Electronic Notes in Theoretical Computer Science 65 No. 5 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 8 pages

Describing non-deterministic reactive systems by means of regular expressions

Pascal Raymond, Yvan Roux [1,2]

VERIMAG (CNRS, UJF, INPG)
Grenoble, France [3]

Abstract

This paper presents a language based on regular expressions for describing non-deterministic reactive systems. It also presents some ideas on how to build (or adapt) tools for exploiting such a language (recognizers, generators and provers).

1 Introduction

A fundamental feature of reactive programs is that, in contrast with most interactive parallel systems, they are deterministic. The basic argument in favour of synchronous programming [7,2,3,1] is that it reconciles (logical) concurrency with determinism. However, there are some situations, in the design of reactive systems, where non-deterministic behaviors must be modeled:

- When validating a reactive program (simulation, testing, verification), a model of the environment must generally be provided. Now, the environment is only partially known, so its model is always non-deterministic.
- For expressing properties (either for verification, or to provide an oracle in automatic testing), one have to specify a set of correct behaviors, which can also be viewed as the possible behaviors of a non-deterministic system.
- To simulate a system under design, when some components are not yet fully specified: some partial specification of these components must be taken into account.

Several solutions have been proposed for describing non-deterministic reactive behaviors in the synchronous framework. In Lustre and Esterel, non-determinism can be modeled mainly in two ways:

- by considering auxiliary inputs, possibly restricted by assertions

[1] Email: Pascal.Raymond@imag.fr
[2] Email: Yvan.Roux@imag.fr
[3] URL: http://www-verimag.imag.fr

– by writing down “observers” [5], i.e., programs that observe relevant variables/signals, and decide, by computing a single output, whether the desired property is satisfied.

Those solutions are not really satisfactory: the first one requires to modify the interface of the program, while the second loses the important distinction between inputs and outputs.
As a consequence, we think that a relational language (like Signal) which directly describes non-deterministic generators of behaviors is more intuitive. However, the style of such a language is very important: the declarative style (of Lustre or Signal) is well adapted to describe properties that mainly involve data, while the sequential style (of Esterel or SyncCharts) is best suited to express properties on the control. Merging those two styles in a deterministic programming language is quite hard, but it appears to be much simpler for describing non-deterministic systems.
Our approach consists in defining a new language for describing non-deterministic systems that merges two styles:

– a declarative style (*à la Lustre*) to describe atomic reactions (a set of possibles events at a given instant),
– a sequential style (regular expressions) to describe the dynamic of the system (how atomic reactions are sequenced in time).

The paper is organized as follows: first we present the basic concepts of the language, together with a simple semantics in terms of sets of behaviors. Those concepts are chosen for their usefulness, conciseness and expressive power, but always keeping in mind that the resulting language must be exploitable by automatic tools.
In the second part, we present general ideas on how one can design tools for exploiting the proposed language (acceptors, simulators, provers).

2 The description concepts

We do not present a concrete syntax, but only the concepts of the language. The presentation is incremental, from basic concepts to more sophisticated ones. At each step, we present the corresponding semantics. A complete, operational semantics, even though interesting, would have been quite long and complex, so we prefer to present a more “intuitive” denotational semantics, whose goal is simply to show that all constructs are mathematically well founded.
In the whole section, we try to keep a pragmatic approach when introducing new features: on one hand they must be useful and natural for the user, on the other hand they must be exploitable by automated tools.

2.1 Basic concepts

A reactive system is characterized by a set of typed variables V (either boolean, integer or real). The corresponding domain of values is noted $\mathcal{D}_{\mathcal{V}}$ (some Cartesian product of $\mathbb{B}$, $\mathbb{Z}$ and $\mathbb{R}$). An event is a particular valuation $v \in \mathcal{D}_{\mathcal{V}}$ of the variables, a behavior is a finite sequence $\sigma \in \mathcal{D}_{\mathcal{V}}{}^*$; the 0-length sequence is denoted ε. The concatenation of two sets of behaviors is defined by $X \cdot Y = \{\nu\sigma \ / \ \nu \in X, \sigma \in Y\}$. The integer power of a set is defined by $X^0 = \{\varepsilon\}$ and $X^{n+1} = X \cdot X^n$. The Kleene-star operator is defined by $X^* = \cup_{n \in \mathbb{N}} X^n$. The prefix partial order over $\mathcal{D}_{\mathcal{V}}{}^*$ is denoted by $\preceq$ ($\nu \preceq \sigma \Leftrightarrow \exists \rho / \sigma = \nu\rho$). A set of behaviors $X \subseteq \mathcal{D}_{\mathcal{V}}{}^*$ is *prefix-closed* if $\forall \sigma \in X \ \ \nu \preceq \sigma \Rightarrow \nu \in X$. Let $X \subseteq \mathcal{D}_{\mathcal{V}}{}^*$, we note $Pfx(X)$ the least prefix-closed subset containing X (we call it the *prefix-closure* of X).

2.2 Formulas

A formula over V is a Boolean expression made of constants, variables from V and logical and arithmetic operators. A well-typed formula trivially denotes a Boolean function $f : \mathcal{D}_{\mathcal{V}} \rightarrow \mathbb{B}$, and defines a set of valuations $\{v \in \mathcal{D}_{\mathcal{V}} \ / \ f(v) = \mathit{true}\}$.

2.3 Basic regular operators

The description language we propose is basically a language of regular expressions where the basic "symbols" are formulas over V. Our idea is that regular operators meet the intuition of temporal operations (sequence, loop, non-deterministic choice).

The semantics of such a regular expression e is a set of behaviors $Bhv(e)$. More precisely, an expression is:

- a formula f, with $Bhv(f) = \{v \in \mathcal{D}_{\mathcal{V}} \ / \ f(v) = \mathit{true}\}$ (i.e. 1-length behaviors made of a valuation satisfying f);
- a sequence $e \cdot e'$; with $Bhv(e \cdot e') = Bhv(e) \cdot Bhv(e')$;
- a non-deterministic choice: $e \mid e'$, with $Bhv(e \mid e') = Bhv(e) \cup Bhv(e')$;
- an unbounded loop (Kleene star): e^*, with $Bhv(e^*) = Bhv(e)^*$;
- a possibility: e^{ε}, with $Bhv(e^{\varepsilon}) = \{\varepsilon\} \cup Bhv(e)$.

Note that such a definition does not allow the definition of all the regular subsets of $\mathcal{D}_{\mathcal{V}}{}^*$: some denotation for $\emptyset$ and $\{\varepsilon\}$ are missing. But, from a practical point of view, $\emptyset$ has no interest at all, and $\{\varepsilon\}$ is useless since we have introduced the "possibility power".

For practical reasons, we may introduce some classical shortcuts like e^+ for $e \cdot e^*$.

2.4 Parallelism and distribution

A operation for expressing *concurrency* may be very useful: two behaviors are running in parallel and the global behavior results from this concurrent

execution. We adopt a *synchronous semantics* for parallelism: events (variables configurations) are sampled on a same global clock. This synchronous approach makes the semantics obvious: in terms of language, this operation is simply the *intersection* $Bhv(e\&e') = Bhv(e) \cap Bhv(e')$.
Such an operator does not change the expressive power of the regular expressions, but, unfortunately, it makes their exploitation much more complex. Depending on the kind of treatment we want to perform (generating some behavior, building a recognizer), this operator might be considered or not (cf.§3).
However, some restriction of the parallel operator may be useful and easy to handle by any tools; this is the case of the *formula distribution*: $f^*\&e$, which simply propagates a formula in each step of a behavior. From now on, we note this operation $f \triangleright e$ (read it *always f in e*).

2.5 System behavior

We have presented the core of the language, consisting of regular expressions where the basic symbols are relations between a given set V of typed variable. We address now the problem of describing a non-deterministic system with such expressions.
• *Implicit prefix-closure* At the main level, we are only interested in prefix-closed sets of behaviors. One solution is to impose that the main expression must denote a prefix-closed set. But writing such an expression maybe tedious; for instance the system that repeats indefinitely a sequence of a, b, c should be written: $(a \cdot b \cdot c \cdot)^* \cdot (a \mid a \cdot b \mid a \cdot b \cdot c \mid a \cdot b \cdot c \cdot d)$. In order to alleviate the user's work, we may at least propose the prefix-closure operator *Pfx* in the language; moreover, we will always consider that the main regular expression (the one describing the behaviors of the considered system) is implicitly guarded by a prefix-closure operator. The semantics of the main expression e is then $Pfx(Bhv(e))$.
• *Inputs and outputs* A reactive system has a notion of input (uncontrollable) and output (controllable) variables. So, instead of a global set of variables V, we consider now the disjoint union of a set of inputs I, and a set of outputs O. A system is then given as a triplet $S = (I, O, e)$, where e is the main regular expression.
The semantics of inputs/outputs can be given in some operational way: a formula over $V = I \cup O$ is interpreted as a function from valuations of I to sets of valuations of O: $f : \mathcal{D}_{\mathcal{I}} \rightarrow \mathcal{P}(\mathcal{D}_{\mathcal{O}})$, with $f(i) = \{o \ / \ f(i, o)\}$.
Such a function is called a *non-deterministic reaction*, and we note $R_{IO} = {\mathcal{D}_{\mathcal{I}}}^{\mathcal{P}(\mathcal{D}_{\mathcal{O}})}$ the set of non-deterministic reactions. According to this definition, the semantics of a regular expression e is now a (arbitrary long) sequence of non-deterministic reactions, that means some $\sigma \in R^*_{IO}$.
This operational semantics is interesting, since it is, for instance, a good basis for a concrete simulation tool. However, it is not really useful to define recognizers. So here again we prefer to adopt a very simple denotational semantics:

we still consider that a regular expression e denotes a set of behaviors $Bhv(e) \subseteq \mathcal{D}_{\mathcal{I}\cup\mathcal{O}}{}^*$, and a system $S = (I, O, e)$ denotes the set of behaviors $Pfx(Bhv(e))$. For a particular sequence of input valuations $\iota = \iota_1\iota_2 \cdots \iota_n \in \mathcal{D}_{\mathcal{I}}{}^*$, the *actual behaviors* of S are those which are compatible with the given input sequence: $B_S(\iota) = \{(\iota_1, o_1)(\iota_2, o_2) \cdots (\iota_n, o_n) \in Pfx(Bhv(e))\}$.

• *Local variables* Using intermediate variables may be useful, in particular to make the formula shorter and more readable. For instance, if one wants to express several constraints $c_1, \cdots, c_k$ on the result of a complex numerical function $h(x, y)$, it may be convenient to give a name to this result, and then use this name to express the constraints: $(r = h(x, y)) \wedge c_1(r) \wedge \cdots \wedge c_k(r)$.
The semantics of local variables is quite obvious: they are existentially quantified. For practical reasons, we define this quantification at the higher level. Let $S = (I, O, L, e)$ be a system, the actual behaviors corresponding to a given input sequence ι is:
$B_S(\iota) = \{(\iota_1, o_1) \cdots (\iota_n, o_n) \ / \ \exists(\iota_1, o_1, l_1) \cdots (\iota_n, o_n, l_n) \in Pfx(Bhv(e))\}$

• *The previous operator* The description language, as it is defined for the moment, makes a clear distinction between instantaneous relations (expressed by formulas) and temporal relations (expressed by regular expressions). For purely Boolean systems, the language is powerful enough to express any safety (temporal) relation. For instance, to express that a Boolean variable x is oscillating, one can write (formulas are inclosed within brackets): $[\neg x]^\varepsilon \cdot ([x] \cdot [\neg x])^*$.
However, expressing Boolean temporal properties with regular operators may be quite tedious, and for numerical variables it is simply impossible. For instance, how can we express with a finite temporal structure that a numerical variable is increasing during an unbounded amount of time?
In order to solve the problem, we introduce a *previous* operator in formulas: $\mathrm{pre}(x)$ intuitively refers to the previous value of the variable x. Note that such an operator [4] is quite common in description formalisms based on the concept of pre- and post-conditions, and it is also the "basic" concept of synchronous languages like Scade/Lustre.
With such an operator, one can express temporal properties over numerical variables: $[n \geq \mathrm{pre}(n)]^*$ express that n is increasing for a certain time. Even for Boolean variables, this operator makes the expression simpler: $[x \neq \mathrm{pre}(x)]^*$.
Together with local variables, it becomes possible to express complex behaviors, like a constrained loop; in the following example, we use a local counter c to express that the behavior denoted by some expression e must be repeated between 10 and 20 times.

$$[c = 0] \cdot (\ [c = \mathrm{pre}\ c + 1] \rhd \mathbf{e}^*) \cdot [10 \leq c \wedge c \leq 20]$$

In order to define the semantics, we first consider that each $\mathrm{pre}(x)$ is a new variable, independent from x. As a consequence, an expression e denotes a set of sequences of pairs over the domain $\mathcal{D}_{\mathcal{V}} \times \mathcal{D}_{\mathcal{V}}$ (previous values/current

[4] or it's dual *next*[6]

values). At the main level, we extract from those sequences of pairs the ones such that the previous and the current part are matching from one step to another: $(pv_1, v_1)(pv_2, v2)\cdots(pv_n, v_n)\ \ /\ \ pv_{i+1} = v_i$

3 From language to tools

Given the description of a non-deterministic system, there are roughly three kinds of problems that automated tools may address:

- can a particular behavior be realized by the system? In term of tools, it is equivalent to build a recognizer of the system's behaviors. In the scope of validation, this is the role devoted to what is usually called the *oracle*.
- generate one (or more) behaviors from the system description. This is equivalent to build a randomized simulator of the system. This is interesting for testing a system under construction (and then not fully specified), and to simulate the environment (by definition non-deterministic) of a system under test.
- are two systems equivalent, in the sense that their sets of behaviors are equal? An equivalent problem is to decide if a system is infeasible (the set of behaviors is empty). This problem is the one of formal verification.

Those problems (and the corresponding tools) are of different complexity, and we may need to restrict the source description language in order to provide realistic solutions.

3.1 Recognizers

Building recognizers is the simplest problem: it does not require to solve formulas, but just to compute them. Regular expressions are not a problem either: there exist efficient algorithms for translating such an expression into an executable machine (observer) as long as the the intersection operator is not allowed. This is the main characteristic of algorithms producing non-deterministic automata ([10,8]).
Unfortunately, allowing the intersection operator (and/or the complement operator) requires the construction of deterministic automata, and then, it has in the worst case an exponential cost (both in time and in space).

3.2 Simulators

For building a simulator, things becomes much more complex, since the problem is equivalent to solve a sequence of formulas. The problem is in general undecidable because of the numerical variables; a good solution seems to restrict the formulas to a decidable (and of reasonable cost) subset, like linear constraints.
Another important problem is introduced by the pre operators. Intuitively, those operators may interfere with the traversal of the control structure, leading to a problem that we call *non-expandability*: after the generation of some

steps, the current values of the pre's are such that none of the possible "futures" of the control structure are satisfiable. One solution is to backtrack in the past to (try to) find another sequence; this solution is interesting, but it is also clearly expensive. We can also simply reject descriptions leading to this problem: in some sense, the user may describe non-deterministic but expandable systems (i.e. systems that are able to indefinitely react to their environment).
A tool for automatic test case generation was made with those restrictions (linear constraints, no backtrack), where the description language was based on Lustre observers [9]. The experience has shown that Lustre observers are not always suitable for this purpose, since they do not provide constructs for controlling the dynamic of the simulation. We are now working on a new tool where the input language is based on the concepts presented here. A simulation is then much more understandable for the user, since it can be related at each step to some part of the control structure (some sub regular expression). Moreover, we are trying to extend the language with probabilistic constructs providing a better control of the simulation.

3.3 Provers

This is the most complex problem. The special case of purely Boolean systems is decidable (but dramatically costly). As soon as numerical variables are involved, the problem is undecidable, and a common solution is then to make conservative abstractions in order to perform at least partial decision. The set of methods and tools are very large, depending on the abstraction, and the basic decision methods involved. A quite common characteristic of those methods is the model of interpreted automata, which is quite close to what is provided by our description language: a finite control structure decorated by semantics labels. Translating the description into an interpreted automaton suitable for a given tool may be possible. In particular, feeding verification tools specialized for Lustre-Scade programs [4] is quite easy, since they use Lustre observers, and we know how to translate regular expressions into Lustre.

4 Conclusion and perspectives

There exist a lot of methods and languages for describing reactive systems properties. Even so, we think that there is a place left for a dedicated language providing a good expressive power while remaining rather simple and intuitive. We have chosen a description style where non-deterministic reactions (viewed as relations between input/output variables) are sequenced using an imperative sequential structure. To express this control structure, we have chosen regular expressions since it is a well know and natural paradigm to talk about discrete time. The problem of exploiting such a language in a validation framework is approached in terms of acceptance, simulation, and proof. The concrete implementation of the language is called Lutin. We have a tool that efficiently translates a Lutin program into a classical observer in Lustre. Such an observer can be executed as an acceptor (oracle) for test purpose, and can also be used in the Lustre verification tools to describe the assumptions or the expected properties. For simulation purposes, it appears that the translation into an observer loses important informations like the explicit control structure, so we are now working on a simulation tool especially dedicated to Lutin. Moreover, for simulation purposes, functional properties are no longer sufficient and we have to extend the language with probabilistic constructs.

References

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

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

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

[4] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[5] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[6] L. Lamport. The temporal logic of actions. *ACM Toplas*, 16(3):872–923, 1994.

[7] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. Programming real-time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[8] 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, jul 1996.

[9] 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.

[10] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–423, June 1968.