

Luciole v2

Pascal Raymond

May 2026

Contents

1	Luciole v2 overview	2
1.1	Differences with <i>luciole</i> v1	2
1.2	Components	2
1.3	Basic usage: <i>luciole</i> command line	2
1.3.1	<i>luciole</i> <file.dro>	3
1.3.2	<i>luciole</i> <file.ec>	3
1.3.3	<i>luciole</i> {<file.lus>} [-lv6] [-dro]	3
1.4	Advanced usage	3
2	Default Reactive GUI	4
2.1	Overview	4
2.1.1	Menus	4
2.1.2	Layout	5
2.1.3	Widgets	5
2.2	Resource file (luciolerc)	5
2.2.1	Tools+Sim2chro	5
3	Luciole in details	6
3.1	Example of luc script	6
3.1.1	Header	7
3.1.2	Connection to the reactive program	7
3.1.3	GUI and main loop	8
3.2	Customizing the GUI	8
3.2.1	Basic layout	9
3.2.2	Leaves	9
3.2.3	Usage of tk's standard options	9
4	Ptk package	10
5	Examples and demos	10
5.1	basic	10
5.2	rotaquin	10

1 Luciole v2 overview

Luciole v2 is a set of libraries and utilities for quickly build GUI's (Graphical User Interface) for reactive synchronous programs. It targets in particular Lustre programs, but can handle other reactive programs encapsulated into DRO libraries (Dynamic-Loaded Reactive Object).

1.1 Differences with *luciole* v1

This section quickly presents the similarities and differences with the previous version of *luciole*. If you are not familiar with the previous version, you can skip this part.

- The new top-level script *luciole* is intended to provide exactly the same features as the previous one. In particular it allows the execution of:
 - *dynamically-loaded reactive object* (.dro files), present since version 1.7,
 - *Lustre expended code* (.ec file), which is the historical supported format.
- Moreover, the script embeds *helpers* to build .ec or .dro (new) from lustre v4 and lustre v6 (new) programs.

Aside these similarities, the core of the tool has been completely remade. In particular, Luciole v2 is almost entirely developed in pure tcltk, while v1 was mostly developed in c++. It makes the portability of the tool much simpler, since it only requires a standard install of tcltk.

- The executable *simec* no longer exists (but the *luciole* script provides the same functionality).
- The .iop (input/output panel) format is abandoned. It was used to customize GUI layouts, and is replaced by a tcltk library (*luc*) that provides a similar programming style.

1.2 Components

Luciole is based on tcltk 8.6, and is made of several components:

- **lustubs.so** is a tcl-stub dynamic library that provides the necessary code for loading and running reactive programs. This is the only part which is machine dependent.
- **ptk** is a tcltk package for developing GUI's in a more structured and abstract way than raw tcltk. It is developed in pure tcltk, and not specifically dedicated to *luciole*.
- **luc** is a tcltk package, based on *ptk*, that provides the bricks for building a reactive GUI and link it to the reactive programs.
- **.luc files** are tcl scripts based on *luc* package. Each .luc file implements a particular GUI for a particular reactive program. They can be written by hand, or (transparently) generated by *luciole*, via an utility called *rp2luc*.
- **luciole** is the top-level script, that builds all the necessary stuff to run a reactive GUI according to the user arguments.

1.3 Basic usage: *luciole* command line

For a basic usage, the *luciole* script is sufficient, in particular it does not require to manipulate a .luc script. The command-line arguments and the functionalities are the

same as *luciole v1*.

- Argument specify which reactive program to run (see subsections for details).
- *luciole* builds and run a default GUI for running this particular program: The program inputs/outputs are implemented as graphical widgets (buttons, sliders, labels etc.), and displayed in a default 2-columns layout. The default GUI is similar to the one of *luciole v1* (for those who are familiar with).

luciole supports two methods for running reactive program:

- execution of native code via a *dro* file. (compiled reactive system, packaged into a dynamic library)
- interpretation of *ec* code (Lustre expanded code) via the *ecexe* library (embedded in *Luciole*).

Moreover, *luciole* provides facilities to deal with other Lustre formats, in which case **external tools** are necessary, as detailed in the sequel.

1.3.1 *luciole* <file.dro>

Executes native code packaged in the *dro* file.

1.3.2 *luciole* <file.ec>

Interprets the *ec* code (with embedded *ecexe* machine).

1.3.3 *luciole* {<file.lus>} [-lv6] [-dro]

Generates a proper *ec* file, requires **external tools**:

- using Lustre v4 *lus2ec* (default), in which case a single lustre-v4 file is expected,
- using Lustre v6 *lv6* compiler, if *-lv6* is specified, in which case one or more lustre-v6 files are expected.

Then:

- if *[-dro]* is specified, a *dro* file is generated (via *ec2dro*), loaded, and executed,
- otherwise the *ec* file is loaded and interpreted, as usual.

1.4 Advanced usage

For advanced usage, some knowledge on *Luciole* components is necessary. Some knowledge on *tcl* programming is indeed also necessary.

Each component is explained in details in the sequel. Here is a list of typical needs, together with the sections to look for:

- **Adding features to the default gui**: see *luciolerc.tcl*, in *default reactive GUI*.
- **Customizing the default in/out layout**: some knowledge on the *luc* package is necessary, more precisely the *luc panel description* section.
- **Slightly/Completely modify the default GUI**: if possible, you can partly rely the *luc* library; for really specific need you will have to program directly at the *ptk* level for the graphical part, and the *lustubs.so* level for executing reactive programs. In this case, skills in *tcl* programming is **mandatory**.

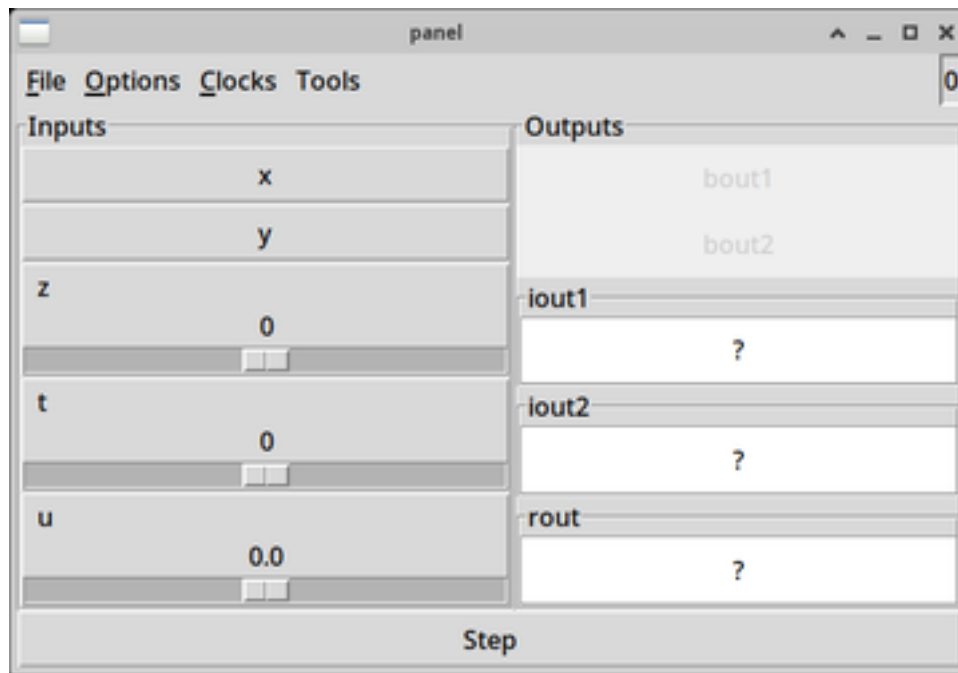


Figure 1: Default GUI example

2 Default Reactive GUI

2.1 Overview

When called with a reactive program (either lustre or dro), Luciole generates (via an utility called *rp2luc*) a luc program with a simple layout: - a generic menubar, - two columns of widgets (inputs and outputs), - a global *step* button.

More precisely, the generated luc scripts ends with the command *stdgui* that: - tries to find and load an optional resource file (see *luciolerc*) - builds the GUI - enters the tcl loop

2.1.1 Menus

- **File**
 - Save basic luc script: save the luc program in a file. The user can then customizes it, and run it directly.
 - Quit
- **Option**
 - misc. layout options
- **Clocks** contains all that concerns execution:
 - Reset the program to its initial state
 - Auto-step mode (dflt): all Boolean inputs are press buttons, whose effect is to set this input and makes a computation step (as a consequence, Boolean inputs are exclusives).
 - Compose mode: all Boolean inputs are check buttons. The user can select one or more inputs and press the global **step** button.
 - Real-time clock mode/Change period: a step is activated each period. Note that real-time is not guaranteed to be (very) accurate since it is based on

tcl/system features.

– **Tools**

- This menu is not part of the Luciole core, but a “standard” custom menu, defined in the default resource file (luciolerc.tcl).
- It is given as an example of what can be done with the resource file manipulation (see below).
- It contains calls to the external tool **Sim2chro**

2.1.2 Layout

Default layout consists in two columns, one for the inputs widgets, one for the outputs.

2.1.3 Widgets

- Default Boolean input widget: either a press or check button, depending on the global variable *compose* (see **Options**)
- Default Numerical input widget (int or real): slider
- Default Boolean output widget: text with grey (false) or red (true) background.
- Default Numerical output widget: text

2.2 Resource file (luciolerc)

The user may customize the default GUI by defining a resource file, whose name must be luciolerc.tcl. The extension outlines the fact that this file is a tcl/tk script file. A standard resource file is provided in the Lustre distribution which can be copied and modified:

```
$LUSTRE_INSTALL/lib/luciolerc.tcl
```

The resource file is automatically searched when luciole has initialized its window. The file is searched first in the current directory, then in the user home directory, and at last in the lustre distribution library:

```
./luciolerc.tcl
./luciolerc.tcl
~/luciolerc.tcl
~/luciolerc.tcl
$LUSTRE_INSTALL/lib/luciolerc.tcl
```

The first encountered file in the previous list is evaluated as it is by the luciole tcl interpret, and, as a consequence, it has (potentially) access to all internal variable defined by luciole. However it is strongly recommended to only use a few set of variables, as it is explained in the standard resource file.

2.2.1 Tools+Sim2chro

The default *luciolerc.tcl* adds a extra **Tools** that allows the call to the external tool **Sim2chro**. This tools shows the execution as timing diagrams and comes in two versions: - pure X11 - GTK2

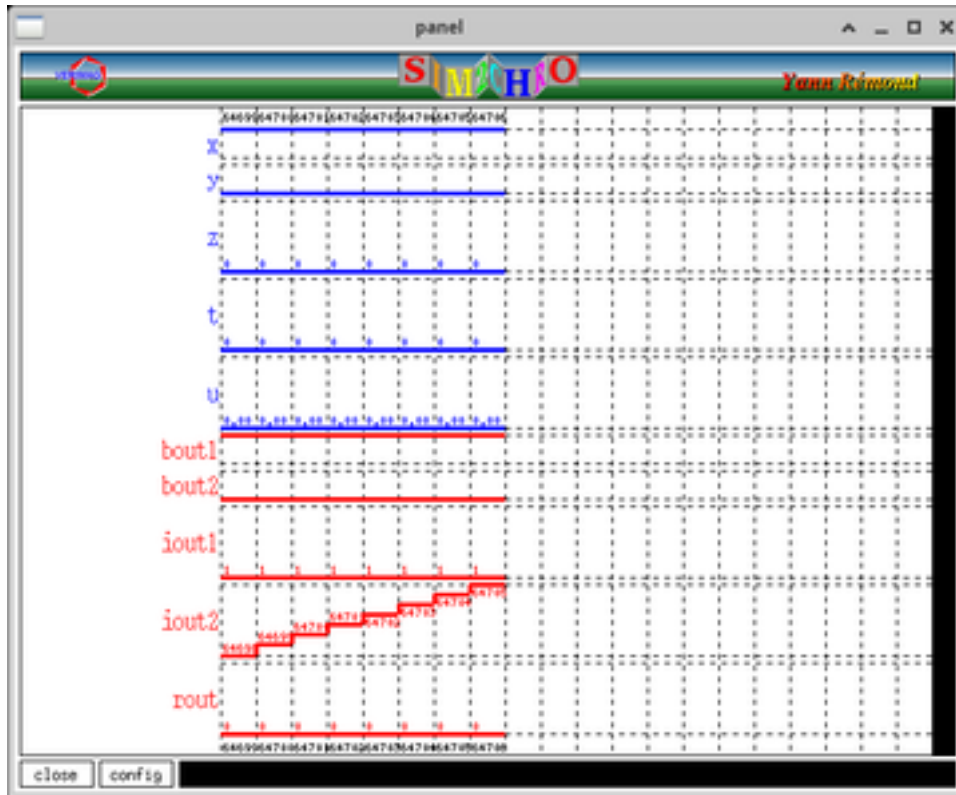


Figure 2: Sim2chro window

3 Luciole in details

The principle of Luciole v2 is to embed the reactive program simulation into a tcl script. As presented before, the `luciole` command is a script that builds such a script (using **rp2luc** tool) and run it.

The core of Luciole is a tcl package providing commands for: - loading a reactive program, and connect its inputs/outputs to tcl variable, - building the gui using (meta) widgets.

The package contains several libraries/namespaces, the most important being **luc**.

3.1 Example of luc script

Here is the content of the Lustre-expanded program `basic.ec`:

```
node basic( x : bool; y : bool; z : int; t : int; u : real;)
returns ( bout1 : bool; bout2 : bool; iout1 : int; iout2 : int; rout : real;);
let
  bout1 = x or y ;
  bout2 = false -> pre y;
  iout1 = (z + t) / 2;
  iout2 = 0 -> pre iout2 + 1;
  rout = 2.0 * u;
tel
```

Let's detail the content of the `luc` script generated by the command:

```
rp2luc ecexe basic.ec basic.luc
```

3.1.1 Header

Setup the environment, and load the library; the `namespace` command makes the use of `luc::` prefix optional:

```
#!/usr/bin/env tclsh
# generated by: rp2luc

lappend auto_path $::env(LUSTRE_INSTALL)/tcl
package require Luciole 2.0
namespace import luc::*
```

Parse the command line arguments:

```
# accepts default luciole opts
luc::parse_args
```

Increase the font (and thus all sizes), since *tk* defaults are rather small on high-resolution screens:

```
# change font magnification (%)
ezfont::scale 20
```

3.1.2 Connection to the reactive program

There are two commands to connect and load reactive programs:

- **ecexe** that takes a (textual) lustre-expanded program, and runs it using the *ecexe* library
- **droexe** that takes a (binary) dynamic library compliant with the *dynamic reactive object* interface

Both take as argument: - the name of the tcl command that will be use to interact with the reactive program (here `_panel_`) warning: beware of shadowing existing tcl commands, e.g. `panel` is an existing `luc` command. - the path to the reactive program (here “`basic.ec`”, meaning that it must be in the same folder. - The lists of inputs and outputs, with their identifier and type.

```
ecexe _basic_ "basic.ec" {
  { "x" "bool" }
  { "y" "bool" }
  { "z" "int" }
  { "t" "int" }
  { "u" "real" }
} {
  { "bout1" "bool" }
  { "bout2" "bool" }
  { "iout1" "int" }
  { "iout2" "int" }
  { "rout" "real" }
}
```

3.1.3 GUI and main loop

For building the GUI, Luciole provides specific widgets, builds over the `ptk` library, itself based on the standard `tk` library. For the sake of clarity, Luciole specific widgets are called *panel* (while widget refer to standard `tk` objects).

A “leaf” panel is automatically associated to each input and output, depending on their type. This default panel is stored in a variable with the same name~; e.g., `$x` refers to the default panel associated to the Boolean input `x` (button/check button), while `$t` refers to a slider.

Basic containers panels allow to organize the GUI in lines and columns:

- `col { <panel-list> }`
- `line { <panel-list> }`

The `panel` command allows to name and store a panel:

- `panel <name> <panel>`
- the option `-text <text>` displays the text in the top of the panel

The **main** command is `stdgui` :

- embeds the panel into the standard simulation window (with all menus and extar buttons)
- runs the `tcl` loop

```
panel inputs col -text "Inputs" {
    $x $y $z $t $u
}
```

```
panel outputs col -text "Outputs" {
    $bout1 $bout2 $iout1 $iout2 $rout
}
```

```
stdgui line {
    $inputs
    $outputs
}
```

Note that using intermediate panels is not mandatory, the following code is completely equivalent:

```
stdgui line {
    col -text "Inputs" { $x $y $z $t $u }
    col -text "Outputs" { $bout1 $bout2 $iout1 $iout2 $rout }
}
```

3.2 Customizing the GUI

As mentionned before, Luc(iole) is based on `ptk`, itself based on `tk`, there are several levels of customization:

- light: using only the **luc** features,
- deep: using **luc** and **ptk** features,
- open bar: mixing all levels including `tk`.

3.2.1 Basic layout

Luc containers allow to organize the GUI in lines and columns:

- line [-text <string>] { <panel-list> }
- col [-text <string>] { <panel-list> }

Horizontal and vertical bars are similar to lines and columns, except that they only expand in one direction when the windows s resized:

- hbar [-text <string>] { <panel-list> }
- vbar [-text <string>] { <panel-list> }

3.2.2 Leaves

- inbool <ident> [-text <string>] (default widget for Boolean input)
- outbool <ident> [-text <string>] (default widget for Boolean output)
- outvar <ident> [-text <string>] (default widget for other types)

Default numerical inputs widgets have specific options:

- intscale <ident> [opts]
- realscale <ident> [opts]

where (recommanded) options are:

- -label <string>
- -from <num>, -to <num> (min and max value)
- -resolution <num> (increment)

Luc provides also a “neutral” box leaf, whose goal is just to occupy some space:

- box [-text <string>]

3.2.3 Usage of tk’s standard options

In addition to the recommanded options presented before, any valid *tk* option can be passed to the *luc* panels. These options:

- modify the look/behaviour of the panel,
- their names and possible values are the same as in the tk library, (e.g. -background yellow), thus a certain knowledge of tk may help,
- they are passed whitout verification (at that time, may change in future versions), thus it is **strongly** recommanded to only use the arguments presented in this section.

These are the options that (should) apply to all *luc* panels:

- -bg, -background
- -fg, -foreground
- -activebackground
- -activeforeground
- -bd, -borderwidth
- -cursor

See tcltk documentation to more details on the arguments of these options:

<https://www.tcl-lang.org/man/tcl8.6/>

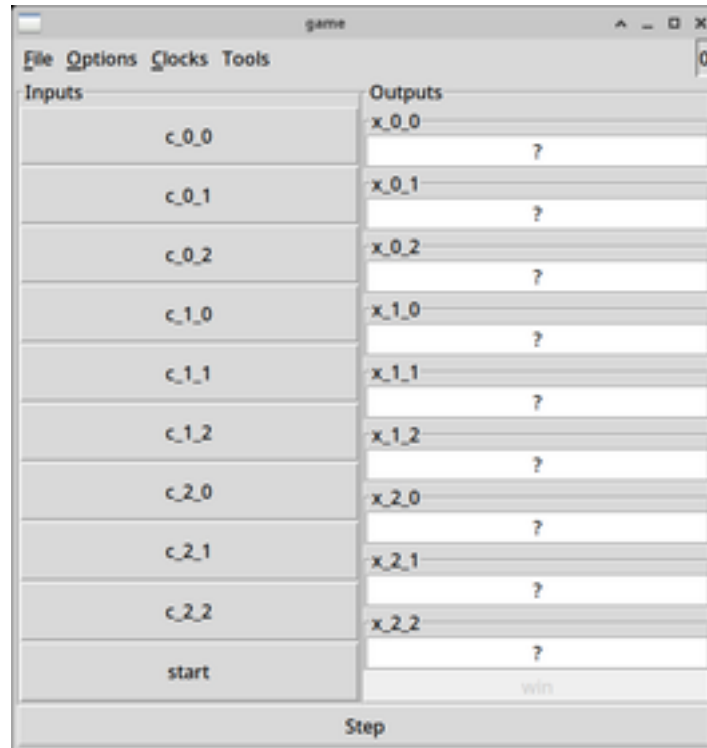


Figure 3: Rotaquin default luciole GUI

4 Ptk package

Ptk is a library to ease the programming of GUI's with tcltk. It is used by, but not specific to luciole.

See Ptk manual for details.

5 Examples and demos

They are stored in the `examples` folder of the distribution.

5.1 basic

Contains the example used in this manual/

5.2 rotaquin

This is a simple puzzle game, whose “logic” is programmed in Lustre and graphical interface made with Luciole facilities.

It illustrates several levels of possible customization:

- Basic interface;
- Less basic, with modified lines/columns layout;
- Advanced customization, requiring (p)tk programming;

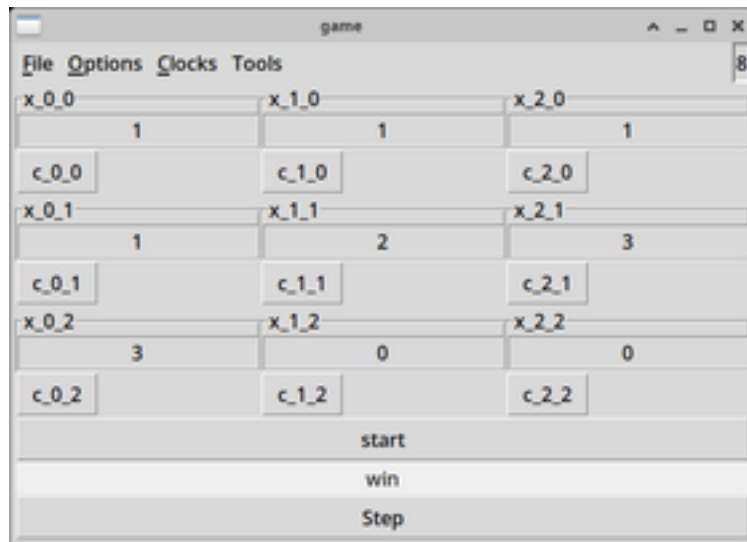


Figure 4: Rotaquin less basic GUI, using luc features



Figure 5: Rotaquin advanced GUI (starting state)



Figure 6: Rotaquin advanced GUI (winning state)