Project 22703: SYRF

Deliverable 3.5.3

Automatic Testing

---

1 Contribution of Inria-Spectre

The previous works on automatic testing resulted in the implementation of the tool *Lurette*. In this tool, the program under test, considered as a black-box, is connected to a module that randomly produces *realistic inputs*. The user must describe realistic inputs by means of a *synchronous observer*, written in Lustre.

The declarative style of Lustre is particularly suitable for describing simple invariant properties, that don't strongly depend on time. For instance, to specify that some numerical input `X` is always greater than twice another input `Y`, the user may simply write the following Lustre equation: "`ok = X > 2 * Y;`".

Writing complex path properties in Lustre is possible, but often complicated. For instance, suppose that we want to state that, as soon as a Boolean input `A` becomes false, some numerical input `X` must remain positive. Such a property can be implemented by the following observer:

```
always_A = A and (true -> pre always_A);
ok = always_A or (X > 0);
```

We want to make easier the specification of such properties, by providing imperative contructs. For instance, regular expressions seem to be very suitable to express such properties; let `loop ... pool` denotes the unbounded loop, and the dot symbol denotes the sequence, the previous property can be written:

```
loop A pool . loop (X > 0) pool
```

1.1 The Lutin language

A new language, called Lutin, has been defined. A Lutin program describes a *set of traces* over a given set of variables V (that can be either Boolean, integer or real). More precisely, this set of traces is defined as a regular language whose alphabet is the set of all possible valuations of V.

Basic features. There are two levels in the language:

- **Formula expressions** are boolean expressions on the variables of V; like "`A`" or "`A and B => (X > 2*Y)`". Such a formula denotes a set of traces of length 1: all the valuations of V that satisfy the formula. All classical Boolean and numerical operators are allowed, following the Lustre syntax.

- **Traces expressions** are obtained by combining formulas with regular operators; those operators are:
  - Sequence: "T . T'" denotes the concatenation of the set of traces denoted by T and the set of traces denoted by T'.
  - Non deterministic choice: "T | T'" denotes the union of the set of traces denoted by T and the set of traces denoted by T'.
  - Unbounded loop: "`loop` T `pool`" denotes the set of traces obtained by concatenating zero or more traces belonging to T.

  Trace expressions can be enclosed within braces, but the default is that sequence has priority on union.

A complete Lutin description is defined with a name and a set of typed variables (typically the input/output variables of a program under test). The body of the descrition consists of a set of equations defining (sub)traces. There must be at least an equation defining the "main" trace:

```
trace Foo ( A : bool; X : int; Y : int);
let
   Foo = loop A pool . loop (X > 0) pool;
tel
```

Advanced features. Some features are added in order to make the language more convenient:

- **Previous operator:** In addition to classical operators, the lustre `pre` operator is allowed in the formulas. This feature is particularly suitable for describing the behaviour of numerical variables. For instance, the following trace expression states that the variable `x` is initially positive, and then is either incremented or equal to its previous value:

  ```
  (x > 0) . loop { (x = pre x + 1) | (x = pre x) } pool
  ```

- **Trace variables:** the body of a Lutin description may contain more than one trace equation. There is no need to *declare* those variables. The semantics is trivial: the left-hand part variable is *equivalent* to the right-hand part expression, so any reference to the variable may be replaced by the corresponding expression. Note that there is no restriction on the use of trace variables: in particular the user may define recursive definitions, and as a consequence non-regular traces. This is not a problem at the language level, but some tools may reject such recursive descriptions.

- **Local variables:** this kind of variables is completely different form the previous ones. A local variable has *almost* the same status as an input of the description (i.e. the variables form V). A local variable must be declared with its type in the `local` part of the description. A typical use of such variable is to bound a loop:

```
trace Foo (A : bool; B : bool);
local cpt : int;
let
   Foo = loop A pool .
         ((cpt = 0) and not A and B) .
         loop
            (B and (cpt = pre cpt + 1) and (cpt < 10))
         pool .
         loop A pool;
tel
```

1.2 The Lutin test generator

Our first attempt is to use Lutin as an alternative way for describing test scenarii. One possibility is to translate a Lutin description into an equivalent Lustre observer [1]. With such a Lutin-to-Lustre compiler, the connection with the existing testing environment (Lurette) would be obvious. But in fact, directly producing the test sequence from the imperative descrition appears to be more suitable, since the clear separation between control and data allows us to design more efficient generation algorithms.

2 Contribution of Inria-Meije

Our contribution to this task for the third year was directly prompted by a collaboration with Texas Instruments as industrial partner. The purpose is to improve state covering by completing an existing test suite, from the knowledge of reachable state space of a formal subcomponent.

Approach

The current design flow used for the specification and implementation of dedicated microprocessors for cellular phones at Texas Instruments consist mainly in establishing a high-level, while informal, reference model known as the *golden specification*. This model is written in C. It is supposed to be cycle accurate and to represent the complete architecture (including buses and memories). Later implementations in hardware should fully comply with this model. There exists a large database of tests used in simulation (in order of tens of thousand runs, some themselves involving thousands of steps), and results in outputs for those tests should match between runs on the golden spec and runs on later reimplementations (implementation are reencoded manually in VHDL).

For safety reasons some critical part of the golden spec where rewritten in Esterel, providing both a more readable and simple specification style, and formal semantics. For complexity reasons only parts of the (huge) golden spec were rewritten, so that Esterel code now coexist with previous C code. The simulator itself was transformed and extended so that, when running tests from the test suite, states reached at the given formally

[1]This approach was already used with the language and compiler *Reglo*. [2]. Reglo was designed for describing observers in a verification framework. It is equivalent to a "Boolean" restriction of Lutin. The associated tool compiles a "regular-like" description into a equivalent Lustre observer

specified subcomponent were dumped and stored on-the-fly. So, by exercising these "light" upgrades into the existing design flow one could obtain two sets: a set of *Reached* states on the Esterel subcomponent, from running the tests; a set of *Reachable* states obtained by classical symbolic state search as used traditionally in model-checking. *Reached* is of course included in *Reachable*, but here *Reachable* might be too large a set due to the fact that it considers only a subcomponent, while the rest of the system taken as environment might not allow all configurations to be freely reached.

The purpose was to complement the test suite by attempting to increase the state covering (augment *Reached*, or use Observer techniques to model more faithfully the rest of the system seen as environment, to decrease *Reachable*). The ultimate goal was to equate both sets.

The work performed here was essentially pragmatic, in the context of a given specification and a given subcomponent programmed in Esterel.

Achieved Results

In addition to the introduction of Esterel as specification language and the other transformations described above, we built a symbolic version of the *Reached* state set produced on-the-fly. We studied various algorithmic techniques to produce new tests as compact as possible (in number and in lengths), while attempting to cover as many (symbolic) configurations in *Reachable - Reached* as possible in short runs. We studied ways to reuse existing runs to have the tests expressed at the operational level of the simulator. Finally we introduced new functions in our XEVE model-checker to be able to project symbolic state spaces on subcomponents, allowing thus the introduction of observers for environment modelling in this context of analysis.

Results were quite satisfactory in that the state covering ratio quickly raised from 30% to 90%, and with more work on modelling the informal part of the refernce model as environment the ratio got fairly close to 100%. To be honest the reason for the fast improvment is in part due to the fact that existing test suites were focusing on covering of functionality rather than of states, but still our results were highly welcomed by the industrial partner.

References

[1] L. Arditi, A. Bouali, H. Boufaied, G. Clavé, M. Hadj-Chaib, R. de Simone. Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP. *FMICS'99 Workshop* , Trento, June 1999.

[2] P. Raymond Recognizing Regular Expressions by means of Dataflows Networks *ICALP'96*, Paderborn, July 1996.