SYRF Case Studies
Schneider example
Formal verification using `xlesar`

Pascal Raymond

VERIMAG*– Centre Equation, 2 Av. de Vignate – F38610 GIERES
http://www-verimag.imag.fr/PEOPLE/Pascal.Raymond
Pascal.Raymond@imag.fr

1 Introduction

The case study, provided by Schneider Electric, is a software taken from a nuclear plant control. The program is provided as a Scade description, made of (about) 40 graphical sheets. The main program (node TDF) has 29 inputs and computes 49 outputs. It results in a flat dataflow network of about 1100 operators.

Moreover, a list of 41 properties is provided by Schneider. Those properties are extracted form the requirement documentation, and are informally given using a mix of natural language and mathematical notations.

Our goal is to use the tool lesar to formaly prove the properties. In this document, we first recall the main characteristics of lesar. In fact, using directly lesar is quite hard, so we have developped a graphical user interface called xlesar. This new tool provides a methodology for the user.

In the next part, we detail the work and the result obtained for three representative properties and briefly present the results for the 19 properties that have been treated.

2 lesar

lesar is a tool dedicated to the formal verification of safety properties for Lustre (or Scade) programs.

2.1 Input data

lesar takes a *verification program*, which is a Lustre program with a single Boolean ouput implementing the property. A verification program may include *assumptions*, which are the "`assert exp;`" statements appearing in the program.

The verification consists in checking that whatever is the (infinite) input sequence of the verification program, the output remains always true as long as the assumptions (i.e. the `assert` statements) remain true.

A classical method to build such a verification program is to put together the program to validate and two special programs (called observers) implementing respectively the property and the assumptions. The figure 1 shows the principle of this method. In Lustre, a typical scheme of verification program is the following:

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

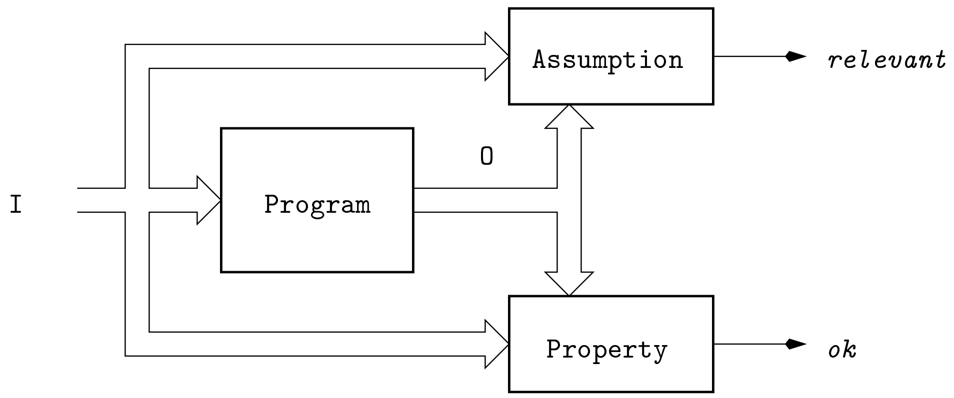

Figure 1: Verification scheme

```
node Property(I, O ) returns (ok : bool);
let ...  tel

node Assumption(I, O ) returns (ok : bool);
let ...  tel

node VERIF(I) returns (ok : bool);
var O;
let
   O = Program(I);
assert Assumption(I, O);
ok = Property(I, O);
tel
```

2.2 Fonctionality

The verification is made by model checking: lesar explores an automaton which is a finite abstraction of the verification program. This abstraction is conservative according to safety properties: if a safety property is satisfied by the abstract model, it is also satisfied by the concrete program, otherwise the proof is inconclusive. Mode precisely, the result of the model-checking is complete if the program only deals with finite domains (e.g. Boolean variables), but it is only partial if the program involves numerical variables.

lesar implements several algorithms for the exploration of the model. The simpliest one enumerates the states of the automaton, verifying at each step that the output is true (option `-enum`). More complex algorithms are said *symbolic*, in the sense that they manipulate set of states encoded by logical formulas. The first one build the formula corresponding to the reachable states, and verify that this formula implies the property (option `-forward`). The second symbolic algorithm runs backward: it builds the set of states that can lead to the violation of the property (option `-backward`), and checks that the initial state does not belong to those "bad states".

At last, lesar provides a module that can deal with simple numerical properties. This module is able to reject unfeasible linear constraints, using a internal representation based on polyhedra (option `-poly`), and must be used with the enumerative algorithm [1].

[1] The polyhedra module is not fully exploited by symbolic algorithms, so properties checked with `-enum -poly`, may be found "false" with `-forward -poly` or `-backward -poly`; indeed the result is still conservative

3 xlesar

Writing and managing verification programs for lesar is quite boring. For instance, in this case study, each verification program needs at least 200 lines of Lustre just for describing the interface and the connections with the `tdf` node. Indeed, those lines are the same for all property, but it results that verification programs are almost unreadables.

We have developped a graphical user interface for lesar, called xlesar. This interface allows the user to manage his verification work in term of "project". A project is dedicated to a program under validation (`tdf` for this case study), and it consists of a list of properties related to the inputs/outputs of this program.

3.1 Main window

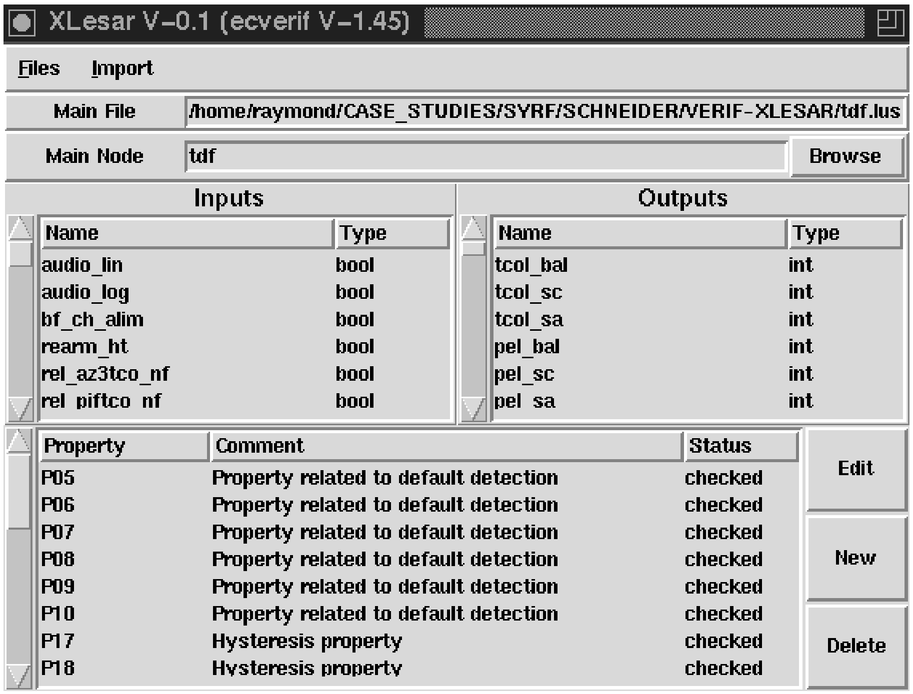

Figure 2: The xlesar main window

Figure 2 shows the main window of xlesar. The top part (Main File and Main Node) allows the user to select the program to check. The names and the types of the main node's inputs/outputs are automaticaly listed : all property must be expressed in term of those identifiers.

At last, the bottom part allows the user to manage the list of properties: it can create and delete a property, but above all edit it: the "edit" button opens a *property editor* on the selected property.

3.2 Editing/checking window

The top part is simply a text editor that allows the user to write comments. The bottom-left part allows the user to write the property. It is made of several small syntactic editors of lustre expressions: the "Property" one gives the definition of the property. But the user can define additionnal code, like assertions and intermediate variables. In the example (Fig. 3), an additional Boolean variable is defined (`code_ok`).

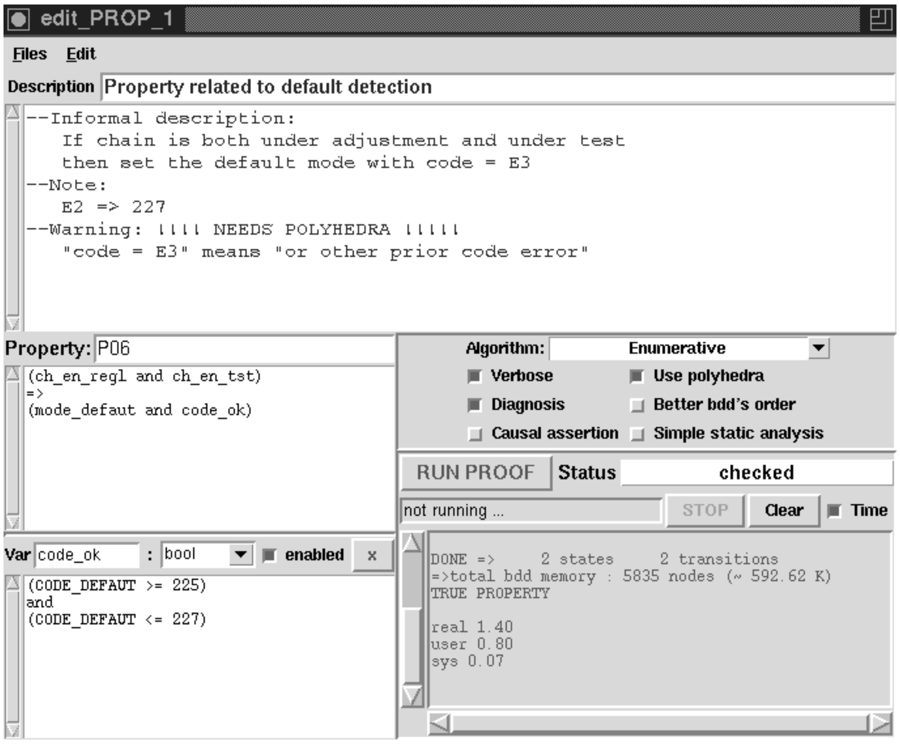

Figure 3: xlesar editing/checking window

At last, the bottom-right part is the "run proof" manager which makes the interface with the model-checker (lesar). The options of the model-checker are available (choice of the algorithm, diagnosis etc...). When the user press the "run proof" button, xlesar builds a verification program and call lesar with the selected options. The output of lesar is "echoed" in the bottom-right text window.

4 The case study

The main problem here is to formalize the properties, and to translate them into Lustre. Here are some sources of problems:

- The program is strongly modular, made of several nodes connected under a main program. The requirement says nothing about this modularity, so we cannot known if a given property has to hold in a particular sub-program or at the main level.
- It is sometimes hard to associate the "names" used in the requirement to the actual variables implemented in the program.
- The requirements are in general ambigous and not complete; in particular the necessary assumptions are missing in the requirement.

We have choosen the following approach to face the first problems:

- All the properties are written at the main level, using only the main node's inputs/outputs(TDF) [2].

[2] It appeared that a very important variable used in the requirement ("mode_defaut" for defect mode) is NOT a output in the actual implementation, but only a local variable; we have modified the program interface in such a way that this variable appears as output in tdf

- In the Scade description, inputs and outputs are organized in tuples with unnmaned fields. We have written (once for all) a main program more suitable to express the properties: it is a program with "flatened" inputs/outputs (we call it `tdf`) which consists of a call of the "real" `TDF` node. The identifiers used in this node are those comming from the requirements.

Within the set of informal properties we have identified three classes of "similar" properties. We detail in the following sections one example for each class.

4.1 Properties related to defect detection

The general scheme for those properties is the following: *if "X" happens, then the defect mode must be set, with the code "C".*

Six properties are of this kind (P05 to P10); we detail the example of P06, which is informally given by: *if chain is both under adjustment and under test then defect mode is set with the code = E3* (see fig. 3).

In order to formalize this property, we first need to find the corresponding variables:

- "chain is under adjustment" and "chain is under test" seems to correspond to 2 Boolean inputs (`ch_en_regl` and `ch_en_test`).
- "defect mode" is an output.
- "defect code" is the integer output `CODE_DEFAUT`.
- in the source file, we have found that the constant `E3` is equal to 227.

Formally, the property is then expressed by the Lustre equation (`=>` is the logical implication):

```
P6 = (ch_en_regl and ch_en_tst) => (MODE_DEFAUT and (CODE_DEFAUT = 227));
```

The verification fails with this definition: the diagnosis gives a counter example where `CODE_DEFAUT = 225`. In fact, this value corresponds to another case of defect which is in some sense prior to this one. This is not really an error: the requirement is simply indeterministic (it says nothing about the priority for the error codes).

We can now state that the "real" properties satisfied by the node are in fact of the form: *if "X" happens, then the defect mode must be set, with a code equal to "C" or other value corresponding to a prior case of defect..*

With the correct definition, the property is checked by lesar in few seconds. Note that the `-poly` option is necesary here because of the linear condition on `CODE_DEFAUT`.

5 Boolean properties

Those properties express simple relations between logical inputs and outputs, so they can easely be written in Lustre. For instance, P27 is defined by: *bf_pkc1 = bf_bf_alim and bf_ch_src*, which is already a Lustre expression.

Those properties cannot be proved without assumptions. By analysing the diagnosis we have found that all those properties are valid only "normal mode" (the source is neither under ajustment or in defect mode) :

```
(not mode_defaut) and (not ch_en_regl)
```

With this assumption, the property is checked in few seconds.

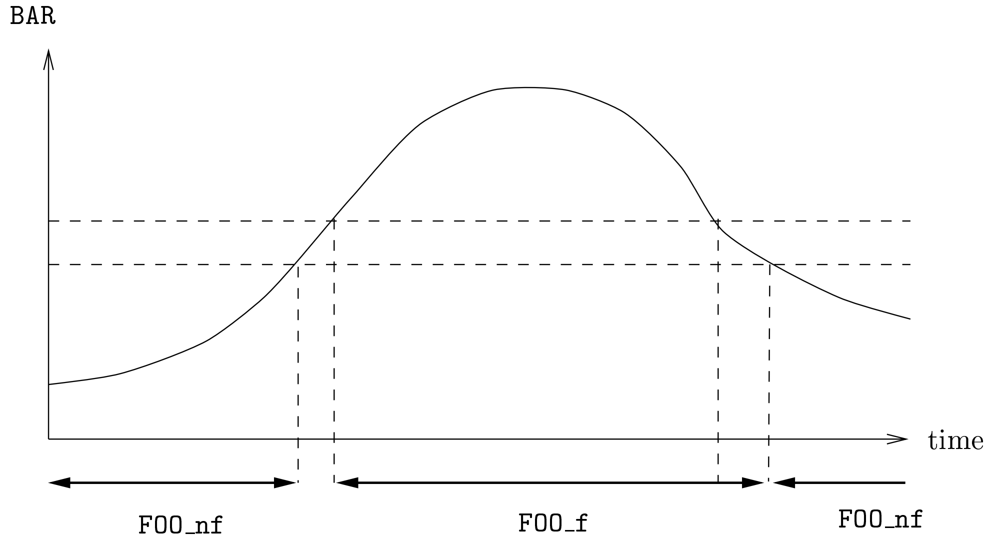

Figure 4: Hysteresis mecanism

6 Properties related to threshold

6.1 General form

Those properties are related to a numerical variable `BAR`, and a threshold designed as `FOO`. More precisely, each property involves:

- a numerical variable of the program: `BAR`
- a numerical constant (but unknown) parameter, `max_FOO`, which is a upper threshold for the values of `BAR`
- a boolean output, `FOO_f`, which means that `BAR` exceeds `max_FOO` ('f' stands for "franchi" i.e. crossed)
- a boolean output, `FOO_nf`, which means that `BAR` does not exceeds `max_FOO` ('nf' stands for "non franchi" i.e. not crossed)
- a numerical constant `hyst_FOO`, which is an "hysteresis margin" that avoids oscillating phenomena around the threshold.

For such a set of objects, there is in fact two properties. The first one is of the form:
`FOO_f` *becomes true if* $\texttt{FOO} \geq \texttt{max\_FOO}$*, and remains true until* $\texttt{FOO} \leq \texttt{max\_FOO} - \texttt{hyst\_BAR}$.
The second one is an approximation of the first one's dual:
$\texttt{FOO} < \texttt{max\_FOO} - \texttt{hyst\_BAR}$ *implies* `FOO_nf`.
Figure 4 shows an example of such a temporal behavior. One can note that those two properties do no completly define the behavior of `FOO_nf` and `FOO_f`.

6.2 Example

The couple P17, P21 is an example where `BAR` is the output `tcor`, and `FOO` is `piftco`. In order to formalise P17, we use the node `cont_from_to_` (for continously ... form ... to ...). This node is a simple two-states automaton, whose definition is:

```
node cont_from_to_( A, B, C : bool)
returns (ok : bool);
```

```
var on : bool;
let
   on = if (false -> pre on) then not C else B;
   ok = on => A ;
tel
```

6.3 First attempt

The first formal definition of is the following:

```
cont_from_to_(
  piftco_f,
  tcor >= maxpiftco,
  tcor <= maxpiftco - hysttco
)
```

Several problems and errors are occuring with this definition:

- `maxpiftco` is declared as a integer, while `tcor` is real. The type-checker integrated in xlesar found this error.
- The hysteresis `hysttco` which seems to be an absolute value in the requirement, is in fact implemented as a percentage (5 percent of the corresponding value).
- The computation of the variables involved in the property is guarded by an *activation condition*: the input `raf_icto`. That means that the value of `tcor` is taken into account only if `raf_icto` is true.
- We must state that the constant `maxpiftco` is positive.
- Like most of the properties in the requirement, this one only holds in normal mode (neither in defect mode or when the chain is under adjustement).

6.4 Final definiton

The correct definition of P17 is:

```
cont_from_to_(
  piftco_f,
  raf_icto and (tcor >= real(maxpiftco)),
  raf_icto and  (tcor <= (real(maxpiftco) * 0.95))
)
```

With the two assumptions:

- `real(maxpiftco) > 0.0`" and
- `not (ch_en_regl or mode_defaut)`,

the property is checked in few seconds, with the `-poly` option.

The property P21 is more simple:

```
raf_icto and (tcor < (real(maxpiftco) * 0.95))
=> piftco_nf
```

It is checked with the same assumptions in few seconds.

7 Results and conclusions

19 properties over 41 have been treated. No major problems were found, and this is not surprising since this software was already widely tested and validated, and it is actually running.

Using lesar directly is quite hard and boring, so we have developped a graphical tool that allows the user to write only what it necessary.

Nevertheless, none of the properties, as they are given in the requirement, were satified: this is, in general, not due to "real bugs" but to a lack of precision in the requirement. It results that the main part of the work was to find the "good" expression for each property (§ 7.1). In all cases, the checked property is quite close the requiered one, except for one (§ 7.3).

Technically, the model checker implemented whithin lesar (together with the "polyhedra module") was largelly sufficient to perform the proofs (§ 7.2). However, we have optimized the front-end of the model checker, in order to perform a better static analysis of the program § 7.4).

The remaining properties seem to be of the "same" kind of those that have been treated: they only involve Boolean properties with simple (and static) numerical conditions. Nevertheless at least one property clearly talks about the dynamic behavior of numerical variables; such a property shows the limits of the approach implemented within lesar(§ 7.5).

We detail in this section those several points.

7.1 Formalysing is the hard part

The major part of the work was to formaly express the properties, and, in particular to find the "good" assumptions (just necessary).

Technically, some of the properties seem to be "complicated" in the sense that they involve numerical conditions. In fact, it appears that the static linear decision module (based on polyhedra) is sufficient to deal with those conditions [3]. At the end, that those properties are not complicated, in the sense that lesar is able to check them in few seconds (more formaly, the associate models are quite small, consisting of few states and few transitions).

7.2 Sizes and times

We detail here the results for all the properties that have treated.

The *network* column gives the size of the verification program in number of basic operators. lesar first try to minimize the network by computing a bisimulation over the set of operators: the second number in the column is the size after minimization.

The column "Enum" gives the results for the enumerative algorithm; the size of the explored automaton is given in "number of states / number of transitions".

For the symbolic forward algorithm we give the number of steps, which is the diameter of the reachable states. For the symbolic backward algorithm we give the number of steps, which is the diameter of the "bad" states.

"Fails" means that the proof were inconclusive

The computation time is not mentioned: it never exceed 1 cpu-second on a SPARC ULTRA-10 station, both for conclusive and inconclusive result.

Properties related to defect detection.
For all those properties, we have used the `-poly` option, but in fact a simpler decision module would have been sufficient (interval analysis on error codes values, or, even more simply, enumerated types for encoding error codes).

Note that, for this kind of very simple "numerical" properties, the symbolic algorithms with the `-poly` option are conclusive.

[3] In some cases, like P6 4.1, this module is clearly too powerful: a module dealing with enumerated types, or even intervals would have been sufficient.

Property	network	Enum	Forward	Backward
P05	1117 → 815	2/2	2	1
P06	1113 → 812	2/2	2	1
P07	1115 → 812	2/2	2	1
P08	1115 → 812	2/2	2	1
P09	1118 → 812	2/2	2	1
P10	1117 → 812	2/2	2	1

Hysteresis properties

Those properties are needing the full power of the polyhedra module since they involved linear constraints over two unknown numerical values (some computed value and some constant but unknown threshold). In half cases, the symbolic algorithms succeed, but there is no general conclusions to emit (just good luck).

Property	network	Enum	Forward	Backward
P17	1123 → 819	4/40	2	2
P18	1123 → 819	4/40	fails	fails
P19	1124 → 820	5/56	2	2
P20	1124 → 820	5/56	2	2
P21	1116 → 813	4/40	fails	fails
P22	1116 → 813	4/40	fails	fails
P23	1117 → 814	5/56	fails	fails
P24	1117 → 814	5/56	fails	fails

Boolean properties
Those properties do not need numerical decision, so we have tried the three available algorithms.

Property	network	-enum	-forward	-backward
P27	1111 → 810	7/126	2	1
P28	1120 → 819	35/3244	2	1
P29	1120 → 816	2/2	2	1
P30	1112 → 811	5/40	2	1
P31	1110 → 809	3/12	2	1

7.3 The property P29: bug?

Intuitively, this property states that the source channel behaves correctly (output `bf_ch_src`) if a conjunction of conditions is true. The exact definition is:

bf_ch_src = not mode_defaut and not SEUIL_DISC_val and not VAL_HT_val and porte_fermee and bf_ventil

As it, the property is found false. More precisely, we can check the implication:

`(not mode_defaut and not SEUIL_DISC_val and not VAL_HT_val and porte_fermee and bf_ventil) => bf_ch_src`

but the opposite implication is false.

By analysing the program, one can find that the problem is due to the condition `not mode_defaut`. In the implementation, this condition appears in the definition of `bf_ch_src` as an *activation condition* rather than a term. In the program, the definition of `bf_ch_src` is of the form:

```
bf_ch_src = if (not mode_defaut) then (...) else (true -> pre bf_ch_src);
```

So it is possible to have `bf_ch_src = true` while `mode_defaut = true`: the property is *scricto sensu* false.

However, we have checked a weaker property under the assertion "`not mode_defaut`":

P06	State var.	Enum: states/trans (cpu)	For: steps (cpu)	Back: steps (cpu)
Syntactic dep.	25	$2907/6.8\,10^6$ ($\sim$ 1 hour)	6 (1.6s)	1 (0.3s)
Semantic dep.	0	2/2 (0.3s)	2 (0.3s)	1 (0.3s)

P27	State var.	Enum: states/trans (cpu)	For: steps (cpu)	Back: steps (cpu)
Syntactic dep.	25	$2905/12.3\,10^6$ ($\sim$ 1 hour)	5 (0.8s)	1 (0.3s)
Semantic dep.	4	7/1008 (0.5s)	2 (0.3s)	1 (0.3s)

Figure 5: Influence of dependency check

```
bf_ch_src = ( not SEUIL_DISC_val and not VAL_HT_val
    and porte_fermee and bf_ventil)
```

Our opinion is that this problem is not really a bug, but it is due to the ambiguity of the requirement. One one hand the property P29 is clear: `mode_defaut` must imply `not bf_ch_src`. On the other hand, when `mode_defaut` becomes true, the whole system must stop (within a elementary delay), so no property has to hold any longer!

7.4 Importance of static analysis

In some sense, the properties are simple because they are almost "written" in the implementation, but not so simple since they are more or less hidden in a huge amount of code. The main technical problem is then to identify in the software the sub-part that is really involved in each property. This is made by a static analysis of the code that find the (minimal) subset of inputs and state variables (i.e. Boolean memories) on wich the property depends on. This static analysis was significantly improved to treat the case study: before this case study, the dependance analysis performed by lesar was purely syntactic. It results that a lot of useless variables was taken into account in the model. Now, the dependance checking is semantic in the sense that it performs full Boolean simplification before dependancy checking. In some cases, this new algorithm reduces the number of states variables on which the property depends to zero; this is the most obvious evidence that the property is simply "hidden somewhere" in the program. The advantage is dramatic while using the enumerative algorithm, and it is still significant while using the symbolic forward algorithm. Note that this optimization is (almost) useless with the backward option, for which the dependancy checking is already a "side-effect" of the algorithm 5.

7.5 Limits of model checking

Most of the remaining properties are similar to the ones that have been treated, but one of them clearly shows the limit of automatic verification: P11 states that *for high counting rate, tcor = tcol.* First, this property is too imprecise (what is a "high counting rate"?), and, above all, this property deals with the dynamic behavior of numerical variables (`tcor` and `tcol`), so it is clearly out of the scope covered by lesar.