Verification of Air Data Computer Software using Formal Methods

JianXu1*, Xinai Zhang2, Yi Zhao3 and Bing Xu4

1COMAC Shanghai Aircraft Design&Research Institute*, Shanghai, 201210, China
2COMAC Shanghai Aircraft Design&Research Institute, Shanghai, 201210, China
3COMAC Shanghai Aircraft Design&Research Institute, Shanghai, 201210, China
4COMAC Shanghai Aircraft Design&Research Institute, Shanghai, 201210, China
5Corresponding author’s e-mail: xujian1@comac.cc

Abstract. With the rapid development of airborne system and software, the requirement of software’s robustness, adaptability and reliability improves continually. Failure of software function may cause severe property loss and personnel casualties. Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems that have obvious advantages in improving safety and reliability of airborne software. This paper takes Air Data Computer software as an example for formal verification in the whole software development process including requirements process, design process, coding process and integration process to demonstrate a set of formal methods application in the airborne software verification process.

1. Introduction

Formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems. The mathematical basis of formal methods consists of formal logic, discrete mathematics, and computer-readable languages. RTCA released DO-333[1] ”Formal Methods Supplement to DO-178C and DO-278A” accompany with the release of DO-178C[2] ”Software Considerations in Airborne Systems and Equipment Certification” in 2011 which adopted formal methods into an established set of processes for development and verification. The use of formal is motivated by the expectation, as in order engineering disciplines, that performing appropriate mathematical analysis can contribute to establishing the correctness and robustness of a design. 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.

Contently there are many successful formal verification cases adopted in airborne software, for example, formal analysis and verification in Ice Detection software based on Expanded Petri Net[3], formal verification in Component-based Airborne Software[4].

Air Data Computer(ADC) receives input signals from pitots and sensors and then transmits them into the digital data for redundancy management and conversion to air data parameters. The converted air data parameters are then transmitted through the A664, and forwarded to the appropriate functions on the airplane. Air Data Computer Software is an important part of Air Data Computer which
contains the redundancy management of the air data input parameters, computation of air data output parameters, and fault detection and reporting.

This paper is based on the assumption that the ADC software was already developed by traditional means which complied with DO-178C Level A objectives and the whole software life cycle data was already established. Because ADC software is a complex software, only a part of ADC software was chosen for formal verification and then compared with the traditional means.

2. Formal verification objectives and technical paths

2.1. Verification objectives

There are ten tables of objectives in Annex C of DO-333. Because ADC software is a level A software according to DO-178C, the appropriate verification objectives from Annex C of DO-333 is identified and shown in Table 1.

| No | Objective Description | Ref | Activity | Applicability by Software Level |
|----|-----------------------|-----|----------|---------------------------------|
| 1  | Low-level requirements comply with high-level requirements | FM.6.3.a, FM.6.3.2.a, FM.6.3.b | FM.6.3.2 | ● ● ◎ |
| 2  | Low-level requirements are accurate and consistent | FM.6.3.c, FM.6.3.2.b | FM.6.3.2 | ● ● ◎ |
| 3  | Low-level requirements are verifiable | FM.6.3.e, FM.6.3.2.d | FM.6.3.2 | ◎ ◎ |
| 4  | Low-level requirements conform to standards | FM.6.3.f, FM.6.3.2.e | FM.6.3.2 | ◎ ◎ ◎ |
| 5  | Source code complies with low-level requirements | FM.6.3.a, FM.6.3.4.a | FM.6.3.4 | ● ● ◎ |
| 6  | Source code is verifiable | FM.6.3.e, FM.6.3.4.c | FM.6.3.4 | ◎ ◎ |
| 7  | Source code conforms to standards | FM.6.3.f, FM.6.3.4.d, FM.6.3.b | FM.6.3.4 | ◎ ◎ ◎ |
| 8  | Source code is accurate and consistent | FM.6.3.c, FM.6.3.4.f | FM.6.3.4 | ● ◎ ◎ |
| 9  | Formal analysis results are correct and discrepancies explained | FM.6.7.2.c | FM.6.7.2 | ● ◎ ◎ |
| 10 | Verification of property preservation between source and object code | FM.6.7.f | FM.6.7 | ● ● ◎ ◎ |
| 11 | Formal method is correctly defined, justified, and appropriate | FM.6.2.1 | FM.6.2.1.a, FM.6.2.1.b, FM.6.2.1.c | ● ◎ ◎ ◎ |

2.2. Verification technical paths

Figure 1 shows the Level A software formal verification process according to DO-333.
Figure 1. Level A software formal verification process.

In this paper, for level A software, the chosen verification path is shown in Table 2:

Table 2. Selected Formal verification path.

| Verification path | Software process involved | Software data be verified | Attributes be verified |
|-------------------|---------------------------|---------------------------|------------------------|
| Compliance from LLR to HLR | Software Requirement Process | SDD | Compliance |
| Compliance from Source Code to LLR | Software Design Process | Source Code | Compliance |
| Source code non-functional check | Software Code Process | Source Code | Correctness |
| Traceability from EOC to source code | Software Integration Process | EOC | Traceability |

3. ADC Software Module selection

ADC software contains initialization module, functional checking module, fault detection module, ADC data computation module boundary parameter processing module and parameter effectiveness module according to the ADC SRD(software requirement data).

By unscrambling the ADC SRD, SDD(software design description) and source code, the demand relation between formal verification and ADC software is concluded as below:

- For the initialization module and boundary parameter processing module there are only assignment statements but no logic relations so that no formal verification is needed.
- For the functional checking module and fault detection module, there are no logic relations so that no need for formal verification.
• For the ADC data computation module, there are only numerical computation statements thus no need for formal verification.

• For the parameter effectiveness module, there are some relationships between parameters. For example, if the Static Pressure and Total Pressure are invalid, the Fixed Airspeed is invalid. Furthermore, there are loops and determination statements in the source code so that the advantage of formal specification can be demonstrated.

Hence, the parameter effectiveness module is selected for formal verification.

4. Formal Verification Implementation
Choose one HLR(high-level requirement) from the parameter effectiveness module of SRD as an example to demonstrate the whole formal verification process to comply with DO-333. The HLR is:

“The static pressure value shall be invalid when
static temperature voltage fault or
static probe reading fault or
15v voltage power fault or
static pressure value is out of range 10 times in series.
Else,
the static pressure value shall be valid”

4.1. Verification of compliance from SDD to SRS
We use the Event-B tool, Rodin tool, and Refinement Theory to build two layers model for SRS and SDD.

We set the four conditions to BOOL value as shown in Table 3 and build the logic that the static pressure value shall be invalid when any of the conditions occur.

| Condition                                      | BOOL value |
|------------------------------------------------|------------|
| static temperature voltage fault               | ps_volt_fail|
| static probe reading fault                     | ps_rom_fail |
| 15v voltage power fault                         | in_15v_fail |
| static pressure value is out of range 10 times in series | ps_fail |

Build two layers of machines in the Event-B tool which is shown in Figure 2 and Figure 3.

Finally, Proof Obligations Effectiveness in Rodin Tool is used to ensure the consistency of two layers of machines and the correctness of the Refinement.

4.2. Verification of compliance from Source Code to LLR
The VCC tool, Z3 Solver tool and Hoare Logic were used for verifying the compliance from Source Code to LLR.
Firstly, the first order predicate logic was generated by Hoare Logic from LLR which was shown below:

```plaintext
if bit_fail.ps_fd_fail=1 ∨ bit_fail.in_ps_volt_fail=1 ∨ bit_fail.ps_rom_fail=1 ∨
bit_fail.in_15v_fail=1 ∨ fail_count.ps_fail & 0x3ff = 0x3ff then
  data_valid.ps_valid := 0
else
  data_valid.ps_valid := 1
Endif
```

Step 2 is to transmit the the first order predicate logic into comments (function specification, data constraints and loop constraints) which VCC tool can recognize. shown below:

```plaintext
requires(
  mutable(&bit_fail)
  &&mutable(&fail_count)
  &&mutable(&in_air_data)
)
writes(span(&data_valid))
ensures(((bit_fail.ps_fd_fail ==1) || (bit_fail.in_ps_volt_fail ==1) || (bit_fail.ps_rom_fail==1)
|| (bit_fail.in_15v_fail ==1) || (fail_count.ps_fail & 0x3ff) == 0x3ff))
  => data_valid.ps_valid == 0)
ensures(((bit_fail.ps_fd_fail ==0) && (bit_fail.in_ps_volt_fail
==0) && (bit_fail.ps_rom_fail==0)
&& (bit_fail.in_15v_fail ==0) && (fail_count.ps_fail & 0x3ff) != 0x3ff))
  => data_valid.ps_valid==1)
```

Finally, the project used Z3 Solver tool to prove the correctness of the comments in source code which decomposed from the LLR. If all the comments were right, VCC could draw a conclusion that the source code comply with the LLR.

4.3. Source code non-functional check

PolySpace tool was used since it adopts Semantics Analysis Technology to address the source code fault detection.

The researcher firstly located the source code to be checked and their head files into the same folder, followed by running Code Prover function in PolySpace. Figure 4 and Figure 5 shows the result of the experiment. Specially, the green, red, gray colours in Figure 4 represent correct code, incorrect code and unreachable code respectively.

Set status and security attributes to the incorrect code and unreachable code and then generate the problem report.
4.4. Verification of traceability from EOC to source code
For the verification of compliance from EOC to source code, the verification method is as below:

- Disassemble the EOC to order by IDA Pro tool.
- Identify the processor of EOC to obtain the processor order set using IDA Pro tool.
- Generate C code according to EOC order and processor order set manually.
- Compare with the C code with source code for static review.

5. Formal Verification Result and analysis
The comparison of the verification result is shown in Table 4.
There are 97 properties designed in the whole project for the analysis of original software requirements. Analysis of these properties by formal tools uncovered 2 errors in the compliance from source code to SDD, as well as 1 error found in thought uncover through source code level non-functional self-test. Moreover, the test team developed a series of test cases from the same software requirement in the same way. Although the test team spent 50% more time in testing than the formal verification team in module checking, there is no error found in the software. The conclusion of the experiment was that formal verification was more effective and efficient than the test in designing error under this case study. Besides, the team of formal verification found that lots of errors were difficult or even impossible to find. For example, the error that a value out of bounds problem because of complex computation finally results in the program crash.

This case shows that the formal test is more effective and efficient than the traditional test in uncovering some errors, although the test can only check a small part of possible input and status. With the help of automation tools of formal verification, the workload in formal verification is much less than those who in verification based on the test. It is easy to find that the whole project case fully shows the effect of the formal methods in software safety and reliability.

| Verification module                      | Sub-function Instance | Number of Properties | Errors found by Formal Verification | Errors found by Testing |
|-----------------------------------------|-----------------------|----------------------|------------------------------------|------------------------|
| Compliance from LLR to HLR              | 2                     | 48                   | 0                                  | 0                      |
| Compliance from Source Code to LLR      | 2                     | 49                   | 2                                  | 0                      |
| Source code non-functional check        | 2                     | 0                    | 1                                  | 0                      |
| Traceability from EOC to source code    | 2                     | 0                    | 0                                  | 0                      |
| Totals                                  | 8                     | 97                   | 3                                  | 0                      |

6. Conclusion
This paper selects the parameter effectiveness module of ADC software for formal verification, and plans to start formal verification with four aspects include formal description of requirements specifications and design documents, checking the consistency between requirements and source code, non-functional verification of source code, verification of compliance between object code and source code. And be carried out at the requirements level, design level, source code level and object code level. Through this project, the effect of formal methods in the airworthiness certification process of airborne software can be demonstrated, and it will be used as a case analysis of the DO-333 formal methods supplementary document to generate a set of theory system of formal methods application in the airborne software development process.
References

[1] RTCA. Formal Methods Supplement to DO-178C and DO-278A, DO-333. Washington, D.C: RTCA Inc, 2011.

[2] RTCA. Software Considerations in Airborne Systems and Equipment Certification, DO-178C. Washington, D.C: RTCA Inc, 2011.

[3] Li Z, Liu B, Lu M Y. (2012) Modeling and Verification of Deicing Software Safety Requirement Based on Expanded Petri Net. Journal of Beijing University of Aeronautics and Astronautics, 38:1.

[4] Xu B F, Huang Z Q, Hu J. (2012) Model-driven Safety Dependence Verification for Component-based Airborne Software Supporting Airworthiness Certification. Acta Aeronautica et Astronautica Sinica, 33:1.