Towards a testing methodology for reactive systems: a case study of a landing gear controller

Laya Madani[1], Virginia Papailiopoulou[1], Ioannis Parissis[2]
University of Grenoble
[1]Laboratoire d'Informatique de Grenoble
BP 53 - 38041 Grenoble Cedex 9 - France
[2]Laboratoire de Conception et d'Intégration des Systèmes
BP 54 - 26902 Valence Cedex 9 - France
{Laya.Madani, Virginia.Papailiopoulou}@imag.fr, ioannis.parissis@lcis.grenoble-inp.fr

***Abstract*—In this case study we test a landing gear control system of a military aircraft with the new version of LUTESS, a tool for testing automatically synchronous software. LUTESS requires the tester to specify the environment of the software under test by means of invariant properties in order to guide the test data generation. This specification can be enriched by operational profile specification in order to obtain more realistic scenarios. Moreover, test generation guided by safety properties makes possible to test more thoroughly the key features of the software, possibly under hypotheses on the software behaviour. In this case, the generator chooses input data which are able to violate the properties. The new version of LUTESS is based on constraint logic programming and provides some additional features (numeric inputs and outputs, hypotheses for safety guided testing, more powerful operational profiles). In this paper, we present the necessary steps for building the test model for LUTESS on a real case study from the avionics. This makes possible to better understand the applicability of the approach and to assess the difficulty and the effectiveness of such a process in real-world applications.**

I. INTRODUCTION

Synchronous programming [1] is widely used in designing and developing safety critical reactive systems as embedded programs used in avionics, transportation and energy. The synchronous hypothesis demands that the system reacts instantaneously to its inputs. In practice, this is verified if the system reacts to the events of the external environment before any evolution of the latter. This property is very useful for abstracting the problem of delay.

Many programming languages have been proposed to specify and implement synchronous applications, such as Esterel [3], Signal [7] or LUSTRE [5]. They ensure efficient code generation and provide formal specification and verification facilities. LUTESS [4], [12] is a testing tool that has been designed to automate the testing of synchronous reactive software; it is based on the LUSTRE language. In order to guide the test data generation process, the tool uses models of the program external environment, specified in a lustre-like language, that describe the behaviour of the program inputs. LUTESS translates this specification into test data generators, randomly simulating the software environment behaviour and feeding the software under test with input data sequences according to several testing techniques, such as operational profiles or the ability of test data to detect safety property violations. Recently, LUTESS has been entirely redesigned and uses Constraint Logic Programming (CLP) for the internal representation of the handled models (input-output FSM) as well as for the test generation [12], [14]. This new CLP-based version of LUTESS is able to handle numeric inputs and outputs and provides more powerful operational profiles. Safety property guided testing has also been adapted to deal with numeric inputs and outputs and has been extended to deal with hypotheses on the program under test.

Other tools have been proposed for testing synchronous programs. Gatel [8] is based on constraint logic programming as LUTESS, but, contrary to LUTESS, it is rather a "white box" testing tool. It translates the program and its environment specification in an equivalent Prolog representation and then statistically computes a test input according to a test objective. Furthermore, LUTESS generates test data with dynamic interaction with the system under test, while Gatel interprets the LUSTRE code. Lurette [11] is similar to LUTESS and it makes it possible to test LUSTRE programs with numeric inputs and outputs. A boolean abstraction of the environment constraints is first built: any constraint consisting of a relation between numeric expressions is assimilated to a single boolean variable in this abstraction. The concrete numeric expressions are handled by an ad hoc environment dedicated to linear arithmetic expressions. The generation process first assigns a value to the variables of the boolean abstraction and, then, tries to solve the corresponding equations to determine the values of the numeric variables. However, LUTESS uses constraint logic programming instead of an ad hoc resolution environment restricted to linear expressions, as Lurette does. Moreover, the LUTESS specification language is an extension of the LUSTRE language while Lurette uses ad hoc scenario description notations.

Recently [10], we proposed a testing methodology for synchronous reactive applications based on LUTESS. In this paper, we apply this approach on a specification of real critical embedded software: a landing gear control system of a military

This work is partially supported by SIESTA (www.siesta-project.com), a project of the ANR (the French Research Agency)

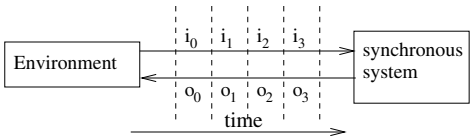

Figure 1. Synchronous software operation.

aircraft, that had been used in [2] to assess some formal verification tools. Our objective here is to assess the difficulties in the test modeling activity and in the test generation. Modeling requires translating the natural language specification into temporal invariants as well as into conditional probability assignments, the latter aiming at defining operational profiles [9] or execution scenarios. Hence, the contribution of this paper is twofold. On the one hand, we illustrate the necessary steps during the test model construction, focusing mostly on the safety property violation. On the other hand, we check the applicability and the scalability of our approach, as far as the latter can be assimilated to the length and the complexity of the specification, and the resources needed to the test generation.

The paper is divided in four main sections. Section 2 provides a brief overview of the essential concepts on testing synchronous software using LUTESS. The case study is presented in Section III. Section IV thoroughly shows the different steps for constructing the test models in order to generate valid and meaningful input sequences with regard to the system specifications. Section V discusses the evaluation aspects of the approach. Finally, we provide experimental data and concluding remark from the case study.

II. LUTESS: A TESTING ENVIRONMENT FOR SYNCHRONOUS PROGRAMS

Synchronous programs have a cyclic behaviour according to a discrete global clock (see Fig. 1): at each tick of the clock, all inputs are read and processed simultaneously, and all outputs are emitted. In practice, a synchronous program reacts to its environment requests before any evolution of this environment. This means that at any instant (say t) the program reads the inputs (i_t) and emits the outputs (o_t) before the new inputs i_{t+1} are available.

LUTESS [4], [12], [14] is an environment for testing synchronous software, based on LUSTRE, a synchronous declarative data-flow language [5]. Within LUSTRE, any variable or expression represents an infinite sequence of values and takes its n-th value at the n-th cycle of the program execution. LUSTRE offers usual arithmetic, boolean and conditional operators and two specific operators: the (`pre`) operator which refers to the *previous* value of an expression, and the `followed-by` (`->`) operator which is used to set the initial value of a flow. Let `E` and `F` be two expressions of the same type denoting the sequences of values $(e_0, e_1, ..., e_n, ...)$ and $(f_0, f_1, ..., f_n, ...)$; f_i is the value of `F` at instant i. Then `pre(E)` = $(nil, e_0, e_1, ..., e_n, ...)$ where nil is an undefined value; while `E->F` denotes the sequence $(e_0, f_1, ..., f_n, ...)$. A LUSTRE program is structured into nodes; a node is a set of equations which defines the node outputs as functions of its inputs. Once a node is defined, it can be used inside other nodes like any other operator. LUSTRE is an executable specification language, which also provides the main characteristics of a linear temporal logic of the past [6]. Therefore, temporal logic formulas can be easily implemented as LUSTRE programs. The user can define her/his own logical or temporal operators to express invariants or properties.

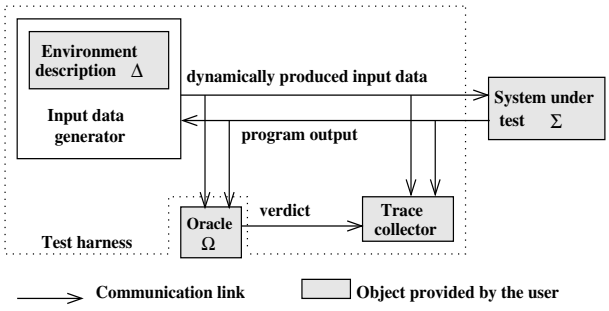

Figure 2. LUTESS architecture.

To perform the test operation, LUTESS requires three components (see Fig. 2): the software environment description (Δ), the executable code of the system under test (Σ) and a test oracle (Ω) describing the system requirements. The system under test and the oracle are both synchronous executable programs. The environment description is composed of a set of properties, stated as invariants, that the environment of the system is assumed to satisfy. These invariants are LUSTRE boolean expressions and constrain the possible behaviours of the environment.

LUTESS builds a random generator from the environment description (i.e. the test specification.) as well as a test harness which links the generator, the system under test and the oracle. LUTESS coordinates their execution and records the inputs and outputs as well as the associated oracle verdicts, thanks to the trace collector. The test is operated on a single action-reaction cycle: the generator randomly selects a valid input vector which respects the environment properties and sends it to the system under test. The latter 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 properties are violated. The testing process is stopped when the user-defined length of the test sequence is reached. Input data generation is random and can be guided by *operational profiles* specification or by the *safety properties*. In the case of operational profiles the selection of the program inputs is performed with respect to probabilities specified by the tester, while safety property guided testing leads the test generation towards situations that could violate the program properties. *Hypotheses* on the program under test can also be specified in order to assist the computation of test data for safety property guided testing.

The software environment description is specified in a separate LUSTRE node called a *testnode*. A testnode has as inputs the outputs of the software under test and inversely, a testnode outputs are the software inputs. The general form of a testnode is given in Fig. 3. There are four operators especially introduced for testing purposes:

```
testnode Env(<SUT outputs>) returns (<SUT inputs>);
var <local variables>;
let
   environment(Ec1,Ec2, ....,Ecn);
   prob(C1,E1, P1);
   ...
   prob(Cm,Em, Pm);
   safeprop(Sp1, Sp2, ...., Spk);
   hypothesis(H1,H2, ....,Hl);
   <definition of local variables>;
tel;
```

Figure 3. Testnode syntax.

- The `environment` operator makes it possible to specify invariant properties of the program environment that always hold during program execution.
- The `prob` operator is used to define conditional probabilities or to create operational profiles. The expression `prob(C,E,P)` means that if the condition `C` holds then the probability of the expression `E` to be true is equal to `P`. `C` and `E` are boolean expressions, while `P` is a real constant within the interval $[0.0 \ldots 1.0]$.
- The `safeprop` operator is used to guide the test generation process towards situations where the safety properties could be violated. The expression `safeprop(Sp)` means that test inputs apt to violate the property `Sp` will be preferably chosen by the generator.
- The `hypothesis` operator is used as a complement to `safeprop` in order to insert assumptions on the program under test which could improve the fault detection ability of the generated data [14]. The expression `hypothesis(H1)` is used to formally introduce such hypotheses in a testnode.

III. Case study: a landing gear control system

A. General description of the system

We study the landing gear control system of a military aircraft [2], in charge of maneuvering landing gears and associated doors. The system is controlled by the command software in normal mode or analogically in emergency mode. It is composed of 3 landing gears: front, left and right. Each gear has an up lock box and a door with two latching boxes. Gears and doors are controlled by an hydraulic system. The figure 4 shows a general description of this system which consists of :

- a set of actuators:
  - a valve to isolate the emergency system;
  - electrovalves to open or close the doors and let down or retract gears;
- a set of sensors giving the state of each component of the system.

To command the retraction and outgoing of gears, the pilot has a set of 2-position buttons and a set of lights giving the current positions of doors and gears. When the command line is working, the controller obeys the orders of the pilot, within the control software, executing a sequence of actions, directly animating the mechanics. The outgoing of gears is decomposed as follows:

1) stimulation of the solenoid isolating the command unit;

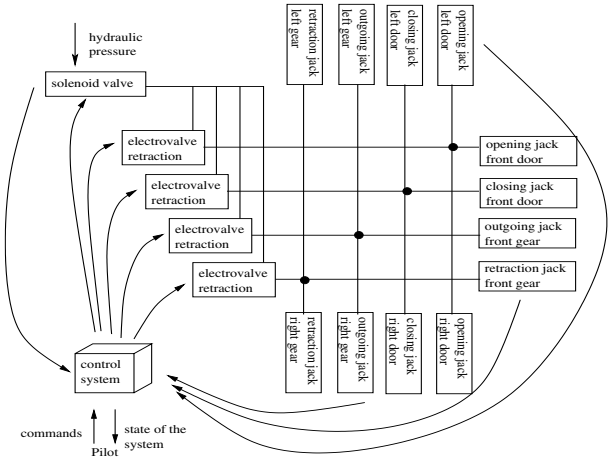

Figure 4. The landing gear system.

2) stimulation of the door opening solenoid;
3) once the doors are opened, stimulation of the gear outgoing solenoid;
4) once the gears are locked down, stop the stimulation of the gear outgoing solenoid and start the stimulation of the door closure solenoid;
5) once the doors are closed, stop the stimulation of the door closure solenoid;
6) finally, stop the isolating electrovalve.

The retraction sequence is symmetric. These two sequences lead from the "cruise state" (i.e. gears up and doors closed) to the "landing state" (i.e. gears down and doors closed) and conversely.

Because of hydraulic constraints, a given timing must exist between stimulation and stop of the valves. Moreover, the sequences should be interruptible if a counter order is given by the pilot, for example a retraction order during the let down sequence.

B. The control software

The control software handles the landing gear physical system in normal mode (no failure), receiving data from the sensors or pilot orders, and producing commands for the physical system. It is decomposed in 3 main functions:

- a monitoring function for gears and doors signaling inconsistencies;
- a monitoring function checking that the system reacts correctly and controlling the functional and temporal coherence of the orders;
- a command function implementing the sequence of outgoing and retraction of gears. This function gives direct orders to the physical system and is decomposed in two main parts:
  - a function computing stimulation command for each component;
  - functions managing the emission of the command (timing constraints).

All input and output variables of the control system are boolean. Through this paper we will consider a single gear, as the three gears are identical.

The input variables are:

- `Lever_Up`: true if the lever is in up position for retracting the gears, false if it is in down position for outgoing the gears.
- `Gears_Off`: true if the shock absorbers are relaxed (the aircraft is flying).
- 7 variables indicating the initial state of the gears and the doors, for instance:
  - `Init_Closed_Door_Up_Gear`: true if the door is closed and the associated gear is up;
  - `Init_Maneuvering_Door_Up_Gear`: true if the door is moving and the associated gear is up;
  - `Init_Open_Door_Up_Gear`: true if the door is opened and the associated gear is up.
- 12 variables providing the state of the gear, the door, the oil pressure and the solenoid valve in case of blocking (failure), for instance:
  - `Gear_Up_Fail`: true if the gear is blocked in the up position;
  - `Gear_Down_Fail`: true if the gear is blocked in the down position.

The output variables are:

- 5 variables giving commands to start and stop stimulation of the electrovalves and solenoid valve:
  - `Solenoid_Valve_Stimulation`: true if the solenoid valve is stimulated for allowing the fluid to pass;
  - `Gear_Outgoing_Stimulation`: true if the electrovalve of outgoing the gears is stimulated for allowing the fluid to pass;
  - `Gear_Retraction_Stimulation`: true if the electrovalve of retracting the gears is stimulated for allowing the fluid to pass;
  - `Door_Opening_Stimulation`: true if the electrovalve of opening the doors is stimulated for allowing the fluid to pass;
  - `Door_Closing_Stimulation`: true if the electrovalve of closing the doors is stimulated for allowing the fluid to pass.
- 6 variables indicating to the pilot the state of gear, the door and the oil pressure after the solenoid valve. The controller calculates these states by considering the pilot actions (the lever movement), the necessary time for the outgoing and the retraction of gears, the opening and the closing of the doors as well as the solenoid valve state in case there is no failure:
  - `Gear_Up`: true if the gear is in up position;
  - `Gear_Down`: true if the gear is in down position;
  - `Door_Closed`: true if the door is closed;
  - `Door_Open`: true if the door is opened;
  - `Under_Pressure_Manometer`: true if the pressure is present in the hydraulic system after the solenoid valve;
  - `Analogic_Link_On` : true when the order coming from the lever is allowed to be transmitted.
- other variables aiming at warning the pilot in case of malfunction or non response of mechanical components.

IV. Testing the landing gear control system with Lutess

The generation of test sequences with Lutess requires a model of the system environment, the so-called test model, constructed according to the system specification. At first, we consider that all system components function correctly, assuming no failures, providing the correct inputs to the controller. Although the test models are specific to the program under test, we claim that the modeling and testing process can follow a general incremental approach:

1) **Domain definition:** Definition of the domain of values that integer or boolean inputs can take at a given instant, regardless of the system evolution (e.g. an integer input corresponding to a temperature value in Celcius degrees could not be lower than 270).
2) **Environment dynamics:** Specification of different temporal relations between the current input values and past input or/and output values. These relations often include, but are not limited to, the physical constrains of the environment (e.g. the temperature variation cannot be higher than 2°C between two successive instants; the temperature would raise if a heater is on).

   The above specifications are introduced in the testnode by means of the `environment` operator. Simple random test sequences can be generated, without a particular test objective, but considering all inputs allowed by the environment.

3) **Test scenarios:** In order to test specific properties of the system, the tester can define more precise scenarios, by specifying additional invariant properties or conditional probabilities. The Lutess `prob` operator provides a mean for specifying conditional probabilities which can be used either to force the test data generator to conform to realistic scenarios, either to simulate failures (e.g. when the heater is on, there is a probability of 90% for the temperature to raise).
4) **Property-based testing:** In this step, the tester might use formally specified safety properties in order to guide the generation towards the violation of such properties. In this case, the generator removes inputs that, given the property expression, cannot lead to its violation; instead, those inputs that can violate the property will be preferably chosen. In order to be effective, this guidance requires the safety properties to be expressed as a relation between inputs implying some outputs (e.g. when the temperature is high, then the heater must be on). Test hypotheses on the software behaviour can also be introduced and possibly help this kind of generation (e.g. the program will turn the heater on only if the on/off button has been pushed).

These steps are described in detail in the following. The particularity in this case study is that, since the system handles only boolean variables, controlling the input variables and the environment dynamics is rather limited. Therefore, we examine more closely the use of conditional probabilities and

Table I
EXCERPT OF A TEST CASE USING ONLY INVARIANT PROPERTIES

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
Lever_Up	1	0	1	0	0	0	0	1	0	1	1	1
Gears_Off	0	0	0	0	1	0	1	1	1	1	1	0
Init_Closed_Door_Down_Gear	1	0	0	0	0	0	0	0	0	0	0	0
Gear_Up_Fail	0	0	0	0	0	0	0	0	0	0	0	0
Gear_Down_Fail	0	0	0	0	0	0	0	0	0	0	0	0
Gear_Up	0	0	0	0	0	0	0	0	0	0	0	0
Gear_Down	1	1	1	1	1	1	1	1	1	1	1	1
Door_Closed	1	1	1	1	1	1	1	1	1	1	1	1

the possible violation of the safety properties (Sections IV-C and IV-D).

A. Domain definition

Domain definition constraints determine physical constraints on an input variable value or on the combination of variable values. This step is used so that the generator produces meaningful values within the set of values that an input is designed to receive. It can refer to integer inputs, which are often used in controllers to represent either a state, consisting of a limited subset of integers, or simply an interval of integers. Dependencies between software input values (integer or boolean) can also be expressed. For instance:

- Assuming that in the first place, we test the controller in its normal functioning, we can state that there is no failure in the components. For instance, the corresponding formal expression of this property in case of a failure in the gears[1] is:
  `not Gear_Up_Fail and not Gear_Down_Fail`
  Of course, this specification may be modified or removed if failures must be taken into consideration.
- At the first step (t_0), there is only one active initial state variable for the doors and the gears and it is active only at this step:
  ```
  (one_active(Init_Closed_Door_Up_Gear, ...) ->
  true)
  and
  true -> neg_conj(Init_Closed_Door_Up_Gear, ...)
  ```
  where `one_active` and `neg_conj` are two user-defined boolean operators, especially built so that the code is easily maintainable and readable. The first one handles the 7 variables that indicate the initial state of the gears and the doors (presented in Section III-B) and allows exactly one of them to be true at the initial step. Similarly, `neg_conj` operates on the same variables prevent them from being true.

The above, very simple, test model, can be directly used to generate test input sequences. Test inputs will be randomly generated at every instant and their values will be chosen in the above defined domains. Nevertheless, these sequences may not be very conclusive, as it is usually the case with random testing. At the next step of the test methodology, to cope with this issue, we specify more thoroughly the environment dynamics, in order to observe more meaningful executions.

[1]The expressions for the other components failures are similar and they are not presented here in sake of simplicity.

B. Environment dynamics

Environment dynamics can be expressed as temporal relations between the current and past values of the system inputs and outputs. As it is mentioned above, in this case study, there are very few of such properties, because of the nature of the variables. Indeed, most of them are directly expressing human actions and only few variables (mainly related to failures) are subject to physical constraints. However, in this test model this feature can be used to express the absence of failures, for instance by means of the following property:

- When the aircraft reaches the ground, the gears are locked down and the doors are closed:
  ```
  implies(not Gears_Off,
  Init_Closed_Door_Down_Gear)->
  implies(falling_edge(Gears_Off), pre Gear_Down
  and pre Door_Closed)
  ```
  The user-defined node `falling_edge` returns a true value when its input was previously true (at instant t_{n-1}) but currently (at instant t_n) it is false.

Table I shows a generated test case resulting from the above test model using invariant properties that define the variable domains and the environment dynamics. The domain constraints are indeed verified, but the test sequences are not absolutely convincing. The command lever moves quite often and prevents many states from being generated (mainly the complete deployment and retraction of the gears). These states need for the lever to be stable for some time. We can also notice that the aircraft lands (`Gears_Off=false`) whenever the gears and the doors are in the correct position (`pre Gear_Down=true` and `pre Door_Closed=true`).

C. Test Scenarios

The previous test cases are generated by specifying invariant properties of the program environment. However, this trace of test cases does not represent a realistic system behaviour. For instance, the lever must be kept in the same position for a given time period in order to change the state to "cruise state" or "landing state" (c.f. Section III-A). So, we need to guide more thoroughly the `Lever_Up` variable. In LUTESS, specific execution scenarios can be obtained either by defining invariant properties or by specifying different conditional probabilities. In this case, we could use the following invariant property scenario (using the `environment` operator):

When the pilot puts the lever in down position for deploying the gears, he maintains it in this position until reaching the "landing state" (gears down and doors closed), and conversely:

Table II
EXCERPT OF A TEST CASE USING THE SCENARIOS SPECIFIED IN SECTION IV-C

	t_{50}	t_{51}	t_{52}	t_{53}	t_{54}	t_{60}	t_{61}	t_{62}	t_{63}	t_{64}	t_{65}	t_{72}	t_{73}	t_{74}	t_{75}	t_{76}	t_{77}	t_{78}
Lever_Up	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	0	0	0
Gears_Off	1	**0**	**0**	**0**	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Gear_Up	0	0	0	0	0	0	**0**	**0**	**0**	**0**	**1**	1	1	1	1	1	1	1
Gear_Down	1	1	1	1	1	**1**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0	0
Door_Closed	1	1	1	1	1	0	0	0	0	0	0	**0**	**0**	**1**	1	1	1	1
Door_Open	0	0	0	0	0	1	1	1	1	1	1	**0**	**0**	0	0	0	0	0

Table III
EXCERPT OF A TEST CASE USING SCENARIO CONCERNING UP POSITION FAILURE SPECIFIED IN SECTIONIV-C

	t_{103}	t_{104}	t_{105}	t_{106}	t_{195}	t_{196}	t_{197}	t_{198}	t_{199}	t_{200}	t_{380}	t_{381}	t_{382}	t_{383}	t_{384}	t_{385}
Gear_Up_Fail	0	0	0	0	0	0	0	0	0	0	0	**1**	0	**1**	0	0
Gear_Up	1	**1**	**1**	0	1	**1**	**1**	0	0	0	**1**	**1**	**1**	**1**	**1**	0
Gear_Outgoing_Stimulation	0	**1**	**1**	1	0	**1**	**1**	1	1	1	**1**	**1**	**1**	**1**	**1**	1

```
  implies (pre not Lever_Up and
(pre not Gear_Down or pre not Door_Closed), not
Lever_Up)
  and
  implies(pre Lever_Up and
(pre not Gear_Up or pre not Door_Closed), Lever_Up)
```

The same scenario may be expressed differently by specifying a certain conditional probability and taking into consideration the time required for a full gear deployment or retraction. In fact, all functions included in the control system are implemented by periodic and sequential processes executed every 40ms. More specifically, according to the system implementation provided in [2], the time needed to fully deploy the gear from the "cruise state" or to fully retract it from the "landing state" corresponds to 36 execution cycles at most[2]. As a result, the property could be expressed by means of the `prob` operator as follows:

- When the lever is in down/up position, it is highly probable that it will remain in this position. Let us assume that the probability is fixed at 97%. So, since 36 steps are needed to fully deploy/retract the gear, the probability to get a sequence of 36 successive steps with the same lever position is approximately $1/3(0.97^{36} \approx 0.33)$.
  ```
  prob(true->pre Lever_Up, (Gears_Off and
  Lever_Up)->Lever_Up, 0.97);
  prob(false->not(pre Lever_Up), ((not Lever_Up)
  and (not Gears_Off))->not Lever_Up, 0.97);
  ```

Clearly, every property in an `environment` operator could be replaced by a similar `prob` expression. The value of the assigned probability must be empirically determined by the tester. Such scenarios describe non-deterministic behaviours.

In another scenario, we could consider the plane in its landing position:

- When the gears have been locked down, the pilot should most probably land the plane:
  ```
  prob(false->pre Gear_Down and pre Door_Closed
  and pre Gears_Off, true->not Gears_Off ,0.9);
  ```
- When the aircraft is on the ground, it may stay there for some time:
  ```
  prob(false->pre not Gears_Off, true->not
  Gears_Off,0.6);
  ```

Table II shows a generated test case for the scenarios specified previously. We notice that the lever rarely changes its position, allowing to observe a full retraction of the gear. In addition, the aircraft remains on the ground for a few cycles (t_{51} to t_{53}). Most importantly, the changes in the gears position and the door state become more clear. For example, at instant t_{60}, the gear is down and until the moment it goes up (t_{65}), it is neither down nor up. Similarly, the door moves (it is neither open nor closed) before it is closed (t_{72} to t_{74}). Eventually, after a few cycles, the aircraft goes in "cruise state" (gear up and door closed) due to the long duration that the lever was held in the up position.

Test scenarios can also be used to simulate system failures. In the present case study, failures are mainly caused by blocked physical components. Such erroneous situations can only take place when a stimulation of a component is issued. For example, let us consider a system failure described by the following scenario: "when the gear is up and a stimulation for its outgoing starts, the sensor can detect the failure in this position of the gear". In order to simulate this scenario using LUTESS, we should replace the environment constraint that prevents the failure of the gear in its up position (`not Gear_Up_Fail`) seen in section IV-A by the following property:

- If there is a failure in the gear in the up position, then the gear was up and there was an outgoing stimulation at the previous cycle :
  ```
  not Gear_Up_Fail -> implies(Gear_Up_Fail, pre
  Gear_Up and pre Gear_Outgoing_Simulation);
  ```
  The absence of failure at the first execution cycle is important in order to reassure that the system starts up normally.

We can also specify the probability for the occurrence of this failure when the right part of the previous implication is verified. To do so, a possible scenario could be the following:

- If the gear is up and there is a stimulation for its outgoing, then there is a small probability that the gear will be blocked in the up position:
  ```
  prob(false->pre Gear_Up and pre
  Gear_Outgoing_Stimulation, Gear_Up_Fail, 0.2);
  ```

Table III shows a generated test case for this scenario simu-

[2]Actually, the initial experiment was intended to verify this property for a 14-seconds time period. However, since explosion of the number of states occurs partly due to timings issued in the system and in the property, timing was divided by 10 (both in the system and the property). Thus, the required time was considered to be equal to 1.4 seconds, which corresponds to approximately 36 execution cycles.

Table IV
EXCERPT OF A TEST CASE GUIDED BY A SAFETY PROPERTY

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
Gears_Off	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	1
Analogic_Link_On	1	1	1	1	1	1	1	1	1	1	1
Gear_Retraction_Simulation	0	0	0	0	0	0	0	0	0	0	0

lating a failure of the gear and the corresponding controller reaction. We observe that the failure occurs only when the gear was up and the gear outgoing stimulation was on at the previous instant. Since the value of the specified probability is rather low, the failure rarely occurs, even if the precondition is verified (e.g. at instants t_{104}, t_{105} or t_{196}, t_{197}).

Note that there is no support for statically checking the consistency of the defined probabilities. Therefore, it may happen that the defined conditional probabilities cannot be satisfied. For this case, two options are implemented in LUTESS [13]: (1) the test operation terminates and the tester must modify the assigned probabilities; (2) the generator tempts to satisfy as much constraints as possible in order to generate input values, so it ignores the specification causing the inconsistency and continues the generation process.

D. Property-based testing

The `safeprop` operator makes it possible to guide the test data generation according to the ability of test inputs to violate a property. Property-based testing guides the test generation by avoiding, when possible, input values that cannot lead to a property violation.

First, we show a simple case of property violation, considering the following property: "there is no retraction of the gears when the aircraft is on the ground, even if the pilot acts on the lever to achieve that". This property is formally expressed as follows:

```
safeprop(implies(not Gears_Off, not
Gear_Retraction_Stimulation or not
Analogic_Link_On))[3];
```

Violating this property requires setting the left part of the implication to true.

Table IV shows the effect of the above specification on the generated test cases. `Gears_Off` is set to false, which as specified, is a necessary condition to violate the property; therefore, this trace helps to observe if the property is verified by the program.

Let us consider a more sophisticated situation, in which the property to be tested is: "if the door is closed or maneuvering, then the gears do not move (remain in the up or in down position)". The formal expression of this property is:

```
safeprop(implies(Door_Closed or (not Door_Closed
and not Door_Open), Gear_Up or Gear_Down));
```

For this property, setting the left part of the implication to true is more complicated, since it contains exclusively some of the program outputs, which cannot be directly handled. The tester can only lead the system into such a state, so that the system gives the desirable outcome at the output. To this end, we use the keyword `hypothesis`, which makes it possible to introduce into the test generation process some knowledge on the behaviour of the program under test. Such hypotheses can provide information, even incomplete, on the way that outputs are computed and hence ease the computation of inputs during safety property guided testing.

[3]The physical action of moving the gear is allowed by the combination of the stimulation and the open state of analogic link.

Table VI
EXCERPT OF A TEST CASE WITH ORACLE VIOLATION

	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}
Gears_Off	**0**	**0**	**0**	1	1	1
Analogic_Link_On	**1**	**1**	**1**	1	1	1
Gear_Retraction_Simulation	**1**	**1**	**1**	0	0	0
Gear_Up	0	0	0	**0**	**0**	**0**
Gear_Down	1	1	1	**0**	**0**	**0**
Door_Closed	1	1	0	**0**	**0**	**0**
Door_Open	0	0	0	**0**	**0**	**0**
Oracle	**0**	**0**	**0**	**0**	**0**	**0**

Some of the hypotheses we could use in order to close the door or maintain it in its position are:

- The software ensures that if the lever is maintained in the same position for the required period (36 cycles) so that the outgoing/retraction operation is finished, the door will be closed:
  ```
  hypothesis(implies(maintained(36, not Lever_Up)
  or maintained(36, Lever_Up), Door_Closed));
  ```
- If the lever is maintained in the down position for the same duration, for the same reason, the gear will be down and vice-versa:
  ```
  hypothesis(implies(maintained(36, not Lever_Up),
  Gear_Down));

  hypothesis(implies(maintained(36, Lever_Up),
  Gear_Up));
  ```

Doubtlessly, program analysis, usually thorough enough, results in this kind of hypotheses. Alternatively, hypotheses might concern some program properties that are considered as satisfied, because they have been successfully tested before. In any case, this step requires the tester to be well trained and experient as well as to be familiar with the system under test.

Table V illustrates the generated test cases from the above specification, where the test generation is guided by a safety property combined with hypotheses on the system. In a sample of 1000 test sequences, the generator managed to rise the number of cases in which the left part of the implication in the safety property is true. Indeed, the door remains closed (`Door_Closed=false`) or the door is maneuvering (`Door_Closed=false` and `Door_Open=false`).

V. EVALUATION ASPECTS

So far, the test oracle (Ω, in Fig. 2) that describes the system requirements has not been taken into account during testing analysis. Towards our attempt to evaluate the LUTESS

Table V
EXCERPT OF A TEST CASE GUIDED BY A SAFETY PROPERTY AND CERTAIN HYPOTHESES ON THE SYSTEM

model with hypotheses	t_{103}	t_{104}	t_{105}	t_{106}	t_{107}	t_{108}	t_{411}	t_{412}	t_{413}	t_{414}	t_{415}	t_{416}	t_{417}
Door_Closed	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**1**	**1**
Door_Open	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
model without hypotheses													
Door_Closed	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Door_Open	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Table VII
EXCERPT OF A TEST CASE WITH ORACLE VIOLATION SIMULATING A FAILURE IN THE UP GEAR

	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	t_{18}	t_{19}	t_{20}	t_{21}	t_{22}	t_{23}
Gear_Up_Fail	0	1	0	0	0	0	0	0	0	1	**1**	0	0
Gear_Up	1	1	1	1	1	1	1	1	1	1	**0**	0	0
Gear_Outgoing_Stimulation	1	1	1	1	1	1	1	1	1	1	1	1	1
Oracle	1	1	1	1	1	1	1	1	1	1	**0**	1	1

testing methodology in terms of its fault detection ability, we introduced certain errors in the program code and, by observing the oracle verdict, we were able to verify if the intentional errors were detected. More precisely, we considered some operator changes in the original program, that is, replacements of some operators by others so that the obtained program is syntactically correct. First, we modified the module that processes the commands concerning the gears and the doors; in the equation that determines the gears retraction command (`Gear_Retraction_Stimulation`) in function of the gears position, the doors state and the shock absorbers state, the `and` operators were replaced by `or` operators. Indeed, in the original program, in order to stimulate gears retraction, the gears must not be up, the doors must be open and the shock absorbers must be relaxed (`Gears_Off`); this last condition indicates that the plane is flying. On the contrary, due to the introduced error, at least one of the above conditions is sufficient for the gear retraction stimulation. As for the environment description, we used a test model which combines the scenarios described in Section IV-C, except the one of failure, and the first safety property presented in Section IV-D. Lastly, in the oracle, we included both the safety properties explained in Section IV-D.

It is worth mentioning that in 94 out of 100 steps, the oracle was violated, hence guiding the test data generation by safety properties results in test sets adequate to validate these properties. An excerpt of the results obtained by the above simulation and the oracle verdict are demonstrated in Table VI. At steps t_7 to t_9, the first property is violated, whereas at steps t_{10} to t_{12} the second property is violated.

In addition, we attempted to detect a system failure, taking into account the failure simulation concerning the gear position, that described in Section IV-C. In this case, we used the environment description with the failure simulation and in the oracle, we added the following property:

- If a gear is blocked in the up position (`Gear_Up_Fail`), then the gear must be up (`Gear_Up`), since this variable value is sent back to the pilot:
  `implies(Gear_Up_Fail, Gear_Up);`

In the program under test, we introduced a fault in the module that handles the system components. In particular, in the equation that counts the required time (execution cycles) for moving the up gear, we removed the `not` operators, since the beginning and the interruption of this counter depends on the gear position, i.e. if it is blocked or not in the up position. The oracle violation resulting from this configuration is shown in Table VII. At step t_{21}, the property concerning the failure is not verified.

Furthermore, an important remark is that in case we do not use property-based testing (i.e. guide the test generation only by invariant properties and test scenarios), the violation of the property is not systematic and depends, in particular, on the seed used to initialize the random generator of LUTESS. On the contrary, when test generation has been guided by the safety property, the latter has been always violated, regardless of the seed.

VI. CONCLUDING REMARKS

The landing gear control system requires exchanging an important number of messages between the system controller and the physical system. The main program is composed of 32 internal functions and handles 21 input and 29 output boolean variables. Each testnode consists of about 20 properties modeling the system environment to which are added several conditional probabilities, safety properties and hypotheses on the system. Test were performed on a P4 2.8 GHz single core with 1 GB of memory under Linux Fedora 9. It takes approximately less than 30 seconds to generate a sequence of hundred steps, for any of the test models we used. The time seems to linearly increase with regard to the number of properties used in the test model (as they are relatively close in complexity). Also, for the same environment model, time increases linearly with the number of steps.

Modeling the land gear controller environment required 30 working days of an undergraduate student. Almost 30% of them were dedicated to learning to use LUTESS. Nevertheless, the effort required to build the test models is more difficult to assess accurately. Actually, this depends on the desired thoroughness of the test sequences which may lead the tester to write several scenarios corresponding to different situations and resulting in different test models. However, building a new testnode in order to generate a new set of test cases

usually requires a slight modification of a previous testnode (adding/deleting some properties). Therefore, a big set of test cases may be generated with relatively small effort, which could be an asset to the proposed methodology when compared to the manual test data construction, which is usually used in practice for this kind of systems.

VII. Conclusion

In this paper, we applied a testing methodology, based on LUTESS, on a real-world application, a landing gear control system. This testing approach requires modeling the system environment according to the system specifications; to do so, four main steps should be carried out. The objective of this case study was to evaluate the test modeling difficulty and the test generation complexity of LUTESS on a real aeronautic system. Even though the system under test deals only with boolean variables, its complexity in terms of both size and functions allows to obtain some quite promising results. We concentrated mostly on the last steps of the testing procedure, aiming at demonstrating how the tester can achieve more thorough and valid test sequences with regard to the system functional requirements. Moreover, we deliberately introduced specific errors in the program under test in order to evaluate the fault-detection ability of the tool. Even if more case studies are needed to evaluate the applicability and the scalability of this testing technique, this experiment suggests that the methodology and the tool could be suitable for real-world industrial applications.

Future work includes extending LUTESS to handle float numbers. Such an extension requires defining an adequate enumeration method during the constraint resolution.

References

[1] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[2] F. Boniol, V. Wiels, and E. Ledinot. Experiences in using model checking to verify real time properties of a landing gear control system. In *ERTS 2006: 3rd European Congress Embedded Real Time Software*, Toulouse, France, January 25-27 2006.

[3] F. Boussinot and R. de Simone. The esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[4] L. du Bousquet, F. Ouabdesselam, I. Parissis, J.-L. Richier, and N. Zuanon. Lutess : a testing environment for synchronous software. In *Tool Support for System Specification, Development and Verification*, pages 48–61. Advances in Computer Science. Springer Verlag, June 1998.

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

[6] Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language lustre. *IEEE Trans. Softw. Eng.*, 18(9):785–793, 1992.

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

[8] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions: Gatel. In *ASE*, pages 229–, 2000.

[9] John D. Musa. Operational profiles in software-reliability engineering. *IEEE Softw.*, 10(2):14–32, 1993.

[10] V. Papailiopoulou, B. Seljimi, and I. Parissis. Revisiting the Steam-Boiler Case Study with Lutess: Modeling for Automatic Test Generation. In *12th European Workshop on Dependable Computing*, Toulouse, France, May 2009.

[11] Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs, and Daniel Weber. Automatic testing of reactive systems. In *IEEE Real-Time Systems Symposium*, pages 200–209, 1998.

[12] Besnik Seljimi and Ioannis Parissis. Using CLP to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *ISSRE*, pages 105–116, 2006.

[13] Besnik Seljimi and Ioannis Parissis. Using clp to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *ISSRE*, pages 105–116, 2006.

[14] Besnik Seljimi and Ioannis Parissis. Automatic generation of test data generators for synchronous programs: Lutess v2. In *DOSTA '07: Workshop on Domain specific approaches to software test automation*, pages 8–12, New York, NY, USA, 2007. ACM.