Describing and Executing Random Reactive Systems

Pascal Raymond, Erwan Jahier, Yvan Roux

To cite this version:

Pascal Raymond, Erwan Jahier, Yvan Roux. Describing and Executing Random Reactive Systems. 4th IEEE International Conference on Software Engineering and Formal Methods, Sep 2006, Pune, India. pp.216 - 225, 10.1109/SEFM.2006.15 . hal-01465776

HAL Id: hal-01465776

https://hal.archives-ouvertes.fr/hal-01465776

Submitted on 13 Feb 2017

Describing and executing random reactive systems

Pascal Raymond, Erwan Jahier, Yvan Roux
VERIMAG (CNRS, UJF, INPG)
Grenoble, France
URL: http://www-verimag.imag.fr

E-mail: {Pascal.Raymond,Erwan.Jahier,Yvan.Roux}@imag.fr

Abstract

We propose a operational model for describing non-deterministic reactive systems, together with a mechanism for expressing probability issues. Some models have already been proposed for this purposes, but they are generally intended to allow global reasoning on systems (e.g. stockastic analysis, formal proofs). Our goal is somehow less ambitious, sinve we are mainly interested in executing such models, which can be useful for testing, prototyping. On the other hand, the proposed model is not strongly restricted because of decidability concerns, so it is likely to be more expressive: in particular, we are not restricted to finite-state models.

The proposed model is rather general: systems are described as implicit state/transition machines, possibly infinite, where probabilities are expressed by means of relative weights. The model itself is more an abstract machine than a programming language. The idea is then to propose high-level, user-friendly languages that can be compiled into the model. We present such a language, based on regular expressions, together with its translation into the model.

1 Introduction

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

There are several approaches to non-determinism: the purely qualitative approach only focuses on *possibles behaviors*, while the stochastic approach also takes into account probabilistic information.

There exist many formalisms to express non-determinism together with probability issues. Some of them are based on classical finite automata [5], and thus, the underlying mathematical model is the one of Markov Chains. The more specific model of I/O automata [12] is extended with probabilistic features in [19]. PCTL [10] is an example of temporal logic extended with probabilities. From a more operational point of view, we can cite stochastic extensions of process algebras [11, 3], or Signalea [2], an extension of the synchronous language Signal.

Those formalisms are mainly designed to allow global analysis of systems: formal proofs, model checking, probabilistic analysis. As a consequence, their expressive power is limited to decidable models (typically finite state machines).

Our approach is less ambitious, since we only focus on *simulation*. More precisely, we do not care if the model is globally undecidable, as long as it can be efficiently simulated.

In other terms, our goal is more to *program* stochastic reactive systems, than to reason about them. Concretely, we separate two problems: the definition of high-level, user-friendly programming languages, and the definition of general abstract machines into which those high-level languages can be compiled. In the paper, we first present the abstract machines, Weight-labelled Transition System (*WTS*), and then a high-level language based on regular expressions, together with its translation to WTS.

2 Weight-labelled Transition Systems

2.1 Overview

We propose a model where a basic qualitative model describing a set of behaviors is extended with a probabilistic mechanism. The main features of this model are presented here.

Symbolic state/transition systems. The basic qualitative model consists in a very general state/transition system, characterized by:

- a memory: a finite set of variables with no special restrictions on their domains (to simplify, we will consider here just Boolean, integer and rational values);
- an interface: variables are declared as inputs, outputs, or locals;
- a finite control structure: an interpreted finite automaton, whose transitions are representing reactions of the machine.

A global state of the system is then a pair made of the current control point in the automaton (the *control-state*), and a current valuation of its memory (the *data-state*). Therefore the set of global states is potentially infinite.

Synchronous relations. We adopt the synchronous approach for the reactions: all values in the memory are changing simultaneously when a reaction is performed. The previous value of the memory corresponds to the source data-state, and the current value to the next data-state. The transitions are labelled with information denoting what are the possible values of the current memory depending on the current data-state. This information is quite general: it is a *relation* between the past and current values of the variables. In particular, no distinction is made between uncontrollable and controllable variables. Performing a reaction will consist in finding solutions to such formula. This problem induces a restriction: we suppose that, once reduced according to the past and input values, the constraints are solvable by some actual procedure[1].

Weights instead of probabilistic distribution. The problem of adding probabilisties to state/transitions systems has been largely studied [14, 6]. Some variants exist, depending on how the choice of the action (in our case the data-state) and the choice of the next control-state are related. According to the terminology of [6], our approach is close to *Generative Models*, where the probabilistic mechanism includes both the action and the next-state choice.

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

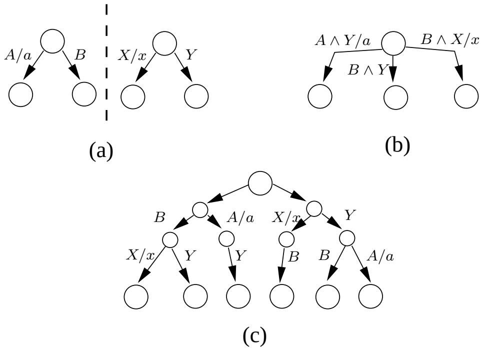
The main idea is to keep the distinction between the probabilistic information and the constraint information. Since constraints are influencing probabilities (zero or non-zero), this information does not express the probability to be drawn, but pragmatically, the probability to be *tried*. In order to emphasize the difference, we do not use distributions (i.e., set of positive values the sum of which is 1) but *relative weights*. A relative weight is a positive rational value, not necessarily less than one, the meaning of which is only defined relatively to another weight: if two possible reactions (i.e., the corresponding constraints are both satisfiable) are labelled respectively with the weights w and w', then the probability to perform the former is w/w' times the probability to perform the latter.

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

Transient states. For the time being, we have only one notion of state: a state is a stable control point, and a transition between two states defines an atomic reaction. However, we think it may be convenient to introduce the notion of *transient state*, and, as a consequence a notion of micro-step: a complete reaction is then a sequence of transitions between two stable states, where all the intermediate states are transient. Transient states do not affect the synchronous interpretation of the variable changes: intuitively, if we abstract probabilities, a reaction $q \xrightarrow{f} t \xrightarrow{g} q'$, is qualitatively equivalent to $q \xrightarrow{f \wedge g}$. In contrast, transient states affect probabilities, and may be helpful to express complex conditional relative weights.

Global concurrency. Concurrency (i.e., parallel execution) is a central paradigm for reactive systems. The problem of merging sequential and parallel constructs has been largely studied: classical solutions are hierarchical automata ”à la StateCharts” [13, 1], or statement-based languages like Esterel [4]. Our opinion is that deeply merg-

[1] concretely, we have developed a constraint solver for mixed Boolean/linear constraints.



(a) (b)

(c)

Figure 1. Weights and parallelism: the parallel comnposition (1a), and, assuming that $A \wedge X$ is unfeasible, the product solution (1b) and the arbiter solution (1c).

ing sequence and parallelism is a problem of high-level language, and that it is sufficient to have a notion of global parallelism: intuitively, local parallelism can always be made global by adding extra idle states. As a consequence, concurrency is a top level notion in our model: a complete system is a set of concurrent automata, each one producing its own constraints on the resulting global behavior.

Weights and parallelism. In terms of control structures, parallelism corresponds to a kind of synchronous product of automata. Transient states make this ”product” more complex than a simple Cartesian product, but do not involve big difficulties. For formulas, the product is simply the logical and. Unfortunately, there is no obvious way for combining stochastic information: as they are defined, they are only local information and they may induce paradoxes when combined into a parallel composition. A simple example is shown in Figure 1a: the first automaton (resp. the second) has the choice between the constraints A or B (resp. X or Y) both satisfiable. In the first automaton, the choice of A has a big weight $a >> 1$ compared to B (1 by default), and in the second, X has a big weight $x >> 1$ comparing to Y. Suppose that the data-state makes it impossible to satisfy $A \wedge X$, it follows that it is impossible to satisfy the stochastic demand of both components. There is mainly two ways to solve the problem:

- Consider that weights are not only local information, but are also influencing the parallel composition: for instance, if a is much bigger than x, that means that the stochastic demand of the first component is much stronger than the one of the second. The simplest way to implement this notion is to combine weights with multiplication, as shown in Figure 1b.
- The problem is treated at the parallel composition level, where some indications are added to express priority for satisfying stochastic demands. Intuitively, the components of a parallel composition are treated sequentially: the first one is perfectly served, according to its own local weights, then the second is served according to what was decided by the first one, etc. The order of components is, in general non-deterministic, and stochastic information may be added to influence it. The Figure 1c shows a product where a first fair choice is made to decide which component will ”play” first (note that all intermediate states are transient).

There is no obvious argument to prefer one solution to another: each are consistent, and none is clearly more natural than the other. As a consequence, both are implemented and the user can choose between them.

2.2 Details

2.2.1 Data structures

Variables. WTS variables are either *input*, *output*, *local*, or *previous*: $V = V_i \uplus V_o \uplus V_l \uplus V_p$. The previous variables are meant to refer to previous values of the other variables in $V_{\setminus p} = V_i \uplus V_o \uplus V_l$. Each previous variable is denoted by $\bullet v$ where $v \in V_{\setminus p}$. Moreover, each variable in $V_{\setminus p}$ is defined with a default value: the value just before the first reaction. Local variables can be seen as output variables that are hidden from the outside. The typing function, $Type : V \rightarrow \{\mathbb{B}, \mathbb{Z}, \mathbb{Q}\}$ associates a definition domain with each variable: Boolean, integer or rational; obviously, $Type(v) = Type(\bullet v)$.

For the sake of (formula) conciseness, we also introduce a concept of *mode* attached to controllable variables: in the *stable mode*, unconstrained variables are bound to their previous value; in the *unstable mode*, unconstrained variables are drawn from their definition domain (this is to avoid boring repetitions of $v = \bullet v$ statements).

Valuations. A *valuation* is a mapping from variables to values. A *data-context* is a pair (σ_i, σ_p), where σ_i (input valuation) associates a value with each input, and σ_p (previous valuation) associates a value with each previous variable. Previous valuations are also called *data-state*. In particular, the default values of variables are defining the *initial data-state*, denoted by σ_p^0.

Formula. A formula is any well-typed Boolean expression made of variables, constants, and classic logical and numerical operators ($\neg$,$\wedge$,$\vee$,$=$,$>$,$<$,$\geq$,$\leq$,$+$,$-$,$*$,$/$). We note $\mathcal{F}$ the set of well-typed formula.

Given a data-context, any formula can be reduced to a formula over the controllable variables ($V_o \uplus V_l$). The notation $(\sigma_i, \sigma_p) : (\sigma_o, \sigma_l) \models f$ states that, given a context (σ_i, σ_p), the controllable valuation (σ_o, σ_l) satisfies f.

2.2.2 Control structure

At the top level, the behavior of a system is described by a non-empty set of concurrent WTS sharing the same variables. Each WTS is an interpreted automaton, where transitions are labelled by qualitative and stochastic constraints, as presented in the sequel.

Nodes. The set of *control-states*, is divided into *stable* states and *transient* states: $Q = Q_s \uplus Q_t$. The initial control state is a particular stable state $q^0 \in Q_s$.

Weights. Weights are positive numerical functions of the uncontrollable variables: $\mathcal{W} : \Sigma_i \times \Sigma_p \rightarrow \mathbb{Q} \cup \{\infty\}$. More concretely, they are given as numerical expressions made of inputs, previous variables, classical operators or ad hoc computable functions. The ∞ value is introduced to express a sound notion of mandatory choice: a transition with the infinite weight has priority on any finite weighted transition. There is a single notion of infinite weight: two feasible transitions with infinite weight have the same probability. In order to express relative probabilities between mandatory choices, it is necessary to detail the control structure by introducing transient states.

Transitions. The set of transitions is a relation: $T \subseteq Q \times \mathcal{F} \times \mathcal{W} \times Q$, and we note $q \xrightarrow[w]{f} q' \in T$ a transition from q to q' labelled by the formula f and the weight w.

Transitional loops. We do not try to give sense to infinite loops of transient states: models containing such combinational loops are statically rejected.

2.3 Operational Semantics

The WTS are defined in such a way that their operational semantics is straightforward. In some sense, they are *executable* by definition. We give here the main lines of the simulation algorithm.

Product. First of all, the behavior which is in general expressed as a a set of concurrent automata, is semantically equivalent to the one of a single *product automaton*. Two different products are defined, depending on the semantics chosen for weights composition (Figure 1). The simplest one is almost a classical *synchronous product*:

- the global state space is the Cartesian product of the component state spaces; the global initial state is the tuple of initial states; a global state is stable if it is composed of stable states only.
- the definition of global transitions is almost a classical composition where constraints are combined with the $\wedge$ operator, and weights with the $*$ operator. The only problem is to enforce the synchronization on stable states; we note s_1, s_2 stable states, t_1, t_2 transient states and q_1, q_2 any states, so:
  - $(s_1, s_2) \xrightarrow[w_1 * w_2]{f_1 \wedge f_2} (q_1, q_2)$ iff $s_1 \xrightarrow[w_1]{f_1} (q_1)$ and $s_2 \xrightarrow[w_2]{f_2} (q_2)$
  - $(t_1, t_2) \xrightarrow[w_1 * w_2]{f_1 \wedge f_2} (q_1, q_2)$ iff $t_1 \xrightarrow[w_1]{f_1} (q_1)$ and $t_2 \xrightarrow[w_2]{f_2} (q_2)$
  - $(s_1, t_2) \xrightarrow[w_2]{f_2} (s_1, q_2)$ iff $t_2 \xrightarrow[w_2]{f_2} (q_2)$, and symmetrically for (t_1, s_2).

The second kind of ”product” (as shown in Figure 1c) is a little bit tricky: the idea is to introduce, for each component, an additional *starting* transient state $\hat{s}$ and an additional *waiting* transient state $\breve{s}$ for each stable state s. The global state space is then defined as the Cartesian product over the extended component state spaces. The definition of initial and stable global states does not change. The transitions are defined in such a way that each component performs its reaction in turn. We only give the rules where the first component starts, the other case is similar:

- from a global stable state, the first component may start while the other waits: $(s_1, s_2) \xrightarrow{} (\hat{s}_1, \breve{s}_2)$,
- the starting component performs its first transition: $(\hat{s}_1, \breve{s}_2) \xrightarrow[w_1]{f_1} (q_1, \breve{s}_2)$ iff $s_1 \xrightarrow[w_1]{f_1} q_1$,
- the starting component is not yet in a stable state, and it performs another transition: $(t_1, \breve{s}_2) \xrightarrow[w_1]{f_1} (q_1, \breve{s}_2)$ iff $t_1 \xrightarrow[w_1]{f_1} q_1$,
- the starting component has reached a stable state, and the waiting one starts its reaction: $(s_1, \breve{s}_2) \xrightarrow[w_2]{f_2} (s_1, q_2)$ iff $s_2 \xrightarrow[w_2]{f_2} q_2$,
- the second component performs transitions until it reaches a stable state: $(s_1, t_2) \xrightarrow[w_2]{f_2} (s_1, q_2)$ iff $t_2 \xrightarrow[w_2]{f_2} q_2$.

In the sequel, we suppose that we have a single automaton, obtained with one of the product operations defined above. Note that, in the concrete implementation, the global product is not statically built: local products are simply build *on the fly* to avoid space state explosion.

Execution. An *WTS execution*, according to a given input history $(\nu^n)_{n\geq 0}$ is a sequence of pairs made of a stable state and a valuation: $(s^n, \sigma^n)_{n\geq 0} \in \mathbb{N} \rightarrow Q_s \times \Sigma$, such that:

- s^0 is the initial control state and σ_p^0 is the default map of the variables,

and for each k:

- $\sigma_i^k = \nu^k$ (the valuation history meets the input history)
- $\sigma_{\setminus p}^k = \sigma_p^{k+1}$ (the current-part of the valuation meets the previous-part of the next valuation)
- $(s^k, \sigma_p^k, \sigma_i^k) \xrightarrow{\sigma_o^k, \sigma_l^k} s^{k+1}$ is a feasible, fair reaction according to the control structure. We detail in the sequel the algorithm for finding such a reaction.

Reaction. Intuitively, a step in an execution is done by drawing, according to weight directives and current values of uncontrollable variables, a path in the automaton from the current (stable) state to a next stable state. More formally, let $s = q_0$ be the current control-state, σ_p the current data-state and σ_i the current inputs. For all k, we note $\Theta_k = \{q_k \xrightarrow[w]{f} q\}$ the set of transitions leaving q_k, and we use the notation w_τ to denote the weight attached to a transition τ. According to the current data context, the sum of weights $W_k = \sum_{\tau \in \Theta_k} w_\tau(\sigma_p, \sigma_i)$ is a numerical constant. If there exist some transition τ from q_k to q_{k+1}, the probability to complete a path $(q_0, q_1, ..., q_k)$ with q_{k+1} is then: $w_\tau(\sigma_p, \sigma_i)/W_k$. This process is repeated until a stable node $q_n = s'$ is reached: $s \xrightarrow[w_1]{f_1} q_1 \xrightarrow[w_2]{f_2} q_2 \cdots q_{n-1} \xrightarrow[w_n]{f_n} s'$, where all $q_1, \cdots q_{n-1}$ are transient.

The conjunction of all formulas labelling the drawn path is the elected formula: $f = \bigwedge_{k=1}^{n} f_k$. We substitute in f input and previous variables ($f_{|\sigma_i \sigma_p}$), and solve it. A valuation of output and local variables for the current step is obtained by performing a fair toss among the solutions of that formula. If $f_{|\sigma_i \sigma_p}$ is unsatisfiable, another path is drawn. If no satisfiable path can be drawn, the machine stops.

As it is presented, this algorithm is quite ineffective, and, moreover it may diverge if no particular precaution is taken to avoid trying several times the same unfeasible path. In the actual implementation[2], unfeasible paths are detected as soon as possible, and they are marked to avoid divergence.

2.4 Example

Figure 2 shows a WTS that simulates the temperature in a room containing a heater and a window which is opened from times to times. Input variables are a Boolean On, which is true if the heater in on, and a rational T, which indicates the temperature outside the room. The only output variable is a rational t, indicating the temperature inside the room. Local variables are the rational δ, which is used to compute the new temperature, and the integer j, which is used to count the number of steps the window remains open. Previous variables are $\bullet t$ and $\bullet j$. Stable states are denoted by s_{on}, s_{off}, and s_{open}. The other unnamed states are transient. The initial node is s_{on}.

If the heater is initially on (resp. off), only 2 transitions are possible among the 3 output transitions of s_{on}, since the transition labelled by $\neg On$ (resp. On) is unsatisfiable. The first possible transition has weight 100, and the other one has weight 1.

- The first transition will therefore be drawn with a probability of $100/101$. It leads to a transient state which has only one output transition leading back to s_{on} (resp. s_{off}); the elected formula is therefore $0 < \delta < 0.2 \wedge t = \bullet t + \delta$. It states that the local variable δ will be uniformly drawn between 0 and 0.2 (resp. -0.1 and 0), and that δ is then used to increase the temperature. This is intended to model that, when the heater is on (resp. off), the temperature slightly increases (resp. decreases).

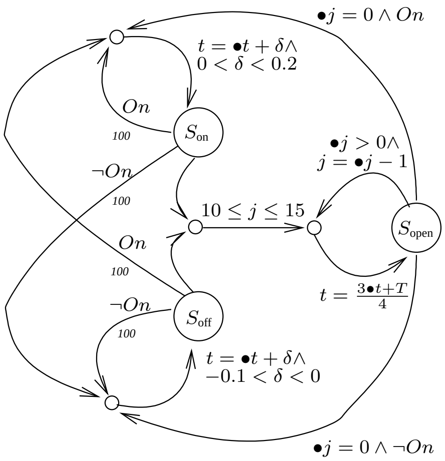
- The second transition will be drawn with a probability of $1/101$. After two transient states, the only stable node that is reachable is s_{open}, and the elected formula is therefore $true \wedge 10 \leq j \leq 15 \wedge t = \frac{3\bullet t + T}{4}$. It states that the local integer variable j is drawn uniformly between 10 and 15, and defines how the new temperature is computed. This is intended to model that, whenever the window is open, the temperature become closer from the temperature outside.

  For the next step, from s_{open}, 3 transitions are possible, but they are labelled by formulas that can not be true at the same time (they form a partition): as long as j is greater than 0, the window will remain open; j is decremented at each step, and when it reaches 0, the control gets back to either s_{on} or s_{off}, depending on the variable On.

3 A language for describing WTS

As it is defined, the WTS model is more an abstract machine model than a user-friendly description language. The idea is then to define high-level programming languages that can be compiled into WTS. A possible solution is to define a graphical language *à la StateCharts* [8, 1] based on the same notion of automata, but with high-level features like hierarchy and refinement.

[2] The tool, `lucky`, is available, at : `www-verimag.imag.fr/∼synchron/tools.html`.



$$V_i = \{On, T\}, V_o = \{t\}, V_l = \{\delta, j\}, V_p = \{\bullet t, \bullet j\},$$
$$Q_s = \{S_{on}, S_{off}, S_{open}\}, q_0 = S_{on}.$$

Figure 2. A WTS that simulates the temperature in a room with a heater and a window.

Another solution is to propose a different style for expressing the control. In [16], we have presented such a language, called Lutin, where the control structure is based on regular expressions instead of automata. We briefly present here the principles of this language, and define its compilation into WTS.

3.1 Overview of the language

Data. For the data aspects, the language is compliant with WTS: a system is declared with a set of variables (inputs, outputs, locals), and the constraints on atomic reaction are defined as relations between the previous and current values of those variables. Concretely, a constraint is a Boolean expression made of logical and numerical operators and references to variables, where the previous value of a variable `v` is denoted by `pre v`. Those Boolean expressions are called *formulas* in the sequel.

Basic control operators. In the body of a Lutin program, formulas (representing a single instant) are combined with temporal statements which are basically the regular operators:

- the sequence `s1 fby s2` states that the program first behaves as `s1`, and then, if and when `s1` stops, as `s2`,
- the n-ary non-deterministic choice {`s1 | ... | sn`} means that the system behaves as one of `s1`, ..., `sn`.
- the loop `loop s pool` means that the behavior `s` is repeated zero or more times. This operator introduce the important notion of *empty behavior* (zero loop) which is precisely studied in the sequel.

The actual language provides more statements. For instance, `assert f in s` means that the constraint `f` must remain satisfied while the system `s` is active.

Probabilistic issues. Non-determinism appears both on data (when choosing a particular solution satisfying a formula) and on control (choices and loops). Like in WTS, there is no feature to influence the selection of data: we suppose the existence of a constraint solver which guarantees a relative fairness in the choice of the solution. The only way to influence the probabilities is attached to the control structure, as explained below.

Weighted choices. For the non-deterministic choice, Lutin provides a mechanism of relative weights similar to the one of WTS: each case in a choice can be completed by a numerical information representing its relative chance to be chosen. For the time being, this weight is a constant, but this notion can be easily extended (like in WTS) to numerical expression over previous and input variables. The default weight is 1, hence, for example {`s1 weight 2 | s2`} means that, whenever both `s1` and `s2` are possible, the first one is chosen with a probability of $2/3$.

Weighted loops. For loops, the user can also use a weight to express how ”continue” is relatively more probable than ”stop” (the weight of ”stop” is 1 by definition): `loop weight 99 s pool` means that, whenever it is both possible to stop or to continue, continue is chosen at 99 percent.

A loop without any probabilistic information has a special meaning: it has implicitly an *infinite weight*, which means that the loop is performed as long as possible.

Iteration laws. Another interesting way of controlling loops is to reason in terms of iterations number. Unfortunately, with simple static weights, it is impossible to control the repartition of this number[3]. Describing complex repartitions is possible by using dynamic weights, depending typically on the number of already made iterations. However, this solution maybe quite complex, even for simple and intuitive repartition. This is why the language provides two predefined repartition mechanism: uniform interval loops and Gaussian average loops.

[3] The actual shape of the repartition induced by static weights is quite hard to define, since it depends on the fact that looping is sometimes mandatory or impossible, but roughly speaking, it follows a kind of truncated, decreasing exponential law.

For uniform loops, the user specifies minimum and maximum numbers of loops: `loop [100,150]` means that the number of iterations must be between 100 and 150, and that any particular value within this interval has the same probability than another one.

An example of Gaussian loop is `loop ~1000:200`, which means that the average number of iterations is 1000, with a standard deviation of 200.

Note that those definitions must be understood *modulo* the fact that looping is sometimes mandatory/impossible, depending on the constraints attached to the loop entry and the loop exit.

Empty behaviors and well founded loops. In terms of set of behaviors, those operators are exactly the classical ones, in particular the `loop` is equivalent to the Kleene's star operator. However, for simulation purpose, the classic Kleene's star rises problems: consider a statement `loop [100,150] s pool`, where `s` can generate the empty behavior. One one hand, this construct suggests that something must be made at least 100 times. On the other hand, since `s` can do nothing at each step, the loop statement can generate the empty behavior. In our opinion, this is counter-intuitive, and not well adapted for simulation. This is why we have adopted the notion of *well founded loops*: when the loop statement takes the control it can decide to generate the empty behavior, but, if the statement decides to loop, each iteration must generate a non-empty behavior.

In terms of language, our operator must be interpreted as ”$\varepsilon + (s \setminus \varepsilon)^+$”. Once again, this is *qualitatively* equivalent to the classic Kleene's star, and the difference only appears in simulation.

3.2 Example

Here is a simple example inspired by a heater, but slightly different from the one in Section 2.4: we do not take into account the opened window, but just focus on the heater behavior and its possible defaults. The system (`Temp`) is controlling the numerical value `t`, according to the commands `on` and `off`. It uses two local variables: `running` indicates the state of the system (increasing or decreasing), and `dt` represents the discrete slope of `t`.

Global invariants state that `dt` is the slope and it is comprised between -0.1 and 0.2, it is positive when the system is running and negative otherwise. The main regular expression expresses the temporal behavior. In the *initial state*, the system is not running and `t` is between 10 and 17. Then, the system enters the *working behaviour*, where, for an average number of 10000 iterations, it works (almost) correctly:

- it may obey a `on` command, which is probable (weight 100),
- it may obey a `off` command, which is as probable (weight 100),
- or it may ignore any command, which is less probable (implicit weight 1).

Then the system *breaks down* and stays on the not running mode forever (implicit infinite weight).

```
system Temp(on, off: bool)
returns(t: real);
var running : bool; dt : real;
let Temp =
  (* GLOBAL INVARIANTS *)
  assert (dt = (t - pre t))
  and (dt > -0.1) and (dt < 0.2)
  and (running => (dt > 0.0))
  and (not running => (dt < 0.0))
  in
  (* INITIAL STATE *)
  (not running and
  t > 10.0 and t < 17.0)
  fby
  (* WORKING BEHAVIOR *)
  loop ~10000:1000 {
    (on and running) weight 100
  | (off and not running) weight 100
  | (running = pre running)
  } pool
  fby
  (* BREAKDOWN BEHAVIOR *)
  loop { not running } pool
tel
```

3.3 Compilation

We present the translation of a Lutin description into WTS. We consider only the core language, and use the following abstract syntax: a Lutin regular expression is either a formula f, a sequence $E_1 \cdot E_2$, a weighted choice $(E_1 : w_1 + \cdots + E_n : w_n)$ or a well-founded loop $E^\diamond$, where the $\diamond$-power is either a single weight w, a uniform law $[l, h]$ or a Gaussian law $\sim m : s$.

The classic approach. Translating regular expressions into non-deterministic automata is quite a classical problem. The most elegant and efficient algorithms are based on the Thompson's idea [18] which consists in using ε-transitions (or equivalently transient states). With this trick, the translation is fully modular and has a linear cost with respect to the size of the regular expression. The counterpart is that those ε-transitions may introduce *combinational cycles*, when some sub-expression of the form E^* is such that E contains the empty string.

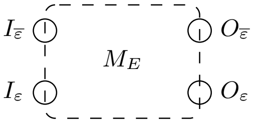
One solution to avoid such combinational cycles is to perform a pre-processing on the regular expression which replaces all sub-expressions of the form E^* by some equivalent F^* such that $\varepsilon \notin F$ [15]. Unfortunately this kind of solution cannot be used in our case: it strongly modifies the structure of the expression, which is incompatible with the probabilistic mechanism.

Hence, we propose a adapted version of the classic algorithm which separates actual paths (taking time) from instantaneous paths: intuitively, this algorithm guarantees by construction that any cycle contains an actual path.

Principles of the modular translation. The classic algorithm recursively associates with each sub-expression E a sub-automaton M_E with a single initial state and a single final state.

The semantics is classically defined in terms of language recognizers, but the adaptation to generators is straightforward; the semantics is even simpler since, in a generator, only one state can have the control. The classic generative semantics is the following: the control reaches the final state if and only if a behavior in E (empty or not) has been generated since the control has been passed to the initial state.

We slightly modify this principle in order to separate the generation of empty and non-empty behaviors: the sub-automaton M_E associated to E has two initial states, and two final states, as shown in the figure beside.



Intuitively, this machine is intended to be used in a prefix context which gives the control to M_E via one of its initial states. If the control is given via $I_{\overline{\varepsilon}}$, it means that M_E must consider that the prefix has already generated a non-empty behavior. If the control is given via I_{ε}, it means that the prefix has not yet generated a non-empty behavior. According to this information (and, of course, the definition of E), M_E will pass the control to $O_{\overline{\varepsilon}}$ if itself or its prefix-context have generated a non-empty behavior, and it will pass the control to O_{ε} if neither itself or the context have generated a non-empty behavior.

The key point of this intuitive definition, it that it guarantees that any path leading from the I_{ε} to $O_{\overline{\varepsilon}}$ state is necessarily a non-instantaneous path. The soundness of the translation will be based on the fact that any cycle must traverse such a path.

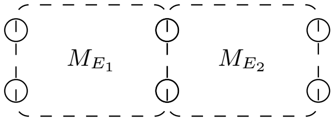
The main inductive property. The intuitive semantics can be formalized in term of languages. We identify each state with its corresponding control language, and we note ε the language reduced to the empty behavior:

$$(1) \quad O_{\overline{\varepsilon}} = I_{\overline{\varepsilon}} \cdot E \cup I_{\varepsilon} \cdot (E \setminus \varepsilon) \qquad (2) \quad O_{\varepsilon} = I_{\varepsilon} \cdot (E \cap \varepsilon)$$

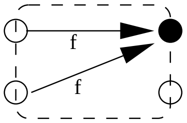
We will demonstrate, when it is not trivial, that the translation inductively satisfies those properties.

In order to simplify the drawings, we always use the same topology for the states: top-left for $I_{\overline{\varepsilon}}$, bottom-left for I_{ε}, top-right for $O_{\overline{\varepsilon}}$ and bottom-right for O_{ε}. Moreover, we put weights under the transitions (default is 1), and constraints over the transitions (default is *true*).

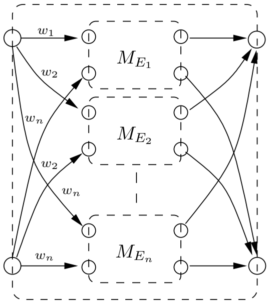
Sequence. The translation of the sequence is the simplest one: the automaton for $E_1 \cdot E_2$ is obtained by confounding pairwise their outputs/inputs states. The inductive properties are trivially preserved.



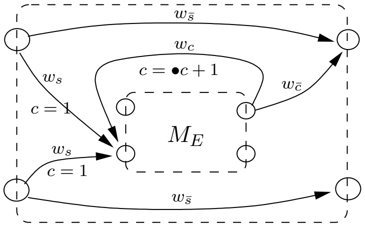
Formula. The formula is the only expression which takes time: whatever be the entry point, $O_{\overline{\varepsilon}}$ is reached after the generation of an atomic behavior satisfying the formula (if such a behavior exists). Note that $O_{\overline{\varepsilon}}$ is stable (filled circle), and that O_{ε} is unreachable.



Choice. The union operator trivially distributes on the desired inductive properties. Moreover, the weight semantics of Lutin is exactly the same as the one of WTS. The schematic translation of $E_1 : w_1 + ... + E_n : w_n$ is presented beside.



Loop. We consider the most general case of well-founded loops, where dynamic weights are necessary. The idea is to introduce a new local variable c in the memory of the WTS. This variable is constrained in such a way that, after each iteration, $\bullet c$ contains exactly the number of already performed iterations. This variable is defined as stable (Section 2.2.1), which means that it keeps its value when it is not constrained.



We also have to introduce two pairs of weights. The first pair is for the very first choice: the loop takes the control and must chose between no iteration ($w_{\overline{s}}$) or at least one iteration (w_s). The second pair is used when at least one iteration has been made: the choice is between performing another iteration (w_c) or not ($w_{\overline{c}}$).

The inductive property (1) is less trivial here, so here are some hints of the proof. We note I^E_{ε}, $I^E_{\overline{\varepsilon}}$ the two input states of the sub-automaton M_E, and $O^E_{\overline{\varepsilon}}$ its top-output state. The outermost states are denoted as usual. We have by construction $I^E_{\overline{\varepsilon}} = \emptyset$ (unreachable), hence, by induction on property (1): $O^E_{\overline{\varepsilon}} = I^E_{\varepsilon} \cdot (E \setminus \varepsilon)$. By construction we also have: $I^E_{\varepsilon} = O^E_{\overline{\varepsilon}} \cup I_{\overline{\varepsilon}} \cup I_{\varepsilon}$, so, $O^E_{\overline{\varepsilon}}$ satisfies a recursive equation whose solution is:

$$O^E_{\overline{\varepsilon}} = (I_{\overline{\varepsilon}} \cup I_{\varepsilon}) \cdot (E \setminus \varepsilon)^+ = (I_{\overline{\varepsilon}} \cup I_{\varepsilon}) \cdot (E^* \setminus \varepsilon)$$

by construction we have $O_{\overline{\varepsilon}} = I_{\overline{\varepsilon}} \cup O^E_{\overline{\varepsilon}}$ (transitions leading to the outermost top-right state), hence:

$$\begin{aligned} O_{\overline{\varepsilon}} &= I_{\overline{\varepsilon}} \cup ((I_{\overline{\varepsilon}} \cup I_{\varepsilon}) \cdot (E^* \setminus \varepsilon)) \\ &= I_{\overline{\varepsilon}} \cdot (\varepsilon \cup (E^* \setminus \varepsilon)) \cup I_{\varepsilon} \cdot (E^* \setminus \varepsilon) \\ &= I_{\overline{\varepsilon}} \cdot E^* \cup I_{\varepsilon} \cdot (E^* \setminus \varepsilon) \end{aligned}$$

At last, let us define the weights involved in the translation, depending on the probabilistic profile:

- Weighted loop characterized by w: the local variable c is useless. We simply have: $w_s = w_c = w$ and $w_{\overline{s}} = w_{\overline{c}} = 1$.

- Uniform law characterized by $[l, h]$: the initial case depends on the lower bound. If $l = 0$, then performing zero loop, is, by definition, as probable as performing any other value in the interval: $w_{\overline{s}} = 1$ and $w_s = h$. If $l \neq 0$, performing no loop is forbidden: $w_{\overline{s}} = 0$ and $w_s = 1$.

  The other weights are used when $\bullet c \geq 1$; as long as $\bullet c$ is less than l, looping is mandatory; when $\bullet c$ is within l and h, the probability to stop is 1 over $h - l$; at last once h is reached, the iterations must stop.

	$\bullet c < l$	$l \leq \bullet c \leq h$	$h < \bullet c$
$w_{\overline{c}}$	0	1	1
w_c	1	$h - \bullet c$	0

- Gaussian repartition characterized by $\sim m : s$: any number of iteration is (virtually) feasible. The weights are always computed in terms of the function $P_{m,s}(n)$ which gives the probability for a Gaussian variable to be greater than n. It is well know that this function (bell curve integral) has no analytic form. We classically use a procedure which approximate this integral via a table of sampled values. With such a procedure $G(m, s, n)$, we have: $w_s = G(m, s, 0)$ and $w_{\overline{s}} = 1 - G(m, s, 0)$
  $w_c = G(m, s, \bullet c)$ and $w_{\overline{c}} = 1 - G(m, s, \bullet c)$

4 Conclusion

Our aim was to propose a way to *program and simulate* non-deterministic, stochastic reactive systems.

We have defined an operational model (WTS), with the main objective to have a compromise between a good expressive power to describe lots of realistic behaviors, and a straightforward operational semantics allowing efficient simulation. We have also shown how such an abstract machine model can be targeted by an higher-level programming language, Lutin, which is based on regular operators.

Both the low-level abstract machine simulator (`lucky`) and the high-level programming language compiler (`lutin`) are implemented and available on the net[4]. Those tools are, in particular, the main components of the new version of Lurette [17, 9], an automated testing tool dedicated to the Lustre language [7].

Some interesting problems are remaining in this framework. The weight mechanism allows to express which constraint are more likely to be tried, but once a constraint is elected, there is no way to act upon the choice of a particular solution. This problem is particularly evident with numerical values: for the time being we guarantee a uniform choice within the set of solutions. It could be interesting to define more sophisticated repartition mechanisms.

The design of high-level languages that can be compiled into WTS is also interesting. For the time being, we have a language where sequential statements are inspired by regular operators. An appealing solution would be to define a language inspired by the Esterel language [4]: this language propose a notion of *instantaneous* sequence and loops which is different to the ones of regular expressions. Moreover, it provides a set of high-level, useful control structures like local parallelism, watch-dogs and traps. Another solution would be to design a graphical language *à la StateCharts*, based on hierarchical concurrent automata; in this case, the statements of the synchronous language SynchCharts [1] are certainly the best example to follow.

[4] `www-verimag.imag.fr/~synchron/tools.html`

References

[1] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, jul 1996.

[2] A. Benveniste. Constructive probability and the SIGNalea language : building and processes via programming. Technical Report RR-1532, INRIA, 1991.

[3] M. Bernardo, L. Donatiello, and P. Ciancarini. Stochastic process algebra: From an algebraic formalism to an architectural description language. *Lecture Notes in Computer Science*, 2459:236ff, 2002.

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

[5] C. Derman. *Finite State Markovian Decision Processes*. Academic Press, 1970.

[6] R. J. V. Glabbeek, S. A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Information and Computation*, 121(1):59–80, 15 Aug. 1995.

[7] 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.

[8] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[9] E. Jahier, P. Raymond, and P. Baufreton. Case studies with lurette v2. In *1st International Symposium on Leveraging Applications of Formal Methods, ISoLA 2004*, Paphos, Cyprus, Oct. 2004.

[10] C. W. Johnson. A probabilistic logic for the development of safety-critical, interactive systems. *International Journal of Man-Machine Studies*, 39(2):333–351, 1993.

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

[12] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989.

[13] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, Aug. 1992. LNCS 630, Springer Verlag.

[14] M. O. Rabin. Probabilistic automata. *Information and Control*, 6:230–245, 1963.

[15] P. Raymond. Recognizing regular expressions by means of dataflows networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)* Paderborn, Germany. LNCS 1099, Springer Verlag, July 1996.

[16] 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, Apr. 2002.

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

[18] K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):419–423, June 1968.

[19] 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.