Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2008, Article ID 753821, 11 pages
doi:10.1155/2008/753821

Research Article

Lutin: A Language for Specifying and Executing Reactive Scenarios

Pascal Raymond, Yvan Roux, and Erwan Jahier

VERIMAG (CNRS, UJF, INPG), 2 avenue de Vignate, Gières 38610, France

Correspondence should be addressed to Pascal Raymond, pascal.raymond@imag.fr

Received 13 September 2007; Accepted 10 January 2008

Recommended by Michael Mendler

This paper presents the language Lutin and its operational semantics. 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. 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 directives is also provided in order to finely influence the selection of a particular class of scenarios.

1. INTRODUCTION

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 matter of fact, the environment behaves itself as a reactive system, whose environment is the program under test. This feedback aspect makes offline 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.

The language can also be useful for early prototyping and simulation, where constrained random programs can implement missing modules.

1.1. Our proposal: Lutin

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

In the language Lutin, nonpredictable 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 [1], the language has evolved with the aim of being a user-friendly, powerful programming language. The basic statements (inspired by regular expressions) have been augmented with more sophisticated control structures (parallel composition, exceptions) and

a functional abstraction has been introduced in order to provide modularity and reusability.

1.2. Related works

This work is related to synchronous programming languages [2, 3]. Some constructs of the language (traps and parallel composition) are directly inspired by the imperative synchronous language Esterel [4], while the relational part (constraints) is inspired by declarative languages like Lustre [5] and Signal [6].

Related works are abundant in the domain of models for nondeterministic (or stochastic) concurrent systems: Input/Output automata [7], and their stochastic extension [8] (stochastic extension of process algebra [9, 10]). There are also relations with concurrent constraint programming [11], in particular, with works that adopt a synchronous approach of time and concurrency [12, 13]. However, the goals are rather different; our goal is to maintain an infinite interaction between constraints generators, while concurrent constraint programming aims at obtaining the solution of a complex problem in a (hopefully) finite number of interactions.

Moreover, 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 specifically designed for simulation rather than general analysis. 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.

1.3. Plan

The article starts with an informal presentation of the language. Then, the operational semantics is formally defined in terms of constraints generator. Some important aspects, in particular constraints solving, are parameters of this formal semantics; they can be adapted to favor the efficiency or the expressive power. These aspects are presented in the implementation section. Finally, we conclude by giving some possible extensions of this work.

2. OVERVIEW OF THE LANGUAGE

2.1. Reactive, synchronous systems

The language is devoted to the description of nondeterministic 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 here 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 1. We illustrate the language with a simple "tracker" program that receives a boolean input (`c`) and a real input (`t`) and produces a real output (`x`). The high-level specification of the tracker is that the output `x` should get closer to the input `t` when the command `c` is true or should tend to zero otherwise. The header of the program is

```
system tracker (c: bool; t: real) returns (x: real)
  = statement.
```
(1)

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 system 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 uniquely the output values, but states some *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 *nonbacktracking principle*.

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

Example 2. Consider the constraint

(`c and` (`x` > 0.0) `and` (`x` < `pre x` + 10.0)), (2)

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*. (3)

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 relations between the current values of

the variables. Scenarios are built by combining such atomic reactions with *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, and have following signatures:

(i) `fby` : `trace` × `trace` → `trace` [sequence]
(ii) `loop` : `trace` → `trace` [unbounded loop]
(iii) | : `trace` × `trace` → `trace` [nondeterministic choice]

Using regular expressions makes the notion of sequence quite different from the one of Esterel, which is certainly the reference in control-flow oriented synchronous language [4]. In Esterel, the sequence (semicolon) is instantaneous, while the Lutin construct `fby` "takes" one instant of time, just like in classical regular expressions.

Example 3. With those operators, we can propose a first version of our example. In this version, the output tends to 0 or `t`according to a first-order filter. The nondeterminism 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
}
```

(4)

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, which states that a program may only deadlock, if all its possible behaviors deadlock.

2.4. Traces, termination, and deadlocks

Because of nondeterminism, 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.

(i) A constraint c, if it is satisfiable, generates a particular solution and terminates, otherwise it deadlocks.
(ii) $st1$ `fby` $st2$ executes $st1$, and if and when it terminates, it executes $st2$. If $st1$ deadlocks, the whole statement deadlocks.
(iii) Loop st, if st is startable, behaves as st `fby loop` st, otherwise it terminates. Indeed, once started, st `fby loop` st may deadlock if the first st, and so on. Intuitively, the meaning is "loop as long as starting a step is possible."
(iv) $\{st1|\cdots|stn\}$ randomly chooses one of the *startable* statements from $st1,\ldots,stn$. If none of them are startable, the whole statement deadlocks.
(v) The priority choice $\{st1|>\cdots|>stn\}$ behaves as $st1$ if $st1$ is startable, otherwise, behaves as $st2$ if $st2$ is startable and so on. If none of them are startable, the whole statement deadlocks.
(vi) `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 us 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 and instantaneous loops, as shown in the following example.

Example 4.

`loop{loop` c}. (5)

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 reject such programs. 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 $st \setminus_\varepsilon$. If st terminates immediately, $st \setminus_\varepsilon$ deadlocks, otherwise it behaves as st. The correct definition of `loop` st follows:

(i) if $st \setminus_\varepsilon$ is startable, it behaves as $st \setminus_\varepsilon$ `fby loop` st,
(ii) otherwise `loop` st terminates.

2.6. Influencing non-determinism

When executing a nondeterministic statement, the problem of which choice should be preferred arises. The solution retained 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 the concept of *relative weights:*

$$\{st1 \texttt{ weight } w1 \mid \cdots \mid stn \texttt{ weight } wn\}. \tag{6}$$

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 an integer expression depending on uncontrollable variables. In this case, weight expressions are evaluated at runtime before performing the choice.

Example 5. In a first version (Example 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))
}
```
(7)

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 nonstartable 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 as follows.

(i) `loop`$[\min, \max]$: the number of iterations should be between the constants min and max.
(ii) `loop`$\tilde{}\,av : sd$: the average number of iteration should be av, with a standard deviation sd.

Note that random loops, just like other nondeterministic 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 (see Sections 4 and 5.3).

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 constraints. The global reaction must satisfy the conjunction of all those local constraints. This approach is similar to the one of temporal concurrent constraint programming [12].

A parallel composition may deadlock for the following two reasons.

(i) Obviously, if one or more branches deadlock, the whole statement aborts.
(ii) It may also appear that each individual statement has one or more possible behaviours, but that none of the conjunctions are satisfiable, in which case the whole statement aborts.

If no deadlock occurs, the concurrent execution terminates, if and when all the branches have terminated (just like in the Esterel Language).

One can perform a parallel composition of several statements as follows:

$$\{st1 \ \& > \cdots \& > stn\}. \tag{8}$$

The concrete syntax suggests a noncommutative operator; this choice is explained in the next section.

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 illustrated in the following example.

Example 6. Consider the statement

$$\{\{X \texttt{ weight } 1000 \mid Y\} \& > \{\{A \texttt{ weight } 1000 \mid B\}\}, \tag{9}$$

where $X, A, X \wedge B, A \wedge Y$ are all startable, but not $X \wedge A$.

The higher priority can be given to

(i) $X \wedge B$, but it would not respect the stochastic directive of the second branch;
(ii) $A \wedge Y$, but it would not respect the stochastic directive of the first branch;

In order to deal with this issue, the stochastic directives are not treated in parallel, but in *sequence*, from left to right.

(i) The first branch "plays" first, according to its local stochastic directives.
(ii) The next ones make their choice according to what has been chosen for the previous ones.

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 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

$$\texttt{exception}\ \textit{ident}\ \texttt{in}\ \textit{st}. \tag{10}$$

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

$$\texttt{raise}\ \textit{ident} \tag{11}$$

and caught with the statement:

$$\texttt{catch}\ \textit{ident}\ \texttt{in}\ \textit{st}1\ \texttt{do}\ \textit{st}2. \tag{12}$$

If the exception is raised in *st*1, the control immediately passes to *st*2. 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;

$$\begin{aligned}&\texttt{let within}\,(\texttt{x}, \min, \max : \texttt{real}) : \texttt{bool}\\ &\quad = (\texttt{x} >= \min)\ \texttt{and}\ (\texttt{x} <= \max).\end{aligned} \tag{13}$$

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

$$\texttt{within}\,(\texttt{a}, 0.8, 0.9) \tag{14}$$

or

$$\texttt{within}\,(\texttt{a} + \texttt{b}, \texttt{c} - 1.0, \texttt{c} + 1.0). \tag{15}$$

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. Therefore, we solve the problem in a similar manner by adding the flag ref to the type of such parameters.

Example 7. The following combinator defines the generic first-order filter constraint. The parameter y must be a real support variable (real ref) since its previous value is required. The other parameters can be any expressions of type real.

$$\begin{aligned}&\texttt{Let fof}\,(\texttt{y} : \texttt{real}; \texttt{gain}, \texttt{x} : \texttt{real}) : \texttt{bool} =\\ &\quad (\texttt{y} = \texttt{gain} * (\texttt{pre y}) + (1.0 - \texttt{gain}) * \texttt{x}).\end{aligned} \tag{16}$$

Trace combinators

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

Example 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{                                  (17)
    X & > {Y fby raise Stop}
}.
```

Local combinators

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

$$\texttt{Let}\ \textit{id}\,([\textit{params}]) : \textit{type} = \textit{statement}\ \texttt{in}\ \textit{statement}. \tag{18}$$

Example 9. We can now write more elaborated scenarios for the system of Example 3. For the very first reaction (line 2), the output is randomly chosen between −100 and +100, then the system enters its standard behavior (lines 3 to 14). A local variablea is declared, which will be used to store the current gain (line 3). An intermediate behavior (lines 4 to 6) is declared, which defines how the gain evolves; it is randomly chosen between 0.8 and 0.9, then it remains constant during 30 to 40 steps, and so on. Note that, this combinator has no parameter since it directly refers to the variable a. Lines 7 to 14 define the actual behavior; the user-defined combinator as long as runs in parallel the behavior gen_gain (line 8) with the normal behavior (9 to 11). In the normal behavior, the system works almost properly for about 1000 reactions; if c is true, x tends to t 9 times out of 10 (line 10), otherwise it tends to 0 (line 11). As soon as the normal behavior terminates, the whole parallel composition

```
(1) system tracker(c : bool; t : real) returns (x : real) =
(2)   within(x, −100.0, 100.0) fby
(3)   local a : real in
(4)   let gen_gain() : trace = loop{
(5)     within(a, 0.8, 0.9) fby loop[30, 40] (a = pre a)
(6)   } in
(7)   as_long_as(
(8)     gen_gain(),
(9)     loop~1000 : 100{
(10)        (c and fof (x, a, t)) weight 9
(11)      | fof (x, a, 0.0)
(12)      }
(13)  ) fby
(14)    loop fof (x, 0.7, 0.0)
```

Figure 1: A full example: the "tracker" program.

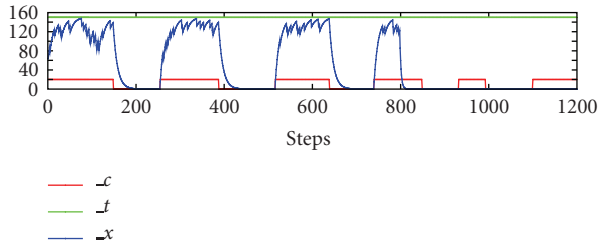

Figure 2: An execution of the tracker program.

terminates (definition of as long as). Then, the system breaks down andx quickly tends to 0 (line 14).

Figure 2 shows the timing diagram, a particular execution of this program. Input values are provided by the environment (i.e., us) according to the following specification, the input `t` remains constant (150) and the command `c` toggles each about 100 steps.

3. SYNTAX SUMMARY

Figure 3 summarizes the concrete syntax of Lutin. The detailed syntax for *expression* is omitted. They are made of classical algebraic expressions with numerical and logical operators, plus the special operator `pre`. The supported type identifiers are currently `bool`, `int`, and `real`.

We do not present the details of the type checking, which is classical and straightforward. The only original check concerns the arguments of the loop profiles and of the `weight` directive, that must be *uncontrollable* expressions (not depending on output or local variables).

4. OPERATIONAL SEMANTICS

4.1. Abstract syntax

We consider here a type checked Lutin program. For the sake of simplicity, the semantics is given on the flat language. User-defined macros are inlined, and local variables are made global through some correct renaming of identifiers. As a consequence, an abstract system is simply a collection of variables (inputs, outputs, and locals) and a single abstract statement.

We use the following abstract syntax for statements, where the intuitive meaning of each construct is given between parenthesis:

$$\begin{aligned} t ::= {} & c\ (constraint). \mid \varepsilon(empty\ behavior). \\ & \mid t \setminus \varepsilon(empty\ filter). t \cdot t\ (sequence). \\ & \mid t^*(priority\ loop). \mid t_k^{(w_c, w_s)}(random\ loop). \\ & \mid \overset{x}{\hookrightarrow}(raise). \mid [t \overset{x}{\hookrightarrow} t']\ (catch). \\ & \mid \succ_{i=1}^{n} t_i(priority). \mid\mid_{i=1}^{n} t_i/w_i(choice). \\ & \mid \&_{i=1}^{n} t_i\ (parallel). \end{aligned} \tag{19}$$

This abstract syntax slightly differs from the concrete one on the following points.

(i) The empty behavior (ε) and the empty behavior filter ($t \setminus \varepsilon$) are internal constructs that will ease the definition of the semantics.

(ii) Random loops are *normalized* by making explicit their weight functions:

  (a) the stop function ω_s takes the number of already performed iterations and returns the relative weight of the "stop" choice;

  (b) the continue function ω_c takes the number of already performed iterations and returns the relative weight of the "continue" choice.

  These functions are completely determined by the loop profile in the concrete program (interval or average, together with the corresponding static arguments). See Section 5.3 for a precise definition of these weight functions.

(iii) The number of already performed iterations (k) is syntactically attached to the loop; this is convenient to define the semantics in terms of rewriting (in the initial program, this number is obviously set to 0).

```
    system  ::= system  ident([params]) returns ([params]) = statement
    params  ::= ident : type{; ident : type}
 statement  ::= expression | {statement} | statement fby statement
              |loop[loop-profile] statement
              |exception ident in statement | raise ident
              |try statement [do  statement] | catch ident in statement
              |let ident ([params]) : type =  statement in statement
              |local ident : type in statement
              |statement[weight expression]{| statement[weight expression]}
              |statement{| > statement}
              |statement{& > statement}
loop-profile ::= [expression, expression]
              |~expression : expression
      type ::= ident[ref])
```

Figure 3: The concrete EBNF syntax of Lutin.

Definition 1. $\mathcal{T}$ denotes the set of trace expressions (as defined above) and $\mathcal{C}$ denotes the set of constraints.

4.2. The execution environment

The execution takes place within an environment which stores the variable values (inputs and memories). Constraint resolution, weight evaluation, and random selection are also performed by the environment. We keep this environment abstract. As a matter of fact, resolution capabilities and (pseudo)random generation may vary from one implementation to another and they are not part of the reference semantics.

The semantics is given in term of constraints generator. In order to generate constraints, the environment should provide the two following procedures.

Satisfiability

the predicate $e \models c$ is true if and only if the constraint c is satisfiable in the environment e.

Priority sort

Executing choices first requires to evaluate the weights in the environment. This is possible (and straightforward) because weights may dynamically depends on uncontrollable variables (memories, inputs), but not on controllable variables (outputs, locals). Some weights may be evaluated to 0, in which case the corresponding choice is forbidden. Then a random selection is made, according to the actual weights, to determine a total order between the choices.

For instance, consider the following list of pairs (trace/weight), where x and y are uncontrollable variables,

$$(t_1/x+y), (t_2/1), (t_3/y), (t_4/2). \tag{20}$$

In an environment, where $x = 3$ and $y = 0$, weights are evaluated to

$$(t_1/3), (t_2/1), (t_3/0), (t_4/2). \tag{21}$$

The choice t_3 is erased and the remaining choices are randomly sorted according to their weights. The resulting (total) order may be

(i) t_1, t_2, t_4 with a probability $3/6 \times 1/3 = 1/6$,
(ii) t_1, t_4, t_2 with a probability $3/6 \times 2/3 = 1/3$,
(iii) t_4, t_1, t_2 with a probability $2/6 \times 3/4 = 1/4$,
(iv) so on.

All these treatments are "hidden" within the function $Sort_e$ which takes a list of pairs (choice/weights) and returns a totally ordered list of choices.

4.3. The step function

An execution step is performed by the function $Step(e,t)$ taking an environment e and a trace expression t. It returns an *action* which is either

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

Definition 2. $\mathcal{A}$ denotes the set of actions and $\mathcal{X}$ denotes the set of termination flags.

4.4. The recursive step function

The step function $Step(e,t)$ is defined via a recursive function $S_e(t,g,s)$, where the parameters g and s are continuation functions returning actions.

(i) $g : \mathcal{C} \times \mathcal{T} \mapsto \mathcal{A}$ is the *goto* function defining how a local transition should be treated according to the calling context.
(ii) $s : \mathcal{X} \mapsto \mathcal{A}$ is the *stop* function defining how a local termination should be treated according to the calling context.

At the top-level, $\mathcal{S}_e$ is called with the trivial continuations,

$$\text{Step}\,(e,t) = \mathcal{S}_e(t,g,s) \text{ with } g(c,\nu) = \xrightarrow{c} \nu, \quad s(x) = \overset{x}{\hookrightarrow} \tag{22}$$

Basic traces

The empty behavior raises the termination flag in the current context. A raise statement terminates with the corresponding flag. At last, a constraint generates a goto or raises a deadlock depending on its satisfiability;

$$\begin{aligned} \mathcal{S}_e(\varepsilon, g, s) &= s(\varepsilon) \\ \mathcal{S}_e(\overset{x}{\hookrightarrow}, g, s) &= s(x) \\ \mathcal{S}_e(c, g, s) &= \text{if } (e \models c) \text{ then } g(c, \varepsilon) \text{ else } s(\delta). \end{aligned} \tag{23}$$

Sequence

The rule is straightforward;

$$\begin{aligned} &\mathcal{S}_e(t \cdot t', g, s) = S_e(t, g', s'), \\ &\text{where } g'(c, n) = g(c, n \cdot t') \\ &\qquad s'(x) = \text{if } x = \varepsilon \text{ then } \mathcal{S}_e(t', g, s) \text{ else } s(x). \end{aligned} \tag{24}$$

Priority choice

We only give the definition of the binary choice, since the operator is right-associative. This rule formalizes the reactivity principle. All possibilities in t must have failed before t' is taken into account,

$$\begin{aligned} \mathcal{S}_e(t \succ t', g, s) = \text{if } (r \neq \overset{\delta}{\hookrightarrow}) \text{ then } r \text{ else } \mathcal{S}_e(t', g, s), \\ \text{where } r = \mathcal{S}_e(t, g, s). \end{aligned} \tag{25}$$

Empty filter and priority loop

The empty filter intercepts the termination of t and replaces by a deadlock,

$$\begin{aligned} S_e(t \setminus \varepsilon, g, s) = \mathcal{S}_e(t, g, s'), s \quad \text{where } s'(x) = \text{if } (x = \varepsilon) \\ \text{then } s(\delta) \text{ else } s(x). \end{aligned} \tag{26}$$

The semantics of the loop results from the equivalence

$$t^* \Longleftrightarrow (t \setminus \varepsilon) \cdot t^* \succ \varepsilon. \tag{27}$$

Catch

This internal operator ($[n \overset{z}{\hookrightarrow} t]$) covers the cases of `try` ($z = \delta$) and `catch` (z is a user-defined exception)

$$\begin{aligned} &\mathcal{S}_e([t \overset{z}{\hookrightarrow} t'], g, s) = \mathcal{S}_e(t, g', s'), \\ &\quad \text{where } g'(c, n) = g(c, [n \overset{z}{\hookrightarrow} t']) \\ &\qquad s'(x) = \text{if } (x = z) \text{ then } \mathcal{S}_e(t', g, s) \text{ else } s(x). \end{aligned} \tag{28}$$

Parallel composition

We only give the definition of the binary case, since the operator is right-associative;

$$\begin{aligned} &\mathcal{S}_e(t \,\&\, t', g, s) = \mathcal{S}_e(t, g', s'), \\ &\quad \text{where } s'(x) = \text{if } (x = \varepsilon) \text{ then } \mathcal{S}_e(t', g, s) \text{ else } s(x), \\ &\qquad g'(c, n) = S_e(t', g'', s''), \\ &\quad \text{where } s''(x) = \text{if } (x = \varepsilon) \text{ then } g(c, n) \text{ else } s(x) \\ &\qquad g''(c', n') = g(c \wedge c', n \,\&\, n'). \end{aligned} \tag{29}$$

Weighted choice

The evaluation of the weights and the (random) total ordering of the branches are both performed by the function Sort_e (cf., Section 4.2).

$$\begin{aligned} &\text{If } \text{Sort}_e(\{t_i/w_i\}_{i=1\cdots n}) = \varnothing : \mathcal{S}_e(|_{i=1}^{n}\ t_i/w_i, g, s) = s(\delta) \\ &\quad \text{otherwise, } \mathcal{S}_e(|_{i=1}^{n}\ t_i/w_i, g, s) = \mathcal{S}_e(\succ Sort_e(t_1/w_1, \ldots, \\ &\quad t_n/w_n), g, s). \end{aligned} \tag{30}$$

Random loop

We recall that this construct is labelled by two weight functions (ω_c for continue, ω_s for stop) and by the current number of already performed iterations (i). The weight functions are evaluated for i and the statement is then equivalent to a binary weighted choice,

$$t_i^{(\omega_c, \omega_s)} \Longleftrightarrow (t \setminus \varepsilon) \cdot t_{i+1}^{(\omega_c, \omega_s)} / \omega_c(i) \mid \varepsilon/\omega_s(i), \tag{31}$$

Note that, the semantics follows the well-founded loop principle.

4.5. A complete execution

Solving a constraint

The main role of the environment is to store the values of uncontrollable variables; it is a pair of stores "past values, input values." For such an environment $e = (\rho, \iota)$ and a satisfiable constraint c, we suppose given a procedure able to produce a particular solution of c : $Solve_{\rho,\iota}(c) = \gamma$ (where γ is a store of controllable variables). We keep this *Solve* function abstract, since it may vary from one implementation to another (see Section 5).

Execution algorithm

A complete run is defined according to

(i) a given sequence of input stores $\iota_0, \iota_1, \ldots, \iota_n$,
(ii) an initial (main) trace t_0,
(iii) an initial previous store (values of `pre` variables) ρ_0.

A run produces a sequence of (controllable variables) stores $\gamma_1, \gamma_2, \ldots, \gamma_k$, where $k \leq n$. For defining this output sequence, we use intermediate sequences of traces $(t_1, \ldots, t_{k+1})$, previous stores $(\rho_1, \ldots, \rho_k)$, environments $(e_0, \ldots, e_k)$, and constraints $(c_0, \ldots, c_k)$. The relation between those sequences are listed below, for all step $j = 0 \cdots k$:

(i) the current environment is made of previous and input values, $e_j = (\rho_j, \iota_j)$,
(ii) the current trace makes a transition, $e_j : t_j \xrightarrow{c_j} t_{j+1}$,
(iii) a solution of the constraint is elected, $\gamma_j = Solve_{e_j}(c_j)$,
(iv) the previous store for the next step is the union of current inputs/outputs: $\rho_{j+1} = (\iota_j \oplus \gamma_j)$.

At the end, we have

(i) either $k = n$, which means that the execution has run to completion,
(ii) or $(\rho_{k+1}, \iota_{k+1}) : t_{k+1} \overset{x}{\hookrightarrow}$ which means that it has been aborted.

5. IMPLEMENTATION

A prototype has been developed in Ocaml. The constraint generator strictly implements the operational semantics presented in the previous section. The tool can do the following.

(i) 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.
(ii) Compile Lutin programs into the internal format of the testing tool Lurette. This format, called Lucky, is based on flat, explicit automata [14]. In this case, Lutin serves as a high-level language for designing test scenarios.

5.1. 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.

A required characteristic of such a solver is that it must provide a constructive, complete decision procedure; methods that can fail and/or that are not able to exhibit a particular solution are clearly not suitable. Basically, a constraint solver should provide the following.

(i) A syntactic analyzer for checking if the constraints are supported by the solver (e.g., linear arithmetics); this is necessary because the language syntax allows to write arbitrary constraints.
(ii) A decision procedure for the class of constraints accepted by the checker.
(iii) A precise definition of the election procedure which selects a particular solution (e.g., in terms of fairness).

Even with those restrictions, there is no obvious best solver as follows.

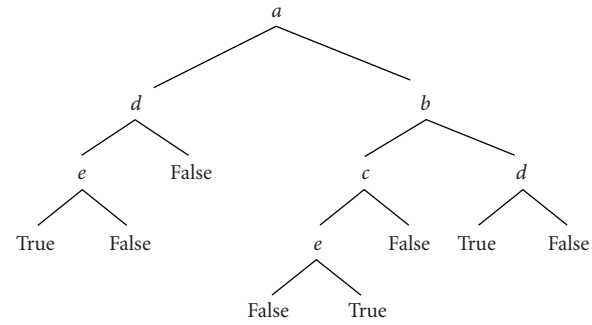

Figure 4: A BDD containing 10 solutions (ade, $\bar{a}bc\bar{e}$, and $\bar{a}\bar{b}d$).

(i) It may be efficient, but limited in terms of capabilities.
(ii) 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.

The solver that is currently used is presented in the next section.

5.2. The Boolean/linear constraint solver

Actually, we use the solver [15] that have been developed for the testing tool Lurette [16, 17]. This solver is quite powerful, since it covers Boolean algebra and linear arithmetics. Concretely, constraints are solved by generating a normalized representation mixing binary decision diagrams and convex polyhedra. This constraint solver is sketched below and fully described in [15]

First of all, each atomic numeric constraint (e.g., $x + y > 1$) is replaced by a fresh Boolean variable. Then, the resulting constraint is translated into a BDD. Figure 4 shows a graphical representation of a BDD; *then* (resp., *else*) branches are represented at the left-hand-side (resp., right-hand-side) of the tree. This BDD contains 3 paths to the true leaf: ade, $\bar{a}bc\bar{e}$, and $\bar{a}\bar{b}d$. When we say that the monomial (conjunction of literals) $\bar{a}bc\bar{e}$ is a solution of the formula. It means that variables a and e should be false; variables b and c should be true; and variable d can be either true or false. The monomial $\bar{a}bc\bar{e}$, therefore, represents two solutions, whereas ade and $\bar{a}\bar{b}d$ represents 4 solutions each, since 2 variables are left unconstrained.

In Figure 4 and in the following, for the sake of simplicity, we draw trees instead of DAGs. The key reason why BDDs work well in practice is that in their implementations, common subtrees are shared. For example, only one node "true" would be necessary in that graph. Anyway, the algorithms work on DAGs the same way as they work on trees.

Random choice of Boolean values

The first step consists in selecting a Boolean solution. Once the constraint has been translated into a BDD, we have a (hopefully compact) representation of the set of solutions.

We first need to randomly choose a path into the BDD that leads to a true leaf. But if we naively performed a fair toss at each branch of the BDD during this traversal, we would be very unfair. Indeed, consider the BDD of Figure 4; the monomial ade has 50% of chances to be tried, whereas $\bar{a}bc\bar{e}$ and $\bar{a}\bar{b}d$ have 25% each. One can easily imagine situation, where the situation is even worse. This is the reason why counting the solutions before drawing them is necessary.

Once each branch of the BDD is decorated with its solution number performing a fair choice among Boolean solutions is straightforward.

Random choice of numeric values

From the BDD point of view, numeric constraints are just Boolean variables. Therefore, we have to know if the obtained set of atomic numeric constraints is satisfiable. For that purpose, we use a convex polyhedron library [18].

However, a solution from the logical variables point of view may lead to an empty set of solutions for numeric variables . In order to chose a Boolean monomial that is valid with respect to numerics, a (inefficient) method would be to select at random a path in the BDD until that selection corresponds to a satisfiable problem for the numeric constraints. The actual algorithm is more sophisticated [15], but the resulting solution is the same.

When there are solutions to the set of numeric constraints, the convex polyhedron library returns a set of generators (the vertices of the polyhedron representing the set of solutions). Using those generators, it is quite easy to choose point inside (or more interestingly, at edges or at vertices) the polyhedron.

Using polyhedra is very powerful, but also very costly. However the solver benefits from several years of experimentation and optimizations (partitioning, switch from polyhedra to intervals, whenever it is possible).

5.3. Notes on predefined loop profiles

In the operational semantics, loops with iteration profile are translated into binary weighted choices. Those weights are dynamic; they depend on the number of (already) performed iterations k.

Interval loops

For the "interval" profile, those weights functions are formally defined and thus, they could take place in the reference semantics of the language. For a given pair of integers (min, max) such that $0 \leq \min \leq \max$ and a number k of already performed iterations, we have the following:

(i) if $k < \min$, then $\omega_s(k) = 0$ and $\omega_c(k) = 1$ (loop is mandatory);

(ii) if $k \geq \max$, then $\omega_s(k) = 1$ and $\omega_c(k) = 0$ (stop is mandatory);

(iii) if $\min \leq k < \max$, then $\omega_s(k) = 1$ and $\omega_c(k) = 1 + \max - k$.

Average loops

There is no obvious solution for implementing the "average" profile in terms of weightss. A more or less sophisticated (and accurate) solution should be retained, depending on the expected precision.

In the actual implementation, for an average value av and a standard variation sv, we use a relatively simple approximation as follows.

(i) First of all, the underlying discrete repartition law is approximated by a continuous (Gaussian) law. As a consequence, the result will not be accurate if av is too close to 0 and/or if st is too big comparing to av. Concretely, we must have $10 < 4 * sv < av$.

(ii) It is well known that no algebraic definition for the Gaussian repartition function exists. This function is then classically approximated by using an interpolation table (512 samples with a fixed precision of 4 digits).

6. 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 strictly implements the operational semantics 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, hierarchic automsata.

REFERENCES

[1] P. Raymond and Y. Roux, "Describing non-deterministic reactive systems by means of regular expressions," in *Proceedings of the 1st Workshop on Synchronous Languages, Applications and Programming (SLAP '02)*, Grenoble, France, April 2002.

[2] A. Benveniste and G. Berry, "The synchronous approach to reactive and real-time systems," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[3] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[4] G. Berry and G. Gonthier, "The Esterel synchronous programming language: design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[5] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[6] T. Gauthier, P. Le Guernic, and L. Besnard, "Signal: a declarative language for synchronous programming of real-time systems," in *Proceedings of the 3rd Conference on Functional*

Programming Languages and Computer Architecture, vol. 274 of *Lecture Notes in Computer Science*, pp. 257–277, Springer, Portland, Ore, USA, September 1987.

[7] N. A. Lynch and M. R. Tuttle, "An introduction to Input/Output automata," *CWI-Quarterly*, vol. 2, no. 3, pp. 219–246, 1989.

[8] S.-H. Wu, S. A. Smolka, and E. W. Stark, "Composition and behaviors of probabilistic I/O automata," *Theoretical Computer Science*, vol. 176, no. 1-2, pp. 1–38, 1997.

D)5(M)-0.2441([8]TM)99)75a0,B75"-2(.9T)11rie429l(nce)o01455839 -1.1664 TD -0.0001 Tc.13Sp1Orr359.9344eai9.93452rarM1-093452ar"