Specification and Validation of Embedded Systems: A Case Study of a Fault-Tolerant Data Acquisition System with Lustre Programming environment†

F. Maraninchi*, N. Halbwachs*, P. Raymond*, C. Parent* and R. K. Shyamasundar**

* Verimag♮ Grenoble University, France, {Florence.Maraninchi, Nicolas.Halbwachs, Pascal.Raymond, Catherine.Parent}@imag.fr
** Tata Institute of Fundamental Research, Mumbai, India, shyam@tifr.res.in

We show how to specify and validate an embedded system using the Lustre programming environment. The case-study considered is a fault-tolerant system for the acquisition of gyroscopic data in a military aircraft. We illustrate the use of Lustre tools for describing, simulating, and verifying the system. Beside, we show how the formalization of the requirements by means of an executable language allows ambiguities to be removed, and how the system can be developed step by step, while simulation and validation take place at each step. We believe that this example is representative of a wide class of embedded systems.

Keywords - Embedded real-time systems,Language design and implementation, Synchronous languages, Formal methods, Formal verification, Executable specifications.

I. Introduction

The family of *synchronous languages* [BB91], [BCE+03] has been quite successful in offering formally defined languages and programming environments for safety-critical reactive systems.

Lustre [CHPP87] has been defined and studied in the Verimag laboratory. It is a dataflow language, well-suited for the description of regulation systems. The industrial programming environment SCADE, developped by Esterel-Technologies[1], is based upon Lustre. SCADE is now a de-facto standard, worldwide, for the development of critical embedded sotware, especially in avionics, automotive, or energy production. On the academic side, several tools have been developed around Lustre at Verimag: a simulator called Luciole, a debugger called Ludic, two verification tools — Lesar is a modelchecker, NBac is a tool based on abstract interpretation and dedicated to the verification of numerical properties — and a tool for automatic testing, Lurette.

In this paper, we illustrate the use of these tools in the design of a component of an avionic software. The example is a fault-tolerant system computing the gyroscopic data for a military aircraft.

a. *What is in the paper:* Apart from demonstrating the use of LUSTRE tools, the goal of the paper is twofold:
   – it intends to show how the use of an executable, formally clean, description language can help in understanding an informal specification, in removing ambiguities, and in communicating, by showing early simulation, with the author of the specification.
   – it illustrates a progressive top-down design, with simulation and validation at each step.

b. *What is not in the paper:* Two important features of the case studies were not considered during this experiment:
   – The considered system aims at fault-tolerance. While the experiment shows that the approach is well-suited for programming this kind of fault-tolerance software, the problem of measuring and validating the faulttolerance itself was not addressed at all. On one hand, it would need the use of completely different validation tools (taking into account stochastic aspects), and on the other hand no quantitative requirements were available for the case study.
   – The real implementation should be *distributed*. In this study, we only considered the functional aspects of the specification, and we did not address

† The authors thank IFCPAR (Indo-French Centre for Promotion of Advanced Research) under which part of the work was done.
♮ Verimag is a joint laboratory of Universit´e Joseph Fourier, CNRS and Grenoble-INP.
[1] see `http://www.esterel-technologies.com/`.

the distribution. Note that an ideal approach would be to validate first a centralized version of such a specification, and to use automatic code distribution tools preserving the functional properties. Some proposals for such an automatic distribution of LUSTRE programs have been made [CGP99], [CMSW99], [SC04].

II. An Overview of Lustre and its Programming Environment

A. The language

In a dataflow language for reactive systems, both the inputs and outputs of the system are described by their *flows* of values along time. Time is discrete and instants may be numbered by integers. If x is a flow, we will note x_n its value at the nth reaction (or nth *instant*) of the program.

A program consumes input flows and computes output flows, possibly using local flows which are not visible from the environment. Local and output flows are defined by *equations*. An equation "x = y + z" defines the flow x from the flows y and z in such a way that, at each instant n, $x_n = y_n + z_n$.

A set of such equations, using arithmetic, Boolean, etc. operators, describes a network of operators, and is similar to the description of a combinational circuit. The same constraints apply: one should not write sets of equations with instantaneous loops, like : $\{x = y + z;\ z = x + 1;\ ...\}$. This is a set of fixpoint equations that perhaps has solutions (see [SBT96], for more detailed discussion), but it is not accepted as a dataflow program. For referencing the past, the operator pre is introduced : $\forall n > 0$; $(\text{pre}(x))_n = x_{n-1}$.

One typically writes `T = pre(T) + i`, where T is an output, and `i` is an input. It means that, at each instant, the value of the flow `T` is obtained by adding the current value of the input `i` to the previous value of `T`. Initialization of flows is provided by the –> operator. E –> F is an expression, the value of which is the one of E at the first instant (i.e., E_0), and then the one of F forever (i.e., $F_n{:}\forall_n > 1$). The equation `X = 0 -> pre(X) + 1` defines the flow of integers; as a reactive program, it produces values on the *basic clock*.

The conditional structure is a ternary combinational operator, and is *strict*: the two branches are always evaluated. One writes: `X = if C then E else F`, where `C` is a Boolean expression and `E1`, `E2` are two expressions of the same type, meaning: $\forall n > 0$, X_n = if C_n then E_n else F_n.

The language is structured by the definition of reusable *nodes* that can be called anywhere in expressions defining variables. Programs usually input a library of small wellidentified reactive behaviors, like a "two-states" with reset, a "bounded counter", etc.

B. An example Lustre program

As a very simple example of program, we give a Lustre node that will be used later in the case study. The node named "`maintain`" denotes an operator that receives an integer `n` and a Boolean `b` as input parameters, and computes a Boolean output `m`, which is true whenever `b` has been maintained high during the last `n` cycles. The node uses a counter `cpt`, which is set to `n` whenever `b` is false, and decremented to 0 otherwise. The output `m` is true when `cpt` is zero.

```
node maintain (n : int ; b : bool)
     returns (m : bool) ;
var cpt : int ;
let
    cpt = n -> if b then
                  if pre(cpt)>0 then
                     pre(cpt) - 1
                  else pre(cpt)
               else n ;
    m   = (cpt = 0) ;
tel
```

C. The programming environment

In addition to the industrial environment SCADE — which proposes a graphical interface, a simulator, and a compiler —, an academic toolset has been developed around Lustre. The following tools or prototypes are available:

- a compiler into C.
- a simulator, called Luciole, which allows an early simulation of Lustre nodes. It displays an interactive board, where inputs can be entered and outputs are displayed. It is connected with the tool Sim2chro, which displays the history of inputs/outputs by means of timing diagrams.
- two verification tools: they are both restricted to the verification of *safety properties*, described by means of *synchronous observers* [HLR93]: a synchronous observer is a Lustre program, taking as inputs the input/output variables of the program under verification, and signaling whenever the property considered is violated. This technique is used both for describing required properties, and assertions about the environment which must be assumed for these properties to hold. The tools differ in the techniques applied for verification:
  - Lesar [RHR91] is quite a standard symbolic modelchecker [BCM+90]. It checks the property by exploring (enumeratively or symbolically) a finite state model of the program, which abstracts away all its numerical aspects. As a consequence, it is not able to verify properties depending on the dynamic behavior of the numerical variables. It should be used for control dominated properties.
  - NBac [JHR99] is able to handle simple numerical properties, thanks to the use of "Linear Relation Analysis" [HPR97], a special case of abstract interpretation [CC77]. However, NBac is much more expensive than Lesar, and should only be applied to fairly small programs.
- a prototype debugging tool [MG00].

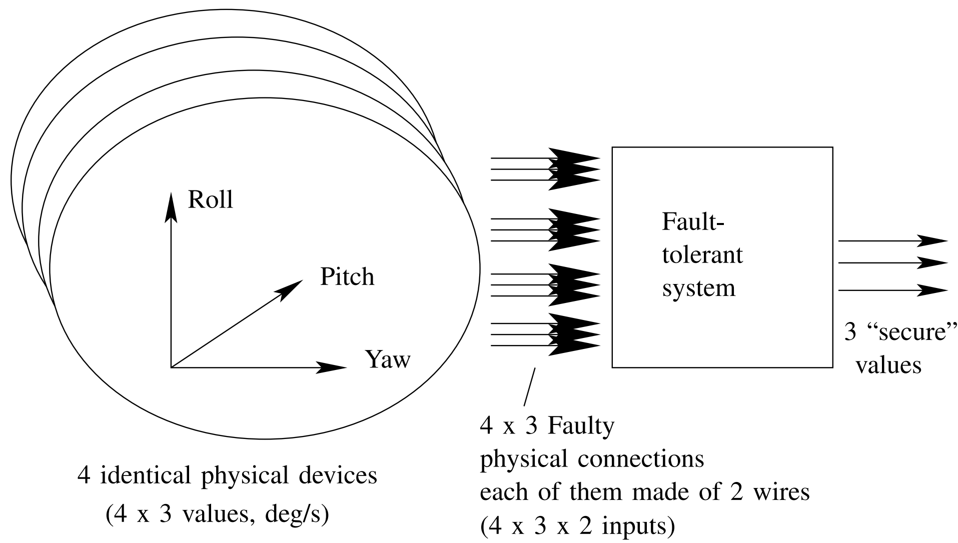

Fig. 2 : Description of the physical system

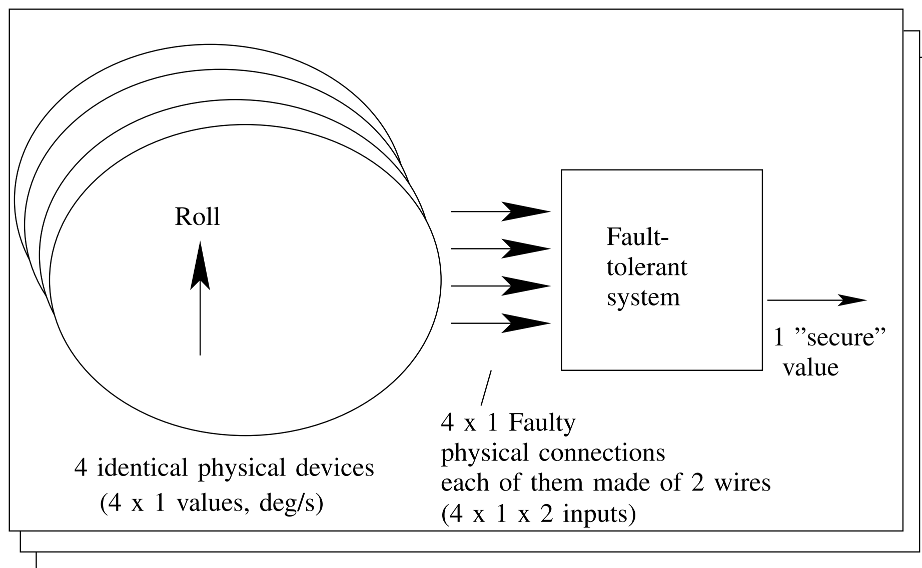

Fig. 3 : Description of the physical system for one axis

Industrial vs. academic tools

Since the release of Scade-V6, there are significant discrepancies between the industrial and academic versions of the language. In particular, Scade-V6 contains a notion of hierarchic automata [CPP05], inspired both by Esterel [BS91] and mode automata [MR03], which is not in the academic version. The mechanisms for defining and handling arrays are also different. In this paper, we will not use these incompatible features, thus conforming to both versions.

III. Informal description of the Gyroscopic System

A. Development of the Case Study

We started from an informal specification in English, made of *functional* requirements and some *timing* requirements, that are called *performance* requirements. For instance, several distinct working rates are required for the parts of the system: 0.05s, 0.0125s, etc.

We first identified the system interface, i.e., the *physical* inputs and outputs, and some additional inputs that model *faults*. Then we wrote a single-clock Lustre program mimicking the internal structure of the informal specification, but forgetting about the multi-rate requirements. This is already an interpretation of the informal specification. For instance, we translated the English "a followed by b immediately" by *"at the next step"*.

At each step of the development, we ran manual simulations, and, as far as possible, we used verification tools to establish important properties.

B. Physical structure

Fig. 2 describes the system and its physical environment. The system is connected to four gyroscopes, each of them measuring the angle variations along three axes named *roll, pitch* and *yaw*. The values obtained by one of these physical devices, for one axis, are transmitted to the computer system along two wires. Hence the system receives 4 x 3 x 2 values. From these 24 values, it has to compute only three, called *secure values*.

The first step in modeling the system in Lustre, is to concentrate on one axis only, since the behaviour on all axes are all the same[2]. We shall be using the diagram depicted in Fig. 3 as the reference in the rest of the paper.

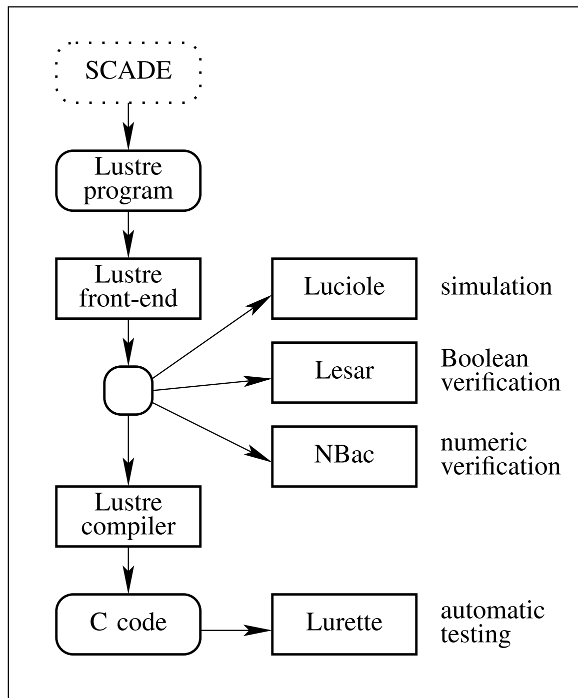

Fig. 1 : The Lustre programming environment

C. The voting principle

The internal structure of the system is as follows: it is made of four *channels*, each of them being in charge of the two wires that come from one of the four gyroscopes (remember we concentrate on one axis only, say *roll*). Each channel delivers one value, and there is a vote to compute one single value out of four, depending on the current fault conditions. The behaviour is as follows: if all the channels are working, then take the Olympic average of the four values (i.e., the average of all the values, except the two extreme ones); if one channel has failed, take the median value, among the other three; if two channels have failed, take the average of the two remaining ones. Three or more channels having failed at the same time is supposed to have a very low probability, but the case is handled by the system emitting a so-called "*safe value*", which is to be maintained for some delay, even when the bad situation disappears.

Of course, the channels are intended to run on different processors, for the purpose of redundancy. For simplicity, we shall ignore these distributed aspects in our modelling.

D. The faults

The difficult part is the detection of faults. First, we have to define what we call faults, and then to show how the redundant structure of the gyroscopic system can handle them.

Definition of faults

The system is able to handle two kinds of faults: *link faults*, that are due to some bad behavior of a physical link between the actual measurement devices and the computer; *sensor faults*, that are due to the measurement devices (the sensors) themselves being broken or not working properly for some time.

Modeling and Detection of faults

Each channel compares the values it receives on the two wires, and is able to detect local discrepancies. This double transmission of values from one gyroscope to the computer system is there to detect *transmission faults*. Note that, for a fault to be reported, the two values have to differ by more than Δ_v during consecutive Δ_t units of time.

Each channel compares the values it receives on the two wires, and is able to detect local discrepancies. This double transmission of values from one gyroscope to the computer system is there to detect transmission faults. Note that, for a fault to be reported, the two values have to differ by more than v during consecutive t units of time. Moreover, in order to support sensor faults, channels talk to each other and exchange values, so that each of them can compare its own value to the other three. If one of the gyroscopes is not working, the value it delivers will probably differ from the values given by the three other devices. Channels also have to exchange their failure statuses, because each one should compare its value to the values of the other channels, but only of those *that do not declare themselves* failed (a channel declares itself failed when it detects a transmission fault).

[2] Actually, the behaviour of pitch is slightly different.

E. Latching faults and resetting

Some faults are considered more serious than others, and should therefore be *latched*: even if the cause of the fault disappears, the channel continues to declare itself failed. Hence, each channel has an internal *state*, that reflects the faults it has encountered.

Typically, transmission faults (discrepancies between the two values received on the two wires) are not latched, because they are considered to be physically temporary.

Conversely, faults that are due to cross-channel comparisons are latched: one of the physical devices is supposed to be off, and it is unlikely to repair during the flight.

Of course, there should be some way of resetting the latches. This is done in our system thanks to two kinds of resets, called *OnGroundReset* and *InAirReset*. *OnGroundReset* can be thought of as a general resetting mechanism that happens only when it is really safe to do so, namely on ground. *InAirReset* is more interesting. First, it is not automatic, but results from a pilot action. Hence the pilot should be given some information about the internal state of the fault-tolerant controller, in order to decide whether he/she should issue a reset. Moreover, it is to be taken into account only under some conditions.

The internal state of a channel, regarding the latched faults, is one of the following:

- Everything is working properly, or there are transmission faults from time to time, but they are not latched
- There has been at least one serious fault in the past, and the failure is latched. The internal state can be driven to the normal one with any of the resets (on ground or in air), but for the *InAirReset* to work, some conditions on the measured values have to hold.
- There have been several serious faults in the past, or one serious fault followed immediately by a transmission one, then the fault is latched *and* reset-inhibited. In this case, only the *OnGroundReset* may restore the normal state. The system should never enter a global state in which more than two channels are reset-inhibited, since it cannot be repaired on board.

A careful reading of the informal documentation gives all the details about the possible *transitions* among these three states. Writing down these transitions allowed us to make precise the priorities, and to remove some ambiguities. We give the automaton point of view on this part of the system, in section VI-C below.

F. Special case for third faults

Finally, the faults are not treated the same depending on their order of occurrence: the first and second one (among four channels) are treated the same, but it is said in the informal documentation that *a third failure should not cause the channels to become reset-inhibited*. Hence, we need to treat the third faults in a special way – which will be clear in the sequel.

IV. Describing the Architecture in Lustre

The architecture of the Lustre program is exactly the same as the one described in the informal specification – thanks to the dataflow style of the language. Figure 4 shows the main structure of the system (for one axis): The four channels will be implemented as four identical nodes. The voter is another node. An additional node, the global allocator, will deal with the problem of third faults.

This direct translation of the specification into the program architecture highlights the advantages of some features of the language:

- the notion of concurrency corresponds to the logical concurrency of the specification;
- moreover, this concurrency can model a physical concurrency, as it is the case, here, if the four channels are to be distributed; the communication — here, it will be simply a delay — is an abstract model of the real communication between distributed processes (see [Cas01], [HB02], [JHR+07] for a more accurate modeling of physical concurrency);
- the connections between nodes clearly reflect the specification, thanks to the data-flow communication between nodes;
- the four channels will be essentially instantiations of a single node – thanks to the functional nature of the language.

A. Global inputs and outputs

The system receives:

- 4 pairs (or 2 4-tuples, Roll_a and Roll_b) of flows of type "real", coming from the sensors;
- 2 Boolean flows, On_Ground_Reset and In_Air_Reset.

It computes a single safe value for Roll.

B. The component interfaces

The channels

Each channel receives a pair (roll_a, roll_b) from outside. The reset signals OnGroundReset and InAirReset (implemented as Boolean flows) also come from outside. It receives also, from each other channel, the computed value foreign_roll and the failure status foreign failure roll. Finally, it receives a Boolean flow Inhib_Roll_Allowed from the global allocator.

The channel computes three outputs: its local value local_roll for roll, its transmission failure status transmit_failure_roll (which will be broadcast both to other channels and to the voter), and a request for "reset inhibition" arbitration, Ask_inhib_roll to the global allocator.

The exchange of values foreign_roll and foreign_failure_roll among channels at once raises a problem, since local roll and transmit_failure_roll will be computed as combinational functions of the inputs foreign roll and foreign failure roll. As a consequence, delays should be introduced somewhere,

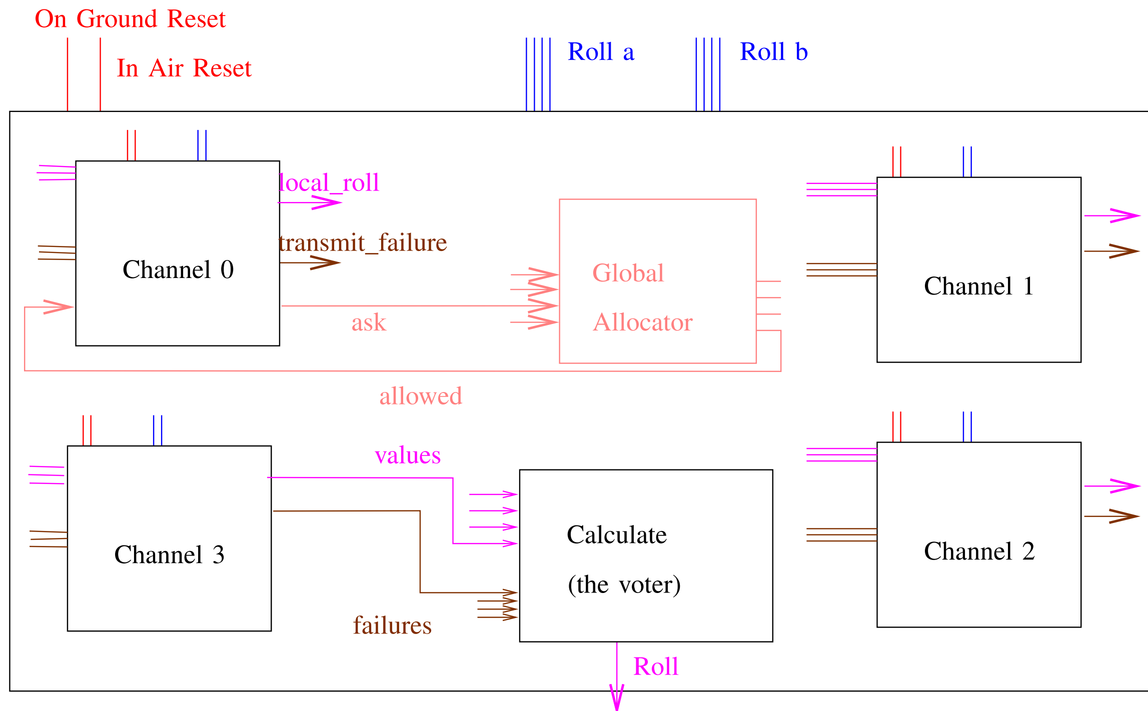

Fig. 4 : Architecture of the Lustre program: connecting the four channels together

to prevent local_roll and transmit_failure_roll to depend instantaneously on themselves (remember that combinational loops are forbidden in Lustre, and rejected by all tools). So, we delay broadcasting the value (using a "pre" operator) when entering the channel. For instance, if we call foreign_roll_j the value entering the channel j coming from channel i, and local_roll_i the value computed by channel i, we will have

$$\text{foreign_roll}_{ij} = \text{pre}(\text{local_roll}_i)$$

The voter

It simply receives the values local roll and transmit_ failure_roll from the channels, and computes the safe value Roll. However, it is not combinational (see §V).

The global allocator

It is in charge of dealing with "reset inhibition". It needs the outputs local_roll, transmit failure_roll and Ask_inhib_roll from the channels, and returns four Boolean allowed (one for each channel). Here again, to avoid combinational loops, the input Inhib_Roll_Allowed to each channel will be the delayed version of the corresponding output allowed of the global allocator.

C. Connecting the four channels together

According to the general architecture described above, we are able to write the main node, which invokes the main components, with suitable connections. It is described[3] in Fig. 5.

D. One channel

We can go one step further in the description of the architecture, by giving the internal structure of one channel (see Fig. 6). It is made of two parts: `Monitor` detects the transmission discrepancies; `FailDetect` talks to the three other channels and knows about the internal fail status of the channel. It also talks to the global allocator, which knows about the failure status of all the four channels, and is able to prevent a third failure from becoming reset-inhibited.

V. The Voter

The voter is only a consumer of values produced by other modules. Thus, it can be designed in isolation. Fig. 7 describes the Lustre code for the complete voter. The details of its parts are described below. Notice that the voter is not a combinational node: it has to memorize a fragment of its past.

A. The timing aspects

Remember that the "*safe value*" must be maintained if three or more channels have been faulty "recently". The first three lines define a counter `cpt_roll`, which is non zero exactly when there was a case with three failures, in the recent past. "recent" means within the last `SAFE_COUNTER_TIME` units of time, where `SAFE_COUNTER_TIME` is a constant. The

[3] Because of the very regular structure of this node, it would be more concisely and elegantly described by means of Lustre-V4 arrays. However, since these arrays differ significantly from those of Scade, we decided not to make use of them.

writing in Lustre is quite simple: just restart the counter with value `SAFE_COUNTER_TIME` each time there are three faults, and then decrement it at each step, until it reaches zero :

```
cpt_roll = 0 ->
    if three_roll then SAFE_COUNTER_TIME
    else if pre (cpt_roll)> 0
         then pre(cpt_roll)-1
         else 0 ;
```

B. Counting faults

The nodes `noneof`, `oneoffour`, `twooffour`, `threeoffour` are intended to count the faults, i.e., the number of Boolean variables that have value true, among `f1`, `f2`, `f3`, `f4`. They can be programmed with integers, of course, like in Fig. reftwooffour.a. However, there is also a form that does not make use of numerical variables, and that can be necessary for decidability reasons when trying to perform formal verification with Lesar. Hence, we will sometimes use the node of Fig. 8.b.

Similar nodes for `noneof`, `oneoffour`, and `threeoffour` are easy to write.

C. The voting mechanism itself

Then comes the voting itself. The conditional expression mimics the informal documentation. The auxiliary nodes OlympicAverage, Median, and Average are straightforward.

D. Validation

It is difficult to express properties of the voter, apart from the whole specification (which would be a rephrasing of the program). For this simple device, we can do a simulation using

```
node GYRO ( Roll_a_1, Roll_b_1, Roll_a_2, Roll_b_2,
   Roll_a_3, Roll_b_3, Roll_a_4, Roll_b_4 : real;
   On_Ground_Reset, In_Air_Reset : bool)
returns (Roll : real);
var
   local_roll_1, local_roll_2, local_roll_3, local_roll_4 : real;
   transmit_failure_1, transmit_failure_2,
   transmit_failure_3, transmit_failure_4 : bool;
   ask_1, ask_2, ask_3, ask_4 : bool;
   allowed_1, allowed_2, allowed_3, allowed_4 : bool;
let
   (local_roll_1, transmit_failure_1, ask_1) =
      Channel(Roll_a_1, Roll_b_1, On_Ground_Reset, In_Air_Reset,
              pre(local_roll_2), pre(transmit_failure_2),
              pre(local_roll_3), pre(transmit_failure_3),
              pre(local_roll_4), pre(transmit_failure_4),
              allowed_1);
   (local_roll_2, transmit_failure_2, ask_2) =
      Channel(Roll_a_2, Roll_b_2, On_Ground_Reset, In_Air_Reset,
              pre(local_roll_1), pre(transmit_failure_1),
              pre(local_roll_3), pre(transmit_failure_3),
              pre(local_roll_4), pre(transmit_failure_4),
              allowed_2);
   (local_roll_3, transmit_failure_3, ask_3) =
      Channel(Roll_a_3, Roll_b_3, On_Ground_Reset, In_Air_Reset,
              pre(local_roll_1), pre(transmit_failure_1),
              pre(local_roll_2), pre(transmit_failure_2),
              pre(local_roll_4), pre(transmit_failure_4),
              allowed_3);
   (local_roll_4, transmit_failure_4, ask_4) =
      Channel(Roll_a_4, Roll_b_4, On_Ground_Reset, In_Air_Reset,
              pre(local_roll_1), pre(transmit_failure_1),
              pre(local_roll_2), pre(transmit_failure_2),
              pre(local_roll_3), pre(transmit_failure_3),
              allowed_4);
   (allowed_1, allowed_2, allowed_3, allowed_4) =
       Allocator(ask_1, ask_2, ask_3, ask_4);
   Roll = Voter(local_roll_1, local_roll_2,
                local_roll_3, local_roll_4,
                transmit_failure_1, transmit_failure_2,
                transmit_failure_3, transmit_failure_4);
tel
```

Fig. 5 : The `main` ***node***

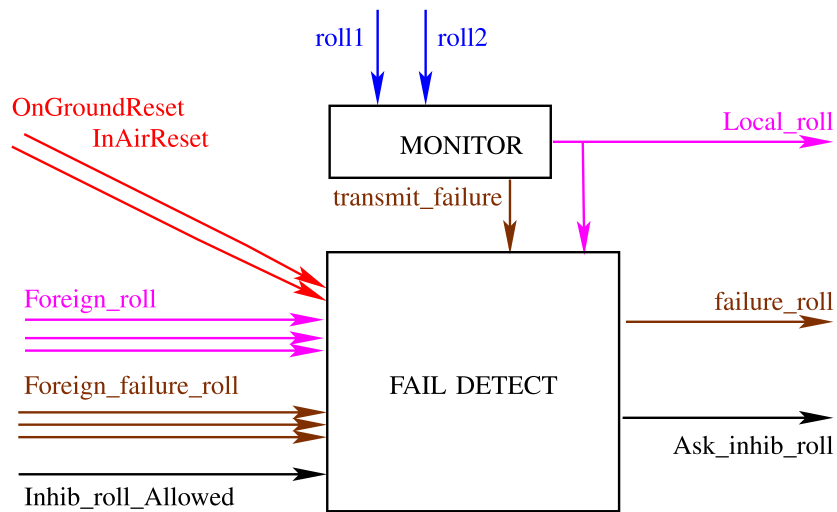

Fig. 6 : Architecture of the Lustre program: a channel

```
node Voter ( x1, x2, x3, x4 : real ; -- four values given by the channels
             f1, f2, f3, f4 : bool ; -- failure statuses seen by the four channels
)
returns  (x : real)
var
    zero_roll, one_roll, two_roll, three_roll : bool ; -- numbers of failures
    cpt_roll : int ; -- a counter
let
    cpt_roll = 0 -> if three_roll then SAFE_COUNTER_TIME
                    else if pre (cpt_roll)>0 then pre(cpt_roll) - 1
                    else 0 ;
    zero_roll  = noneof      (f1, f2, f3, f4) ;
    one_roll   = oneoffour   (f1, f2, f3, f4) ;
    two_roll   = twooffour   (f1, f2, f3, f4) ;
    three_roll = threeoffour (f1, f2, f3, f4) ;
    x = if (zero_roll and cpt_roll = 0 ) then
                   OlympicAverage (x1, x2, x3, x4)
        else if (one_roll and cpt_roll = 0 ) then
                   Median (x1, x2, x3, x4, f1, f2, f3, f4 )
        else if (two_roll and cpt_roll = 0 ) then
                   Average (x1, x2, x3, x4, f1, f2, f3, f4 )
        else FAIL_SAFE_ROLL_VALUE ;
tel ;
```

Fig. 7 : The `voter` ***in Lustre***

```
node twooffour (f1, f2, f3, f4 : bool)
   returns (r : bool)
let
   r = ((if f1 then 1 else 0) +
        (if f2 then 1 else 0) +
        (if f3 then 1 else 0) +
        (if f4 then 1 else 0)) = 2 ;
tel
```

(a) A version with counter

```
node twooffour (f1, f2, f3, f4 : bool)
   returns (r : bool)
let
    r = f1 and
          (f2 and not f3 and not f4  or
           f3 and not f2 and not f4  or
           f4 and not f2 and not f3) or
        f2 and
          (f1 and not f3 and not f4  or
           f3 and not f1 and not f4  or
           f4 and not f1 and not f3) or
        f3 and
          (f2 and not f1 and not f4  or
           f1 and not f2 and not f4  or
           f4 and not f2 and not f1) or
        f4 and
          (f2 and not f3 and not f1  or
           f3 and not f2 and not f1  or
           f1 and not f2 and not f3) ;
tel
```

(b) A purely Boolean version

Fig. 8 : The node `twooffour`

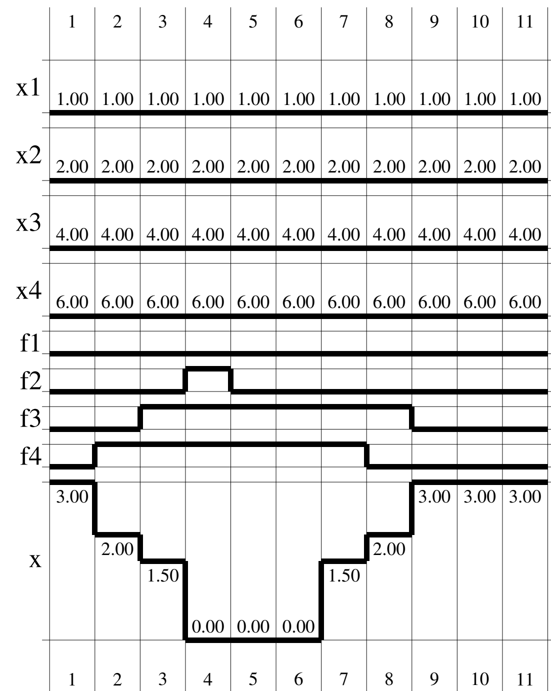

Fig. 9 : A Luciole simulation of the voter — Initially, there is no fault, so the output is the olympic average of the inputs (i.e., the average of 2 and 4). At step 2, a fault occurs on channel 4, so the outputs is the median of `x1`***,*** `x2`***,*** `x3`***, which is 2. At step 3, channel 3 becomes faulty, so the result is the average of*** `x1` ***and*** `x2`***. At step 4, a third fault occurs, so the output takes the "safe" value 0 for 3 units of time.***

Luciole. Fig. 9 shows such a simulation: we choose constant inputs for the values `x1`, `x2`, `x3`, `x4` coming from the channels, and just play with the occurrences of transmission faults `f1`, `f2`, `f3`, `f4`, observing that the correct output is computed in each case. The simulation is done with `SAFE_COUNTER_TIME = 3` and `FAIL_SAFE_ROLL_VALUE = 0`.

VI. Incremental implementation of functionality

Once we have designed the global architecture of the program, and taken decisions about where to place Lustre *pre's*, we can develop the whole program progressively. We can start by giving each node some trivial behavior (like `output = constant`), just to see whether the architecture indeed compiles. It allows typing and self-dependence problems to be detected.

Then we can start writing more and more appropriate code for each of the nodes. We used four steps, each of which with simulation :

- We start (§VI-A) with the detection of transmission failures only (the channels do not talk to each other). This can be observed on one channel only, first, and then we can put together all the four channels.
- Then (§VI-B), we add the detection of faults that are due to cross-channels comparisons, but without latching them.
- Then we implement the latching of faults and the effect of resets (§VI-C).
- Finally, we implement the global allocator (§VI-D) that allows a special behavior to be given to third fault.

A. Local detection of transmission faults only

Lustre code

In each channel, the node `Monitor` determines if the two values received by the channel differ too much for too long a time. It also transmits some combination of the two values received as its "local" value. Nothing is said about the combination function in the informal documentation.We have chosen to output the first value. Any other choice could be implemented in a very simple way.

```
node Monitor (
    xa, xb : real ;  -- two input values
)
returns (
    local_value  : real ;
    -- the value seen by this channel
    failure : bool ;
    -- detection of a transmission fault
)
let
    failure = maintain(TIME_ROLL, abs(xa - xb)
                > DELTA_ROLL);
    local_value = xa ;
tel
```

TIME_ROLL and DELTA_ROLL are two constants; maintain is the Lustre node presented in §II-B.

Simulations

Fig. 10 shows a simulation of the node Monitor, with constants

DELTA_ROLL = 14 and TIME_ROLL = 3

B. Adding non-latched cross-channel comparisons

Lustre code

For cross-channel comparisons, the local value xi is compared to those among the foreign ones that are not declared failed (that is why we need the fail status of the three foreign channels). Intuitively, the local value is faulty if it differs too much (i.e., more than a constant CROSS_CH_TOL_ROLL) from all the other (supposedly correct) values. Moreover, a cross-channel failure is reported only if this situation lasts for some delay (TIME_CROSS_ROLL). This detection is performed by a node values_nok, called by the component FailDetect of the channel and which is given below, along with a description in the sequel.:

```
node values_nok (
    pfother1, pfother2, pfother3 : bool ;
    -- foreign values status: true if faulty
    xi : real ; -- local value
    pxother1, pxother2, pxother3 : real ;
    -- foreign values
)
returns (
    fault : bool
    -- there is a cross channel fault
)
var
    diff1, diff2, diff3 : bool ;
    -- comparisons of xi with the three
    -- foreign values
let
    diff1 = abs (xi - pxother1)
                        > CROSS_CH_TOL_ROLL ;
    diff2 = abs (xi - pxother2)
                        > CROSS_CH_TOL_ROLL ;
    diff3 = abs (xi - pxother3)
                        > CROSS_CH_TOL_ROLL ;
    fault =
      maintain(TIME_CROSS_ROLL,
         if pfother1 then
         -- don't take this one into account
           if pfother2 then -- the same
             if pfother3
               then false else diff3
         else if pfother3 then diff2
                    else (diff2 and diff3)
         else if pfother2 then
               if pfother3 then diff1
                  else (diff1 and diff3)
         else if pfother3 then
               (diff1 and diff2)
         else (diff1 and diff2 and diff3)) ;
tel
```

In Section VII-A, this node will be formally checked for equivalence with another version.

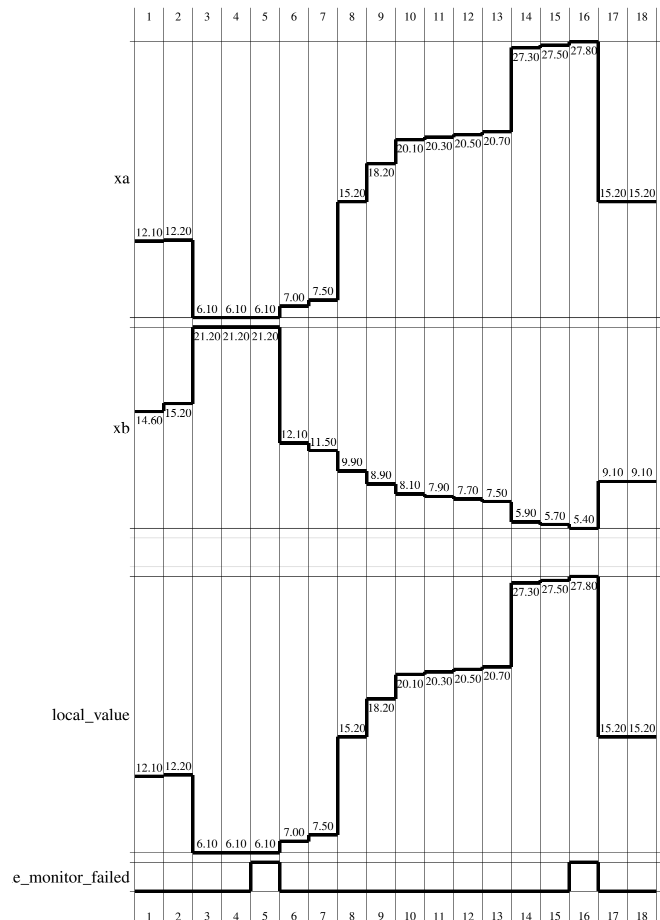

Fig. 10 : A simulation of the Monitor

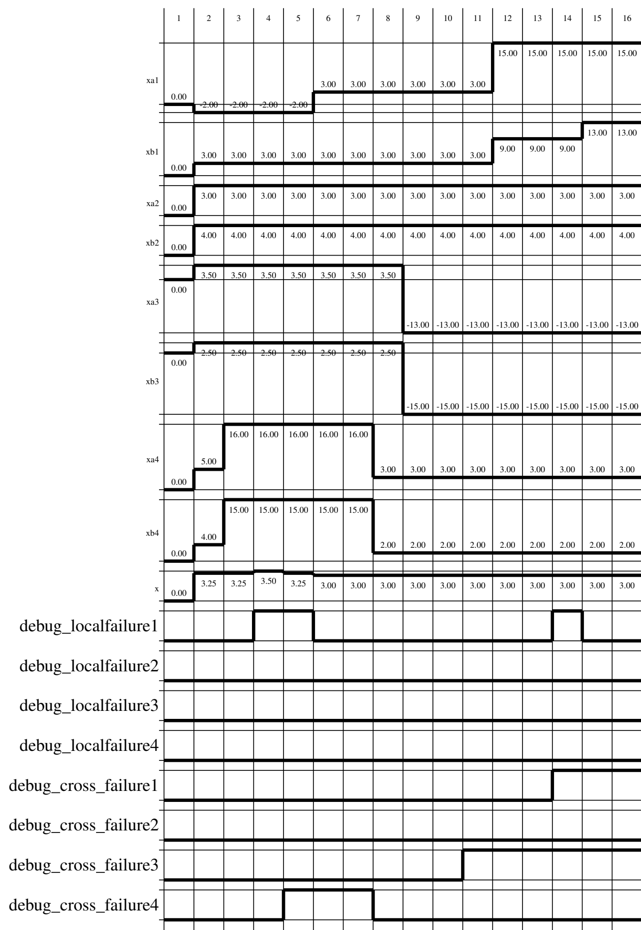

Fig. 11 : A simulation of transmission and cross failures

Notice that the informal specification is unclear: we decided to report a failure when the conjunction "`diff1` and `diff2` and `diff3`" holds for the given delay. We could have chosen another solution, first detecting if each foreign value differs from the local one for the given delay, and then reporting the conjunction of these conditions.

Whatever be the correct choice, the body of the node `FailDetect` is simply:

```
failure = transmit_failure or cross_failure;
cross_failure =
   values_nok(pfother1, pfother2, pfother3,
          xi, pxother1, pxother2, pxother3);
```

Simulations

We can run a simulation on this program, using Luciole, the Lustre simulator. Since the current version of Luciole only shows the values of input/output variables, while it is interesting to see also the internal variables corresponding to failures (`transmit failure`$_i$; `cross failure`$_i$), we first modify the program so that these variables be output. The simulation is performed with the following values for relevant constants:

```
DELTA_ROLL = 4.0 ;
CROSS_CH_TOL_ROLL = 10.0 ;
TIME_ROLL = TIME_CROSS_ROLL = 3 ;
```

Fig. 11 shows an execution (where we introduce first a transmission fault on the first channel (steps 2 to 5), then a cross failure on channel 4 (steps 3 to 7), then a cross failure on channel 3 alone (from step 9), on which an other cross failure, on channel 1, is combined (from step 12)).

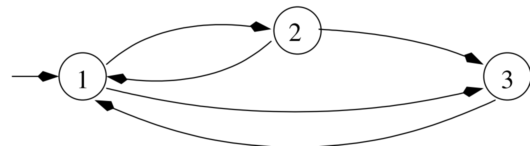

Fig. 12 : The automaton driving failure latching

C. Latching the cross-channel faults

It may be noted that some serious faults must be latched (i.e., sustained even when their cause has disappeared), until the occurence of some reset command. This latching is performed by the node `FailDetect`, and depends on the state of an automaton, as described by Fig. 12:

- State 1 is the normal state: everything is working properly, or there are transmission faults from time to time, which are not latched;
- In state 2, there has been at least one serious fault in the past, which is latched, but may be reset either by "`OnGroundReset" or by "InAirReset`";
- In state 3, there have been several serious faults in the past, or one serious fault followed immediately by a transmission one, and the fault is latched *and* reset inhibited. In this case, only the "`OnGroundReset`" can restore the normal state.

The `failure` output is true when in state 2 or 3, and is equal to `transmit_failure` when in state 1. The transitions between these states are driven by the following conditions:

- from state 1: if the locally computed variable involves a cross-failure, it is considered more serious if its value is "in the nominal range" (since it can be later considered as correct). So, in this case, one moves to state 3. If the erroneous value lies outside the nominal range, one moves to state 2.
- from state 2: any reset signal restores the normal state (1); a cross-failure immediately followed by a foreign failure involves a move to state 3; a transmission failure occurring in state 2 also involves a move to state 3.

from state 3: only the "`OnGroundReset`" can restore the normal state.

Lustre code

Programming such an automaton in Lustre is tedious,

but rather systematic. The state is encoded by an integer variable, ranging from 1 to 3. Notice that the order in which the transition conditions are tested for is relevant: for instance, it allows priority to be given to `reset`:

```
state = 1 ->
   if pre(state)=1 then
     if pre(reset) then 1
        -- reset has priority
     else if pre(from1to2) then 2
     else if pre(from1to3) then 3 else 1
   else if pre(state)=2 then
     if pre(from2to1) then 1
     else if pre(from2to3) then 3 else 2
   else --  pre(state)=3
     if pre(from3to1) then 1 else 3 ;
```

The definitions of transition conditions follow the specification:

```
from1to2 = cross_failure and
             not InNominalRange (xi) ;
from1to3 = cross_failure and
             InNominalRange (xi) ;
from2to3 = (pre(cross_failure) and
                        foreign_failure)
                or transmit_failure ;
from2to1 = inairreset or ongroundreset;
from3to1 = ongroundreset;
```

Fig. 13 : shows the whole code for the node `FailDetect`.

D. Taking into account the requirement on third fault

In the real system, a special device — called "Global Allocator" — is in charge of preventing more than two units from becoming reset-inhibited (i.e., from entering state 3). The node `Fail-Detect` must be changed as follows: whenever the automaton should move to state 3, it sends a request, say r, to the global allocator, and only performs the move if the allocator sends it an authorization back, say a. The node has an additional Boolean input a, and sends an additional Boolean output r. The transitions are then modified as follows:

- The request is the disjunction of the condition which involved a move from state 1 to state 3, and the one which involved a move from state 2 to state 3.

```
r = false -> (pre(state) = 1 and try1to3)
          or (pre(state) = 2 and try2to3);
```

where `try1to3 and try2to3` obey the previous definitions of `from1to3 and from2to3`:

```
try1to3 = cross_failure and InNominalRange(xi);
try2to3 = (pre(cross_failure)
             and foreign_failure)
        or transmit_failure ;
```

The actual moves occur only when the authorization is given:

```
from1to3 = try1to3 and a;
from2to3 = try2to3 and a;
```

The global allocator is quite simple: it receives requests r_i from the units, together with the "`OnGroundReset`" signal, and returns authorizations. An internal counter nb is used to count the number of units that are "`reset-inhibited`", and authorizations are given in order to prevent this counter from reaching 3. The "`OnGroundReset`" signal resets the allocator in its initial configuration (because it causes all automata to leave state 3)[4].

```
node allocator(r1,r2,r3,r4,reset: bool)
returns (a1,a2,a3,a4: bool);
var nb_aut, already: int;
let
   already = if (true -> reset)
                       then 0 else pre(nb_aut);
   a1 = r1 and already <= 1;
   a2 = r2 and
        ((not r1 and already <= 1)
         or (r1 and already = 0)
        );
   a3 = r3 and
        ((not r1 and not r2  and already <= 1)
         or (#(r1,r2) and already = 0)
        );
   a4 = r4 and
        ((not r1 and not r2 and not r3 and
                                already <= 1)
        or (#(r1,r2,r3) and already = 0)
        );
   nb_aut = if (true -> reset) then 0
            else pre(nb_aut) +
                        (if a1 then 1 else 0) +
                        (if a2 then 1 else 0) +
                        (if a3 then 1 else 0) +
                        (if a4 then 1 else 0) ;
tel
```

Notice that there is an "instantaneous dialogue" between the units and the allocator: in the very same step, the unit asks the allocator for an authorization, the allocator replies, and the unit takes the transition or not, according to this reply.

In Section VII-B, some properties of this allocation mechanism will be formally verified.

VII. Some Experiences in Formal Verification

Ideally, the formal verification of such a program should consist of comparing it with a global, abstract specification. As it is often the case with real case-studies, this specification is not available. The problem even more serious, here, since the abstract specification of such a fault-tolerant system

[4] The "#" operator, in Lustre, is a n-ary Boolean operator, which returns "true" if and only if at most one of its operand is true.

```
node FailDetect (
   transmit_failure : bool ;
   xi : real ;
   ongroundreset, inairreset : bool ;
   choffi : bool ;
   pxother1, pxother2, pxother3 : real ; -- other values (pre)
   pfother1, pfother2, pfother3 : bool ; -- other failures (pre)
)
returns (
   failure : bool ; -- failure detected by this channel
)
var
   cross_failure : bool ;
   state : int ; -- only 1, 2, 3 are relevant
   from1to2, from1to3, from2to3, from2to1, from3to1 : bool ;
   reset, foreign_failure : bool ;
let
   -- the state ------------------------------------------------------
   state     = 1 ->
          if pre(state)=1 then
             if pre(reset) then 1     -- reset has priority
             else if pre(from1to2) then 2
             else if pre(from1to3) then 3
             else 1
          else if pre(state)=2 then
             if pre(from2to1) then 1
             else if pre(from2to3) then 3
             else 2
          else -- pre(state)=3
             if pre(from3to1) then 1
             else 3 ;
   reset = ongroundreset or (inairreset and not cross_failure) ;
   foreign_failure = pfother1 or pfother2 or pfother3 ;
   -- The output -----------------------------------------------------
   failure = (state = 2) or (state = 3) or (state = 1 and transmit_failure) ;
   -- All the transitions --------------------------------------------
   from1to2 = cross_failure and not InNominalRange (xi) ;
   from1to3 = cross_failure and InNominalRange (xi) ;
   from2to3 = (pre(cross_failure) and  foreign_failure) or transmit_failure ;
   from2to1 = reset ;
   from3to1 = ongroundreset ;
   -- Cross channel comparisons --------------------------------------
   cross_failure = values_nok (pfother1, pfother2, pfother3,
                               xi, pxother1, pxother2, pxother3) ;
tel
```

Fig. 13 : Latching Failures

should probably involve probabilistic properties, which are stated nowhere, and which could not be handled by usual verification tools.

So, we don't know "what to verify" on the complete program. However, there are two common cases where verification tools can be applied "locally":

- When there are two ways of writing a node, one can write both and try to show that they are equivalent. If they are not, a bug is detected, at least, in one of them. If they are, this increases the confidence one can have in any of them. We will illustrate this situation on two versions of the node `values_nok`, which detects cross-channel faults.
- When some consistency properties are clearly expressed in the requirements: for instance, we will try to prove that at most two channels can become "reset-inhibited".

A. Cross-channel fault detection

In the version of the node `values_nok` given in Section VI-B, the output `fault` is defined as `maintain(TIME_CROSS_ROLL, cond)`, where cond is a Boolean expression carefully detailing all the possible combinations of failures. One could look for a more compact and symmetrical condition, say cond1, expressing that all the significant other measures are too different from the local measure:

$$\text{cond1} = \forall i \in [1...3];\ (\neg \text{pfother}_i))\ \text{diff}_i$$

or

```
fault = maintain(TIME_CROSS_ROLL,
                 pfother1 or diff1)
                 and (pfother2 or diff2)
                 and (pfother3 or diff3));
```

So, we can write two complete versions of the node `values_nok`, with the same definition for the diff_i and a

different definition for `r`. Then, we try, using our verification tools, to show that, whatever be the input sequences, they provide the same output sequence:

- with the verification tool Lesar, the verification fails. The diagnosis returned by Lesar shows that it is due to the weakness of the tool: Lesar considers only the Boolean aspects of the program, and abstracts away all the numerical expressions. As a consequence, it produces a counter-example where the Boolean variables `diff1`, `diff2`, and `diff3` (which are defined by numerical expressions) appearing in the two versions of the node have distinct values. It is a case of *false negative*.
- the prototype NBac is able, to some extend, to take into account numerical aspects in the verification. It also fails in proving the equivalence of the two nodes, but indicates that the output may differ when all the inputs $pfother_i$ are true. This is an actual discrepancy: in this case, the first version outputs `fault = false` (since none of other values is significant, the local one is assumed to be good), while the second version outputs `fault = true`. It is probably a bug in the second version, which can be fixed as follows:

```
fault = maintain(TIME_CROSS_ROLL,
       pfother1 or diff1)
    and (pfother2 or diff2)
    and (pfother3 or diff3)
    and not(pfother1 and pfother2
    and pfother3));
```

Now, NBac shows that this fixed version is equivalent to the first one.

B. Allocation of "reset-inhibition"

In section VI-D we designed the "Global Allocator", the role of which is to prevent more than two channels to become "reset-inhibited. We can try to verify the behavior of the allocator alone: we know that `nb_aut`, the number of authorizations, should stay smaller than 2. Of course, because of numerical variables, Lesar is not able to prove this property (the range of the counter being small, the node could be rewritten only with Boolean variables, but it is neither very natural, nor efficient). NBac proves the property very easily (in about 0.5 sec.).

A more ambitious verification consists in proving, on the integrated system, that at most two units can become resetinhibited, i.e., that not only are the authorizations correctly delivered, but also that they are correctly obeyed by the units.

The current version of NBac is not able to perform this verification, for two reasons:

- on one hand, there are too many numerical variables, which makes the symbolic computations very complex. An interesting remark is that the variables corresponding to "roll" measures, while being used to determine failures, have no real influence on the property; the tool is not able to detect this fact, and to "slice" these variables away.
- on the other hand, the determination of the suitable control structure is quite complex. Obviously, the automata involved in "FailDetect" should be taken into account, but the tool takes a very long time to find it.

These limitations suggest some improvements to the tool, which will be discussed in the conclusion.

So, we decided to use Lesar for this verification. For that, we have to modify the program, so that the property involves only Boolean computations. Since it only uses counters up to 4, it is quite easy:

- "FailDetect" must be changed to work only with Boolean variables: instead of encoding states by integers, we use pairs of Booleans, and we define a node to compare such pairs for equality:

```
const state1 = [false, true];
      state2 = [true, false];
      state3 = [true, true];
node EQState (s1, s2: [bool,bool])
                returns (eq: bool);
let
   eq =(s1[0]=s2[0]) and (s1[1]=s2[1]);
tel
```

Then, we change the definition of the state as follows:

```
state= state1 ->
  if EQState(ps,state1) then
     if pre(reset) then state1
        else if pre(from1to2) then state2
        else if pre(from1to3) then state3
        else state1
     else if  EQState(ps,state2) then
        if pre(from2to1) then state1
        else if pre(from2to3) then state3
        else state2
     else
        if pre(from3to1) then state1
        else state3 ;
```

- The property must be expressed only with Boolean variables: the observer receives the states of the channels, and returns a single Boolean which is false when at least three of them are in state 3.

```
node verif(st1, st2, st3, st4:[bool,bool])
    returns (ok: bool);
var three: bool;
    inhib1, inhib2, inhib3, inhib4: bool;
let
 -- at most 2 channels  reset inhibited
   ok = not three;
 -- counting the number of inhibited units
  three =(inhib1 and inhib2 and inhib3) or
         (inhib1 and inhib2 and inhib4) or
         (inhib1 and inhib3 and inhib4) or
         (inhib2 and inhib3 and inhib4);
  inhib1 = EQState(st1,state3);
  inhib2 = EQState(st2,state3);
  inhib3 = EQState(st3,state3);
  inhib4 = EQState(st4,state3);
tel
```

Lesar proves this property in 69 sec.

An other interesting property is that the allocator allows a maximum number of channels to become reset-inhibited. In other words, we want that, whenever at least two channels have requested to become reset-inhibited since the last "*On-GroundReset*", then at least two of them (in fact, exactly two, because of the previous property) are actually reset-inhibited. This property can be expressed by the following observer: it receives the requests from channels to become reset-inhibited, the *OnGroundReset*" signal, and the state of the channels. The requests are memorized from each reset signal. Then the output is true iff whenever there is at least two memorized requests, at least two channels are in state 3 (remember that, in Lustre, `#(x1,x2,...,xn)` is a Boolean expression which is true iff at most 1 of its Boolean arguments x_i is true; so, its negation expresses that at least two of them are true):

```
node verif(ask1, ask2, ask3, ask4, reset: bool;
             st1, st2, st3, st4: [bool, bool])
returns (ok: bool);
var morethantwoasks : bool;
    req1, req2, req3, req4: bool;
     inhib1, inhib2, inhib3, inhib4: bool;
let
  -- if more than 2 requests, then
  -- at least 2 channels in state 3
  ok = (morethantworeqs =>
        not #(inhib1, inhib2, inhib3, inhib4));

  inhib1 = EQState(st1,state3);
  inhib2 = EQState(st2,state3);
  inhib3 = EQState(st3,state3);
  inhib4 = EQState(st4,state3);

  req1 = if (true -> reset) then false
         else pre(req1) or ask1;
  req2 = if (true -> reset) then false
         else pre(req2) or ask2;
  req3 = if (true -> reset) then false
         else pre(req3) or ask3;
  req4 = if (true -> reset) then false
         else pre(req4) or ask4;

  morethantworeqs =
                 not #(req1, req2, req3, req4);
tel
```

Lesar finds that this property is false. This is due to the fact that requests only produce state changes *at the next step*. The correct property must be written:

```
ok = true ->
    (pre(morethantworeqs) =>
      not #(inhib1, inhib2, inhib3, inhib4));
```

Now, Lesar finds, in 130 sec., that the property is satisfied.

VIII. Conclusion

We presented a real case study[5] described in Lustre, and validated with associated tools. The study throws light on several aspects: the language itself, the design methodology, and the validation tools.

The study demonstrates that case study is quite wellsuited for Lustre. A similar experiment performed with Esterel [RS00], [SS03] showed that a data-flow language is much more natural for most features of the system. Now, we restricted ourselves to using only the kernel of Lustre which is compatible with the industrial tool Scade. In the full academic version of Lustre, we also have a powerful notion of arrays, the use of which would have significantly reduced the size of the description: all data symmetrically processed in each channel should be structured in arrays. This usefulness of arrays, both concerning the structure of the source program, and the quality of the generated code, was confirmed by other case studies. The most imperative part of the system is the automata used in `FailDetect` for defining reset-inhibition. While the description of such small automata is not difficult in Lustre, it would be obviously easier with an extension based on automata [MR03].

Concerning programming methodology, we have tried to promote a progressive approach: the whole architecture is designed first, and its components are progressively detailed in turn. The interest is that an integrated — yet still incomplete —, version of the program is always available for simulation and validation. This approach completely differs from the "progressive refinement" generally advocated (e.g., in B [Abr95]), where a non-deterministic specification is progressively refined (i.e., made more deterministic) into a program. Here, since we use a programming language, all our descriptions are deterministic. So the design proceeds in the reverse direction: an initial, very simplified description, is progressively enriched until it matches the whole specification. Of course, this pragmatic approach is less "formal", but we believe that it is more realistic, since it better meets engineers uses. Also, because of the weaknesses of formal validation tools, in their present state, early error detection by early simulation — which is allowed by our approach — seems to be more practical than early proof.

Finally, it was an interesting challenge for our validation tools. We showed, first, that the Luciole simulator is extremely useful, at each stage of the development, to quickly check either the whole program or a single node. Concerning verification, we faced the usual problems due to lack of global specification. However, some properties were available. A first remark is that proving properties directly on the source program requires, in most cases, a verification tool with some numerical capabilities. Such capabilities are provided by NBac, but the size of problems that it can handle must be increased. The case study suggests some ways in which the user can help NBac: as mentioned already, there should be a way of abstracting away some Boolean variables whose dependences on numerical values obviously don't influence the validity of the property; it was the case, in our example, of the failure detections. On the other hand, the user should

[5] For concision and confidentiality reasons, the case study was slightly simplified in the paper, but it is still representative of the complexity of actual system.

be able to suggest an initial control structure; for instance the automata used to determine the reset-inhibition were obviously relevant to the verification. We were not able to apply our automatic testing tool to this example, mainly because of missing knowledge, both about the assumptions on the environment and about the global properties to be checked on the whole system. This is a challenging area of pursuit.

References

[Abr95] J.-R. Abrial. *The B-Book*. Cambridge University Press,

[BB91] 1995. A. Benveniste and G. Berry. Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), September 1991.

[BCE+03] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, pages 428–439, June 1990.

[BS91] F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Cas01] P. Caspi. Embedded control: From asynchrony to synchrony and back. In *1st International Workshop on Embedded Software, EMSOFT2001*, Lake Tahoe, October 2001. LNCS 2211.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CGP99] P. Caspi, A. Girault, and D. Pilaud. Automatic distribution of reactive systems for asynchronous networks of processors. *IEEE Transactions on Software Engineering*, 25(3):416–427, 1999. Research report INRIA 3491.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, Munich, January 1987.

[CMSW99] P. Caspi, C. Mazuet, R. Salem, and D. Weber. Formal design of distributed control systems with Lustre. In *Proc. Safecomp'99, volume 1698 of Lecture Notes in Computer Science*. Springer Verlag, September 1999.

[CPP05] Jean-Louis Colac¸o, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software* (EMSOFT' 05), Jersey city, New Jersey, USA, September 2005.

[HB02] N. Halbwachs and S. Baghdadi. Synchronous modeling of asynchronous systems. In *EMSOFT'02*. LNCS 2491, Springer Verlag, October 2002.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology*, AMAST'93, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of realtime systems using linear relation analysis. Formal Methods in System Design, 11(2):157–185, August 1997.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Fil´e, editors, Static Analysis Symposium, SAS'99, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.

[JHR+07] E. Jahier, N. Halbwachs, P. Raymond, X. Nicollin, and D. Lesens. Virtual execution of AADL models via a translation into synchronous programs. In *EMSOFT 2007*, Salzburg, Austria, 2007.

[JRB04] E. Jahier, P. Raymond, and P. Baufreton. Case studies with Lurette V2. In *First International Symposium on Leveraging Applications of Formal Method, ISoLa 2004*, Paphos, Cyprus, October 2004.

[MG00] F. Maraninchi and F. Gaucher. Step-wise + algorithmic debugging for reactive programs: Ludic, a debugger for lustre. In *AADEBUG'2000 – Fourth International Workshop on Automated Debugging*, Munich, aug 2000.

[MR03] F. Maraninchi and Y. R´emond. Mode-automata: a new domainspecific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous dataflow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[RS00] B. Rajan and R. K. Shyamasundar. A fault tolerant distributed system in Multiclock Esterel. In *FORTE/ PSTV*, pages 301–316. North Holland, October 2000.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[SBT96] T. R. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *International Design and Testing Conference IDTC'96*, Paris, France, 1996.

[SC04] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro conference on Real-Time Systems (ECRTS'04)*, Catania, Italy, June 2004.

[SS03] A.D. Shabbir and R. K. Shyamasundar. Specification of fault tolerant Gyroscopic controller in Esterel. In *Internal Report of External Funded Project*. TIFR, 2003.

About the Authors

Florence Maranichi, is a Professor, Grenoble INP / ENSIMAG (Director of International Relations and is in charge of the "Embedded Software and Systems" master curriculum) at VERIMAG laboratory (Head of the group Synchronous Languages and Reactive Systems)

Dr. Nicolas Halbwachsis Directeur de Recherche at CNRS, and Director of Verimag Laboratory

Dr. Pascal Raymond is Chargé de Recherche au CNRS.

Dr. Catherine Parent-Vigouroux is an Associate Professor at University Joseph Fourier, Grenoble

Prof. R. K. Shyamasundar, Fellow IEEE, Fellow ACM is a Senior Professor and JC Bose National Fellow at TIFR.