Revisiting the Steam-Boiler Case Study with LUTESS : Modeling for Automatic Test Generation

Virginia Papailiopoulou, Besnik Seljimi, Ioannis Parissis

Revisiting the Steam-Boiler Case Study with LUTESS : Modeling for Automatic Test Generation*

Virginia Papailiopoulou[1], Besnik Seljimi[1], Ioannis Parissis[2]
{virginia.papailiopoulou, besnik.seljimi}@imag.fr, ioannis.parissis@lcis.grenoble-inp.fr
University of Grenoble - France
[1]Laboratoire d'Informatique de Grenoble, [2]Laboratoire de Conception et d'Intégration des Systèmes

Abstract

LUTESS *is a testing tool for synchronous software making possible to automatically build test data generators. The latter rely on a formal model of the program environment composed of a set of invariant properties, supposed to hold for every software execution. Additional assumptions can be used to guide the test data generation. The environment descriptions together with the assumptions correspond to a test model of the program. In this paper, we apply this modeling principle to a well known case study, the steam boiler problem which has been presented in the past. The aim of this work is to illustrate the process of building the test model and to assess the difficulty of such a process in a realistic case study. The steam boiler case study is a quite suitable problem to use, in point of both problem size and complexity, for our purposes. Taking advantage of the new features recently added in* LUTESS *, we show a way of defining a test model so that the testing is efficient.*

1. Introduction

Synchronous programming [2] is widely used in safety-critical domains such as avionics, transportation and energy. The synchronous approach requires the software to react to its inputs instantaneously. In practice, that means that its reaction is sufficiently fast so that every change in the external environment is taken into account. As soon as the order of all the events occurring both inside and outside the program are specified, temporal constraints describing the behavior of a synchronous program can be expressed. Many programming languages have been proposed to specify and implement synchronous applications, such as Esterel [3] or Lustre [5]. They ensure efficient code generation and provide formal specification and verification facilities.

Several testing tools have been proposed for programs specified in Lustre. In this paper we use LUTESS V2 [9], a testing environment which automatically transforms formal specifications into test data generators. Its main application domain is the validation of the control part of a software. Gatel [6] is also based on constraint logic programming but, contrary to LUTESS , it is rather a "white box" testing tool. It translates a Lustre program and its environment specification in an equivalent Prolog representation and then computes a test input according to test objectives. Furthermore, LUTESS generates test data with dynamic interaction with the system under test while Gatel interprets the Lustre code. Lurette [8] is similar to LUTESS and it makes 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.

Testing a reactive system using LUTESS requires modeling the external environment, that is, expressing the conditions that the system inputs should invarianltly satisfy i.e hypotheses under which the software is designed. Different testing techniques (conditional probabilities, safety-property guided testing) can be used to guide the test data generation process, to avoid unrealistic or unreasonable test cases, or to reach suspicious situations.

In [9], we have presented the new version of LUTESS based on constraint logic programming, illustrated on a sim-

*Work supported by the TAROT Marie Curie Network (MRTN-CT-2004-505121) and SIESTA (www.siesta-project.com), a project of the French National Research Foundation (ANR).

ple reactive program example, an air-conditioner controller. Towards our effort to examine how hard this task is for real-world applications, we present in this paper, an application of this testing methodology on a more realistic and well known case study, the steam boiler control system [1]. This system operates on a significantly large set of input/output variables as well as of internal functions and has been used to assess the applicability of several formal methods [1]. The objective of the case study is to assess the difficulties of the test modeling activities and of 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 [7] 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. On the other hand, we check 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 structured in three sections. Section 2 provides a brief overview of the essential concepts on testing synchronous software using LUTESS . Section 3 presents the steam boiler specifications while in section 4 we thoroughly demonstrate our approach to build the test model and the environment specification.

2. Synchronous Programs & LUTESS

LUTESS is a "black box" testing tool designed for synchronous reactive software. The basic characteristic of such software is that the environment reacts instantaneously to the system requests. A synchronous program has a cyclic behavior : at each tick of a global clock (say *t=1*), inputs (i_1) are read and processed simultaneously and outputs (o_1) are emitted (see Figure 1).

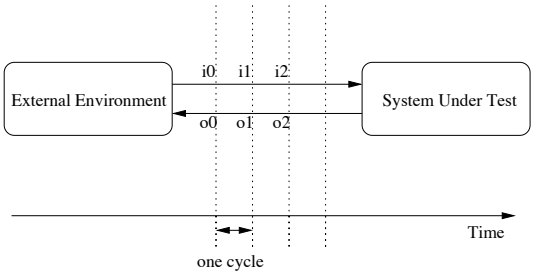

Figure 1. Synchronous software operation.

LUTESS specifications are based on Lustre, a data-flow language appropriate for programming real-time critical reactive systems [5]. 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. A Lustre program is

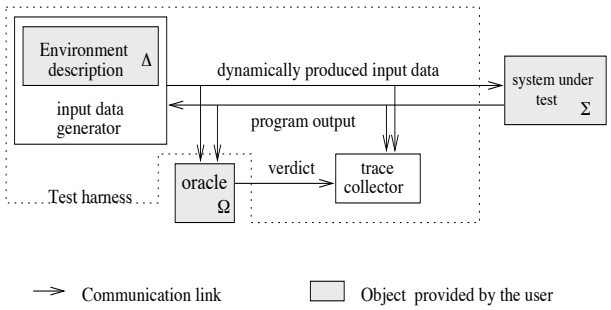

Figure 2. The LUTESS testing environment.

structured into nodes; a node is a set of equations which define the node's outputs as a function of its inputs. Each variable can be defined only once within a node and the order of equations is of no matter. Specifically, when an expression E is assigned to a variable X, X=E, that indicates that the respective sequences of values are identical throughout the program execution; at any cycle, X and E have the same value. Once a node is defined, it can be used inside other nodes like any other operator.

The operators supported by Lustre are the common arithmetic and logical operators (+, -, *, /, and, or, not, ...) as well as two specific temporal operators: the *precedence* (`pre`) and the *initialization* ($\rightarrow$). The `pre` operator introduces to the flow a delay of one time unit, while the $\rightarrow$ operator -also called *followed by* (`fby`)- allows the flow initialization. Let $X=(x_0, x_1, x_2, x_3, ...)$ and $E=(e_0, e_1, e_2, e_3, ...)$ be two Lustre expressions. Then `pre`$(X)=(nil, x_0, x_1, x_2, x_3, ...)$, where *nil* is an undefined value, while $X \rightarrow E=(x_0, e_1, e_2, e_3, ...)$.

To perform the test operation, LUTESS requires three components: the software environment description (Δ), the executable code of the system under test (Σ) and a test oracle (Ω) describing the system requirements, as shown in Figure 2. The system under test and the oracle are both synchronous executable programs.

LUTESS builds a test input 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 input and output sequences as well as the associated oracle verdicts thanks to the trace collector.

The test is operated on a single action-reaction cycle: the generator produces an input vector sends it to the system under test; the later reacts with an output vector sent back to the generator. The generator produces a new input vector and so on. The oracle observes the exchanged inputs and outputs to detect failures.

In order to automatically generate test sequences, LUTESS needs an environment description (Δ component in Figure 2) defining the valid input sequences of the program under test. This description is made in an extended version

```
testnode Env(<SUT outputs>) returns (<SUT inputs>);
var <local variables>;
let
   environment(Ec1); ... environment(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.

of Lustre as a new file, the main node of which is called *testnode* and the general form of its syntax can be seen in Figure 3. The inputs (outputs) of a *testnode* are the outputs (inputs) of the program under test. The *testnode* is automatically transformed into a test data generator.

As a rule, a *testnode* contains four operators specifically introduced for testing purposes. The `environment` operator is used to specify a list of invariant properties, stated as Lustre expressions, that should hold at each execution step of sequence. The definition of the input domains for variables can be stated as an instant invariant as well as a temporal property. At any cycle, the test generator chooses a sequence of valid random input values to supply the software with. Therefore, the environment constraints can only depend on the previous values of the output signals[1]. The `prob` operator is used to define conditional probabilities, while the `safeprop` operator is used to guide the test generation process towards situations where the safety properties could be violated. 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 [10].

3. The Case Study

The steam boiler specification has been used more than ten years ago as a common case study for several formal specification methods [1]. We are considering here the Lustre implementation of this system, proposed in [4].

According to the informal specifications provided in [1], the physical system of the boiler is composed of four pumps which supply the boiler with water while it turns it into steam at its output. In order to avoid any malfunction or erroneous situation, the controller must maintain the level of water in the steam boiler, within some safe limits. More precisely, as it is shown in Figure 4, the physical units of the system are:

[1] Supposing that at the current instant *t*, the input signal *i(t)* must be issued before the software computes the output *o(t)*, then, if the environment definitions were referring to the current output *o(t)*, the generation of valid input sequences at the instant *t* would be impossible, since the actual value *o(t)* would be yet unknown.

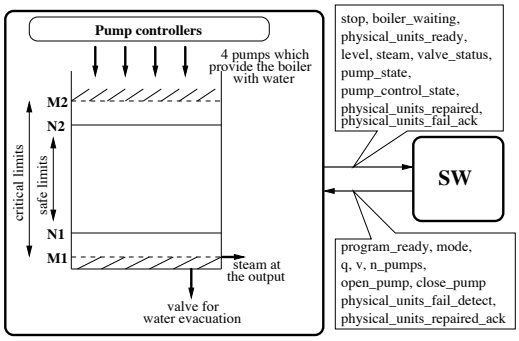

Figure 4. The steam boiler control system.

- **The boiler itself** comprises a valve, which at the initialization phase, evacuates the remaining water. It has a total capacity of C litres and produces a maximum steam quantity of W litres/sec. The minimum and maximum water limits are M_1 and M_2 respectively; outside these limits, the system will be endangered after five seconds, due to either lack of water supply or water overflow.
- **Four pumps** provide the boiler with water. Each pump is characterized by its capacity (p litres/sec) and its state, "on" or "off". Although a pump can be stopped instantly, when it is being started, it needs a whole cycle before it is being opened.
- **Four controllers** (one for each pump) inform if there is flow of water from the pumps to the boiler or not.
- **A water unit** measures the quantity of the water (q) in the boiler, measured in litres.
- **A steam unit** measures the quantity of the steam (v) at the output of the boiler, measured in litres/sec.

The physical system communicates with the program that controls its function, i.e. the controller, via messages. At each execution cycle, the controller receives and analyzes the messages from the physical system, then it sends back new commands.

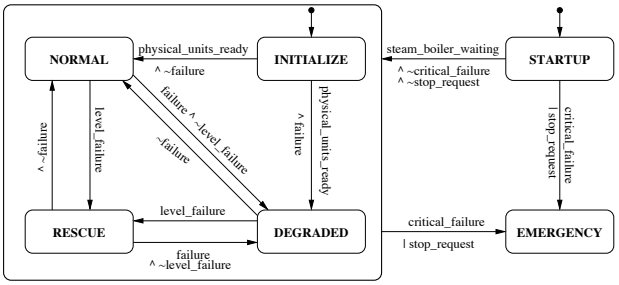

Figure 5. Operational modes of the system.

According to the messages sent by the environment and the detected failures, the controller can operate in different modes. Figure 5 illustrates these modes.

1. startup mode: This is the very beginning of the program, where there is no critical failure neither a stop request detected and the program is waiting for the appropriate message from the physical system that it is ready to begin functioning. If a critical failure is detected or there is a stop request, the program goes to the *emergency mode*.

2. initialize mode: As soon as the physical system is ready to start functioning, the program enters the initialization mode. In this mode, the program sends continuously to the environment a message denoting that it is ready to function; that happens until it receives back from the environment the corresponding positive response. Then, if there is no failure detected, the program enters the *normal mode*; otherwise, i.e. if there is a failure detected in a physical unit, the program enters the *degraded mode*.

3. normal mode: This is the mode where the program tries to maintain the quantity of water within the normal limits, N_1 and N_2, with all the physical units operating correctly, of course. Once a failure in the water unit is detected, the program enters the *rescue mode*, whereas in case of any other kind of failure, the program enters the *degraded mode*.

4. degraded mode: In this mode, the program tries to maintain the quantity of water in a satisfactory level, despite of a possible failure in a physical unit other than the water unit. When this failure is repaired, the program returns to the *normal mode*. When a failure in the water unit is detected, the program goes to the *rescue mode*.

5. rescue mode: In this mode, the program tries to maintain the quantity of water in a satisfactory level, despite of the failure in the water unit. In this case, the quantity of the water in the boiler is estimated, taking into account the quantity of steam at the output and the intake of water that the pumps supply. When the failure is repaired, the program goes back to *normal mode*, or to the *degraded mode* when a failure in another physical unit is detected.

6. emergency mode: This is the mode where the program must enter any time there is a critical failure or a stop request. Once the program reaches the *emergency mode*, it stops its execution and the physical environment is responsible to take appropriate functions.

4. Modeling and testing the boiler

The primary function of the boiler controller is to keep the water level between the given limits, based on inputs received from different boiler devices. Thus, in order to test the controller in its normal functioning, we first consider that all devices behave correctly and provide the correct inputs to the controller. In a later stage, we consider different faults that are tolerated by the controller in order to test its reaction in these cases.

This said, the generation of test sequences with LUTESS requires a model of the system environment (test model). Although the test models are specific to the program under test, we claim that the modeling and testing process can follow an incremental approach:

1. **Domain definition**: Definition of the domain for integer inputs. For example, the water level cannot be negative or overflow the boiler capacity.

2. **Environment dynamics**: Specification of different temporal relations between the current inputs and past inputs/outputs. These relations often include, but are not limited to, the physical constrains of the environment. For example, we could specify that when the valve opens, the water level can only decrease.

   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. **Scenarios**: Having in mind a specific test objective, the tester can specify more precise scenarios, by specifying additional invariant properties or conditional probabilities. The *prob* operator of LUTESS 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. As a simple example, consider the *stop* input which stops the controller when true; a completely random value will stop prematurely the controller and thus prevent the testing of all the behaviors. In this case, lowering the probability of *stop* being true keeps the controller running.

4. **Property-based testing**: This step uses formally specified safety properties in order to guide the generation toward the violation of such a property [10]. The resulting generation will remove inputs that, given the property expression, obviously cannot lead to its violation. In order to be effective, this guidance requires the safety properties to be expressed as a relation between inputs that imply some outputs. Test hypotheses

can also be introduced and possibly make this guidance more effective.

In the following, these steps are described in details and applied to the steam boiler case study.

4.1. Domain definitions

The domain of an input is the set of meaningful values that an input is designed to receive. Integer inputs used in controllers are often used to represent a state, consisting of a limited subset of integers, like in the case of *valve_status* or *pump_state* variables. In other cases, such as *level*, they represent an interval of integers.

We use the following expressions to define the domain of each integer input in the controller[2] (inside parentheses are given the values of constants used):

- The water level value should be between 0 and the maximum capacity of C (1000):

```
0 <= level and level <= C
```

- The water steam value should be between 0 and the maximum quantity of W (25):

```
0 <= steam and steam <= W
```

- The valve can be closed (0) or open (1):

```
closed <= valve_status and valve_status
<= open
```

- The pump state can be closed or open:

```
AND(N_pump, closed <= pump_state and
pump_state <= open)
```

where `N_pump` is the number of pumps (4); `AND` is a boolean operator applying the above boolean expression to the whole array *pump_state* and resulting in the conjunction of the corresponding values.

So far, the above model can be directly used to generate random test sequences. Nevertheless, these sequences are not very conclusive. The controller cannot deal with the random behavior of all the devices, and as soon as it is started it goes to emergency mode: either a stop request has been sent (stop being true for 3 consecutive steps), either a failure has occurred while initializing. Note that, according to the specification, any message received when not expected is considered as a failure. Thus, in order to observe meaningful executions, we should specify, more thoroughly, the environment dynamics.

[2]Note that these invariants are meaningful only under the assumption that all the devices are functioning.

4.2. Environment dynamics

Environment dynamics can be expressed as temporal relations between the current and past values of the software inputs and outputs. We can derive directly from the specification, the properties that identify the correct behavior of the devices, some of which are presented below:

- The *steam_boiler_waiting* message is sent only in *startup* mode:

```
steam_boiler_waiting = (false -> (pre mode =
startup));
```

- The *physical_units_ready* message is sent as a notification of the received message *program_ready*:

```
physical_units_ready= (false -> (pre
program_ready));
```

- The water level is equal to its previous value, to which is added the quantity of water entering through the open pumps and removed the quantity exiting through the steam or the valve. Thus, the expected water level is:

```
expected_level = level -> pre(expected_level)
+ Dt*sum_flow(N_pump, pump_control_state) -
Dt*steam - Dt*valve_status*V;
```

where the `sum_flow` node calculates the sum of the flow passing through all the pumps, based on the controller state.

If the expected level is negative, this means that no more water is remaining in the steam boiler (*level=0*). If the expected level is greater then the total capacity, it means that not all the expected water has been flowing through the pumps and in this case the boiler is full (*level=C*). When the expected level is between 0 and C, it represents the actual quantity of water (*level=expected_level*):

```
  true -> if expected_level < 0 then level=0
else if expected_level > C then level=C else
level=expected_level;
```

- When the boiler starts, the steam flow is supposed to be zero:

```
implies(true -> pre(mode=startup), steam=0);
```

- The state of the valve (*valve_status*) changes only when the *valve* message is sent by the controller:

```
(valve_status=closed) -> (pre
valve=(valve_status <> pre valve_status));
```

- The pump is opened (resp. closed) when it receives open_pump (resp. close_pump). This behavior is implemented in the *correct_pump_state* node (that we don't detail here for sake of simplicity) and applied to all the pumps:

```
AND(N_pump, correct_pump_state(open_pump,
close_pump, pump_state));
```

Table 1. Excerpt of a test case using all the possible invariant properties.

	t_0	t_1	t_2	t_3	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{998}	t_{999}	t_{1000}
stop	0	0	0	0	0	0	0	0	0	0	0	0	0
steam_boiler_waiting	0	1	0	0	0	0	0	0	0	0	0	0	0
physical_units_ready	0	0	0	0	0	0	1	0	0	0	0	0	0
level	964	964	859	804	519	434	389	374	329	459	534	544	509
steam	0	0	11	1	13	17	9	3	24	4	22	13	22
valve_status	closed	closed	open	open	open	closed	closed	closed	closed	closed	closed	closed	closed
pump_state[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[1,1,0,0]	[1,1,1,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
pump_control_state[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[1,1,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
program_ready	0	1	0	0	0	0	0	0	0	0	0	0	0
mode	start	init	init	init	init	init	normal	normal	normal	normal	normal	normal	normal
valve	0	1	0	0	1	0	0	0	0	0	0	0	0
q	964	964	859	804	519	434	389	374	329	459	534	544	509
v	0	0	11	1	13	17	9	3	24	4	22	13	22
open_pump[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[0,1,0,0]	[0,0,1,0]	[1,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]
close_pump[0..3]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]	[1,0,0,0]	[1,0,0,0]	[1,0,0,0]	[0,0,0,0]	[0,0,0,0]	[0,0,0,0]

- The pump controller indicates that a pump is opened after a delay of one cycle (due to the time needed to balance the pressure as specified in [1]) and immediately when stopped:

```
AND(N_pump, pump_control_state = (pump_state
= open^N_pump and pre(pump_state =
open^N_pump)) );
```

- When a fault message is received from the controller, the device acknowledges the controller that it has received the message. This behavior is applied to all the pumps and their controllers, the level and the steam. We give here the example of the level unit:

```
level_failure_acknowledgement = (false -> pre
level_failure_detection);
```

- When a fault detection message is received, the devices send the repaired message until they receive acknowledgement from the controller. This behavior is applied to all the pumps and their controllers, the level and the steam. We show below how to apply this to all the pumps:

```
AND(N_pump, pump_repaired = (false^N_pump
-> pre(pump_repaired) and not(pre
pump_repaired_acknowledgement) or pre
pump_failure_acknowledgement))
```

Table 1 shows a generated test case resulting from the above test model. In order to avoid premature stop of the system, we added the `not(stop)` invariant. The generated test sequences simulate the normal function of the steam boiler. At the first execution cycle, the controller requests the opening of the valve, because of the high water level, until the latter is reduced to the safe limits (t_7). Afterward, the controller continues on *normal* mode, since no failure is signaled by the physical units.

4.3. Test scenarios

Previously generated test cases are obtained by adding systematically all the invariants that define correct functioning of the boiler devices. This can be seen as a normal execution of the software. But, often when testing, one can try to put the software into abnormal execution scenarios. In LUTESS , specific execution scenarios can be obtained either with invariant properties, either by specifying different conditional probabilities.

In section 4.2, using an invariant property, we have set *stop* to be always false, which can be seen as a simple scenario: "the boiler is never stopped". If we want to test whether the software behaves correctly in case of shutdown, we may want to allow "sometimes" the *stop* message to be true, by specifying a small probability: `prob(true, stop, 0.05);`

In a *testnode* (see Figure 3), 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`. Hence, with the above probability, the obtained sequences have rare occurrences of the *stop* message and the system can be observed for some time, before being stopped late in the testing process.

Failures of the system can also be simulated this way. To do so, invariant properties expressed in sections 4.1 and 4.2 can be replaced by `prob` expressions. For instance:

- A possible failure of the system can occur in the *initialize mode*, if we assume that there is a small probability to get the message *physical_units_ready*, when expected:

```
prob(true, physical_units_ready = (false ->
(pre program_ready)), 0.2);
```

Table 2. Excerpt of a test case with broken level device scenario.

	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}
physical_units_ready	0	1	0	0	0	0	0	0	0	0	0
level	579	519	509	464	429	334	-1069082252	389	514	574	614
level_repaired	0	0	0	0	0	0	0	0	1	0	0
level_failure_acknowledgement	0	0	0	0	0	0	0	1	0	0	0
mode	init	normal	normal	normal	normal	normal	rescue	rescue	normal	normal	normal
q	579	519	509	464	429	334	279	389	514	574	614
level_failure_detection	0	0	0	0	0	0	1	0	0	0	0
level_repaired_acknowledgement	0	0	0	0	0	0	0	0	1	0	0

- Another failure may consist of a change of the valve state, even if no command has been sent to the valve:

```
prob(true, (valve_status=closed) -> (pre
valve = (valve_status <> pre valve_status)),
0.8);
```

Of course, this list is not exhaustive. Every property in an `environment` operator could be replaced by such a `prob` expression. The value of the assigned probability must be empirically determined by the tester.

We show here, a more advanced simulation, of a nonsignaled failure of the level measurement unit. To do so, we first remove any domain constraints for the level input. Then, we consider the previously shown invariant specifying the current level value:

```
level_inv = true -> if expected_level < 0 then
level=0 else if expected_level > C then level=C
else level = expected_level;
```

We keep the same invariant conditions when not in *normal* mode:

```
implies( true -> pre(mode)<>normal, 0<=level
and level<=C -> level_inv );
```

And, while in *normal* mode, we introduce:

```
prob( false -> pre(mode)=normal, level_inv,
0.9 );
```

Table 2 shows a generated test case for this level device fault simulation and the corresponding controller reaction. At instant t_{13}, an arbitrary negative level value has been generated and the controller has detected a fault in the water level measuring device (`level_failure_detection=1`). We can notice that the mode has changed to *rescue* and the level value has been estimated. In the next two steps, the device has been repaired and the controller goes back to *normal* mode.

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 [9]: (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.

4.4. Property-based testing

Safety properties are specified in a testnode using the `safeprop` operator. When this operator is used, testing is performed to check if these properties are satisfied by the program under test. Property-based testing guides the test generation by avoiding, when possible, input values that cannot lead to a property violation. Consider, for instance, the property: "issuing the stop message for 3 consequent steps leads the controller into emergency mode". To formally express this property we use two local boolean variables `pre_stop` and `pre_pre_stop`, referring to the value of `stop` for the past 2 steps:

```
  pre_stop = false -> pre stop;
pre_pre_stop = false -> pre pre_stop;
safeprop( implies(false -> pre(mode)=normal
and stop and pre_stop and pre_pre_stop,
mode=emergency));
```

Violating this property requires setting the left part of the implication to true. The table 3 shows the effect of the above specification on the generated sequence. Immediately after the controller has passed into *normal* mode, the value of `stop` is set to true for the following 3 steps.

4.5. Concluding remarks

The objective of the case study was to assess the test modeling difficulty and the test generation complexity of LUTESS V2. The overall study showed that relevant test models for the steam boiler controller were not difficult to build: Modeling the steam boiler environment required a few days of work. Of course, the effort needed for a complete test operation is not easy to assess as it depends on the desired thoroughness of the test sequences which may lead the tester to write several conditional probabilities corresponding to different situations (and resulting to different testnodes). It must be noted that building a new testnode to generate a new set of test sequences usually requires a slight modification of a previous testnode. This makes easy to generate a big number of test sequences with a small effort. Thus, when compared to manual test data construction,

Table 3. 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}	t_{11}
stop	1	0	1	0	1	1	0	1	1	1	0	0
physical_units_ready	0	0	0	0	0	0	1	0	0	0	0	0
mode	start	init	init	init	init	init	normal	normal	normal	emergency	emergency	emergency

that the test professionals often use in practice, such an automatic generation of test cases could certainly facilitate the testing process.

The steam boiler problem requires exchanging an important number of messages between the system controller and the physical system. The main program handles 38 inputs and 34 outputs, boolean or integer, and it is composed of 30 internal functions. The main node is made, when unfolded, of 686 lines of Lustre code. Each testnode consists of about 20 invariant properties modeling the boiler environment to which are added various conditional probabilities or safety properties. The average size of a testnode, together with the auxiliary nodes, approximates 200 lines of Lustre code. It takes approximately less than 30 seconds to generate a sequence of hundred steps, for any of the test models we used (tests performed on a Linux Fedora 9, Intel Pentium 2GHz and 1GB of memory; LUTESS uses the ECLiPSe[3] environment for constraint solving.

5. Conclusions and Future Work

Realistically and efficiently testing a reactive application with LUTESS requires building test models (tesnodes) of the program. We have defined a methodology suggesting several modeling stages, starting from domain definitions and environment dynamics and following with scenarios and safety properties. To assess the difficulty to carry out this testing methodology, we have revisited the steam-boiler system specification, a well-known case study. We have identified several constraints and assumptions on the system inputs and outputs which contribute to well-formed test models of the program and we have thoroughly described the steps of the proposed test methodology including guidelines to the test model construction and recommendations on the use of the conditional probabilities and the safety-property guided test generation. The results are encouraging both in terms of modeling difficulty and of required generation time. Even if other case studies are needed to assess the applicability and the scalability of the approach, 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.

[3] http://www.eclipse-clp.org

References

[1] Jean-Raymond Abrial. Steam-boiler control specification problem. In *Formal Methods for Industrial Applications*, pages 500–509, 1995.

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

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

[4] T. Cattel and G. Duval. The steam boiler problem in lustre. *Formal Methods for Industrial Applications*, pages 149–164, 1996.

[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, 1991.

[6] Bruno Marre and Agnès Arnould. Test sequences generation from lustre descriptions: Gatel. In *IEEE International Conference on Automated Software Engineering*, pages 229–237, Grenoble, France, October 2000.

[7] J. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, 10(2):14–32, 1993.

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

[9] 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.

[10] 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.