Project 22703: SYRF

Deliverable 3.5.1

Automatic Testing

---

1 Approach

Formal verification of synchronous programs will always be faced with undecidable problems, and with limitations caused by the intrinsic complexity of the programs to be verified. On the other hand, formal verification cannot be directly applied to programs, some part of which are written in other languages, or which are implemented by hand, particularly on distributed architectures. As a consequence, formal verification is only complementary to more classical validation techniques, based on simulation and testing. In view of this complementarity, a tool devoted to program testing should concentrate on cases where formal verification fails, i.e.,

- programs with significant numerical aspects
- programs the source code of which is not necessarily available — either because it is partly written in languages which are not supported by verification tools, or because they are implemented by hand.

The test of reactive programs has not been widely studied[1]. Such a 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 which are submitted to the program must satisfy the properties of the actual environment.

Our approach consists in using *synchronous observers*, already used in verification, to describe:

1. The expected behavior of the program, in order to allow testing diagnosis to be automatically generated;

2. The known properties of the environment, in order to avoid the generation of unrealistic test cases.

3. The test objectives, by which the user can concentrate the test on some classes of scenarios.

The first two roles of observers are standard in verification. The third one is new; a testing objective generally needs two kinds of observers:

[1] Notable exceptions are [TMC94, OP94, FJJV96], which generally either assume a full knowledge of the program under test, or do not deal with numerical programs.

- "safety" observers, defining variables that must *remain* true during the test;
- "liveness" observers, defining variables that must *become* true during the test.

On the other hand, we intend to adapt automata-based techniques, already used in verification, to the synthesis of test sequences satisfying assumptions about the environment and testing objectives. The result would be an observer deciding, at each step of the program execution, which input values are acceptable. Finally, instead of randomly choosing these input values before checking them against the observer (used as an *acceptor*), it appears more realistic to transform the observer into a *generator*, in order to consider only suitable values. Symbolic techniques, based on BDDs and linear constraints, can be applied for that.

2 Results

A testing tool, called LURETTE, is under development at INRIA/Spectre. For the time being, it is devoted to observers written in LUSTRE, and contains the following components:

A test case generator, which, from a set of assumptions and objectives described by an observer written in LUSTRE, automatically generates test sequences. This tool takes into account both control and numerical — restricted to linear algebra — aspects.

A test bed, which uses the industrial code generator developped by VERILOG to run the program under test together with the test case generator.

An "oracle": The pair *"generator/program"* can be connected with an observer of the expected behaviour, which decides on-line whether the observed sequence of inputs/outputs fulfills the requirements.

The test cases generator is based on an algorithm, combining *binary decision diagrams* (BDDs), for the logical aspects, and *convex polyhedra*, for the numerical part. The tool allows quite complex temporal assumptions to be handled. A typical example is

> *The input x increases during some time interval, with a derivative in some interval, then stabilizes until some event, before increasing again etc.*

As mentionned before, observers are used to express

- assumptions about the environment; the corresponding output, which should always be true, is called "realistic".
- expected behaviour of the program; when the corresponding output, called "erroneous", is false, an error has been detected.
- test objectives; they have two outputs, "interesting" and "goal_reached", which are used to drive and stop the generation of a test sequence: "interesting" should remain true, while "goal_reached" should become true.

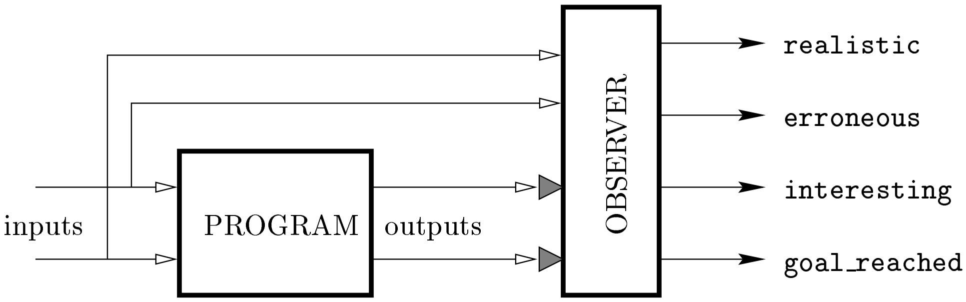

Figure 1: Using observers for testing

All these observers express properties about input/output sequences to the program, which is considered as a "black box", whose behavior, while being deterministic, cannot be foreseen by the tool. For this reason, these properties can depend, at a given instant, on the current or past values of inputs, but only on the past values of outputs. In other words, at each instant, an observer may constrain the input values, but not the outputs, which are computed by the program. This is why, in the general diagram shown by Fig. 1, we introduce a (one tick) delay between program outputs and the observer.

2.1 Global behavior of the system

Let P be the program under test, Ω be the observer, i and o be respectively the input and output vectors of the program, and α the vector of output signals of the observer.

A state of a program is a valuation of all of its internal variables. A basic 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.

In our case, a global state of the system is a pair $\langle S_P, S_\Omega \rangle$, made of a state of the program under test and a state of the observer. The precise structure of S_P is not known, since P is considered as a black box. P is only known as a procedure which computes an output o and a next state S'_P as functions of an input i and a current state S_P. We will write "$(o, S'_P) = P(i, S_P)$".

2.2 Relevant test sequences

A relevant test sequence is a sequence of inputs $(i_1, i_2, \ldots, i_n)$ which results in a sequence of outputs $(\langle o_1, \alpha_1, , \rangle \langle o_2, \alpha_2, , \rangle \ldots, \langle o_n, \alpha_n, \rangle)$ of the pair $\langle P, \Omega \rangle$ such that

- $\forall i = 1 \ldots n,\ \alpha_i \Rightarrow$ realistic $\wedge$ interesting, i.e, assumptions and safety objectives are always fulfilled;
- $\alpha_n \Rightarrow$ erroneous $\vee$ goal_reached, i.e., the sequence leads either to the violation of the property, or to the fulfilment of the liveness objective.

2.3 A general algorithm

The algorithm is given as a recursive procedure, which, from a global state $\langle S_P, S_\Omega \rangle$, obtained after applying a sequence of inputs σ, produces a set of relevant test sequences T:

```
Generate (⟨S_P, S_Ω⟩, σ)
    repeat
        i := choose(S_Ω);                     select an input vector
        (o,S'_P) := P(S_P,i);                 run one step of the program
        if erroneous(S_Ω, i,o) or
            goal_reached(S_Ω, i,o) then
            T := T ∪ {σ.i};                   new relevant sequence found
            return;
        else
            S'_Ω := Ω(S_Ω, i, o);             run one step of the observer
            Generate (⟨S'_P, S'_Ω⟩, σ.i);     complete the current sequence
        if enough(T) then return;             stop branching
    end repeat
end Generate
```

This algorithm generates a *branching* set of test sequences: Each call completes a common prefix σ with several suffixes. The algorithm is only a general skeleton, several parameters of which must be provided:

- "goal_reached" limitates the length of sequences, without regards to their fulfilment of the oracle (defined by "erroneous"). This limit can be either an explicit length, or a more complex liveness objective.
- "enough" limitates the branching of the search, thus the number of generated sequences.
- "choose" uses the observer's state to select a "realistic" and "interesting" input vector.

This general skeleton is intended to be used for studying and adjusting the various parameters.

2.4 Choosing test cases

While the program under test can be considered as a black box (one only needs to run it step by step), the observer is used to build, at each step, the *set* of relevant (i.e., realistic and interesting) inputs. This is performed using symbolic techniques, requiring that the observer structure (stae and functionality) be completely known. In a given state S_Ω, the observer outputs are logical formulas depending on

- Boolean inputs;
- Boolean conditions on numerical inputs.

An analysis of these formulas can show that:

- Only some valuations of Boolean inputs and conditions are "realistic" and "interesting" — let us note R this set —, and that some of these valuations can fulfil the "goal" or violate the oracle – we note B ($\subseteq R$) this set.

- Each valuation in R or B induces constraints on numerical inputs.

From such an analysis, a selection procedure ("`choose`") must

- select a valuation b of Boolean inputs and conditions, preferably in B if it is not empty, or simply in R. This selection can be random, or based on more clever heuristics.

- select a valuation n of numerical inputs, satisfying the constraints induced by b. We restrict ourselves to the case where these constraints are *linear constraints* (linear equations and inequalities). In this case, the set of solutions is a set of polyhedra, to which natural selection heuristics can be applied: limit values (vertices or edges), or random selection inside a polyhedron.

2.5 A simple example

To illustrate the approach, let us consider a program which is intended to regulate a physical variable. 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 producing u (see Fig. 2). The regulator is intended to maintain u inside a given interval (e.g., $[-10, +20]$).

Clearly, the program alone cannot satisfy this property, which requires a cooperation between the program and its environment. Before generating test sequences, we have to know assumptions about the behavior of the environment:

- After receiving a p signal, the value of u increases with a slope in $[0, 2]$.
- After receiving an m signal, u decreases with a slope in $[-5, 0]$.
- Initially, the value of u belongs to $[0, 10]$, and increases with a slope in $[0, 2]$.

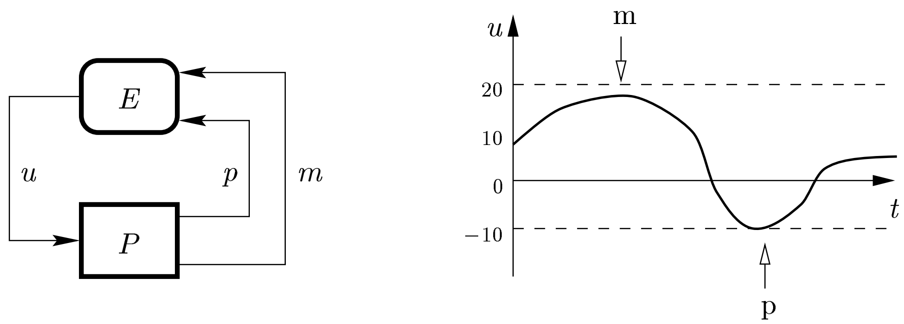

Figure 2: A regulator and its environment

The LUSTRE observer: These assumptions may be easily expressed by a synchronous observer, for instance in LUSTRE. Inputs to the observer are "u", "plus" and "minus", its output are "realistic" and "erroneous". An auxiliary Boolean variable "increasing" is used to remember whether "u" should increases of decreases. This variable is initially true, and then depends on the control signals "plus" and "minus" emitted by the program. Recall that the observer can only use past values of the program outputs, this is why we use the LUSTRE delay operator "pre". The definition is

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

Initially, the value of "u" should be in $[0, 10]$, which results in the initial LUSTRE constraint:

```
(u>=0) and (u<=10)
```

Afterwards, the behavior of "u" depends on the value of "increasing" et on its slope, which can be defined by the LUSTRE variable:

```
slope = u - pre(u);
```

By means of this variable, the "realism" of the behavior of "u" is defined by:

```
if increasing  then (slope>=0) and (slope<=2)
               else (slope>=-5) and (slope<=0);
```

Finally, the program behavior is erroneous if the value of "u" is out of bounds:

```
erroneous = (u<-10) or (u>20);
```

The whole observer is as follows:

```
node OBS (u: real; plus, minus: bool) returns (realistic, erroneous: bool);
var increasing: bool; slope: real;
let
    increasing = true -> if pre(plus) then true
                         else if pre(minus) then false
                         else pre(increasing);
    slope = u - pre(u);
    realistic = (u>=0) and (u<=10) ->
                    if increasing  then (slope>=0) and (slope<=2)
                                   else (slope>=-5) and (slope<=0);
    erroneous = (u<-10) or (u>20);
tel
```

Some steps of the generation: A test sequence generation starts in the initial state of the observer. The definition of "realistic" reduces to

```
realistic = (u>=0) and (u<=10)
```

while the one of "erroneous" is

erroneous = (u<-10) or (u>20);

No value of "u" can make both these variables true (i.e., B is empty), so one has to select a realistic value (e.g., $u = 9$) which is submitted to the program. Assume the returned signals are

plus = minus = false;

The generation goes on, in a non initial state. Previous values of "plus" and "minus" being both false, the variable "increasing" does not change (i.e., remains true). So we get

realistic = (slope>=0) and (slope<=2)

that is (since pre(u) = 9),

realistic = (u>=9) and (u<=11);

Since no realistic value of "u" can violate the oracle, one select, for instance, the value $u = 11$, and so on...

2.6 The tool LURETTE

A prototype tool, called LURETTE, is being implemented at INRIA/Spectre:

- it takes as input an observer, written in LUSTRE, describing the assumptions about the environment (i.e., the definition of "realistic"), and possibly describing the property to check (i.e., the definition of "erroneous").
- the program under test is in C. Concretely, it is obtained by compiling a LUSTRE program using the code generator SAOCG developped by Vérilog S.A.
- The generation algorithm is similar to the one presented here, but does not perform branching generation: each call to the generator produces only one test sequence. The tool is designed to allow easy modification of the various parameters.

3 Future Work

At this time, we have got an experimental workbench, to study the efficiency of various testing strategies. Experimentation can start, now, but we need first to choose efficiency criteria. Two tracks will be considered:

- Test coverage: LURETTE will be connected to the tool LOGISCOPE, also developped by Vérilog S.A, one capability of which is to measure the coverage of a set of test cases on the generated C code, according to several criteria: exercized statements, condition–to–condition paths, etc..
- Mutation analysis: This technique [DeM80] consists in introducing slight changes in a program, and in measuring the ration of such changes which are detected by a given set of test cases.

Experimentation on SAAB and Schneider case studies has been started.

A longer term research will concern liveness objectives. An idea is to compute an upper approximation of the set of states from which the objective is reachable. This can be done by backward linear relation analysis (see task 3.3). Such an approximation can be used to avoid exploring states which have no chance to lead to the objective. Moreover, by adding a counter variable, which counts the number of steps needed to reach the objective, and including this variable in the analysis, one can hope to get a measure of the distance to the goal, which could be used in test data selection.

References

[DeM80] R. A. DeMillo. Mutation analysis as a tool for software quality assurance. In *COMPSAC'80*, Chicago, October 1980.

[FJJV96] 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.

[OP94] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *5th International Symposium on Software Reliability Engineering (ISSRE'94)*, Monterey, USA, November1994.

[TMC94] 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.