Formal Verification of Avionics Software Products

Jean Souyris[1], Virginie Wiels[2], David Delmas[1], and Hervé Delseny[1]

[1] Airbus France S.A.S.
316, route de Bayonne
31060 TOULOUSE Cedex 9, France
[2] Onera / DTIM*
2 avenue E. Belin, BP 74025
31055 Toulouse cedex, France
jean.souyris@airbus.com, Virginie.Wiels@onera.fr,
david.delmas@airbus.com, herve.delseny@airbus.com

Abstract. This paper relates an industrial experience in the field of formal verification of avionics software products. Ten years ago we presented our very first technological research results in [18]. What was just an idea plus some experimental results at that time is now an industrial reality. Indeed, since 2001, Airbus has been integrating several tool supported formal verification techniques into the development process of avionics software products. Just like all aspects of such processes, the use of formal verification techniques must comply with DO-178B [9] objectives and Airbus has been a pioneer in this domain.

Keywords: avionics software, safety, development process, verification, formal verification, Abstract Interpretation, static analysis.

1 Introduction

Industrial context. Avionics software products in onboard computers are major components of the systems of an aircraft. Such software products are developed according to very stringent rules imposed by the DO-178B standard. Of course verification, although being one activity among others, is the heaviest task of the development of an avionics software product. Verification, as defined by DO-178B, is performed by reviews, analyses or tests. The first two ones are purely intellectual while the latter basically consists in executing the program to be verified and in checking whether the results of this execution are those expected.

Airbus technological research in Formal Verification. The above mentioned verification means constituted the state of the art at the time DO-178B was written. During the last decade, new verification techniques coming from research in Computer Science have become usable in the industry of critical embedded software. These techniques are formal and are usually categorized as follows: Abstract Interpretation based static analysis, theorem proving and model-checking.

* Onera is the French aerospace lab and is working with Airbus on methods and certification aspects of formal verification.

A. Cavalcanti and D. Dams (Eds.): FM 2009, LNCS 5850, pp. 532–546, 2009.

Transfer to operational teams. Since 2001, Airbus has been transferring formal verification tools – and associated methods of use – to its teams who develop avionics software. The first set of tools to be transferred have been: Caveat [18], aiT [12] and Stackanalyzer [23] . They are all used for achieving some DO-178B verification objective. This means that they have been *qualified* in the sense of this standard.

The aim of this paper is to show how the development of avionics software could benefit from formal verification techniques far beyond their first use mentioned just above. This paper is based on the synthesis of ongoing technological research work at Airbus, in close cooperation with academic and industrial labs. The various aspects of this research are handled – or have been handled – in the context of the following past or ongoing research projects: DAEDALUS [5], ASTREE [1], THESEE [24], CAT [2], U3CAT [25], ASBAPROD (French civilian aviation project), ES_PASS [11].

Structure of the paper. Section 2 is a quick overview of the development and verification process of a DO-178B conforming avionics product. In section 3, the formal verification technologies used by Airbus are presented, whether already used industrially or close to be. Sections 4 and 5 show what development activities it is possible to base on the use of the tools introduced in section 3, and what are possible development processes including these activities. Considerations about the compliance of these new processes to DO-178B and, beyond, to DO-178C (the standard being defined) are discussed in section 6. Section 7 concludes and introduces future work.

2 DO-178B Compliant Development Process of an Avionics Software Product

The development of avionics software products has to conform to the DO-178B [9] standard. DO-178B does not prescribe a specific development process, it identifies important steps inside a development process and defines objectives for each of these steps. DO-178B distinguishes the development processes from "integral" processes that are meant to ensure correctness control and confidence of the software life cycle processes and their outputs. The verification process is part of the integral processes. In this section, we give an overview of the development and verification processes.

2.1 Development Processes

Four processes are identified:

- The software requirements process develops High Level Requirements (HLR) from the outputs of the system process;
- The software design process develops Low Level Requirements (LLR) and Software Architecture from the HLR;
- The software coding process develops source code from the software architecture and the LLR;
- The software integration process loads executable object code into the target hardware for hardware/software integration.

Each of the above mentioned processeses is a step towards the actual software product, Figure 1 presents the different steps.

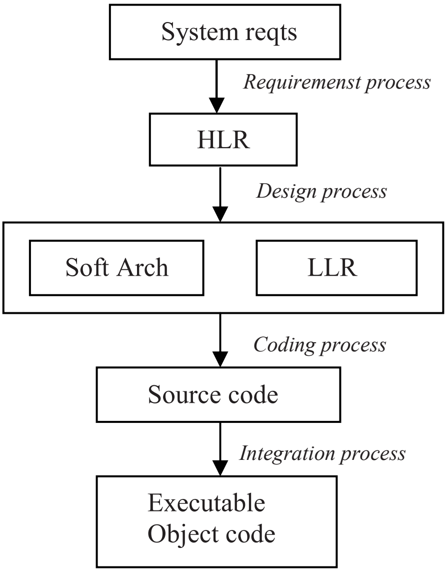

Fig. 1. DO-178B development processes

So far, for the software products it develops, Airbus has been defining the Low Level Requirements as being applied to design entities that are later implemented in the form of modules and functions of the programming language (C, most of the time) in a one-to-one manner. The way those design entities collaborate in order to implement the High Level Requirements is first defined during the software architecture phase.

2.2 Verification Process

The results of all activities[1] of the development must be verified. Detailed objectives are defined for each step of the development, typically some objectives are defined on the output of a development process itself and also on the compliance of this output to the input of the process that produced it. For example, Figure 2 presents the objectives related to LLR. Arrows are labeled with verification objectives; the loop arrow on LLR means the objectives only concern LLR while the arrow between LLR and HLR means that objectives address relationships between LLR and HLR.

On one hand, LLR shall be accurate and consistent, compatible with the target computer, verifiable, conform to requirements standards, and they shall ensure algorithm accuracy. On the other hand, LLR shall be compliant and traceable to HLR.

Verification means identified by DO-178B are reviews, analyses and test. Reviews provide a qualitative assessment of correctness. Analyses provide repeatable assessment of correctness. Reviews and analyses are used for all the verification objectives regarding HLR, LLR, software architecture and source code. Test is used to verify

[1] We use the terms activity and process, a process is a set of activities.

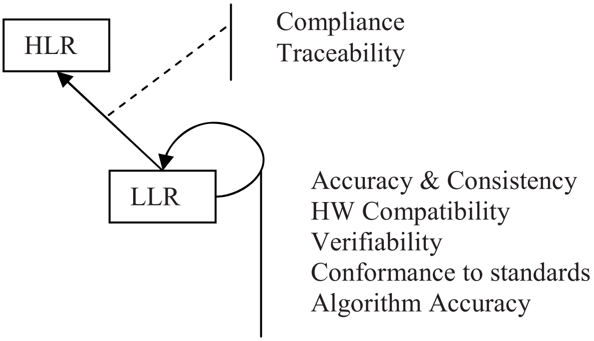

Fig. 2. Verification objectives associated to LLR

that the executable object is compliant with LLR and HLR. Test is always based on the requirements (functional test) and shall include normal range and robustness cases. A structural coverage analysis is performed to ensure that the software has been tested enough (different coverage criteria are used depending on the criticality level of the software).

3 Formal Verification Technologies Applicable to Avionics Programs

In this section, we briefly present the two kinds of formal techniques used for the verification of avionics programs (deductive methods and Abstract Interpretation based static analysis) and we describe the associated tools.

3.1 Deductive Methods

The first kind of formal technique we consider for the verification of programs is deductive proof based on Hoare logic [15], and the computation of Dijkstra's weakest precondition predicate transformer [8]. The objective is to prove user defined properties on a given program. Properties must be formally expressed in logic. This technique proceeds in two steps:

- computation of the verification conditions: post-conditions (properties that should hold after the execution of the program) are defined, this first step analyses the program and computes the conditions that must hold for these post-conditions to be verified;
- proof of the verification conditions: a theorem prover is used to prove the conditions computed before.

The first step is completely automated, the second step usually requires interaction with the user, but automation can be improved by the definition of specific heuristics.

Several tools exist for different programming languages (mostly C and java). The tools considered in this paper are Caveat and Frama-C [14].

3.2 Abstract Interpretation Based Static Analysis

The second kind of techniques is techniques based on Abstract Interpretation [4]. The principle of Abstract interpretation is the construction of a sound approximation of the semantics of programs. A specific approximation is generated for each particular property being analysed. Abstract interpretation is a completely automated technique. It may produce so called "false positives" (errors that can occur on the approximation of the program that has been computed, but cannot occur on the real program). The challenge is thus to be able to build a precise enough approximation in order to have as few false positives as possible. This usually implies a specialisation of the technique with respect to the analysed programs.

The Abstract Interpretation based tools considered in this paper are Astrée [3], aiT [12], Stackanalyzer [23] and Fluctuat [6].

3.3 Tools

We only present briefly the tools, the use of tools in the Airbus process will be described in section 4.

Caveat [18] is the first formal verification tool that Airbus has been using in development (since 2002). Caveat analyses C programs (with some restrictions in terms of language constructs) and has its own specification (or property) language based on first order logic.

Caveat proposes two main functionalities:

- data and control flow analysis;
- proof of user-specified properties.

Data and control flows analyses are fully automatic on the set of C modules given to Caveat.

Proof of user-specified properties is in general not automatic. For completing a proof or understanding why it cannot be completed, the user can use Caveat Interactive Predicate Transformer. This interactive part of the tool takes a first order logic formula as input that the user can handle in order to prove it equivalent to *true* or to understand that it is not possible. Each predicate transformation is performed under the control of the tool.

Frama-C [14] is a toolbox that aims at analysing C programs. It is extensible by means of plug-ins. A plug-in implements a specific analysis and can exchange data with other plug-ins or with the core of Frama-C thanks to a common specification language called ACSL.

Examples of existing plug-ins are:

- Abstract Interpretation based value analysis;
- Slicing;
- Weakest Precondition (WP) computation whose proof obligations are given to the WHY platform of provers.

It must be noticed that the development of simple but useful plug-ins is accessible to industrial Frama-C users.

Whereas Frama-C mixes several techniques coming from research in Computer Science, the following tools are all based on Abstract Interpretation.

Astrée [3] analyses – a subset of - C programs on which it aims at proving the absence of Run-Time Errors (RTE). Since it has been designed in the context of the Abstract Interpretation theory [13], it might produce *false alarms*, also called *false positives*, due to the abstraction of the concrete semantics of the analysed program. In order to make it industrially usable on safety-critical programs, Astrée had to be specialized for a family of programs. This has been made for control-command synchronous programs produced from SCADE (or SAO, SCADE ancestor) models. The result is that Astrée precision is very high (almost zero *false positives*) when analysing programs that belong to the family for which it has been specialized. Scalability is also very good, i.e., 500,000 lines of code are analysed successfully within a timescale compatible with industrial development constraints.

aiT [12] analyses a program in its binary form for computing an upper bound of the Worst Case Execution Time (WCET) of the program tasks. This static analyser contributes to proving that the timing constraints assigned to a program are met. Indeed all kinds of schedulability analyses take the WCET of the tasks of the system as input. Because the execution time of a piece of code also depends on the hardware on which it is intended to be executed, aiT includes a model of the target processor and its associated memory controller. Whereas the drawback of abstraction is the *false positive* in the case of an Abstract Interpretation based static analyser dealing with RTE, the counterpart for aiT is the overestimation of the WCET (upper bound).

Stackanalyzer [23] analyses a program in its binary form for computing an upper bound of the amount of memory actually used by the program task stack. This static analysis contributes to proving that no execution of the program will cause a stack overflow.

Fluctuat [6]. Whereas in mathematics the set of real numbers is infinite, the set of floating-point numbers is finite, be it float, double, etc. So, during the float operations performed by a program, rounding errors affect the results. This might lead to a significant difference between a floating-point value and the real one that should have been computed. Furthermore, a calculus scheme might be stable in the real arithmetic and become instable in the floating-point arithmetic. With respect to this problem, Fluctuat analyses C programs – note that there is a Fluctuat for a specific assembly language (TMS320C33 processor) – for computing safe ranges for:

- The floating-point values the variables still alive at the end of the program may have;
- The error between the floating point value and the real one that should have been computed if operations were in the real numbers, for each variable still alive at the end of the program.

Fluctuat does not only compute these ranges, it also allows the user to find the origin of imprecisions in its code. Problems like lacks of precision, instability, sensitivity are detected by this static analyser.

Certified compilation. There are various approaches for proving that a program in its binary (or assembly) form is semantically equivalent to the source program (in C, for instance) from which it has been compiled. Two of them are being considered: the

Translation validation [19] and the Certified compiler [17]. The first one consists in proving that after each production of a binary file, this executable program is semantically equivalent to the input source program (e.g., in C files). This is a kind of validator separated from the compiler. The second selected approach, i.e., the Certified compiler, consists in developing a compiler formally and proving once and for all that it produces target programs semantically equivalent to source programs.

Certified compilation is of utmost importance in itself, especially for safety-critical software products. It is also natural to consider it when formal verification is performed on source programs. Indeed, a bug of a compiler might lead to produce a code on which some proof of a property made on the source code does not longer hold.

4 Development Process Activities Based on the Use of Formal Techniques

4.1 Operational Use of Formal Methods

Unit Proof [10, 21]. Within the development process of the most safety-critical avionics programs, the unit verification technique is used for achieving DO-178B objectives related to the verification of the executable code with respect to the Low Level Requirements, the classical technique being the Unit Tests. Since 2002, a formal approach to Unit Verification is also used industrially: Unit Proof. The tool used for this activity is Caveat (see section 3.3). Basically, it consists in:

- Writing formal Low Level Requirements in Caveat property language during the detailed design activity of the development process;
- Once a C module has been written during the coding activity, the formal requirements of this C module and the module itself are given to Caveat for proving. This activity is performed for each C function of each C module. When a C function is called by the one being proved it is stubbed according to a sound technique.

Worst Case Execution Time analysis [22]. In real-time systems, computing correct values is not enough. Indeed, the program must also compute these values in due time in order to remain synchronised with the physical environment. The scheduling of the most critical avionics real-time programs is an *off-line scheduling*. This means that the serialisation (single processor) of the various program tasks is performed at design time, leading to a fixed interleaving of these tasks. In this context, schedulability analysis boils down to the safe computation of an upper bound of the Worst Case Execution Time of the program tasks, almost exclusively. This computation is performed with aiT (see section 3.3).

Maximum stack usage computation. The amount of memory given to a task of an avionics program is determined statically when the program is built. If any task stack of a program actually requires more memory than what has been allocated statically, a stack overflow exception is raised during execution. In order to avoid this serious problem, a safe upper bound of each stack of the program must be computed. With these figures, the computation a safe upper bound of the total amount of memory used for stacks is performed, by means of an analysis that takes into account some mechanisms such as interrupt tasks or Operating System calls.

4.2 Envisaged Use of Formal Methods

Integration Proof. The kind of defects that are covered by the Unit Proof technique does not include the ones that arise when a C function calls another one with a wrong interpretation of the service provided by the latter. Let us call this sort of bugs "design bugs" since they are introduced during the activity which aims at defining the interfaces between the future C functions.

Integration Proof is being elaborated in the frame of the research project ASBAPROD and can be defined as an extension of the Unit Proof technique. Indeed, instead of considering C functions individually, Integration Proof deals with sub-trees of the program call tree. Let us consider an example. Suppose four C functions: f(), g(), h() and i(), f() being the entry point of a call-tree (sub-tree of the whole program call-tree) containing the other C functions. Whereas the Unit Proof technique aims at proving that f (), g(), h() and i() satisfy their individual formal requirements without taking into account the semantics of their callees (the C functions they call), the goal of Integration Proof technique is to prove that the formal requirements of function f() are satisfied by taking into account the semantics of all C functions contained in "its" call-tree. The relevant design entities are bigger than the ones considered in Unit Proof but smaller than the whole program. The reason why we did not move from the proof of each C function individually to the proof of the whole – sequential – program made of these C functions is the fact that we want to keep a great automatic proof rate, for obvious industrial reasons. It is a design-time issue to define these intermediate-level entities in such a way that their further proof is as automatic as possible.

Proof of absence of Run-Time error [7, 20]. The underlying notion has been presented in several academic papers, such as [3, §2]: "*The absence of runtime errors is the implicit specification that there is no violation of the C norm (e.g., array index of bounds), no implementation-specific undefined behaviours (e.g., floating-point division by zero), no violation of the programming guidelines (e.g., arithmetic operators on short variables should not overflow the range* `[-32768,32767]` *although, on the specific platform, the result can be well-defined through modular arithmetic).*"

This includes checking that no floating-point overflow can occur, as suggested by DO-178B. So far, this need has been addressed through a combination of design and coding guidelines, testing activities and source code reviews. Today, the ASTRÉE static analyzer makes it possible to perform sound global proofs of absence of run-time errors on complete applications. The analysis process is highly automatic, especially when dealing with Airbus large control programs, generated from SCADE models.

Quality of floating-point calculus [6]. Freedom from run-time errors is not enough when dealing with complex control programs that make massive use of floating-point arithmetic. The accuracy of computations has to be addressed also, as requested by DO-178B. The usual way to deal with this issue is to conduct:

- a set of dedicated test cases on real hardware;
- intellectual analyses of the numerical precision of all floating-point operations. The goal is to check that the program parts using floating-point arithmetic can only generate negligible rounding errors, and cannot propagate errors on inputs (sensitivity analysis). Such an activity is both time-consuming and error-prone.

Today, the FLUCTUAT static analyser enables us to automate the latter activity in a sound and precise way for libraries of widely-used basic operators of control programs. Besides, this tool can also be used to assess the numerical accuracy of some critical system-level functions, through static analyses of the C code generated from limited sets of SCADE nodes.

Certified compilation. As stated in section 2, the development of an avionics program is made of four basic steps to which verification activities are applied. One step being the production of the object code from the source code by compilation (and production of the absolute binary code), it is natural to think about checking that this step does not introduce bugs. In the "traditional" development process (see section 2), an important verification activity consists in testing the program against its Low Level requirements and, later on, against its High Level Requirements, by execution on the real target (or on a very representative hardware). This verification covers the compiler outputs. With the use of formal verification techniques that apply to source code, the compiler outputs are not included in what is verified; the risk being that proofs made on the source code no longer hold on the binary code. This almost new activity will be supported by the use of either a "Certified" compiler [17] or by a validator [19] in order to prove that source and binary programs are semantically equivalent (see section 3.3).

4.3 Lessons Learnt and Deployment Aspects

We will give here some quantitative data for the techniques that have been deployed operationally at Airbus.

- Stack Analyzer is used on all the embedded software products developed by Airbus teams, on more than 10 projects for A380 and A400M aircrafts. All software developers use it, no specific training is necessary.
- AiT is used on approximately 6 projects and more are on the way. All software developers use it, without specific training, there is one specialised engineer who has more specific knowledge, is responsible for the tuning of the tools and can be consulted for advice by the other engineers. It is important to note that this specialist is not a formal method specialist, but a specialist of execution time estimation.
- Unit proof is deployed on three projects and necessitates a specific three-day training.

In the course of experimenting formal techniques, Airbus has defined five criteria for the choice of the techniques and the conditions of their operational use. These criteria are given and explained below.

- Soundness: the technique used has to be sound, i.e. it does not say a property is true if it might not be true.
- Applicability to the code that will be embedded onboard the aircraft: no specific model has to be developed to perform the verification, it is done directly on the code.
- Usability by "normal" engineers on "normal" computers: formal verification is performed on the computers that are used for software development (no need for super computers) and by the engineers (no need for formal method gurus).
- Ability to optimise an existing industrial process: formal techniques must bring better performances than classical methods.
- Certifiability: the objective is to get certification credits for the use of formal methods.

The three techniques that have been deployed operationally meet these five criteria. Moreover, in some cases, formal methods are the only way to keep the same rigor in verification and have an acceptable precision. For example, for stack analysis and worst case execution time computation, classical methods lead to safe but much overestimated results (because of the complexity of the software), formal tools provide better results that allow an optimisation of hardware resources. Finally, formal methods are used if they are automated. The experimental phase is used to augment automation (by defining proof heuristics for proof techniques or code annotations for abstract interpretation based techniques), deployment of the technique is done when sufficient automation is reached. Automation brings high efficiency to the maintenance phase, verification can indeed be redone very easily.

The operational deployment of several formal techniques necessarily modifies the verification process and more generally the development process. The next section will present foreseen evolution of the processes.

5 Towards Product-Based Assurance

5.1 Process and Product Based Assurances

In the Process based Assurance, the confidence in the fact that any execution of the software product conforms to the system specification for that product is obtained by the strict observance of DO-178B development process rules. It is the whole development process that allows to get reasonable confidence in the software product. In other words, if a software product is developed by performing the activities prescribed by DO-178B successfully it will be considered as "good for flight" by the regulation authorities. The main reason why DO-178B emphasizes the quality of the development process is that classical verification techniques do not make it possible to prove the absence of software errors.

In the product based assurance, the confidence is obtained by making sure that the software product has the required characteristics (or properties). The most ambitious goal would be to have a set of formal requirements of the program to develop which specify all aspects of the program execution, and to be able to prove that all possible executions of the binary program satisfy these requirements. If it was possible, this would prove that there is no software error.

5.2 Formal Verification Activities and Product Based Assurance

Executability. By this term we refer to the ability of the program to have well defined behaviours with respect to:

- The “ISO/IEC 9899:1999 (E)" standard (including IEEE 754 standard [16]);
- Specific coding and code generation rules;
- Timing constraints;
- Numerical precision constraints;
- Synchronisation / communication mechanisms.

It must be noticed that without proving the above properties, any proof of user defined requirements (see below) by partial formal verification techniques might be invalidated by some undefined behaviour. Therefore, whenever a formal verification technique not covering the detection of undefined behaviours is used, additional activities must be performed, either based on tools or on intellectual analyses.

Formally proving the executability of a program is the basic kind of Product based assurance. Furthermore, most of the tools mentioned in section 3 are able to analyse whole applications.

Proof of user-defined requirements. So far, there is no cost-effective industrial technique able to verify that whole avionics C programs satisfy their user-defined requirements formally. As stated in section 4, Unit Proof and Integration Proof techniques aim at such formal verification but on program pieces taken individually. The fact that the pieces considered in the Integration Proof technique are bigger than the ones of the Unit Proof technique leads to a better coverage but cannot stand for a formal verification of the whole application with respect to its High Level Requirements.

Nevertheless, we can look at the program pieces which are formally verified as intermediate software products, each of them being specified formally, and then consider such verifications as an application of the Product based assurance paradigm within the development process (Process based assurance).

Certified compilation. As stated in section 4.2, evidences that proofs performed at source code level still hold on the executable program is mandatory. This is another way of saying that in the Product based assurance, the actual software product is the executable program.

5.3 Mix of Formal Verification and Tests

Checking real program executions on real hardware will always be required by DO-178. The basic reason for that is the activity called software/hardware integration.

Beyond this reason, one must also take into account that the huge test campaigns performed on the real hardware during the “traditional” avionics software development process also allow to detect hardware defects. Indeed, most of the time, especially for flight control functions, both hardware and software are developed almost from scratch when a new aircraft is developed. This means that software tests on the real – new – hardware contribute to achieve hardware maturity earlier.

This second reason makes the reduction of the amount of tests an issue. It is clear that the test amount will not be reduced down to the sole software / integration tests.

Therefore, a trade-off between tool-aided formal verification and testing will have to be set, which combines the main advantages of both kinds of techniques, i.e., the automation and good coverage on the one hand, maximal representativity of the tests by execution on the hardware, on the other hand.

5.4 Development Processes Including Some Product Based Assurance

A "traditional" DO-178B conforming process could use static analysers to replace or strengthen some intellectual analyses in order to prove executability (see above). Since some static analysers deal with source code, one must trust the compilation in order to get sure that proofs still hold on the binary code.

Another way of improvement is to introduce Unit Proof technique (see section 4) for formal verification of the source code against its Low Level Requirements. One can also introduce Integration Proof technique (see section 4) for formal verification of the source code against its Low Level Requirements. In both cases, certified compilation is a way to secure the formal verification process.

Actually, there are many ways to introduce formal verification techniques in an avionics development process. An important criterion of such an introduction is whether a "certification credit" is based on the use of a technique or not. So far, Airbus has always been introducing formal verification techniques from which a certification credit has been derived. Nevertheless, it might be the case that some formal verification technique could be used for debugging rather than for achieving some DO-178 objective.

6 Certification Aspects

In this section, we will highlight the specificities of the certification process when formal methods are used for part of the verification. We consider certification with respect to DO-178B, the current software certification standard for avionics software. DO-178 is currently being updated by a dedicated international working group, version C of the standard should be available in 2010, we will end the section with a brief presentation of the current proposal regarding formal methods.

Several cases exist for what concerns certification:

- Formal techniques are used in places where reviews or analyses were used previously to reach the same verification objective. In that case formal methods are simply an alternative means to reach the objective, the main difference being that formal techniques are implemented by a tool and so this tool must be qualified with respect to DO-178B rules for the qualification of verification tools. More information on qualification of tools is given in subsection 6.1.
- Formal techniques replace verifications that were previously done by test.

- A first difference that occurs is that the verification is thus done on the source code instead of the object code. To reach the same level of confidence than with test, complementary analyses must be performed to ensure that the properties that are verified on source code are still satisfied by the object code (this can be done using formal methods also, see the work on certified compilation in section 3).
- The most complex case for certification is the unit or integration proof case, where formal methods are used to verify properties of a C program and replace unit test or integration test. The issue here is the coverage of the verification with respect to the c code. This issue is discussed in subsection 6.2.

6.1 Qualification of Tools

DO-178B distinguishes two kinds of tools: development tools that have an effect on the code being produced (for example code generators) and verification tools that are used to verify some properties on the code (but cannot insert errors). Formal methods tools have to be qualified as verification tools. It must be shown that the tool complies with its operational requirements under normal operational conditions. In practice, it means that a set of representative cases will be defined and it will be checked that the tool provides the expected results for these cases. Stackanalyzer, aiT and Caveat have been qualified as verification tools. No specific requirements are defined for formal method tools in DO-178B, but it might change in version C of the standard where the current proposal is to add objectives targeted for this kind of tools.

6.2 Coverage

When test is used to verify a function against its requirements, a set of requirement-based test cases are defined and executed. A functional coverage analysis is performed to ensure that test cases have been defined for every requirement and a structural coverage analysis is performed to ensure that all the code has been covered and that there is no dead code (for level A software, the most critical one, 100% MC/DC [13] is required). When formal proof is used to verify a C function against a set of properties, it ensures an exhaustive coverage for a given property, but it must also be demonstrated that the set of properties that has been defined covers all the behaviours of the code.

In the case of the unit proof, the argument provided to the certification authorities was based on a demonstration that the set of properties was complete (demonstration using formal proof and reviews). In the case of integration proof, the argument is still the object of research. The general issue that will have to be solved is to be able to measure the coverage of the code obtained by formal verification, and in some cases to be able to mix it with a coverage obtained by test in order to argument the complete coverage of code for certification authorities.

6.3 DO-178C

The update of DO-178 will leave the core of the standard mostly unchanged but will propose several technical supplements dealing with the use of specific techniques such as object-oriented languages or formal methods, a technical supplement on tools is also expected. The current draft of the formal method technical supplement defines what formal methods are, gives criteria for such methods, explains how and under which conditions formal methods can be used to reach DO-178 verification objectives at each step. For verification objectives on the executable object code, it replaces the testing objectives by more generic verification objectives, some testing is still required but some objectives can also be reached using formal methods.

7 Conclusion and Future Work

In this paper it has been shown how formal verification techniques can be used in the development process of avionics software products.

The authors are convinced that the story is far from being finished and that more and more formal verification techniques will be used in the future, as tools become available for industrial use. These techniques are the only way to face the dramatic increase of software complexity, especially when safety is at stake. Technological research is therefore continued in the following three areas: Computer Science research, by means of collaboration with labs, contribution to the development of tools and definition of methods of use.

Regulation aspects are a crucial issue for the industrial use of formal methods, the authors are working on means to conform to the standards but also on evolution of standards in the hope to facilitate future use of formal techniques.

Acknowledgements. The authors warmly thank Famantanantsoa Randimbivololona for his careful reading of this paper.

References

1. The ASTREE project (Analyse Statique de logiciels Temps-REel Embarqués). RNTL (2003), `http://www.di.ens.fr/~cousot/projets/ASTREE/`
2. The CAT project (C analysis toolbox). RNTL (2005)
3. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyser. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
4. Cousot, P., Cousot, R.: Basic Concepts of Abstract Interpretation. In: Jacquard, R. (ed.) Building the Information Society, pp. 359–366. Kluwer Academic Publishers, Dordrecht (2004)
5. DAEDALUS project. IST-1999-20527 of the european IST Programme of the Fifth Framework Programme (FP5) on the « validation of software components embedded in future generation critical concurrent systems by exhaustive semantic-based static analysis and abstract testing methods based on abstract interpretation » (DAEDALUS lasted from October 1st, 2000 to September 30th 2002)
6. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 53–69. Springer, Heidelberg (2009)

7. Delmas, D., Souyris, J.: ASTRÉE: From research to industry. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 437–451. Springer, Heidelberg (2007)
8. Dijkstra, E.W.: A discipline of programming; automatic Computation. Prentice Hall Int., Englewood Cliffs (1976)
9. DO-178B/ED-12B. Software Considerations in Airborne Systems and Equipment Certification. RTCA/EUROCAE (1992)
10. Duprat, S., Souyris, J., Favre-Félix, D.: Formal verification workbench for avionics software. In: SIA (ed.) European Congress ERTS 2006 (European Real Time Software). R-2006-01-2A2 (2006)
11. ES_PASS project. ITEA 2 06042 (October 2007), http://www.itea2.org/public/project_leaflets/ES_PASS_profile_oct-07.pdf
12. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real life processor. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 469–485. Springer, Heidelberg (2001)
13. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on Modified Condition/Decision Coverage. NASA/TM-2001-210876 (2001)
14. Frama-C, http://frama-c.cea.fr/
15. Hoare, C.A.R.: An axiomatic basis for computer programming. Communication of the ACM 12(10) (October 1969)
16. The Institute of Electrical and Inc Electronics Engineers. IEEE standard for binary floating-point arithmetic. Technical Report ANSI/IEEE Std 745. IEEE Computer Society, Los Alamitos (1985)
17. Leroy, X.: The Compcert verified compiler, software and commented proof (August 2008), http://compcert.inria.fr/
18. Randimbivololona, F., Souyris, J., Baudin, P., Pacalet, A., Raguideau, J., Schoen, D.: Applying Formal Proof Techniques to Avionics Software: A Pragmatic Approach. In: Woodcock, J.C.P., Davies, J., Wing, J.M. (eds.) FM 1999. LNCS, vol. 1709, pp. 1798–1815. Springer, Heidelberg (1999)
19. Rival, X.: Symbolic Transfer Functions-based Approaches to Certified Compilation. In: 31st Symposium on Principles of Programming Languages (POPL 2004), Venice. ACM, New York (2004)
20. Souyris, J., Delmas, D.: Experimental assessment of astrée on safety-critical avionics software. In: Saglietti, F., Oster, N. (eds.) SAFECOMP 2007. LNCS, vol. 4680, pp. 479–490. Springer, Heidelberg (2007)
21. Souyris, J., Favre-Felix, D.: Proof of properties in avionics. In: IFIP Congress Topical Sessions 2004, pp. 527–536 (2004)
22. Souyris, J., Le Pavec, E., Himbert, G., Jégu, V., Borios, G., Heckmann, R.: Computing the worst case execution time of an avionics program by abstract interpretation. In: 5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, pp. 21–24 (2005)
23. Stackanalyzer, http://www.absint.com/stackanalyzer/
24. Projet 2005 THÉSÉE du RNTL (Réseau National des Technologies Logicielles) de l'ANR
25. Projet 2008 U3CAT de l'Agence nationale de la recherche (ANR)