Case Studies with Lurette V2

Erwan Jahier, Pascal Raymond, Philippe Baufreton

Case Studies with Lurette V2[⋆]

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

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

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 reimplemented 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 dedicated language called *Lucky*, a language for programming non-deterministic reactive machines.
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 the SUT environment.

Keywords : 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. They require to be strongly validated before being used.

Formal exhaustive validation methods like model-checking 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 in general 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 raises specific problems. The most obvious is that the execution of such 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 random environment, and the properties of the environment 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

[⋆] This work was partially supported by the SAFEAIR II IST-2001-34563.

in general 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 of the system under test (glass-box) [TMC94], and/or do not deal with numerical programs [MHM95,BORZ98,FJJV97,JPP$^+$97]. Another difference with our work and the so-called model-based approach [FJJV97,JPP$^+$97] is the following. The model-based approach supposes that 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 (model-checking, partial orders, bisimulation) are then applied to derive expected traces to be compared with actual traces, produced by the SUT.

On the other hand, our objective is less ambitious in some sense (and more ambitious in the sense that we deal with numerical aspects): we do not suppose we have such a model of the SUT, and 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 such test case models.

In this work, we use a testing tool called Lurette [RWNH98] which is black-box oriented and able to treat numerical aspects. This tool supports Lustre [HCRP91], Scade[1] [Dio03] and Sildex[2] programs. It can also be easily extended to other languages like Esterel [BG92], 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 program is described and simulated.

After a brief recall of the Lurette principles, we introduce Lucky, a new language used to describe and simulate the environment . Then, we present three case studies that emphasize the main characteristics of the tool. Those examples are extracted (or just inspired, for confidential reasons) from an application developed in Scade, and provided by Hispano-Suiza in the framework of the IST project Safeair II.

– The first one is a resistance to temperature converter. 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 example.
– The second computes a propulsion nozzle position. It is still a combinational program, but slightly more complex. One bug was detected.
– The last is a typical example of fault-tolerant controller, where the feedback aspects are important. For this case study, we illustrate how simple environment models can be defined quickly, and then refined to more accurate models.

[1] Scade is an integrated programming environment based on the Lustre language, see www.esterel-technologies.com
[2] Sildex is an integrated programming environment based on the Signal language [LBBG86], see www.tni-world.com/sildex.asp

2 Lurette, an automated testing tool

We recall in this section the principles of Lurette [RWNH98]. More details about the new version of the tool can be found in the *Lurette V2 user guide* [Jah04].

Automatic generation of realistic inputs: the SUT Environment. The main challenge in automating the test process is the ability to generate *realistic* input sequences to feed the SUT. 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.

Note that realistic input sequences cannot be generated off-line, since the SUT generally influences the behavior of the environment it 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.

Basically, the Lurette input sequence generation engine is a linear constraint solver and drawer. The Boolean part of the solver is based on Bdd [Som98] and the numeric part is based on a convex Polyhedron Library [Jea02]. 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 non-deterministic machines; it is presented in Section 3.

Automatic test decision: the oracle. The second thing that needs to be automated is the test decision. To do that, we will use the technique *observers* used in verification [HLR93]. An *observer* is a program that returns exactly one Boolean variable. It lets one express any safety property [Lam77].

For Lurette, users therefore need to write a Lustre (or a Scade, or Sildex) program for which the inputs are the SUT inputs and outputs, and which outputs a single Boolean that is true if and only if the test vectors are correct w.r.t. a given temporal property.

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.

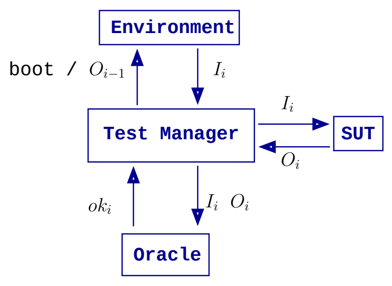


Fig. 1. The Lurette data flow loop.

The environment outputs serve as SUT inputs, and SUT outputs serve as environment inputs, apart from the first step. Therefore, in order 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), we specify that the environment start 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.

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 back the output vector O_1. Lurette then sends both I_1 and O_1 to the oracle. The oracle returns back a single Boolean ok_1, which is true if and only if I_1 and O_1 satisfy the property.

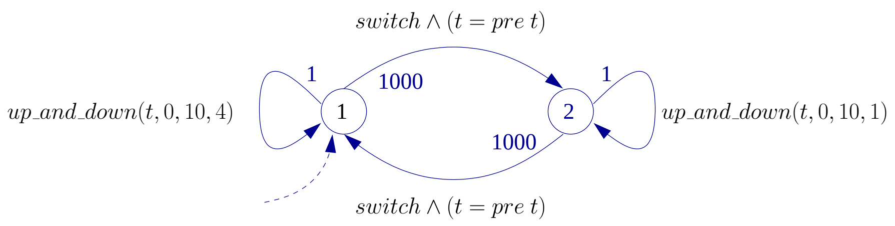
If ok_1 is false, then the testing process stops and a counter example that violates the property has been found. If ok_1 is true, then the testing session continues in exactly the same manner, except that this time, O_1 is sent to the environment, which returns yet another input vector I_2, and so on.

3 Lucky, a language to program non-deterministic machines

In the first Lurette prototype, the SUT environment behavior was described by Lustre observers made of a set of (linear) constraints over Boolean and numeric variables. The work of Lurette was to solve those constraints, and to draw a value among the solutions to produce one SUT input vector.

But, from a language point of view, Lustre is not very convenient, in particular for expressing sequences of different test scenarios, or to have some quantitative control over the probabilistic distribution of the solutions. It was precisely to overcome those limitations that a new language, Lucky, was designed.

A Lucky program is an interpreted automaton whose transitions define the machine reactions. Each transition is labelled by (1) a set of constraints (a relation) that defines the set of the possible outputs for one step, and (2) a weight that defines the relative probability of its transition being taken. We explain the operational semantics of Lucky on the automaton of Fig. 2, which has one Boolean input $switch$, and one real output t.


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

where: $up\_and\_down(X, Min, Max, Bound) = |X - pre\ X| < Bound \ \wedge$
$(\textbf{if}\ \ (pre\ X < Min)\ \ \vee\ \ ((pre\ X < Max) \wedge (pre\ pre\ X \leq pre\ X))$
$\textbf{then}\ (X > pre\ X)\ \ \textbf{else}\ \ (X < pre\ X))$

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

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 step of this synchronous machine.

– 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.
– 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 never 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 will increase (resp. decrease) at the current step, unless its previous value is bigger than 10 (resp. smaller than 0). See Fig. 4 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.

The control level. Suppose now that the current node is *1* (the behavior of this automaton is symmetric if the current node is 2). If the input $switch$ is false, then only one transition is possible: the one labelled by $up\_and\_down(t, 0, 10, 4)$. This constraint will therefore be solved, and one solution will be drawn among its set of solutions. If the input $switch$ is true, then the transition from 1 to 2 is also possible. One is labelled by a weight of 1, and the other by a weight of 1000; the latter will be drawn with a probability of $1000/1001$.

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.

Dynamic weights and transient nodes. Two other important concepts in Lucky are not described on this example, but will be illustrated later:

1. the concept of *dynamic weights*; weight labels can also be numerical functions of the inputs and the past-values. This can be particularly useful to model an *alive process* where the system has a known average life expectancy before breaking down; at each reaction, the probability working properly depends *numerically* on an internal counter of the process age.
2. and the concept of *transient* and *stable nodes*, which is simply a facility that help to structure Lucky programs better. A complete reaction is a sequence of transitions between two stable nodes, where all the intermediate nodes are transient. The constraint that is used to perform one reaction is the conjunction of the constraints labelling the transitions between the two stable nodes.

Restrictions. One restriction on the current version of Lucky is that the constraints on inputs, at a given point of a sequence, may only depend on the *past* values of outputs (as in Lustre). Another restriction is that numeric constraints on outputs should be linear. Please refer to the *Lucky language reference manual* [JR04] for more information.

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 user-friendly enough.

Anyhow, we designed another language, Lutin [RR02], 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 non-deterministic 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 node 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 case study 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.

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:

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

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

The test session. Fig. 3 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 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. 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. 4 displays a visualization 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`.

```
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. 3. A Lucky program modelling the converter environment.

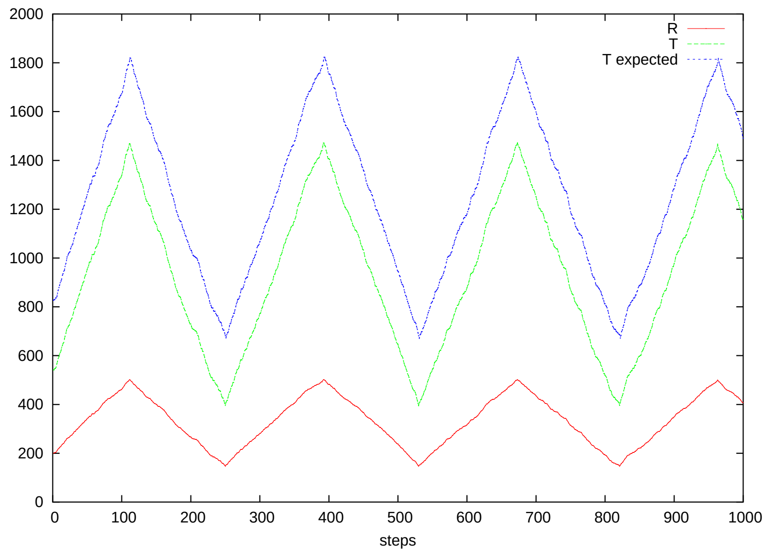


Fig. 4. The timing diagram of an execution of the converter.

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 `(R-100)`^{2} was computed by an exponentiation function of type $float \to float \to float$, which raises an exception when its first argument is negative – whereas it should probably have used a exponentiation function of type $float \to int \to float$ which makes sense even if the first argument is negative.

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.

```
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. 5. A Lucky program modelling the nozzle environment.

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` are which Boolean values that state whether the tensions `U1` and `U2` are valid. It has and two outputs: `X`, a real value that indicates the nozzle position; and `VX`, a Boolean value that states whether the nozzle position is valid.

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

$$\texttt{VU1} \wedge \texttt{VU2} \wedge (1 \leq \texttt{U1} \leq 5) \wedge (1 \leq \texttt{U2} \leq 5) \wedge (4 \leq \texttt{U1}+\texttt{U2} \leq 8) \wedge X = f(\texttt{U1}, \texttt{U2}) \quad (2)$$

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

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. 5 is somewhere between those two extreme solutions: `U1` and `U2` vary between 0 and 5. 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. 6. Note that the oracle was really useful in deciding automatically whether the tests succeeded or not. Indeed, for very long sequences[3], 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. 6 has been generated with a corrected version of the component.

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

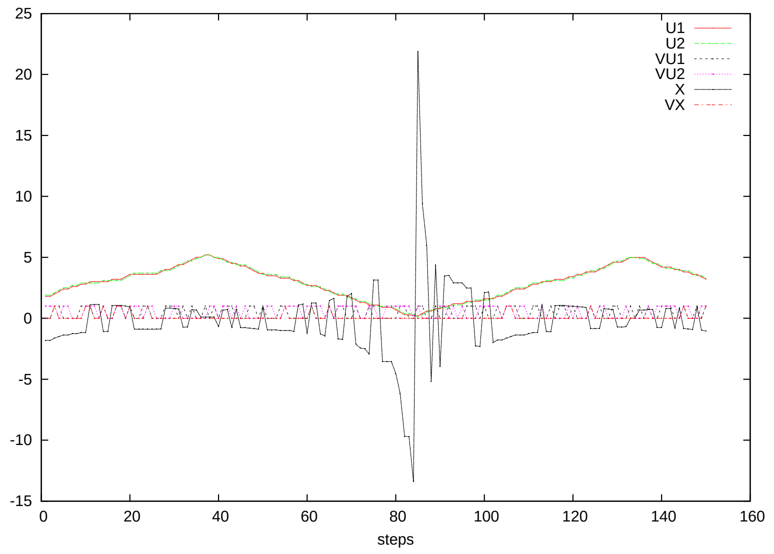


Fig. 6. The timing diagram of an execution of the nozzle.

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.

6 Case study 3: a fault tolerant heater

This case study does not come directly from 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.

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 [JR04].

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 decides not to heat in that case.

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. 7 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 -0.1 and 0.1 (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 has an additional input `T`, which it does not use.

A Lurette run using the Lucky program of Fig. 7 produced the timing diagram shown in Fig. 8. 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.

The test oracle. The property that we propose to check is described by the Lustre observer of Fig. 9 which states that the temperature should never be bigger than `TMAX+1` even if all sensors are broken. If we run again our program, we observe that indeed this oracle is never violated.

```
inputs { Heat:bool }
outputs { T1, T2, T3 : real ; T:real ~min 0.0 ~max 50.0 ~init 7}
locals { eps1, eps1, eps3 : real ~min -0.1 ~max 0.1 }
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. 7. A Lucky program modelling undegradable sensors.

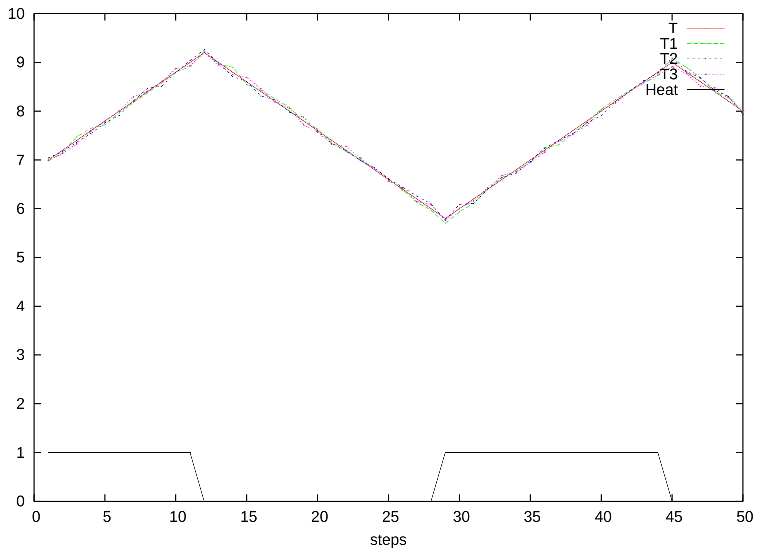


Fig. 8. The timing diagram of an execution generated with the undegradable sensors.

A test session using degradable sensors. The Lucky program of Fig. 10 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. 7 – 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. 7) 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` in Fig. 7: `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.

```
node not_a_sauna(T, T1, T2, T3 : real; Heat_on: bool)
returns (ok:bool);
  let
     ok = true -> pre T < TMAX + 1.0;
  tel
```

Fig. 9. A possible oracle: make sure that temperature never becomes too hot.

```
locals { cpt  : int; eps : real ~min 0.0 ~max 0.2;
   INV  : bool ~alias cpt = pre cpt+1 -- Invariant
                 and T = pre T + (if Heat then eps else -eps) }
nodes { t1, t2, t3, t4 : transient; 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. 10. A Lucky program modelling degradable sensors.

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

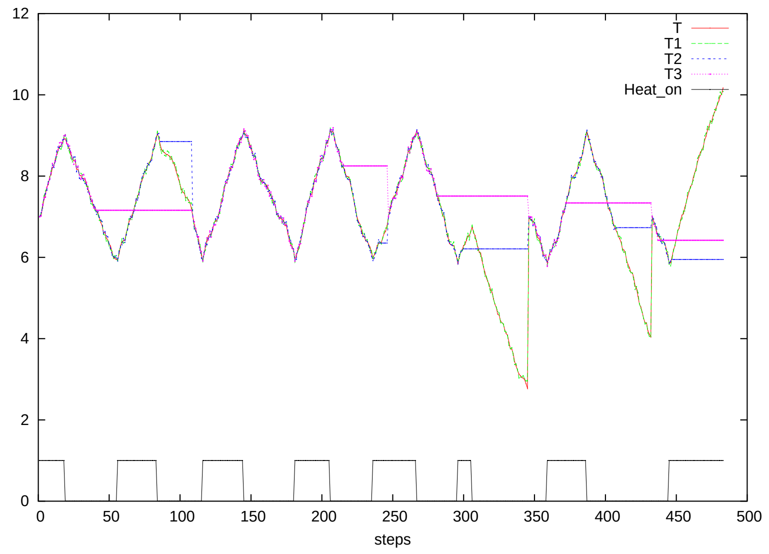


Fig. 11. The timing diagram of an execution generated with degradable sensors exhibiting test failure.

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. 10 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. 11.

The bug explanation. This time, the bug is in the specification itself[4]. 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. 11 was generated in less than 2 seconds on a Pentium 4 clocked at 3.00GHz, with 512 KB of RAM.

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

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.

7 Conclusion and further work

Conclusion. Lurette has turned out to able to analyze several designs extracted from 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 bugs 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 non-deterministic 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.

Glass-box testing with Lurette and verification tools. 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 [Jea01] 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 [GJMJ03], 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.

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.

References

BG92. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992. 2

BORZ98. L. Bousquet, F. Ouabdesselam, J. Richier, and N. Zuanon. Lutess: testing environment for synchronous software, 1998. 1

Dio03. Bernard Dion. Correct-by-construction methods for the development of safety-critical applications, 2003. 1

FJJV97. 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. 1

GJMJ03. F. Gaucher, E. Jahier, F. Maraninchi, and B. Jeannet. Automatic state reaching for debugging reactive programs. November 14 2003. 7

HCRP91. 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. 1

HLR93. 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. 2

Jah04. E. Jahier. The Lurette V2 User guide. Technical Report TR-2004-5, Verimag, 2004. www-verimag.imag.fr/∼synchron/tools.html. 2

Jea01. B. Jeannet. Dynamic partitioning in linear relation analysis. Application to the verification of reactive systems. *Formal Methods in System Design*, 2001. 40 pages. 7

Jea02. B. Jeannet. *The Polka Convex Polyhedra library Edition 2.0*, May 2002. www.irisa.fr/prive/bjeannet/newpolka.html. 2

JPP$^+$97. 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. 1

JR04. E. Jahier and P. Raymond. The Lucky Language Reference Manual. Technical Report TR-2004-6, Verimag, 2004. www-verimag.imag.fr/∼synchron/tools.html. 3, 6

Lam77. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. 2

LBBG86. 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. 2

MHM95. M. Müllerburg, L. Holenderski, and O. Maffeis. Systematic testing and formal verification to validate reactive programs. *Software Quality Journal*, 4(4), 1995. 1

RR02. 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. 3

RWNH98. 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. 1, 2

Som98. F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.0*, 1998. 2

TMC94. 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. 1