An Overview of Lutess
A Specification-based Tool for Testing Synchronous Software[*]

L. du Bousquet N. Zuanon
LSR-IMAG, BP 72, 38402 Saint Martin d'Hères, France

E-mail: {ldubousq, zuanon}@imag.fr

Abstract

Test data generation and test execution are both time-consuming activities when done manually. Automated testing methods promise to save a great deal of human effort. This especially applies to reactive programs which have complex behaviors over time and which require long test sequences.

In this article, we present Lutess, a testing environment for synchronous reactive software. Lutess produces automatically and dynamically test data with respect to some environment constraints of the program under test. Moreover, it allows to trace the test execution and spot the situations where the program violates its properties.

Lutess offers several specification-based testing methods. They aim at simulating more realistic environment behaviors, producing relevant data to test thoroughly a given property or driving the program under test into interesting situations. To produce the test data, the methods use different types of guides: statistical distribution of the input generation, properties, or behavioral patterns.

Lutess proved to be powerful and easy to use in industrial case studies. Lutess won the Best Tool Award of the first Feature Interaction Detection Contest. The tool is described hereafter from a practical point of view.

1 Introduction

A reactive system must continually respond to signals from its environment, and must satisfy temporal constraints so that it can capture all the external events of concern.

Synchronous programs are a sub-class of reactive software programs. The synchrony hypothesis [2] states that every reaction of the software application to external events is theoretically instantaneous (actually, fast enough to guarantee that the environment remains invariant during the computation of the reaction).

Reactive and/or synchronous systems are often safety-critical and must be thoroughly validated to ensure that they meet their requirements. Since formal verification is often impracticable because of lack of memory and/or time, alternative solutions such as testing are needed.

Several points make synchronous system testing specific. First, since the validation of a reactive system requires that the latter does maintain its relation with its environment over long sequences of exchanges, the number of input-output relations (test cases) to be managed is really large. These relations can't be computed by hand, since the reactive system output usually depends on the system history (and not only on its current input). Besides, the system specifications can be only partial. It is therefore very difficult to calculate the input-output relations needed to evaluate the test results. These facts discard the choice of a testing process based on human involvement. Thus, testing should be automated in order to make it easier, improve its quality and lower its cost.

We have developed Lutess, a testing environment that supports highly automated testing of synchronous reactive systems [14]. Lutess offers different testing methods in order to fit the tester needs as well as possible. Indeed, we are convinced that a single testing method cannot meet all the needs of a tester. For instance, some methods produce test data so that the most used operations would receive the most testing [13], others produce data randomly and/or based upon an input partition [11]... The foundations of Lutess (the formal description of its testing methods) can be found in [8].

The aim of this paper is to provide an overview of the tool and to show how it is easy to use. The usefulness of each method is illustrated with a single example. This example concerns the validation of a telecommunication feature specification, namely the Call Forwarding No Reply.

The paper is organized as follows. Section 2 gives a brief description of the principles of our tool Lutess. Section 3 provides an example of synchronous program. In sections 4 and 5, we detail the test data generation methods provided by our tool. Section 6 is devoted to the implementation of the tool. Section 7 explores the advantages, scope and limitations of the tool. Section 8 introduces related work.

2 Lutess

2.1 Architectural overview

The validation process of reactive systems consists in showing that the system under consideration satisfies its required properties, provided that its environment meets some given requirements. An important point is that the validation is done under assumptions about the possible environment behaviors. When one is not concerned with the system robustness, it makes no sense to take into account impossible environment behaviors. For example, it is physically impossible for the user of a telephone to go on the hook twice without going off the hook in between. When considering a telephony system, only sequences among which “go off” et “go on” actions alternate are meaningful with respect to testing.

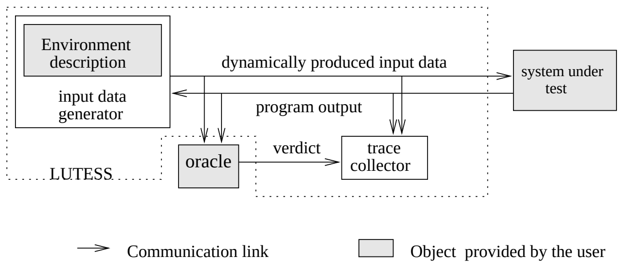


Figure 1. Lutess

For this reason, Lutess requires three elements: the *system under test*, its *environment description* and an *oracle* (as shown in figure 1). Lutess constructs automatically the test harness which builds a test data generator, links the generator, the system under test and the oracle, coordinates their executions and records the sequences of input-output relations and the associated oracle verdicts (test sequences).

The test is operated on a single action-reaction cycle, driven by the generator. The generator randomly selects an input vector for the system under test and sends it to the latter. The unit under test reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated. The oracle observes the program inputs and outputs, and determines whether the software requirements are violated.

The test data generator is automatically built by Lutess from an environment description written in Lustre[1] [3]. This description is provided as a single syntactical unit, called a *testnode* [15]. Examples of environment description and oracle properties are given in section 3.

[1]Lustre is both a synchronous programming language and a temporal logic.

The unit under test and the oracle are both synchronous executable programs with boolean inputs and outputs. Optionally, they can be supplied as Lustre programs.

To begin the test data generation, one has to feed Lutess with a probability seed, which is used to initialize a classical C random number generator. Keeping in mind that the behavior of a synchronous program is deterministic, i.e. in a given state, its response to a given input vector is always the same, one can note that the use of such a generator allows to reproduce any experiment, by using the same seed. For a given program and a given environment description, Lutess requires different seeds in order to produce different test sequences.

Moreover, the user has to specify the number (n) and the length (l) of the test sequences Lutess has to produce. The process which produces the n test sequences of l values is called a *test run*. During a test run, the program is reset in its initial state at the beginning of each new test sequence (while the random number generator is not).

Test data generation is not stopped when the oracle detects an error. It is stopped when the last sequence of the test run has been fully generated. This enables a same test sequence to reveal several different errors.

Lutess includes a “trace collector” which provides 3 functions: a data recorder, a data translator and a data analyzer. The recorder saves the input, output and oracle data (boolean values) into specific files. The translator displays the boolean values in a textual mode (defined by the user). This makes the manual trace analysis more comfortable than the analysis of sequences of boolean vectors. The analyzer allows the user to replay a test sequence with different oracles.

2.2 Lutess testing methods

During a test run, at each cycle (or step), the Lutess generator randomly selects an input vector for the system under test. Basically, the input is selected using the environment description (black-box testing), and assuming that the data distribution is uniform. But the user can also define:

- an input statistical (partial) distribution; the generator will produce inputs according to the given distribution;
- some (safety) properties; the generator will select preferably inputs which potentially drive the system under test toward those properties violation;
- some scenarios (behavioral patterns); the generator will select preferably inputs which follow the scenario.

These methods are described in sections 4 and 5.

3 Example

As an illustration of Lutess application, we consider an executable specification of a telephony system offering the Call Forwarding No Reply feature (CFNR)[2]. This feature allows a subscriber to have his incoming calls redirected when he does not answer within a given delay. The feature is dynamically activated and deactivated. The number to which calls are redirected is also dynamically set.

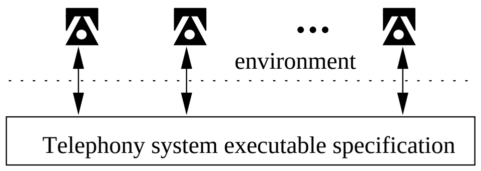


Figure 2. Telephony System Model

The telephony system is modeled from the users' viewpoint. Its environment includes the physical telephones which are linked to the system (figure 2). The system we consider is composed of 4 users (called A,B,C,D).

System inputs (produced by the environment) are events describing the actions performed on the phones : On_i, Off_i, $\text{Dial}_i(j)$, $\text{CFon}_i(j)$, CFoff_i, with i and $j \in \{\text{A,B,C,D}\}$). The event $\text{CFon}_i(j)$ indicates that user i requires the activation of his CFNR feature to forward his calls towards j; the event CFoff(i) means that user i demands his CFNR feature to be deactivated.

Outputs are signals which produce specific tones at the terminal (such as Busy-Tone, Ringing-Tone, ...). Each output signal identifies the state of the phone. In this example, a phone has 7 states, which are *idle* (I), *dialing* (D) *waiting* (W), *alerting* (A), *talking* (T), *ringing* (R), and *exception* (E)[3].

In order to perform the validation of this system, the human tester has to exhibit the environment description and the system requirements (oracle properties).

Lustre [3] is a programming language for synchronous programs, which is declarative and data-flow oriented. It corresponds to a linear past temporal logic which offers usual arithmetic, boolean and conditional operators and two specific temporal operators : **pre**, the "previous" operator, and $->$ the "followed-by" operator. Let E and F be two expressions of the same type denoting the sequences $(e_0, e_1, \ldots, e_n \ldots)$ and $(f_0, f_1, \ldots, f_n \ldots)$; **pre**$(E)$ denotes the sequence $(nil, e_0, e_1, \ldots, e_{n-1} \ldots)$ where nil is an undefined value. $E -> F$ denotes the sequence $(e_0, f_1, \ldots, f_n \ldots)$.

Lustre allows the specifier to define its own logical or temporal operators to express invariants. For example, in this paper, we use the temporal operator **Once_from_to**(A,B,C) to specify that property A must hold at least once between the instants where B and C occur.

[2]This example is taken from a case study aiming at modeling feature specifications from their ETSI descriptions [6].

[3]A phone is waiting when a number has been dialed and the connection has not been established yet. It is alerting when the connection is established but the party has not gone off the hook yet.

Environment description

1. At most one event can be produced at each instant of time. The events being On_i, Off_i, $Dial_i(j)$, $CFon_i(j)$, $CFoff_i$, with i and $j \in \{\text{A,B,C,D}\}$, this contraint is written in Lustre as below:
   (E1) $\#(On_A, Off_A, Dial_A, \ldots, CFoff_D)$
   where # is the Lustre operator which is true when "at most one element of the parameter list is true".

2. A user can't go off (resp. on) the hook twice without going on (resp. off) the hook in between:
   (E2) **once_from_to**(On_i, **pre** Off_i, Off_i) **and** **once_from_to**(Off_i, **pre** On_i, On_i).

3. A user can dial only if his telephone is in the state *Dialing*, which is identified by the *DialingTone*:
   (E3) $Dial_i \Rightarrow DialingTone_i$

4. A user can (try to) activate and deactivate the CFNR service only when his telephone emits the *dialing* tone:
   (E4) ($CFon_i$ **or** $CFoff_i$) $\Rightarrow DialingTone_i$

The environment constraints E1, E2, E3, E4 have to be inserted in a testnode.

A testnode is a description of the test data generator characteristics. Hereafter is an example of a testnode. As it can be noted, the testnode inputs (resp. outputs) are the system's outputs (resp. inputs). This should be understood as "the generator receives the program outputs as inputs, and generates (i.e. returns) input data for the programs". The use of local variables to express more easily environment properties is possible.

```
testnode Environment (o1, o2,.., o_m : program_outputs)
returns (i1, i2,.., i_n : program_inputs)
  var l1, l2,.., l_k : local_variables;
  let
     environment(E1, E2, E3, E4);
  tel
```

Oracle properties (system requirements)

As a preliminary definition, we say that the CFNR feature is *invoked* for a user, if the latter is a CFNR subscriber which has activated this service, and if he/she does not answer a call within the time delay.

1. A call will be forwarded if (1) the callee feature is invoked and (2) the maximum number of forwards is not reached. This bound is a service provider option which was set to 3 for our example.

2. A call can be forwarded only if the service has been previously activated by the callee, and if the latter did not deactivate the service in the meantime.

3. A forwarded call will be redirected to the last user which has been designated by the subscriber.

Using Lustre, it is easy to build an oracle program from these properties. First, one has to express each property in Lustre. This can be easily done, by defining intermediary variables and by using Lustre classical operators. For instance, for the last property, we defined a predicate *LastUser(x)* that takes into account the last activation of the feature by user x:

$$LastUser(x) \;=\; \textbf{if}\; CFon(x,y)\; \textbf{then}\; y\; \textbf{else}\; \textbf{pre}\; LastUser(x)$$

We then used this predicate to express the property:

$$p_3 \;=\; (CallForward(x,y) => LastUser(x) = y)$$

CallForward(x,y) is a predicate which is true whenever a call for x is forwarded to y ($x, y \in \{A, B, C, D\}$). For sake of simplicity, we don't detail its definition.

Then, from the Lustre expressions of these properties, say p_1, p_2 and p_3, we build a Lustre program. The inputs of the latter are the program under test inputs and outputs. Its unique output is a single boolean variable, whose value is the conjunction of the oracle properties expressed in Lustre:

```
node Oracle (program_inputs; program_outputs)
returns (res : boolean)
  let
     res = p1 and p2 and p3 ;
  tel
```

This Lustre program has then to be compiled to obtain the executable oracle.

4 Random Testing by Environment Simulation

4.1 Basic random specification testing

Test data are generated only with respect to the environment constraints (black-box testing). This is the weakest test data selection criterion one can define for synchronous software. The test data generation is performed in such a manner that the data distribution is uniform. Table 1 gives an example of trace that Lutess has produced with this method.

```
 1: -     -   -     -   -     -   -     -   I I I I True
 2: OffA  -   -     -   -     -   -     -   D I I I True
 3: CFonA (D) -     -   -     -   -     -   E I I I True
 4: -     -   -     -   -     -   -     -   E I I I True
 5: -     -   -     -   -     -   OffD  -   E I I D True
 6: -     -   -     -   -     -   DialD (D) E I I W True
 7: -     -   -     -   OffC  -   -     -   E I D W True
 8: -     -   -     -   CFonC (B) -     -   E I E E True
 9: -     -   OffB  -   -     -   -     -   E D E E True
10: -     -   DialB (A) -     -   -     -   E W E E True
11: -     -   -     -   OnC   -   -     -   E W I E True
12: -     -   -     -   -     -   OnD   -   E E I I True
13: OnA   -   -     -   -     -   -     -   I E I I True
14: -     -   -     -   -     -   -     -   I E I I True
15: OffA  -   -     -   -     -   -     -   D E I I True
16: CFonA (C) -     -   -     -   -     -   E E I I True
17: -     -   OnB   -   -     -   -     -   E I I I True
18: -     -   OffB  -   -     -   -     -   E D I I True
19: -     -   CFonB (D) -     -   -     -   E E I I True
20: -     -   -     -   OffC  -   -     -   E E D I True
21: -     -   -     -   OnC   -   -     -   E E I I True
<a><------------------- b --------------><--(c)--><d >
```

(a) step number;
(b) user actions (Off$_x$,On$_x$, Dial$_x$(y), CFon$_x$(y), CFoff$_x$; $x, y \in \{A, B, C, D\}$);
(c) Phone state (Idle, Dialing, Waiting, Alerting, Talking, Ringing, Exception);
(d) Oracle verdict (issued by the oracle defined in section 3).

Table 1. A trace generated by Lutess

Empirical observations

For complex systems, a uniform distribution is far from the reality. Indeed, test data in table 1 show that some users' phones stay off the hook for long periods of time in Exception state (i.e. after receiving a Busy Line indication), e.g. user A between states 4 and 13. In reality, a user would have quickly gone on the hook in such a situation. Similarly, many generated behaviors consist in alternating going off and on the hook, performing no action in between (user C, step 20 and 21), which is not a common behavior. We also noticed that, on the whole, every user tries to call himself/herself as often as any other user (user D, step 6) or to activate the CFNR feature several times in a row (user A, steps 3 and 16). In the real world, such behaviors rarely occur, and are most of the time the result of wrong actions.

In order to test or analyze more realistic simulations, one may want to specify its own statistical environment distribution. With Lutess, this is possible thanks to conditional probabilities that one can associate with program inputs.

4.2 More realistic random specification testing

Lutess offers facilities to define in the testnode a multiple probability distribution [17] in terms of conditional probabilities associated with the unit under test input variables [5]. The variables which have no associated conditional probabilities are assumed to be uniformly distributed. A conditional probability assignation defines, for an input variable, its probability to be set to true when a given condition is met. The conditions are Lustre expressions. An algorithm is implemented in Lutess to automatically translate a set of conditional probabilities into an operational profile (and vice versa).

Let us try this method on our example. The conditional probabilities are chosen in order to overcome the problems exhibited by the previous empirical observations. For instance, to decrease the time spent by one user's phone in the Exception state, one can specify that the probability to go on the hook is high while the phone is in the Exception state.

$$\langle \mathit{OffA}, 0.9, \mathbf{pre}\, \mathit{ExceptionA} \rangle$$

Let $c_1, c_2..., c_s$ be a list of conditional probabilities. Similarly to the environment constraints, the conditional probabilities have to be declared in the testnode, in the following way:

```
testnode Environment (o1, o2,.., $o_m$ : program_output_events)
returns (i1, i2,.., $i_n$ : program_input_events)
  var l1, l2, . . . , $l_k$ : local_variables;
  let
      environment(E1, E2, E3, E4);
      proba($c_1, c_2, ..., c_s$);
  tel
```

Empirical observations

Regarding the last unrealistic aspect mentioned in the previous subsection, we defined about 60 conditional probabilities (15 for each user). There are 5 possible actions for each user, and approximately 3 conditional probabilities per action. Indeed, an action may have different probabilities depending on the phone states. For instance, the probability to go on the hook is usually different in the states Exception, Dialing and Talking.

5 Guided testing

A realistic environment simulation may not produce data which test rare but important and interesting features of the program. To overcome this problem, Lutess has two different methods which consist in testing in a more relevant manner some given properties or to drive the program into interesting situations. These methods produce data according to two types of guides: (invariant) properties and behavioral patterns.

5.1 Property-oriented testing

The property-oriented testing method is aimed at selecting test data which facilitate the detection of property violations. At each cycle, this method automatically generates values which are relevant to test the considered properties.

We say that a input data is relevant to test a property, when the program reaction is liable to cause an instantaneous failure with respect to this property. For instance, let's consider the simple property $\mathcal{P} : \ i \Rightarrow o$, where i (resp. o) is an input (resp. output) of the unit under test. When i is false, the unit under test cannot falsify $\mathcal{P}$. When i is true, the unit under test will falsify $\mathcal{P}$ if it returns the value false for o. Hence, $i =$true is relevant to test $\mathcal{P}$.

Input values which are relevant to the considered properties are favored over the values only associated with the environment. But the random selection process is fair enough to let those latter values be exercised. In Lutess, the properties chosen to guide the generator ($s_1, s_2..., s_z$) have to be defined with the environment description, in the testnode. Conditional probabilities can also be used in combination with this method.

```
testnode Environment (o1, o2,.., $o_m$ : program_output_events)
returns (i1, i2,.., $i_n$ : program_input_events)
  var l1, l2,.., $l_k$ : local_variables;
  let
      environment(E1, E2, E3, E4);
      proba($c_1, c_2, ..., c_s$);
      safety($s_1, s_2, ..., s_z$);
  tel
```

Empirical observations

One property of the telephony system is that the user's phone goes back to its idle state every time its user goes on the hook. Driving the generation with such a property led to favor the considered action, thus improving the tester's confidence in the system's reaction to this input. However, this resulted in every user tending to go on the hook as soon as possible; thus, many more realistic behaviors are never tested.

5.2 Behavioral Pattern-based Testing

As complexity grows, reasonable behaviors for the environment may reduce to a small part of all possible ones with respect to the constraints. Some interesting features of a system may not be tested efficiently since their observation may require sequences of actions which are too long and complex to be randomly frequent.

The behavioral pattern-based method aims at guiding further the input generation so that the most interesting sequences are produced. A behavioral pattern characterizes those sequences by listing the actions to be produced, as well as the conditions that should hold on the intervals between two successive actions (fig. 3). Regarding input data generation, all sequences matching the pattern are favored and get higher chance to occur. To that, desirable actions appearing in the pattern are preferred, while inputs that do not satisfy interval conditions get lower chance to be chosen. The generation method is usually invoked with environment constrained test data. Patterns are stated using graphical notations; Lutess automatically translates them into Lustre expressions.

In Lutess, the behavioral patterns have to be defined with the environment description, in the testnode.

Empirical observations

To avoid loops in the forwarding, specifying the CFNR feature requires that no more than 3 redirections are ever performed on a single call in a row. When checking what could happen in the case of more then 3 redirections, we noticed that this situation had little chance to occur. On the contrary, the use of a pattern has proved that it increases the situation likelihood in shorter test sequences. Figure 3 shows the graphical representation of such a pattern.

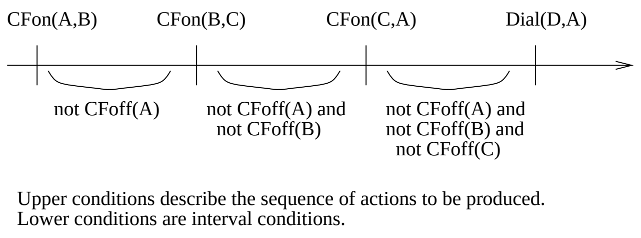

CFon(A,B)
CFon(B,C)
CFon(C,A)
Dial(D,A)
not CFoff(A)
not CFoff(A) and not CFoff(B)
not CFoff(A) and not CFoff(B) and not CFoff(C)
Upper conditions describe the sequence of actions to be produced.
Lower conditions are interval conditions.

Figure 3. Example of a behavioral pattern

6 Tool implementation and validation

The tool code represents 26000 lines of C++. Lutess has been used intensively during several case studies, among which the "Feature Interaction Detection Contest" held in association with the 5th Feature Interaction Workshop [7, 9]. The goal was to detect possible and undesired interactions between twelve telecommunication services. For this case study, the test process for each of the 78 configurations involved 10 to 20 sequences of 1000 to 10000 steps each. On the whole, each configuration has been tested for around 1 million test cases. The Lutess tool was run over 1500 times.

For this case study, we also considered applying a model-checker Lesar [10] to evaluate the ability of verification method to detect feature interactions [4]. Preliminary results show that the model-checker cannot deliver a result in most of the 78 configurations, because of lack of time and/or memory amount. On the contrary, Lutess always returns a verdict.

The generator obtained by compiling the environment constraints is coded using a symbolic notation in which the states are represented by a set of variables, and the transitions by boolean functions. These functions are implemented as a single Binary Decision Diagram (BDD) [1]. Building the BDD structure corresponding to a given environment is the most expensive part of the testing process. In our experiments, environments included between 32 and 45 constraints, plus up to 8-step patterns or 40 conditional probabilities. It was always possible to perform this computation and to run the test on a Sparc Ultra-1 station with 128 MB of memory. Maximum of required virtual memory amounts to 100 MB. Though, as the number of constraints describing the environment increases, the BDD complexity rises and its generation lasts longer. For the less-constrained environments that we produced, 6 seconds on CPU were necessary, while the most-constrained environments required 33 minutes for the corresponding BDD to be generated. As a comparison, a 1000 test run lasts 120 seconds once the BDD has been generated[4]. So, the more the environment is constrained, the more relevant is the test (since the whole test case is more realistic), but the longer is the BDD generation.

Several χ^2 tests were performed in order to check that the statistical methods produce data according to the different assumptions. Those assumptions are that the basic statistical method produces data in an equally-probable way, and that the method guided with conditional probabilities produces data with respect to the defined probabilities. Those assumptions appear to be valid.

7 Advantages, scope and limitations

In this section, we summarize the advantages we see to use Lutess. Then, we address the scope of the tool, and finally, we explore some of its limitations.

7.1 Advantages

The three components required by Lutess (the system under test, the environment description and the oracle) are just connected to one another and not linked into a single executable code. The construction of the test harness doesn't take much time.

Lutess offers a unified framework for synchronous program testing. Basically, a generator produces test data which satisfy an environment description. Lutess proposes different types of guidelines the user can use to describe a more realistic environment or make the test more relevant. Unlike the environment description, these additional guidelines are not to be strictly enforced. As a result, all valid behaviors are still possible, while the more reasonable

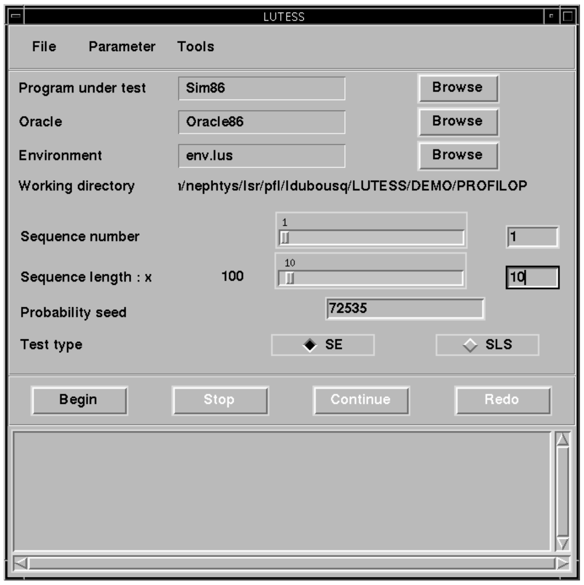
[4]This second phase of the testing process is proportional to the sequence length.

ones are more frequent. The model of the environment is thus more "realistic". The environment description and the guidelines have to be described in the same language (Lustre) and in the same framework (the testnode).

The use of conditional probabilities or patterns proved to be highly profitable when prototyping the application: these techniques allow to have a quick feedback on the correction of the implementation. Then, when it comes to validate the implementation (test its conformance to the specification), these techniques drive the environment to follow a realistic evolution. Meanwhile, thanks to the probabilistic aspect introduced in both methods, the behaviors of the environment may vary and involve rare and unforeseen scenarios. Such cases, close to the expected behavior -yet unexpected- are realistic and thus worth to be tested.

Lutess has a user-friendly interface (fig. 4). It offers the user an integrated environment:

- to define the environment description, the oracle and the unit to be tested, (in the fields *Program under test, Oracle,* and *Environment*);
- to command the construction of the test harness, and to build constrained random generators, (with *Begin, Stop* and *Continue* buttons;
- to set the random seed, the number and the length of the data sequences,
- to compile Lustre programs, to format the sequences of inputs, outputs and verdicts and to replay a given sequence with a different oracle, (with *Tools* menu and *Redo* button);
- to visualize the progression of the testing process, (in the message box, in the lower part of the interface).



Program under test: Sim86
Oracle: Oracle86
Environment: env.lus
Working directory: ı/nephtys/lsr/pfl/ldubousq/LUTESS/DEMO/PROFILOP
Sequence number: 1
Sequence length : x 100: 10
Probability seed: 72535
Test type: [✓ SE] SLS

Figure 4. Lutess interface

7.2 Scope

Lutess is a testing environment for synchronous reactive systems. Since an executable program (resp. oracle) is required, the unit under test (resp. oracle) source code language doesn't matter. If the program (resp. oracle) is developed in Lustre, Lutess can also be used to edit and to compile it.

Moreover, Lutess can be applied at different stages of the development. For instance, in the example given in this paper and during the FIW contest [9], we used Lutess to validate an executable *specification* of a telephony system.

7.3 Limitations

Lutess can only generate data for boolean input and output synchronous programs. We have always been able to by-pass this potential drawback yet, by using boolean vectors for enumerated data types.

For the moment, it is possible to use property-oriented testing in combination with conditional probabilities. But it isn't possible to use behavioral patterns with conditional probabilities. We are currently working on this point.

Specifying the software environment by means of invariant properties is a rather delicate task. Indeed, one should adequately choose a set of properties which do not "overspecify" the environment. Overspecifying may prevent some realistic environment behaviors from being generated.

It is difficult to evaluate when the test should be stopped. In fact, it is quite impossible to define a meaningful coverage criterion. For instance, classical coverage criteria (coverage of code instructions or branches of control flow graph) are very loosely related to the set of the possible program behaviors.

8 Related work

Jagadeesan et al. have presented a technique and a toolset that represent the most similar work to Lutess [12]. Compared to Lutess, this approach appears to be limited in several respects. The testing process is solely directed towards safety violations, and, thus, finds only errors related to this paradigm. Environment constraints are only taken into account to restrict the size of the input space. Inputs are selected with uniform weights. The whole process is based on the compilation of the oracle, the application and the test harness into one single executable code; recompiling is necessary after each modification, which caused the biggest dissatisfaction, according to what the authors said.

As we said before, Lutess can only generate data for synchronous programs with boolean inputs and outputs. In

[16], Halbwachs et al. describe another synchronous testing tool Lurette, which was built to take into account numerical data. Lurette requires also three elements, and like Lutess, needs a Lustre environment description. Lurette has no elaborated strategies for boolean data generation, but has a strategy for integer and real data generation.

9 Conclusion and future work

In this article, we presented Lutess, a highly automated testing environment for synchronous software and illustrated its use on an example. This automation allows to transfer the human efforts from the classical tester's chores (selecting the data, determining the result validity) to more defect prevention tasks (e.g., developing specifications).

Lutess offers several specification-based testing methods in order to fit the tester needs as well as possible. These methods aim at simulating more realistic environment behaviors, producing relevant data with respect to some properties or interesting situations. These methods produce test data using different type of guides, which are conditional probabilities, properties, and behavioral patterns.

We mainly conducted two experiments: a first case study of feature specification validation based on the ETSI recommendations [6], and a second one in the framework of the FIW contest [7]. Experience has confirmed that this approach is highly cost-effective. Both case studies showed that the guiding techniques were excellent at finding problems involving rare scenarios. This positive experience was reinforced by the valuable application of Lutess in the software specification stage, which helped get confidence in these specifications. All this has certainly contributed to make Lutess the "best tool" of the FIW contest [9].

Trace analysis is an important task, even if the verdict is automatic, since it can reveal unsuspected problems. Besides, writing relevant specifications in the appropriate format for test data generation should be facilitated. An environment to support these tasks is under consideration. It should integrate proving techniques to decide on formulae equivalence. Future directions also include criteria to determine when to stop testing and a notion of error coverage associated with the existing testing techniques.

References

[1] S. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516, 1978.

[2] A. Benveniste and al. Synchronous technology for real-time systems. In *The 1994 Real-Time Conferences*, pages 104–122, Teknea, 1994.

[3] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages (POPL 87), Munich*, pages 178–188. ACM, 1987.

[4] L. du Bousquet. Feature interaction detection using testing and model-checking, experience report. In *Formal Method*, Toulouse, France, September 1999.

[5] L. du Bousquet, F. Ouabdesselam, and J.-L. Richier. Expressing and implementing operational profiles for reactive software validation. In *9th International Symposium on Software Reliability Engineering*, Paderborn, Germany, 1998.

[6] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Incremental feature validation : a synchronous point of view. In *Feature Interactions in Telecommunications Systems V*, pages 262–275. IOS Press, 1998.

[7] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Feature interaction detection using synchronous approach and testing. *Computer Networks and ISDN Systems*, to be published, 1999.

[8] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering*. ACM, May 1999.

[9] N. Griffeth, R. Blumenthal, J.-C. Gregoire, and T. Otha. Feature interaction detection contest. In K. Kimbler and L. Bouma, editors, *Feature Interactions in Telecommunications Systems V*, pages 327–359. IOS Press, 1998.

[10] 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*, pages 785–793, september 1992.

[11] D. Hamlet and R. Taylor. Partition Analysis Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, pages 1402–1411, december 1990.

[12] L. Jagadeesan, A. Porter, C. Puchol, J. Ramming, and L. Votta. Specification-based Testing of Reactive Software: Tools and Experiments. In *19th International Conference on Software Engineering*, 1997.

[13] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, pages 14–32, march 1993.

[14] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, Monterey, USA, 1994.

[15] I. Parissis. *Test de logiciels synchrones spécifiés en Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, september 1996.

[16] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*. IEEE, 1998.

[17] J. Whittaker. *Markov chain techniques for software testing and reliability analysis*. PhD thesis, University of Tenessee, 1992.