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 present an operational model for describing random reactive systems. Some models have already been proposed for this purpose, but they generally aim at performing global reasoning on systems, such as stochastic analysis, or formal proofs. Our goal is somehow less ambitious, since we are rather interested in executing such models, for testing or prototyping. But on the other hand, the proposed model is not restricted by decidability issues. Therefore it can be more expressive: in particular, our model is not restricted to finite-state descriptions.

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 possible behaviors, while the stochastic approach also takes into account probabilistic information.

There exist many formalisms to express non-determinism and 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 [14] is extended with probabilistic features in [22, 20]. PCTL [12] is an example of temporal logic extended with probabilities. From a more operational point of view, we can cite stochastic extensions of process algebras [13, 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 article, we first present the abstract machine language, named 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 basic qualitative model describing a set of behaviors 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 represent 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 reactions: all values in memory are changing simultaneously when a reaction is performed. The previous values of the memory correspond to the source data-state, and the current values 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 leads to a restriction: we need to 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 probabilities to state/transitions systems has been largely studied [16, 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 influence 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 smaller than one, which meaning 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 then the expressive power would be 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 find 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 \(q \rightarrow_t q'\), is qualitatively equivalent to \(q \rightarrow^t q'\). 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 issue for reactive systems. The problem of merging sequential and parallel constructs has been largely studied: classical solutions are hierarchical automata “à la StateCharts” [15, 1], or statement-based languages like Esterel [4]. Our opinion is that deeply merging sequence and parallelism is a problem of high-level language, and that it is sufficient to have a notion of global parallelism: intuitively, local parallelism can always be made global by adding ex-

\(^1\)concretely, we have developed a constraint solver for mixed Boolean/linear constraints.
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 illustrates this point in Figure 1a: the first automaton (resp. the second) has the choice between the constraints $A$ and $B$ (resp. $X$ and $Y$), that are 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 automaton, $X$ has a big weight $x >> 1$ comparing to $Y$. Suppose that the data-state makes it impossible to satisfy $A \land X$, then it is impossible to satisfy the stochastic demand of both components. There are mainly two ways to solve the problem:

- Consider that weights are not only local information, but also influence 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 handled 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: $V = V_i \cup V_o \cup V_l \cup V_p$. The previous variables are meant to refer to previous values of the other variables in $V_p = V_i \cup V_o \cup V_l$. Each previous variable is denoted by $\bullet v$ where $v \in V_p$. Moreover, each variable in $V_p$ 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, $Type(\bullet v) = Type(v)$.

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 $(\sigma_i, \sigma_p)$, where $\sigma_i$ (input valuation) associates a value with each input, and $\sigma_p$ (previous valuation) associates a value with each previous variable. Previous valuations are also called data-state. In particular, the default values of variables define the initial data-state, denoted by $\sigma_0^p$.

Formula. A formula is any well-typed Boolean expression made of variables, constants, and classic logical and numerical operators ($\land, \lor, \neg, =, >, <, \geq, \leq, +, -, /$). 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 ($V_c \cup V_i$). The notation $(\sigma_i, \sigma_p) : (\sigma_o, \sigma_l) \models f$ states that, given a context $(\sigma_i, \sigma_p)$, the controllable valuation $(\sigma_o, \sigma_l)$ satisfies $f$. 

![Figure 1. Weights and parallelism: the parallel composition (1a), and, assuming that $A \land X$ is unfeasible, the product solution (1b) and the arbiter solution (1c).](image-url)
2.2.2 Control structure

At 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: \( Q = Q_s \cup Q_t \). The initial control state is a particular stable state \( q^0 \in Q_s \).

Weights. Weights are positive numerical functions of the uncontrollable variables: \( \mathcal{W} : \Sigma_i \times \Sigma_p \rightarrow Q \cup \{ \infty \} \). More concretely, they are numerical expressions made of inputs, previous variables, classical operators or ad hoc computable functions. The \( \infty \) 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 transition 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: \( T \subseteq Q \times F \times \mathcal{W} \times Q \), and we note \( q \xrightarrow{f,w} q' \in T \) 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 combinatorial loops are statically rejected.

2.3 Operational Semantics

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 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 definition of global transitions is almost a classical composition where constraints are combined with the \( \wedge \) operator, and weights with the \( \ast \) operator. The only problem is to enforce synchronization on stable states.
- let \( s_1, s_2 \) be stable states, \( t_1, t_2 \) be transient states, and \( q_1, q_2 \) be transient or stable states:
  \[
  \begin{align*}
  & (s_1, s_2)_{w_1} f_1 (q_1, q_2) \text{ iff } s_1 f_1 w_1 (q_1) \text{ and } s_2 f_2 w_2 (q_2) \\
  & (t_1, t_2)_{w_1} f_1 (q_1, q_2) \text{ iff } t_1 f_1 w_1 (q_1) \text{ and } t_2 f_2 w_2 (q_2) \\
  & (s_1, t_2)_{w_2} f_2 (s_1, q_2) \text{ iff } t_2 f_2 w_2 (q_2), \text{ and symmetrically for } (t_1, s_2).
  \end{align*}
  \]

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 \( s \) and an additional waiting transient state \( \tilde{s} \) for each stable state \( s \). The global state space is then defined as the Cartesian product over the extended component state spaces. The definitions of initial and stable global states do 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 being similar:

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

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 build on the fly to avoid state space 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 $\sigma_p^0$ is the default map of the variables,

and for each $k$:

• $\sigma^k = \nu^k$ (the valuation history meets the input history)

• $\sigma^k_p = \sigma^k + 1$ (the current-part of the valuation meets the previous-part of the next valuation)

• $(s^k, \sigma^k_p, \sigma^k)$ 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 the next stable state. More formally, let $s = q_0$ be the current control-state, $\sigma_p$ the current data-state, and $\sigma_i$ the current inputs. For all $k$, we note $\Theta_k = \{q_k, \ldots, q_0\}$ the set of transitions leaving $q_k$, and we use the notation $w_\tau$ to denote the weight attached to a transition $\tau$. 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 exists some transition $\tau$ from $q_k$ to $q_{k+1}$, the probability to complete a path $(q_0, q_1, \ldots, q_k)$ 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: $s_1 \nu_1 q_1 \nu_2 q_2 \cdots q_{n-1} \nu_n q_n s'$, where all $q_1, \ldots, q_n$ 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 $(f_\sigma^0, \nu_0)$, 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 $f_\sigma^0, \nu_0$ 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 time to time. Input variables are a Boolean $On$, which is true if the heater is on, and a rational $T$, which indicates the temperature outside the room. The only output variable is the rational $t$, indicating the temperature inside the room. Local variables are the rational $\delta$, 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 $t$ and $j$. 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 outlet transition leading back to $s_{on}$ (resp. $s_{off}$); the elected formula is therefore $0 < \delta < 0.2$ (resp. $-0.1$ and $0$), and that $\delta$ 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 $true \land 10 \leq j \leq 15 \land t = 3 \frac{t - T}{4}$. It states that the local variable $\delta$ will be uniformly drawn between 0 and 0.2 (resp. $-0.1$ and $0$), and that $\delta$ is then used to increase the temperature. This is intended to model that, whenever the window is open, the temperature becomes closer to 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.

Another solution is to propose a different style for expressing the control. In [18], 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.

\(^2\)The tool, LUCKY, is available, at : www-verimag.imag.fr/~synchron/tools.html.
the sequence \( s_1 \) \( \text{fby} \) \( s_2 \) states that the program first behaves as \( s_1 \), and then, if and when \( s_1 \) stops, as \( s_2 \).

- the \( n \)-ary non-deterministic choice \( \{ s_1 \mid \ldots \mid s_n \} \) means that the system behaves as one of \( s_1, \ldots, s_n \).

- the loop \( \text{loop} \ s \text{ pool} \) means that the behavior \( s \) is repeated zero or more times. This operator introduces the important notion of empty behavior (zero loop) which is studied in the sequel.

The actual language provides more statements. For instance, \( \text{assert} \ f \ in \ s \) 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 a 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 \( \{ s_1 \ \text{weight} \ 2 \mid s_2 \} \) means that, whenever both \( s_1 \) and \( s_2 \) 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 "to continue" is relatively more probable than "to stop" (the weight of "stop" is 1 by definition): loop weight \( 99 \ s \text{ pool} \) 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.

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 all values within this interval have the same probability.

An example of Gaussian loop is loop \( ^{\sim}1000:200 \), which means that the average number of iterations is 1000, with a standard deviation of 200.

---

\(^3\)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.

---

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

\( V_i = \{ On, T \}, V_o = \{ t \}, V_f = \{ \delta, j \}, V_p = \{ \bullet t, \bullet j \}, \)

\( Q_s = \{ S_{on}, S_{off}, S_{open} \}, q_0 = S_{on}. \)
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 done at least 100 times. On the other hand, since s can do nothing at each step, the loop statement can generate the empty behavior. One one hand, this construct suggests that something must be done at least 100 times. On the other hand, since s can do nothing at each step, the loop statement can generate the empty behavior. 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).

```plaintext
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 program 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 \( E^\circ \), where the \( \circ \)-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 [21] which consists in using \( \varepsilon \)-transitions (or equivalently transient states). Using 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 \( \varepsilon \)-transitions may introduce combinational cycles, when some sub-expression of the form \( E^\circ \) 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 \( E^\circ \) by some equivalent \( F^\circ \) such that \( \varepsilon \not\in F \) [17]. 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 an 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 below.

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 $I_\sigma$, it means that $M_E$ must consider that the prefix has already generated a non-empty behavior. If the control is given via $I_\varepsilon$, 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 $O_\sigma$ if itself or its prefix-context have generated a non-empty behavior, and it will pass the control to $O_\varepsilon$ if neither itself or the context have generated a non-empty behavior.

The key point of this intuitive definition is that it guarantees that any path from the $I_\varepsilon$ to $O_\sigma$ 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 non-instantaneous 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 $\varepsilon$ the language reduced to the empty behavior:

\begin{enumerate}
\item $O_\sigma = I_\sigma \cdot E \cup I_\varepsilon \cdot (E \setminus \varepsilon)$
\item $O_\varepsilon = I_\varepsilon \cdot (E \cap \varepsilon)$
\end{enumerate}

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 $I_\sigma$, bottom-left for $I_\varepsilon$, top-right for $O_\sigma$ and bottom-right for $O_\varepsilon$. 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 is the entry point, $O_\sigma$ is reached after the generation of an atomic behavior satisfying the formula (if such a behavior exists). Note that $O_\varepsilon$ is stable (filled circle), and that $O_\varepsilon$ 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 for the translation. 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, $\bullet \cdot c$ 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 \( (w_z) \) 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 \( (w_e) \).

The inductive property (1) is less trivial here, so here are some hints of the proof. We note \( I^E_x \), \( I^E_x \) the two input states of the sub-automaton \( M_x \), and \( O^E_x \) its top-output state. The outermost states are denoted as usual. We have by construction \( I^E_x = 0 \) (unreachable), hence, by induction on property (1): \( O^E_x = I^E_x \cdot (E \setminus e) \). By construction we also have: \( I^E_x = O^E_x \cup I_x \cup I_x \), so, \( O^E_x \) satisfies a recursive equation whose solution is:

\[
O^E_x = (I_x \cup I_x) \cdot (E \setminus e) = (I_x \cup I_x) \cdot (E^* \setminus e)
\]

by construction we have \( O_x = I_x \cup O^E_x \) (transitions leading to the outermost top-right state), hence:

\[
O_x = I_x \cup ((I_x \cup I_x) \cdot (E^* \setminus e))
\]

\[
= I_x \cdot (E \setminus e) \cup (E^* \setminus e) \cup I_x \cdot (E^* \setminus e)
\]

\[
= I_x \cdot E^* \cup I_x \cdot (E^* \setminus e)
\]

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: \( w_z = w_c = w \) and \( w_s = w_e = 1 \).

- 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: \( w_z = 1 \) and \( w_e = h \). If \( l \neq 0 \), performing no loop is forbidden: \( w_z = 0 \) 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; if \( \bullet c \) is within \( l \) and \( h \), the probability to stop is 1 over \( h - l \); at last once \( h \) is reached, the iterations must stop.

<table>
<thead>
<tr>
<th>( w_z )</th>
<th>( w_c )</th>
<th>( w_e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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 approximates this integral via a table of sampled values. With such a procedure \( G(m, s, n) \), we have: \( w_z = G(m, s, 0) \) and \( w_z = 1 - G(m, s, 0) \) \( w_z = G(m, s, c) \) and \( w_e = 1 - G(m, s, c) \).

4This interpreter is available, together with an exhaustive description of its concrete syntax [9], at the url: www-verimag.imag.fr/~synchron/tools.html.
Both the low-level abstract machine simulator (LUCKY) and the high-level programming language compiler (LUTIN) are implemented and available on the net. Those tools are, in particular, the main components of the new version of Lurette, an automated testing tool dedicated to the Lustre language.

Some interesting problems remain. The weight mechanism allows to express which constraints 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 a high-level language on top of WTS is still in progress. One objective is to provide a functional mechanism to improve modularity and reusability. Another objective is to propose more sophisticated control structures, such as local concurrency, exceptions and catch points.

References