Guidance for Using Formal Methods in a Certification Context

Duncan Brown[1], Hervé Delseny[2], Kelly Hayhurst[3], Virginie Wiels[4]

1: Aero Engine Controls, SINC-4, PO Box 31, Derby, DE24 8BJ, England; email: duncan.brown.JVAEC@rolls-royce.com
2: AIRBUS Operation SAS, 316 route de Bayonne, 31060 Toulouse Cedex 9, France; email: herve.delseny@airbus.com
3: NASA Langley Research Center, 1 South Wright St, MS 130 Hampton VA 23681 USA; email: kelly.j.hayhurst@nasa.gov
4: ONERA/DTIM, 2 avenue Edouard Belin, BP 74025, Toulouse cedex 4, France; email: virginie.wiels@onera.fr

Abstract

This paper discusses some of the challenges to using formal methods in a certification context and describes the effort by the Formal Methods Sub-group of RTCA SC-205/EUROCAE WG-71 to propose guidance to make the use of formal methods a recognized approach. This guidance, expected to take the form of a Formal Methods Technical Supplement to DO-178C/ED-12C, is described, including the activities that are needed when using formal methods, new or modified objectives with respect to the core DO-178C/ED-12C document, and evidence needed for meeting those objectives.

Keywords

Certification, aeronautics, formal methods

1. Introduction

DO-178B/ED-12B, *Software Considerations in Airborne Systems and Equipment Certification* [1], is the current basis for software assurance in the civil aeronautical domain. Formal methods can be applied to many of the development and verification activities required for software used in this domain. When DO-178B/ED-12B was published in 1992, formal methods were briefly mentioned as a possible alternative method. Since that time, advances and practical experience have been gained in techniques and tools supporting formal methods, to the extent that many formal methods have become sufficiently mature for routine application on today's avionics products.

For the past four years, RTCA and EUROCAE have sponsored a joint special committee/working group (SC-205/WG-71) to update DO-178B/ED-12B to take into account and facilitate the appropriate use of new software engineering techniques that have emerged since 1992. The committee is updating the core document DO-178B/ED-12B, and four new documents, called technical supplements, are being developed to handle specific topics including tool qualification, object-oriented approaches, model based development, and formal methods. Sub-groups have been created within the joint committee to define these technical supplements. Sub-group 6 is in charge of formal methods. This paper reports on sub-group 6 work and achievements to date towards creating a technical supplement for formal methods.

After briefly synthesizing the current content of DO-178B/ED-12B, this paper first highlights the essential characteristics of formal methods, then describes the objectives proposed for the Formal Methods Technical Supplement (FMTS). The main goal of this supplement is to define how formal methods can be used within a DO-178/ED-12 project.

2. DO-178B / ED-12B

Developing airborne avionics software in compliance with the DO-178B/ED-12B standard is the primary means of securing regulatory approval [2]. DO-178B/ED-12B does not prescribe a specific development process, but instead identifies important activities and design considerations throughout a development process and defines objectives for each of these. DO-178B/ED-12B distinguishes 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 along with configuration management and quality assurance. This section gives an overview of the development and verification processes, since the use of formal methods affects those processes.

2.1 Development processes

Four processes are identified as comprising the software development processes in DO-178B/ED-12B:

- 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 these processes is a step towards the actual software product. Figure 1 presents the relationships among life cycle data items from the development processes.

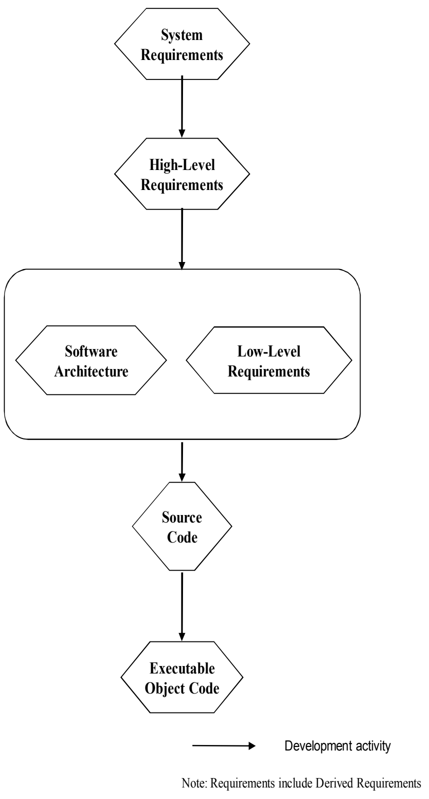

Fig. 1. DO-178B/ED-12B development processes

2.2 Verification process

The results of the four development processes must be verified. Detailed objectives are defined for each step of the development, with some objectives defined on the output of a development process itself and some on the compliance of this output to the input of the process that produced it. Figure 2 presents the verification objectives and activities in relationship with the development artifacts. Each verification activity is depicted by a dashed arrow and the objectives which it satisfies are annotated on the arrows. For example, 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.

DO-178B/ED-12B identifies reviews, analyses and test as means of meeting these verification objectives. 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 that the executable object is compliant with LLR and HLR. Test is always based on the requirements (functional test) and includes normal range and robustness cases.

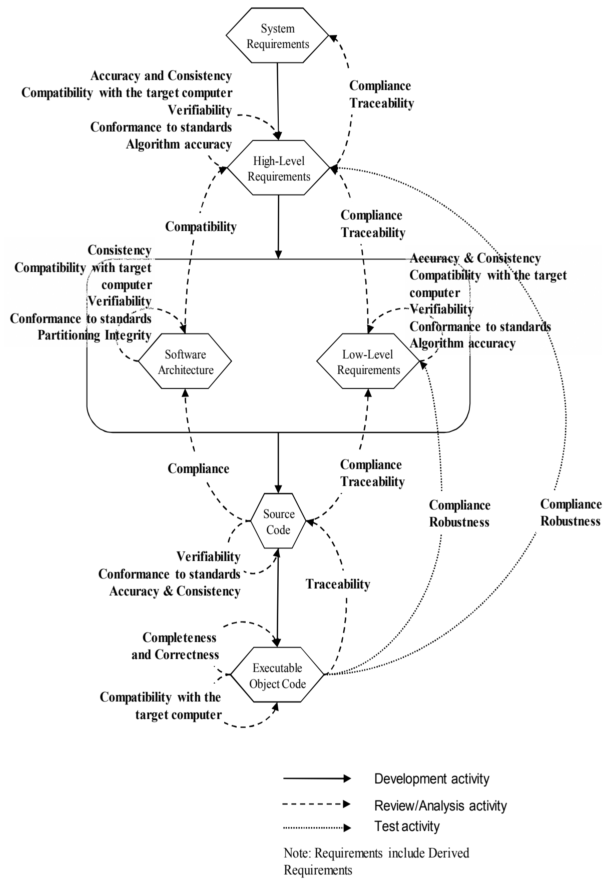

Fig. 2. Verification objectives and activities

In addition to these verification objectives in DO-178B/ED-12B, there are also objectives defined for verification of verification; that is, how to be sure the software has been sufficiently verified. The verification of verification objectives require that a coverage analysis be done. A functional coverage analysis is required to ensure that test cases have been developed for each requirement. A structural coverage analysis is required to ensure that the software structure has been sufficiently exercised, with different coverage criteria depending on the criticality level of the software.

3. Characteristics of formal methods

The proposed guidance for the FMTS begins by defining what formal methods are from the perspective of DO-178B/ED-12B: that is, a formal method is a formal analysis carried out on a formal model. This perspective is important because it permits discussion of formal methods according to the major life cycle processes called out in DO-178B/ED-12B, especially development and verification processes. Development processes are applicable to formal models, and verification processes are applicable to formal analyses.

In general, a model is an abstract representation of a given set of aspects of the software that is used for analysis, simulation and/or code generation. In a certification context, to be formal, a model must have an unambiguous, mathematically defined syntax and semantics. This makes it possible to use automated means to obtain guarantees that the model has certain specified properties.

Although there are important benefits in creating formal models of life cycle artifacts, the most powerful benefits of formal methods are in the formal analysis of those models. Formal analysis can provide guarantees or proofs of software properties and compliance with requirements. Proof, or guarantee, implies that all execution cases are taken into account. For the purpose of the FMTS, an analysis method can only be regarded as formal analysis if its determination of a property is sound. Sound analysis means that the method never asserts a property to be true when it may not be true.

There are many different kinds of formal analysis, but they can typically be classified in three categories: (1) deductive methods, such as theorem proving, (2) model checking, and (3) abstract interpretation.

Deductive methods involve mathematical arguments, such as mathematical proofs, for establishing a specified property of a formal model. A correct proof of a property provides rigorous evidence of that property for the formal model. These proofs are typically constructed using an automated or interactive theorem proving system. Even with such assistance, constructing proofs can be difficult, or impossible in some cases. However, once a proof is completed, automatic checking of the correctness of that proof is usually trivial.

Model checking explores all possible behaviors of a formal model to determine whether a specified property is satisfied. In cases where the property is not satisfied, a counter-example is generated automatically illustrating where and how the property fails to hold. In some cases, a model checker may not be able to determine if the given property holds; for example, in cases where the complexity of the model exceeds the capacity of the model checker.

Abstract Interpretation is a theory for formally constructing conservative representations (i.e. enforcing soundness) of the semantics of programming languages. In practice, this method is used for constructing semantics-based analysis algorithms to determine dynamic properties of infinite-state programs. With abstract interpretation, a formal model is generated, usually within a tool that is specific to the particular property under analysis. It can be viewed as a partial execution of a computer program which determines specific effects of the program (e.g., control structure, flow of information, stack size, number of clock cycles) without actually performing all the calculations.

4. Development using formal methods

The development artifacts shown in Figure 1 can be specified using a formal model. This is no different, in effect, from using any other language to specify a development artifact, except that using a formal notation allows some of the verification objectives to be satisfied by the use of formal analysis. Formalizing requirements or design may increase the effort required to specify them, compared with using more conventional languages, but may result in additional errors being found and removed earlier during this process due to the additional scrutiny inherent in applying a formal notation.

It is worth noting that not all of the requirements for any given development artifact need to be defined formally when using a formal method. For those requirements which are not formally defined, DO-178/ED-12 guidance should be used.

From the perspective of meeting the DO-178B/ED-12B objectives for development, no special guidance is needed when using formal methods. If the applicant does not plan to use formal analysis in the verification, then there is no need to comply with the FMTS.

In cases where formal analysis is planned, the development activities should ensure that there is sufficient definition in the formal model to verify properties about the artifacts being analyzed. The software architecture can be analyzed independently of low level requirements. In the same way low level requirements can be analyzed independently of high level and architectural requirements. Some methods create a formal model at the source code level, embedding information flow from the design into the code, which is then checked by formal analysis. It may even be possible to apply formal methods to analyze properties of object code. In this case, object code is a formal model whose semantics are treated the same by the formal analysis as they are by the target hardware.

5. Verification using formal methods

With formal analysis, the correctness of life cycle data with respect to a formal model or property can generally be proved or disproved; therefore, formal analysis is able to replace the conventional methods of review, analysis, and test, as specified in DO-178B/ED-12B, for some verification objectives. The guidance proposed for the FMTS details the potential use of formal methods at each development level and gives the conditions for the use of a given formal analysis for a given verification objective.

Meeting the DO-178B/ED-12B objectives for verification of verification process results is more complicated when formal analysis is used. In cases where testing alone is used, verification of verification is accomplished by a coverage analysis to guarantee that software has been tested enough. However, test coverage metrics, such as decision coverage or modified condition/decision coverage, are not meaningful for analyzing the sufficiency of a formal analysis. For the FMTS, an alternative approach to coverage is proposed when formal analyses are used to replace some testing.

5.1 Reviews and analyses

For requirements (HLR and LLR), architecture and source code, the use of formal methods is a particular case of analysis. Thus the “only” guidance needed for formal analysis in these cases is the criteria and conditions for the use of formal methods for each development process; the objectives have not been modified. That is, formal analysis can be used to satisfy the objectives for software reviews and analysis, as shown in Figure 2, as follows:

a. Compliance: If the life cycle data items that comprise the inputs and outputs of a software development process are formally modeled, then formal analysis can be used to verify compliance. Compliance can be demonstrated by showing that the output satisfies the input. Formal methods cannot show that a derived requirement is correctly defined and has a reason for existing; this must be achieved by review.

b. Accuracy: Formal notations are precise and unambiguous, and can be used to demonstrate accuracy of the representation of a life cycle data item.

c. Consistency: Life cycle data items that are formally expressed can be checked for consistency (the absence of conflicts). If a set of formal statements is found to be logically inconsistent then the inconsistencies must be addressed before any subsequent analysis is conducted.

d. Compatibility with the target computer: Formal analysis can be used to detect potential conflicts between a formal description of the target computer and the life cycle data item.

e. Verifiability: Being able to express a requirement in the formal notation defined in the software verification plan is evidence of verifiability in the same way as being able to define a test case. In some cases, formal analysis is better able to verify a requirement then testing. For example, requirements involving “always/never” cannot in general be verified by a finite set of test cases, but may be verified by formal analysis.

f. Conformance to standards: Life cycle data items that are formally expressed must be compliant with any standards defining the formalism. Invalid results will be obtained if ill-formed requirements are allowed. Since formal notations have a well-defined syntax, automated syntax checkers are appropriate for verifying that the formally stated requirements are well-formed with respect to syntax. In addition, an automated checker may enforce other restrictions on the notation (e.g., complexity of notational constructs and other design constructs that would not comply with the system safety objectives). Automated checking will need to be supplemented by review for those standards not amenable to automated checking.

g. Traceability: Traceability ensures that all input requirements have been developed into lower level requirements or source code. Traceability from the inputs of a process to its outputs can be demonstrated by verifying with a formal analysis that the outputs of the process fully satisfy its inputs. Traceability from the outputs of a process to its inputs can be demonstrated by verifying with a formal analysis that each output data item is necessary to satisfy some input data item.

h. Algorithm aspects: If life cycle data items are formally modeled, then algorithmic aspects can be checked using formal analysis.

i. Requirement formalization correctness: If a requirement has been translated to a formal notation as the basis for using a formal analysis, then review or analysis should be used to demonstrate that the formal statement is a conservative representation of the informal requirement. It is important to note that the preciseness of formal notations is only an advantage when they maintain fidelity to the intent of the informal requirement. If the semantic gap between an informal statement of the requirement and its embodiment in a formal notation is too large, then establishing that the formal statement is conservative may be difficult.

5.2 Test

Using formal analysis to meet the verification objectives for executable object code is the biggest challenge for providing guidance because test is the only means envisaged in DO-178B/ED-12B for meeting those requirements. Replacing all testing at this level by formal analysis is not possible at this time; but formal methods can replace test for some properties, such as worst case execution time or stack usage. Formal methods can also be used to verify the compliance of the executable object code with respect to high level or low level requirements. However, testing will always remain mandatory and will be the primary means for verification of the executable object code.

Because formal methods cannot replace all testing, verification objectives for executable object code are the same when using formal methods as for test in DO-178B/ED-12B. However, an additional objective, as follows, is needed when formal analysis is used to verify properties of the executable object code:

- Analysis of property preservation between source code and object code. For the formal analysis of source code to be used as verification evidence for the target system, verification should be performed to establish property preservation between source and object code. By verifying the correctness of the translation of source to object code*,* formal analysis performed at the source code level against high or low level requirements can be used to infer correctness of the object code against high or low level requirements. This is similar to the way that coverage metrics gained from source code can be used to establish the adequacy of tests to verify the target system.

This allows for alternative verification paths for the executable object code, as shown in Figure 3. For example, compliance of object code to LLR can be demonstrated using formal analysis on source code and analysis of property preservation between source code and object code.

5.3 Verification of verification

Coverage analysis in DO-178B/ED-12B is defined in two steps:

- requirements-based coverage analysis, to ensure that test cases exist for each requirement;
- structural coverage analysis, to ensure that the code structure has been sufficiently exercised by test cases.

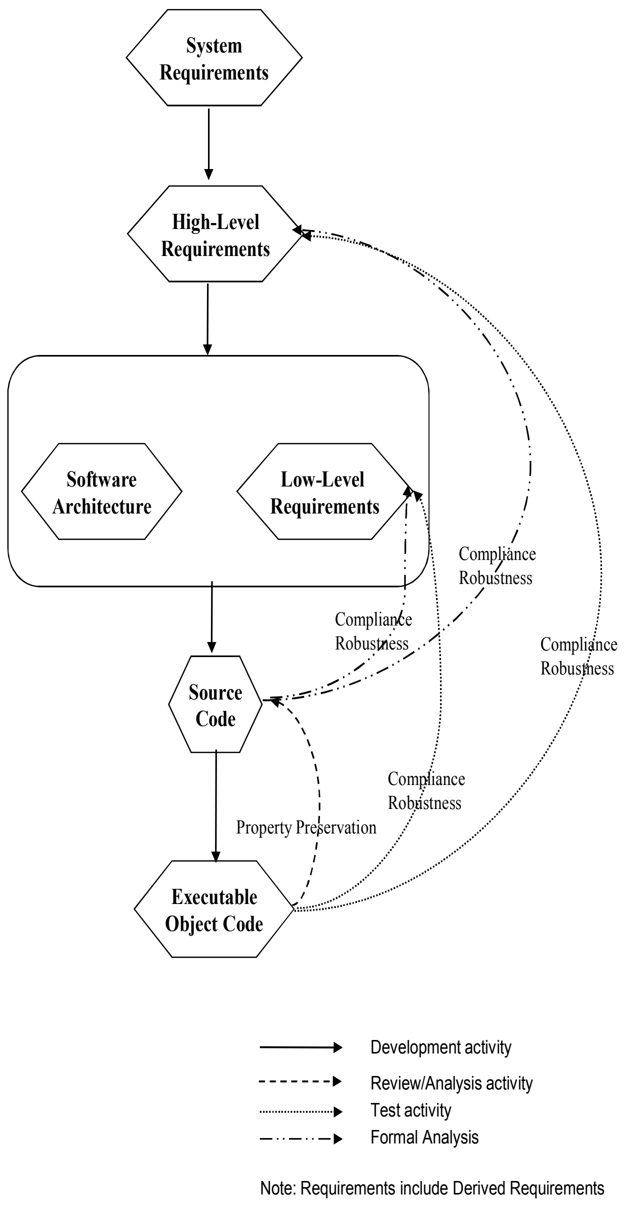

Fig 3. Alternative verification paths

Requirements-based coverage analysis can be done directly when using formal methods by showing that formal analysis cases exist for each requirement.

Structural coverage however is a notion strongly connected with test and execution of code. To be able to define an alternative to structural coverage analysis, the objectives of structural coverage analysis have to be examined. These objectives, as given in DO-178B/ED-12B, are to detect:

- shortcomings in requirements-based cases or procedures,
- inadequacies in software requirements,

- dead or deactivated code.

When a code structure is not covered by some test cases, then either some test cases are missing for a given requirement, a requirement is missing, or the code structure is dead or deactivated.

When using formal methods, verification is exhaustive, so a requirement that has been verified formally has been completely covered. However, there is no guarantee that a requirement has not been forgotten. Thus, it is necessary to have a means to ensure the completeness of the set of requirements. Similarly, it is necessary to have a means to ensure that there is no unidentified dead or deactivated code.

Consequently, the solution proposed to provide coverage when using formal analysis at the executable object code level is meet the following set of fours objectives:

<u>Complete Coverage of Each Requirement</u>

The most thorough verification would be where all possible paths through the code with all possible data values are considered. Formal methods ensure this but sometimes the formal analysis requires assumptions to be made about the software system. All such assumptions should be verified.

<u>Completeness of the Set of Requirements</u>

Where requirements are formally modeled, it should be demonstrated that the set of requirements is complete with respect to the intended functions. That is:

- For all input conditions the required output has been specified.
- For all outputs the required input conditions have been specified.

If the set of requirements is found to be incomplete then additional requirements (generally not derived requirements) should be added. If a demonstration of completeness cannot be achieved, then structural coverage analysis must be used.

<u>Detection of Unintended Dataflow Relationships</u>

Verifying that the information flow in the source code complies with the requirements ensures that there are no unintended dependencies between the inputs and outputs of the code. If unintended dependencies exist then these must be resolved either by adding the missing requirements (generally not derived requirements) or removing the erroneous code.

<u>Detection of Dead Code and Deactivated Code</u>

Guidance for dead and deactivated code is not different when using formal analysis from using test. In both cases, dead and deactivated code should be identified by review or analysis and dealt with as per the guidance in DO-178B/ED-12B.

6. Specific objectives

Previous sections have addressed how formal methods can satisfy the objectives currently defined in DO-178B/ED-12B. The proposed guidance for the FMTS also includes some additional objectives concerning the formal methods that are used.

When formal analysis is used to meet the verification objectives of DO-178/ED-12, the formal method should be correctly defined and justified as follows:

a. All notations used for formal analysis should be verified to have precise, unambiguous, mathematically defined syntax and semantics; i.e., they are formal notations.

b. The soundness of each formal analysis method should be justified. A sound method never asserts that a property is true when it may not be true.

c. All assumptions related to each formal analysis should be described and justified (e.g., those associated with the target computer; those about the data range limits; etc.).

The first objective requires demonstration that the method used is formal. The second one restricts the use of formal methods to sound methods. (Not all formal methods are sound.) The last one focuses on assumptions because assumptions are often used by formal analyses and it is important for the correction of the analysis that they are all justified.

7. Benefits of using formal methods

Formal methods were developed as a branch of computer science in order to reason more scientifically about software. Initially the advantages of formal methods were in analyzing the behavior of source code to understand where this was incorrect. Since those early applications of this approach in the 1970's, the problem of error prone source code has reduced and instead most of the errors in software development are now generally accepted as being attributable to requirement errors.

It has become apparent that creating formal representations or models of requirements can help to address this problem in the same way as it did with source code. In that respect, formal methods have the potential for both increasing safety and decreasing the cost of certifying flight-critical systems. Specific benefits include improving requirements, reducing error introduction, improving error detection, and reducing cost. Secondly, the formality of the description allows us to carry out rigorous analyses. Such analyses can verify useful properties such as consistency, deadlock-freedom, satisfaction of high level requirements, or correctness of a proposed design.

7.1 Improve Requirements

Experience shows that the act of capturing requirements using formal notations is of benefit: it forces the writer to ask questions that would otherwise be postponed until coding. Using a formal notation to capture requirements provides a simple validation check, as it forces a level of explicitness far beyond that needed for informal representations. Requirements expressed in a formal notation can also be analyzed early to detect inconsistency and incompleteness and therefore remove errors that would normally not be found until later in the development process.

7.2 Reduce Error Introduction

Writing high or low level requirements formally improves the quality of the development artifacts. Formalized requirements prevent misunderstandings that lead to error introduction. As development proceeds, compliance can be continually checked using a formal analysis to ensure that errors have not been introduced.

A further advantage of using formal methods at the requirements level is the ability to derive or refine from these requirements the code itself, thus ensuring that no errors are introduced at this stage. Alternatively their use at the requirements level allows formal analysis to establish correctness between requirements and code.

7.3 Improve Error Detection

Formal analysis can provide exhaustive verification at whatever levels it is applied: high level requirements, low level requirements, source code or executable. Exhaustive verification means that all of the structure is verified over all possible inputs and states. This can detect errors that would be difficult or impossible to find using only a test based approach.

7.4 Reduce Cost

In general, software errors are less expensive to correct the earlier in the development lifecycle they are detected. The effort required to generate formal models is generally more than offset by the early identification of errors. That is, when formal methods are used early in the lifecycle, they can reduce the overall cost of the project. When requirements have been formalized the costs of downstream activities are reduced. Formal notations also reduce cost by enabling the automation of verification activities.

8. Conclusion

A primary motivation for developing guidance for using formal methods in a certification context was to better enable the routine use of mature formal methods. That guidance, as proposed to date in a FMTS to DO-178/ED-12, allows adoption of formal methods into an established set of processes for development and verification of an avionics system to be an evolutionary refinement rather than an abrupt change of methodology. Formal methods might be used in a very selective manner to partially address a small set of objectives, or might be the primary source of evidence for the satisfaction of many of the objectives concerned with development and verification.

This paper has presented a synthesis of the effort of the Formal Methods Sub-group of RTCA SC-205/EUROCAE WG-71 in developing guidance for using formal methods in these ways in a certification context with DO-178/ED-12.

Ongoing work in the formal methods sub-group concerns a discussion paper that is intended to give several examples of the use of formal methods.

9. Acknowledgement

This paper presents the work of RTCA SC-205/ EUROCAE WG-71 sub-group 6, we thank all members of SG6.

10. References

[1] RTCA/DO-178B, EUROCAE/ED-12B: Software Considerations in Airborne Systems and Equipment Certification, December 1, 1992.

[2] Advisory Circular # 20-115B. U. S. Department of Transportation, Federal Aviation Administration, issued January 11, 1993.