, loop, choice or parallel composition. Since simulation (execution) is the goal, the language also provides stochastic constructs to express that some scenarios are more interesting/realistic than others.

Since the first version [10], the language has evolved with the aim of being a user-friendly, powerful programming language. The basic statements (inspired by regular expressions), have been completed with more sophisticated control structures (parallel composition, exceptions) and a functional abstraction has been introduced in order to provide modularity and reusability.

This work is clearly related to synchronous programming languages [3]. Some constructs of the language (traps and parallel composition) are directly inspired by the imperative synchronous language Esterel [2], while the relational part (constraints) is inspired by the declarative language Lustre [4].

Related works are abundant in the domain of models for non-deterministic (or stochastic) concurrent systems: Input/Output automata [7], and their stochastic extension [15]; stochastic extension of process algebra [6,1]. There are also relations with concurrent constraint programming [13], in particular with works that adopt a synchronous approach of time and concurrency [14,8]. A general characteristic of these models is that they are defined to perform analysis of stochastic dynamic systems (e.g., model checking, probabilistic analysis). On the contrary, Lutin is designed with the aim of being a user-friendly programming language. On one hand, the language allows to concisely describe, and then execute a large class of scenarios. On the other hand, it is in general impossible to decide if a particular behavior can be generated and even less with which probability.

The paper is organized as follows: after a general introduction, the main part presents the language constructs, illustrated by an incremental example. Then we briefly present the associated tool and conclude.

2 Overview of the language

2.1 Reactive, synchronous systems

The language is devoted to the description of reactive systems. Those systems have a cyclic behavior: they react to input values by producing output values and updating their internal state. We adopt the synchronous approach, which in this case simply means that the execution is viewed as a sequence of pairs "input values/output values".

Such a `system` is declared with its input and output variables; they are called the *support variables* of the system.

Example 2.1 We illustrate the language with a simple program that receives a Boolean input (`c`) and a real input (`t`) and produces a real output `x`. The high-level specification is that `x` should get closer to `t` when `c` is true, or should tend to zero otherwise. The header of the program is:

```
system foo(c: bool; t: real) returns (x: real) = statement
```

The core of the program consists of a *statement* describing the program behavior. The definition of *statement* is developed later.

During the execution, inputs are provided by the environment: they are called *uncontrollable variables*. The program reacts by producing outputs: they are called *controllable variables*.

2.2 Variables, reactions and traces

The core of the `system` is a statement describing a sequence of atomic reactions.

In Lutin, a reaction is not deterministic: it does not define precisely the output values, but states *constraints* on these values. For instance, the constraint `((x > 0.0) and (x < 10.0))` states that the current output should be some value comprised between 0 and 10.

Constraints may involve inputs, for instance: `((x > t - 2.0) and (x < t))`. In this case, during the execution, the actual value of `t` is substituted, and the resulting constraint is solved.

In order to express temporal constraints, *previous values* can be used: `pre` *id* denotes the value of the variable *id* at the previous reaction. For instance `(x > pre x)` states that `x` must increase in the current reaction. Like inputs, `pre` variables are uncontrollable: during the execution, their values are inherited from the past and cannot be changed: this is the *non-backtracking principle*.

Performing a reaction consists in producing, if it exists, a particular solution of the constraint. Such a solution may not exist.

Example 2.2 Consider the constraint:

```
(c and (x > 0.0) and (x < pre x + 10.0))
```

where `c` (input) and `pre` x (past value) are uncontrollable. During the execution, it may appear that `c` is false and/or that `pre` x is less than -10.0. In those cases, the constraint is unsatisfiable: we say that the constraint *deadlocks*.

Local variables may be useful auxiliaries for expressing complex constraints. They can be declared within a program:

```
local ident : type in statement
```

A local variable behaves as a hidden output: it is controllable and must be produced as long as the execution remains in its scope.

2.3 Composing reactions

A constraint (Boolean expression) represents an atomic reaction: it defines the relations between the current values of the variables. Scenarios are built by combining such atomic reactions *temporal statements*. We introduce the type `trace` for typing expressions made of temporal statements. A single constraint obviously denotes a trace of length 1; in other terms, expressions of type `bool` are implicitly cast to type `trace` when combined with temporal operators.

The basic trace statements are inspired by regular expression:

- sequence (`fby`: `trace` × `trace` → `trace`),
- unbounded loop (`loop`: `trace` → `trace`)
- non-deterministic choice (`|`: `trace` × `trace` → `trace`).

Because of this design choice, the notion of sequence differs from the one of Esterel, which is certainly the reference in control-flow oriented synchronous language[2]. In Esterel, the sequence (semicolon) is instantaneous, while the Lutin construct `fby` "takes" one instant of time, just like in classical regular expressions.

Example 2.3 With those operators, we can propose a first version for our example. In this version, the output tends to 0 or `t` according to a first order filter. The non-determinism resides in the initial value, and also in the fact that the system is subject to failure and may miss the `c` command.

```
((-100.0 < x) and (x < 100.0)) fby -- initial constraint
loop {
     (c and (x = 0.9*(pre x) + 0.1*t)) -- x gets closer to t
|    ((x = 0.9*(pre x))                 -- x gets closer to 0
}
```

Initially, the value of `x` is (randomly) chosen between -100 and +100, then, forever, it may tend to `t` or to 0.

Note that, inside the loop, the first constraint (x tends to t) is not satisfiable unless `c` is true, while the second is always satisfiable. If `c` is false, the first constraint *deadlocks*. In this case, the second branch (x gets closer to 0) is necessarily taken. If `c` is true, both branches are feasible: one is randomly selected, and the corresponding constraint is solved.

This illustrates an important principle of the language: the *reactivity principle* states that a program may only deadlock if all its possible behaviors deadlock.

2.4 Traces, termination and deadlocks

Because of non-determinism, a behavior has in general several possible first reactions (constraints). According to the reactivity principle, it deadlocks only if all those constraints are not satisfiable. If at least one reaction is satisfiable, it must "do something": we say that it is *startable*.

Termination, startability and deadlocks are important concepts of the language; here is a more precise definition of the basic statements according to these concepts:

- A constraint *c*, if it is satisfiable, generates a particular solution and terminates, otherwise it deadlocks.
- *st1* `fby` *st2* executes *st1*, and, if and when it terminates, executes *st2*. If *st1* deadlocks, the whole statement deadlocks.
- `loop` *st*, if *st* is startable, behaves as *st* `fby loop` *st*, otherwise it terminates. Intuitively, the meaning is "loop as long as possible".
- `{`*st1* `|`... `|`*stn* `}` randomly chooses one of the *startable* statements from *st1*...*stn*. If none of them are startable, the whole statement deadlocks.
- The 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.
- `try` *st1* `do` *st2* catches any deadlock occurring *during the execution* of *st1* (not only at the first step). In case of deadlock, the control passes to *st2*.

2.5 Well-founded loops

Let's denote by ε the identity element for `fby` (i.e. the unique behavior such that b `fby` ε = ε `fby` b = b). Although this "empty" behavior is not provided by the language, it is helpful for illustrating a problem raised by nested loops.

As a matter of fact, the simplest way to define the loop is to state that "`loop` c" is equivalent to "c `fby loop` c `|>`ε", that is, try in priority to perform one iteration, and if it fails, stop. According to this definition, nested loops may generate infinite, instantaneous loops, as shown in the following example:

Example 2.4 `loop {loop` *c*`}`

Performing an iteration of the outer loop consists in executing the inner loop `loop` *c*. If *c* is not currently satisfiable, `loop` *c* terminates immediately, and thus, the iteration is actually "empty": it generates no reaction. However, since it is not a deadlock, this strange behavior is considered by the outer loop as a normal iteration. As a consequence, another iteration is performed, which is also empty, and so on: the outer loop keeps the control forever but does nothing.

One solution is to state that such programs are incorrect. Statically checking whether a program will infinitely loop or not is undecidable: it may depend on arbitrarily complex conditions. Some over-approximation is necessary, which will (hopefully) reject all the incorrect programs, but also lots of correct ones. For instance, a program as simple as: "`loop` { {`loop` a} `fby` {`loop` b } }" will certainly be rejected as potentially incorrect.

We think that such a solution is too much restrictive and tedious for the user, and we prefer to slightly modify the semantics of the loop. The solution retained is to introduce the *well-founded loop principle*: a loop statement may stop or continue, but if it continues it must do something. In other terms, empty iterations are dynamically forbidden.

The simplest way to explain this principle is to introduce an auxiliary operator $st\backslash_\varepsilon$: if *st* terminates immediately, $st\backslash_\varepsilon$ deadlocks, otherwise it behaves as *st*. The correct definition of `loop` *st* follows:

- if $st\backslash_\varepsilon$ is startable, behaves as $st\backslash_\varepsilon$ `fby loop` *st*,
- otherwise terminates.

2.6 Influencing non-determinism

When executing a non-deterministic statement, the problem of which choice should be preferred arises. The default is that, if k out of the n choices are startable, each of them is chosen with a probability $1/k$.

In order to influence this choice, the language provides a mechanism of *relative weights*:

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

Weights are basically integer constants, and their interpretation is straightforward: a branch with a weight 2 has twice the chance to be tried than a branch with weight 1. More generally, a weight can depend on the environment and on the past: it is given as a integer expression depending on uncontrollable variables. In this case, weight expressions are dynamically evaluated before performing the choice.

Example 2.5 In a first version (example 2.3), our example system may ignore the command `c` with a probability 1/2. This case can be made less probable by using weights (when omitted, a weight is implicitly 1):

```
loop {
     (c and (x = 0.9*(pre x) + 0.1*t)) weight 9
|    ((x = 0.9*(pre x))
}
```

In this new version, a true occurrence of `c` is missed with the probability 1/10.

Note that weights are not only directives. Even with a big weight, a non startable branch has a null probability to be chosen, which is the case in the example when `c` is false.

2.7 Random loops

We want to define some loop structure where the number of iterations is not fully determined by deadlocks. Such a construct can be based on weighted choices, since a loop is nothing but a binary choice between stopping and continuing. However, it seems more natural to define it in terms of expected number of iterations. Two loop "profiles" are provided:

- `loop[`*min*,*max*`]`: the number of iterations should be between the constants *min* and *max*

- `loop~`*av*:*sd*: the average number of iteration should be *av*, with a standard deviation *sd*.

Note that random loops, just like other non-deterministic choices, follow the *reactivity principle*: depending on deadlocks, looping may sometimes be required or impossible. As a consequence, during an execution, the actual number of iterations may significantly differ from the "expected" one.

Moreover, just like the basic `loop`, they follow the *well-founded loop principle*, which means that, even if the core contains nested loops, it is impossible to perform "empty" iterations.

2.8 Parallel composition

The parallel composition of Lutin is synchronous: each branch produces, at the same time, its local constraint. The global reaction must satisfy the conjunction of all those local constraints. This approach is similar to the one of temporal concurrent constraint programming [8].

The normal termination of the concurrent execution is defined by:

- if one or more branches abort (deadlock), the whole statement aborts,
- otherwise, the parallel composition terminates if and when all the branches have terminated.

The concrete syntax suggests a non-commutative operator, this choice is explained below.

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

2.9 Parallel composition versus stochastic directives

It is impossible to define a parallel composition which is fair according to the stochastic directives (weights), as shown in the following example.

Example 2.6 Consider the statement:
`{ {`X `weight` 1000 `|` Y `} &> {`A `weight` 1000 `|` B `} }`
where X, A, $X \wedge B$, $A \wedge Y$ are all startable, but not $X \wedge A$.
The higher priority can be given to:

- $X \wedge B$, but it does not respect the stochastic directive of the second branch,
- $A \wedge Y$, but it does not respect the stochastic directive of the first branch.

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

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

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

The concrete syntax (`&>`) has been chosen to reflect the fact that the operation is not commutative: the treatment is parallel for the constraints (conjunction), but sequential for stochastic directives (weights).

2.10 Exceptions

User-defined exceptions are mainly a means for by-passing the normal control flow. They are inspired by exceptions in classical languages (Ocaml, Java, Ada) and also by the trap signals of Esterel.

Exceptions can be globally declared outside a system (`exception` *ident*) or locally within a statement, in which case the standard binding rules hold:

`exception` *ident* `in` *st*

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

`raise` *ident*

and caught with the statement:

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

If the exception is raised in *st1*, the control immediately passes to *st2*. The `do` part may be omitted, in which case the control passes in sequence.

2.11 Modularity

An important point is that the notion of `system` is not a sufficient modular abstraction. In some sense, systems are similar to main programs in classical languages: they are entry point for the execution but are not suitable for defining "pieces" of behaviors.

Data combinators.

A good modular abstraction would be one that allows to enrich the set of combinators. Allowing the definition of data combinators is achieved by providing a functional-like level in the language. For instance, one can program the useful "within an interval" constraint:

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

Once defined, this combinator can be instantiated, for instance:

```
within(a, 0.8, 0.9)
```

or

```
within(a + b, c - 1.0, c + 1.0)
```

Note that such a combinator is definitively not a function in the sense of computer science: it actually computes nothing. It is rather a well typed *macro* defining how to build a Boolean expression with three real expressions.

Reference arguments.

Some combinators specifically require support variables as argument (input, output, local). This is the case for the operator `pre`, and, as a consequence, for any combinator using a `pre`. This situation is very similar to the distinction between "by reference" and "by value" parameters in imperative languages. We classically solve the problem by adding the flag `ref` to the type of such parameters.

Example 2.7 The following combinator defines the generic first order filter constraint. The parameter `y` must be a real support variable (`bool ref`) since its previous value is required. The other parameters can be any expressions of type real.

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

Trace combinators.

User-defined temporal combinators are simply macros of type `trace`.

Example 2.8 The following combinator is a binary parallel composition where the termination is enforced when the second argument terminates:

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

Local combinators.

A macro can be declared within a statement, in which case the classical binding rules hold; in particular, a combinator may have no parameter at all.

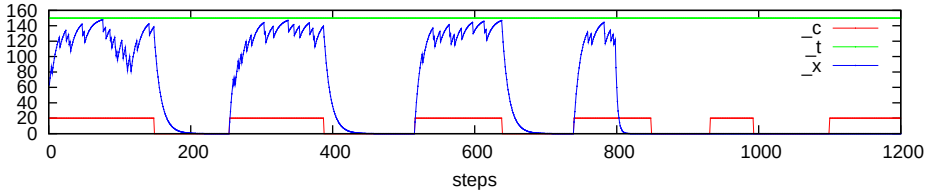
```
let id (params): type = statement in statement
```

Example 2.9 We can now write more elaborated scenarios for the system of example 2.3. In this new version, the system works almost properly for about 1000 reactions: if `c` is true, `x` tends to `t` 9 times out of 10, otherwise it tends to 0. During this phase, the gain for the filters (`a`) randomly changes each 30 to 40 reactions. At last, the system 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)
```

The following timing diagram shows an execution of this program. Inputs values are provided by the environment (i.e., us) according to the following specification:

- the input `t` is constant (150),
- the command `c` toggles each about 100 steps.



3 Semantics and implementation

This section briefly presents the principles of the language execution. The complete description of the semantics is presented in the extended version [11].

3.1 The execution environment

An important point is that the reference semantics defines how a Lutin program generates a sequence of constraints, not how those constraints are solved. In some sense, the constraint solver should be viewed as a parameter of the execution: several solvers may be provided, and the user may choose according to his needs a more or less sophisticated one.

The constraint solver, and also the variables management, are "hidden" in the *execution environment*. This environment:

- stores the variable values (inputs and memories),
- evaluates the weights,
- performs pseudo-random choices,
- solves the constraints and selects particular solutions.

3.2 Execution steps

A Lutin program can be viewed as the encoding of a state/transition machine: a state is characterized by the statement t to execute. A transition consists in producing a satisfiable constraint c_j, and going to the corresponding next-state t_{j+1}. The initial state t_0 is obviously the top statement of main program.

Generating a transition (we note $e_j : t_j \xrightarrow{c} t_{j+1}$) is the "normal behavior". But the program may also stops. In this case, we note $e_j : t_j \uparrow^x$, where x is a *termination flag*, which is either:

- ε, denoting a normal termination (the program has reached its end),
- δ (for *deadlock*), denoting that no satisfiable constraint has been found,
- some user-defined exception, produced by a `raise` statement.

When a transition is fired, the environment evolves according to the produced constraint into an updated environment e_{j+1}:

- it randomly selects a particular solution of c_j,
- it emits the outputs,
- it stores the memory and reads new inputs.

We denote this rewriting of the environment by $c_j : e_j \vdash e_{j+1}$.

3.3 Complete execution

For the sake of simplicity, we do not take into account neither the inputs nor the distinction between output and local variables: as a consequence, the observed result of an execution is simply a sequence of values matching the constraints generated by the program $(v_0, v_1, \cdots v_n)$. Finally, a complete execution is a sequence of pairs (environment, statement):

$$(e_0, t_0) \xrightarrow{v_0} (e_1, t_1) \xrightarrow{v_1} \cdots (e_n, t_n) \xrightarrow{v_n} (e_{n+1}, t_{n+1})$$

such that it exists a sequence of constraints $c_0, c_1, \cdots c_n$ and a termination flag x, with, for all $j \leq n$:

- each v_j is a solution of c_j,
- $e_j : t_j \xrightarrow{c_j} t_{j+1}$,
- $c_j : e_i \vdash e_{i+1}$,
- $e_{n+1} : t_{n+1} \overset{x}{\downarrow}$

3.4 Notes on constraint solvers

The core semantics only defines how constraints are generated, but not how they are solved. This choice is motivated by the fact that there is no "ideal" solver:

- it may be efficient but limited in terms of capabilities,
- it may be powerful, but likely to be very costly in terms of time and memory.

The idea is that the user may choose between several solvers (or several options of a same solver) the one which best fits his needs.

Actually, we use the solver that have been developed for the testing tool Lurette [12,5]. This solver is quite powerful, since it covers Boolean algebra and linear arithmetics.

3.5 Implementation

A prototype has been developed. This tool can:

- interpret/simulate Lutin programs in a file-to-file (or pipe-to-pipe) manner. This tool serves for simulation/prototyping: several Lutin simulation sessions can be combined with other reactive process in order to animate a complex system.
- compile Lutin programs into the internal format of the testing tool Lurette. This format, called Lucky, is based on flat, explicit automata [9]. In this case, Lutin serves as a high-level language for designing test scenarios.

4 Conclusion

We propose a language for describing constrained-random reactive systems. Its first purpose is to describe test scenarios, but it may also be useful for prototyping and simulation.

We have developed a compiler/interpreter which implements the operational semantics informally presented here. Thanks to this tool, the language is integrated

into the framework of the Lurette tool, where it is used to describe test scenarios. Further works concerns the integration of the language within a more general prototyping framework.

Other works concern the evolution of the language. We plan to introduce a notion of signal (i.e., event), which is useful for describing values that are not always available (this is related to the notion of clocks in synchronous languages). We also plan to allow the definition of (mutually) tail-recursive traces. Concretely, that means that a new programming style would be allowed, based on explicit, concurrent and hierarchic automata.

References

[1] Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. Stochastic process algebra: From an algebraic formalism to an architectural description language. *Lecture Notes in Computer Science*, 2459, 2002.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, sep 1991.

[5] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. In *Software Tools for Technology Transfer*, volume 8, pages 517–530, November 2006.

[6] B. Jonsson, K. Larsen, and W. Yi. Probabilistic extensions of process algebras, 2001.

[7] Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.

[8] Mogens Nielsen, Catuscia Palamidessi, and Frank D.Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput*, 9(1):145–188, 2002.

[9] P. Raymond, E. Jahier, and Y. Roux. Describing and executing random reactive systems. In *SEFM 2006, 4th IEEE International Conference on Software Engineering and Formal Methods*, Pune, India, September 2006.

[10] P. Raymond and Y. Roux. Describing non-deterministic reactive systems by means of regular expressions. In *First Workshop on Synchronous Languages, Applications and Programming, SLAP'02*, Grenoble, April 2002.

[11] P. Raymond, Y. Roux, and E. Jahier. Lutin: a language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, Model-driven High-level Programming of Embedded Systems Selected papers from Sla++p'07 and Sla++p'08, to appear 2008.

[12] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[13] V. A. Saraswat, editor. *Concurrent Constraint Programming*. MIT Press, 1993.

[14] Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *LICS*, pages 71–80, 1994.

[15] S.-H. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1-2):1–38, 1997.