Lurette: User Manual

Pascal Raymond Daniel Weber

VERIMAG*– Centre Equation, 2 Av. de Vignate – F38610 GIERES
`http://www-verimag.imag.fr/PEOPLE/Pascal.Raymond`
`Pascal.Raymond@imag.fr, daniel_weber@mail.schneider.fr`
`Fax : (33) 4.76.63.48.50`

Lurette is an environment dedicated to the automatic generation of test sequences for Lustre programs. Section 1 briefly presents the principles of Lurette. Section 2 presents the notion of observers : this presentation is neither formal nor exhaustive, it simply shows the notions that are necessary for using the tool. Sections 3, 4 and 5 present the elements that are used in Lurette. Section 6 presents the `lurette` command, that opens the graphical interface to the Lurette environment.

1 Principles

The general behavior of the test generation is the following: Lusrette builds, step by step, a sequence of input vectors satisfying a set of constraints defined by the user (this set of contraints is called *the assertion*).

Input vectors are given, one by one, to the program under test, which, as a matter of fact, runs concurrently with the input generator.

At each step, the output values from the progam are given to an "oracle" module, which checks whether the results satisfy a set of properties defined by the user (this set of properties is called *the oracle*).

Figure 1 shows how those three elements are connected together, and run in parallel.
Here is a description of the test generation, given as an algorithm:

initializations
TestSequence $\Leftarrow$ *empty*
loop
  Inputs $\Leftarrow$ *values satisfying the assertion*
  Outputs $\Leftarrow$ *run Program(Inputs)*
  Ok $\Leftarrow$ *run Oracle(Inputs, Outputs)*
  if not Ok or TestSequence large enough then STOP
  else TestSequence $\Leftarrow$ { *TestSequence ; (Inputs,Outputs)* }
end loop

One can note that the selection of an input vector (by the assertion module) may depend on the previous values of outputs (this restriction is outlined by the `pre` operator in figure 1), *but this selection can not depend on the current values of outputs.*

*Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG

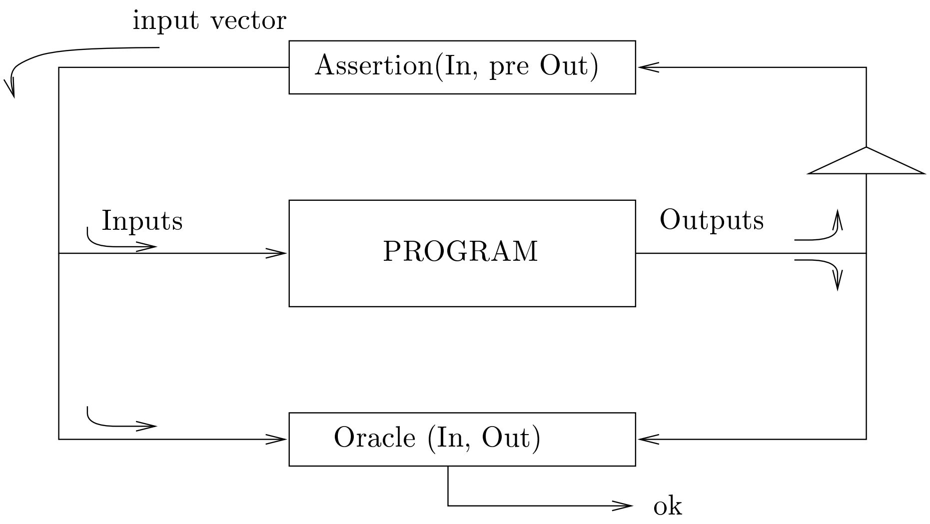

Figure 1: bevavior of the test generation

2 Observers

Both the assertion (constraints on the environment) and the oracle (expected properties of the program) must be described by *observers*.

An observer is a reactive program (in our case, a Lustre program) with several input variables (for instance $I_1 : \mathrm{Type}_1; \cdots; I_n : \mathrm{Type}_n$) and a single Boolean output (for instance ok : `bool`, but the output name is not relevant).

2.1 Sequence accepted by an observer

An observer accepts the sequence $\vec{e_1}, \vec{e_2}, \cdots \vec{e_k}$, where each $\vec{e_t}$ is a vector on its input domain $\mathrm{Type}_1 \times \cdots \times \mathrm{Type}_n$, if and only if the corresponding output sequence is always true ($ok_1 = ok_2 = ... = ok_t = true$).

2.2 Language recognized by an observer

By extension, we define the *language* recognized by the observer to be the set of all sequences (of arbitrary length) accepted by the observer.

2.3 Safety properties and observers

A property expressing the invariance along time of a given relation is said to be a `safety property`. For instance "*A and B are never true at the same time*", "*X is always greater than two times Y*" are (informal) descriptions of safety properties. Safety properies can be more complex, they can state something on the temporal behavior of variables: "*A is never true twice in a row*", or "*instants where A is true and instants where B is true alternate*".

On the contrary, a property like "*an occurrence of A is followed by a later occurrence of B*" is not a safety property. The problem is that the property refers to an event (occurrence of B) which may take place within an unbounded time interval [1].

[1] This kind of property are called *liveness properties*

This notion is quite important since safety properties are exactly those that can be exploited by Lurette: *a language recognized by an observer is a safety property, and vice versa.* As a consequence, *liveness properties can not be handled by Lurette.* The first problem when trying to express a complex temporal property is then "is it possible?".

Practically, this restriction is not really important, since a liveness property can be replaced by some more restrictive safety property. For instance, the previous one can be augmented by a "timeout" event that bounds the next occurrence of B: "*an occurrence of A is followed by a later occurrence of B, which must occur before the next occurrence of T*".

2.4 Examples

Here are some examples of properties expressed with Lustre observers.

- "A and B are never true at the same time"

```
node never_A_and_B(A, B : bool) returns (ok : bool);
let
ok = not (A and B);
tel
```

- "X is always greater to two times Y"

```
node always_X_g_2Y(X, Y : real) returns (ok : bool);
let
ok = X > (2 * Y);
tel
```

- "A is never true twice in a row"

```
node never_2A(A : bool) returns (ok : bool);
let
ok = true -> not (A and pre A);
tel
```

- "instants when A is true an instants when B is true alternate"

```
node Alternate_A_B(A, B : bool) returns (ok : bool);
var
A_forbidden, B_forbidden : bool;
let
A_forbidden = false -> pre (A or (A_forbidden and not B));
B_forbidden = false -> pre (B or (B_forbidden and not A));
ok = not(A and A_forbidden) and not(B and B_forbidden);
tel
```

3 Observer of the assertion

In order to use Lurette, one may write an *assertion observer* describing constraints on input values. There are some restrictions on writing such an observer. In order to understand those restrictions, the user must keep in mind that Lurette must solve at each instant a problem which is, without restrictions, undecidable : "what are all the input values such that the assertion's output is true?".

3.1 Interface of the observer

The inputs of the assertion must correspond (in the same order) to all the program's inputs, and then all the program's outputs.

The observer must have a single Boolean output (whose name is not important). It is recommended to scrupulously follow this rule by simply copying the actual inputs and outputs declaration of the program into the assertion header. For instance, if the program under test has the following header:

```
node Program(<params_in>) returns (<params_out>);
```

A correct header for the observer is:

```
node Observer(<params_in>; <params_out>) returns (ok : bool);
```

3.2 Data types

Imported types are not allowed in observers: only predefined types (`bool`, `int` et `real`), and also structured types (arrays, tuples) built with those basic types are supported. This restriction holds for the observer's inputs, and as a consequence for the inputs/outputs of the program under test (see section 5 for more details).

3.3 Core of the observer

The observer definition may only contain predefined operators, or node calls, that can be recursively expanded into a network of predefined operators. In particular, imported functions are not allowed.

The observer output (`ok`) may not **instantaneously** depend on the outputs of the program under test; in other terms, the current input values (which are deduced from `ok`) may only depend on the past values of the outputs.

All Boolean operators are supported, but there is a restriction on the way arithmetic operators are used in the observer. This restriction (*underlying linearity*) can be syntactically characterized, but this definition is long and boring like a day without bread. Intuitively, all numerical constraints (operators `<`, `>`, `=`, `<>` etc) must be rewritable into a Boolean formula depending only on *linear constraints*. For instance, let X, Y et Z be three numerical inputs, and C be a Boolean expression:

- `X >= 2 * Y` is linear (as it is)
- `2 >= X * Y` is not linear
- `(if C then 2 else 3) * X <> Y` is linear (rewritable in `if C then (2*X <> Y) else (3*X <> Y)`)
- `(if C then 2 else Z) * X <> Y` is not linear

In fact, a weaker restriction would have been sufficient: at each step of the test generation, the previous values of numerical variables are known. As a consequence `pre` expressions can be considered as constants while checking the linearity of the assertion. For instance `(X * pre X) > 2 * Y` can be considered as "instantaneously linear". This feature is not implemented in the current version of Lurette.

3.4 Restrictions summary

Inputs of the assertion are (in the same order) all the inputs, then all the outputs of the program under test.

No external objects : no imported type, constant or function.

References to the outputs of the program under test are only allowed under `pre` operators.

Linearity restriction : the underlying constraints on numerical inputs must be linear.

4 Oracle observer

The user may write an observer that is supposed to express the desired properties of the program under test. This *oracle observer* must follow some restrictions that are weaker than for the assertion. In particular, the oracle may instantaneously depend on the program outputs; moreover relations between numerical variables do not have to be linear. As a summary, Lurette only expects that:

Inputs of the oracle are (in the same order) all the inputs and then all the outputs of the program under test.

No external object : no imported type, constant or function.

5 The program under test

Unlike the assertion and the oracle, the program under test is considered as a *black-box* by Lurette: the tools does not deal with the Lustre source program, but with the target code (produced by some compiler Lustre→C). Nevertheless, Lurette must compile and link this C-code with its own executable libraries, so it must "know" at least how the Lustre→C compiler treats its target interface. Specific restrictions, related to the Lustre→C compilers that are supported by Lurette, are described in annex 7.

The only general restriction concerns the program inputs/outputs, which cannot contain external types (it simply reflects the restriction for the observers, that are connected to those inputs/outputs). On the contrary, the program core is considered as a black-box and may contain local variables of external types, external function calls etc.

6 Graphical interface

The `lurette` command opens a graphical interface that allows the user to perform automatic test generation (figure 2).

6.1 Building a test generator

The first step consists in defining the program under test. For this purpose, use the **Browse** button in the line labelled by "**Main Node**". The user may select a supported format for the program [2], the source file, and then the main node.

The button **Build Test Program** runs the compilation/instrumentation of the program under test. This step strongly depends on the actual format of the source program (cf. annex 7).

The compilation may need additionnal C-code, that must be declared first using the **Import Code** command. This command allows the user to manage a list of files to add in the compilation process. One more time, this step strongly depends on the source type.

At last, the code of the program under test is linked with the Lurette library to optain a stand-alone executable, called the *test generator*.

[2] Supported formats are lustre-v4 files (academic version), and Scade or SAO+ models (SC-Verilog version).

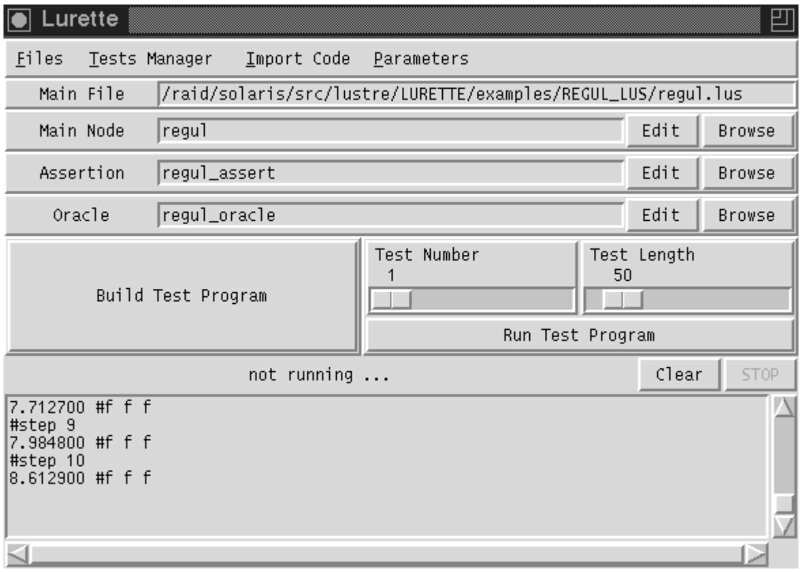

Figure 2: Lurette: graphical interface

6.2 Test sequence generation

Once built, the test generator, dedicated to the current main node, can be run by pressing the **Run Test Program**. For this purpose, the user may first define the following parameters:

The assertion node using the **Browse** button in the row **Assertion**.

The oracle node using the **Browse** button in the row **Oracle**.

The number of sequences and their maximum length using the corresponding scale widgets.

Note that the generation of a test sequence may stop before the maximum length is reached. The generation stops if the assertion can no longer be satisfied by any input values [3].

6.3 Test sequences management

The list of generated test sequences can be managed by the **Tests Manager** command. This command opens a window that allows the user to view, suppress or export to a directory a test sequence (figure 3).

The `View` button open a viewer on the currently selected test sequence. The **Inputs** (resp. **Outputs**) button allows the user to select the inputs (resp. outputs) to display. The default is to show the values of the five first inputs (resp. outputs) (figure 3).

The **Chrono** button starts `sim2chro`, a program that displays the test sequence as a chronogram [4].

Test sequences are stored in file with the `.lres` extension. Such a file is simply an ASCII text that lists the sequence of input/output values. It can then be easily used by other tools (like `sim2chro`, mentionned above).

[3] The problem has been studied in other contexts; such assertions are said to be *non-causal*

[4] Many thanks to Yann Rémond, the author of this useful program

Figure 3: Test manager and viewer

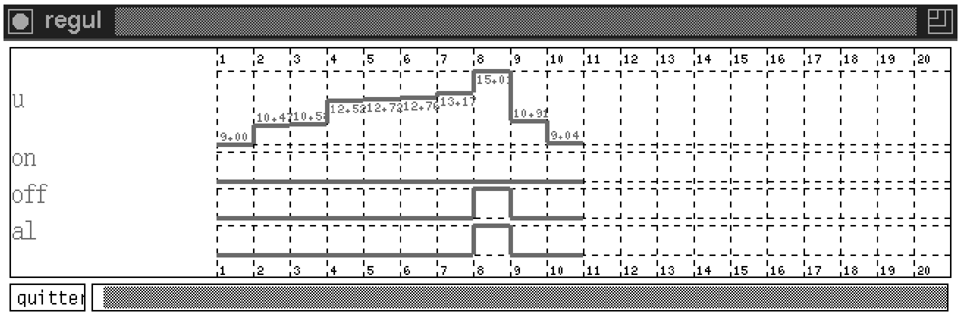

Figure 4: Test sequence viewed in sim2chro

6.4 Saving and loading a Lurette session

The **Files** menu contains commands for saving the current working environment : all information (including the test sequences that have been generated) is saved in a file with the extension `.lts`. Such a file can be loaded using the command **Load Session**.

7 Technical annex

7.1 Installation

Lurette requires the definition of several environment variables. A typical `setup` script is provided, which may be customized by the user. This script defines the path to:

- the tcl-tk libraries (version 8.0 or more is required),
- the SC-Verilog tools (scade or the previous version called SAO+),
- the academic Lustre distribution.

7.2 Associated tools

The academic distribution of Lustre (lustre-v4) is necessary [5]. The gnu-C++ (`gcc` version 2.7 or more) compiler is also necessary.

[5] This distribution is free, contact Pascal.Raymond@imag.fr, Nicolas.Halbwachs@imag.fr more informations.

7.3 Scade

Lurette supports `saofdm` files, that are produced by the tool `scade_ed` (or the previous version `sfdedit`).

The construction of the test generator for such a program is shown in figure 5.

All external code required by the Scade program must be imported using the **Import code** command. All of these imported files must follow the conventions defined by the Scade compiler (`scade-cg`). It is strongly recommended to import only source code (`.c`, `.cc` or `.h` files), and to let Lurette compile them. Nevertheless, the user may directly import machine-dependent object (`.o` files), but this feature is only guaranted to work for objects generated by `gcc`.

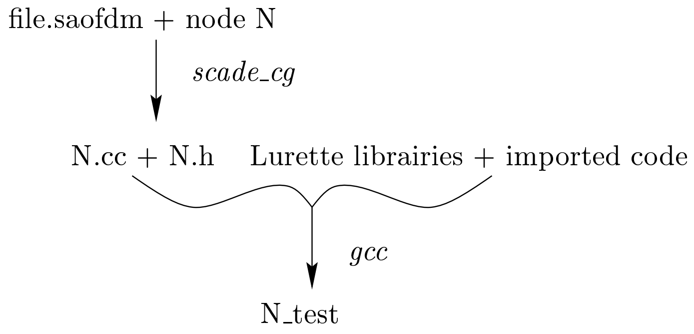

Figure 5: Building a test generator for a Scade node

7.4 Lustre

Lurette supports `lustre-V4` files (academic version). A correct installation of the lustre-v4 distribution is required.

The `Parameters` button (main menu) allows the user to choose between two ways for compiling the source program:

- `lus2oc` and then `poc` produce a C program structured as an automaton. This kind of program is efficient at run-time, but its size (resp. the compilation time) can be exponential with respect to the size of the Lustre source.

- `lus2ec` and then `ec2c` produce a C program with no control structure (single-loop program). It is not really efficient at run-time, but the size (resp. the compilation time) is always linear with respect to the size source.

Imported code (headers, C-files, or even object files) must follow the conventions defined by the `poc` compiler [6].

The construction of the test generator for a lustre program is shown in figure 6.

[6] Those conventions are fully explained in the lustre distribution manual.

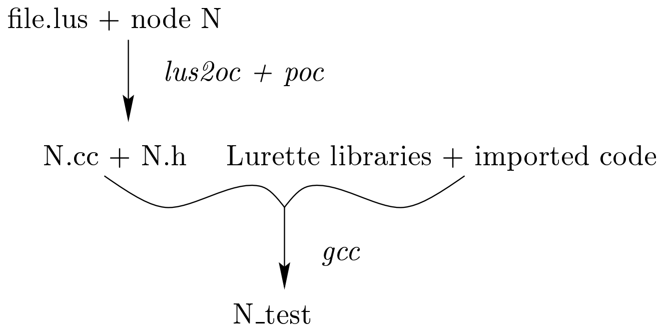

Figure 6: Building a test generator for a Lustre node