LUSTRE-V4 manual (draft)

Pascal RAYMOND

March, 2008

This document presents the tools associated to the language LUSTRE_V4. The language itself is not presented here, see *A Tutorial of Lustre* [1] for an introduction to the language, or one of the following references for a more exhaustive presentation : [7, 11, 30]. The Lustre distribution is a set of tools dealing with several formats; some of them are executables, while others are just scripts and shortcuts. The first part gives an overview of this set. Each tool is then presented in details in its own section. A section is also dedicated to the useful related tools that are not included in the distribution. The last section gives the answers to the most frequently asked questions.

[1] available in the lustre distribution

Contents

1 lustre

lus2ec, ecexe, luciole, simec, lus2oc, ec2oc, ocmin, lus2atg, oc2atg, ec2c, poc, lux, lesar, ecverif, xlesar - lustre v4 tools

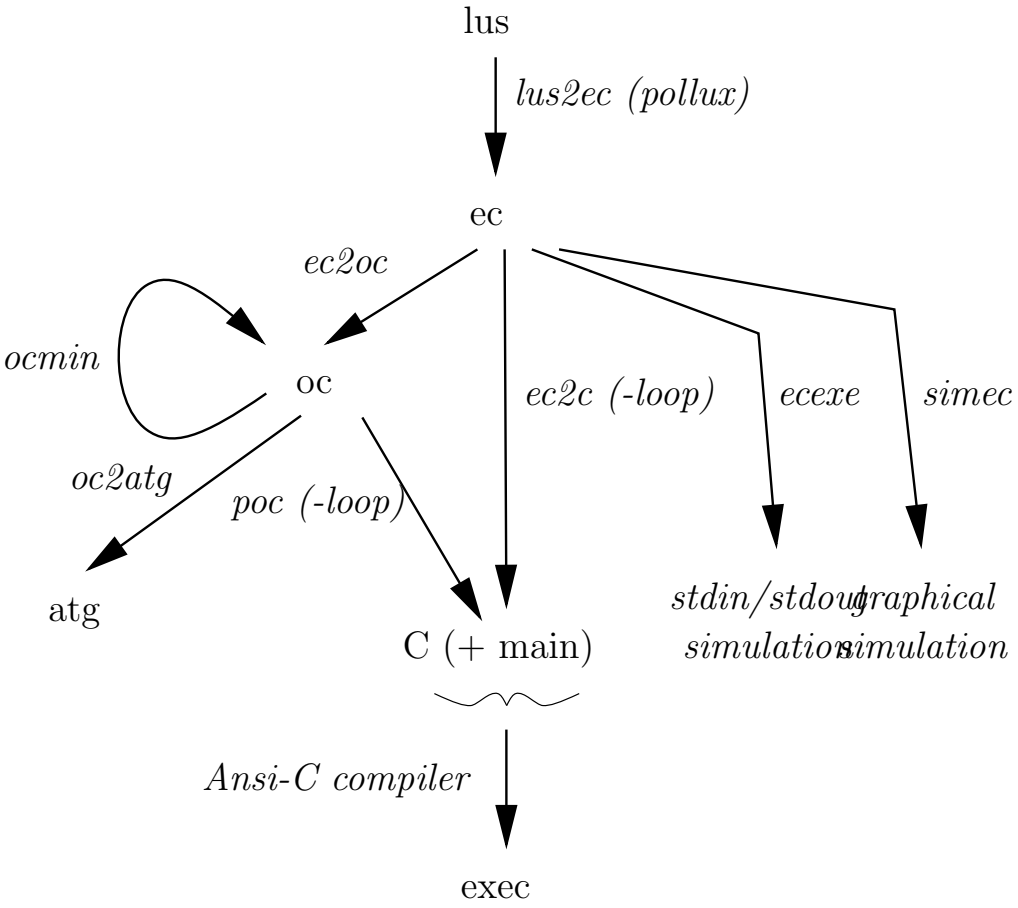

Figure 1: Formats and tools overview

DESCRIPTION

Lustre and ec

The front-end for Lustre-V4 tools is the pre-processor **lus2ec** [2]

This compiler transforms a Lustre-V4 program (**.lus** file, with modularity, arrays, recursion) into a Lustre-expended-code program (**.ec** file, with a

[2] This program is based on `pollux`, by F. Rocheteau, a lustre-to-circuit compiler which is no longer maintained.

single node, no arrays, no recursion).
All other tools (compilers, simulators ...) are actually running on the **.ec** format, but the distribution provides (in general) shell scripts combining the front-end (**lus2ec**) with the various back-ends (compilers and simulators).

Simulation

The Lustre-V4 distribution provides simulation tools that interpret **ec** code. They only run on *basic programs*, that do not require external types, constants or functions; however, some classical functions are supported, corresponding to the *math* C library. All those tools are based on the same interpreter, and only differ on the user interface:

File to file simulation:

> **ecexe** is a *unix-filter like* tool, reading on standard input and writing to standard output.

Graphical simulation:

> **xecexe** (script **xsimlus**) provides a graphical interface to the **ec** interpreter; this tool, based on X-intrinsics and Athena widgets is quite old and no longer maintained.
>
> **simec** (script **luciole**) provides a more friendly interface, based on *tcl-tk widgets*. Moreover, it allows the user to (slightly) customize the graphical interfaces.

Automata generation

Originally, the lustre compiler was designed to use an intermediate format, called **oc** (for object code). This format was initiated by a collaboration with the Esterel team, and several releases where defined. The main characteristic of this format is that the control structure consists of a finite state automaton. The tool **ec2oc** (script **lus2ec** or **lustre**) supports **oc** version 2 (**oc2**) and 5 (**oc5**). It also provides lot of options that allow the user to choose the automaton stucture of the generated code.
Some tools based on the **oc** format are provided by the Esterel team, in particular C and Ada code generators (**occ**, **ocada**). An alternative Ansi-C code generator, **poc**, is provided within the lustre distribution.
The distribution also provides a tool that performs minimization of oc automata (**ocmin**) and a translator to the *autograph* format (**oc2atg**).

C code generation

The low-level target format in Lustre-V4 is Ansi-C. This code can be obtained either:

- via **ec2oc**, using the **poc** compiler,
- directly from the **ec** code, using the compiler **ec2c**.

Note that the code generated by **ec2c** is different from the one generated by **ec2oc**; in particular **ec2c** does not build any kind of automaton. On the contrary the generated interface is the same for both compiler, so one can (normally) easily swap between code generated by **poc** and code generated by **ec2c**.
By default, those compilers only provide a *transition function*, and the user has to write his (her) own input/output and main procedures. Moreover, the user has to provide the implementation of all external objects (types, constants and functions) declared in the Lustre source.
Nevertheless, both **poc** and **ec2c** have an option **-loop** that builds an additional main procedure. This simple "loop" works as a unix filter (just like **ecexe**). In order to obtain an executable, this main program may be completed by the implementation of the (possible) external objects. Anyway, it can be used as a pattern for more complex application.
If the code does not require external objects, the main generated by the **-loop** option can be linked "as it is" with the transition function, in order to build a stand-alone application. This is the case for almost all programs that can be simulated using **ecexe**; by the way, the execution of such a stand-alone program is completly similar to a simulation with **ecexe**: standard input to standard output, interactive when called form a terminal.
The script **lux** is the best way for quickly building a stand-alone application: it can take either **lus**, **ec** or **oc** files as input, and uses the most suitable compilers calls and build, if possible, a sstand-alone application. Note that C compilation and the link-editing are performed by the host Ansi-C compiler; by default, **lux** calls the GNU C compiler **gcc**, but one may customize this script.

Formal verification

The Lustre model-checker is **ecverif** (shell **lesar**). It provides several algorithms to check the validity of safety properties on Lustre programs. **lesar** (resp. **ecverif**) takes as input special **lustre** (resp. **ec**) programs, called *verification programs*. Roughly speaking, such a program must be the parallel product of a program to validate, a program *observing* that the desired property is satisfied, and another one *observing* that the hypothesis on the environment are satisfied. The user may build this product himself, and at last, **lesar** only checks that its input has a single Boolean output, wich is supposed to be the property to check; the hypothesis are supposed to be the conjunction of all the assertions appearing in the verification program.
xlesar is a graphical interface to **lesar**/**ecverif**. This tool is particularly suitable for managing a set of verifications on the same Lustre programs.

2 lus2ec

Lustre expansion

SYNOPSIS

lus2ec *file*.**lus** *node* [**options**]

DESCRIPTION

Lustre to expanded-code compiler: it requires a Lustre-V4 input file and a main node name. It produces a file *node*.**ec**. This file contains the code of the main node where node calls are inlined (recursively replaced by their definitions) and where structured variables (arrays, tuples) are expanded into sets of atomic variables (**bool**, **int**, **real** or external type).

OPTIONS

-o *file*.ec

define the name of the output file.

-nos

output the result on *stdout*

3 ecexe

Lustre expanded code simulation

SYNOPSIS

ecexe *file*.**ec** [**options**]

DESCRIPTION

This tool takes a *stand-alone* ec program, i.e. a program that does not require external constants and functions (see below for details). The predefined types are supported; more precisely, Booleans and integers are implemented by the machine type **int**, and reals are implemented by **double** values. It simulates the reactive behavior of the program, reading input values on **stdin**, and writing outputs on **stdout**.

Stand-alone ec program

Basically, an ec node is said to be *stand-alone* if it only deals with predefined types (**bool**, **int**, **real**), and does not require any external function or constant. This is the general rule, but however, there exist several exceptions:

- External types are interpreted as enumerated types. The only values of this type are supposed to be the declared constants of this type. Polymorphic operators are supported (`=`, `->`, `if then else`, `pre`, `when`, `current`).
- Usual double-precision functions are supported; their names are those of the standard C header **math.h**, and, indeed, they are implemented by the corresponding function in **libm.a** library (see below for details).

Supported mathematical functions

Those functions must be declared in the lustre source with their exact names and parameter types. A file **luslib/math.lus** is provided in the distribution; it contains the following declarations:

```
function acos  (x: real) returns (y: real);
function asin  (x: real) returns (y: real);
function atan  (x: real) returns (y: real);
function cos   (x: real) returns (y: real);
function sin   (x: real) returns (y: real);
function tan   (x: real) returns (y: real);
function cosh  (x: real) returns (y: real);
function sinh  (x: real) returns (y: real);
function tanh  (x: real) returns (y: real);
function exp   (x: real) returns (y: real);
function log   (x: real) returns (y: real);
function log10 (x: real) returns (y: real);
function pow   (x: real) returns (y: real);
function sqrt  (x: real) returns (y: real);
function fabs  (x: real) returns (y: real);
function ceil  (x: real) returns (y: real);
function floor (x: real) returns (y: real);
```

Reactive Input Format (rif)

At each step, the interpretor reads on **stdin** a value for each input. The input flow is supposed to follow the **rif** (Reactive Input Format) conventions: the input flow consists of a sequence of basic values (Booleans, integers,

reals, strings) separated by spaces (predicate **ispace** from **ctype.h** library). Moreover, comments and pragmas can be written in the input flow:

- All characters comprised between a `#` and the next new-line are considered as a comment (resp. a pragma), and ignored (resp. treated if the pragma is supported).
- All characters comprised between `#@` and the next `@#` is also considered as a comment.

The only pragma supported by **ecexe** is the string **reset**, which causes the interpreter to restart in its initial state.
The syntax for integer and real values is the ansi-C one.
For Boolean values, the interpreter accepts:

- **0**, **f** or **F** for false,
- **1**, **t** or **T** for true.

When the interpreter has read all input values, it performs a computation step, writes the corresponding outputs on **stdout**, reads a new input vector and so on.
For instance, there is a valid input file for a node that takes one Boolean input and two integer inputs:

```
t 0 2
#this line is ignored
f 5 6 t 56 -12 #two steps on a single line
#@
all
those
characters are
ignored @# f 42 #this end of line is ignored
-10  #this is the end of the previous step
#reset
#@ the interpreter has been restarted @#
t 5 6
```

Note that new-lines have no special meaning: a new step is performed as soon as a whole input vector is available.

Missing values

In order to interpret programs whose inputs are not always defined (clocked inputs) the value **?** is accepted for any type. For instance, the following input sequence is correct for a node whose header is
node CLOCKED(c : bool; (x : real) when c):

```
t 42e2
f ?
f ?
t 24  #integer notation is accepted for real values
t -3.14
f ?
f 22.2E-10 #this value will be ignored anyway...
```

The symbol **?** is also used for undefined outputs.

Nil value

When some output takes the value of an uninitialized variable, the interpreter normally stops with error code 1. But there is an option that inhibits this feature: in this case the interpreter outputs the string **nil** and goes on.

Reset

All comments begining with the string **reset**, are interpreted as a reset command: the interpretation restarts from the begining, just as if a new process had been called for the remaining input file.

OPTIONS

-r (reactive)

inhibits all buffer mechanism on files (to be used with command pipes).

-n

outputs the string **nil** for non initialized value, instead of exiting with error code **1**.

4 luciole, simec

Lustre graphical simulation

SYNOPSIS

luciole *file*.**lus node** [**options**]
luciole *file*.**ec** [**options**]
simec *file*.**ec** [**options**]

DESCRIPTION

The Lustre graphical simulation is based on the same library than the file-to-file simulator **ecexe**. The main tool is **simec**, based on tcl-tk for the graphical aspects.
When called with a lustre file name, the script **luciole** calls **lus2ec** and then **simec** with the remaining options (otherwise it behaves like **simec**).

Main window

simec opens a main window containing a widget for each input and each output of the ec program. Input widgets allow the user to set the input values, while output widgets just show the current values of the program output. Basically, Boolean inputs are implemented by buttons, while numerical inputs are implemented by "scale widgets".

Boolean mode: auto step vs compose

The behavior of Boolean inputs depend on the *mode*:

- In *auto-step* mode, Boolean inputs are supposed to be exclusive, and a step is performed as soon as an input button is pressed.
- In *compose* mode, Boolean inputs are displayed as "check buttons": the user may select/deselect Boolean inputs without performing a computation step.

Whatever is the current mode, the pre-defined button **Step** allows the user to provoque a computation step.
The current mode can be changed via the **Clocks** menu.

Input/output layout

Input/output widgets are organized into lines and columns. The tool normally opens a panel with a column for the inputs, and a column for the outputs, but the user may define a customized layout by defining an associated *Input Output Panel* file (**iop** extension). When called with *foo.ec*, **simec** first searches for a file whose name is *foo.iop*, and tries to use it as a layout description.
Since no specific tool is available, the best way to customize the input/output panel for a program *file.ec* is to first get the default *iop* description by using the **save file.iop** command in the **Files** menu, and then edit the resulting file.

OPTIONS

-h

(help) display available options.

-v

set verbose mode: every reaction is echoed on standard output. The output format follows the *rif* conventions; as a consequence the outputed text can be used as a correct Reactive Input Flow description by tools that support this format (cf. **sim2chro**).

-auto

start in *auto step* mode.

-comp

start in *compose* mode.

-p *file*.iop

(panel) specify a particular layout description.

IOP FORMAT

Lexical aspects

- Comments: all characters between "`//`" and the end of the line are ignored; all characters between "`/*`" and the following "`*/`" are ignored.
- *number* is an integer or real notation.
- *ident* means any string of alphanumeric characters, including *number*.
- *string* means any string of printable characters enclosed whithin quotes.

Syntax

Parts whithin brackets are optional; *item-list* simply means one or more "space-separated" items.

iopfile ::= `module` *string*
`inputs` *var-decl-list*
`outputs` *var-decl-list*
`panels` *panel-list*

var-decl ::= *ident* `:` *type* [`label =` *string*] `;`

Additional information can be added to integer and real types.

type ::= `bool`
| `int` [`min =` *number*] [`max =` *number*]
| `real` [`min =` *number*] [`max =` *number*] [`step =` *number*]

A panel expression is built with the n-ary operators **line** and **col**. Leaf expressions are references to input, output or panel identifiers. The leaf **box** has no meaning: it simply "takes place" in the layout. An identifier must be declared before it can be used. The panel **top** must be the last declared.

panel-decl	::=	*ident* `=` *panel-exp* `;`
panel-exp	::=	`col` { *panel-exp-list*}
	|	`line` { *panel-exp-list*}
	|	`$`*ident*
	|	`box`

THE RESSOURCE FILE

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

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

The ressource 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 ressource file. The most useful variables are:

Global(verbose)

(read/write) holds a Boolean value (**1** or **0**) indicating wheter the verbose mode is set or not.

Global(verbose_channel)

(read/write) holds a tcl channel identifier (initially **stdout**) indicating where to put messages in verbose mode.

Global(show_step_ctr)

(read/write) is a Boolean indicating if the step counter is shown (**1**) or not (**0**).

Global(show_step)

(read/write) is a Boolean indicating if the step button is shown (**1**) or not (**0**).

Global(auto_step)

(read/write) is a Boolean indicating if luciole runs in *auto-step mode* (**1**) or in *compose mode* (**0**).

Note that some command line options (**-v**, **-auto**, **-comp**) may override commands in the ressource file.
Informations on the current program are also available; *those variables may not be modified*:

Global(module_name)

(read only) is the name of the running lustre node (string).

Global(input_names)

(read only) is the list of input names (string list).

Global(input_types)

(read only) is the list of input types (string list).

Global(output_names)

(read only) is the list of output names (string list).

Global(output_types)

(read only) is the list of output types (string list).

At last, a tk container widget (i.e. a *frame*) is reserved in the lurette window for user's customization. Most preciselly, this widget (initially empty) is located in the luciole menubar, and its typical use is to add one or more user-defined menu buttons

Global(user_menu)

(read only) contains the tk-path of a frame where the user can pack his/her own menus.

The standard ressource file is an example of how to create such a menu: it adds a menu button **Tools**, with a command **sim2chro** that dynamically launches the chronogram manager *sim2chro*.

ENVIRONMENT

The environement variable **LUSTRE_INSTALL** must exist and hold the path of the lustre v4 distribution.

5 lus2oc, ec2oc

Lustre to Lustre/Esterel Object Code

SYNOPSIS

lus2oc *file*.**lus node** [**options**]
ec2oc *file*.**ec** [**options**]

DESCRIPTION

The **lus2oc** command requires a file and a main node; it first calls **lus2ec** to build the corresponding expanded code, then calls **ec2oc** with all remaining options.
The **ec2oc** command builds a sequential program structured as an automaton. Several options allow the user to adjust the automaton generation. The default target format is *oc version 5*, but the version 2 is still supported, for compatibility with old tools. The default is to write the result in the file "`node.oc`".

OPTIONS

Miscellaneous

-v

set the verbose mode.

-help

print available options.

-default

print default options.

-o *file*.oc

define the output file.

-pure

implement Boolean inputs/outputs with "pure" signal (i.e. side-effects). The default is to implement Boolean inputs/outputs with "classic" variables, just like integer or reals.

-double

implement the lustre type **real** with the target type **double** (the default target type is **float**).

Static optimization

-min

(default) the initial operator network is minimized before code generation. This minimization is a kind of "common sub-expression checking" except that the network may contain cycles; as a consequence this phase can be expensive, so the user can disable it (cf. **-nomin**).

-nomin

disable the minimization of the operator network (cf. **-min**).

-const

perform static computation of constant expressions.

Output format

-oc2

generate an *oc V2* file.

-oc5

(default) generate an *oc V5* file.

Automaton

The main control structure is an automaton, obtained by static simulation of the Boolean variables (called state variables). The user can select the set of state variables:

-0

no state variable; the resulting automaton has only one state. This mode is not supported if the source program contains clocks.

-1

only clocks are simulated. For single-clocked program, the resulting automaton has always 2 states (the initial one and the non-initial one).

-2

(default) all Boolean variables are simulated; the resulting program is the "fastest" code that can be generated, but unfortunately its size may be exponentially greater that the source code one. It it advisable

to use the **-v** option, in order to stop the compilation if the automaton size grows too much.

The user may choose between two algorithms for the generation:

-data

(default) the data-driven algorithm simulates the Boolean variables without taking into account their actual influence on the outputs. The resulting automaton is in general non-minimal according to the output computation.

-states *integer*

works in data-driven mode. The generation stops after the specified number of states (default 10000). This limit avoids the generation of too huge code.

-demand

the demand-driven algorithm builds a automaton just "big" enough to perform the computation of the outputs. Using this option is quite equivalent to perform a standard minimization on the automaton produced by the data-driven algorithm.

The user can select the way conditional statements are treated:

-S1

forbid code duplication, i.e. tests are closed as soon as possible.

-S2

(default) use an heuristic for opening/closing tests; it forbids duplication of "big" parts of code.

-S3

forbid test duplication, i.e. tests are closed only when they are no longer needed.

-S4

forbid test closing; this option is the worst one in term of code size, but it can be very useful for analysing the program (see in particular **lus2atg**).

Bdds

The bdd (binary decision diagrams) library is one of the most critical (time and space cost) part of the compiler. The user may change some parameters in order to improve compilation time.

-merge

perform a "clever" variable ordering before building bdds; this option is sometimes useful when the state generation step cannot even start.

-bddpage *integer*

the space devoted to bdds is allocated by pages. Before allocating new pages, the program first performs garbage collection; for big programs, frequent garbage collection may dramatically slow down the compilation, so the user may increase the size of pages, expressed in kilo-unit (default 10).

6 ocmin

Automata minimization

SYNOPSIS

ocmin *file*.**oc** [**options**]

DESCRIPTION

ocmin

This tool performs state equivalence checking to reduce the size of the automaton. It takes either **oc2** or **oc5** files, and outputs the minimized automaton in the same format version; the default output file is *module-name*_**min.oc**.

OPTIONS

-v

set verbose mode

-o *file*

define the output file (default is *module-name*_**min.oc**)

7 lus2atg, oc2atg

Automata visualisation

SYNOPSIS

lus2atg *file*.**lus** *node* [**options**]
oc2atg *file*.**oc** [**options**]

DESCRIPTION

oc2atg extracts basic information about the underlying automaton in the source file, and outputs this information in the **atg** format, suitable for the tool *autograph*;
Most of the information is lost: the **atg** file only contains the skeleton of the program. This skeleton consists of a set of states and transitions labelled by "input/output" presence conditions.
It is quite hard to compile a lustre program in such a way that the resulting **oc** program produces a meaningful **atg** automaton, so it is recommended to use the script **lus2atg**. Using this script, the Boolean part of the source program will be exactly reflected in the resulting automaton.

OPTIONS

Miscellaneous

-v set the verbose mode.

-help print available options.

-o *file* define the output file (default is *module-name*.atg)

Labels style

Labels are representing the input/output presence on a transition. Note that this information is relevant only if the oc program uses *pure signals* (see **ec2oc**). The *lus2atg* script automatically produces such a program. There are actually two styles for the labels: reactive machine style, and Boolean function style.

default (reactive machine style)

For the inputs, the default is to write **+inname** if the input is present, **-inname** if it is absent and **~inname** for it is don't care.

For the output, only presence is represented by writing **!outname** (which means that the output is emited during the transition).

Exemple: **+x+y-z~t/!a!b** means that, for the transition to be taken, inputs **x** and **y** have to be true (i.e. present), **z** has to be false (i.e. absent), **t** and any other input (if it exist) can either true or false (don't care); as a consequence outputs **a** and **b** are true (emited), and any other output is false.

-bf (Boolean function style)

The label is made of the list of the input values (0 or 1) and the list of the corresponding outpt values, separated by a /. For instance, if the program has 2 inputs **x**, **y** and 1 output **s**, **10/1** means that the transition is taken when **x** is true, **y** is false, and that the output **s** is true.

Partial labels

Parts of the labels can be hiden.

-ht hides the don't care inputs

-hp hides the true inputs

-hm hides the false inputs

-hi hides all the input part

-ho hides all the output part

-h hides the / separator between inputs and outputs

8 ec2c, poc

Ansi C code generator

SYNOPSIS

ec2c *file*.**ec** [**options**]
poc *file*.**oc** [**options**]

DESCRIPTION

These tools are Ansi-C code generators. They are presented together, since they share the same conventions for the generated user interface. Otherwise, they are completely separated tools, working in different formats, and performing different tasks:

- **poc** takes as input a **oc** file, which is already a sequential imperative program; the work of **poc** is then a simple translation between similar formalisms.
- **ec2c** compiles an **ec** program into Ansi-C code. Since **ec** (which is a subset of Lustre) is a declarative language, a lot of work remains to do for building sequential code. In this sense, **ec2c** is closer (but

anyway simpler) to a tool like **ec2oc**. Roughly speaking, using **ec2c** is almost equivalent to using **ec2oc** with the **-0** option, and then **poc**. The main characteristic of this compiler is that both the compilation time and the size of the code are linear with respect to the size of the source code. Moreover the compilation algorithm strictly follows the Lustre formal semantics, just like the simulator **ecexe** does.

The consequence is that, on one hand, the code is not very efficient (no control structures, lots of intermediate variables), but in the other hand, it is very safe.

TARGET CODE

The code generated by those compilers consists essentially in a procedure implementing a step of the reactive program described in the source file. In order to run the reactive program, the user must write a main loop around the "step" procedure.
In the following, we precise what it is automatically generated, and what the user has to write. Let us call *foo.oc* (resp. *foo.ec*) the source file, *FOO* the **oc** module (resp. the **ec** node) defined in the source file, *foo.c* and *foo.h* the generated files, and *loop.c* the main program written by the user.

Execution context

The generated code allows multiple allocations of a reactive module. The memory needed for an instance of the reactive module is defined in *foo.c*, and declared in *foo.h*:

```
struct FOO_ctx;
```

The user does not have to know what this structure is made of, he is only allowed to manipulate pointers. The user can get a new context using a procedure declared in *foo.h*:

```
struct FOO_ctx * FOO_new_ctx(void* client_data);
```

The user can associate an extra information to a new execution context using the `client_data` argument. This information is necessary if the user wants to run several instances of a same reactive module concurrently.

Step procedure

The procedure implementing a step of the reactive module is declared in *foo.h*:

```
void FOO_step(struct FOO_ctx * ctx);
```

This procedure is called with an execution context previously created by a call to `FOO_new_ctx`. This step procedure has no input/output parameters, since communication between the main loop and the reactive module is made via input/output procedures. More precisely, the user must call input procedures to set the input values before he calls the step procedure. The step procedure calls output procedures to send its outputs to the environment.

Inputs and outputs

Communications between *foo.c* and *loop.c* are made via input and output procedures. The input procedures are defined in *foo.c* and used in the main loop, the output procedures are defined by the user (in *loop.c* for instance), and used in *foo.c*.
For each input **IN**, of type **TYP**, *foo.c* contains the definition of the procedure:

```
void FOO_I_IN(struct FOO_ctx* context, TYP value);
```

Note that if **IN** is a pure signal (**poc** only), the *value* parameter is omitted:

```
void FOO_I_IN(struct FOO_ctx* context);
```

For each output **OUT**, of type **TYP**, the user must define a procedure:

```
void FOO_O_OUT(void* client_data, TYP value);
```

Note that if **OUT** is a pure signal (**poc** only), the *value* parameter is omitted:

```
void FOO_O_OUT(void* client_data);
```

Output procedures are called within **FOO_step(FOO_ctx* ctx)**, using the client data which has been associated with **ctx** when it was created.

Example

Here is a simple example of a main loop using a reactive module (whose name is `sum`) with two real inputs (`x` and `y`) and a single real output `s`; note that the *client data* is not necessary, since the loop uses only one instance of the reactive module:

```
#include <stdlib.h>
#include "sum.h"

void sum_O_s(void* cdata, float _V){
   printf("result: %f\n", _V);
}
```

```
main(){
   _float x;
   _float y;
   struct sum_ctx* prg = sum_new_ctx(NULL);

   while(1){
      printf("(float) x ?\n");
      scanf("%f", &x);
      sum_I_x(prg, x);
      printf("(float) y ?\n");
      scanf("%f", &y);
      sum_I_y(prg, y);
      sum_step(prg);
   }
}
```

External objects

Each external object declared in the source file is supposed to be implemented by the user. Some information is necessary for the compilation of the **c** generated code, other is necessary only for linking.

- **Compiling** The *foo.c* program generated by **poc** cannot be compiled unless the external types are defined. **poc** supposes that those definitions are in a file called *foo_ext.h*.
- **Linking** External constants, functions and procedures are declared in *foo.c* as imported objects, so *foo.c* can be compiled separately. Indeed the user must define those objects somewhere, and link the corresponding code with the **poc** object code if he wants to build an executable program!

Standard main loop

When called with the **-loop** option, **poc** (resp. **ec2c**) produces an extra c-file *FOO_loop.c*. This code contains a main procedure implementing a loop which reads inputs on **stdin** and write outputs to **stdout**. If *foo.c* does not need external object, it is a simple way to obtain executable code. For instance:

```
poc sum.oc -loop
gcc sum.c sum_loop.c -o sum
```

produces an interactive program **sum**, which allows the user to test his code. If external objects are needed, the user must write all the necessary code plus two procedures for each external type **TYP**:

- `TYP _get_TYP(char* name)` reads a value of type **TYP** on **stdin**, and returns it. The argument "`name`" is the name of the input, used to make the procedure more "interactive".
- `void _put_TYP(TYP val)` prints the value "`val`" on **stdout**.

Pragmas

In order to allow automatic manipulation of the code generated, the compiler generates special comments (pragmas) in the header file. Pragmas are single line Ansi C comments, begining with the string **poc:**. Ther is a pragma which gives the name of the module, and a pragma for each input and each output, giving its type and its name. Here is an example of pragma section:

```
//poc:MODULE sum
//poc:IN _float x
//poc:IN _float y
//poc:OUT _float s
```

OPTIONS

-v

set the verbose mode.

-o *name*

define the prefix for the target files. The default is to use the name of the module (i.e. node), which is not necessarily the name of the source file.

-loop

generate an extra main file called *name*_**loop.c**. This main is sufficient to build a stand-alone application if the reactive module does not need any external definitions.

9 lux

All in one Lustre compilation

SYNOPSIS

lux *file*.**lus** *node*
lux *file*.**ec**
lux *file*.**oc**

DESCRIPTION

This shell script performs all the compilation stages necessary to produce a binary file from a lustre (resp. **ec** or **oc**) program. A main node is expected when starting from the lustre level.
The binary generation only works for *basic* lustre programs (resp. **ec** or **oc**), that means programs that do not need any external definitions to work (external types, constants, functions).
The several stages are depending on the input format:

lus:

call **lus2ec**, then **ec2c** with the **-loop** option, and then the gnu C-compiler/linker **gcc**.

ec:

call **ec2c** with the **-loop** option, and then **gcc**.

oc:

calls **poc** with the **-loop**, and then **gcc**.

When the compilation succeeds, the name of the resulting program is the name of the main lustre node (resp. the ec node or the oc module).

NOTES

Indeed, Lustre distribution does not provide a C compiler. The **lux** script supposes that the gnu-C compiler is properly installed, but the user may customize the script in order to use another available compiler.

10 lesar, ecverif

Formal verification

SYNOPSIS

lesar *file*.**lus** *node* [**options**]
ecverif *file*.**ec** [**options**]

DESCRIPTION

The **lesar** command first calls **lus2ec**, then **ecverif**. The **ecverif** tool takes a program whose first output is Boolean, and try to prove that this Boolean output remains true, for any execution of the program (i.e. whatever is the infinite sequence of input values received by the program). Moreover, the tool supposes that every assertion (**assert** statements) appearing in the

program denotes an hypothesis on the program environement, so the goal of the tool is at last to prove that:
Whatever the sequence of inputs, as long as the assertions (the hypothesis) are satisfied, the first output (the property) always remains true.
The input program, often called the *verification program*, is generally built as a combination of the program to validate (the implementation) together with another program (the observer) expressing the safety property. The user may read the *Lustre Tutorial* which explains how to build verification programs suitable for **lesar/ecverif**.
ecverif is a *model-checker*: it explores a finite model (an automaton) of the program. This model is an abstraction that represents an upper-approximation of all the possible executions of the program. The abstraction made on the program is conservative: if the verification suceeds on the model, the property *is also satisfied* by the program. In this case the tool answer *True Property*. If the verification fails on the model, the result is inconclusive: either the property is not satisfied by the program, or the property is too complex for the tool. So when the tool answer *False Property*, it simply means *I don't know*!
More precisely, the Boolean part of the program is completly reflected in the model, but everything else is abstracted (numerical variables, external types and functions ...). Note that, as a consequence of this, the model-checking is complete for purely Boolean programs (e.g. logical circuits).
Nevertheless, some knowledge on numerical properties has been added to the model checker. A library based on *polyhedra manipulation* can be used to check whether linear constraints are feasible or not. For instance, using the polyhedra library, the tool will realise that the condition "`(y < x) and (0 < e) and (x + 2*e < y)`" cannot be satisfied by any numerical values. This information is then taken into account to obtain a more precise model, but note that the verification still remains partial in general.
The successive stages of the computation are involving complex algorithms that may take (a lot of) time. It is recomended to use the verbose mode: in this case **ecverif** outputs information on the current stage, that allows the user to "guess" if the computation has a chance to end in a reasonable delay. The stages are:

Static analysis

The source code is optimized for the proof: this stage performs dependence checking, syntactic minimisation, and other optimization at the source level. The complexity here is reasonable (linear or quadratic).

Bdds construction

The Boolean part of the program is identified and transformed in a set of logical functions, represented by bdds (Binary Decision Diagrams).

The result is an implicit representation of the model to check. This stage may be exponential in the worst case.

Exploration of the model

Several algorithms can be selected. In all cases, the time necessary for the traversal of the model can be exponential in the worst case.

OPTIONS

Miscellaneous

-v

set the verbose mode. Since the algorithms used in the tool are very expensive, it is strongly recomended to use this option in order to have a feed-back on the verification progress.

-help

print available options.

Static analysis

-nomin

Normally, the first stage of the verification consists in minimizing the source program, according to syntactic equivalence of expressions. This checking also takes into account equivalence of *recursive definitions*. The result is much more precise than a simple "common sub-expression checking", but indeed more expensive, so the user can disable it with this option.

-split

split the property into several smaller ones (if possible).

-optb

force static Boolean optimization of the source program. This option was supported by older versions, but it is (almost) obsolete since the bdds construction stage is now optimized.

Model checking

The main stage of the model checker consists in exploring the underlying automaton (the model). The user may choose between several algorithms for this exploration.
Assertions are taken into account during the exploration to throw away unfeasible transitions. A state whose all outgoing transitions have all been

discarded is said to be a *sink state*. Assertions that can produce sink states are said to be *non-causal*.

-enum

(default) use the *enumerative algorithm*. The automaton is checked state by state, starting from the initial one. The verification fails as soon as a state violating the property is reached. An error occurs if the assertions are found to be non-causal.

-states *n*

only works in enumerative mode. The exploration stops after n states are visited.

-forward

use the *symbolic forward* algorithm. The set of reachable states is build as a Boolean formula over the state variables. The verification fails if this set contains states violating the property.

-backward

use the *symbolic backward* algorithm. This algorithm builds a symbolic representation of the *bad states*, i.e. states that can lead to the violation of the property. The verification fails if the initial state belongs to this set.

-causal

compute a causal assertion equivalent to the initial one, before starting the model-checking. This computing is expensive, so it is recommended to use it only when needed (when a first attempt have failed because of non-causal assertions). Moreover, this option must not be used with the *forward algorithm*, which implements its own treatment for non-causal assertions.

-poly

force the model checker to use the *polyhedra library* to check wether linear constraints on numerical values are feasible. Without this option, semantics of numerical values is completly ignored, and any condition is supposed to be feasible as soon as integers or reals are involved.

Diagnosis

-diag

print a diagnosis when verification fails. The diagnosis is a sequence of input values that may leads to a state violating the property.

-lurettediag *n filename*

Due to abstractions applied to non-boolean programs, the counter-examples produced by the **-diag** option are not always very useful, mainly because (1) new inputs are created to the model to take into account the (now unpredicatable) non-boolean elements of the program, and (2) as already explained, abstraction may lead to false negatives: counter-examples, which have no counterpart in the concrete model. In general, associating a long and complex abstract counter-example with a concrete one is a long and tedious exercise. This option generates richer output to the file called *filename*, which, in turn, can be fed into the tool **lurette** to try and produce concrete counter-examples automatically.

The **lurette** file contains three Lustre nodes *node_main*, *node_assertion* and *node_oracle* (where *node* is the name of the main node in the verified program), which are to be given to **lurette** as the main, assertion and oracle node respectively. The length of the counter-example produced is reported by **lesar**. The first (numeric) parameter specifies the minimum length counter-example to produce. Typically, this is set to 0, for the shortest counter-examples, but sometimes the user may wish to force **lesar** to produce longer counter-examples.

This option automatically enforces **-forward**.

Bdds

-merge

perform a "clever" variable ordering before building bdds; this option is sometimes useful when the state generation pahse cannot even start.

-bddpage *n*

the space devoted to bdds is allocated by pages. Before allocating new pages, the program first performs garbage collection; for big programs, frequent garbage collection may dramatically slow down the model-checking, so the user may increase the size of pages, expressed in kilo-unit (default 10).

NOTES

Building verification programs to "feed" **lesar** is quite hard. The graphical proof manager **xlesar** can be a simpler way for starting with formal verification.

11 xlesar

Graphical proof manager for lustre

SYNOPSIS

xlesar

DESCRIPTION

This tool is a graphical interface to the lustre verification tool **ecverif**.
More precisely, this tool allows the user to manage a set of properties for a given lustre program. Each property can be edited, annotated, completed with local assumptions. A window allows the user to select the desired options and to launch the prover **ecexe**.

Main window

The main window allows the user to select a main program and a main node within this program. Once selected, the lists of inputs and outputs are displayed and the user can define properties involving those variables.
The property manager is part of the main window: the user can create, delete and edit properties.

Property editor

In the main window, one must select a property (by clicking in the properties list) and launch a property editor by clicking **Edit**. In the property editor window, the user can:

- edit the property name
- define the property, by typing a correct Lutre Boolean expression (see below for details).
- declare a local input (menu **Edit**, field **New input**). A local input behaves as a "random" value in the proofs.
- declare and define a local variable (menu **Edit**, field **New variable**).
- define a local assertion (menu **Edit**, field **New assertion**). The assertion must be a correct Boolean lustre expression.
- write a description of the property. The command **Show description** from the **Edit** menu opens a text editor where the user can type his description.

Correct lustre expressions

Each lustre expression entered in **xlesar** is checked for consistency before the prover is launched. Such an expression must be syntactically correct, but **xlesar** also checks type and the identifier consistency. Every identifier appearing in the expression must exists in the current context. A valid identifier is:

- any global input or output (those from the main node, if it exists).
- any local input or variable that is currently enabled.

Imported nodes

The tool does not allow the user to define complex objects like nodes. In order to use a node call in the definition of a property (resp. assertion, local variable), the user must *import* its definition (main window menu **Import**, menu **Files**, command **Load file**). Once loaded, the imported file is parsed by xlesar to find and display all the nodes declaration. From this moment on, the user will be able to use those nodes in the property (resp. assertion or variable) definitions.

Prover launcher

A prover launcher is displayed in the property editor window. This launcher allows the user to select the options and run the proof. It also display the current status of the property (checked, not checked or unknown). When the prover is running, all messages are "echoed" in the window, and the user may kill the proof process by pushing the **Stop** button.

Saving and loading the session

A xlesar session is called a *project* (extension **.lesar**). The user may save, load or restart a project via the **Files** menu in the main window.

ENVIRONMENT

The environement variable **LUSTRE_INSTALL** must exist and hold the path of the lustre v4 distribution.

12 Related tools

12.1 OC tools

The **oc** format was designed to be a common object-code for both lustre and esterel compilers. The occ (oc to c) compiler is distributed as part of the esterel distribution. See `http://www-sop.inria.fr/meije/esterel`.

12.2 Autograph

Autograph is a graphical editor that can load automata descriptions generated by `lus2atg` (see §7). It is distributed by the MEIJE team at Sophia-Antipolis. See `http://www-sop.inria.fr/meije/verification`.

12.3 Sim2chro

This tool converts flows of values into a "viewable" chronogram. The input format (called rif for *reactive input flow*) is compliant with the output produced by lustre simulators (`ecexe`, `luciole/simec`). So one can saves a simulation outputs into a file, and then feeds sim2chro with this file.
If the `rif` producer is non-blocking, a command pipeline can even be defined. This is for instance the case when the producer is `luciole`. For instance:

```
luciole file.lus node -v | sim2chro -ecran > /dev/null
```

starts a process where each steps of `luciole` "increases" the chronogram displayed by `sim2chro`. This feature is exploited in the standard `luciolerc` file: a menu is added in `luciole` which allows the user to dynamically call/kill a `sim2chro` process (see §4).
The default for `sim2chro` is to output the chronogram as a latex picture. The `-ecran` (french word for screen) option displays the chronogram into a X window. Much more formats are available (xfig, postscript, tcltk), and a huge amount of options are defined for customizing the result. The tool (version 3.0) is present in the lustre distribution, but unfortunately, no documentation is available for the time being. The best way to get information is then to contact the author `Yann.Remond@imag.fr`.

12.4 Reglo

Reglo is a compiler which translates regular expressions into sequential circuits, implemented as lustre nodes. This tool can be useful for describing observers in a imperative way and to automatically translate them into lustre observers suitable for the verification tools (cf. §10, §11). The tool is distributed in this distribution, see the related documentation (`doc/reglo_doc.ps`) for details.

12.5 Bdd calculator

The tool `bddc` implements an interactive processor for the evaluation of logical formula. The tool is distributed in this distribution, see the related documentation (`doc/reglo_doc.ps`) for details.

13 Frequently asked questions

13.1 Generalities

13.1.1 I've just write my first lustre program, how can I run it?

Let `foo.lus` be this file, and `bar` the node (defined in `foo.lus`) you whant to run. If this program only uses basic features (no external declarations), you can either use the graphical simulator by typing:

```
luciole foo.lus bar
```

or compile it by typing:

```
lux foo.lus bar
foo
```

13.2 Language

13.2.1 I get type checking errors when I try to use arrays

The most common errors come form the *homomorphic extension* of the operators. This means that all operator (e.g. `and`, from `bool` $\times$ `bool` to `bool`) is implicitly extended to arrays of any size (e.g. form `bool`$^n \times$`bool`^{n} to `bool`^{n}). This rule holds for the `if...then...else` operator, so for instance:

```
node bad(c : bool; A, B : int^n) returns (X : int^n);
let
   X = if c then A else B; --TYPE ERROR: c IS NOT AN ARRAY
tel
```

In order to make that, just build an array with `n` "copies" of the condition:

```
node good(c : bool; A, B : int^n) returns (X : int^n);
let
   X = if c^n then A else B; --OK
tel
```

13.3 Code generation

13.3.1 I run lus2oc (ec2oc) and nothing happens

All the tools that are based on automata generation may provoque (in the worst case) an exponential "explosion" of computation time and space. For instance, the underlying automaton of the following node has 2^{64} states, which is much more than the age of earth in seconds !

```
const n = 64;
node boom(dummy : bool) returns (end : bool);
var
   cpt : bool^n;
   carry : bool^n;
let
   cpt = false^n -> carry xor pre cpt;
   carry[0] = true;
   carry[1..n-1] = carry[0..n-2] and pre cpt[0..n-2];
   end = false -> carry[n-1] or pre end;
tel
```

First, it is recommended to use the verbose option (`-v`) while using `lus2oc`: you will see the progression of the automaton generation, then see question 13.3.2.

13.3.2 The number of states seems to grow infinitely when I run lus2oc (ec2oc)

The number of states is always finite, but maybe exponential, which is practically quite equivalent (see the example in question 13.3.1). If the number of states grows dramatically, just kill the compilation process and try a "cheaper" way:

- `lus2oc` with the option `-0` or `-1`
- `ec2c` which generates a naive (but always linear) code

13.4 Lustre/C interface

13.4.1 Well, now I get a C file form my lustre program, but where is the main procedure?

The goal of the C-code generator (eitheir `ec2c` or `poc`) is to produce a step procedure, that implements a computation step of the lustre program. In order to run this code, the user must write the main procedure that "loops" around the step procedure. Writing such a main procedure from scratch is a little bit hard, so `ec2c` (resp. `poc`) provides a `-loop` option, that generates a standard main loop procedure. This main is quite basic: it reads inputs on stdin, writes outputs on stdout. Moreover, it can only runs "as it is" for Lustre programs that do not require external definitions (imported types, constants, functions). However, this standard loop is a good starting point for designing mode complex applications.

13.4.2 What kind of C code is necessary for imported types, constants, functions?

All external objects declared in a Lustre program `bar` must be declared and/or defined in a file whose name is `bar_ext.h`. In the generated files `bar.h` and `bar.c` you will find (as a comment) the list of all objects whose definition is expected in `bar_ext.h`. There is no strong restrictions on the way external objects are actually implemented:

- external types can be any ansi-C types (records, arrays, enum or whatever). Indeed the actual size of the C type must be known at compile time, so `bar_ext.h` must contain the definition of the time, not only its declaration. Note that for each external type `FOO`, some procedures must be declared too:
  - the assignment: `void _assign_FOO(FOO*,FOO)`
  - the comparizons: `_boolean _eq_FOO(FOO,FOO)`
    `_boolean _ne_FOO(FOO,FOO)`
  - the condition: `FOO _cond_FOO(_boolean,FOO,FOO)`
- external constants can be global constants, variables or even "#"-defined constant expressions. The declaration is sufficient in `bar_ext.h`.
- external procedures must follow the expected parameter profile. In particular, external functions are (in general) supposed to be implemented by *procedures*: functions returning "void", where the first parameters are pointers to the output variables. The actual declaration has no importance since the profile is correct: external function, inlined function, or even "#"-defined function.

13.4.3 I get "syntax errors" (resp. "undefined reference") while compiling the C code

This probably means that external objects are not properly declared in the external header file (the default is `bar_ext.h` for a lustre program `bar`). See question 13.4.2.

13.4.4 Can I run concurrently several Lustre node in a C application?

Yes. The ansi-c code produced by `poc` or `ec2c` is "object-oriented". That meens that a lustre node `bar` is translated into a structured type (the context) with several functions to manage it (the methods: creation, copy, step call, input procedures etc...). See the `poc` manual for details (§8). You can even run concurrently several instances of the same node: a simple mechanism of "client data" allows the user to differentiate at run-time outputs comming from different instances of a same node.

13.4.5 My external procedure is sometimes called twice (resp. not called at all) each single step!

That may happen if you use the `lus2oc` compiler. This compiler supposes that external functions are "safe" (just like predefined operator) so the function can be called only when needed: zero, one or times in a step. In fact, only output procedures are supposed to perform side-effect: those procedures are called exactly once in each step if and only if the output clock is true.
If you use the `ec2c` compiler, things are quite different: any operator (including procedure calls) appearing in the program is executed once in each step if the corresponding clock is true.

13.5 Verification

13.5.1 I run lesar (ecverif) and nothing happens

Like lus2oc (see question 13.3.1, 13.3.2), this tool is based on automata exploration, and both computation time and size may grow exponentially. It is recommended to use the verbose option to see where the computation is "blocked":

- while `building bdds`. Try the `-merge` option, which computes a better variable ordering.
- while `exploring states`. Try another algorithm for the automaton exploration, `-forward` is the best in general, but `-backward` is sometimes surprising.

13.5.2 I wrote a property which is trivially true and lesar (ecverif) answers FALSE PROPERTY

If the property only involves Boolean variables and operators, lesar is likely to be right...
If the property involves numerical values, this answers simply meens *I don't know* : the default behavior of lesar is to ignore everything that is not purely logical. For instance lesar does not even know that "$(X \geq 2) \Rightarrow (2{*}X > 3)$". In order to "increase" the knowledge of the tool, you may use the `-poly` option. With this option, lesar uses a *polyhedra library* to "compute" linear constraints: in particular, "$(X \geq 2) \Rightarrow (2 * X > 3)$" will be replaced by "true".
In all cases, properties that involve temporal behavior of numerical variables can not be checked.

13.5.3 I replace an assertion by an equivalent implication, and lesar (ecverif) behaves differently

A common error is to think that a verification scheme such that:

```
node Proof1 (Hypothesis : bool; Property : bool) returns (ok : bool);
let
   assert Hypothesis;
   ok = Property;
tel
```

is equivalent to the following one:

```
node Proof2 (Hypothesis : bool; Property : bool) returns (ok : bool);
let
   ok = Hypothesis => Property;
tel
```

Intuitively, in the first case, lesar try to check that *if Hypothesis is always true, then Property is always true too*, while in the second one, lesar try to check that *the proposition "Hypothesis implies Property" is always true.* More formally, let's consider that Hypothesis and Property are infinite sequences of Boolean values, indexed by naturals ($H = H_0, H_1, ..., H_t, ...$ and $P = P_0, P_1, ..., P_t, ...$). In `Proof1`, lesar try to verify that:

$$(\forall t \in N H_t) \Rightarrow (\forall t \in N P_t),$$

which is different (in general) than the case of `Proof2`:

$$\forall t \in N (H_t \Rightarrow P_t).$$

References

[1] J-L. Bergerand. LUSTRE: un langage déclaratif pour le temps réel. Thesis, Institut National Polytechnique de Grenoble, 1986.

[2] J-L. Bergerand, P. Caspi, N. Halbwachs, D. Pilaud, and E. Pilaud. Outline of a real-time data-flow language. In *1985 Real-Time Symposium, San Diego*, December 1985.

[3] J-L. Bergerand, P. Caspi, N. Halbwachs, and J. Plaice. Automatic control systems programming using a real-time declarative language. In *IFAC/IFIP Symp. 'SOCOCO 86, Graz*, May 1986.

[4] B. Berkane. Vérification des systèmes matériels numériques séquentiels synchrones : Application du langage LUSTRE et de l'outil de vérification Lesar. Thesis, Institut Polytechnique de Grenoble, October 1992.

[5] C. Buors. Sémantique opérationnelle du langage LUSTRE. DEA Report, University of Grenoble, June 1986.

[6] P. Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94:125–140, 1992.

[7] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, POPL'87*, Munchen, January 1987.

[8] A. Girault and P. Caspi. An algorithm for distributing a finite transition system on a shared/distributed memory system. In *PARLE'92, Paris*, July 1992.

[9] A-C. Glory. Vérification de propriétés de programmes flots de données synchrones. Thesis, Université Joseph Fourier, Grenoble, France, December 1989.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. Programmation et vérification des systèmes réactifs à l'aide du langage flot de données synchrone LUSTRE. *Technique et Science Informatique*, 10(2), 1991.

[11] 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.

[12] N. Halbwachs and F. Lagnier. Sémantique statique du langage lustre - version 3. Technical Report SPECTRE L15, IMAG, Grenoble, February 1991.

[13] N. Halbwachs, F. Lagnier, and C. Ratel. An experience in proving regular networks of processes by modular model checking. *Acta Informatica*, 29(6/7):523–543, 1992.

[14] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, September 1992.

[15] N. Halbwachs, A. Lonchampt, and D. Pilaud. Describing and designing circuits by means of a synchronous declarative language. In *IFIP Working Conference "From HDL Descriptions To Guaranteed Correct Circuit Designs"*, Grenoble, September 1986.

[16] N. Halbwachs and D. Pilaud. Use of a real-time declarative language for systolic array design and simulation. In *International Workshop on Systolic Arrays*, Oxford, July 1986.

[17] N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.C. Glory. Specifying, programming and verifying real-time systems, using a synchronous declarative language. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[18] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. LNCS 528, Springer Verlag.

[19] D. Pilaud and N. Halbwachs. From a synchronous declarative language to a temporal logic dealing with multiform time. In M. Joseph, editor, *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Warwick, September 1988. LNCS 331, Springer Verlag.

[20] J. A. Plaice. Sémantique et compilation de LUSTRE, un langage déclaratif synchrone. Thesis, Institut National Polytechnique de Grenoble, 1988.

[21] J. A. Plaice and N. Halbwachs. LUSTRE-V2 user's guide and reference manual. Technical Report SPECTRE L2, IMAG, Grenoble, October 1987.

[22] C. Ratel. Etude de la conformité d'un programme LUSTRE et de ses spécifications en logique temporelle arborescente. DEA Report, Institut National Polytechnique de Grenoble, June 1988.

[23] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Thesis, Université Joseph Fourier, Grenoble, June 1992.

[24] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[25] P. Raymond. Compilation séparée de programmes LUSTRE. Technical Report SPECTRE L5, IMAG, Grenoble, June 1988.

[26] P. Raymond. Compilation efficace d'un langage déclaratif synchrone : Le générateur de code LUSTRE-v3. Thesis, Institut National Polytechnique de Grenoble, November 1991.

[27] F. Rocheteau. Programmation d'un circuit massivement parallèle à l'aide d'un langage déclaratif synchrone. Technical Report SPECTRE L10, IMAG, Grenoble, June 1989.

[28] F. Rocheteau. Extension du langage Lustre et application à la conception de circuits: Le langage Lustre-V4 et le système Pollux. Thesis, Institut National Polytechnique de Grenoble, June 1992.

[29] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits, a hardware implementation of LUSTRE. In REX *Workshop on Real-Time: Theory in Practice, DePlasmolen (Netherlands)*, pages 195–208. LNCS 600, Springer Verlag, June 1991.

[30] F. Rocheteau and N. Halbwachs. POLLUX, a LUSTRE-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II, Chateau de Bonas*, June 1991.

[31] G. Thuau and B. Berkane. Using the language LUSTRE for sequential circuit verification. In *International Workshop on Designing Correct Circuits, Lingby (Denmark)*, January 1992.