The Lurette V2 User guide

Erwan Jahier

Verimag Research Report n^{o} TR-2004-5

Initial version: 12th March, 2004
Last update: 14th October, 2015
Software Version: 1.56

Keywords: Reactive systems, validation, automatic test case generation, Lurette, Lustre, Lutin

Reviewers: Nicolas Halbwachs

How to cite this report:

```
@techreport {TR-2004-5,
   title = {The Lurette V2 User guide},
   author = {Erwan Jahier},
   institution = {{Verimag} Research Report},
   number = {TR-2004-5},
   year = {2004}
}
```

Contents

1 Lurette Principles

1.1 Introduction

The functional testing of reactive systems raises specific problems. We adopt the usual point of view of synchronous programming, considering that the behavior of such a system is a sequence of atomic reactions — which can be either time-triggered or event-triggered, or both —. Each reaction consists in reading inputs, computing outputs, and updating the internal state of the system. As a consequence, a tester has to provide test sequences, i.e., sequences of input vectors. Moreover, since a reactive system is generally designed to control its environment, the input vector at a given reaction may depend on the previous outputs. As a consequence, input sequences cannot be produced off-line, and their elaboration must be *intertwined with the execution of the system under test* (SUT). Finally, in order to decide whether a given test succeeds or fails, the sequence of pairs (input-vector, output-vector) can be provided to an observer [3] which acts as an oracle at each reaction.

Let us illustrate this process with a very simple example: consider a device whose role is to regulate the water temperature of a tank, by opening or closing a gate that controls the arrival of hot water. A reaction consists in sampling the water temperature, comparing it to the target temperature, and sending an order to the gate. In a realistic input sequence, the temperature is assumed to increase at some rate when the gate in open, and to decrease when it is closed. Hence the input at a given reaction depends on the output sent at the previous reaction. The property to be checked could be that, if the target temperature did not change during a given delay, the temperature should belong to a given interval around the target temperature. This global property of the combination [system-environment] can be checked after each reaction by an observer, which must have an internal memory to count the delay from the last target change. Moreover, from this property coverage point of view, we want to generate sequences where the target temperature do not change too often.

1.2 Environment

The main challenge to automate the testing process is to able to automate the generation of *realistic* input sequences to feed the SUT. In other words, we need an executable model of the environment which inputs are the SUT outputs, and which outputs are the SUT inputs.

The environment is also a reactive system that executes in closed-loop with the SUT. It can be very versatile or underspecified. This motivated the design of Lutin [7], a language to program stochastic reactive systems and environment models. For more information on Lutin, please refer to the Lutin language reference manual [5]. A tutorial is also available.

- http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lurette/lutin-tuto-html.html
- http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lurette/doc/lutin-man.pdf
- http://www-verimag.imag.fr/Lutin.html

1.3 Oracles

The test decision is a deterministic process, which can be automated using a formalization of SUT expected properties (predicate over traces). By convention, a Lurette oracle is a program that returns as first output a Boolean that formalizes one or several requirements. Lurette reports a property violation each time one oracle returns false. As expected properties of reactive systems often involve time, a language where time is a first-class concept like Lustre [1] is a sensible choice. Moreover, Lustre allows formalizing any safety property [2].

- http://www-verimag.imag.fr/The-Lustre-Toolbox.html (Lustre V4)
- http://www-verimag.imag.fr/Lustre-V6.html

1.4 Oracle Coverage

When performing functional testing, structural coverage criteria are relevant to give insights about whether or not enough tests have been done. We need coverage criteria attached to requirements. Consider for instance the following property, which expresses that, when a threshold is exceeded, and when the system is in its nominal mode, then an alarm must be raised:

```
(T>100 and nominal) => Alarm
```

There are several ways for the SUT to satisfy this property: (1) `T` can be smaller than 100 or (2) the system can be in a degraded (non-nominal) mode; (3) otherwise, `Alarm` must be true. From the coverage point of view, it is obviously the latter case that is interesting. Its seems fair to consider that this oracle is not covered as long as no simulation has been run where `T>100` and both `nominal` and `Alarm` are true. The case `T<100` is also interesting to cover, but can be attached to a property related to the absence of false alarms.

Hence, we define the *coverage of an oracle* as a set of Boolean conditions. A *run* (or a *trace*) of the SUT is a sequence of the SUT input/output vectors generated during a simulation. The oracle *coverage rate of a set of runs* is the rate of coverage conditions that have been true at least once during those runs. The coverage of a property is arguably part of its specification. If it is not the case, the persons in charge of formalizing requirements into oracles are in the best position to define the coverage at the same time.

cf Section 2.4 for a more detailled description of how coverage is handled in Lurette.

1.5 The Lurette workflow

The design of the system, its oracles, and its stimulus generators is not a linear process. Several iterations are necessary, that are described below, and outlined in Fig. 1.

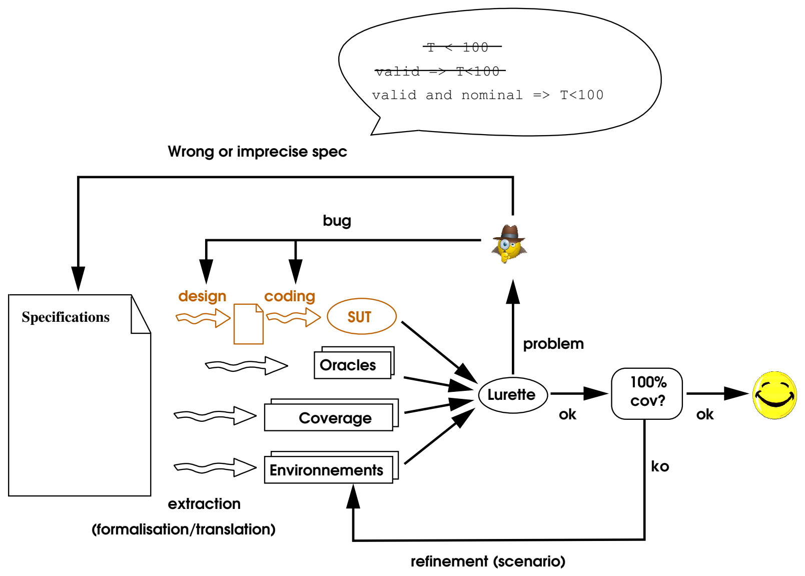

Figure 1: The Lurette iterative process loops. Oracles and environments of the SUT are extracted from heterogeneous specifications. When an oracle is invalidated, it can be due to a design error, a coding error, or to a wrong or imprecise specification. Once the system is running without invalidating oracles, in order to improve the coverage rate, the tester needs to refine test scenarios.

Refining the SUT. When an oracle is violated, it can be, of course, because of a design or coding error, which results in a erroneous SUT. Detecting such incorrect behaviors of the SUT is indeed the original motivation of all this infrastructure.

Refining oracles. An oracle violation can also be due to a wrong formalization. Despite the fact that sequence recognizers (oracles) are much simpler to develop than sequence generators (SUT), they are still the result of a human work and thus exposed to errors.

Refining ambiguous requirements. Lurette can also detect ambiguous requirements, when they are interpreted differently by the SUT and the oracle designers. It happened quite frequently during the project.

Refining inconsistent requirements. Formalizing requirements in a language such as Lustre that is equipped with a model-checker allows to detect inconsistencies, i.e., the absence of correct behaviors.

Refining imprecise requirements. Another reason that results in invalidated oracles is when they are based on imprecise requirements. One typical case encountered in the project was a requirement formulated as follows: “when `x` exceeds the threshold `t`, the alarm `a` should be raised”. In a distributed system like the one of COMON, where sub-systems communicate over buses and networks, such a requirement will be immediately violated if interpreted literally. One should permit some reaction delay, and specify its bounds.

Refining incomplete requirements. Another very common source of oracle violations is a lack of completeness. A typical example, also encountered during the project, is a requirement that states that “the temperature of tank `t` should never exceeds `100` degrees”, whereas the correct requirement was “the temperature of tank `t` should never exceeds `100` degrees when the system is in the nominal mode and the validity bit associated to its sensor is 1”.

One outcome of the project is that the Lurette tool and methodology was actually helpful for debugging and refining requirements.

Refining scenarios When the coverage is not complete, we must enrich the set of possible behaviors of the environment with new scenarios. Note that new scenarios may lead to properties violations, which lead to changes in the SUT (or in oracles); and changes in the SUT may change the coverage in turn.

2 The Lurette toplevel commands interpreter (`lurettetop`)

2.1 The interactive command interpreter

Lurette handles the test harness, by reading test parameters, executing all the reactive systems in turn (SUT, environment, oracles), computing requirements coverage, and displaying a test report. It does not impose the use of Lustre or Lutin. The reaction steps can be either time-triggered, event-triggered, or both. Presentations of Lurette can be found in [4, 6].

The interactive command interpreter is named `lurettetop`. The first command you can try is the `man` (or `m` for short), that will display a small online manual.

```
[shell] lurettetop
<lurette 1> man
```

```
This is Lurette Version 1.54 (8eb71ec)
<lurette 1>
Once lurettetop has been launched, a prompt is printed waiting for
user queries. One first need at least to set the sut (system under
test) and the environment fields like that:

    [your shell prompt] lurettetop
    <lurette> add_rp "sut:v6:heater_control.lus:heater_control"
    <lurette> add_rp "env:lutin:env.lut:main"
    <lurette> add_rp "oracle:v6:heater_control.lus:not_a_sauna"
or
    <lurette> add_rp "oracle:ec:not_a_sauna.ec:"

And then the testing process can start:

    <lurette> run
       ... [... testing ...]

Equivalently, you can directly set values at the command line:

    [your shell prompt] lurettetop -rp "sut:v6:heater_control.lus:heater_control"
                                   -rp "oracle:v6:heater_control.lus:not_a_sauna"
                                   -rp "env:lutin:env.lut:main"
    <lurette> run
       ... [... testing ...]

You migth also want to try the info command (i for short) to get the values
of the test parameters, and the h (help) command to obtain the list of
possible commands.
<lurette 2>
```

Here and in the following, `[shell]` is the shell prompt; `<lurettetop>` is the `lurettop` prompt. Let's paraphrase what this small online-help says.

```
[shell] lurettetop
<lurette 1> man add_rp XXXX il faudrait un truc comme ca. ca y est dans ldbg -> Merger ?
```

As this manuals say, you can get the list of available commands with `help` (or `h`). Another very useful command is `info` (or `i`) that returns the current values of test parameters.

```
[shell] lurettetop
<lurette> info
```

```
This is Lurette Version 1.54 (8eb71ec)
<lurette 1> The current test parameters are:
     sut:
     env:
     oracle:
     test length: 10
     precision: 2
     seed:  chosen randomly
     verbosity level: 0
     rif file name: lurette.rif
     overwrite rif file? no
     coverage file name: lurette.cov
     do we stop  when an oracle returns false? no
     display local var? no

<lurette 2>
```

2.2 The resource file (`.luretterc`)

When `lurettetop` is launched, it first reads the content of the resource file named `.lurette_rc` in the current directory. The syntax of this file is just the one of interactive commands described by the online manual (`man`) displayed above.

```
<lurette> add_rp "sut:v6:heater_control.lus:heater_control"
<lurette> add_rp "env:lutin:env.lut:main"
<lurette> set_test_length 1000
```

2.3 The batch command interpreter

`lurettetop` can also be used non-interactively, using the `--batch` option. In order to set the various parameters, you can either set a `.luretterc` resource file in your current directory, or use command line options. The list of options can be obtained using the `--help` option when invoking `lurettetop`.

```
[shell] lurettetop --help
```

```
usage: lurettetop [<options>]

lurettetop is a top level loop that let one use lurette.
Type help and/or man at the prompt for more info.

launch 'lurettetop --help' to see the available options.

  --reactive-program <string>.
  -rp <string>
  To specify a reactive program to be used in the test session, one should
  use a string with the following format: "machine_kind:language:file:node"
      where:
     - machine_kind can be : 'sut', 'oracle', or 'env'
     - language can be :
        + 'v4'     to use the Lustre V4 programs
        + 'v6'     to use the Lustre V6 programs
        + 'lutin'  to use the Lutin programs
        + 'ocaml'  to use the Ocaml programs [ocaml]
        + 'ec'     to use the ec programs
        + 'ec_exe' to use a standalone executable obtained from an .ec file [ex_exe]
     - file should be an existing file (compatible with the ''compiler'' field)
     - node should be a node of file (if meaningful) or empty
```

```
[ocaml] In the 'ocaml' mode, the file can be an f.ml file, or a f.cmxs file.
If an ml file is provided, lurettetop try to generate a cmxs file from it.
If your caml program depends on library, or on other files, please generate
the f.cmxs file by yourself (cf the ocaml documentation).

[ec_exe] In the 'ec_exe' mode, lurette suppose that 'file.ec' has been compiled
into an executable that is named 'file' (for instance, via ec2c -loop).
That executable must read/write its I/O using the RIF convention.
The rationale for that mode is to be able to deal with Lustre programs that
use C code. The 'file.ec' is just used to retrieve the I/O var names
and types actually.

An alternative format is the following: "machine_kind:socket:sock_addr:port" where
   - machine_kind is as above
   - sock_addr is a valid internet adress
   - port is a free port on sock_addr

The lurettetop process play the role of the client ; exchanges on the socket
should respect the RIF (Reactive Input Format).

Hence, to sum-up, we currently support:

"<sut|env|oracle>:lutin:prog:node"        For lutin programs
"<sut|env|oracle>:v6:prog:node"           For lustre V6 programs
"<sut|env|oracle>:v4:prog:node"           For lustre V4 programs
"<sut|env|oracle>:ec:prog:"               For lustre expanded code programs
"<sut|env|oracle>:ec_exe:prog:"           For lustre expanded code programs that have been compiled
"<sut|env|oracle>:socket:addr:port"       For reactive programs that read/write on a socket
"<sut|env|oracle>:socket_init:addr:port"  Ditto + read I/O init values before the first step

  Examples:
   "sut:v6:controler.lus:main"
   "env:lutin:train:tgv"
   "oracle:socket:127.0.0.0:2042"

--sut <string>        File name of the system under test [works with --old-mode only!].
-sut  <string>

--sut-cmd <string>    Command that launches the system under test [works with --old-mode only!].
-sut-cmd  <string>

--oracle-cmd <string> Command that launches the oracle [works with --old-mode only!].
-oracle-cmd  <string>

--main-sut-node <string>      Main node name of the system under test [works with --old-mode only!].
-msn  <string>

--main-env-node <string>      Main node name of the environment (meaningful for lutin only) [works with --o
-men  <string>

--oracle <string>     File name of the oracle [works with --old-mode only!].
-oracle  <string>

--main-oracle-node <string>   Main node name of the oracle [works with --old-mode only!].
-man  <string>

--sut-compiler <string> (lv4, lv6, scade)      Compiler used for the sut [works with --old-mode only!].
--oracle-compiler <string> (lv4, lv6, or scade)        Compiler used for the oracle [works with --old-mode
--cov-file <string>   file name coverage info will be put into
--test-length <int>   Number of steps to be done
-l <int>              (currently, 10).

--precision <int>     number of digit after the dot used for floating points.

-p  <int>
```

```
--fair                Compute the polyhedra volumes before drawing:
--compute-poly-volume more fair, but more expensive.

--thick-draw <int>    Number of draw to be done in each formula
-td <int>             at each step (currently, 1).

--draw-inside <int>   Draw on the edges of the convex hull of solutions.
--draw-edges <int>    Draw on the edges of the convex hull of solutions.
--draw-vertices <int> Draw among the vertices of the convex hull of solutions.

--draw-all-formula    Tries all the formula reachable from the current state.
--draw-all-vertices   Tries all the polyhedra vertices.

--seed <int>          Seed the random engine is initialised with.
-seed  <int>

--dbg  debug mode (to debug lurettetop)

-ldbg  use the lurette debugger

--output <string>     Set the output file name (currently,  "lurette.rif").
-o <string>

--overwrite-output    Overwrite "lurette.rif" if it exists without tring to invent a new name
-oo

--batch                       Start the testing process directly, without prompting.
--go

-go

--step                Run lurette step by step.
-s

--socket-inet-addr            Set the socket address.

--socket-io-port              Set the socket io port.

--socket-err-port             Set the socket error port.

--show-step           Set on the show step mode.
--do-not-show-step    Set off the show step mode.

--verbose             Set the verbose level.
-v

--reactive            Set on the reactive mode.
-r

--prompt              Replace the default prompt.

--extra-source-files          Set the EXTRA_SOURCE_FILES environment variable.

--extra-libs          Set the EXTRA_LIBS environment variable.

--extra-libdirs       Set the EXTRA_LIBDIRS environment variable.

--extra-includedirs   Set the EXTRA_INCLUDEDIRS environment variable.

--step-mode           Set the step mode used to perform the step.

--delay-env-outputs    Delay the outputs of the environements before transmitting them to the oracles.
--luciole      Call lurette via luciole.
--pre-processor       Pre-processor for Lucky files (e.g., cpp).
```

```
  -pp

  --prefix              A string to append before the call to lurette (e.g., "/usr/bin/times ").

  --tmp-dir             Use that directory to put temporary files.

  --log                 Redirect stdout to a log file (lurette_stdout.log)

  --gnuplot             Call gnuplot.
  --no-gnuplot  Do not call gnuplot.
  -ngp

  --sim2chro            Call sim2chro.
  --no-sim2chro         Do not call sim2chro.
  -ns2c

  --local-var           Display environment local variables in sim2chro (on).
  --no-local-var        Do not display environment local variables in sim2chro.

  --direct      Set the direct mode.

  --old-mode    Unset the direct mode.

  --ocaml-version               Display the version ocaml version lurette was compiled with and exit.
  --version             Display the version and exit.
  --help                Display this list of options.
```

If a resource file exists in the current directory, `lurettetop` will first interpret its content, and then interpret the command-line batch options (and thus override the `.lurette_rc` commands).

The `batch` commands generates a lurette.batch file with current test parameters.

2.4 Oracle coverage

In Lurette, in order to define the coverage of an oracle, one just needs to add additional Boolean variables to its output profile. By convention, the first output holds the oracle result, and the following outputs define the oracle coverage. Lurette updates the coverage rate from one execution to another (via a file). This coverage rate is reset each time either the oracle or the SUT is modified.

cf Section 2.4.

The coverage file (`.cov`). Lurette maintains the coverage information via a `.cov` file, which looks like this:

```
SUT: v6:heater_control.lus:main
ORACLE: v4:heater_control.lus:not_a_sauna
ORACLE: v6:heater_control.lus:not_a_fridge
RIF: test.rif0 - generated at 16:28:15 the 18/4/2011 ; the coverage rate is 50.0%
RIF: test.rif0 - generated at 16:29:20 the 18/4/2011 ; the coverage rate is 60.0%
VAR: c1 t
VAR: c2 f
VAR: c3 t
VAR: c4 t
VAR: c5 f
VAR: c6 t
```

The coverage is a function from

- a sut
- an oracle
- a set of runs (rif files).

The header contains information about the sut and the oracle we measure the coverage of (`SUT:` and `ORACLE:`). Then comes the set of runs that have been performed on them (`RIF:`). Finally, comes the list of coverage variables, and their status indication if there had been true at least once during a run (`t` meaning covered).

When no coverage file is specified, Lurette creates such a file using as coverage conditions all the Boolean outputs of oracles, the first one excepted (as it is used to hold the test verdict).

If one wants to remove a coverage condition without changing the oracle profiles, it suffices to remove (or comment) the corresponding line in this coverage file. When the SUT or the oracle changes, the coverage is reset. One can also force the coverage resetting using the `--reset-cov-file true` option (in batch mode) or the `reset_cov_file true` commande (in interactive mode).

cf also the `check-rif` bacth tools in Section 3.7.

3 Third-party tools used by `lurettetop`

Some of the `lurettetop` commands actually rely on stand-alone executables that can be used in command-line or scripts. We present them briefly in the following.

When strange messages are displayed by `lurettetop`, it migth be easier to understand what happens with those stand-alone tools. For example, imagine you want to test with Lurette a Lustre V4 program that requires several environment programs with several oracles, it migth

3.1 A summary of tools and dataflow

Lurette handles the test harness, by reading test parameters, executing all the reactive systems in turn (SUT, environment, oracles), computing requirements coverage, and displaying a test report. Figure 2 outlines what happens during a test run.

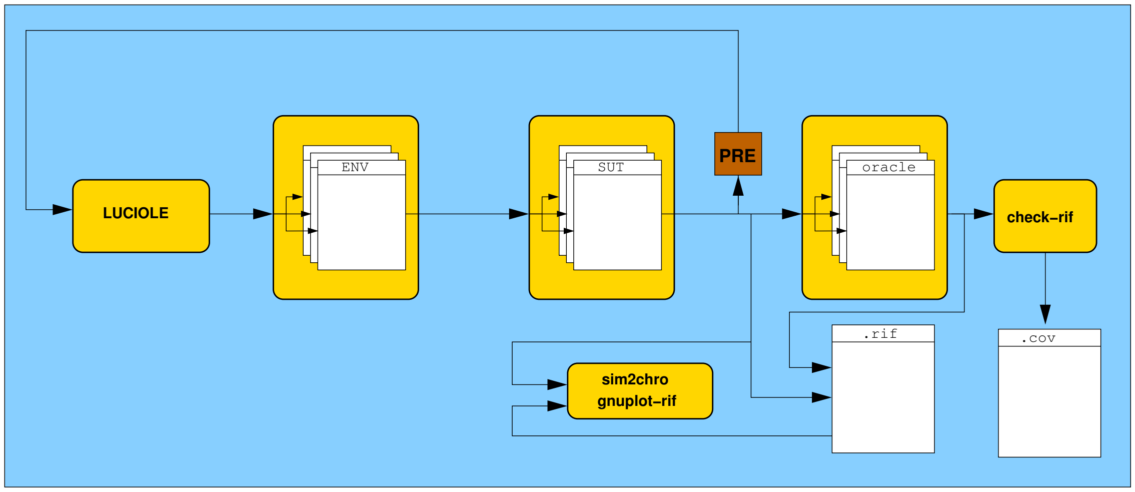

Figure 2: Lurette data flow. At each step, each orange box is activated from left to right. A Luciole process is used only if necessary (i.e., if a SUT or an ENV input is missing). At least one SUT or one ENV is necessary. The sim2chro process can be launched from Luciole, or post-mortem from the Lurette UI via the generated rif file.

The environment outputs are the SUT inputs, and SUT outputs are the environment inputs. In order to be able to start such a looped design, one entity have to start first. The choice has been made that the environment will. This means that a valid environment for Lurette is one that can generate values without using its input at the first instant.

If one input is missing to the SUT or to its environment, a tcl-based GUI (Luciole) is launched to generate the missing variables. The Luciole process is started before the environment ones.

The SUT, the environment, and oracles can be splitted into several reactive programs. Each variable should be produced exactly by one program. This is checked by Lurette before the test is launched. The binding between programs is done by name.

3.2 The Lutin interpreter (`lutin`)

```
[shell] lutin -help
```

```
usage: lutin [options] <file> | lutin -help

-n, -m, -node, -main <string>
            Set the main node
--version, -version
            Print the current version and exit
-v          Set the verbose level to 1
-vl <int>   Set the verbose level
-gnuplot, -gp
            call gnuplot-rif to display data (type 'a' in the gnuplot window to refresh it).
-rif, -quiet, -q, -only-outputs
            display only outputs on stdout (i.e., behave as a rif input file)
-o <string> output file name
-L, -lib <string>
            Add a dynamic library where external code will be searched in
-seed <int> Set a seed for the pseudo-random generator
-boot       Perform ther first step without reading inputs
--max-steps, -l <int>
            Set a maximum number of simulation steps to perform
--step-inside, -si
            Draw inside the convex hull of solutions (default)
--step-edges, -se
            Draw a little bit more at edges and vertices
--step-vertices, -sv
            Draw among the vertices of the convex hull of solutions
-precision, -p <int>
            Set the precision used for converting float to rational (default: 2)
-locals, -loc
            Show local variables in the generated data.
--ocaml, -ocaml
            Generate ocaml glue code that makes it possible to call the lutin interpreter
            from ocaml with the current set of arguments.
-luciole, --2c-4luciole
            Call Luciole the provide inputs
-h, -help, --help
            Display this message
-more       Show hidden options (for dev purposes)
```

3.3 The Lustre V4 interpreter (`ecexe`)

The Lustre V4 compilation tools chain rely on so-called « Lustre expanded code », or `ec` for short.

`ec` stands for expanded code.

XXX un xfig decrivant la chaine d'outil lustre V4 avec lus2ec, ec2c, ec2ec, exexe, lesar

```
<lurettetop> add_rp "sut:v4:heater_control.lus:heater_control"
```

When adding a reactive programs flagged the the `v4` option, `lurettetop` actually uses `ecexe` to interpret the `ec` generated by the `lus2ec` compiler.

3.4 The Lustre V6 interpreter (`lus2lic`)

3.5 Interactive simulation (`luciole`)

Sometimes it's useful to have interactive test sessions, typically be before writing a first Lutin environment for a SUT.

```
[shell] luciole -h
```

```
 nil
```

3.6 Gnuplot-based data visualisation (`gnuplot-rif`)

```
[shell] gnuplot-rif --help
```

```
 gnuplot-rif [options] <f>.rif

Generates a <f>.gp file such that gnuplot can plot the rif file.

gnuplot-rif try to read the content of a file named .gnuplot-rif (in the
current directory). With something like:

   hide T
   hide toto*

It will ignore all I/O which names begin by 'toto', as well as the variable 'T'.
If you write in this file something like:

   show xx*

it will show show only I/O beginning by 'xx'. With

  plot_range 12 42

it will plot data from step 12 to 42 only. With

  dynamic
  window_size 56

will show only the last of 56 steps of the simulation (40 by default).

If one 'show' statement is used, all hide statements are ignored.
If several plot_range or window_size are used, the last one win.

All these values can be overriden by setting options.

Command-line options are handled afterwards.

  -no-display    generate the .gp file, without launching gnuplot
  -dyn   dynamically display the end of the rif file
  -size         <s> set the size of the sliding window in -dyn mode
  -min  <min> only display steps > min (ignored in -dyn mode)
  -max  <max> only display steps < max (ignored in -dyn mode)
  -nogrid remove the grid (useful with -dyn)
  --hide-var <string> hide a variable (one can use the wildcard '*')
  --show-var <string> show a wildcard-hided variable

  -wxt   launch gnuplot with the wxt terminal (default)
  -x11   launch gnuplot with the X11 terminal
  -jpg   output in a jpg file
  -pdf   output in a pdf file
  -ps    output in a B&W post-script file
  -cps   output in a color post-script file
  -eps   output in a color encapsulated post-script file
  -latex  output in a latex file

  -v             set on a verbose mode
  -h             display this help message
```

3.7 Post-mortem coverage analysis: (`check-rif`)

Lurette met à jour la couverture des test (par l'entremise du fichier `.cov`) au fur et à mesure que les tests se déroulent. Mais il est également possible s'utiliser un petit utilitaire en ligne de commande : `check-rif`.

Cet utilitaire utilise un oracle et un fichier .rif généré lors d'une précédente session de test pour calculer (où mettre à jour) la couverture fonctionnelle associée à cet oracle. La commande `check-rif -help` explique tout ce qu'il y a à savoir sur cet outil.

```
[shell] check-rif --help
```

```
Usage:
        check-rif [options]* -ec <file>.ec <Rif File name to check>

 Performs post-mortem oracle checking using ecexe.

 The set  of oracle Inputs  should be included  in the set of  the RIF
 file inputs/outputs.

 At the first  run, the coverage information is  stored/updated in the
 coverage file  (cf the  -cov option to  set its name).  The variables
 declared in  this file should be  a subset of the  oracle outputs. If
 the coverage  file does not  exist, one is  is created using  all the
 oracle outputs. If  not all those outputs are  meaningfull to compute
 the coverage rate, one just need to delete corresponding lines in the
 coverage file.  The format of  the coverage file  is straightforward,
 but deserves respect.

 Options are:

  -ec <string>  ec file name containing the RIF file checker (a.k.a., the oracle)
  -cov <string> Override the default coverage file name (<oracle name>.cov by default).
  -reset-cov    reset the coverage rate (to 0%) before running
  -stop-at-error        Stop processing when the oracle returns false
  -debug        set on the debug mode
  --help        Display this list of options.
```

3.8 An experimental debugger for Lustre (V6) and Lutin (`ldbg`)

```
[shell] ldbg -help
```

```
  This is ldbg Version 1.54 (a122bf0)
type 'man' for a small on-line manual
Usage: ocaml <options> <object-files> [script-file [arguments]]
options are:
  -absname  Show absolute filenames in error messages
  -I <dir>  Add <dir> to the list of include directories
  -init <file>  Load <file> instead of default init file
  -labels  Use commuting label mode
  -no-app-funct  Deactivate applicative functors
  -noassert  Do not compile assertion checks
  -nolabels  Ignore non-optional labels in types
  -noprompt  Suppress all prompts
  -nopromptcont  Suppress prompts for continuation lines of multi-line inputs
  -nostdlib  Do not add default directory to the list of include directories
  -ppx <command>  Pipe abstract syntax trees through preprocessor <command>
  -principal  Check principality of type inference
  -rectypes  Allow arbitrary recursive types
  -short-paths  Shorten paths in types
  -stdin  Read script from standard input
  -strict-sequence  Left-hand part of a sequence must have type unit
  -unsafe  Do not compile bounds checking on array and string access
  -version  Print version and exit
  -vnum  Print version number and exit
  -w <list>  Enable or disable warnings according to <list>:
        +<spec>   enable warnings in <spec>
        -<spec>   disable warnings in <spec>
        @<spec>   enable warnings in <spec> and treat them as errors
     <spec> can be:
        <num>             a single warning number
        <num1>..<num2>    a range of consecutive warning numbers
        <letter>          a predefined set
     default setting is "+a-4-6-7-9-27-29-32..39-41..42-44-45"
  -warn-error <list>  Enable or disable error status for warnings according
     to <list>.  See option -w for the syntax of <list>.
     Default setting is "-a"
  -warn-help  Show description of warning numbers
  -dsource  (undocumented)
  -dparsetree  (undocumented)
  -dtypedtree  (undocumented)
  -drawlambda  (undocumented)
  -dlambda  (undocumented)
  -dinstr  (undocumented)
  - <file>  Treat <file> as a file name (even if it starts with `-')
  -help  Display this list of options
  --help  Display this list of options
```

4 Installation process

XXX start me

5 A small tutorial

ccc

A RIF: Reactive Input Format

RIF stands for *Reactive Input Format*. It is the format used by the synchronous Verimag tools for writing and reading sequences of input and output data vectors. We recall in this section what this format looks like.

Data. A RIF file is a sequence of data values separated by spaces, newlines, horizontal tabulations, carriage returns, line feed and form feeds. A data value can be either an integer, a floating-point or a Boolean ({t}, {T}, or {1} stands for {true} ; {f}, {F} or {0} stands for {false}).

Comments. Single line comments are introduced by the two character **#** and terminated by a new line. Multi-line comments are introduced by the two characters **@#**, and terminated by the two characters **#@**.

Pragmas. Pragmas are special kinds of comments, that migth (or not) be taken into account by tools that reads RIF data. One-line pragmas have the form **#pragma_ident ...** , and multi-line pragmas the form **@#pragma_ident ... #@**.

The most common pragmas used by verimag tools are (using BNF notation):

- **@#inputs** (var name : var type)+ **#@** or
- **#inputs** (var name : var type)+ to declare the list of input variable names and types;
- **@#outputs** (var name : var type)+ **#@** or
- **#outputs** (var name : var type)+ to declare the list of output variable names and types;
- **@#locals** (var name : var type)+ **#@**

or

- **#locs** to indicate that the following data correspond to local variables; to declare the list of local variable names and types;
- **#outs**, to indicate that the following data correspond to output variables;
- **#step** int, to indicate that a new step is starting, and that the following data correspond to input variables.

Note that those pragmas are necessary for RIF file viewers such as {sim2chro} and {gnuplot-rif} to work properly.

A RIF file example is provided in Figure **??**; it corresponds to the timing diagram of Figure **??**.

```
# seed = 97040004
#program "lurette chronogram (degradable-sensors.luc) "
#@inputs
"T":real
"T1":real
"T2":real
"T3":real
@#
```

```
#@locals
"degradable-sensors__cpt":int
"degradable-sensors__eps":real
"degradable-sensors__eps1":real
"degradable-sensors__eps2":real
"degradable-sensors__eps3":real
@#
#@outputs
"Heat_on":bool
@#
#step 1
7.00 7.00 7.00 7.00 #outs T
#locs 0 0.08 -0.05 -0.05 0.10
#step 2
7.13 7.20 7.16 7.18 #outs T
#locs 1 0.13 0.07 0.03 0.05
#step 3
7.27 7.37 7.27 7.18 #outs T
#locs 2 0.14 0.10 -0.00 -0.09
#step 4
7.45 7.47 7.38 7.36 #outs T
#locs 3 0.18 0.02 -0.07 -0.09
#step 5
7.59 7.68 7.61 7.56 #outs T
#locs 4 0.14 0.09 0.02 -0.03
#step 6
7.65 7.58 7.64 7.55 #outs T
#locs 5 0.06 -0.06 -0.01 -0.09
#step 7
7.84 7.91 7.94 7.90 #outs T
#locs 6 0.20 0.07 0.10 0.06
#step 8
8.00 8.07 8.00 8.09 #outs T
#locs 7 0.15 0.07 0.00 0.09
#step 9
8.12 8.09 8.17 8.16 #outs T
#locs 8 0.13 -0.03 0.05 0.04
#step 10
8.26 8.29 8.30 8.20 #outs T
```

References

[1] 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.

[2] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language lustre. In *ISLIP'93, Quebec*, 1993.

[3] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, The Nederlands, 1993. Workshops in Computing, Springer Verlag.

[4] Erwan Jahier, Nicolas Halbwachs, and Pascal Raymond. Engineering functional requirements of reactive systems using synchronous languages. In *International Symposium on Industrial Embedded Systems, 2013. SIES'13.*, Porto, Portugal, 2013.

[5] Erwan Jahier and Pascal Raymond. Lutin Reference Manual. http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lurette/doc/lutin-man.pdf.

[6] Erwan Jahier, Pascal Raymond, and Philippe Baufreton. Case studies with lurette v2. *Software Tools for Technology Transfer*, 8(6):517–530, nov 2006.

[7] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: a language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems*, 2008.

B Connecting to Lurette using sockets

B.1 The socket plugin API

Lurette entities (SUT, oracles, environments) can also be a stand-alone program that reads and writes its inputs/outputs on a socket. Of course, such programs ougth to respect a precise protocol that we describe below.

1. First it must **connect to an inet socket** (defined by an address and a port), using the listen command;
2. Then it must write its **input variable names and types** (that will be received from the lurette process) using the RIF convention
3. Then it must write its **output variable names and types** (that will be send to lurette process) using the RIF convention
4. Then, it enter a loop where it
   - reads its input on the socket (in their declaration order)
   - writes its output on the socket (in their declaration order)

In order to stop that loop and inform lurette it wants to stop playing, the program just have to send the #quit command.

Here a small but complete example of a C program that is able to communicate with Lurette (and that is part of its non-regression test suite) :

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <locale.h>
#ifdef _WINSOCK
  #include <windows.h>
  #include <process.h>
  #pragma comment(lib, "Ws2_32.lib")
#else
  #include <sys/socket.h>
  #include <netinet/in.h>
  #include <netdb.h>
#endif
#define MAX_BUFF_SIZE 2048 // To be set with care!!
#define SOCK_ADDR "127.0.0.1"
#define SOCK_PORT 2000
typedef int _bool;
// A little program with 3 inputs, and 3 outputs, that dialogs with lurette on a socket.
int main(){
  int i = 0;
  int rc = 0;
  char buff [MAX_BUFF_SIZE];
  int sock, sockfd, newsockfd, clilen;
  struct sockaddr_in serv_addr, cli_addr;
  // The program Inputs:
  int a;  _bool b;  double c;
  char b_char; // used for reading booleans on the socket
  // The program Outputs:
  int x=0;  _bool y=0;  double z=0.0;
  // Socket administration
#ifdef _WINSOCK
  WSADATA WSAData;
  WSAStartup(MAKEWORD(2,0), &WSAData);
#endif
  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0) printf("Error: opening socket");
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = inet_addr(SOCK_ADDR);
  serv_addr.sin_port = htons(SOCK_PORT);
  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) ) {
    printf("Error when binding %s:%d\n", SOCK_ADDR, SOCK_PORT); exit(2);
  }
  listen(sockfd,5);
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  if (newsockfd < 0) printf("Error: on accept");
  sock = newsockfd;
  // Make sure that the program uses a "." for reals, and not a ","
  setlocale(LC_ALL, "English");
  // sending I/O declarations
  memset(buff, 0, MAX_BUFF_SIZE);
  sprintf(buff, "@#inputs \nx:int \ny:bool\n z:real \n#@\n"); // multi-line rif decl
  send(sock, buff, (int) strlen(buff),0);
  sprintf(buff, "#outputs a:int b:bool c:real\n"); // single-line rif decl (just for fun)
  send(sock, buff, (int) strlen(buff),0);
  for (i=1; i<10; i++){ // The main loop
    // 1 - Reading inputs
    memset(buff, 0, MAX_BUFF_SIZE);
    rc = recv(sock, buff, MAX_BUFF_SIZE, 0);
    if (rc<0)  { printf("Error: cannot read on socket\n"); exit(2); };
    sscanf(buff, "%d %c %lf", &a, &b_char, &c);
    // Translate char into int
    if ((b_char == '0') || (b_char == 'f') || (b_char == 'F')) b = 0;
    if ((b_char == '1') || (b_char == 't') || (b_char == 'T')) b = 1;
    // 2 - Computing the outputs using the inputs
    x = a+1; if (b) { y = 0; } else { y = 1; }; c = z+0.1;
    // 3 - Writing outputs
    memset(buff, 0, MAX_BUFF_SIZE);
    sprintf(buff, "%d %d %lf \n", a, b, c);
    send(sock, buff, (int) strlen(buff),0);
    // A small debug-printf to see what's going on...
    printf("#step %d \n%d %d %f #outs %d %d %f\n", i, a, b, c, x, y, z);
  }
  sprintf(buff,"#quit\n");
  send(sock, buff, (int) strlen(buff),0);
  return 0;
}
```

B.2 The liosi format

Liosi stands for "Lurette input output socket interface". The objective of the liosi format is to define in a single file all the necessary information to be able to connect any reactive system to Lurette using sockets. A liosi file contains the following information:

- sock_addr: the IP adress of the machine where the SUT runs;
- sock_port: the port number of the socket;
- input: SUT inputs name and type;
- output: SUT outputs name and type;

In EBNF, the syntax grammar is the following:

```
"sock_addr" <int>"."<int>"."<int>"."<int>
"sock_port" <int>

{ "input"  <string> [ <string> ] ( "bool" | "int" | "real" ) }
{ "output" <string> [ <string> ] ( "bool" | "int" | "real" ) }
```

Here is an example with 3 inputs and 3 outputs.

```
sock_addr 127.0.0.1
sock_port 2999

input EXE100BA_Panne1_OnOff bool
input EXE101MN_Panne2_OnOff bool
input EXE100BA_Panne1_Valeur real

output N1_501MT_Mes_mV_N1 real
output N1_502MT_Mes_mV_N1 real
output N1_503MT_Mes_mV_N1 real
```

The optionnal string in the I/O declaration list is used to make a mapping between the variable names used by the SUT and by Lurette is ever their differ.

```
sock_addr 127.0.0.1
sock_port 2999

input EXE100BA_Panne1_OnOff EXE100BA:Panne1_OnOff: bool
input EXE101MN_Panne2_OnOff EXE101MN:Panne2_OnOff: bool
input EXE100BA_Panne1_Valeur EXE100BA:Panne1_Valeur: real

output N1_501MT_Mes_mV_N1 N1_501MT:Mes_mV_N1: real
output N1_502MT_Mes_mV_N1 N1_502MT:Mes_mV_N1: real
output N1_503MT_Mes_mV_N1 N1_503MT:Mes_mV_N1: real
```

For example, EXE100BA_Panne1_OnOff is the Lurette nane, and EXE100BA:Panne1_OnOff: is the name used by the SUT. Most of the time, we use in Lurette the variable name used by the SUT. However, in this particular example, the SUT variable name uses semi-colons, which is not a valid variable name for Lurette.

We do have a C++ liosi parser that allows to connect easily any SUT that can be interfaced with C++ (available on demand).

C Connecting to Lurette using shared libraries (DLL/SO)

Its works, but it's not documented yet.

XXX start me