Experience Report: Using Objective Caml to Develop Safety-Critical Embedded Tools in a Certification Framework

Bruno Pagano
Olivier Andrieu
Thomas Moniot

[1] Esterel Technologies,
8, rue Blaise Pascal,
78890 Elancourt, France
{Bruno.Pagano,Olivier.Andrieu,
Thomas.Moniot}
@esterel-technologies.com

Benjamin Canou[2,3]
Emmanuel Chailloux
Philippe Wang

[2] Laboratoire d'Informatique de Paris 6,
(LIP6 - CNRS UMR 7606),
Université Pierre et Marie Curie, UPMC,
104, avenue du Président Kennedy,
75016 Paris, France
{Benjamin.Canou,Emmanuel.Chailloux,
Philippe.Wang}@lip6.fr

Pascal Manoury

[3] Laboratoire Preuves, Programmes et Systèmes,
(PPS - CNRS UMR 7126),
Université Pierre et Marie Curie, UPMC,
175 rue du Chevaleret,
75013 Paris, France.
Pascal.Manoury@pps.jussieu.fr

Jean-Louis Colaço *

[4] Prover Technology S.A.S
21 Rue Alsace Lorraine, 31000 Toulouse, France
Jean-Louis.Colaco@prover.com

Abstract

High-level tools have become unavoidable in industrial software development processes. Safety-critical embedded programs don't escape this trend. In the context of safety-critical embedded systems, the development processes follow strict guidelines and requirements. The development quality assurance applies as much to the final embedded code, as to the tools themselves. The French company Esterel Technologies decided in 2006 to base its new SCADE SUITE 6™ certifiable code generator on Objective Caml. This paper outlines how it has been challenging in the context of safety critical software development by the rigorous norms DO-178B, IEC 61508, EN 50128 and such.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; D.2.1 [*Requirements/specifications*]: Tools; D.2.5 [*Testing and Debugging*]: Testing tools

General Terms Reliability, Experimentation, Measurement, Verification

Keywords safety critical, DO-178B, Objective Caml, SCADE SUITE 6™ code generator

* This work started while the author was at Esterel Technologies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08. . . $5.00

1. Introduction

The civil avionics authorities defined a couple of decades ago the certification requirements for aircraft embedded code. The DO-178B standard (RTCA/DO-178B 1992) defines all the constraints ruling the aircraft software development. This procedure is included in the global certification process of an aircraft, and now has equivalents in other industrial sectors concerned by critical software (FDA Class III for the medical industry, EN 50128 for railway applications, IEC 61508 for the car industry, etc).

The Esterel Technologies company markets SCADE SUITE 6™ [1] (Berry 2003; Camus and Dion 2003), a model-based development environment dedicated to safety-critical embedded software. The code generator (KCG [2]) of this suite that translates models into embedded C code is DO-178B compliant and allows to shorten the certification process of avionics projects which make use of it. Using such a code generator allows the end user (the one that develops the critical embedded application) to reduce the development costs by avoiding the verification that the generated code implements the SCADE model (considered here as a specification). The verification and validation activities are reduced to provide evidence that the model meets the functional requirements of the embedded application. In this way, a large part of the certification charge weighs on the SCADE framework and this charge is shared (through the tool provider) between all the projects that make use of this technology.

The first release of the compiler was implemented in C and was available in 1999. It was based on a code generator written in an Eiffel's dialect (LOVE) (ECMA 2005) and was, at that time, rewritten in the mainstream C language to avoid the risk of being rejected by certification authorities.

Then, since 2001, Esterel Technologies has investigated new compiling techniques (Colaço and Pouzet 2003) and language ex-

[1] SCADE stands for *Safety Critical Application Development Environment.*

[2] KCG stands for *qualifiable Code Generator*.

tensions (Colaço et al. 2005). The aim was to demonstrate that using an academic approach for the specifications of the language (types systems, etc.) and Objective Caml (OCaml) for its implementation was also an efficient and clean approach for an industrial project The project quickly led to the expected good technical results but took some time to convince managers that such an approach should be accepted by a reasonable certification authority. It has now appeared that OCaml allowed to significantly reduce the distance between the specifications and the implementation of an engineering tool, to have a better traceability between a formal description of the input language of the tool and its compiler implementation. Thus, Esterel Technologies has designed its new SCADE SUITE 6™ in OCaml.

This paper describes the specific development activities performed by Esterel Technologies to certify KCG with the several norms: DO-178B, IEC 61508 and EN 50128. The differences and particularities of these standards are not in the scope of this paper; for convenience, we focus on the FAA standard (DO-178B, level A).

2. Certification of safety critical code

The well known V-cycle dear to the software engineering industry is the traditional framework of any certified/qualifiable development project. Constraints are reinforced by DO-178B but the principles stay the same: the product specifications are written by successive refinements, from high level requirements to low level design and then implementation. Each step involves several independent verification activities: checking complete traceability of requirements between successive steps, testing each stage of code production with adequate coverage, code and coding rules reviews.

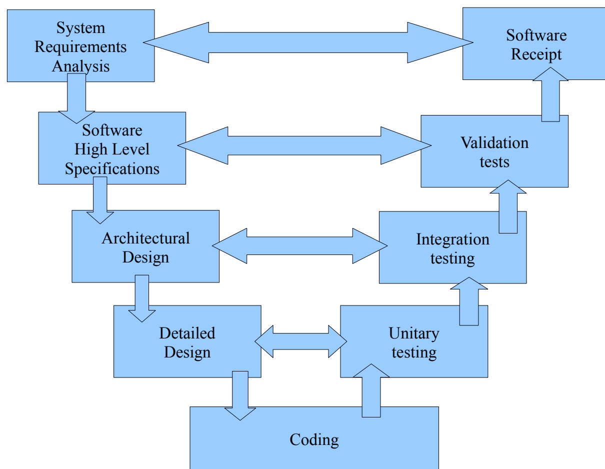

Figure 1. V-cycle

2.1 The programming language in the development process

Traceability is one of the keywords of the compliance to DO-178B, any part of any activity of the project cycle pertains to other parts of the previous and of the following activities. For instance, any requirement of the detailed design has to be related to one or several requirements of the architecture design and to the lines of code implementing this requirement. Furthermore, the relation has to exist with the corresponding verification activity. In our example, the detailed design requirement has to be related to unitary tests that exercise this requirement. The evidence of these relations is one of the most important documents of a certification file.

In a DO-178B compliant project, to ensure that the software satisfies all the requirements and that any single line of the software is necessary to its purpose, nothing can appear in the code without being clearly specified and identified first with a good traceability to high-level requirements (the specifications). These traceability links pass through architectural design and detailed design requirements.

The choice of a programming language close and adapted to the software to develop is very important since a well-suited language leads to a simpler and more direct way to encode the software requirements and consequently, a better and simpler traceability.

In the same vein, when the programming language is adapted to the developed software, the architecture of the software is close to the functional description of the software. The links between architecture and specifications, and between architecture and detailed design are simpler to establish and verify.

The code is tested but it is also reviewed by other developers. To ease this verification, the code must be short (in the sense that it contains more about fundamental algorithms than on resource management) and readable.

Furthermore, the libraries and especially the system library have to be treated in the same way as the main source code: it is mandatory to have the same traceability and verifications on any specific part of the code.

So, the choice of a suitable programming language is relevant for the various verification activities required in DO-178B compliant projects. This is, of course, always true, but becomes crucial when one has to defend a project in front of a certification authority.

2.2 Code coverage

The DO-178B defines several verification activities and, among these, a test suite has to be constituted to cover the set of specifications of the software in order to verify and to establish the conformity of their implementation. As any activity during a DO-178B compliant development process, the verification activities are evaluated. Some criteria must be reached to decide that the task has been completed. One of these criteria is the activation of any part of the code during a functional test. On this particular point, more than a complete structural exploration of the code, the DO-178B standard requires that a complete exploration of the control flow has to be achieved following the *Modified Condition / Decision Coverage* (MC/DC) measurement that we explain below.

- A *decision* is the Boolean expression evaluated in a test instruction to determine the branch to be executed. It is covered if tests exist in which it is evaluated to `true` and `false`.
- A *condition* is an atomic subexpression of a decision. It is covered if there exist tests in which it is evaluated to `true` and `false`.
- The *MC/DC* requires that, for each condition c of a decision, there exist two tests which must change the decision value while keeping the same valuations for all conditions but c. It ensures that each condition can affect the outcome of the decision and that all contribute to the implemented function (no dead code is wanted).

The MC/DC is properly defined on an abstract Boolean data flow language (Hayhurst et al. 2001) with a classical automata point of view. The measure is extended to imperative programming languages, especially the C language, and is implemented in verification tools able to compute this measure.

The challenging consequences of the choice of OCaml instead of the usual C or ADA on MC/DC test campaigns is described in section 4.

2.3 Source to object code traceability

The code verification takes place essentially on the source code. But, the real need is to assert that all verified properties of the source code are also properties of the object code and, indeed, the executable binary. Most of the time these verifications activities are neither possible to do on the binary, nor on the object file.

To handle this contradiction, the process requires to verify that the properties of the source code are also properties of the object code. The compiler analysis focuses on three points:

- *the traceability of the object code generation:* by transitivity, one can deduce from that, the traceability of the requirements in the object code.
- *the management of the system calls:* processes for safety critical applications are very suspicious about calls of system subroutines.
- *conservation of the control flow:* the code coverage measurement is relevant if and only if the control flow is traceable from source to object code.

More than the choice of a programming language, a DO-178B project manager has to choose the complete development suite, integrating the code generator and test management tools which will be the most convenient to manage all the development activities; including coding but also all the verification activities about this coding.

Section 5 describes how the three above requirements can be addressed.

3. Using OCaml in the development process

OCaml is a functional, imperative and object oriented ML dialect. The development environment provided by INRIA contains a native compiler dedicated to the most common architectures[3].

As a functional language, OCaml is adapted to symbolic computation and so, particularly suitable for compiler design and formal analysis tools which rely mainly on symbolic computation. As well for its bootstrapping (Leroy et al. 2008), OCaml is used in Lucid Synchrone (Pouzet 2006), the à la Lustre language for reactive systems implementation, or the Coq (Project 2006) proof assistant implementation. Some years ago Dassault major avionics industry approached the use of OCaml in software engineering for safe real-time programs development. The experience of Surlog with AGFL[4] and the usage of Astrée (Cousot et al. 2005) by AIRBUS industries show that tools written in OCaml can be integrated in a critical software development process.

The Esterel Technologies project presented in this paper is a code generator, named KCG, that translates SCADE models (data-flow with state machines) into embeddable C code. SCADE is a Lustre(Halbwachs et al. 1991) dialect (program directed by equations with time constructions) enhanced by powerful control flow constructions (automata).

KCG has a classical architecture: a front-end with several steps of type-check, a middle-end performing a scheduling and translation of the equational and temporal source language into an imperative intermediate language, and a back-end which generates a bunch of C files. It also contains several optimization passes. A particularity of KCG compared to other compilers resides in its ability to ensure a maximum of traceability between the input model and the generated C program. KCG is specified in a 500 page document containing more than a thousand high-level requirements: one third of them describe the functional requirements of the tool, the others explain the semantics of the input language.

[3] In the sequel OCaml compiler will design the INRIA OCaml compiler

[4] `www.surlog.com`

The high-level requirements that specify the static and dynamic semantics of the Scade language involve logical inference rules. The distance between such a form of requirements and a program written in ML is small and the implementation is very routine, even straightforward for some parts. Indeed:

- the functional abstraction and the modularity of OCaml are high-level enough to be used as architectural requirements (direct traceability).
- the extensive usage of algebraic data types and pattern matching meets the algorithmic description.
- this functional architecture based on well identified compiler phases allows an independent validation of each pass.

As any modern functional language, OCaml benefits from a compiler that produces trustable applications, safer than most of the mainstream languages which require to make use of dedicated verification tools. In particular, the safety of its static typing allows to skip some verifications that would be mandatory with other languages: among the most evident are the memory allocation, coherency, initialization checks, which are no longer relevant and can therefore be omitted when using OCaml.

The OCaml code is compact, which allows to concentrate the verification efforts on the real difficulties, i.e. the algorithmic ones, and very little efforts are devoted to data encoding or resource management issues.

On the other hand, some of the high-level constructs of this programming language may have a bad incidence on the verification activities. We decided not to support the complete OCaml language, and thus forbade or restricted the usage of the most complex parts:

- the object-oriented paradigm is not used for the reason that the control it offers is very difficult to manage statically,
- modules and functors constructions are allowed but without some unnecessary constructs such as the manifest types and other artifacts,
- exceptions and higher-order constructions are restricted by specific coding rules to avoid complex behaviors that would otherwise be hard to verify.

While using OCaml in a development process has undeniable advantages, it remains to answer the specific requirements of the safety-critical software context. This point is addressed in the two following sections.

4. Code Coverage for OCaml programs

An OCaml program such as KCG uses two kinds of library code: the OCaml *standard library*, written mainly in OCaml, and the *runtime library*, written in C and assembly language. Both are shipped with the OCaml compiler and linked with the final executable. The difficulty of specifying and testing such low-level library code led us to adapt and simplify it.

The bulk of the modifications of the *runtime library* was to remove unessential features according the coding standard of KCG. In particular, the support for concurrency and serialization was removed.

Most of the work consisted in simplifying the efficient but complex memory management subsystem. We successfully replaced it by a plain Stop&Copy collector with a reasonable loss of performance.

As most of the *standard library* is written in plain OCaml, its certification is no more difficult than that of any OCaml application.

Regarding the OCaml part, we developed a tool, called *mlcov*[5], capable of measuring the MC/DC rate of OCaml programs. The tool first allows to create an instrumented version of the source code that handles a trace file. Running the instrumented executable then

[5] http://www.esterel-technologies.com/technology/free-software/

leads to (incremental) update of the counters and structures of the trace file. Finally, the coverage results are presented through HTML reports generated from the trace file.

MC/DC for OCaml sources Since OCaml is an expression language, we have to address the coverage of expression evaluation: we state that an expression has been covered as soon as its evaluation has ended. Expressions are instrumented with a mark allowing to record by side-effect that this point of the program has been reached. Some constructions of the OCaml language (such as `if then else`) may introduce several execution branches. Coverage of expressions entails tracing the evaluation of each one of the branches independently. These transformations are detailed in (Pagano et al. 2008).

The* mlcov *implementation The *mlcov* tool is built on top of the front-end of the OCaml compiler. For our specific purposes, a first pass is done, prior to the instrumentation stage, in order to reject OCaml programs that do not comply with the coding standard of KCG.

The figure 2 shows a source code annotated according to test programs: conditions in light gray fulfill the MC/DC criterion, while those in dark gray are not completely covered. And the figure 3 gives the structural coverage and MC/DC statistics for these tests.

```
let all_positive1 a b c =
  (a > 0) && (b > 0) && (c > 0) ;;
(* all_positive1 1 1 1 ;; *)
(* all_positive1 1 1 0 ;; *)

let all_positive2 a b c =
  (a > 0) && (b > 0) && (c > 0) ;;
(* all_positive2 1 1 1 ;; *)
(* all_positive2 1 0 1 ;; *)
(* all_positive2 1 1 0 ;; *)

let all_positive3 a b c =
  (a > 0) && (b > 0) && (c > 0) ;;
(* all_positive3 1 1 1 ;; *)
(* all_positive3 0 1 1 ;; *)
(* all_positive3 1 0 1 ;; *)
(* all_positive3 1 1 0 ;; *)
```

Figure 2. Annoted source code

Performance Results Performances are good enough for code coverage analysis since this activity mainly consists in applying a lot of pretty small examples targeting specific requirements.

5. Traceability from sources to binaries

A DO-178B level A software development imposes to give evidence about the trustability of the tools and compilers used in the process. To reach this goal, we expertized the OCaml compiling process in order to set up hints for the traceability from the source code to the object code. On the basis of this expertise, among other required documentation, test sets have been produced and are part of the bunch of documents for the certification of KCG.

We present in this section the guidelines of this study, mainly focused on two points: the safe management of system calls and the traceability of control flow.

Structural coverage statistics

Function name	Covered points	Total points	Percentage
all_positive1	1	1	100 %
all_positive2	1	1	100 %
all_positive3	1	1	100 %
TOTAL	**3**	**3**	**100 %**

MC/DC statistics

Decision number	Covered conditions	Total conditions	Percentage
#1 (all_positive1)	1	3	33 %
#2 (all_positive2)	2	3	66 %
#3 (all_positive3)	3	3	100 %
TOTAL	**6**	**9**	**66 %**

Figure 3. Coverage rates

Actually, not only does the executed object code of an OCaml program consist of the generated code but also includes some service assembly code, the runtime library and the so-called *standard* OCaml libraries. All those components are linked together at the end of the compilation step.

As noticed in section 4, the set of OCaml libraries had been slightly simplified to keep only the ones written in OCaml, thus it falls under the regular treatment of pieces of OCaml code. The runtime library, developed in C, is mainly concerned with garbage collection. A little static assembly code provides mechanisms for external calls to memory management C functions and for exception handling. As for any OCaml application, when compiling KCG, an *ad hoc* piece of assembly code is generated to set the optimized mechanism of functional application of OCaml. The code for all the standard and runtime libraries used in KCG is reasonably compact, especially after the drastic simplification of the GC. External calls are well confined in small static assembly code and no use of the `libc` library can escape from it. So, fulfilling the two requirements cited above (traceability and safe management of system calls) for this part of OCaml programs can be done by following the usual process.

To deal with the generated code, we first benefited from the facts that the source code of the OCaml compiler is *open* and its functional architecture designs a clear process of refining step by step the intermediate languages, from the abstract syntax tree to the assembly code. The OCaml compilation is itself traceable in the sense that all the intermediate rewritings of the source program can be pretty-printed. It is notable that the bootstrapped OCaml compiler itself naturally offers the traceability facilities that were intentionally designed for the KCG code generator (see section 3).

It is possible to stop the OCaml compiling process after the emission of assembly code. Then, one can assemble *by hand* and link all the components, using the same command as the one the compiler would have used, and finally obtain the same executable as the one the full compiling process would have produced. As a consequence, it is enough to establish traceability from source code to assembly code: a test set can consist in a piece of OCaml code as input and its corresponding piece of assembly code as expected output.

At this level of the expertise, three main points had to be taken into consideration:

- the translation of explicit controls of the source code, including pattern matching and exception handling;

- the controls introduced by the compiler itself which are indeed few and have been tracked;
- the so called *primitive functions* which may either be translated to assembly language or generate calls to external functions.

Concerning the first point, rather than an unfeasible full correctness test of the OCaml compiler, we proceeded to a review of its design principles, deep enough to set a methodology able to ensure the above intended properties of the compiler for a given OCaml application (restricted to the coding standard of the project).

Concerning the second point, a detailed review of the code led us to enumerate the few occurrences where tests are generated: memory allocation, call to the GC, division by 0, access to array or string elements and the mechanism of functional application. In each case, either one can design test sets to cover them, or the branching may stop the program in a safe state[6].

Concerning the third point, all the primitives actually appear in the intermediate lambda code and an exhaustive study of their appearance in the generated assembly code has been performed.

6. Conclusion

In the field of safety-critical avionics software, the mainstream programming languages are exclusively C and ADA. Even to develop tools, which are not embedded themselves but which are used to implement embedded applications, the usage of object-oriented programming languages as Java or C++ is not considered relevant due to the complexity of their control flow. The restrictions needed to develop safety-critical Java/C++ software remove all the features that differentiate OOP languages than C/ADA.

At the very beginning of the project, using OCaml instead of C was a challenge; the point was to have a programming language closer to the functional specifications but further away from the executable program. The main risk resided in the problems that could have been met to show the traceability between the different levels of specifications and the binary resulting from the compilation of a highly functional and polymorphic source code. This project has shown that this was not an issue thanks to the good traceability of the OCaml compiler and its compilation schemes.

Another risk was to express and reach a full code coverage with respect to the MC/DC measure. It was managed by the development of a tool and the performing of a classic test campaign, which revealed neither longer nor more expensive than the previous experiences of code coverage involving code generators written in C code. The additional cost of development of a specific tool (*mlcov*) is balanced by a gain when qualifying as a verification tool a software that is completely designed for our purpose.

The new KCG, developed with OCaml, is certified with respect to the IEC 61508 and EN 50128 norms. It is used in several civil avionics DO-178B projects (such as the A380 Airbus plane, for instance) and will be qualified simultaneously to the project qualifications (with the DO-178B, the tools are not qualified by themselves, but by their usage in a project). The project has been accomplished with the expected delays and costs. The software consists in 65k lines of OCaml code, including a lexer and a parser, plus 4k lines of C code for the runtime library. The development team was composed of 6 software engineers and 8 test engineers during almost 2 years. It is a real DO-178B project, yet with only one singularity compared to other tool development in this certification framework: the use of OCaml as the main programming language.

There are others industrial usages of OCaml in some big companies in the field of embedded avionics systems and they have an increasing interest on the usage of this kind of language for building software engineering tools. In the transportation domain, Prover Technology also provides certifiable solutions for automating verification activities. To meet a high level of certification (SIL 4 in IEC 61508 standard) required by these applications, a diversified implementation of some software modules present in the toolchains. This diversification consists in having two implementations each using its own implementation technology and comparing the result. For this purpose, OCaml has been chosen jointly with the mainstream C language. This different approach of certification is another opportunity for functional languages.

The main result for the ICFP community is that the use of our favorite languages to build compilers is starting to be well understood and accepted by industrial processes and certification authorities in the context of software engineering tools. We can be optimistic to see that, in the middle of all the mainstream (and efficient for other purposes) languages, there is a room for functional technologies and culture.

[6] This is not acceptable for embedded code, but it is for development tools in the sense that it ensures that no faulty code will ever be silently produced.

References

Gérard Berry. The Effectiveness of Synchronous Languages for the Development of Safety-Critical Systems. Technical report, Esterel-Technologies, 2003.

Jean-Louis Camus and Bernard Dion. Efficient Development of Airborne Software with SCADE Suite^{TM}. Technical report, Esterel-Technologies, 2003.

Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, Philadelphia, Pennsylvania, USA, oct 2003.

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

P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyser. In *European Symposium on Programming*. LNCS, April 2005.

ECMA. *ECMA-367: Eiffel analysis, design and programming language*. ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA:adr, June 2005.

N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, pages 1305–1320, 1991.

Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson. A Practical Tutorial on Modified Condition/Decision Coverage. Technical report, NASA/TM-2001-210876, May 2001.

Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. *The Objective Caml system, documentation and user's manual – release 3.11*. INRIA, December 2008. URL `http://caml.inria.fr/pub/docs/manual-ocaml/`.

Bruno Pagano, Olivier Andrieu, Benjamin Canou, Emmanuel Chailloux, Jean-Louis Colao, Thomas Moniot, and Philippe Wang. Certified development tools implementation in objective caml. In Paul Hudak and David Scott Warren, editors, *Tenth International Symposium on Practical Aspects of Declarative Languages (PADL)*, volume 4902 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2008.

Marc Pouzet. *Lucid Synchrone version 3.0 : Tutorial and Reference Manual*, 2006. (`www.lri.fr/%7Epouzet/lucid-synchrone`).

The Coq Development Team LogiCal Project. *The Coq Proof Assistant Reference Manual*, 2006. (`coq.inria.fr/V8.1beta/refman`).

RTCA/DO-178B. Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics RTCA*, pages 31,74, December 1992.