Automatic Testing of Reactive Systems *

Pascal Raymond, Xavier Nicollin, Nicolas Halbwachs
Vérimag,† *Grenoble - France*
{*raymond,nicollin,halbwachs*}*@imag.fr*

Daniel Weber
Schneider Electric, Grenoble - France
daniel_weber@mail.schneider.fr

Abstract

This paper addresses the problem of automatizing the production of test sequences for reactive systems. We particularly focus on two points: (1) generating relevant inputs, with respect to some knowledge about the environment in which the system is intended to run; (2) checking the correctness of the test results, according to the expected behavior of the system. We propose to use synchronous observers *to express both the relevance and the correctness of the test sequences. In particular, the relevance observer is used to randomly choose inputs satisfying temporal assumptions about the environment. These assumptions may involve both Boolean and linear numerical constraints. A prototype tool, called* LURETTE*, has been developed and experimented, which works on observers written in the* LUSTRE *programming language.*

1 Introduction

The term *reactive system* was introduced by David Harel and Amir Pnueli [13], and is now commonly accepted to designate systems that permanently interact with their environment, and to distinguish them from *transformational systems* (e.g., compilers).

In this paper, we understand this term in a more restrictive way, distinguishing between "*interactive*" and "*reactive*" systems: *interactive systems* permanently communicate with their environment, but at their own speed (e.g., operating systems, data base interface, web server), while *reactive systems* have to react to an environment which cannot wait. Typical examples appear when the environment is a physical process (e.g., process control in industry, power plants; embedded systems in trains, aircrafts...). In contrast with most interactive systems, reactive systems are generally *deterministic*: from an abstract point of view, the execution of a reactive system can be viewed as an infinite sequence of input/output vectors, where, at each step, the output values are completely determined by the past and present inputs. This is the point of view of synchronous programming languages [5, 2, 15], and we will adopt it throughout the paper. Notice that, with this point of view, the "real-time" constraints on the system generally reduce to a maximal reaction time; they are clearly separated from its functional requirements.

Because of their application domain, reactive systems are intrinsically critical. They need to be strongly validated before being run in their actual environment. All the techniques and tools that can increase confidence in the design of reactive systems are of high interest: high level, special purpose, languages for their specification and design, compilers for automatic coding, validation techniques, both for the code and the specification.

The most appealing validation approach is formal verification, which intends to prove that the system satisfies some given properties. This approach is intrinsically limited by theoretical problems of decidability (in particular, because of numerical aspects of the system), and technical problems of cost (time and space necessary for the proof). There are, roughly speaking, two approaches to formal verification. Techniques based on general *theorem proving* make use of inference systems with deductive rules. Such techniques generally involve a great amount of user interaction. The second approach is based on *model checking*. The proof is made by an exhaustive exploration of a finite abstraction of the system. The proof is partial, but conservative, in the sense that if the model meets the properties, then so does the real system, otherwise the result is unconclusive. These techniques are (mostly) automatic, but they are dramatically limited by the "size explosion" of the models.

Program testing is a more popular validation technique, which is complementary to formal verification. Their goals are rather different, since testing aims at finding bugs, but cannot generally provide any absolute, positive result. The classical approaches to testing focus on the following aspects:

- *Glass-box vs Black-box:* Some techniques assume a

*This work has been partially supported by the ESPRIT-LTR project "SYRF".

†Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG (*www-verimag.imag.fr*).

full knowledge of the program under test (Glass-box, or structural testing), while others suppose that (almost) nothing is known about it (Black-box, or functional testing).
- *Unit - Integration - System:* The test process is generally hierarchical. First, the functionality of small components is widely tested (unit testing); then the interaction between those components is tested (integration testing); and finally, the whole system is tested "in place" (this last step is in general unfeasible for embedded systems, so the test is performed in a simulated environment).

Our approach does not really fall into this classification. Our main goal is to concentrate on the complementarity of the verification and test methods. Program testing is the only way of increasing the confidence about systems that are outside the scope of formal verification. So, in our opinion, a tool devoted to program testing should concentrate on cases where formal verification fails:
- programs with significant numerical aspects, which cannot be abstracted away;
- programs the source code of which is not necessarily available — e.g., because it is partly written in languages that are not supported by verification tools;
- programs which are used as specifications, and are implemented by hand (e.g., distributed implementations from centralized programs).

Amazingly, testing of reactive programs has not been widely studied. In particular, little attention seems to have been paid to the fact that such a system is embedded into a specific environment, with which it forms a *closed loop*: at a given instant, constraints on the current inputs may depend on the exact past behavior of the system. This distinguishes our approach from those based on temporal logic specifications (e.g., [17, 14]). The most advanced works on the subject [24, 19, 20, 9, 18] either assume a full knowledge of the program under test, or do not deal with numerical programs. Notice also that we are only interested in testing the functional behavior of the system. As mentionned before, the real-time aspects are a clearly separated concern (maximum reaction time) which is not considered here (see, e.g., [7, 23] for references on testing real-time constraints).

In this paper, we follow a *black-box* approach, since the program is not supposed to be fully known. Moreover, we will focus on the problem of automatic generation and analysis of test sequences, without distinguishing between unit, integration and system testing. The problems we are trying to solve are:
- Producing *realistic* inputs, with respect to the environment in which the system will be embedded.
- Moreover those sequences must be *interesting* for the user, who generally has some goal in mind.
- Analyzing outputs, to check whether the test has passed successfully.

To solve these problems, we will make use of some results coming from the field of formal verification. In particular, we propose that both the relevance of inputs and the success of testing be described by means of *synchronous observers*. Synchronous observers are programs implementing acceptors of sequences. They are used in synchronous program verification to specify program properties and assumption about the environment.

A *test sequence generator* (in short, a *tester*) is built from these observers and the interface of the program under test. A run of the tester generates one input test sequence together with the corresponding outputs and the flag indicating if the test has passed.

The paper is organized as follows. Section 2 focuses on the problem of producing relevant input sequences. The solution we propose yields an algorithm which is the core of a tester. We introduce in section 3 the specification of oracles defining the expected behavior of the system. They may be used to check whether a test succeeds or fails. We present in section 4 a tool called Lurette, developed at Verimag following these principles. Finally, section 5 provides some ideas for future work.

Throughout the paper, we will use a simple example to illustrate our approach. Let us consider a program which is intended to regulate a physical value (for instance a temperature). It takes as input the numerical variable u (to be regulated), and has two Boolean outputs p(lus) and m(inus), which control the physical process (for instance a heater) producing u (see Fig. 1).

The regulator is intended to maintain u inside a given interval (e.g., $[10, 25]$). Clearly, the program alone cannot satisfy this property, which requires a cooperation between the program and its environment. In order to generate realistic test sequences, we have to know assumptions about the behavior of the heater. The most obvious is that the heater obeys the program: if the program sends the p (resp. m) command, the temperature is intended to increase (resp. decrease). Moreover, we can have information on the way the temperature may vary (e.g., bounds on its derivative).

2 Modeling the environment

2.1 General framework

A reactive program is intended to run in a given environment, the properties of which are partially known and should obviously be taken into account during the testing process: test cases submitted to the program must satisfy the known properties of the actual environment. This is what we call the *realism* constraint. Moreover, even if a test

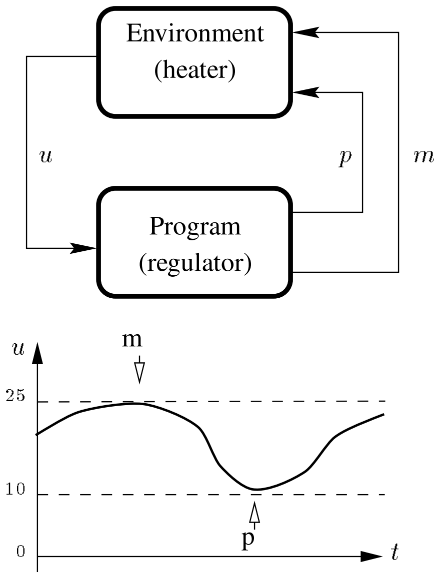


Figure 1: A regulator and its environment

case is realistic according to a given environment, it may be not *interesting*: the user usually has in mind a more or less precise goal. Distinguishing between *realism* and *interest* of test cases is surely relevant, from a methodologic point of view; however, technically speaking, there is no real difference: the test cases must satisfy constraints. In the following, we use the word *relevance* to talk about both *realism* and/or *interest*.

Examples of relevance constraints may be: "*two Boolean inputs are exclusive*", or "*a numerical input is always between 0 and 100*". More complex constraints deal with the temporal behavior of the variables: e.g., "*a Boolean input cannot be true at two consecutive instants*". Finally, the constraints on the environment may depend on the outputs of the considered program. As said before, in the heater example, the variation of the temperature (i.e., the environment) depends on the commands sent by the regulator program. Thus, from a general point of view, relevance constraints are any temporal properties involving the inputs/outputs of the considered reactive program.

2.1.1 Synchronous observers

The first problem addressed in this section is "how can the user describe such constraints?"

Our approach consists in using *synchronous observers*: observers have been proposed to describe properties in the framework of formal verification [22, 12]. A synchronous observer is a reactive program, taking as inputs the inputs/outputs of the program it observes, and computing a single Boolean output (Fig. 2) telling if the considered property is satisfied. The property is considered to be violated as soon as the observer answers "false". In terms of temporal logic, the class of properties that can be expressed by such observers are the *safety properties*.

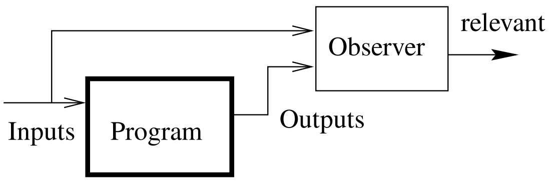


Figure 2: An observer of the relevance

In our opinion, this description approach has several advantages:

- There is no need to learn a new and complex formalism (such as temporal logics): in some sense, if you can program you can express properties.
- Modeling the environment is sometimes as difficult as designing the program itself. Since the model is a program, it can be analyzed and tested in order to increase the confidence in its adequacy.

As for any deterministic reactive program, the abstract behaviors of both the observer and the program can be described in terms of state/transition systems. A state S of a reactive program is a valuation of all its internal variables. A fundamental feature of a deterministic program is that, at each instant t, its state S_t is completely determined by the sequence of inputs received before t. Moreover, both the current output o_t and the next state S_{t+1} are completely determined by the current input i_t and the current state S_t[1].

In the case of the synchronous composition of a program and its observer, a global state of the system is a pair $\langle S_P, S_\Omega \rangle$ consisting of a state of the program P under test, and a state of the observer Ω.

The structure of S_P is unknown, since P is considered as a black box. We just suppose that we are able to *initialize* the program, and to *execute* it step by step, by calling its transition procedure. Given a current value for the input, such a procedure computes the corresponding output and modifies the internal state of the program. In fact, we will often suppose that we have two distinct procedures, one for computing the output, and one for modifying the state. This assumption is a basic feature in the considered scope (i.e., for programs obtained by compiling synchronous languages such as Lustre 4.1). It can always be satisfyed, either by storing the state as a memory dump, or by retrieving the state by running again the program on the same input sequence. Moreover, this assumption is not crucial: it just

[1] i_t (resp. o_t) is actually the vector of input (resp., output) values of the program. In the following, we write "the input" and "the output" for "the vector of inputs" and "the vector of outputs".

allows us to define more efficient algorithms that can "test" several inputs without changing the state of the program. The three primitives are:

- An *initialization procedure*, init_P, putting P in its initial state;
- An *output function* $o = \mathsf{out}_P(i)$ providing the output o computed by the program in its current state for a given input i; we suppose that the program may compute its outputs without changing its state.
- A *next-state procedure*, $\mathsf{next}_P(i)$, making P change to its next state according to a given input i.

On the contrary, we have a complete knowledge of the observer Ω. Its initial state is noted S^0_Ω, and its behavior is given by the two functions $\rho = \Omega_O(i, o, S_\Omega)$ computing its output (the value of the Boolean "relevant") and $S'_\Omega = \Omega_N(i, o, S_\Omega)$ computing its next state.

A *relevant test sequence* is a sequence of inputs $(i_1, i_2, \ldots, i_n)$ which results in a sequence of outputs $(\langle o_1, \rho_1\rangle, \langle o_2, \rho_2\rangle, \ldots, \langle o_n, \rho_n\rangle)$ of the pair $\langle P, \Omega\rangle$ such that: $\forall i,\ \rho_i = true$.

Intuitively, the observer behaves as a "language acceptor": the relevant sequences (i.e., the accepted language) are those for which the output of the observer is always $true$. Note that, in this way, we can only express *safety properties*.

Observers may be written in any formal language equivalent to the state/transition model. In our tool, we use the language Lustre. Before presenting a concrete example, we recall the useful aspects of this language.

2.1.2 Overview of the language Lustre

Lustre [10, 11] is a synchronous data-flow language. Each expression or variable denotes a *flow*, i.e., a function of discrete time (the positive natural numbers). The Lustre equation `x = 2*y + z` means: "*at each instant t*, $x_t = 2 \times y_t + z_t$". Lustre expressions are built from variables, constants (interpreted as constant flows), usual arithmetic and logical operators (considered as applying pointwise to flows), and two specific operators: the *previous* operator introduces a unit delay, and the *followed-by* operator is used to initialize flows. Let `E` and `F` be two Lustre expressions, then `pre E` and `E -> F` are Lustre expressions and we have at any instant $t > 1$:

- $(\texttt{pre E})_t = \texttt{E}_{t-1}$, while $(\texttt{pre E})_1$ is undefined.
- $(\texttt{E -> F})_t = \texttt{F}_t$, while $(\texttt{E -> F})_1 = \texttt{E}_1$.

For instance, the recursive Lustre equation "`n = 0 -> pre n + 1`" defines `n` as the flow of naturals "*0, 1, 2 ...*", and the equation "`k = true -> not pre k`" describes the oscillating Boolean flow "*true, false, true, false ...*".

A Lustre program is structured into *nodes*. A node defines its output parameters as functions of its input parameters. This definition is given by an unordered set of equations, possibly involving local variables. Once declared, a node may be instanciated in an expression, just like any predefined operator. For instance, the following node will be used to check whether a variable `X` belongs to an interval $[\texttt{LB}, \texttt{HB}]$:

```
node in_interval(X, LB, HB : real)
returns(S : bool);
let
   S = (X >= LB) and (X <= HB);
tel
```

2.1.3 Example: writing an observer in Lustre

Let us now consider more precisely the example of the heater. The inputs of our observer are the temperature `u` (the input of the program under test) and the commands `plus` and `minus` emitted by the program. The local variable `heat_on` defines at each instant the state of the heater, while the local variable `dudt` holds the slope of the temperature. We suppose that, in a realistic environment, the slope of the temperature may vary between 0 and 2 if the heater is on, and between -5 and 0 otherwise. Moreover, we wish to concentrate on "interesting" environments, where, e.g., the temperature is initially between 15 and 20, and where the heater is initially running. The Lustre code of such an observer may be:

```
node obs(u : real; plus, minus : bool)
returns (relevant : bool);
var heat_on : bool; dudt : real;
let
  heat_on = true -> if plus then true
                    else if minus then false
                    else pre heat_on;
  dudt = u - pre u;
  relevant = in_interval(u, 15, 20) ->
             if heat_on then
                  in_interval(dudt, 0, 2)
             else in_interval(dudt, -5, 0);
tel
```

For instance, the equation of `heat_on` states that it is initially true, and then it is set to true whenever `plus` is true, otherwise it is set to false whenever `minus` is true, otherwise it keeps its previous value. The slope `dudt` is defined to be always the difference between the current value and the previous value of `u`.

Note that such a Lustre program fits the state/transition model. Intuitively, the state variables of the program are the `pre` expressions; there is also an implicit state variable in the `->` operator: `E -> F` can be interpreted as `if init then E else F`, where $\texttt{init}_1 = \textit{true}$, and $\texttt{init}_t =$

false $\forall t > 1$. All other variables (including the output) are then defined as pointwise functions of inputs and state variables.

2.2 Exploiting the observer

The problem addressed in this section is "how to exploit the observer in order to generate relevant test sequences?" The simplest solution consists in randomly generating input values and using the observer to reject those which are not relevant, as done by the following algorithm:

```
σ := ε; S_Ω := S_Ω^0; init_P;   initializations
repeat
  repeat
    i := Random;                select randomly an input value
    o := out_P(i);              get the output of the program
    ρ := Ω_O(i, o, S_Ω);        compute relevance
  until ρ = true;               stop when relevant
  σ := σ · i;                   extend the sequence
  next_P(i);                    make the program progress
  S_Ω := Ω_N(i, o, S_Ω);        compute next state of the observer
until Length(σ) = MaxLength ;
```

MaxLength is a parameter of the algorithm allowing the length of the sequence to be limited.

This algorithm is quite ineffective (and rather stupid). For instance, if the observer has to check the equality of two real input values, the probability of randomly satisfying the constraint is null.

This is why we propose, rather, to transform the observer (which is an acceptor) into a *generator* of relevant input sequences. Intuitively, this consists in finding a solution of the equation $\Omega_O(i, \mathtt{out}_P(i), S_\Omega) = \mathit{true}$. In order to solve this equation, we have to introduce some restrictions:

- Since the program is considered as a black box, the function $\mathtt{out}_P$ is unknown. So we cannot solve an equation involving this function. The first restriction is that the value of ρ at instant t may not depend on that of $o = \mathtt{out}_P(i)$ at instant t. That is, the output function of the observer should be $\rho = \Omega_O(i, S_\Omega)$. However, the next state S'_Ω may still depend instantaneously on o: the transition function remains $S'_\Omega = \Omega_N(i, o, S_\Omega)$.
- With this first restriction, the problem is reduced to solving the equation: given a state S_Ω, find some i such that $\Omega_O(i, S_\Omega) = true$. Such a problem is in general undecidable (in particular because of numerical constraints). We restrict ourselves to a decidable subproblem: we will only consider equations involving logical operators and *linear constraints*.

In the sequel, we will note $i = \mathsf{Choose}(\Omega, S_\Omega)$ a function which selects a solution of the equation for a given observer Ω and a given state S_Ω.

The general algorithm for test sequence generation is the following; it is parameterized by the Choose function (more precisely studied in §2.4) and by the constant MaxLength.

```
σ := ε; S_Ω := S_Ω^0; init_P;   initializations
repeat
  i := Choose (Ω, S_Ω);         select an input value
  o := out_P(i);                get the output of the program
  σ := σ · i;                   extend the sequence
  next_P(i);                    make the program progress
  S_Ω := Ω_N(i, o, S_Ω);        compute next state of the observer
until Length(σ) = MaxLength ;
```

2.3 Example: generating a test sequence for the heater

Let us come back to the example of the heater. As it is written, the observer node (see §2.1.3) does not satisfy the first constraint: the value of `relevant` instantaneously depends on the program outputs `plus` and `minus` (via the local variable `heat_on`). We must slightly modify the observer by introducing delay operators in the definition of `heat_on`:

```
heat_on = true ->
          if (pre plus) then true
          else if (pre minus) then false
          else (pre heat_on);
```

The new observer is *not equivalent to the previous one*: the `heat_on` variable still represents the state of the heater, but with a delay. In practice, this restriction is not very important.

Here are some steps of a test sequence generation:

1st step. We start with the initial state of the observer where the implicit state variable `init` is true (i.e., we consider the left-hand argument of arrow operators). We have:

`heat_on = true,`

`relevant = in_interval(u, 15, 20).`

According to the definition of the node `in_interval`, `relevant` is then equivalent to the following constraint: $(u \geq 15) \wedge (u \leq 20)$. This constraint is linear and can be solved. More precisely, the Choose function builds an internal representation of the whole set of numerical solutions (in fact, a *polyhedron*) and randomly chooses a value in this set, for instance 18.2. Then, a step of the program is executed with this value; let us suppose it returns:

`plus = minus = false.`

2nd step. The internal state is now given by:

- `init = false` from now on,
- `pre u = 18.2` is the value chosen at the previous step,

- `pre plus = pre minus = false` are the values returned by the program at the previous step,
- `pre heat_on = true` is the value computed at the previous step.

In this state (i.e., with these "`pre`" values) we have:
`heat_on = pre heat_on = true`, and
`relevant = in_interval(dudt, 0, 2)`
with `dudt = u - 18.2`. The numerical constraints are:
$(u - 18.2 \geq 0) \wedge (u - 18.2 \leq 2)$, i.e.
$(u \geq 18.2) \wedge (u \leq 20.2)$.
Suppose that the algorithm selects $u = 19.9$. Then, we run a step of the program and we obtain (for instance):
`plus = minus = false`.

3rd step. Now the state of the observer is given by:
- `pre u = 19.9`
- `pre plus = pre minus = false`
- `pre heat_on = true`

So we have:
`heat_on = true`, and
`relevant = in_interval(dudt, 0, 2)`.
The constraints are then: $(u \geq 19.9) \wedge (u \leq 21.9)$. Assume that $u = 21.5$ is randomly chosen, and that the program returns:
`plus = false, minus = true`.

4th step. The state is now given by: `pre u = 21.5`, `pre plus = false`, `pre minus = true`, `pre heat_on = true`.
So we have: `heat_on = false`, and `relevant = in_interval(dudt, -5, 0)`, which implies: $(u \geq 16.5) \wedge (u \leq 21.5)$; so we can choose $u = 17.4$, and so on ...

2.4 The Choose function

This technical section presents our realization of the Choose function more precisely.

For a given state S_t, the value of the observer's output ρ is given by a Boolean function ρ_t depending only on the current input values: $\rho_t(i) = \Omega_O(i, S_t)$. The input variables are either Boolean or numerical. Let I_B be the set of Boolean inputs, and I_N the set of numerical inputs.

The first step consists in clearly separating the Boolean and numerical parts of the function. We express the output as a purely Boolean function of I_B augmented by a (minimal) set C of auxiliary Boolean variables, each of which representing a linear constraint: $\rho_t : I_B \times C \rightarrow \mathbb{B}$. Each constraint variable c is defined as a function $c : I_N \rightarrow \mathbb{B}$ of the form:

$$\sum_{x \in I_N} \alpha_x x + \beta \bowtie 0$$

where $\bowtie$ is $>$ or $\geq$.

Example: Let us consider the following Lustre definition:

```
relevant = (X = 0) ->
           if A then (B or (X > Y))
           else (X + Y <= 10) and
                (X * pre Y < 12);
```

where `A` and `B` are Boolean inputs and `X` and `Y` are numerical input. We suppose that we are in a non-initial state, where the value of `pre Y` is 3. The Choose function identifies 3 linear constraints:

$$C_1 = (\mathtt{X} > \mathtt{Y}) \quad C_2 = (\mathtt{X} + \mathtt{Y} <= 10) \quad C_3 = (\mathtt{X} < 4)$$

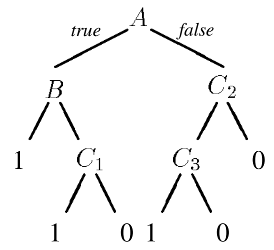
The observer's output in the current state is interpreted as a Boolean function over A, B, C_1, C_2 and C_3:

$$\rho_t = A \wedge (B \vee C_1) \vee \neg A \wedge C_2 \wedge C_3$$

The search for an input value such that ρ_t is true is performed in two steps: we concentrate first on the Boolean part, and then on the numerical constraints.

The Boolean choice is made as in [19]: we randomly choose a value for the variables $\langle I_B, C \rangle$ such that $\rho_t = true$. This choice is made using the Binary Decision Diagram (or bdd [1, 3]) of ρ_t. A bdd is a concise representation of the Shannon decomposition of a Boolean function.

Example: The bdd of our example is shown below (more precisely, we have represented the Shannon tree of the function; in a bdd common sub-graphs are shared).



Each node is labeled with a variable, its left-hand subtree corresponds to the "true" value of the variable, while the right-hand subtree corresponds to the false value. Leaves are either 1 (i.e., the always true function) or 0 (the always false function).
Each path in the bdd leading to the 1 leaf gives a monomial satisfying the formula. Here, there are 3 such paths:

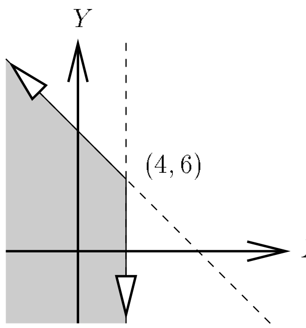
$$A \wedge B \quad A \wedge \neg B \wedge C_1 \quad \neg A \wedge C_2 \wedge C_3$$

Finding a value for $\langle I_B, C \rangle$ is made by randomly choosing a path in the bdd leading to the leaf "1". This search is quite simple, but raises a problem of fairness: if the choice is made with a $1/2$ probability on each node of the bdd, the probability for each path to be chosen is of the form $1/2^n$. Those kinds of probabilities are intrinsically unfair: it may happen that the same path be chosen "half of the time" or, on the contrary, that a path be (almost) "never" chosen. In order to avoid this problem, we use classical techniques of weighted graphs.

The numerical choice. Once a Boolean monomial, satisfying the formula, has been selected, it directly provides the value of all Boolean inputs (if some variables do not appear in the corresponding monomial, their values are chosen randomly). Let us now focus on the numerical part: the selected monomial, reduced to the variables in C, involves a conjunction of linear constraints. The set of points (i.e., values of numerical inputs) satisfying such a linear system is a convex polyhedron, that can be efficiently represented and manipulated in terms of its frame (vertices and extremal rays [6, 16]).

If the polyhedron is empty, this means that the whole Boolean monomial was a bad choice (numerical unfeasibility): the corresponding leaf in the bdd is set to "0", and another monomial is randomly chosen. Otherwise, the polyhedron representation allows us to select a point using several heuristics: limit values (vertices or edges), or random selection inside the polyhedron.

Example: Suppose that the selected monomial is $\neg A \wedge C_2 \wedge C_3$. The value of A is imposed by this choice: $A = false$. The value of B is randomly selected, for instance $B = true$. Then, the induced linear constraint is: $(X + Y <= 10) \wedge (X < 4)$. The corresponding polyhedron is shown below. The polyhedron is in gray; the corresponding frame consists of one vertex $(4, 6)$ and two rays, represented as vectors with white arrows.



3 Analyzing outputs

The second important part of the testing process consists in using the test sequences to validate the program or to detect its faults. This involves running the program with each input sequence and analyzing the corresponding output sequence it produces. The later part requires the presence of an *oracle* which compares the outputs with the expected ones. This comparison may be more complex than a simple point-to-point equality. For instance, one may expect that, for a given input sequence, the value of some numerical output at some instant be in some interval, or that some Boolean output become true after some condition and within a given delay.

The oracle is often a human being, but, in many cases, it may be automatized by a program, which may be as "simple" as consulting association tables between inputs and outputs, or more complicated, involving temporal properties on the input/output sequence. We propose to describe the oracle using the synchronous observer technique.

3.1 The oracle as a synchronous observer

An oracle may always be viewed as a synchronous observer in the sense of section 2.1.1. Indeed, we can consider that it observes the input and output sequences and produces at each instant a Boolean answer `correct` stating that the two sequences satisfy the properties up to the current instant (Fig. 3).

Example: A typical property that can checked on the heater, is that the observed temperature `u` always remains between two given constants (for instance 10 and 25). The Lustre code of such an observer may be:

```
node oracle(u:  real; plus, minus:  bool)
returns (ok :  bool);
let
  ok = in_interval(u, 10, 25);
tel
```

This framework is very similar to that of the environment's observer, that will thereafter be called the *assertion*, in order to avoid confusion. In both cases, the observers express properties of "good" interactions between the program and its environment: the assertion (resp., the oracle) expresses what is under the responsibility of the environment (resp., of the program) in a satisfactory interaction.

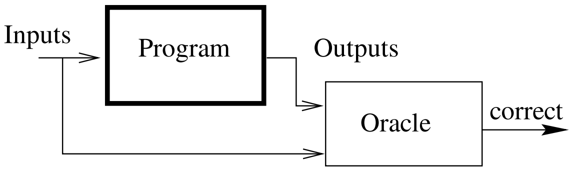


Figure 3: An oracle as a synchronous observer

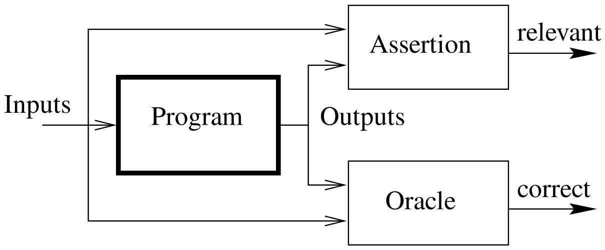


Figure 4: The complete framework of test generation

Another important remark is that there is no need to wait until the end of the test generation to start analyzing the correctness of the result: the assertion and the oracle may be put together in a unified framework (Fig. 4).

3.2 Using the oracle

The oracle and the assertion are very similar, but they are used quite differently in the testing process. We only need to execute the oracle, so there is no restriction on the way it is written. In particular, the `correct` output may depend instantaneously on both inputs and outputs of the program, and the function defining `correct` may be arbitrarily complicated.

In fact, the oracle is used in a way similar to that of the program under test: this observer can be viewed as a "black-box", and we only have access to it via primitives similar to those of the program interface:

- The procedure init_Γ puts the oracle in its initial state;
- The function $\mathsf{out}_\Gamma(i, o)$ provides the `correct` output of the oracle in its current state for the values (i, o) of the program's inputs/outputs;
- The procedure $\mathsf{next}_\Gamma(i, o)$ makes the oracle advance to its next state, according to the values of (i, o).

The use of the oracle is integrated in the generation algorithm; thus the correctness of the test can be checked during the test generation. The $out_\Gamma(i, o)$ function is computed at each step to check whether the current values meet the property; if they do not, the generation is stopped, and the current sequence is returned as a counter-example.

A more clever use the oracle observer is to try several input values at each step, exploiting the fact that both the program and the oracle may be called for their outputs, without modifying their internal states. In the following algorithm, we use a function $\mathsf{SelectPoints}(\Omega, S_\Omega)$, returning a finite set of "notable" points in the underlying polyhedron (e.g., the vertices). All these points are "tested" and, if no counter-example is found, a random point is chosen to complete the sequence.
We obtain the following algorithm:

σ := ε; S_Ω := S^0_Ω; init_P; init_Γ;	*Initializations*
repeat	
for each i	*Test several*
in SelectPoints (Ω, S_Ω) do	*... interesting points*
o := $\mathsf{out}_P(i)$;	
if $\neg$ $\mathsf{out}_\Gamma(i, o)$	*Stop with a*
then Stop($\sigma \cdot i$);	*... counter-example*
end;	
i := Choose (Ω, S_Ω);	*Select an input to*
	... complete the sequence
o := $\mathsf{out}_P(i)$;	*Get the output*
	... of the program
if $\neg$ $\mathsf{out}_\Gamma(i, o)$	*Stop with a*
then Stop($\sigma \cdot i$);	*... counter-example*
σ := $\sigma \cdot i$;	*Extend the sequence*
$\mathsf{next}_P(i)$;	*Make the program progress*
$\mathsf{next}_\Gamma(i, o)$;	*Make the oracle progress*
S_Ω := $\Omega_N(i, o, S_\Omega)$;	*Compute next state*
	... of the observer
until Length(σ) = MaxLength ;	

4 Putting it all together: the Lurette tool

A prototype tool, called Lurette, has been developed, following the ideas presented in this paper. The concrete realization of such a tool raises some technical problems which are sketched in this section.

4.1 Building a test generator

The program under test is a "C" file, supposed to implement a reactive program. In general, such programs are the "object code" obtained by the compilation of higher level languages such as Lustre. In order to use such a file, the Lurette tool must be able to extract a minimal set of informations:

- The names and types of its inputs/outputs, in order to "connect" them to the observers;
- The way of calling (or simulating the calls of) the procedures init_P, $\mathsf{out}_P(i)$ and $\mathsf{next}_P(i)$.

For the time being, Lurette is able to extract these informations from the C code generated by two different compilers:

- The academic *Lustre Compiler*, which is a free tool developed at Verimag;
- The *SCADE Code Generator*, which is an industrial tool distributed by the Verilog company. The SCADE environment is based on a graphical implementation of the Lustre language.

Note that this (small) list could be extended to any other language dedicated to reactive programs, given the convention used by its compilers. For instance, accepting code coming from other synchronous programming languages (like Esterel [2] or Signal [15]) would be easy since the academic compilers of those languages use (almost) common objet codes [21, 4].

Once the needed informations are extracted, the C program is compiled and linked with the *Lurette library*, which includes the Choose function, and bdd and polyhedron packages. The result is an executable *test generator* (or "tester"), dedicated to the given reactive program.

4.2 Calling a test generator

The tester implements the algorithm presented in section 3.2. It takes as input the description of the two observers (assertion and oracle), and the parameter giving the maximum length of the test sequence.

The assertion observer must be fully analyzed by the tester, so it must be written in a high level language with a well-defined semantics. The tester first makes some static analysis on this observer; in particular, it checks that its output does not depend instantaneously on the outputs of the tested program, and that all the involved numerical constraints are linear.

The oracle observer is dynamically loaded, instead of being linked with the tester. This way, the user can manage the test production by means of pairs assertion/oracle. The drawback of this choice is that the oracle must be interpreted instead of being compiled. So, even if not necessary in principle, this observer must be given (just like the assertion) in a high level language. However, the restrictions on the oracle are weaker: the output may instantaneously depend on both inputs and outputs of the program, and the code may be as complicated as needed. For the time being, both observers must be written either in Lustre or in SCADE.

5 Conclusion: experiment and future work

We have proposed a method to automatically generate and analyze test sequences for reactive deterministic systems. It is based on the use of synchronous observers to describe both the constraints on the environment (assertion) and the expected properties of the program under test (oracle). The main problem is to randomly generate inputs satisfying the assertion, which can be a complex temporal property, involving both Boolean and numerical constraints. This problem is solved by restricting ourselves to linear constraints, and by using bdds- and polyhedra-based techniques, which make the generation particularly efficient. At this time, we have got an experimental workbench (the Lurette prototype) connected to both an academic language (Lustre) and an industrial design tool (SCADE).

An experiment has been made on a case study provided by the Schneider Electric company. The program, written using the SCADE environment (very close to Lustre) is interesting for several reasons:

- The scope of this application is very critical (control/command in a nuclear power plant).
- Its size is significant: it has 29 inputs, 50 outputs, and corresponds to a data-flow network of 1000 operators. Once compiled, it results in a C program of 2600 lines.
- Numerical aspects are important in the program. Moreover some numerical procedures are directly written in C, therefore it is (almost) impossible to perform formal verification.

We wrote three assertion observers in Lustre, according to the knowledge of the Schneider Electric engineers. Each observer is a data-flow network with about 130 operators and leads to an internal model of 40 Boolean constraints (i.e. bdds) on 40 Boolean variables, and 15 linear constraints on 9 numerical variables. We have observed that the test generator runs very fast: sequences of hundreds inputs were generated in a few seconds.

An oracle was written in Lustre: it observes about 20 safety properties, extracted from the software requirement specifications. Those properties were not violated by the generated test. Note that we did not expect to find bugs in this software, that is actually running in a nuclear power plant.

A lot of work remains to be done. The most urgent is to evaluate and increase the quality of test selection (function Choose). There are several, commonly accepted, criteria for evaluating the quality of a set of test sequences:

- The *test coverage* is a measure of the part of the code that is actually executed by the test sequences. Several kinds of measures can be used, generally based on the control structure of the code (exercised statements, condition–to–condition paths, etc..).
- The *mutation analysis* [8] consists in introducing slight changes in a program, and in measuring the ratio of such changes that are detected by a given set of test cases.

Another important goal is to take *test objectives* into account during the generation of the test. For the moment, we only deal with *invariant* constraints (i.e., safety properties): the input sequence is build step-by-step in such a way that the assertion remains true. An additional, much more difficult, criterion would be to drive the generation in such a way that a property eventually becomes true (liveness property). Such a goal-directed generation requires some knowledge about the program under test. We can at least exploit the test objective to avoid generating sequences that have no chance to reach the goal, whatever the behavior of the program. Another approach is to use the oracle as an approximation of the program.

References

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6), 1978.

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

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[4] C2A-SYNCHRON. The common format of synchronous languages – The declarative code DC. Technical report, Eureka-SYNCHRON Project, October 1995.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, München, January 1987.

[6] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[7] D. Clarke and I. Lee. Testing real-time constraints in a process algebraic setting. In *Proceedings of the 17th International Conference on Software Engineering*, 1995.

[8] R. A. DeMillo. Mutation analysis as a tool for software quality assurance. In *COMPSAC'80*, Chicago, October 1980.

[9] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In R. Alur and T. Henzinger, editors, *8th International Conference on Computer Aided Verification, CAV'96*, Rutgers (N.J.), 1996.

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

[11] 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, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[12] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[13] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems,* NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems*. Springer Verlag, 1985.

[14] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 525–537, Berlin - Heidelberg - New York, May1997. Springer.

[15] P. LeGuernic, A. Benveniste, P. Bournai, and T. Gautier. SIGNAL, a data flow oriented language for signal processing. *IEEE-ASSP*, 34(2):362–374, 1986.

[16] H. LeVerge. A note on Chernikova's algorithm. Research Report 635, IRISA, February 1992.

[17] S. Morasca, A. Morzenti, and P. S. Pietro. Generating functional test cases in-the-large for time-critical systems from logic-based specifications. In *ISSTA 1996, ACM-SIGSOFT International Symposium on Software Testing and Analysis*, San Diego, CA, January 1996.

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

[19] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, Monterey, USA, November1994.

[20] F. Ouabdesselam and P. Parissis. Testing safety properties of synchronous reactive software. In *7th International Software Quality Week*, San Francisco, May 1994.

[21] J. A. Plaice and J.-B. Saint. The LUSTRE-ESTEREL portable format. Unpublished report, INRIA, Sophia Antipolis, 1987.

[22] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[23] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. Technical Report CTIT 97-17, University of Twente, 1997. Also as CSI Report R9712, University of Nijmegen.

[24] P. Thevenod-Fosse, C. Mazuet, and Y. Crouzet. On statistical testing of synchronous data flow programs. In *1st European Dependable Computing Conference (EDCC-1)*, pages 250–67, Berlin, Germany, 1994.