Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Case Studies with Lurette V2

Erwan Jahier[1], **Pascal Raymond**[1], **Philippe Baufreton**[2]

[1] VERIMAG,CNRS – Grenoble, France
[2] Hispano Suiza – Moissy Cramayel, France

Received: date / Revised version: date

Abstract. Lurette is an automated testing tool dedicated to reactive programs. The test process is automated at two levels: given a formal description of the System Under Test (SUT) environment, Lurette generates realistic input sequences; and, given a formal description of expected properties, Lurette performs the test results analysis.

Lurette has been re-implemented from scratch. In this new version, the main novelty lies in the way the SUT environment is described. This is done by means of a new language called *Lucky*, dedicated to the programming of probabilistic reactive systems.

This article recalls the principles of Lurette, briefly presents the Lucky language, and describes some cases studies from the IST project Safeair II. The objective is to illustrate the usefulness of Lurette on real case studies, and the expressiveness of Lucky in accurately describing SUT environments. We show in particular how Lurette can be used to test a typical fault-tolerant system; we also present case studies conducted with Hispano-Suiza and Renault.

Key words: Automated testing, tool environment, real-time embedded systems, reactive programs, synchronous languages, stochastic machines.

1 Introduction

This article presents some case studies in testing reactive embedded programs. This kind of system can be found in domains such as transportation and control/command, and are in general safety critical. Indeed, they require to be strongly validated before being used.

Synchronous languages [4,1,13,12], which provide a formal semantics of time and concurrency, played a significant role in the introduction of formal methods in industry. Since those languages have a precise semantics, it is possible to use formal verification techniques, mainly based on model-checking [15,6]. Formal validation methods are appealing, but they are limited for theoretical and practical reasons to relatively simple and small systems. For complex and big systems, in particular those where numerical aspects are important, testing is the only tractable method. Testing is obviously not exhaustive, but it can help to discover bugs, and increase confidence in the system.

Testing reactive systems. Testing reactive systems raises specific issues. First of all, the execution of reactive systems is (virtually) infinite; a test case is then an arbitrary long sequence of input vectors. Moreover, the system is not intended to run in a completely random environment, and some properties must be taken into account in order to generate relevant (or even interesting) test sequences. More specifically, the relevance of the inputs may depend on the behavior of the system itself, since the system influences the environment which in turn influences the system. This feed-back aspect is important for reactive systems, and it makes off-line generation of test sequences impossible. In some sense, testing a reactive system requires running it in a simulated environment.

Several methods and tools have been proposed for testing reactive systems, which in general assume a full knowledge (glass-box) of the System Under Test (SUT) [30], and/or do not deal with numerical programs [26,7, 10,17].

The "model-based" approach [10,17] supposes there exists some formal description of the SUT. This model is used in combination with the hypothesis made on the environment plus the properties to be checked (the test purposes). Verification techniques based on model-checking, partial orders, or bisimulation, are then applied to derive expected traces to be compared with actual traces produced by the SUT.

However, *black-box* testing is sometimes the only possibility because the code of the system is either not available or not tractable (e.g., because of numerics); the only thing we can do then is to execute it and observe the results.

Compared to other methods, our objective is more ambitious since we want to deal with numerical values. But on the other hand, as we do not suppose we have a model of the SUT, we only focus on providing a very general and efficient machinery to describe and generate sets of test cases. We do not deal with how to obtain test case models.

The Lurette tool. In this work, we use a testing tool called Lurette [28] which is black-box oriented and able to handle numerical values. This tool currently supports Lustre [14], Scade[1] [9] and Sildex[2] programs. It can also be easily extended to other languages like Esterel [5], or even C programs as far as they meet some interfacing conventions.

This tool is the second version of Lurette, where the main difference lies in the way the environment of the SUT is described and simulated. We started from the idea that, since the SUT behaves as a reactive system, it is natural to describe it as a reactive program. But conversely to the SUT which is a "real" program, the environment is not intended to be deterministic: we have some knowledge on how it behaves, but we are in general unable to exactly predict its behavior. Moreover, for testing purpose, non-determinism is not always sufficient: we do not only want to express that some behaviors are possible but more precisely that some behaviors are more *probable* than others. As a consequence, the SUT environment is described as a stochastic reactive system.

Stochastic reactive systems. Many formalisms exist to describe non-deterministic and probabilistic systems. Some of them are based on classical finite automata and Markov Chains [8]. The more specific model of I/O automata [25] is extended with probabilistic features in [31]. PCTL [21] is an example of temporal logic extended with probabilities. For a more operational point of view, we can cite stochastic extensions of process algebras [22,3], or Signalea [2], an extension of the synchronous language Signal.

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

[1] Scade is an integrated programming environment based on the Lustre language, see `http://www.esterel-technologies.com`

[2] Sildex is an integrated programming environment based on the Signal language [24], see `http://www.tni-world.com/sildex.asp`

Our approach is less ambitious, since we focus on *simulation* only. More precisely, we do not care if the model can not be analyzed, as long as it can be efficiently simulated[3]. In other terms, our goal is more to *program* stochastic reactive systems, rather to reason about them.

Organization of the paper. After a brief recall of the Lurette principles, we introduce Lucky, a new language used to describe and simulate the environment. Then, we present four case studies that emphasize the main characteristics of the tool.

1. The first one is a resistance to temperature converter, developed in Scade, and provided by Hispano-Suiza in the framework of the IST project Safeair II. It is not a typical reactive system, in the sense that there is no feedback between the environment and the program, but it illustrates the numerical capabilities of the tool. Two bugs were found in this (untested) program.
2. The second one, also developed in Scade by Hispano-Suiza, computes a propulsion nozzle position. It is still a combinational program, but slightly more complex. One bug was detected.
3. The third one is a fault-tolerant controller, where the feedback aspects are important. It is written in Lustre. It is not an industrial case study, but it has been inspired from a real one, and we believe it is typical of what a fault-tolerant system is. For this case study, we illustrate how simple environment models can be defined quickly, and then refined to more accurate models.
4. The last one is a brake-by-wire system developed in Sildex, and provided by Renault in the framework of the IST project Safeair II. It presents an original use of the oracle to compare two different versions of a software. It also illustrates the expressiveness of Lucky.

2 Lurette, an automated testing tool

We recall in this section the principles of Lurette [28]. More details about the new version of the tool can be found in the *Lurette V2 user guide* [18].

2.1 Automatic generation of realistic inputs: the System Under Test (SUT) Environment

The main challenge in automating the test process of a reactive program is the ability to generate *realistic* input sequences to feed the SUT. Indeed, realistic input sequences cannot be generated off-line, since the SUT generally influences the behavior of the environment it

[3] Note that, even for simulation, some restrictions are necessary. But they are likely to be weaker than those required for global reasoning

is supposed to control, and vice-versa. Imagine, for example, a heater controller for which the input is the temperature in the room, and the output is a Boolean signal controlling the heater.

In other words, we need an executable model of the environment in which the inputs are the SUT outputs, and the outputs are the SUT inputs.

Basically, the Lurette input sequence generation engine is a linear constraint solver and drawer. The Boolean part of the solver is based on Bdd [29] and the numeric part is based on a convex Polyhedron Library [20]. Constraints on the environment variables (that may depend on memories and on SUT outputs), are solved, and one solution is drawn at each step to feed the SUT. Such SUT environment constraints are described by Lucky programs. Lucky is a language dedicated to the construction of probabilistic machines; it is presented in Section 3.

2.2 Automatic test decision: the oracle

The second thing that needs to be automated is the test decision. To do that, we use the technique of *observers* used in verification [16]. An *observer* of a program P is a program that takes in input the inputs and the outputs of P, and that returns exactly one Boolean variable. It lets one express any safety property [23].

In the context of program testing, those observers play the role of automated *oracles.* A test data sequence is considered as correct with respect to a temporal safety property if the oracle that encodes it always returns true.

Such oracles can be written in (almost) any programming language, as soon as (1) it can store informations from one call to the other in a persistent memory (since the oracle may depend of the history of inputs/outputs), and (2) it can compile into an executable procedure. For Lurette, we propose to use the same data-flow programming languages that Lurette targets, namely, Lustre, Scade and Sildex.

2.3 The Lurette data flow loop

Fig. 1 outlines the Lurette data flow between the different entities, namely, the SUT, its environment, and the test oracle.

The environment outputs serve as SUT inputs, and SUT outputs serve as environment inputs, apart from the first step. Therefore, to be able to start such a looped design, one entity has to start first. In order to avoid putting hypotheses on the SUT (for instance, the SUT should be able to produce outputs without inputs at the first step), the environment starts first. This means that a valid environment for Lurette is one that can generate values without any input at the first instant. The role of the `boot` keyword of Fig. 1 is precisely to signal the environment it should start generating values.

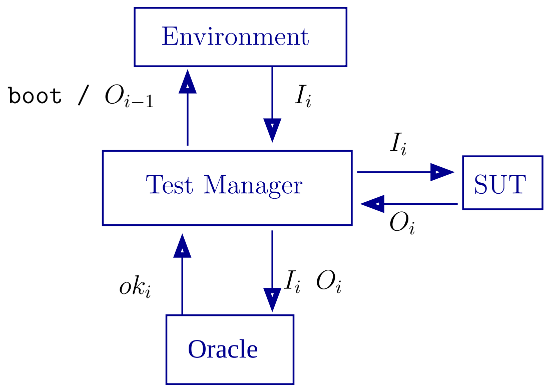


Fig. 1. The Lurette data flow loop.

Hence, once the environment has received the `boot` signal, it (non-deterministically) produces a vector of values I_1. Lurette sends this input vector I_1 to the SUT, which returns the output vector O_1. Lurette then sends both I_1 and O_1 to the oracle. The oracle returns a single Boolean ok_1, which is true if and only if I_1O_1 satisfies the property.

If ok_1 is false, then the testing process stops and a counter example that violates the property has been found. If ok_1 is true, then the testing session continues in exactly the same manner, except that this time, O_1 is sent to the environment, which returns yet another input vector I_2. I_2O_2 is sent to the oracle which returns true if the trace $(I_1O_1;\ I_2O_2)$ satisfies the property.

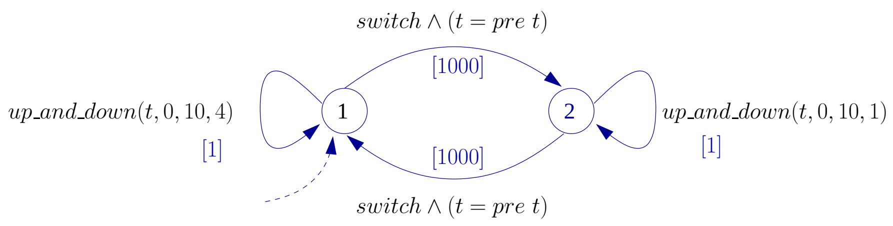
3 Lucky, a language to program stochastic machines

A first version of Lurette [28] was designed in the late nineties. In this version, the SUT environment was described by means of temporal constraints expressed with the Lustre language. The role of Lurette was then to solve those constraints in order to build a relevant input sequence for feeding the SUT.

From a practical point of view, the use of Lustre for describing input sequences was not completely satisfactory. First of all, the declarative style is not adapted for describing sequential scenario. Moreover, it does not provide any means for expressing that some input sequence is more probable than another.

In order to overcome those limitations, a new language named Lucky, has been defined. Unlike Lustre, this language provides an explicit control structure which makes it easier to express sequential scenario and also to express probability issues.

Basically, a Lucky program is a probabilistic interpreted automaton where a transition represents an atomic reaction of the SUT environment. Each transition is labeled by:


$switch \wedge (t = pre\ t)$
[1000]
$up\_and\_down(t, 0, 10, 4)$
[1]
1
2
$up\_and\_down(t, 0, 10, 1)$
[1]
[1000]
$switch \wedge (t = pre\ t)$

where:

$$\begin{aligned} up\_and\_down(X, Min, Max, Bound) = {} & (|X - pre\ X| < Bound) \quad \wedge (\\ & \textbf{if } (pre\ X < Min) \vee ((pre\ X < Max) \wedge (pre\ pre\ X \leq pre\ X)) \\ & \textbf{then } (X > pre\ X) \textbf{ else } (X < pre\ X)\ ) \end{aligned}$$

Fig. 2. A Lucky automaton with one Boolean input *switch* and one real output t.

– a constraint (a relation) defining the possible SUT input values,
– a weight (an integer) defining the relative probability for this transition to be taken.

The operational semantics of Lucky is first presented on a very simple example. The features of the language are then presented in more details.

3.1 A simple Lucky program

A simple Lucky program is drawn on Fig. 2. It has one Boolean input *switch*, and one real output t, which means that it is supposed to implement the environment of a SUT which inputs t and outputs *switch*.

The relation level. Each transition in this automaton is labelled by a relation (a conjunction of constraints) and a weight (an integer). Each relation, which holds over the automaton input, output, and memories (e.g., $pre\ t$), defines how to compute one reaction of the program.

– The relation "$switch \wedge (t = pre\ t)$" is satisfiable if and only if the input *switch* is true; it states that t keeps its previous value in such a case.
– The relation "$up\_and\_down(t, 0, 10, 1)$" constrains the output t in the following manner: the difference between t and its previous value is always smaller than 1 ($|t - pre\ t| < 1$); if t was increasing (resp. decreasing) at the previous step ($pre\ pre\ t \leq pre\ t$) then t increases (resp. decreases) at the current step, unless its previous value is bigger than 10 (resp. smaller than 0). Refer to Fig. 7 in order to see the shape of a timing diagram of a variable constrained by such a relation. We will use that macro in most of the examples in this article.
– The relation "$up\_and\_down(t, 0, 10, 4)$" is similar, except that the bound over the derivative of t is set to 4 instead of 1.

Note that constraints are mixing controllable variables (outputs), and non-controllable variables (inputs and past values). As a consequence, a constraint may or may not be satisfiable.

The control level. The automaton describes how constraints are evolving among time: at each "instant" the control belongs to some state of the automaton. A *feasible* transition starting from this state is elected, and the control passes to the corresponding target state. A transition is feasible if the corresponding constraint is satisfiable according to the current values of the uncontrollable variables. When several transitions are feasible, the random selection is made according to their *relative weights*.

In Fig. 2, the program starts in state 1 (marked by a dashed arrow). If the input *switch* is false, only one transition is feasible, labelled by $up\_and\_down(t, 0, 10, 4)$. This constraint is solved, and one solution is drawn among its set of solutions.

If the input *switch* is true, then the transition from 1 to 2 is also feasible. One is labelled by a weight of 1, and the other one by a weight of 1000, which means that the latter has a probability of 1000/1001 to be elected.

Such an automaton models the fact that the control can move from one mode to another only when the input *switch* is true. It also models the fact that the current mode might not change even if *switch* is true (with a probability of 0.1 %), which can be convenient, for example, to model occasional errors.

Note that there are two sources of non-determinism in Lucky: one at the relation level, where several solutions to a set of constraints exist; and one at the control level, over which we have some quantitative control via the use of weights.

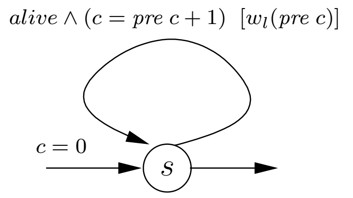


Fig. 3. An alive process description.

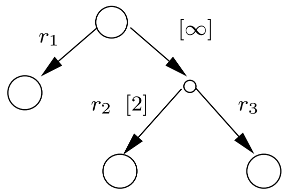


Fig. 4. Transient state and infinite weight.

3.2 Other Lucky concepts

The language provides some concepts that are not used in Fig. 2, but that will be needed later. We present here the most important ones.

Dynamic weights. In the example of Fig. 2, weights are integer constants expressing that some transitions are *always* more probable than others. This notion of static weights is quite restrictive since it does not allow to express that the fact that the history and/or the outside may influence the probabilities. This is why Lucky provides a more general notion of *dynamic weights*: a weight is a numerical function over the inputs and the past values.

A good example of the usefulness of dynamic weights 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.

Fig. 3 illustrates the use of dynamic weights to model such an alive process. An internal variable c is used to count the number of loops on the state s: its initialization is enforced when this state is reached, and its increment is enforced each time a loop is performed. The weight associated to the loop (w_l) is a decreasing function of $pre\ c$, for instance $w_l(pre\ c) = 1000 - pre\ c$.

Infinite weight. Lucky provides a special notation for infinite weight, in order to express a sound notion of mandatory choice. A transition labelled with the infinite weight has priority on any finite weighted transition. Note that there is a single notion of infinite weight: two feasible transitions with infinite weight have the same probability. The next paragraph illustrates how to assign finer-grained probabilities to different mandatory choices.

Transient states. For the time being, there is only one notion of control state: a state is a stable control point, and a transition between two states defines an atomic reaction. However, 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 $s \xrightarrow{f} t \xrightarrow{g} s'$, is qualitatively equivalent to $s \xrightarrow{f \wedge g} s'$. In contrast, transient states affect probabilities, and may be helpful to express complex conditional relative weights.

Figure 4 shows an example where a transient state is used to assign different relative probabilities to several mandatory choices: r_2 or r_3 both have the priority over r_1, and r_2 is twice more probable than r_3.

3.3 Restrictions on constraints

From an abstract point of view, a constraint is a relation between uncontrollable variables (inputs and internal memories) and controllable ones (outputs). At each step, the value of uncontrollable variables is known, hence the relation is reduced to a constraint over the controllable variables. This constraint must then be solved in order to find (if is exists) some actual solution.

The constraint solver is the core of the Lurette tool. Since it is fully automatic, it requires some restriction on the nature of the constraints, in particular for those concerning the numerical variables. Actually, Lurette requires numerical constraints to be linear: it uses a decision module based on polyhedra to solve such constrains.

3.4 Concurrency

In order to design large and complex systems, Lurette allows to define the environment as a set of Lucky automata running concurrently.

Intuitively, in this case, each Lucky automaton behaves as a generator of constraints, and the global behavior results in the conjunction of those constraints. In other term, the global behavior is defined as a synchronous product of the automata.

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 major difficulties. For constraints, the product is simply the logical `and`. Unfortunately, there is no obvious way for combining probabilistic information: as they are defined, transitions carry local information that may induce paradoxes when combined into a parallel composition. A simple example is shown in Fig. 5a: the first automaton

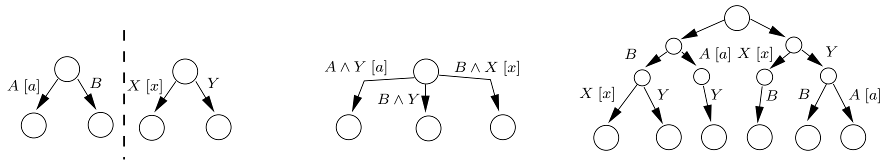


Fig. 5. Weights and parallelism: the parallel composition (5a), and, assuming that $A \wedge X$ is not feasible, the product solution (5b) and the arbiter solution (5c).

(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 $x >> 1$ compared 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 are mainly two ways for solving the problem.

1. Consider that weights are not only local information, but are also influencing the parallel composition: for instance, if a is much bigger than x, it 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 Fig. 5b.
2. 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 a stochastic information that can be added to influence it. The Fig. 5c 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.

3.5 Lucky, a target language

One of the goals when designing Lucky was to have a language with a simple operational semantics (it is a simple interpreted automaton) that is general enough to model any non-deterministic formal description. It was not necessarily meant to be a language for users but rather a target language for other higher-level languages, or third-party tools. However, as the examples provided in this article will illustrate, we believe that this language is readable enough.

Anyhow, we designed another language, Lutin [27], which compiles into Lucky. Lutin also aims at describing and simulating non-deterministic systems, but it is based on regular expressions instead of an explicit automaton, which sometimes makes the description of stochastic systems easier.

Moreover, a gateway from Lustre observers to Lucky programs can be done straightforwardly: it will result in a degenerate Lucky automaton with a single control state and a single (looping) transition labelled by the Lustre observer equations. Using Lucky instead of Lustre does not change the underlying synchronous computation model, but it gives a more "operational" style of description, in which non-determinism is explicit.

4 Case study 1: a resistance to temperature converter

We first illustrate the use of Lurette on a (untested) program that has been kindly provided by Hispano-Suiza, and which is written in Scade. Even if this node is rather small, Lurette still let us find two problems with it very quickly.

4.1 The specification of the converter

Hispano-Suiza has provided the following specification. The converter is a Scade node with one input `R`, representing a resistance (in Ohms) that comes from a sensor. It has one output `T`, representing the corresponding temperature (in Kelvin). The output is computed from the input using the function:

$$\begin{aligned} &\text{if } R > 0 \text{ then } T = C * R^2 + D * R + 273.15 \\ &\qquad \text{else } T = A * R^4 + B * R^3 + C * R^2 + \\ &\qquad\qquad D * R + 273.15 \end{aligned} \tag{1}$$

where A, B, C, and D are constants that we do not provide here.

4.2 The test session

Fig. 6 shows a Lucky program that models a possible environment for stimulating this converter. The first two lines declare the Lucky machine interface. Then come the node and transition declaration definitions. This very

```
inputs { T : real }
outputs { R : real ~init 200.0}
nodes { 1 : stable }
start_node  { 1 }
transitions { 1 -> 1 ~cond
                   up_and_down(R, 150.0, 500.0, 5.0) }
```

Fig. 6. A Lucky program modelling the converter environment.

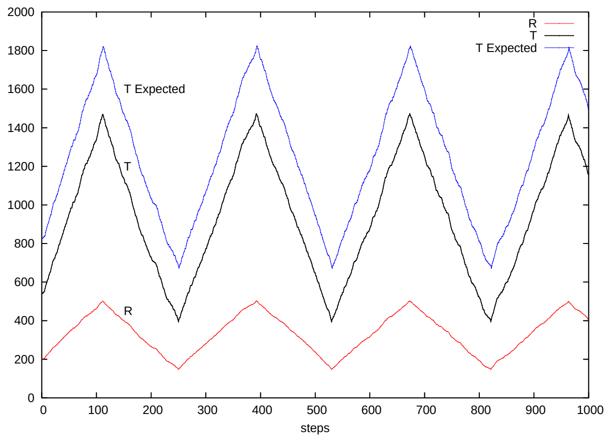


Fig. 7. The timing diagram of an execution of the converter.

simple automaton contains only one node and one transition. The transition states that the output `R` should vary up and down between 150 and 500, with a slope smaller than 5[4]. Note that in this particularly simple case, we make use neither of the input `T` nor any memory.

We use as oracle a direct translation in Lustre of Equation 1, and we observe that this oracle is violated on the first step. Fig. 7 displays a visualisation of the data produced by Lurette. The expected output of the converter node is displayed in the graphic in the variable `T_expected`. The expected output would be to see the two upper curves superimpose.

As a matter of fact, the bug was in the specification and not in the code. Indeed, in the definition of `T`, `R` should be `R-100.0`.

A second problem with this node was revealed by Lurette when we tried `R` values smaller than 100. Indeed, in such a case, `R-100` is negative, and $(\texttt{R-100})^2$ was computed by an exponentiation function of type $float \rightarrow float \rightarrow float$, which raises an exception when its first argument is negative – whereas it should probably have used a exponentiation function of type $float \rightarrow int \rightarrow float$ which makes sense even if the first argument is negative.

Of course, this sub-program is not representative of typical reactive systems as it involves no feedback. However, it let us illustrate the use of Lurette smoothly. Moreover, it has allowed Hispano-Suiza to discover two (simple) bugs in no time, as Lurette is able to generate the simple environment of Fig. 6 automatically. Indeed, no sophisticated/stressful environment is necessary to find such kind of bugs in such kind of programs.

[4] It is not particularly meaningful to bound the derivative of the input for such a combinatorial program; we did it here only because it makes the visualisation of data easier on Fig. 7.

5 Case study 2: computing a propulsion nozzle position

This second case study has also been provided by Hispano-Suiza. The task of this component is to compute the position of a propulsion nozzle according to the values of two sensors that measure electric tension. Lurette allowed us to discover one bug in a preliminary and unvalidated version of the code. This bug has already been corrected when we signaled it to the Hispano-Suiza production team.

5.1 The specification of the component

The propulsion nozzle position Scade node has four inputs: `U1` and `U2`, which are real values that come from sensors; `VU1` and `VU2` which are Boolean values that state whether the tensions `U1` and `U2` are valid. It has two outputs: `X`, a real value that indicates the nozzle position; and `VX`, a Boolean value that states whether the nozzle position is valid or not.

The specification of that component says that the output `VX` ought to be true if and only if the following equation holds:

$$\begin{aligned} &\texttt{VU1} \wedge \texttt{VU2} \wedge (1 \leq \texttt{U1} \leq 5) \wedge \ (1 \leq \texttt{U2} \leq 5) \\ &\wedge (4 \leq \texttt{U1} + \texttt{U2} \leq 8) \wedge \ X = f(\texttt{U1}, \texttt{U2}) \end{aligned} \qquad (2)$$

where f is a deterministic function of `U1` and `U2` that we do not provide here.

5.2 A possible test session

From this specification, there are numerous ways we can use Lurette for an automatic test session. A first extreme way would be to put Equation 2 both in the oracle and in the SUT environment. But then, we would never see `VX` becoming false, e.g., when `U1+U2` is smaller than 4.

Another way would be to put no constraint at all in the environment and thus to generate completely random input values for the SUT, and to put equation 2 in the oracle only. But then, the probability of getting "interesting" values (i.e., around the interval [0; 10]) would be very low.

Therefore, the environment we propose in Fig. 8 is somewhere between those two extreme solutions: `U1` and `U2` vary between 0 and 5 using the macro of Fig. 2.

```
inputs { X : real ; VX : bool }
outputs { U1,U2:real ~init 1.8 ; VU1,VU2:bool }
nodes { 0 : stable }
start_node { 0 }
transitions { 0 -> 0 ~cond
              abs(U2 - U1) < 0.1
              and up_and_down(U1, 0.0, 5.0, 0.1)
              and up_and_down(U2, 0.0, 5.0, 0.1)    }
```

Fig. 8. A Lucky program modelling the nozzle environment.

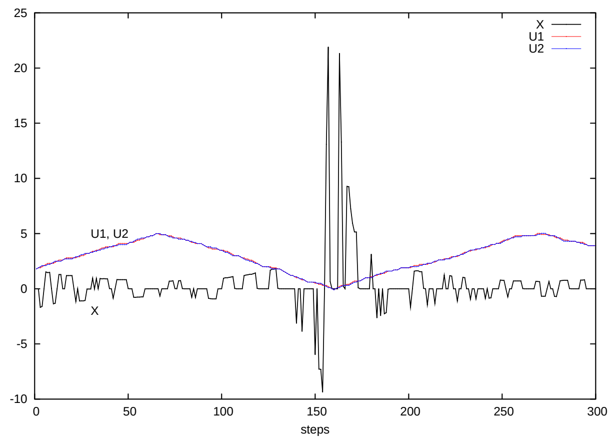


Fig. 9. The timing diagram of an execution of the nozzle.

In order to make the environment more realistic, we add a constraint that enforces `U1` and `U2` to be close: `abs(U2-U1)<0.1`. Moreover, `VU1` and `VU2` are left unconstrained. The **init** option is used to set the previous values of `U1` and `U2` at the first instant. The oracle is again just a straightforward translation in Lustre of equation 2.

The timing diagram of a Lurette run with this environment is shown in Fig. 9. Note that the oracle was really useful in deciding automatically whether the tests succeeded or not. Indeed, for very long sequences[5], performing the test decision manually (i.e., by data file inspection) would be very tedious.

A preliminary and unvalidated version of this program violated the oracle. The bug was that the equation 2 was encoded in Scade with `or` instead of `and` gates. The timing diagram of Fig. 9 has been generated with a corrected version of the component.

5.3 Other possible test sessions

Of course, several other test scenarios can be useful. For instance, one could enforce `VU1` and `VU2` to be always true, so that the checking of the result of the numeric function f is done at each step. One could also play with the tension slopes, or let the two tensions evolve independently.

It is precisely the point of having the flexibility of a plain programming language: be able to tackle the diversity of all the possible situations. In the next case study, we illustrate how one can write more sophisticated environments.

[5] For this kind of environment, Lurette can generate several thousands of test vectors per second.

6 Case study 3: a fault tolerant heater

This case study in not an industrial application, but has been inspired from a real one. We believe it is representative of what testing a fault tolerant controller could be. It lets us illustrate several aspects of the use of Lurette, and in particular how simple environments can be defined in Lucky, and then refined.

6.1 A fault-tolerant heater controller

We want to test a fault-tolerant heater controller which has three sensors (namely, three real inputs) measuring the temperature in a room, and which returns a Boolean value indicating to the heater whether it should heat or not. We only provide its informal specification, which is enough from the Lurette black-box testing point of view. The full Lustre code for this controller can be found in Appendix A.

The main task of the controller is to perform a vote to guess what the temperature is. Then, if that guessed temperature is smaller than a minimum value (`TMIN`), it heats; if it is bigger than a maximum value (`TMAX`), it does not heat; otherwise, it keeps its previous state. The voting works as follows: the values of each sensor is compared pairwise, and two sensors are considered suspicious as soon as they differ by a threshold value (`DELTA`).

```
V12 = abs(T1-T2) < DELTA;
V13 = abs(T1-T3) < DELTA;
V23 = abs(T2-T3) < DELTA;
```

Hence, there are four cases, depending on the values of `V12`, `V13`, and `V23`.

1. If the three comparisons are true, it returns the median value of the three sensors;
2. If only one comparison is false, it considers it as a false alarm (e.g., because `DELTA` was too small) and still returns the median value.
3. If two comparisons are false (say `V12` and `V13`), it deduces the broken sensor (`T1`) and returns the average of the other two (`T2+T3/2.0`);
4. If the three comparisons are false, it is difficult to know whether two or three sensors are broken, and it (safely) stops to heat in that case.

```
inputs { Heat:bool }
outputs { T1, T2, T3 : real ;
          T:real ~min 0.0 ~max 50.0 ~init 7}
locals {eps1,eps1,eps3:real ~min -0.3 ~max 0.3}
nodes { 0 : stable }
start_node { 0 }
transitions { 0 -> 0 ~cond
      T = pre T + (if Heat then 0.2 else -0.2)
      and T1 = T + eps1 and T2 = T + eps2
      and T3 = T + eps3 }
```

Fig. 10. A Lucky program modelling undegradable sensors.

6.2 A test session using undegradable sensors

In order to test that program, there are two things we need to simulate: the real temperature in the room, and the sensors that measure that temperature.

The Lucky program provided in Fig. 10 has one input variable (the output of the SUT): the Boolean `Heat` which is true iff the heater is heating. It has four output variables (the inputs of the SUT): the true temperature in the room `T`, as well as the temperature as it is measured by the 3 sensors: `T1`, `T2`, and `T3`. It also have three local variables (`eps1`, `eps2`, and `eps3`) that are uniformly drawn between -0.3 and 0.3 (the **min** and the **max** options in the variable declaration are syntax that lets one define global constraints). Those local variables are used to disturb the value of the temperature `T` and simulate the noise a sensor may have (`T1 = T + eps1`).

We then need to simulate `T`. `T` is initialized to `7.0` via the **init** option. A single transition updates `T` as follows: if `Heat` is true, then `T` is incremented by 0.2; otherwise, it is decremented of 0.2. This model is quite simple, but it will be refined further later.

A priori, the real temperature could be a local variable of the SUT environment. However, in order to write oracles that have access to that temperature, we need to add it to the SUT interface. That is the reason why the controller (cf node `heater_control` in Appendix A) has an additional input `T`, which it does not use.

A Lurette run using the Lucky program of Fig. 10 produced the timing diagram shown in Fig. 11. There, we can convince ourselves that everything seems to work fine; the temperature increases and `Heat_on` is true until `TMAX` is reached. At step 11, `Heat_on` becomes false and the temperature decreases until `TMIN` is reached, and so on.

6.3 The test oracle

The property that we propose to check is that the temperature in the room never becomes bigger than `TMAX` even if all sensors are broken. This safety property is encoded by the Lustre observer of Fig. 12. This Lustre *node* called `not_a_sauna`, takes as input the input and

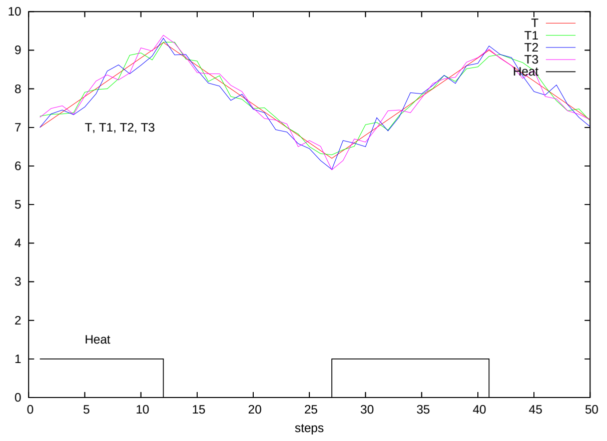


Fig. 11. The timing diagram of an execution generated with the undegradable sensors.

```
node not_a_sauna(T,T1,T2,T3: real; Heat_on: bool)
returns (ok:bool);
  let
     ok = true -> pre T <= TMAX;
  tel
```

Fig. 12. A possible oracle: make sure that temperature never becomes too hot.

output flows of the SUT (the reals `T`, `T1`, `T2` and `T3` as well as the Boolean `Heat_on`). It returns a single boolean (`ok`).

The core of the program (between **let** and **tel**) contains the *equation* defining the values of ok:

- at the very first call (right hand side of the `->` operator) the value is true.
- for any other calls, the value is given by the left hand-part of `->`: ok is true if and only if the `pre`(vious) value of T is less or equal than `TMAX`.

Note that the only "observed" variable is actually `T`. However, all other useless input/output variables must appear in the profile in order to provide the profile expected by Lurette.

If we run again our program, we observe that indeed this oracle is never violated.

6.4 A test session using degradable sensors

The Lucky program of Fig. 13 models more realistic sensors that can degrade. The input, output, as well as the `epsi` local variables are the same as in the Lucky program of Fig. 10 – we have omitted them from the figure for the sake of conciseness.

There are two additional local variables: `cpt`, that is incremented at each cycle, and `INV`, an invariant that states how the temperature `T` is simulated (basically as in Fig. 10) and how to update `cpt` at each cycle.

The two transitions `s1 -> t1, t1 -> s1` describe exactly the same kind of behavior as transition `1 -> 1`

```
locals { cpt: int; eps: real ~min 0.0 ~max 0.2;
   -- Invariant
   INV: bool ~alias cpt = pre cpt+1
         and T = pre T + (if Heat then eps else -eps)
}
nodes { t1, t2, t3, t4 : tran-
sient; s1, s2, s3, s4 : stable }
start_node  { t1 }
transitions {
-- No sensor is broken
  t1 -> s1 ~cond INV and T1=T+eps1 and T2=T+eps2
                      and T3=T+eps3;
  s1 -> t1 ~weight 1000;
  s1 -> t2 ~weight pre cpt;
-- One sensor is broken
  t2 -> s2 ~cond INV and T1=T+eps1 and T2=T+eps2
                      and T3 = pre T3;
  s2 -> t2 ~weight 1000;
  s2 -> t3 ~weight pre cpt;
-- Two sensors are broken
  t3 -> s3 ~cond INV and T1=T+eps1 and T2 = pre T2
                      and T3 = pre T3;
  s3 -> t3 ~weight 1000;
  s3 -> t4 ~weight pre cpt;
-- Three sensors are broken
  t4 -> s4 ~cond cpt = 0 and T = pre T
             and T1 = pre T1 and T2 = pre T2
                      and T3 = pre T3;
  -- Start again from the beginning
  s4 -> t1 }
```

Fig. 13. A Lucky program modelling degradable sensors.

in Fig. 10: T1, T2, and T3 are computed as disturbed versions of T. Transitions `t2 -> s2, s2 -> t2` simulate the case where one sensor is broken: T3 keeps its previous value (`pre T3`) whatever the temperature. Transitions `t3 -> s3, s3 -> t3` and transitions `t4 -> s4, s4 -> t4` respectively simulate cases where respectively two and three sensors are broken.

Let us detail the execution of that automaton. The initial node is the one labelled by `t1`. The output values for the first cycle are given by the equation that labels the transition `t1 -> s1`, which states that outputs T, T1, T2, and T3, are set to `7.0`, and the local counter `cpt` is set to `0`.

The values for the second cycle are computed via one of the two transitions outgoing from node `s1`: `s1 -> t1`, which is labelled by `1000`, and `s1 -> t2` which is labelled by `pre cpt`. The meaning of those weights is the following: use the first transition with a probability of $\frac{1000}{1000+pre\ cpt}$ and the second one with a probability of $\frac{pre\ cpt}{1000+pre\ cpt}$. At the second cycle, since `pre cpt` is bound to `0`, the only possible transition is `s1 -> t1`, which leads to a correct behavior of all sensors. Since `t1` is transient, `t1 -> s1` is also used to compute the output of the current cycle.

At the third cycle, the situation is roughly the same: `s1` is the current node, but the transition `s1 -> t2` is now possible, with a probability of $\frac{1}{1001}$. If this transition is chosen, we enter in a mode where one sensor is broken. Note that as time progresses, the probability of going to node $t2$ increases; this models the situation where the probability of failure increases with time. The behavior is similar at stable nodes `s2` and `s3`. When all sensors are broken, we go back to the initial state and start a new test session ($cpt = 0$).

If we launch a Lurette run with the program of Fig. 13 often enough or with a test length that is long enough, we can exhibit sequences that violate the oracle. An example of such a sequence is displayed in the timing diagram of Fig. 14. One can see at step 20 the first sensor breakdown (it keeps its previous value), at step 45 the second sensor breakdown, and at step 55 the last sensor breakdown. Then, at step 56, a new session is launched (namely, the automaton control go back to node `t1`). The oracle violation occurred at the fifth session at step 346.

6.5 The bug explanation

This time, the bug is in the specification itself[6]. We modelled sensor breakdowns by making them keep their previous value – which is questionable. Therefore, if ever two sensors broke down with similar values, the voter will not be able to realize that they are broken. Hence, the controller keeps on heating forever, which violates the oracle.

Note that such a configuration is not very probable, hence the need for being able to generate values fast enough to have a chance to detect it. The timing diagram of Fig. 14 was generated in less than 2 seconds on a Pentium 4 clocked at 3.00GHz, with 512 KB of RAM.

One way to correct that bug would be to check that sensor values do change during a given number of cycles, and to consider them, at least temporarily, invalid. Note however that in general, Fault-tolerant systems vendors only promise that their systems work under certain hypotheses made on their environments. For example, there is nothing that can be guaranteed for a controller if all its sensors agree to provide wrong values.

7 Case study 4: a brake-by-wire system

The last case study is a brake-by-wire system, provided by Renault as a Sildex model. The objective of this controller is to lower the influence of external conditions on the performance of the braking system. Those external

[6] Note that this specification was inspired by another (confidential) case study. That bug was introduced (unintentionally at first!) by us, for didactic purposes.

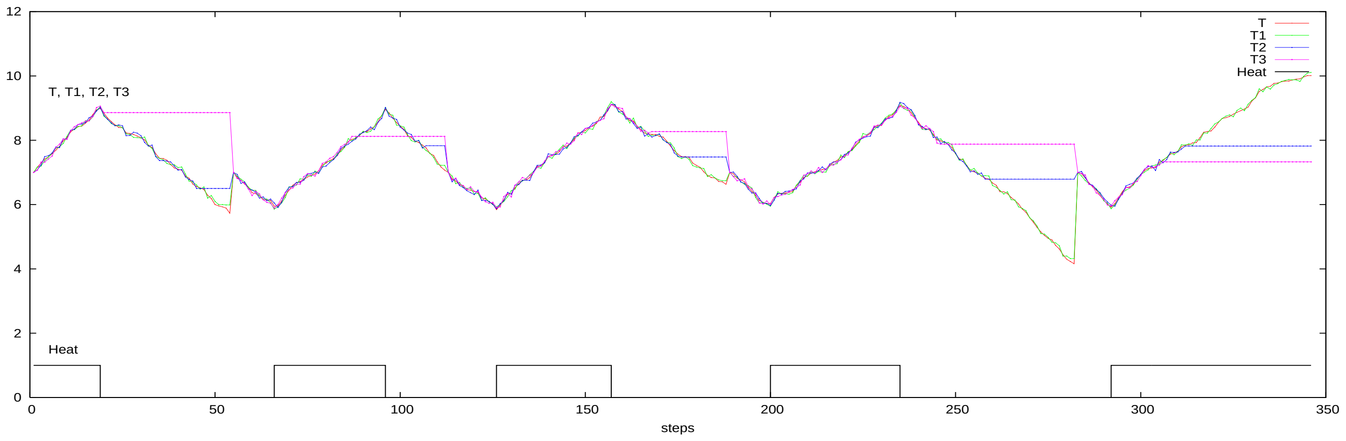


Fig. 14. The timing diagram of an execution generated with degradable sensors exhibiting a test failure (the temperature exceeded 10) at step 346.

conditions can be, for instance, the slope of the road, the brake pads waste, or the vehicle weight.

This controller makes extensive use of non-linear mathematical functions, and is therefore untractable for validation tools performing exhaustive formal verifications. It is therefore a good candidate for Lurette.

7.1 The system under test

In order to test the controller at the software level, we need a model of the vehicle. Renault has kindly provided us with such a model (in Sildex). Of course, the accuracy of the testing process depends on the quality of this model.

In this article, for the sake of simplicity and without loss of generality, the only external condition we take into account is the slope of the road. All other parameters are set to constant values. The interface of the SUT is therefore the following.

- It has four inputs:
  - two reals, α_{brake} and α_{accel} that carry the brake and the accelerator pedal angles;
  - one Boolean, `clutch_pedal_on`, that indicates if the clutch pedal is pressed;
  - and one integer, `gear_lever`, that carries the gear level.
- It has 2 real outputs, `Speed` and `Accel`, which respectively carry the speed and the acceleration of the vehicle.

Note that in Fig. 15, only the brake pedal angle is displayed, but the gear-box commands and the accelerator pedal angle are also sent to the SUT and the oracle.

7.2 The oracle

We want to verify that, given the same braking request from the driver, the vehicle deceleration is almost the same whatever the slope is. In order to do that, we use an instance of the SUT model to build the oracle. This instance serves as a reference: it receives the same pedal angle as the actual SUT, but a constant slope of 0. In other terms, the reference vehicle is running on a perfectly flat road, while the SUT vehicle is running through hills. The oracle compares the deceleration of the two vehicles and check that they are not too different (see Fig. 15).

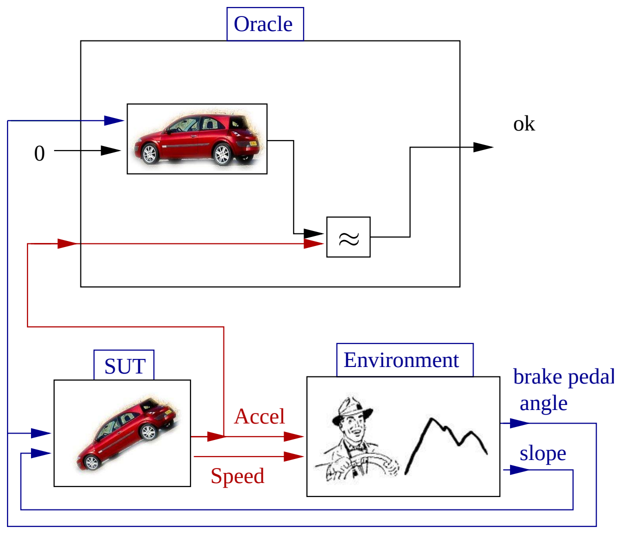
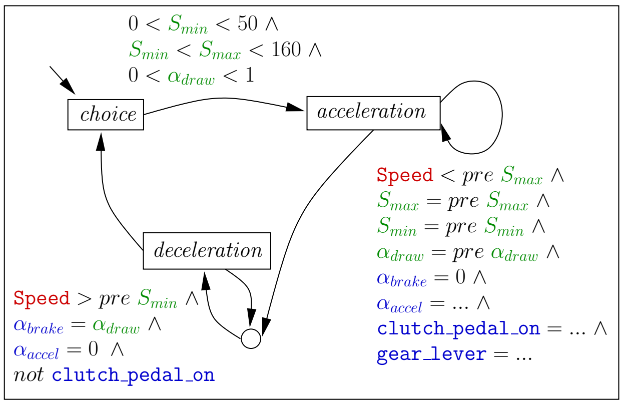


Fig. 15. The brake-by-wire testing data-flow.

7.3 The environment

The environment is shown in Fig. 16. The node labelled by *choice* is the initial state. From that state, the only possible transition is labelled by a constraint that draws values for the 3 local variables: α_{draw} which is the angle that will be used in the deceleration phase; S_{max} which is the speed at which we start braking, and S_{min}, the speed at which we stop braking.


$0 < S_{min} < 50 \wedge$
$S_{min} < S_{max} < 160 \wedge$
$0 < \alpha_{draw} < 1$
choice
acceleration
deceleration
Speed $<$ *pre* $S_{max} \wedge$
$S_{max} =$ *pre* $S_{max} \wedge$
$S_{min} =$ *pre* $S_{min} \wedge$
$\alpha_{draw} =$ *pre* $\alpha_{draw} \wedge$
$\alpha_{brake} = 0 \wedge$
$\alpha_{accel} = ... \wedge$
clutch_pedal_on = ... $\wedge$
gear_lever = ...
Speed $>$ *pre* $S_{min} \wedge$
$\alpha_{brake} = \alpha_{draw} \wedge$
$\alpha_{accel} = 0 \wedge$
not clutch_pedal_on

inputs = { `Speed`, `Accel`: int }
outputs = { α_{brake}, α_{accel}:float ; `clutch_pedal_on`:bool ; `gear_lever`:int }
locals = { α_{draw}, S_{min}, S_{max}: float }

Fig. 16. The brake-by-wire environment.

In the *acceleration* mode, the locals variables keep their previous values, the brake pedal angle α_{brake} is set to 0, the gear-box and the accelerator pedal are set appropriately to accelerate. For the sake of clarity, the corresponding constraints are not fully represented on the figure. Intuitively, the desired speed is reached by using one stable state per gear level, or, in a less distinguished manner, using only the fifth level and the maximal value of the accelerator.

When the speed exceeds S_{max}, the transition that loops to the *acceleration* state is no more satisfiable, and the control moves to the *deceleration* mode. In this mode, the clutch pedal is pressed, and the brake pedal angle α_{brake} is set to α_{draw}. This is done in loop until `Speed` is smaller than S_{min}; then another choice for the test parameters is done.

Note that unlike the fault-tolerant heater of the previous section, this automaton contains no weight at all. The control is entirely guided by the environment input `Speed`.

The slope is computed in an other Lucky automaton run in parallel (cf. Section 3.4), and that is made of a single state and a single transition labelled by the constraint up_and_down(`slope`,-30,30,1).

On a Pentium 4 clocked at 3.00 GHz, with 512 MB of RAM, a million of cycles can be generated in 2 minutes and 20 seconds (the time to perform the SUT and the oracle cycles included).

8 Conclusion and further work

8.1 Conclusion

Lurette has turned out to able to analyze several designs of real industrial applications and likely to discover errors before the designs were subjected to module testing. This article reports how Lurette let us detect three of them in an application provided by Hispano-Suiza. One interesting point is that those bugs were found with little effort. However, the situation was particularly favorable, since the application was in the development stage, and as a consequence more likely to contain errors.

This article also illustrates the use of Lucky, a new language to describe and simulate stochastic machines. It shows how simple environments can be quickly defined, and then how they can be refined into more accurate and complex ones. It also demonstrates that, even if Lurette targets reactive systems, it can be used to test purely combinational programs.

Note that we insist on the language expressiveness to allow the description of realistic environments. But of course, the tool can be used with different motivations. One can use it to stress the SUT in arbitrary manners, for instance by trying limit values.

Some further work concerns the weakening of Lurette's black box hypothesis, via the use of verification tools. The idea is the following: the abstract interpretation verification tool Nbac [19] provides semi-decision results: if a property is shown to be true, it is for sure; but otherwise, the (false) negative answers might be due to some approximations performed by tool. In such a case, Nbac returns an abstract automaton, for which it is impossible to know whether a concrete path from the initial to the final node exists (if we knew it, we would have solved an undecidable problem). Nbac is already able to output this abstract automaton in the Lucky format [11], which can then be used to try to find (randomly) a concrete path in it. Some more work and experimentation are required.

A second step would then to be able to translate Lucky descriptions (forgetting the weight annotations) of the environment into a format the verification tool can handle. Indeed, the scheme we use in Lurette is the same as in verification: a formal description of the property (in our case, the oracle) is checked against the program using some formal hypotheses made on its environment. In our case, Nbac could be tried before launching a Lurette test session. If the proof failed, one could then use in Lurette the result of Nbac in combination with the environment to perform a testing session that is oriented towards the violation of the oracle.

8.2 Code coverage for data-flow languages

Another important point that has not been addressed yet is to have a suitable notion of code coverage for synchronous data-flow languages such as Lustre. Indeed, data-flow languages are very different from sequential ones, for which it is easier to define coverage metrics based on the control structures.

Acknowledgements. This work was partially supported by the SAFEAIR II IST-2001-34563. Thanks to Thierry Gaudre, Hugo Chale-Gongora, and Nicolas François from Renault, for providing the brake-by-wire case study and for helping in setting up its Lurette session. Romain Ballet, during a training period at Hispano-Suiza, helped enthusiasticly to set up the Hispano-Suiza case studies.

We are also very grateful to Norman Scaife for correcting a lot of English errors in a preliminary version of this article.

References

1. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, sep 1991.
2. Albert Benveniste. Constructive probability and the SIGNalea language : building and processes via programming. Technical Report RR-1532, INRIA, 1991.
3. Marco Bernardo, Lorenzo Donatiello, and Paolo Ciancarini. Stochastic process algebra: From an algebraic formalism to an architectural description language. *Lecture Notes in Computer Science*, 2459:236–260, 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. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. A. Bouali. Xeve: an Esterel verification environment. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.
7. L. Bousquet, F. Ouabdesselam, J. Richier, and N. Zuanon. Lutess: testing environment for synchronous software, 1998.
8. C. Derman. *Finite State Markovian Decision Processes.* Academic Press, 1970.
9. Bernard Dion. Correct-by-construction methods for the development of safety-critical applications, 2003.
10. Jean-Claude Fernandez, Claude Jard, Thierry Jeron, and Cesar Viho. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.
11. F. Gaucher, E. Jahier, F. Maraninchi, and B. Jeannet. Automatic state reaching for debugging reactive programs. In *AADEBUG, Fifth Int. Workshop on Automated and Algorithmic Debugging.* HAL - CCSd - CNRS, November 14 2003.
12. T. Gauthier, P. Le Guernic, and L. Besnard. Signal, a declarative language for synchronous programming of real-time systems. In *Proc. 3rd. Conf. on Functional Programming Languages and Computer Architecture.* LNCS 274, Springer Verlag, 1987.
13. 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.
14. 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.
15. 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*, pages 785–793, September 1992.
16. 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.
17. L. Jategaonkar Jagadeesan, A. A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments (experience report). In *International Conference on Software Engineering*, pages 525–535, 1997.
18. E. Jahier. The Lurette V2 User guide. Technical Report TR-2004-5, Verimag, 2004. www-verimag.imag.fr/∼synchron/tools.html.
19. B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design*, 2001. 40 pages.
20. B. Jeannet. *The Polka Convex Polyhedra library Edition 2.0*, May 2002. www.irisa.fr/prive/bjeannet/newpolka.html.
21. 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.
22. Bengt Jonsson, Kim G. Larsen, and Wang Yi. *Probabilistic Extensions of Process Algebras*, chapter the Handbook of Process Algebras, pages 685–710. Elsevier, North Holland, 2001.
23. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.
24. P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. Signal , a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986.
25. Nancy A. Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
26. M. Müllerburg, L. Holenderski, and O. Maffeis. Systematic testing and formal verification to validate reactive programs. *Software Quality Journal*, 4(4), 1995.
27. P. Raymond and Y. Roux. Describing non-deterministic reactive systems by means of regular expressions. In *First Workshop on Synchronous Languages, Applications and Programming, SLAP'02*, Grenoble, April 2002.
28. P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.
29. F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0*, 1998.
30. P. Thevenod-Fosse, C. Mazuet, and Y. Crouzet. On statistical testing of synchronous data flow programs. In *1st European Dependable Computing Conference (EDCC-1)*, pages 250–67, Berlin, Germany, 1994.
31. 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.

A The Lustre code of the fault-tolerant heater

```
const FAILURE = - 999.0;
-- temperature when all sensors are broken
const TMIN = 6.0;
const TMAX = 9.0;
const DELTA = 0.5;
--------------------------------------------
node heater_control(T, T1, T2, T3 : real)
returns (Heat:bool);
var
 V12,  V13,  V23 : bool;
 Tguess : real;
let
  V12 = abs(T1-T2) < DELTA;
  V13 = abs(T1-T3) < DELTA;
  V23 = abs(T2-T3) < DELTA;
  Tguess =
   if noneoftree(V12, V13, V23)
        then FAILURE
   else if oneoftree(V12, V13, V23)
        then Median(T1, T2, T3)
   else if alloftree(V12, V13, V23)
       then Median(T1, T2, T3)
-- 2 among V1, V2, V3 are false, one is true
   else if V12
       then Average(T1, T2)
   else if V13
       then Average(T1, T3)
   else
-- V23 is necessarily true, hence T1 is wrong
       Average(T2, T3) ;
  Heat = true ->
   if Tguess = FAILURE then false else
   if Tguess < TMIN then true  else
   if Tguess > TMAX then false else pre Heat;
tel
--------------------------------------------
node Average(a, b: real)
returns (z : real);
let
  z = (a+b)/2.0 ;
tel

node Median(a, b, c : real)
returns (z : real);
let
  z = a + b + c - min2 (a, min2(b,c))
                - max2 (a, max2(b,c));
tel

node noneoftree (f1, f2, f3 : bool)
returns (r : bool)
let
  r = not f1 and not f2 and not f3 ;
tel
```

```
node alloftree (f1, f2, f3 : bool)
returns (r : bool)
let
  r =  f1 and f2 and f3 ;
tel

node oneoftree (f1, f2, f3 : bool)
returns (r : bool)
let
   r = f1 and not f2 and not f3  or
   f2 and not f1 and not f3  or
   f3 and not f1 and not f2 ;
tel
--------------------------------------------
-- The oracle
node not_a_sauna(T, T1, T2, T3 : real;
             Heat: bool)
returns (ok:bool);
let
  ok = true -> pre T < TMAX + 1.0;
tel
```