Environment-Model Based Testing of Control Systems: Case Studies *

Erwan Jahier[1], Simplice Djoko-Djoko[1], Chaouki Maiza[1], and Eric Lafont[2]

[1] VERIMAG-CNRS, Grenoble, France
[2] ATOS-WORLDGRID, Grenoble, France

Abstract. A reactive system reacts to an environment it tries to control. Lurette is a black-box testing tool for such closed-loop systems. It focuses on environment modeling using Lutin, a language designed to perform guided random exploration of the System Under Test (SUT) environment, taking into account the feedback. The test decision is automated using Lustre oracles resulting from the formalisation of functional requirements.
In this article, we report on experimentations conducted with Lurette on two industrial case studies. One deals with a dynamic system which simulates the behavior of the temperature and the pressure of a fluid in a pipe. The other one reports on how Lurette can be used to automate the processing of an existing test booklet of a Supervisory Control and Data Acquisition (SCADA) library module.

Keywords: Reactive systems, Control-command, Dynamic systems, scada, Test plans, Black-box testing, Requirements engineering, Synchronous Languages.

1 Introduction

Lurette is a black-box testing tool for reactive systems that automates the tests decision and the stimulation of the System Under Test (SUT). Lurette is based on two synchronous languages: Lustre [1], to specify test oracles, and Lutin [2], to model reactive environments. Lurette does not require to analyze the code, thus it can deal with any kind of reactive systems, as the experimentations reported below illustrate.

The COMON project* gathered three industrial companies that conceive control-command systems of nuclear plants. *Corys Tess* designs plant simulators used in particular for training operators. *Atos Worldgrid* designs software and hardware of computerized control rooms. *Rolls-Royce* designs the software and hardware of classified automatisms in charge of the plant security. The goal for this consortium was to take advantage of the partners complementarity to set up a development framework based on early simulations, model refinements, continuous integration, and automatic testing. During the project, the consortium has crafted a case study representative of each of the partners activity [3]. They also wanted to experiment on their own designs the Lurette languages and methodology. This article presents those experimentations.

* This work was supported by the COMON Minalogic project [2009-2012] funded by the french government (DGCIS/FUI), *la Metro*, and the city Grenoble – http://comon.minalogic.net/

We first recall Lurette principles in Section 2, and briefly present enough of the Lustre and the Lutin languages to be able to understand the examples. Section 3 presents the Corys case study and demonstrates the use of Lurette on a library object used to simulate the behavior of the temperature and the pressure of a fluid in a pipe. Section 4 presents the Atos case study that illustrates how Lurette can be used to automate the run of existing test plans designed for a Supervisory Control and Data Acquisition (SCADA) library object. We discuss related work and conclude in Sections 5 and 6.

2 Black-box Testing of Reactive Systems using Lustre and Lutin

Test of reactive systems. A *reactive system* is an combination of hardware or software (or both) that (1) acquires inputs (`set_inputs`), (2) performs a computation step (`step`), (3) and provides outputs (`get_outputs`). Testing a reactive system consists in writing or generating scripts that call the `set_inputs` and the `step` functions in turn. Such test scripts can be done offline, but a reactive system is meant to react to stimuli coming from its environment (e.g., from sensors), and to control it (e.g., using actuators). Thus a realistic test sequence should use `get_outputs` to provide input vectors that take into account the SUT inputs/outputs sequence history (i.e., its trace).

Stimulation. The environment is also a reactive system that executes in closed-loop with the SUT. It can be very versatile or underspecified. This motivated the design of Lutin [2], a language to program stochastic reactive systems and environment models.

Oracles. The test decision is deterministic and can be automated by formalizing the SUT expected properties via predicate over traces. A Lurette oracle is a program that returns as first output a Boolean that formalizes some requirements. Lurette reports a property violation each time one oracle returns false. As oracles for reactive systems often involve time, a language where time is a first-class concept like Lustre [1] is a legitimate choice. Moreover, Lustre can formalise any safety property [4].

Coverage. To decide when to stop generating tests, we use a notion of *requirements coverage*. Consider the requirement "`stable(X,30)` ⇒ `stable(Y,5)`" where the "`stable(V,n)`" predicate states if a variable `V` was stable during the last `n` seconds; it is always satisfied if `X` is never stable. But from a coverage point of view, it is more interesting to generate input sequences where `X` is stable. That is an example where random simulations are not sufficient and a language to program some guided scenarios is useful. Note also that if `X` is a SUT input, covering this requirement is easy. If `X` is an output, it is more complicated as it requests to drive the SUT, which sometimes require a deep expertise on its internals (but it is always the case when designing tests).

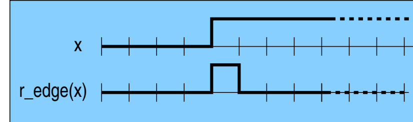
Lurette. Lurette handles the test harness, by reading test parameters, executing all the reactive systems in turn (SUT, environment, oracles), computing requirements coverage, and displaying a test report[3]. It does not impose the use of Lustre or Lutin. The reaction steps can be either time-triggered, event-triggered, or both. In the experimentations we report in this article, the SUT was time-triggered. More detailed presentations

[3] cf http://www-verimag.imag.fr/lurette.html for tools, manuals, and tutorials.

of Lurette can be found in [3,5]. We now present a few Lustre and Lutin programs to illustrate their main characteristics. We use those examples in the forthcoming sections.

Lustre. Lustre allows defining reactive programs via sets of data-flow equations that are virtually executed in parallel. Equations are structured into nodes. Nodes transform input sequences into output sequences. The Lustre node `r_edge` below processes one Boolean input sequence, and computes one Boolean output sequence.

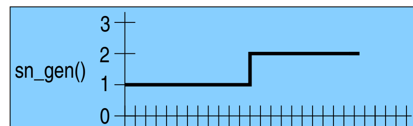
```
node r_edge(x:bool)returns(r:bool);
let
  r = x -> x and not pre(x);
tel
```



This node defines its output with one equation and four operators (i.e., predefined nodes). The memory operator "`pre`" gives access to the previous value in a sequence: if `x` holds the sequences (x_1,x_2,...), then `pre(x)` holds ($\perp$,x_1,x_2,...), where $\perp$ denotes an undefined value. The arrow operator "`->`" modifies the value of the first element of a sequence: if `x` holds (x_1,x_2,x_3,...), then `init->x` holds (init,x_2,x_3,...). This operator is useful for sequences that are undefined at their first instant, such as `pre(x)`. The "`and`" and "`not`" operators are the logical conjunction and negation lifted over sequences. Hence, `r_edge(x)` is equal to `x` at the first instant, and then is true if and only if `x` is true at the current instant and false at the previous one. This node detects rising edges.

Lutin. Lutin is a probabilistic extension of Lustre with an explicit control structure based on regular operators: sequence (`fby`, for "followed by"), Kleene star (`loop`), and choice (`|`). At each step, the Lutin interpreter (1) computes the set of reachable constraints, which depends on the current control-state; (2) removes from it unsatisfiable constraints, which depends on the current data-state (input and memories); (3) draws a constraint among the satisfiable ones (control-level non-determinism); (4) draws a point in the solution set of the constraint (data-level non-determinism). This chosen point defines the output for the current reaction. The solver of the current Lutin interpreter uses Binary Decision Diagrams (BDD) and convex polyhedron libraries [6]. It is thus able to deal with any combination of logical operators and linear constraints. Let us first illustrate the Lutin syntax and semantics with a program using equality constraints.

```
node sn_gen() returns (sn:int) =
 loop[10,20] sn=1 fby
 loop[20,30] sn=2
```



This node generates an integer finite sequence, without using any input. It first uses an atomic constraint that binds `sn` to 1, during between (uniformly) 10 and 20 reaction steps. Then it uses `sn=2` during between 20 and 30 steps, and then stops. A constraint can actually have any number of solutions, as in the `x_gen` node below.

```
node x_gen(i:real) returns (x:real) = loop { 0<x and x<i }
```

At each step, the elected constraint is simplified by constant propagation of inputs and memories values, and solved. Here, when `i` is negative, the constraint is not satisfiable and the program stops. Otherwise, one solution is drawn in the solution set `]0;i[`.

Lutin also has a notion of typed macro, which is useful to structure constraints.

```
let abs(z: real):real = if z < 0.0 then -z else z
let zone1(x,y:real):bool = abs(x+3.0*y) < 3.0 and abs(20.0*x-y+2.0)<5.0
let zone4(x,y:real):bool = abs(x-y+6.0) < 3.0 and abs(-5.0*x+y-2.0)<7.0
```

The first macro defines the absolute value of any real. The next ones define two zones where a couple of real values (x,y) evolve. We present below a last example (used later) that illustrates how to use Lutin to guide the random exploration of the environment.

```
node x_y_gen() returns (x,y:real) =
 loop { {|3: zone1(x,y) |1: zone4(x,y)} fby loop~50:5 x=pre x and y=pre y }
```

For the first reaction, a point is drawn in `zone1` with a probability of 3/(3+1)=0.75 or in `zone4` with a probability of 1/(3+1)=0.25. Then `x` and `y` keep their previous values for 50 steps in average, with a standard deviation of 5. This process then starts again thanks to the outer `loop`. Preventing the environment outputs to change at each reaction produces better coverage for requirements guarded by stability conditions (which is common in control-command applications). More generally, a too chaotic environment might set the SUT into degraded modes, which would prevent the test of nominal modes. Lutin also has constructs to execute in parallel nodes (`run`) or constraints (`&>`), as well as exceptions [4].

3 Automatic Testing of an Alices Library Object

In this section we report on a case study provided by Corys, a 300 persons company that develops and commercializes the Alices workbench. Alices is a data-flow graphical programming language tool for modeling, simulating and analyzing dynamic systems in the domain of energy and transportation. Simulators of energy production plants implemented in Alices are typically used to train operators.

3.1 The SUT: the `Node_Liquid_SPL` Alices object

Corys asked us to test one of their most frequently used library object, which is named `Node_Liquid_SPL`. This object simulates the behavior of the temperature and the pressure of a fluid in a pipe transporting homogeneous liquids through hydraulic networks. It is defined using mass and energy conservation equations:

$$\frac{dM}{dt} = \sum_i Q_{mi} \qquad\qquad \frac{dh}{dt} = \frac{\sum_i Q_{ei} - h \sum_i Q_{mi}}{M}$$

where $\sum_i \mathrm{Q_{mi}}$ and $\sum_i \mathrm{Q_{ei}}$ are respectively the sum of the mass flow (kg/s) and the sum of the powers arriving in the node; M and h are the mass (kg) and the mass enthalpy (J/kg) of the system; t is the time. The SUT is made of this object connected to two pipes, themselves connected to two objects (load loss) that models the fluid mass flow and transported power. The resulting equations are discretized and solved using the Newton-Raphson method. Table 1 describes the SUT input/output variables. We have shortened some variable names for the sake of readability.

[4] cf http://www-verimag.imag.fr/Lutin.html for more information.

Name	Producer	Meaning	Unit
Pin	Env.	Limit condition for input pressure	Pa
Pout	Env.	Limit condition for output pressure	Pa
Tin	Env.	Limit condition for input temperature	°c
Tout	Env.	Limit condition for output temperature	°c
T_amb	Env.	Temperature of the ambiant env	°c
Qe_amb	SUT	Power exchanged with the ambiant env	W
Qe1	SUT	Power exchanged with the first pipe	W
Qe2	SUT	Power exchanged with the second pipe	W
Qm1	SUT	Mass flow exchanged with the first pipe	kg/s
Qm2	SUT	Mass flow exchanged with the second pipe	kg/s
M	SUT	Mass of the system	kg
h	SUT	Mass enthalpy of the system	J/kg
T	SUT	Temperature of the system	°c

Table 1. Description of the SUT input/output variables

3.2 The SUT Environment

The input variables to stimulate this node are the limit conditions for the pressure (Pin and Pout), the temperature (Tin and Tout), and the ambient temperature (T_amb). The admissible values for those inputs are part of the object documentation, which states that pressure values vary within [10000.0, 190.0e5], and temperature values vary within [5.0, 365.0]. Moreover, Corys wanted to test this node in average conditions, and therefore required that the stimuli generator satisfies the following constraints:

- temperature and pressure cannot vary more than 10% between two instants;
- orders change only when mass and temperature values are stable (i.e., they do not change of more than 1% between two steps).

To stimulate the SUT, we therefore designed a Lutin program that is a direct formalization of the preceding constraints. We use the limit_der macro, which can be used both to test if an input varies more than a given percentage (limit_der(1.0,M) to test if M varies less than 1%), or to constraint the derivative of some output (limit_der(10.0,Pin) to constraint Pin to vary less than 10%).

```
let limit_der(pc:real; x:real ref):bool = abs(x-pre x) < abs(pc/100.0*pre x)
node liquid_spl_env(M, T: real) returns(
   Pin, Pout: real [10000.0; 190.0e5]; Tin,Tout,Tamb: real [5.0; 365.0];
) =
 -- a few aliases to make it more readable
let inputs_are_stable = limit_der(1.0,M) and limit_der(1.0,T) in
let dont_change = -- outputs keep their previous values
  Pin  = pre Pin  and Tin  = pre Tin  and
  Pout = pre Pout and Tout = pre Tout and Tamb = pre Tamb in
let change = -- outputs do not vary more than 10%
  limit_der(10.0,Pin) and limit_der(10.0,Pout) and
  limit_der(10.0,Tin) and limit_der(10.0,Tout) and limit_der(10.0,Tamb)
in -- a simple scenario
  true -- the first instant
  fby loop {if inputs_are_stable then change else dont_change}
```

The main node liquid_spl_env has two real inputs (produced by the SUT), and five real outputs. At the first instant, the only constraints on output variables are the

ones mentioned in their declarations; a random value is drawn in their respective interval domains. For example, `Tamb` is drawn between `5` and `365`. Then, for the remaining instants, variables keep their previous values if one of the environment input (`M` or `T`) varies more than `1%`; otherwise they vary at random, but without exceeding `10%`. One could of course imagine scenarios that are more complex. However, it hasn't been necessary to cover the expected properties we present in the following.

Note the feedback loop: the SUT reacts to its environment, which itself reacts to the SUT by testing the stability of `M` and `T`. This is typical of what offline test vectors generators cannot do when they ignore the reactive nature of the SUT.

3.3 The oracles

In order to automate the test decision, we need to formalize the SUT expected properties. Actually, such requirements were not explicitly written in the object documentation. Hence we asked to the Corys engineer responsible for the Alices library to write down how he expects this object to behave. He came up with the following requirements.

1. **if** the sum of powers (coming from `Qe1`, `Qe2`, and `Qe_amb` sensors), **and** the sum of incoming mass flows (coming from `Qm1` and `Qm2` sensors) are positive, **then** the mass **and** the temperature of the node increase;
2. **if** the sum of powers `Qe`, **and** the sum of flows mass `Qm` are negative, **then** the mass and temperature of the node decrease;
3. **if** the sum of powers is zero, **and** the sum of mass flow rate `Qm` is positive, **then** the mass increases;
4. **if** the sum of `Qe` is zero, **and** the sum of mass flow rate `Qm` is negative, **then** the mass decreases;
5. **if** the sum of mass flows `Qm` is zero, **and** the sum of powers `Qe` is negative, **then** the temperature decreases;
6. **if** the sum of mass flows `Qm` is zero, **and** the sum of powers `Qe` is positive, **then** the temperature increases.

A possible Lustre formalization of the first requirement is:

```
Qe = Qe1+Qe2+Qe_amb;
Qm = Qm1+Qm2;
ok1 = (Qe >= 0.0 and Qm >= 0.0) => (increase(M, 0.0) and increase(T, 0.0));
```

where `increase` is defined like that:

```
node increase(x: real; threshold: real) returns (y: bool);
let y = true -> (x-pre(x) >= threshold); tel
```

When we run Lurette with the SUT, the environment, and the oracles we described, all oracles are violated after a few steps. After several discussions with the person who wrote down the requirements, we ended up in Lurette runs that worked fine for hours. We now sum-up the fixes we needed to perform.

First problem. We have formalized the sentence “the sum of powers (coming from `Qe1`, `Qe2`, and `Qe_amb` sensors)”, and “the sum of mass flows (coming from `Qm1` and `Qm2`”)

as Qe=Qe1+Qe2+Qe_amb and Qm=Qm1+Qm2. However, the node connectors are oriented: the first pipe flows in, whereas the second pipe flows out. Hence the correct interpretation leads to the following definitions: Qe=Qe1-Qe2+Qe_amb and Qm=Qm1-Qm2.

Second problem. We have performed a bad interpretation of "are positive/negative" in the requirements. Indeed, when one compares to 0 a sum of values that are computed up to a certain precision (0.1 for flow mass, and 100 for powers), one has to specify some tolerance levels. Hence, for example, the second property should be rewritten as: "if Qe<=-Tol_Qe and Qm<=-Tol_Qm then the mass and temperature of the node decrease", where Tol_Qe=300 (three times the precision of power sensors) and Tol_Qm=0.2 (two times the precision of flow mass sensors).

Third problem. In properties 5 and 6, the statements "the sum of powers Qe is positive" should take into account the mass enthalpy of the node (Qe-h.Qm instead of just Qe).

Fourth problem. At this stage, the requirements fixes we have performed allow running simulations that last several minutes without violating oracles. After more steps (around 1000 steps in average), property 5 is violated. This time, the problem was more subtle and required a deeper investigation to the Corys engineer. His conclusion was that the convergence criteria (thresholds parametrizing the differential equation solver) in this simulation were too small. By setting a convergence criterion of 1 (versus 0.1) for the power, and of 1000 (versus 100) for the flow mass, no oracle is violated, even if we run the simulation for hours. Since the convergence criterion implies the precision of sensors computations, we need to modify again the values of Tol_Qm and Tol_P. Those new convergence criteria are actually the ones that are typically used in Alices for modeling pipes in power plants, which explain why this problem was (probably) never triggered before by Alices users.

Name	Unit	Meaning	Version 1	Version 2	Version 3	Involved Req.
Qm	kg/s	Sum of mass flow	Qm1+Qm2	Qm1-Qm2	ditto	all
Qe	W	Sum of powers	Qe1+Qe2+Qe_amb	Qe1-Qe2+Qe_amb	ditto	all
P	W	Node power	Qe	Qe-h.Qm	ditto	all
Tol_Qm	kg	Mass tolerance	0	0.2	2	3,4
Tol_P	W	Power tolerance	0	300	3000	1,2,5,6

Table 2. Summary of requirements fixes. Version 2 arises from the fixing of the first three problems. Version 3 arises from the fixing of the fourth problem.

3.4 Discussion and lessons learned from this first experiment

The first three problems were due to a lack of precision when formulating requirements. One could argue that a specialist in physical systems simulators design would have interpreted such requirements correctly in the first place. Still, undoubtedly, the less a requirement is subject to a bad interpretation, the better it is. This experiment stresses out that Lurette can be seen as an engineering tool that helps to write consistent and precise requirements. The fourth problem was much more interesting for the Corys engineers and revealed a real feature of this very frequently used object that behaves unexpectedly when used with an unusual convergence criterion.

The principal lesson of this experimentation is that writing executable requirements is not that difficult and can be very effective. Indeed, the experiment was conducted by an engineer that was ignorant about Lustre, Lutin, Alices, and dynamic systems modeling. Still, he was able to pinpoint four issues in less than one week of work with a few interactions with the Alices libraries supervisor.

We performed a similar study during the COMON project on voters designed in Scade by Rolls-Royce. Their voters were much simpler, with no internal state. Hence their formalization into Lustre oracles ended up into something equivalent to the Scade implementation. We believe that using oracles in this context is still useful, as it amounts to have two teams implementing the same specification, which is a classical strategy to gain confidence in software implementations. In such cases, Lutin stimulators can still be useful to compare thoroughly two implementations. In the particular case of Rolls-Royce voters, it was not necessary as we were able to prove their equivalence by state exploration (using the Lesar model-checker [7]). This illustrates the synergy we can have between formal-based testing and formal verification.

4 Timed Test Plans Automation

4.1 Test plans: a standard practice in industry

A standard practice in industry is to base test campaigns on *test plans*. The test plans of our three partners in the COMON project were actually very similar, and were made of a three columns table: one for the time (physical or logical); one for the stimulation, that specifies what the tester should do to perform the test; and one last column that specifies what the tester should observe in reaction to its stimulations. Corys developed in collaboration with EDF and AREVA a tool (I&C Simulation) to automate the play of such test plans, both for the stimulation and the decision parts. This tool processes scripts, where one can ask to set a variable to a particular value at a specific time; and then one can check that another variable take a specific value at another specific time.

One problem with such test plans, being automated or not, is that they are overly deterministic, both at the data and at the temporal levels. In the case studies we addressed so far with Lurette, the strategy was different as it consisted in writing high-level constraints both for generating several stochastic scenarios (Lutin), and for checking several traces (Lustre). This allows covering much more cases with the same specifications.

Nevertheless, engineers are used to write test plans, and several years of know-how are associated to their design. This is why we find interesting to report how Lutin and Lustre could be used to implement test plans, and to show how easy it can be to add a little bit of data and temporal looseness.

In this section, we present a test plan provided by Atos, targeting a generic library object. This plan was extracted from an existing test campaign conducted some years ago. We first demonstrate how to automate the play of this test plan in a very faithful and deterministic manner, as it could have been done with the I&C Simulation tool of Corys for example. Then we demonstrate the benefits of our languages to relax and generalize the constraints on both the stimulation and the observation part, which leads to tests that cover more cases, and are easier to maintain.

4.2 The SUT: a SCADA generic object

A SCADA (Supervisory Control and Data Acquisition [8]) is a remote management system for large-scale processing in many real time telemetry and remote control industrial installations (manufacturing, food processing, energy). It typically handles in real-time thousands of data (e.g., coming from sensors), and presents a relevant synthesis in graphical form to operators so they can monitor and control the system. Atos develops and commercialises several SCADA dedicated to the supervisory and control of power generating plants (nuclear, fuel, gas).

The purpose of the generic object we want to test is to monitor the operating area of a pair of numeric values (which typically comes from the physical process) and to raise alarms when dreaded events occur. The space where the monitored point evolves in is divided into several operating domains (nominal, degraded, etc.), and into several zones. When the point enters in a *forbidden zone*, an alarm should be raised; when it remains in an *accumulating zone* too long, another alarm should be raised; in an *authorized zone*, there is nothing to check. The zones shapes differ for each domain. The system chooses a domain, depending on various criteria on the evolution of the operating point. The operator can ask to favor some domain, and he can force it (i.e., ask more categorically). The number and the shape of domains (that can overlap) and zones are parameters of the generic object.

The SUT is such a parametrized object, with four domains and five zones; zone 2 is forbidden; zones 3 and 5 are accumulating; zones 1 and 4 are authorized. The SUT environment is made of two integers (`X` and `Y`) that hold the monitored point coordinates, and three Boolean inputs per domain so that the operator can ask to choose a domain (`dd1` to `dd4`), force a domain (`fd1` to `fd4`), or un-force it (`ud1` to `ud4`).

The test plan `CRT_019_S04`. The existing test campaign we based our work on consisted in 21 test plans. The `CRT_019_S04` is one of them, and is shown in Table 3. This test plan is split into seven logical steps, and four stages. At each step, the operator sets the values of variables mentioned in the action column, and checks (visually) that the system behaves as specified in the expected result column.

The Atos I/O stimulator. In order to ease the test of their SCADA objects, Atos developed an in-house tool called the Input/Output stimulator. This tool processes scripts, and is basically able to (1) set SCADA internal variable values; (2) display messages; (3) suspend the script until the operator presses a key (`WAIT`). This stimulator is used to ease the play of test plans by automating the run of the action column, and to limit the intervention of the operator to a few key presses. In the `CRT_019_S04` plan, each of the four stages actually corresponds to a `WAIT` statement in the corresponding I/O stimulator script. The tester does the expected results checking.

4.3 Implementing automated test plans with Lutin and Lustre

The first step to implement with Lurette an automated version of this test plan was to connect our languages APIs to the Atos SCADA. To do that, we re-used the infrastructure that was set up for the I/0 stimulator. We also added a layer in charge of interfacing an event-triggered workbench (SCADA) with time-triggered programs (Lutin/Lustre).

step nb	Action	Expected result / Comment
1	Launch stage 1 which elects domain 1 and sets X,Y to (25,40) (in zone 1)	Check the image display
2	Launch stage 2 which sets X,Y to (40,28) in the forbidden zone 2	Check the operating point (position, color) Check the alarm raised in the alarm function Write down the timestamp
3	Launch stage 3, which elects domain 2 instead of domain 1	Check that the alarm above remains at the timestamp of step 2 X,Y remains in the forbidden zone 2
4	Force domain 3	Check that the alarm above remains at the timestamp of step 2 X,Y remains in the forbidden zone 2
5	Force domain 4	The alarm above disappear X,Y is now in an authorized zone 4
6	Unforce domain 4	The alarm is raised at the current timestamp domain 2 is elected X,Y is back in the forbidden zone 2
7	Launch stage 4 which sets X,Y to (-9,25) in the authorized zone 4	The alarm disappears X,Y in the authorized zone 4

Table 3. The `CRT_019_S04` test plan

From Lurette to SCADA, we generate an event each time a variable value changes (up to a given threshold). From SCADA to Lurette, we perform a periodic sampling of the variable values. This sampling is done at 1 hertz, to avoid data race problems and to remain deterministic and reproducible: indeed, 1 second is enough for the SUT to address all events resulting from the change of all interface variables. Note that it would have been easy and interesting to test what happens at higher rates.

The « Expected result » column of Table 3 in Lustre. In order to detect bad behaviors, we formalize the observation column of the `CRT_019_S04` test plan with a Lustre oracle that monitors the following inputs: the step number (`sn` $\in$ [1,7]) ; the current zone (`czone` $\in$ [1,5]); the alarm of zone 2 (`A2`); the elected domain (`d_elec`); the current timestamp (`ts_c`); and the timestamp of alarm `A2` (`ts_a2`). Here again, we have shortened variable names for the sake of readability.

```
node crt019_s04(sn:int; czone:int; A2:bool; d_elec,ts_c,ts_a2:int)
returns(ok : bool);
var ok1,ok2,ok3,ok4,ok5,ok6,ok7:bool; lts_a2:int;
let
 lts_a2 = 0 -> if r_edge(A2) then ts_c else pre(lts_a2);
 ok1 = (sn=1 => (czone=1));
 ok2 = (sn=2 => (czone=2 and A2));
 ok3 = (sn=3 => (czone=2 and A2 and ts_a2=lts_a2));
 ok4 = (sn=4 => (czone=2 and A2 and ts_a2=lts_a2));
 ok5 = (sn=5 => (czone=4 and not A2));
 ok6 = (sn=6 => (czone=2 and d_elec=2 and ts_a2=ts_c));
 ok7 = (sn=7 => (czone=4 and not A2));
 ok  = ok1 and ok2 and ok3 and ok4 and ok5 and ok6 and ok7;
tel
```

The local variables `ok1` to `ok7` encode the seven steps of the third column. In order to « write down the timestamp » at step 2, we define a local variable `lts_a2` as follows:

initially set to 0, it then takes the value of the current timestamp `ts_c` when `A2` is raised (`r_edge(A2)`), and keeps its previous value otherwise (`pre(lts_a2)`). To encode the expected result of steps 3 and 4, we compare the timestamp of the `A2` provided in input `ts_a2` with its counterpart computed locally `lts_a2`.

The « action » column of Table 3 in Lutin. We first present a completely deterministic Lutin program that mimics the behavior of an operator that processes this test plan. Then we show how slight modifications of this program can lead to a stimuli generator that covers much more cases. Let us first define a few Boolean macros to enhance the programs readability. The `tfff` macro below binds its first parameter to true, and all the other ones to false.

```
let tfff(x,y,z,t:bool):bool = x and not y and not z and not t
```

Similarly, we define `ftff`, which binds its second parameter to true; `f7` and `f8` bind all their parameters to false. The integer input `sn` is used to choose the instant at which we change the step. It can be controlled by a physical operator or by another Lutin node that sequentially assigns values from 1 to 7 (similar to the `sn_gen` node of Section 2). The fourteen outputs of this node controls the domain to display (display domain `i` if `ddi` is true), to force (force domain `i` if `fdi` is true), or to un-force (un-force domain `i` if `udi` is true).

```
node crt019_s04(sn:int) returns
(X, Y: real; dd1,dd2,dd3,dd4, fd1,fd2,fd3,fd4, ud1,ud2,ud3,ud4: bool) =
 loop {
   sn=1 and X=25.0 and Y=40.0 and tfff(dd1,dd2,dd3,dd4) and
   f8(fd1,fd2,fd3,fd4,ud1,ud2,ud3,ud4)
```

As long as the `sn` input is equal to 1, the outputs of the `crt019_s04` node satisfy the constraint above that states that only the first domain should be displayed, and no domain is forced or unforced. `X` and `Y` are set in the authorized zone 1. When `sn` becomes equal to 2, the control passes to the constraint below, which is the same as the previous one except that the point is set somewhere in zone 2.

```
} fby loop {
  sn=2 and X=40.0 and Y=28.0 and tfff(dd1,dd2,dd3,dd4) and
  f8(fd1,fd2,fd3,fd4,ud1,ud2,ud3,ud4)
} fby loop {
  sn=3 and X=40.0 and Y=28.0 and ftff(dd1,dd2,dd3,dd4) and
  f8(fd1,fd2,fd3,fd4,ud1,ud2,ud3,ud4)
} fby loop {
  sn=4 and X=40.0 and Y=28.0 and ftff(dd1,dd2,dd3,dd4) and
  fd3 and f7(fd1,fd2,    fd4,ud1,ud2,ud3,ud4)
} fby loop {
  sn=5 and X=40.0 and Y=28.0 and ftff(dd1,dd2,dd3,dd4) and
  fd4 and f7(fd1,fd2,fd3,    ud1,ud2,ud3,ud4)
} fby loop {
  sn=6 and X=40.0 and Y=28.0 and ftff(dd1,dd2,dd3,dd4) and
  ud4 and f7(fd1,fd2,fd3,fd4,ud1,ud2,ud3    )
} fby loop {
  sn = 7 and X=-9.0 and Y=25.0 and ftff(dd1,dd2,dd3,dd4) and
  f8(fd1,fd2,fd3,fd4,ud1,ud2,ud3,ud4)
}
```

This Lutin program, once run with the oracle of Section 4.3, allows test automation. However, it suffers from the same flaw as its original non-automated counterpart: it can

be tedious to maintain. Indeed, if for some reason, the shape of zone 1 is changed, the chosen point (25,40) might no longer be part of zone 1. Choosing pseudo-randomly any point in zone 1 using the Lutin constraint solver makes the plan more robust to software evolution. Moreover, with the same effort, it covers much more cases. In the same spirit, we can further loosen this plan by replacing “choose a point in the authorized zone 1” by “choose a point in any authorized zone” (cf the `x_y_gen` node of Section 2). In step 3, 4, and 5, we could also toss the choice of the domain to be forced. Actually, by loosening this plan in this way, we obtain a plan that covers more cases than the twenty other plans of the test campaign!

4.4 Discussion and lessons learned from this second experiment

The original test plan was not deterministic, since the time between each step change was controlled by a physical tester. However, this non-determinism is easy to simulate with Lutin, for example using the `sn` generator (`sn_gen`) presented in Section 2. The advantage of the Lutin non-determinism over the human one is its reproducible nature. Indeed, one just needs to store the seed used by the Lutin pseudo-random engine to be able to replay the exact same simulation.

This test plan does not illustrate the feedback capability of Lurette. However, plans where the tester should perform some specific actions depending on some behavior of the SUT are very common.

We have shown a way to use Lurette and its associated languages to automate the run of an existing test plan, designed to be exercised by a human operator. The initial set-up for automated plans seems to require more effort, as each variable behavior has to be described precisely at each step, while the original plan was more allusive. But the Lurette version has four major advantages: (1) it can be run automatically, (2) each run is reproducible, (3) it covers (much!) more cases, and (4) it is more robust to software (or specification) evolutions.

The two last points are the most important. Indeed, Atos experimented with completely automated test scripts, but gave up as they were too difficult to maintain. One reason was that their scripts were too sensitive to minor time or data values changes. The use of languages with a clean semantics with respect to time and parallelism eases the writing of more abstract and general properties that can serve as oracles for several test scenarios. The concision and the robustness arguments hold both for the oracles and the stimulators, and from the data and the temporal points of view.

In previous experimentations ([3,5] and Section 3), the methodology was to derive oracles and stimulators from informal requirements. The initial stimulator is made of very general constraints. Then, to increase oracles coverage, Lutin scenarios are designed. During the COMON project, we also experimented with this methodology on the SCADA object, and the coverage was actually comparable. This “direct formalization approach” is more modular, as some variables sets can be defined separately, whereas with test plans, one needs to describe all variables at the same time. Moreover, it allows writing specific scenarios only when it is necessary, as some oracles are covered in the first place using simple constraints on the environment. Once all the easy cases are explored at random with a minimal effort, it remains the difficult work that consists in driving the SUT to set it in some configurations that exhibit interesting cases. This

is a work of SUT experts. Leveraging testers from the tedious and systematic part, and letting them focus on interesting parts using high-level languages could restore the interest in testing, which often has a poor reputation. Writing Lutin programs is a creative activity, and generalising its use could ease to relocate test teams.

5 Related Work

Automating the test decision with executable oracles is a simple and helpful idea used by many others. The real distinction between Lurette and other tools lies in the way SUT inputs are generated. In the following, we group approaches according to the input generation techniques: source-based, model-based, or environment-model based. We found no work dealing with automated testing of SCADA systems. For dynamical systems workbenches (such as Alices), the literature is quite abundant, and mostly concerns Simulink [9]. Hence we focus here on works targeting Simulink, and refer to the related work section of [3] for a broader and complementary positioning of Lurette in the test of reactive systems.

Source code based testing (white-box). The White-box testing approach consists in trying to increase structural coverage by analysing the SUT source using techniques coming from formal verification such as model-checking [10], constraint solving, or search-based exploration [11,12]. Such approaches are completely automated, but can be confronted to the same limitations as formal verification with respect to state space explosion. Several industrial tools use white-box techniques to test Simulink designs, e.g., Safety Test Builder [13], or Design Verifier [14].

Model-Based Testing (grey-box). A very popular approach in the literature [15,16] consists in viewing the SUT as a black-box, and designing a more or less detailed model of it. This model should be faithful enough to provide valuable insights, and small enough to be analyzable. The model structure is sometimes used to define coverage criteria. The model is used both for the test decision and the stimuli generation. T-VEC [17,18] and Reactis Tester [19] are an industrial tools using this approach to generate tests offline. With Lurette, we also use a model of the SUT, but this model is only used for oracles. The input generation is developed by exploration of environment models. A way to combine this approach with Lurette would be to use such models of the SUT to generate Lutin scenarios to guide the SUT to specific states and increase coverage.

Environment Model based testing (black-box). While the white-box approach intends to increase structural coverage, the main goal of black-box testing is to increase (functional) requirements coverage [20]. Time Partition Testing (TPT) is an industrial black-box tool distributed by Piketec [21]. As Lurette, TPT have its own formalism to model the environment and automate the SUT stimulation [22,23]. It is a graphical formalism based on hierarchical hybrid automata that is able to react online on the SUT outputs. The major difference with Lutin is that those automata are deterministic. It uses python oracles to automate the test decision, although Lustre is arguably better for specifying high-level timed properties.

Another way to explore the environment state space, which has been experimented on Simulink programs [24,25], is to perform heuristic search (evolutionary algorithms,

simulated annealing [26]). The idea is to associate to each SUT input a set of possible parametrized generators (ramp, sinus, impulse, spline). The search algorithms generate input sequences playing with several parameters, such as the number of steps each generator is used, their order, or the amplitude of the signal. A fitness function estimates the distance of the trace to the requirements. Then, another trace is generated with other parameters, until an optimal solution is found. A limitation of their generators is that they are not able to react to SUT outputs. More generally, for systems that have a complex internal state, it can be very difficult to drive it in some specific operating mode; to do that, the knowledge of the expert is mandatory (and being able to react to SUT outputs too). Instead of guiding a random exploration via heuristics, Lurette proposal consists in asking experts to write programs that performs a guided random exploration of the SUT input state space. A way to combine both approaches could be to let some evolutionary algorithms choose some parameters of Lutin programs, such as choice point weights or variable bounds.

6 Conclusion

The main lesson of the first experimentation is that writing executable requirements is not that difficult and allows to write precise and consistent requirements. This study gave new insights to Corys engineers on one of their most frequently used object.

The second experimentation demonstrates a way to automate the execution of timed test plans. Test plans are commonly used in industry, and automating their process aroused a big interest within our industrial partners. Lutin and Lustre allows improving their use by permitting the design of more abstract test plans that are more robust to temporal and data changes. One noteworthy outcome of this study is that the resulting randomized and automated test plan actually covers more than the 21 test plans of the original test suite.

There is a synergy between automated oracles and automated stimulus generation. Indeed, generating thousands of simulation traces would be useless without automatic test decision. Conversely, designing executable requirements to automate the decision of a few scenarios generated manually might not be worth the effort.

This work also demonstrates that synchronous languages are not only useful for designing critical systems (as the success of Scade gives evidence of), but can also be used to validate dynamic systems models (Alices) or event-based asynchronous systems (SCADA). The language-based approach of Lurette allows performing several kinds of test (unit, integration, system, non-regression) on various domains [3,5].

From an industrial use perspective, a general-purpose library and specialized domain-based ones are still to be done. That situation may progress in the near future, as the interest expressed in Lurette by the three industrial partners of the COMON project is one of the reasons that convinced people to establish in 2013 the Argosim company. Argosim is developing the Stimulus tool based on the Lurette principles [27].

References

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

2. Raymond, P., Roux, Y., Jahier, E.: Lutin: a language for specifying and executing reactive scenarios. EURASIP Journal on Embedded Systems (2008)
3. Jahier, E., Halbwachs, N., Raymond, P.: Engineering functional requirements of reactive systems using synchronous languages. In: International Symposium on Industrial Embedded Systems, 2013. SIES'13., Porto, Portugal (2013)
4. Halbwachs, N., Fernandez, J.C., Bouajjanni, A.: An executable temporal logic to express safety properties and its connection with the language lustre. In: ISLIP'93, Quebec. (1993)
5. Jahier, E., Raymond, P., Baufreton, P.: Case studies with lurette v2. Software Tools for Technology Transfer **8**(6) (nov 2006) 517–530
6. Jahier, E., Raymond, P.: Generating random values using binary decision diagrams and convex polyhedra. In: CSTVA, Nantes, France (2006)
7. Raymond, P.: Synchronous program verification with lustre/lesar. In: Modeling and Verification of Real-Time Systems. ISTE/Wiley (2008)
8. Bailey, D., Wright, E.: Practical SCADA for industry. Elsevier (2003)
9. The Mathworks: Simulink/stateflow. http://www.mathworks.com
10. Hamon, G., de Moura, L., Rushby, J.: Generating efficient test sets with a model checker. In: Software Engineering and Formal Methods, 2004. (2004) 261–270
11. Satpathy, M., Yeolekar, A., Ramesh, S.: Randomized directed testing (redirect) for simulink/stateflow models. In: Proceedings of the 8th ACM international conference on Embedded software. EMSOFT '08, New York, NY, USA, ACM (2008) 217–226
12. Zhan, Y., Clark, J.A.: A search-based framework for automatic testing of MATLAB/Simulink models. Journal of Systems and Software **81**(2) (2008) 262 – 285
13. TNI Software: Safety Test Builder. http://www.geensoft.com/fr/article/safetytestbuilder/
14. The Mathworks: Design verifier. http://www.mathworks.com/products
15. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A.: Model-Based Testing of Reactive Systems: Advanced Lectures (LNCS). Springer-Verlag New York, Inc. (2005)
16. Zander, J., Schieferdecker, I., Mosterman, P.J.: 1. In: A Taxonomy of Model-based Testing for Embedded Systems from Multiple Industry Domains. CRC Press (2011) 3–22
17. T-VEC: T-vec tester. http://www.t-vec.com
18. Blackburn, M., Busser, R., Nauman, A., Knickerbocker, R., Kasuda, R.: Mars polar lander fault identification using model-based testing. In: 8th IEEE International Conference on Engineering of Complex Computer Systems. (2002) 163–169
19. Reactive Systems: Testing and validation of simulink models with reactis white paper.
20. Cu, C., Jeppu, Y., Hariram, S., Murthy, N., Apte, P.: A new input-output based model coverage paradigm for control blocks. In: Aerospace Conference, 2011 IEEE. (2011) 1–12
21. Piketec: Tpt. http://www.piketec.com
22. Lehmann, E.: Time partition testing: A method for testing dynamic functional behaviour. In: Proceedings of TEST2000, London, Great Britain (2000)
23. Bringmann, E., Kramer, A.: Model-based testing of automotive systems. In: Software Testing, Verification, and Validation, 2008 1st International Conference on. (2008) 485–493
24. Vos, T.E., Lindlar, F.F., Wilmes, B., Windisch, A., Baars, A.I., Kruse, P.M., Gross, H., Wegener, J.: Evolutionary functional black-box testing in an industrial setting. Software Quality Control **21**(2) (2013) 259–288
25. Baresel, A., Pohlheim, H., Sadeghipour, S.: Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms. In: Genetic and Evolutionary Computation. Volume 2724 of LNCS. Springer Berlin Heidelberg (2003) 2428–2441
26. McMinn, P.: Search-based software test data generation: a survey: Research articles. Softw. Test. Verif. Reliab. **14**(2) (June 2004) 105–156
27. Argosim: Stimulus. http://www.argosim.com