# Model Checking Flight Control Systems: the Airbus Experience

Thomas Bochot[1,3], Pierre Virelizier[1], Hélène Waeselynck[2] and Virginie Wiels[3]
firstname.lastname@airbus.com, @laas.fr, @onera.fr
[1]AIRBUS France, 316 route de Bayonne 31060 Toulouse Cedex 03, France
[2]LAAS-CNRS ; Université de Toulouse ; 7 avenue du Colonel Roche, F-31077 Toulouse, France
[3]ONERA/DTIM, 2 avenue Edouard Belin, 31055 Toulouse, France

## Abstract

*This paper presents experiments realized by Airbus on model checking a safety critical system, lessons learnt and ways forward to extend the industrial use of formal verification at design level.*

## 1. Introduction

The Flight Control System (FCS) is one the most critical systems inside an aircraft and, like many other embedded systems, its size and complexity has grown in recent years. Verification and validation (V&V) of such a system is thus essential but not easy. Classical V&V methods consist in simulation and tests but formal verification is now a credible method to complement the classical ones. This paper presents experiments realized by Airbus on model checking the FCS, lessons learnt and ways forward.

Section 2 briefly describes the current Airbus development process; Section 3 presents the SCADE environment, language and the notion of observer that is used for formal verification; Section 4 summaries the results of three past R&D studies; Section 5 presents a successful experiment we have realized on the ground spoiler function (part of FCS); Section 6 discusses insights gained from the study; Section 7 gives conclusions and proposes avenues for future work.

## 2. Airbus software development process

Figure 1 summaries the development cycle at Airbus. Three levels are distinguished: aircraft, system and equipment. The most critical systems, like flight control systems, are designed using the formal language SCADE [1]. A qualified code generator automatically generates most of the embedded code. Some verification is raised from the code to the design level. Main V&V activities at Airbus are the following:

- Model tests: the considered system is validated in a simulated environment. This is performed on desktop simulators providing a panel of commands representing possible pilot actions. Testers define test scenarios with respect to the detailed requirements. The scenarios are executed on the simulator; the testers then decide whether or not the test is successful.
- Aircraft level simulation: several systems are validated in a simulated environment.
- Formal verification at code level: Airbus uses abstract-interpretation based tools to verify non functional properties of programs (such as absence of run-time errors) and uses proof based tools to verify functional properties on the parts of the programs that are manually coded. These uses of formal methods have been very successful [2] and are being extended. They are not addressed in this paper; we consider the system level and a different technique (model checking).

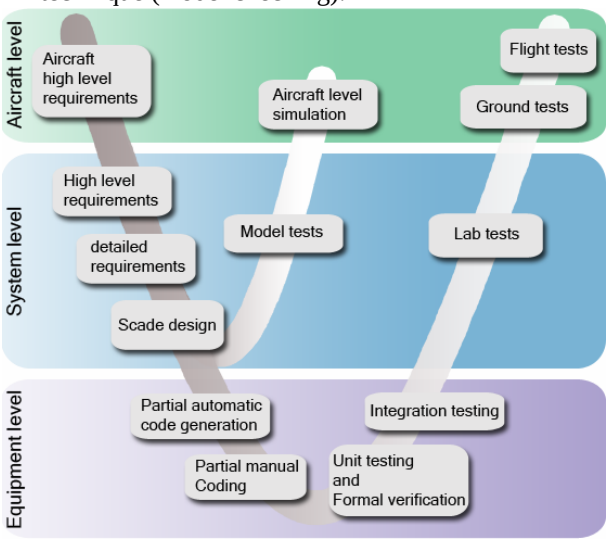



**Figure 1: Airbus development process**

- Software integration tests.
- Lab tests: first tests with real equipments, on a single system or on several systems.
- Ground tests and Flight tests.

This paper focuses on the verification activity at system design level, which currently involves model tests. As the design is expressed in the SCADE formal language, the possibility of using formal analysis for part of the verification at this level has been studied. It must be noted that the SCADE model is a detailed design from which the embedded code is generated; it is thus more complex than models that are specifically defined for formal verification purposes like in [3].

## 3. SCADE

Before reporting on the Airbus experience, it is worth introducing the SCADE environment. The SCADE environment [1] ("Safety Critical Application Development Environment" commercialized by Esterel Technologies) was defined to assist the development of critical embedded systems. This environment is composed of several tools including:

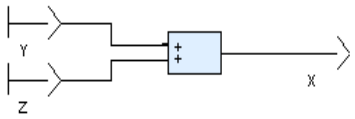
- a graphical editor,
- a simulator,
- a code generator that automatically translates graphical specifications into C code,
- a model checker.

We focus here on the SCADE language and the model-checking facility offered by the SCADE environment.

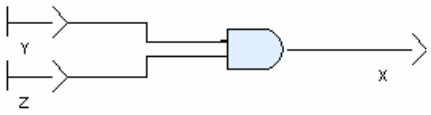
### 3.1 SCADE language

The SCADE language is a formal graphical notation based on Lustre [11]. It is thus (like Lustre) a synchronous data-flow language. Synchronous languages rely on the synchronous hypothesis. In practice, this hypothesis stipulates that the system is able to react to an external event, before any further event occurs.

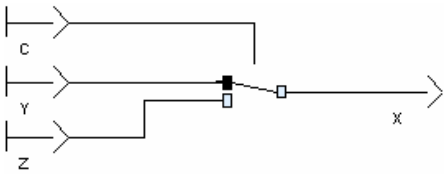
In the SCADE language, any expression X represents a flow. A flow is an infinite sequence ($x_0$, $x_1$, …, $x_n$, …) of values. A system is intended to have a cyclic behavior, and $x_n$ is the value of x at the $n^{th}$ cycle of the execution. A system is described using SCADE sheets with inputs, outputs and local variables. A sheet defines a set of equations to compute the output flows from the input ones. An equation can be seen as a temporal invariant, e.g. x = y+z defines the flow ($x_0$, $x_1$, …) = ($y_0+z_0$, $y_1+z_1$, …). Figure 2.a shows how the previous equation is graphically represented in the SCADE language.



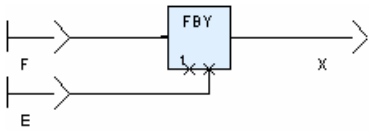

a. the *add* operator




b. the logical *AND*




c. the *if-then-else* operator




d. the *Followed-by* operator

**Figure 2: Some SCADE operators**

Equations are made of variable identifiers, constants, usual arithmetic, Boolean and conditional operators. For example, Figure 2.b and 2.c respectively give the representation of the *AND* operator and the *if-then-else* operator. A specific temporal operator denoted *followed-by* (FBY), allows to consider time in design.

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 X = FBY(E, 1, F) is an expression denoting the sequence ($e_0$, $f_0$, $f_1$, ...). The graphical notation for this operator is given in Figure 2.d.

### 3.2 Flight control system design

Flight control systems are formally described through a set of SCADE sheets.

A library of operators dedicated to the design of flight control systems was developed by Airbus. These domain-specific operators are called *symbols*. As symbols are commonly used in flight control functions, they are directly described in a low-level language, such as C code or Assembly, in order to optimize execution time. The code of the symbols is validated according to level A of RTCA DO-178B.

### 3.3. Synchronous observer

Before applying model-checking techniques on SCADE design, we need to introduce the notion of synchronous observer [4]. Informally speaking, an observer monitors the behavior of the system and decides whether it is correct or not. Technically, an observer is a synchronous program that is connected to the design sheets under verification, without feedback. Figure 3 describes a typical verification architecture. It includes two categories of observers: one for the property to be verified, and one for some hypotheses about the system environment. The observers, and their connection to the target design, are specified in a SCADE sheet.

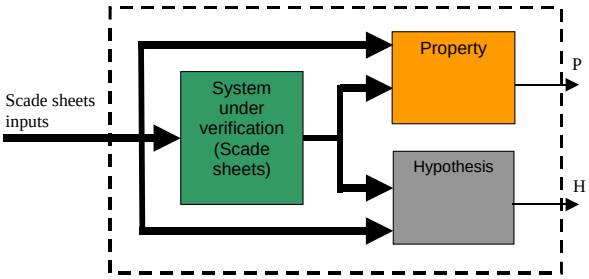



**Figure 3: Verification using synchronous observers**

The SCADE model-checker allows us to verify that the design satisfies the property under the hypotheses.

The property is a safety property expressing that something will never happen. Hypotheses are used to constrain inputs behavior throughout the analysis.

The property observer computes a unique Boolean flow P that is true as long as the design satisfies the property.

The other observer computes a Boolean H (not necessarily unique) and the model-checker is told to keep H true throughout the analysis of P.

If the model-checker concludes that P always holds, this guarantees the absence of error for all the executions of the design under the stated hypotheses. However, if (at least) one error exists, a counterexample is provided by the model checker. The counterexample is a sequence of inputs that, when applied to the design, allows us to reach a state violating the property. The SCADE simulation facilities may then be used to replay the counterexample and try to understand the reason for the violation.

### 3.4 Example

To illustrate the previous notions, we present a toy example: an air brake control function. This textbook case is given for clarification purposes, and is clearly not representative of a real avionics function.

The design of the function uses one of the symbols mentioned in section 3.2, namely a reset-biased *latch*. This operator has two inputs *Set* and *Reset* and one output *Output*. At initial step, *Output* is false. It becomes true each time *Set* is true and *Reset* is false, and remains true until *Reset* is true. The SCADE implementation of the *latch* operator is shown in Figure 4.

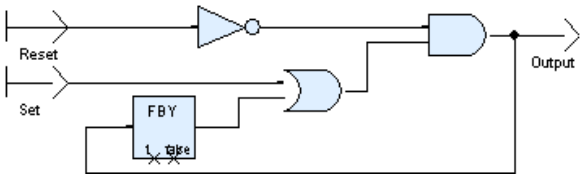



**Figure 4: The reset-biased *latch* operator**

Figure 5 shows the description of the function using the *latch* operator. Sending an order *airBrakeExtension* to an actuator activates aircraft air brake. It is disactivated by sending *airBrakeRetraction*. Completion of the execution of an order may be checked by sensing the physical position of the controlled surface. In this basic example, the function uses two sensors *airBrakeDown* and *airBrakeUp* that respectively indicate retracted or extended position. The other inputs *airBrakeLeverOn* and *airBrakeLeverOff* come from two sensors located on the air brake lever. Input *airBrakeLeverOn* is true when the pilot wants to extend air brakes and *airBrakeLeverOff* is true when the pilot wants to retract air brakes.

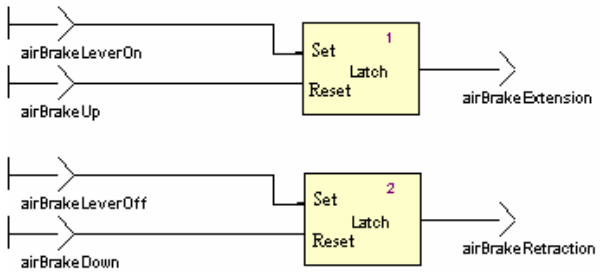



**Figure 5: Air brake control example**

On that design we want to verify that there exists no state where orders airBrakeRetraction and

airBrakeExtension are true simultaneously. We suggest the following safety property P:

**Not (airBrakeExtension and airBrakeRetraction)**

From a physical point of view, *airBrakeLeverOn* and *airBrakeLeverOff* cannot be true at the same time as the lever cannot be set *on* and *off* at the same time. We introduce the following hypothesis H to account for this physical impossibility:

**Not (airBrakeLeverOn and airBrakeLeverOff)**

We can now define the observers of Figure 6 in which P is connected to outputs *airBrakeRetraction* and *airBrakeExtension* of the *airBrakeControl* function. The textual notation above the H observer (v H) is an assertion indicating that H is assumed to hold throughout formal analysis.

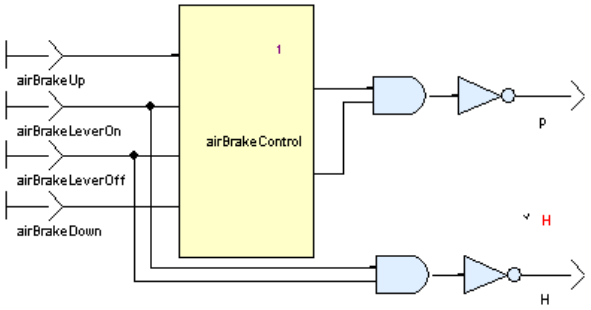



**Figure 6: Observers of the example**

As the *airBrakeControl* function does not satisfy the property, a counterexample is provided allowing us to identify the error. At a first step, an airbrake extension order is requested by the pilot (*airBrakeLeverOn* is true), setting *airBrakeExtension* to true. At a second step, the previous airbrake extension order is canceled (*airBrakeLeverOff* becomes true and *airBrakeLeverOn* becomes false) setting *airBrakeRetraction* to true. As *airBrakeUp* never appeared between those two events, *airBrakeExtension* remains true conducting the system to a state violating the property P.

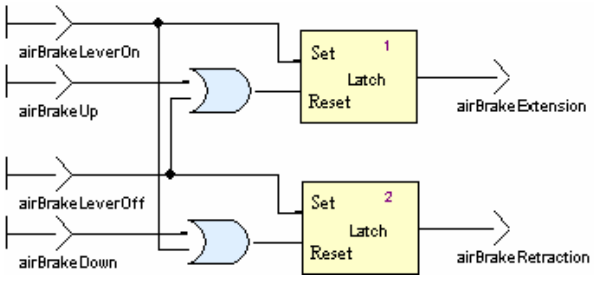



**Figure 7: Air brake control example corrected**

This counterexample leads us to fix the design of the system as shown in Figure 7.

Using the same observers as previously, we can now demonstrate that the corrected system design satisfies the property P.

## 4. Previous R&D studies

Airbus has conducted experimental work on formal verification of flight control systems for several years. Three studies are reported here, the last two being unpublished so far. All studies intended to verify design models against their requirements, using the synchronous observer method described in the previous section.

### 4.1. First study

The first study [5], done in collaboration with Onera, aimed at assessing the feasibility of a formal approach on the A340 500-600 Flight Control Secondary Computer design model. The study dealt with high-level system requirements. The experimented tools were NP-TOOLS [6] and Lesar [7]:

- NP-TOOLS, from Prover technology AB, is a general-purpose verification toolbox for combinatorial circuits based on Stålmarck's proof procedure [8].
- Lesar is a BDD-based model checker for Lustre. The algorithm enumerates the reachable states.

The tools obtained similar results. Generally speaking, experimental results highlighted the technical limitations encountered by model checkers to handle numerical calculations (on integers only). However, results on the Boolean parts of the design were encouraging.

A problem that was identified at the time was that some symbols were only implemented in C code. In order to be able to use formal verification on systems including these symbols, a SCADE version of the symbols had to be defined (or an assertion abstracting the symbol behavior). This has now been done for all symbols.

At the end of the study, Esterel Technologies and Prover Technology decided to add a model checking functionality (SCADE Design Verifier) to the SCADE environment.

### 4.2. Second study

Following the encouraging results of the previous study, the second study intended to test the capability of SCADE Design Verifier to verify different types of properties existing in FCS. To this end, twelve experiments have been performed on the A340 500-600 Flight Control Secondary Computer design model.

Table 1 summarizes the experiments. They are representative of different facets of FCS:

1. Multiprocessor. Airbus FCS runs on a fault-tolerant asynchronous multiprocessor architecture [9].
2. Several computation periods. On a given computer, different pieces of code are cyclically executed at different periodicity.
3. Numerical calculation. Part of the models may include complex, either floating-point or integer, computations (e.g. second order filters).

**Table 1. Results of the second study**

| # | On multi processor system | Several computation periods | Includes numerical calculation | Ability to perform analysis |
|---|---|---|---|---|
| 1 | ✓ | | | Yes |
| 2 | ✓ | ✓ | | Yes |
| 3 | | | | Yes |
| 4 | | | ✓ | Failure |
| 5 | | ✓ | | Yes |
| 6 | ✓ | ✓ | ✓ | Failure |
| 7 | ✓ | | | Yes |
| 8 | | | | Yes |
| 9 | | | | Yes |
| 10 | ✓ | ✓ | | Failure |
| 11 | | | ✓ | Yes |
| 12 | | | ✓ | Failure |

"Yes" in the rightmost column means that the model checker is able to perform the analysis (whether the property is valid or not). When none of the three middle columns is ticked, the experiment deals with Boolean reasoning in a single processor, single clock framework. Such is the case for experiments 3, 8, 9, which demonstrates the ability to solve such problems.

Experiments with multiprocessor consideration raised modeling issues. Two phenomena had to be modeled: communication delays and clocks asynchrony, which increased the complexity of the models. Experiments 1, 2, 7 were successful because they used a simplified model of the phenomena whereas experiment 10 failed. There must be a tradeoff between model accuracy and state number explosion. An important remark is that the SCADE language is not dedicated to model asynchronous behavior.

Accounting for the different execution periods of SCADE sheets also required some manual modeling. This was cumbersome, and a dedicated tool has now been implemented. The verification part was not a problem. For example, experiment 5 succeeded in analyzing a pretty large model consisting of 100 SCADE sheets and involving four different computation periods.

Concerning numerical problems, experiment 11 succeeded in performing the analysis of a relatively simple floating-point problem. This is an improvement compared to the first study (Section 4.1) that reported very poor performance as soon as numerical calculation was involved. The better result is due to an evolution of the tool between the two studies. But analysis of non-linear calculation remains intractable, yielding experiments 4, 6, and 12 to fail.

The conclusion of the study is that model checking can be used for many Airbus functions provided that they involve simple numerical calculation and a single computer. The characteristics of the design as identified in Table 1 are thus relevant to decide whether the verification can be done by model checking.

### 4.3. Third study

A flurry of experiments aimed at comparing the "return on investment" between a model test approach and a formal analysis approach to verify detailed requirements.

A total of 135 requirements on three functions of the A400M Primary Flight Control System were selected for experimentation.

The study concludes that error detection, comparing to the current model test approach, has not been improved while taking two to three times longer. The poor detection power might be due, at least in part, to the fact that the study dealt with detailed requirements. Most of the time, this makes properties close and somewhat similar to the design model. Error detection power seems to decrease as property syntax comes closer to the model syntax. Thus, the proposed approach is not suitable for detailed requirements. The detail level of the requirement to be verified constitutes then a criterion to decide whether the verification is worth being done by model checking.

## 5. Model checking the Ground Spoiler function

We now report on a recent study addressing the A380 Ground Spoiler function. This section gives a high-level view of the corresponding experiments. Unfortunately, a more detailed account is not allowed for non-disclosure reasons, but the presented material should be quite sufficient to understand the main findings of the study (to be subsequently discuss in Section 6).

The Ground Spoiler function aims at keeping the aircraft on the ground at landing or in case of rejected take-off. The spoilers and ailerons (see Figure 8) are used to decrease lift as much as possible in order to maximize wheel braking. An airbrake effect is also produced. The ground spoiler function is designed in such a way that an undue extension is "extremely improbable" when the aircraft is flying.

The case study was chosen as an example where formal analysis could usefully complement the current SCADE design validation. The property to be checked is the high-level safety requirement stipulating that the ground spoilers must not extend while the aircraft is flying. We have two versions of the function design: a flawed version and a corrected one. The first one exemplifies a design flaw that was not revealed during model tests. It was revealed during lab tests. Our aim was to investigate whether the flaw could have been found earlier by model checking, which would have been less costly. We also intended to demonstrate that the corrected version fulfills the safety requirement.

The Ground Spoiler function was expected to fall well into the applicability domain of model checking. It mainly involves Boolean calculation. There are few numerical variables and their processing is basic (e.g., comparison with thresholds, integer addition). However, the presence of temporal counters may cause problems, due to the temporal windows to be considered.

Building on previous studies, the formal verification scope is deliberately kept limited. We focus on few SCADE sheets extracted from the complete design. We exclude multiprocessor aspects from consideration. Also, the focus is on nominal behavior: no computer reset, no sensor fault.

Analysis of the flawed version started by considering a single SCADE sheet, the one computing the ground spoiler command sent to the surfaces. From this starting point, analysis required several verification steps:

1. A number of initial steps were performed to obtain observers tailored to the verification problem. We explored different formulations of the safety property and the hypotheses restricting the scope of verification.
2. Once formalization was established, we obtained a counterexample that was not easy to interpret. Violation of the property was caused by a latch component not being properly reset: its reset condition (*RCond*) was stuck to false. In order to determine whether or not this behavior could happen, analysis had to include the SCADE sheet computing *RCond*. We thus modified the observers to accommodate the enlarged model, now consisting of two sheets.
3. Formal analysis of the enlarged model confirmed the stuck-at-false counterexample.
4. We decided to continue the search for violation patterns. As we did not want to modify the target SCADE sheets, we had to modify the formulation of the property, in such a way that it explicitly excludes the previous counterexample. Formal analysis then returned a new counterexample with *RCond* stuck at true.

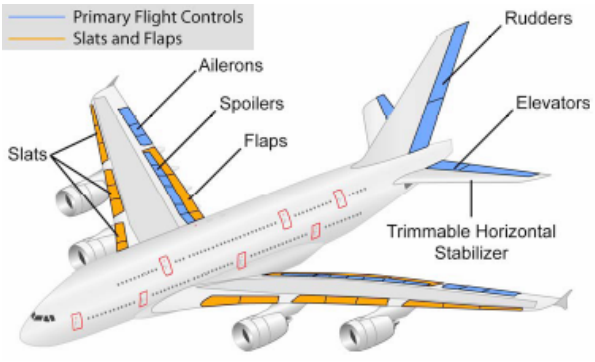



**Figure 8: A380 control surfaces**

The stuck-at-false and stuck-at-true counterexamples correspond to different aircraft scenarios. In each case, the latch reset problem results from a specific combination of pilot actions and some (unusual) aircraft trajectory. The second scenario is the one found by testing. The other one exemplifies an alternative manifestation of the same flaw.

The corrected version of the Ground Spoiler function involves a different internal logic (no longer using *RCond*). To verify the logic, we used essentially the same observers as the ones obtained at the end of Step 1, with slight adaptation to account for changes in the function interface. First attempts to check the property were unsuccessful (analysis did not terminate). We had to introduce assertions about

intermediate variables, and were then successful. Dedicated model checking experiments allowed us to establish the validity of the assertions, hence completing the formal analysis.

## 6. Insights Gained from the Study

After this factual presentation of the study, we comment on its main outcomes.

### 6.1. Effectiveness of formal verification

Although the Ground Spoiler function involves little numerical computation, it is sufficiently complex to challenge the verification tool because of the presence of temporal counters. It took about 48 hours to exhaustively analyze the correct version, analysis being run on a 1.7-GHz Pentium 4 processor with 256 MB of RAM. As regards the incorrect version, the production of counterexamples lasted from minutes to hours, depending on the length of the counterexample and the chosen exploration strategy (SCADE offers two strategies). The returned counter-examples were between 50 and 160 cycles length.

Formal verification proved quite effective. Let us recall that the model tests did not manage to trigger property violation. This is due to the fact that highly dynamic aspects are not the focus of these early tests: unusual aircraft trajectories, such as the one associated with the violation scenarios, are not considered at this stage for validating the logic. The flight dynamics is more deeply addressed by lab tests (using a full flight simulator) and of course by flight tests. However, the earlier the flaws are detected, the less costly it is. Our study shows that model checking may be a practical means to achieve detection at design level. The study also provided rich feedback on the flawed behavior, by allowing us to identify two violation scenarios (corresponding to aircraft trajectories that are quite different in each case).

### 6.2. Expression of properties

Bridging the gap between informal requirements and formal properties (to be used for automated verification) is known to be a difficult problem. In our case, the problem is exacerbated by the fact that the models to be verified are at a detailed unit level, while the informal safety requirements are given at an abstract aircraft-wide level. As a result, it is not obvious how properties can be formulated in terms of the variables of the target SCADE sheets.

The problem was already mentioned in the first study discussed in Section 4.1. At that time, an example of difficulty was the formal expression of a property involving the notion of “active” sidestick. Informally, the pilot or copilot sidestick is active if it can have an effect on the aircraft control surfaces. Capturing this notion proved a tricky issue and a number of alternative formulations were investigated. In the Ground Spoiler study, we faced the issue of how to formalize the notion of “flying” aircraft, which once again required some elaboration.

Care must be taken to avoid paraphrasing the SCADE model (see also the conclusions of the third study, Section 4.3). For example, the Ground Spoiler function computes an intermediate variable to determine whether or not the aircraft is seen on ground. If the same computation is used in the P observer, then the model trivially satisfies the property. We thus had to investigate an independent expression of the property.

It is worth noting that this pitfall is less likely to occur when checking whether test results conform to requirements. For model tests as well as for lab tests, the simulation environments provide an external reference to determine the “real” aircraft configuration (as opposed to the configuration *perceived* by the software functions). In the Ground Spoiler example, determining whether the aircraft is flying or on ground is no longer a problem if the flight simulation data can be accessed.

### 6.3. Determination of hypotheses

Determining a set of reasonable hypotheses, that narrows the scope of verification, is a recurring issue whatever the chosen verification method.

Our study involved two categories of hypotheses:

- Hypotheses to exclude unrealistic behavior,
- Hypotheses to simplify the verification problem and make it amenable to automated analysis.

As regards the first category, it is not practically possible to finely discriminate between realistic and unrealistic behavior. There is an inherent difficulty in accounting for the flight dynamics. While complex dynamic models are routinely integrated into test environments, their consideration for formal verification is not conceivable: automated analysis would be intractable. Also, scalability issues force us to focus on a (small) subset of SCADE sheets, meaning that the inputs for the sheets are not necessarily system-level inputs. As a result, there is no pretension from us to characterize the valid input

domain of the target function. Hypotheses are merely intended to eliminate some grossly unrealistic input patterns (e.g., we introduce ranges for the numerical values, and account for obvious correlations between inputs).

In contrast to the previous category of hypotheses, simplification hypotheses yield us to deliberately exclude some realistic behavior from consideration. For example, we mentioned that the analysis is focused on a single processing unit under nominal conditions: we do not consider sensor faults or resets. Another example concerns the landing gear wheels. We assume that the wheels are always all stationary or all rotating at the same speed. While considering an asymmetric behavior of wheels is perfectly legitimate (and in fact more realistic than strictly identical behavior), it is deemed to unnecessarily compound the analysis of the Ground Spoiler logic.

To sum up, the verification exercise is performed under hypotheses that concern only a subset of operational behaviors, while failing to exclude all unrealistic behaviors. A given set of hypotheses may always be questioned. A benefit from formalization is to make the scope of verification explicit. As a general rule, we avoided to put hypotheses on pilot actions: the satisfaction of the studied requirement does thus not depend on crew operating procedures.

### 6.4. Interpretation of counterexamples

If formal verification successfully terminates, then the target function is proved to fulfill the property under the stated hypotheses. But if a counterexample is returned, further analysis is required to conclude on property violation. Let us recall that verification is performed at a unit level: counterexample input configurations need to be interpreted in terms of system-level input configurations. Also, the verification hypotheses are not sufficient to exclude unrealistic behaviors. It has then to be determined whether the counterexample may correspond to a real aircraft scenario, which requires some effort.

The situation is quite different for current tests. All categories of tests, including model tests, involve system-level scenarios that have a clear physical meaning (e.g., a landing scenario). This makes it easier to analyze test results. For example, to analyze the result of model tests, testers typically use time diagrams to visualize test traces. The interpretation of the observed evolution of variables is guided by the understanding of the applied scenario.

Time diagrams are not so helpful for analyzing counterexamples. The evolution of variables appears to be meaningless from a physical viewpoint (e.g. variables that do not directly contribute to the violation are assigned arbitrary values by the tool). Still, despite some unrealistic details, the counterexample may indicate a real problem in the target logic.

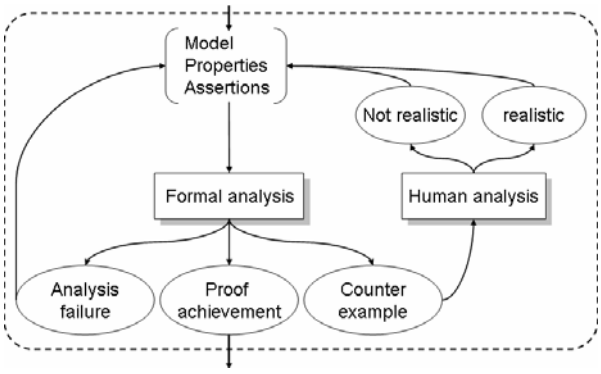



**Figure 9: Methodology for analysis**

As time diagrams were poorly informative, another analysis approach had to be adopted. It consists of:

1. Identifying the structural parts of the model that are activated by the counterexample over time;
2. Trying to exhibit a meaningful system scenario that reproduces the observed activation scheme.

The second step is the most difficult and requires expertise in the avionics domain. The first step could be automated, provided that the notion of *activation scheme* is precisely defined.

### 6.5. Iterative approach to verification

Due to the above-mentioned problems (expression of properties and hypotheses, interpretation of counterexamples), conducting formal verification is far from a single push-button experiment. It is more like an iterative "what-if" approach, as shown in Figure 9.

Iterations are required to explore alternative formulations of the verification problem (including both the property and the hypotheses). The results of initial verification steps provide a useful guide to revise formulation. Analysis failure indicates the need for additional hypotheses. Irrelevant counterexamples question the adequacy of either the hypotheses or the property. Figure 9 emphasizes the importance of human analysis when a counterexample is found. Note that in some cases, attempts to find a system-level interpretation may yield questions on other parts of the SCADE model (for example, the part corresponding to the computation of Boolean *RCond* used by the Ground Spoiler logic). Those new parts come with new verification artifacts to be formulated.

Given a verification problem, the first

counterexample returned by the tool is not necessarily a meaningful one. It may represent some unrealistic behavior that could not easily be excluded by means of input hypotheses. Analysis must continue to determine whether or not other counterexamples can be found, that may reveal a real violation problem. Or in case a real problem is revealed, it may be useful to gain deeper insights into the various violation scenarios it induces. During the study, we encountered those various cases justifying the search for additional counterexamples. Unfortunately, SCADE Design Verifier (like most model-checkers) does not allow for several counterexamples. Repeating verification can only yield the same counterexample to be produced again and again. To overcome this problem, we had to use a trick. We manually modified the P observer so as to exclude the counterexample previously found. This way, we managed to explore alternative paths toward property violation. We eventually retrieved the flaw found by testing, and demonstrated that it could yield at least two different violation scenarios.

## 7. Conclusion

The experiments reported in this paper were targeting the Airbus development process for Flight Control Systems. The Airbus context particularizes the application of the model checking technique as follows:

- In the target process, a SCADE formalization occurs at system design. The analyzed models are thus at a concrete level of details.
- The FCS functions run on a multi-processor and multi-clock environment (multi-clock aspects also arise inside a single processor).
- FCS functions range from simple Boolean functions to functions involving complex numerical calculation. Determining their valid input domain is usually non-trivial, because of the need to consider the flight dynamics.
- The target properties may be, or not, at the same level of abstraction as the models. It depends on whether they come from the detailed or high-level requirements.

Some of the above characteristics are likely to be shared by other industrial companies also developing control systems. For example, the automated generation of code from design models is not uncommon: whether such design models are suitable to formal analysis is a question that may be relevant to different safety-critical domains.

The study of the Ground Spoiler function came after a series of experiments that investigated the applicability of model checking in the Airbus context. The aim of this additional study was not only to confirm the previous findings (i.e., confirm the expected applicability), but also to tackle a real problem that was revealed during lab testing. The results show that model checking can be quite effective in revealing subtle flaws that are otherwise found late. We thus believe that an industrial use of model checking is possible and worthwhile at system design level. At the moment, it seems that the most efficient use is to consider lightweight approaches [10], where formal analysis is applied to few critical functions and focused on a subset of their possible behaviors. Model checking is not yet ready for a more extended use (such as the one already done at code level [2]).

In addition to providing feedback on the applicability and effectiveness of model-checking in the Airbus context, the reported material allows us to identify some industrial needs for tool support.

Even in the case of lightweight usage, performance may be a problem. For the Ground Spoiler function, first attempts to formally verify the correct version were unsuccessful, and the final analysis took about 48 hours. It would be quite useful if the verification tool could provide some metrics to indicate the difficulty of the analysis, for a target model and a target property. Such metrics would be a useful guide for deciding whether or not to attempt formal verification, and for allocating appropriate computing resources in case the decision is positive.

Another important conclusion of the Ground Spoiler study is that model checking is far from a single push-button experiment. It is rather an iterative process as described in Section 6.5. Moreover, this process will be iterated several times: even if the proof is successful, the user may want to go further in the analysis by suppressing an hypothesis or modifying the property or enlarging the model that is taken into account.

Verification tools insufficiently support this iterative process. There is clearly a need for better support to manage the proof configuration versions, e.g. to make archives from the successive experiments, to extract a history from the archives, and to facilitate comparison of the used configurations. Another shortcoming is the inability to supply several counterexamples. Manual modification of the observers to avoid the previous counterexamples is not satisfactory. Our future work will investigate an automated approach. An interesting issue is how to force the tool to produce counterexamples that illustrate *different* violation patterns. When interpreting counterexamples, a first

step was to identify the structural parts of the model that were activated over time. We will study whether structural coverage analysis can be useful not only to facilitate the understanding of counterexamples, but also to guide the model-checker toward the exploration of different violation patterns.

## 8. References


[1] www.esterel-technologies.com/products/scade-suite

[2] S. Duprat, J. Souyris, D. Favre-Félix, "Formal verification workbench for Airbus avionics software," in *Embedded Real-Time Software (ERTS), 2006*.

[3] S. Miller, A. Tribble, M. Whalen, M. Heimdahl, "Proving the shalls – early validation of requirements through formal methods," *Int. Journal of Software Tools for Technology Transfer*, 8(4), pp. 303-319, August 2006.

[4] H. Halbwachs, F. Lagnier, and P. Raymond. "Synchronous observers and the verification of reactive systems," in *Third Int. Conf. on Algebraic Methodology and software Technology (AMAST'93)*, Workshops in computing, Springer Verlag. Twente, June 1993.

[5] O. Laurent, P. Michel and V. Wiels, "Using formal verification techniques to reduce simulation and test effort," In *Formal Methods for increasing software productivity*, FME2001. Springer, 2001.

[6] M. Ljung, "Formal modelling and automatic verification of lustre programs using NP-Tools," Master thesis, Prover Technology AB and dept. of Teleinformatics, KTH, Stockholm, 1999.

[7] C. Ratel, « Définition et réalisation d'un outil de vérification formelle de programmes Lustre : le système LESAR », Ph.D dissertation, INPG, July 1992.

[8] M. Sheeran, G. Stalmarck, "A tutorial on Stalmarck's proof procedure for propositional logic," in *Formal Methods in Computer-Aided Design*, LNCS 1522, pp. 82-100. Springer Verlag, 1998.

[9] P. Traverse, I. Lacaze, J. Souyris, "AIRBUS Fly-By-Wire: A Process Toward Total Dependability," in 25th *Int. Congress of the Aeronautical Sciences (*ICAS), 2006.

[10] H. Saiedian (Ed.), "An invitation to formal methods," *IEEE Computer*, 29(4), pp. 16-30, April 1996.

[11] N. Halbwachs, P. Caspi, P. Raymond, D. Pilaud, "The synchronous dataflow programming language Lustre", *Proceeding of the IEEE*, 1991.