Using formal verification techniques to reduce simulation and test effort

O. Laurent[1], P. Michel[2], V. Wiels[2]

[1] Aerospatiale-Matra-Airbus, A/BTE/SY/MS
316 route de Bayonne,
31000 Toulouse cedex 03, France
odile.laurent@airbus.aeromatra.com
[2] ONERA-CERT/DTIM
BP 4025, 2 avenue E. Belin,
31055 Toulouse Cedex 4, France
{michel,wiels}@cert.fr

Abstract. This paper describes an experiment in using formal methods in an industrial context. The goal is to use formal verification techniques in order to alleviate the simulation and test activities. The application is a flight control computer of the Airbus A340.

Introduction

Aerospatiale has used formal methods for several years in the development of embedded systems for Airbus and it has already significantly increased productivity. Embedded systems are specified using SCADE [8] a commercial tool based on the Lustre language [1, 3]. The SCADE environment provides a qualified automatic code generator, so there is no gap between the specification and the embedded code. This eliminates the need for unit test: only the specification needs to be validated. The development cycle is no more a "V", but a "Y". This use of formal methods thus already implies cost savings.

But it is possible to go further. For the moment, specifications are validated using simulation and test. The goal of the project presented in this paper is to study whether formal verification techniques could be used to verify some properties on these specifications in order to reduce the simulation and test phases which are still very costly. Several tools exist that allow verification of Lustre specifications and we would like to study the feasability of using these tools in an industrial context.

Section 1 presents the case study chosen for the project and the methodology. Section 2 describes the first part of the project which has consisted in identifying and formalising properties of the system. In section 3, we present the ongoing experiments of verification of such properties. Section 4 gives conclusions and suggests avenues for future work.

1 Methodology and case study

The case study we are working on for this project is the Flight Control Secondary Computer (FCSC) which is part of the A340 flight control system. The flight control surfaces of an Airbus A340 are all electrically controlled and hydraulically activated. The sidesticks are used to fly the aircraft in pitch and roll. The flight control system computes the order to be applied to the surfaces from the pilot inputs and the current state of the aircraft. Moreover, regardless of the pilot's input, computers will prevent excessive maneuvers and going beyond the safe flight envelope.

For safety reasons, the flight control system presents several redundancy levels:

– Reconfiguration redundancy by duplication of the computers: when an active computer detects a failure, it is replaced by the following one in a reconfiguration list. Each aircraft control surface has its own reconfiguration list, the number and the order of the involved computers depends on the surface.
– Conceptual redundancy. Two kinds of computers have been developed: Flight Control Primary Computer (FCPC) and Flight Control Secondary Computer (FCSC). The A340 flight control system is composed of 3 FCPCs and 2 FCSCs.

FCPCs have been defined to control all kinds of surfaces except flaps and slats. An FCSC can control all kinds of surfaces except flaps, slats and Trimable Horizontal Stabilizer (THS). It uses laws generated by an FCPC in a normal configuration, but when this fails, it generates direct pitch, roll and yaw laws itself. The computer operates as long as the aircraft is electrically energized.

Figure 1 shows the surfaces managed by the flight control system.

The FCSC specification consists of approximately 600 SCADE sheets, dealing with validation of inputs, logic, flight laws and servocommands.

In the current development process, the code is automatically generated from the SCADE specifications and validation is done by simulation and test. Two major steps are carried out in the validation process:

– tests in a simulation environment to verify the design of the system,
– tests on the iron bird with the target computers to validate the system within the real environment.

The goal of our study is to replace or alleviate tests performed during the simulation phase of the validation.
Our approach is broken down into three steps:

– identification of properties that are to be satisfied,
– expression of these properties in logic,
– verification.

The methodology must allow:

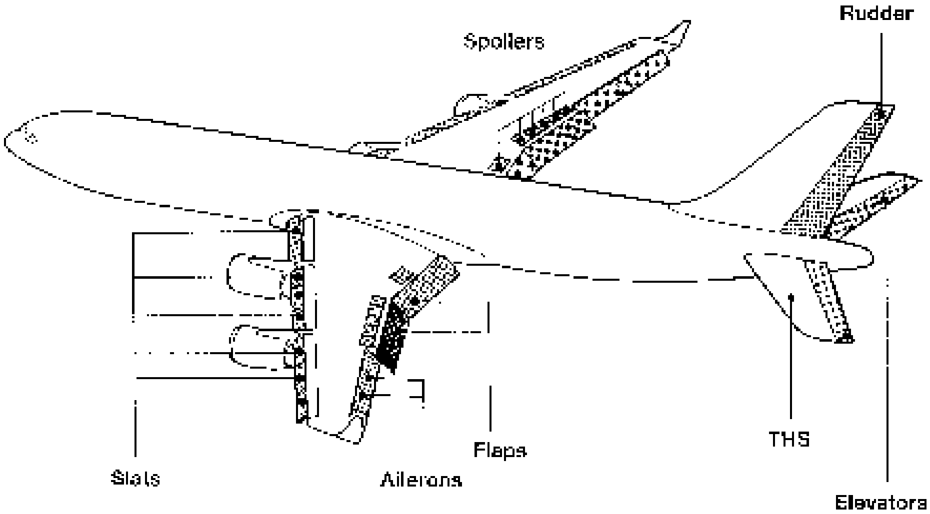

Fig. 1. Plane surfaces

- identification of properties on the inputs and outputs of the system or of a main part of the system which will be called subsystem. Indeed, we want to be able to express properties without entering into the detailed description of *how* the system works.
- automatization of the process.

The first two steps of the approach are described in the next section, and the third one in section 3.

2 Identifying and expressing properties

2.1 Identification

Properties have been identified by two means: discussions with designers of the system and study of different design and validation documents such as Safety Requirement Document, Verification and Validation plan or System Descritpion Notes.

This step is a difficult one. It is hard to identify the relevant properties for the system which address inputs and outputs. Detailed discussions with the system designers are essential to get good understanding of the more critical objectives of the system. Moreover many documents contain precise scenarios that are used to simulate and test the system. An issue is then to extract, from a list of scenarios, the underlying property that is in fact the test objective.

Classes of properties. We have identified three classes of properties:

– properties related to value domains,
– functional safety properties,
– system reconfiguration properties.

The first class of properties is concerned with precision constraints, surveillance (to control that values stay in a predefined domain) and error estimation.

The properties in the second class characterize the correct functional behaviour of the different elements of the system.

Finally, the third class is broken down into two sub-classes: robustness properties, which describe the consequences of external events on the system and redundancy properties.

Examples. In this paper, we consider two examples of properties. The first one is a functional safety property concerning the sidesticks. The informal expression of the property (which was given to us by a designer) is the following:

P1: *if we suppose that the two sidesticks cannot be faulty at the same time, there is always an active stick.*

The second one is a redundancy property on inner ailerons. The reconfiguration list for these surfaces mentions two primary computers PC1 and PC2 followed by two secondary computers SC1 and SC2. If we focus on the first FCSC, the redundancy property can be expressed, in a first attempt, as :

P2: *if the two primary computers PC1 and PC2 are faulty, then the secondary computer SC1 becomes operational.*

2.2 Formal expression of the properties

The next step is to find a formal expression of the informal properties that were identified. This step is also difficult and requires good knowledge of the application.

P1. In the case of P1, the difficulty is to define what an "active" sidestick means. There are two sidesticks (pilot and copilot) and a system of priorities (one sidestick can take priority over the other). The first idea we had was to define "active" in terms of transmission of order: the order from the pilot or the order from the copilot will indeed be transmitted to the surfaces. So we stated:

$$\begin{aligned} P1 : &(PILOT\_ORDER \neq 0) \vee (COPILOT\_ORDER \neq 0) \\ &\rightarrow (TRANSMITTED\_ORDER \neq 0) \end{aligned}$$

that is to say if the pilot order is different from 0 or the copilot order is different from 0 then the transmitted order is different from 0.

But this expression is not correct and we detected it by looking at the counter-example found when we first tried to verify the property: the tool gave us a case where one sidestick was faulty, the other was operational but with value 0. The property was false while it is a valid case. The problem is that a correct order from an active sidestick can be 0.

If we want to express the property in terms of transmission of order, we would have to go into the specification more thoroughly (because of course the value of the transmitted order is not exactly the value of the input order) and this would be in contradiction with our methodology. So we tried to express the property in a different way and we obtained:

$$\begin{aligned} P1: \neg((&PILOT\_PRIORITY \wedge PILOT\_STICK\_FAULT) \vee \\ &(COPILOT\_PRIORITY \wedge COPILOT\_STICK\_FAULT)) \end{aligned}$$

which means it is never true that the pilot has priority and his sidestick is faulty or that the copilot has priority and his sidestick is faulty.

This example illustrates the difficulty of going from an informal text to a formal expression.

P2. If we look more carefully at the property P2, we can see that the formulation given in section 2.1 is incomplete. Two implicit but important features are hidden in this formulation:

- Firstly, we can be more precise, saying that when the primary computers PC1 and PC2 are faulty, secondary computer SC1 becomes operational only "if it is possible". If SC1 cannot become operational, then secondary computer SC2 is requested and becomes operational if it is possible. If not, there is a major problem for the aircraft but the probability of simultaneous failure of four computers is very small (less than 10^{-9} per hour).
- Secondly, the property P2, as formulated in §2.1, is a conditional expression without a counterpart (i.e. without else part). But it is obvious for the designers that secondary computer SC1 cannot be operational if the two primary computers PC1 and PC2 are not simultaneously faulty. This counterpart has to be expressed in a rigorous approach.

So, taking into account these two observations and using a semi-formal notation, we can give a more accurate formulation of P2:

```
P2: if (the two primary computers PC1 and PC2 are faulty)
    then if (possible)
         then (the secondary computer SC1 becomes operational)
         else (the secondary computer SC2 is requested)
    else (the secondary computer SC1 is not operational)
```

It is important to note here that this property concerns only one kind of surface, i.e. the inner ailerons, and when we read "SC1 is not operational", we must understand that SC1 is not operational for the inner ailerons, but can be operational for another kind of surface.

This property is now quite complex and should be broken down into several sub-properties. We will focus here on the "else" part which involves fewer propositions and is easier to express than the direct part. If it is expressed alone, the "else " part gives the P2b sub-property:

```
P2b: if not (the two primary computers PC1 and PC2 are faulty)
     then not (the secondary computer SC1 is operational)
```

which can be formulated in the following simpler reverse way:

```
P2b: if (the  secondary computer SC1 is operational)
     then (the two primary computers PC1 and PC2 are faulty)
```

For this property, we have to express the behavior of several computers from the point of view of one of them. Inside the secondary computer SC1, we have to express the following two propositions: " secondary computer SC1 is operational" and "the two primary computers PC1 and PC2 are faulty".

SC1 can check to be operational by the fact that two of the gates it manages are closed. In the SC1 specification, these gates are represented by two output parameters G1 and G2.

Two boolean input parameters of computer SC1, respectively $PRIMARY\_COMPUTER1\_OK$ and $PRIMARY\_COMPUTER2\_OK$ express that computers PC1 and PC2 operate correctly. The second proposition is then represented by a conjunction on the negations of these two booleans.

So we obtained the following formulation for P2b:

$$P2b: \; ((G1 = 1) \wedge (G2 = 1)) \rightarrow$$
$$((\neg PRIMARY\_COMPUTER1\_OK) \wedge (\neg PRIMARY\_COMPUTER2\_OK))$$

3 Verifying properties

We first briefly describe SCADE (based on the Lustre language) and the observer technique which is the usual way of proving properties of Lustre specifications. Then we explain the practical problems we have to solve in order to implement verification. Finally, we give the current results of the project.

3.1 Formal verification in Lustre

SCADE. The SCADE environment [8] was defined to assist the development of critical embedded systems. This environment is composed of three tools:

- a graphical editor,
- a simulator,
- a code generator that automatically translates graphical specifications into C code.

The SCADE language is a graphic data flow specification language that can be translated into Lustre [1, 3].

Lustre. Lustre is a synchronous data flow specification language. A system is described by several interconnected nodes. A node computes values for its outputs from values of its inputs. A node is defined by a set of equations and assertions. An equation `VAR=EXPR` specifies that the variable `VAR` is always equal to `EXPR`. An assertion `assert BOOLEXPR` means that the boolean expression `BOOLEXPR` is assumed to be always true during the execution of the program.

Any variable or expression is considered to represent the sequence of values it takes during the whole execution of the system and Lustre operators operate globally over these sequences. For example, `VAR=EXPR` means that the sequences of values associated with `VAR` and `EXPR` are identical.

Expressions are made of variable identifiers, constants, usual arithmetic, boolean and conditional operators and two specific temporal operators: previous `pre` and followed-by `->`.

- If `E` is an expression denoting the sequence $(e_0, e_1, e_2, ...)$, then `pre(E)` denotes the sequence $(nil, e_0, e_1, ...)$ where *nil* is an undefined value.
- If `E` and `F` are two expressions of the same type respectively denoting the sequences $(e_0, e_1, e_2, ...)$ and $(f_0, f_1, f_2, ...)$, then `E -> F` is an expression denoting the sequence $(e_0, f_1, f_2, ...)$.

Synchronous observers. Properties are also expressed in Lustre. Verification is done using synchronous observers [2]. Synchronous observers are used to express invariant properties, i.e. properties that should always hold. The idea behind synchronous observers is to add an extra output to the system which states if the property holds at each time instant.

More precisely, the observed system S is considered as a black box and only its interface (i.e. the input and output variables) can be involved in the formulation of a property P. In order to prove that a property P is satisfied by a system S under a set of hypotheses H, we build a system S' by composition of S, P, H as shown on figure 2.

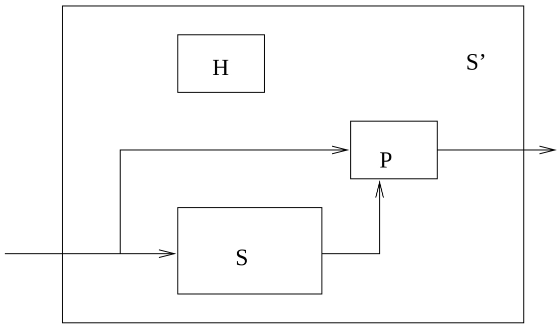

Fig. 2. Observer for a property P on a system S.

The only output of S' is the boolean value of P. The verification then consists in checking that output of system S' is always true.

Verification tools for Lustre. Lesar [6, 9] is the model checker associated with Lustre. Lesar can apply three verification techniques: an exhaustive enumeration of the states of the system, similar to standard model checking; a symbolic construction of the set of states which satisfy the property, analogous to symbolic model checking; and a symbolic construction of the set of states violating the property.

Another verification tool for Lustre is available: NP-TOOLS with the Lucifer translator [5, 4]. NP-TOOLS is a general purpose formal verification toolbox for combinatorial circuits based on Stalmarck's method [7]. Lucifer is a translator from Lustre to NP-TOOLS. The technical difficulty in the translation is that Lustre models dynamic systems, i.e. systems whose outputs depend on the current inputs and all previous inputs while NP-TOOLS deals with static systems whose outputs depend only on the current inputs. The dynamics of Lustre models has to be modelled in NP-TOOLS. This is done by representing each Lustre node as an NP-TOOLS macro with additional inputs corresponding to the memory of the node. For example, an edge detector is specified in Lustre as follows:

```
node edge(x: bool) returns (e: bool);
let
e = false -> x and not pre x;
tel
```

It uses two temporal operators: `pre` and `->`. It is modelled in NP-TOOLS by adding two inputs: `pre(x)` representing the memory of x, and `init` a boolean that is only true at the initial time. In fact, the NP-TOOLS circuit represents the transition function of the corresponding Lustre node. To model a dynamic Lustre system, several instances of the NP-TOOLS circuit can be connected. The `init` boolean is only true in the first instance. The output `x` of the first instance is connected to the input `pre(x)` of the second one, and so on. This unrolled specification can then be used to verify the properties of the system.

We plan to experiment both tools in the project. We have mostly used Lucifer and NP-TOOLS up to now and have just begun experimenting with Lesar.

3.2 Practical problems

Several practical problems had to be solved before starting the experiments.

Building the sub-system. In order to verify a property on a part of the system, we have to be able to identify the nodes involved in the property and then to build the main node corresponding to the sub-system. The set of nodes which are necessary to build this main node are manually identified using the variables involved in the property. The main node is then automatically generated in Lustre from the list of selected nodes (connections are made between inputs of a node and outputs of another node if they have the same name).

Symbols. The specification includes several symbols. Symbols are numerical functions that were implemented directly in C for efficiency or simplicity reasons. Some of these functions have now been written in SCADE, but there are still a few symbols that are only implemented in C.

The solution is to define assertions characterising the behaviour of these symbols and to add them to the specification.

Reals. The SCADE specification of the case study contains a lot of real variables. But Lucifer does not allow reasoning on reals, it can only handle booleans and integers. We thus had to translate reals to integers. As we wish to be as automatic as possible, we have adopted the following method: we first find the precision needed, then we multiply every real to translate it into an integer. In the case study, we multiplied reals by 100. But of course, this translation makes value intervals notably bigger and this may cause problems for the verification.

Lesar accepts real values, it is able to handle numerical properties if the computations are linear.

3.3 Current results

We present here the current status of the ongoing experiments.

Property P1. Let us recall the informal expression of property P1: *if we suppose that the two sidesticks cannot be faulty at the same time, there is always an active stick.*

Ten nodes were selected concerning this property, nodes that express the relationship between the inputs (pilot and copilot order, faulty sticks) and the outputs (transmitted order, priorities). These nodes are essentially composed of nodes for the management of priorities, nodes for monitoring and nodes for the functional part. Then a main node was automatically built from these selected nodes.

Symbols used in this subsystem are written in SCADE so we had no problem for that part. Reals were translated by hand using the technique described previously for Lucifer. No translation was needed for Lesar.

We first made several attempts with our first formal expression of the property using the transmitted order:

$$\begin{aligned}&P1 : (PILOT\_ORDER \neq 0) \vee (COPILOT\_ORDER \neq 0)\\&\quad \rightarrow (TRANSMITTED\_ORDER \neq 0)\end{aligned}$$

It leads us to several refinements of the property to finally get a more detailed expression that was found valid. This was a first interesting result, but

– Several assertions were necessary on inputs of the subsystem for the property to be valid. These assertions have to be assessed by designers or proved on another subsystem.
– We had to enter in the details of *how* the subsystem was implemented. This is in contradiction with the methodology defined at the beginning.

Consequently, we decided to make other experiments using the second formal expression of the property:

$$\begin{aligned} P1 : \neg((&PILOT\_PRIORITY \wedge PILOT\_STICK\_FAULT)\vee \\ &(COPILOT\_PRIORITY \wedge COPILOT\_STICK\_FAULT)) \end{aligned}$$

This property was proved valid with no assertion except the initial one stating that the two sticks were not faulty at the same time. This assertion was very easy to express because there was a boolean in the specification corresponding exactly to the fact that both sticks were faulty: TWO_STICKS_FAULTY. So the assertion simply was:

$$assert(\neg TWO\_STICKS\_FAULTY)$$

Both properties were proved by Lucifer and Lesar. Verification was efficient in both cases (a few seconds).

Property P2. A difficulty which often arises when we want to express properties on a system like SC1 is to select the relevant variables among hundreds of inputs and outputs that the system holds. That was the case for P2 and the first step was to select the right input and output variables to get the formulation shown in §2.2.

Then, we had to select a sub-part of the SC1 specification. Our process was twofold :

– in a first attempt, we selected the nodes whose topic is "inner aileron" (the topic is given at the head of the SCADE sheets) plus the nodes used by these sheets;
– then, we selected the nodes using the variables involved in the property, as suggested in §3.2.

Fortunately, these two ways gave the same result, and led to the selection of thirteen nodes.

The selected sub-part of the specification uses some symbols. Most of them are quite simple and were already described by Lustre specifications. We could use these specifications for our purpose, except for one symbol whose specification was rewritten. So it was not necessary to describe the behaviour of symbols by assertions.

The rewritten symbol was a little less simple than the other symbols. It mainly consists in interpolation calculus between real values. In order to use Lucifer and NP-TOOLS, we had to transform it into a function on integers.

This was done using the conversion presented in §3.2 (multiplication of reals by their precision to get integer values).

The conversion from reals to integers mainly consists in changing declarations of real variables into declarations of integers and to adapt some calculus involving multiplication or division operations in order to respect the conversion. This happened only three times in the considered sub-system. We made these changes manually, but they can easily be automated and integrated in the Lucifer translator.

The selected sub-system uses temporal operators which induce an initial phase where the system behaviour is not stabilised and where some properties which are normally valid in the current phases may not be satisfied. This is the case for property P2. We have defined the following *begin_ok* operator, which masks the value of a boolean expression `E` during the n first instants, in order to skip the initial phase and to focus on the current behaviour of the system.

```
node begin_ok (E:bool; n:int)  returns(S:bool);
var  ct : int;
let
  ct = 0 -> (pre (ct) + 1) ;
  S = (ct < n) or E ;
tel;
```

Property P2b has been modified, as shown below, and was checked and proved by Lucifer.

$P2b:\ ((G1 = 1) \wedge (G2 = 1)) \rightarrow$
$((\neg PRIMARY\_COMPUTER1\_OK) \wedge (\neg PRIMARY\_COMPUTER2\_OK))$

$P2b':\ begin\_ok(P2a, 5)$

4 Conclusion and future work

In this paper, we have presented ongoing experiments on the use of formal verification for critical systems in an industrial context. We only have qualitative results for the moment and more experiments are needed to get quantitative results as well.

However, current results are encouraging. This first use of such techniques in an industrial environment shows it is a valid way to replace some simulation activities. Once the property is formalised, it is straightforward to verify it with the tool. We sometimes had problems understanding counter-examples, but the reason was we lacked knowledge about the application. Designers would not have such difficulties. This technique thus allows an exhaustive check of the system with respect to a given property.

The next essential step is the integration of these techniques in the current verification and validation process. Three major issues need to be addressed in order to complete this integration:

- automation of the different steps of the proof process described above (construction of the main node, translation of real variables);
- integration in the SCADE environment with a suitable user friendly interface in order for it to be directly usable by the system designers;
- definition of a methodology to help designers express expected behaviors as properties instead of describing them as lists of scenarios.

As an attempt to address these issues, a prototype called L4 is under development at Prover Technology. L4 is an extension module that will add formal verification capabilities to SCADE. It will provide the same functionalities as Lucifer but with a user-friendly interface and a full integration with SCADE. The first version of this tool will be available in october 2000.

Finally, it is necessary to study more deeply how verification techniques complement the test and simulation phases. We have to be able to find a way to determine which parts of test and simulation can be eliminated and to measure the benefit in practice. The correspondance between properties and scenarios could help in understanding the complementarity.

References

1. P. Caspi, N. Halbwachs, P. Pilaud, and P. Raymond. The synchronous dataflow programming language Lustre. *Proceedings of IEEE, Another Look at Real-time programming*, 79(9):1305–1319, September 1991.
2. N. Halbwachs, Lagnier F., and Raymond P. Synchronous observer and the verification of reactive systems. In *Third International Conference on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 93.
3. N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real time systems by means of the synchronous data-flow language Lustre. *IEEE Transactions on Software Engineering, special issue on the specification and analysis of real time systems*, september 1992.
4. Magnus Ljung. Formal modelling and automatic verification of Lustre programs using np-tools. Master's thesis, Royal Institute of Technology, Department of Teleinformatics, 1999.
5. Prover Technology AB. *NP-Tools 2.4 Reference manual*, 1999.
6. C. Ratel. *Définition et réalisation d'un outil de vérification formelle de programmes Lustre : le système LESAR*. PhD thesis, Institut National Polytechnique de Grenoble, Juillet 1992.
7. Mary Sheeran and Gunnar Stalmarck. A tutorial on Stalmarck's proof procedure for propositional logic. In *Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD)*, LNCS. Springer Verlag, 1998.
8. Verilog. *SCADE Language - Reference Manual 2.1*.
9. Verimag. Lustre/lesar home page. http://www-verimag.imag.fr/SYNCHRONE/lustre-english.html.