

# Testing or Formal Verification:

## DO-178C Alternatives and Industrial Experience

Yannick Moy, AdaCore

Emmanuel Ledinot, Dassault-Aviation

Hervé Delseny, Airbus

Virginie Wiels, ONERA

Benjamin Monate, TrustMySoft

*// Software for commercial aircraft is subject to stringent certification processes described in the DO-178B standard, Software Considerations in Airborne Systems and Equipment Certification. Issued in late 2011, DO-178C allows formal verification to replace certain forms of testing. Dassault-Aviation and Airbus have successfully applied formal verification early on as a cost-effective alternative to testing. //*



**AVIONICS IS THE** canonical example of safety-critical embedded software, where an error could kill hundreds of people. To prevent such catastrophic events, the avionics industry and regulatory authorities have defined a stringent certification standard for avionics

software, DO-178 and its equivalent in Europe, ED-12, which are known generically as DO-178. The standard provides guidance—objectives as well as associated activities and data—concerning various software life-cycle processes, with a strong emphasis on verification.

The current version, called DO-178B,<sup>1</sup> has been quite successful, with no fatalities attributed to faulty implementation of software requirements since the standard's introduction in 1992. However, the cost of complying with it is significant: projects can spend up to seven times more on verification than on other development activities.<sup>2</sup> The complexity of avionics software has also increased to the point where many doubt that current verification techniques based on testing will be sufficient in the future.<sup>3</sup> This led the avionics industry to consider alternative means of verification during the DO-178B revision process. The new standard, DO-178C,<sup>1</sup> includes a supplement on formal methods (see the “What Are Formal Methods?” sidebar), known as DO-333<sup>4</sup>, which states the following:

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.

Although this permission to replace part of testing with formal verification is quite new, we've successfully applied this new guidance into a production-like environment at Dassault-Aviation and Airbus. The use of formal verification for activities previously done by testing has been cost-effective for both companies, by facilitating maintenance leading to gains in time on repeated activities.

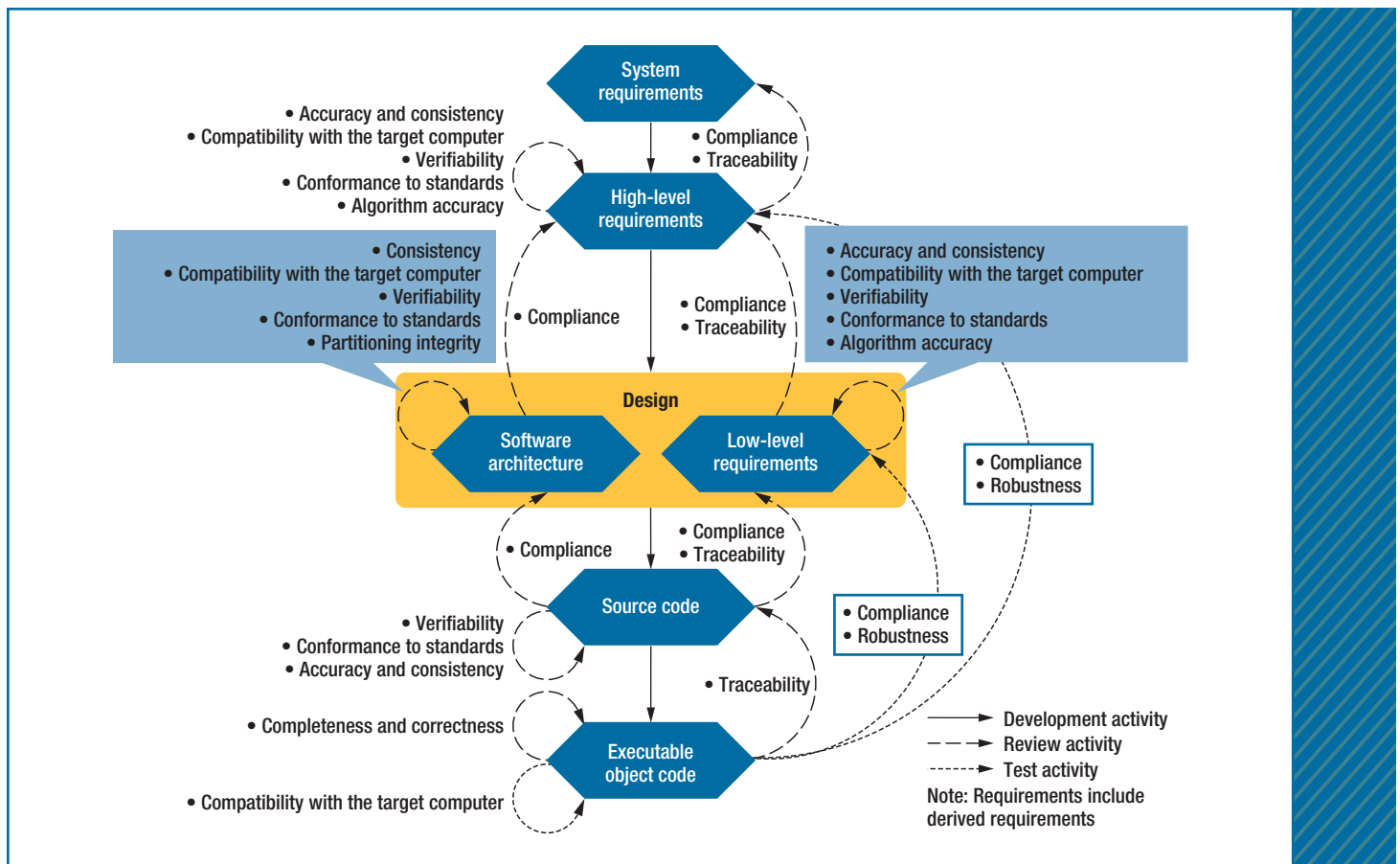
### Formal Verification at the Source-Code Level

DO-178 requires verification activities to show that a program in executable form satisfies its requirements (see Figure 1). For some requirements, verification, which can include formal analysis, can be conducted directly on the binary

representation. For example, Airbus uses formal analysis tools to compute the worst case execution time (WCET) and maximum stack usage of executables.<sup>5</sup> For many other requirements, such as dataflow and functional properties, formal verification is only feasible via the source-code representation. DO-178 allows this approach, provided the user can demonstrate that properties established at the source level still hold at the binary level. The natural way to fulfill this objective is to show that requirements at source-code level are traceable down to the object-code level.<sup>6,7</sup> Demonstrating traceability between source and object code is greatly

## WHAT ARE FORMAL METHODS?

According to RTCA DO-333, formal methods are mathematically based techniques for the specification, development, and verification of software aspects of digital systems. The first work on formal methods dates back to the 1960s, when engineers needed to prove the correctness of programs. The technology has evolved steadily since then, exploiting computing power that has increased exponentially. In DO-333, a formal method is defined as “a formal model combined with a formal analysis.” A model is formal if it has unambiguous, mathematically defined syntax and semantics. This allows automated and exhaustive verification of properties using formal analysis techniques, which DO-333 separates into three categories: deductive methods such as theorem proving, model checking, and abstract interpretation. Today, formal methods are used in a wide range of application domains including hardware, railway, and aeronautics.



**FIGURE 1.** Activities mandated by DO-178C to fulfill objectives (the labels on the arcs). Verification against requirements is shown in two white boxes with blue borders. (Note that the legend says “Test activity,” but DO-333 allows formal verification to replace these testing activities; artwork reproduced with permission of RTCA/EUROCAE.)

## WHAT ARE FUNCTION CONTRACTS?

The concept of program contracts was invented by the researcher C.A.R. Hoare in 1969 in the context of reasoning about programs. In the mid-1980s, another researcher, Bertrand Meyer, introduced the modern function contract in the Eiffel programming language. In its simplest formulation, a function contract consists of two Boolean expressions: a precondition to specify input constraints and a postcondition to specify output constraints. Function contracts have subsequently been included in many other languages, either as part of the language (such as CodeContracts for .NET or contracts for Ada 2012) or as an annotation language (such as JML for Java or ACSL for C). Contracts can be executed as runtime assertions, interpreted as logic formulas by analysis tools, or both.

facilitated by using qualified tools for purposes such as enforcing coding restrictions against features that would complicate traceability, by applying appropriate compiler options to preserve control flow, and by using code traceability analyses prepared by compiler vendors.

Assuring the correctness of the compiler's translation of source code into object code is, of course, important. Trust can be based on examination of the compiler itself (the tool qualification process) or the compiler's output. The former approach (qualifying the compiler) is rare because of the effort involved. The latter approach provides the relevant degree of assurance through the multiple and overlapping activities required by DO-178, including the hardware/software integration testing and the verification of untraceable object code.

The form of verification required by DO-178 is mostly based on requirements, both for verifying high-level requirements, such as "HLR1: the program is never in error state *E1*," and for verifying low-level requirements, such as "LLR1: function *F* computes outputs *O1*, ..., *On* from inputs *I1*, ..., *Im*." For both HLRs and LLRs, the DO-178 guidance requires in-range (compliance) and out-of-range (robustness)

verification, either by testing or by formal verification.

Compliance requirements focus on a program's intended nominal behaviors. To use formal verification for these requirements, you first express the requirement in a formal language—for example, HLR1 can be expressed as a temporal logic formula on traces of execution or as an observer program that checks the error state is never reached. Then, you can use symbolic execution techniques to check that the requirement is respected. The Java PathFinder tool used at NASA and the Aoraï plug-in of Frama-C implement this technique.<sup>8</sup> As another example, you can express LLR1 as a logic function contract (see the "What Are Function Contracts?" sidebar). Then, you use various formal analyses to check that the code implements these formal contracts, although deductive methods typically perform better here, as demonstrated by the operational deployment of tools such as Caveat/Frama-C<sup>5,8</sup> and SPARK.<sup>9</sup>

Robustness requirements focus on a program's behaviors outside its nominal use cases. A particularly important robustness requirement is that programs are free from runtime errors, such as reading uninitialized data, accessing out-of-bounds array elements, dereferencing null pointers, generating

numeric overflows, and so on, which might be manifest at runtime by an exception or by the program silently going wrong. Formal analyses can help check for the absence of runtime errors. Model checking and abstract interpretation are attractive options because they don't require the user to write contracts, but they usually suffer from state explosion problems (meaning the tool doesn't terminate) or they generate too many false alarms (meaning the tool warns about possible problems that aren't genuine). A successful example of such a tool is Astrée,<sup>5</sup> which was specifically crafted to address this requirement on a restricted domain-specific software. Deductive verification techniques require user-written function contracts instead of domain-specific tools and don't suffer from termination problems or too many false alarms. These techniques are available in Caveat,<sup>5</sup> Frama-C,<sup>8</sup> and SPARK.<sup>9</sup>

## Replacing Coverage with Alternative Objectives

To increase confidence in the comprehensiveness of testing-based verification activities, DO-178 requires coverage analysis. Test coverage analysis is a two-step process that involves requirements-based and structural coverage analyses. Requirements-based coverage establishes that verification evidence exists for all of the software's requirements—that is, that all the requirements have been met. This also applies to formal verification. Structural coverage analysis during testing (for example, statement coverage) aims to detect shortcomings in test cases, inadequacies in requirements, or extraneous code.

Structural coverage analysis doesn't apply to formal verification. Instead, DO-178C's supplement on formal methods, DO-333, defines four alternative activities to reach the structural coverage goals when using formal

verification:<sup>6,7</sup> *cover*, *complete*, *dataflow*, and *extraneous*. The four alternative activities aim to achieve the same three goals, substituting verification cases for test cases in the first one.

#### **Cover: Detect Missing Verification Evidence**

Unlike testing, formal verification can provide complete coverage with respect to a given requirement: it ensures that each requirement has been sufficiently—in other words, mathematically—verified. But unlike testing, formal verification results depend on assumptions, typically constraints on the running environment, such as the range of values from a sensor. Thus, all assumptions should be known, understood, and justified.

#### **Complete: Detect Missing or Incomplete Requirements**

Formal verification is complete with respect to any given requirement. However, additional activities are necessary to ensure that all requirements have been expressed—that is, all admissible behaviors of the software have been specified. This activity states that the completeness of the set of requirements should be demonstrated with respect to the intended function:

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

Checking that the cases don’t overlap and that they cover all input conditions is sufficient for demonstrating the first bullet point. Furthermore, it’s easy to detect obvious violations of the second point by checking syntactically that each case explicitly mentions each output. A manual review completes this verification. Note that formal methods can’t handle the more general problem of detecting all missing requirements.

#### **Dataflow: Detect Unintended Dataflow**

To show that the coding phase didn’t introduce undesired functionality, the absence of unintended dependencies between the source code’s inputs and outputs must be demonstrated. You can use formal analysis to achieve this

be executed and unintended functionalities—those that could be executed but aren’t triggered by the tests derived from requirements. When you use formal analysis, the previous activities give some degree of confidence that unintended functionalities can be detected.

Unit proof has replaced some of the testing activities at Airbus on the A400M military aircraft and the A380 and A350 commercial aircraft.

objective. Formal notations exist to specify dataflows, such as the SPARK dataflow contracts<sup>9</sup> or the Fan-C notation in Frama-C,<sup>8</sup> and associated tools automate the analysis.

#### **Extraneous: Detect Code That Doesn’t Correspond to a Requirement**

DO-178C requires demonstrating the absence of “extraneous code”: any code that can’t be traced to a requirement. This includes “dead code” as defined in DO-178C: code that’s present by error and unreachable. The relevant section of DO-333 explicitly states that detection of extraneous code should be achieved by “review or analysis (other than formal).” Although formal analysis might detect some such code, computability theory tells us that any practical formal analysis tool (which doesn’t generate so many false alarms that it’s useless in practice) will be unsound, meaning it will fail to detect some instances of extraneous code. DO-178C doesn’t allow unsound tools.

The effort required by this review or analysis depends chiefly on the degree of confidence obtained after completing the previous activities (cover, complete, and dataflow). Testing detects extraneous code as code that isn’t executed at runtime. This step detects both unreachable code that can never

It only remains to detect by review or analysis the unreachable code. Because this is a manual activity, its details vary from project to project.

#### **Formal Verification of Functional Properties: Airbus**

Since 2001, a group at Airbus has transferred formal verification technology—tools and associated methods—from research projects to operational teams who develop avionics software.<sup>5</sup> The technology for verifying nonfunctional properties such as stack consumption analysis, WCET assessment, absence of runtime errors, and floating-point accuracy isn’t seen as an alternative to testing and won’t be discussed here. Instead, we focus on unit proof,<sup>4,10</sup> which we developed for verifying functional properties. It has replaced some of the testing activities at Airbus for parts of critical embedded software on the A400M military aircraft and the A380 and A350 commercial aircraft.

Within the classical V-cycle development process of most safety-critical avionics programs, we use unit proof for achieving DO-178 objectives related to verifying that the executable code meets the functional LLRs. The term “unit proof” echoes the name of the classical technique it replaces: unit

testing. The use of unity proof diverged from the DO-178B standard (more accurately, it was treated as an alternative method of compliance), so we worked with the certification authorities to address and authorize this alternative. The new DO-178C standard—together with the formal methods supplement

the functional properties defined during the design phase. Finally, the engineer analyzes the proof results. The theorem-proving tool is integrated into the standard process management tool, so that this proof process is entirely automated and supported during maintenance.

The technique of unit proof reduces the overall effort compared to unit testing, in particular because it facilitates maintenance.

DO-333—fully supports the use of unit proof.

Unit proof is a process comprising three steps:

- An engineer expresses LLRs formally as dataflow constraints between a computation's inputs and outputs, and as preconditions and postconditions in first-order logic, during the development process's detailed design activity.
- An engineer writes a module to implement the desired functionality (this is the classical coding activity). The C language is used for this purpose.
- An engineer gives the C module's formal requirements and the module itself to a proof tool. This activity is performed for each C function of each C module.

Different steps are needed when using the theorem-proving tool. An engineer first defines the proof environment, and then the tool automatically generates the data and control flows from the C code. The engineer then verifies these flows against the data and control flows defined during the design phase. Next, the tool attempts to prove that the C code correctly implements

As discussed earlier, because we perform a verification activity at the source level instead of the binary level, we also analyze the compiler-generated object code, including the effects of the compiler options on the object code, to ensure that the compiler preserves in the object code the property proved on the source code. Within this development cycle, HLRs are expressed informally, so integration verification is done via testing, which includes verification of timing aspects and hardware-related properties. Even when taking into account these additional activities, the technique of unit proof reduces the overall effort compared to unit testing, in particular because it facilitates maintenance.

This approach satisfies the four alternative objectives to coverage:

- *Cover*. Each requirement is expressed as a property, each property is formally proved exhaustively, and every assumption made for formal verification is verified.
- *Complete*. Completeness of the set of requirements is verified by verifying that the dataflow gives evidence that the data used by the source code is conformant with decisions made during design. Based on this

guarantee, the theorem-proving tool verifies that the formal contract defined in the design phase specifies a behavior for all possible inputs. Then, we manually verify the formal contracts, to determine that an accurate property exists and specifies the value of each output for each execution condition.

- *Dataflow*. The dataflow verification gives evidence that the operands used by the source code are those defined at the design level.
- *Extraneous*. Except for unreachable code (which can't be executed), all the executable code is formally verified against LLRs. Thus, the completeness of the properties and the exhaustiveness of formal proof guarantee that any code section that can be executed will have no other impact on function results than what's specified in the LLRs. Identification of unreachable code, including dead code, is achieved through an independent, focused manual review of the source code.

There are two manually intensive, low-level testing activities in DO-178: normal range testing and robustness testing. While Airbus has been using formal verification to replace both types of testing (excluding runtime errors), Dassault-Aviation has experimented with formal verification to replace the robustness testing (including runtime errors).

## Formal Verification of Robustness: Dassault-Aviation

Since 2004, a group at Dassault-Aviation has used formal verification techniques experimentally to replace integration robustness testing,<sup>6</sup> where robustness is defined as “the extent to which software can continue to operate correctly despite abnormal inputs and conditions.”<sup>1</sup> We've applied these



techniques to flight control software developed following a model-based approach, specifically on the Falcon family of business jets equipped with digital flight control systems. C source code is automatically generated from a graphical model that includes a mix of dataflow and statechart diagrams. The average size of the software units verified by static analyzers is roughly 50 KLOC.

Normal conditions for this software are defined as intervals bounding the model's input variables and the permanent validity of a set of assertions stated at the model level. These assertions are assumptions expected to be met in both normal and abnormal input conditions for the model to operate properly—typically, they're range constraints on arguments to library functions at the model's leaf nodes. Apart from runtime errors, the robustness assertions amount to a few hundred properties stated at the model level and then propagated to the generated C code.

On such software, integration testing is functional, based on pilot-in-the-loop and hardware-in-the-loop activation of the flight control laws. Designing test cases to observe what might happen if some internal assertions break was determined to be costly and inconclusive, so we handle robustness by manually justifying that normal and abnormal external inputs can't lead to assertion failures. A set of design rules facilitate the checking of range properties; we apply them at the software-modeling level and use a custom checker to verify them. These rules made a manual justification possible.

We anticipated that strengthening the manual analysis of range constraints through mechanized interval propagation and abstract interpretation would be beneficial. But we couldn't compare the benefits of this process evolution on the baseline process by simply comparing past testing cost and

present formal verification cost: formal verification supplements an activity that was never performed through testing, just through human analysis.

To mechanize the analysis through formal proof of the assertions, we use two static analyzers that collaborate and share results on the Frama-C platform. Approximately 85 percent of these assertions are proved by abstract interpretation using Frama-C's value-analysis plug-in, and the remaining assertions are proved by deductive verification using Frama-C's WP plug-in and a set of automated theorem provers. The value-analysis plug-in takes into account IEEE 754-compliant numerical precision; while propagating intervals, it also verifies the absence of runtime errors, in particular, the absence of overflows and underflows.

As far as the verification process is concerned, once the integrated flight control software is sufficiently stable, a static analysis expert, in cooperation with a model expert, initially performs the formal robustness verification. The critical issue is to add a few extra assertions to be conclusive about the return values for the numerically intensive library functions. Finding them requires

the model requires revisiting the extra assertions, possibly with some support from the formal verification expert.

Design-rule verification and manual assertion analysis is estimated to take a person-month of effort by the independent control engineers (not software engineers) in charge of model verification. This effort must be repeated for every software model release, so there's no economic gain for a single release. However, because robustness verification is a recurrent task that's automated once the setup phase is complete, this rather long preparation provides a significant competitive advantage for repetitive analyses. The gain is roughly a person-month per flight software release.

This approach satisfies the following alternative objectives to coverage:

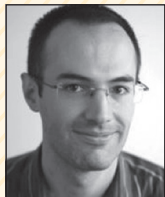
- *Cover.* An engineer handles abnormal input conditions through larger intervals and no other assumptions. The tool performs abstract interpretation with no assumptions other than those required to ensure hardware-dependent numerical consistency.
- *Complete.* A manual peer review of the set of assertions in the libraries

Because robustness verification  
is a recurrent task, the gain is roughly  
a person-month per flight software release.

both deep knowledge of the model and abstract interpretation expertise. It takes roughly a person-month effort to set up the Frama-C analysis script and to tune any manually added assertions. Then the model verifiers—an independent group from the model development team—can autonomously replay and update the analysis until some substantial algorithmic change in

and in the model ensures that robustness requirements are complete. This is facilitated by the simplicity of typical assertions, 90 percent of which are interval constraints.

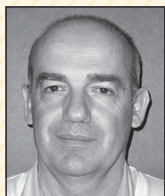
- *Dataflow.* An engineer formally specifies dataflows at the model level, using a dataflow formalism. Qualification of the code generator ensures no unintended dataflow



**YANNICK MOY** is a senior engineer at AdaCore, working on static analysis and formal verification tools for Ada and SPARK programs. He previously worked on similar tools for C/C++ programs at PolySpace, INRIA research labs, and Microsoft Research. Moy received a PhD in formal program verification from Université Paris-Sud. Contact him at moy@adacore.com.



**EMMANUEL LEDINOT** is a senior expert in formal methods applied to software and system engineering at Dassault-Aviation and was Dassault's representative in the ED-12/DO-178 formal methods group. Ledinot graduated as an engineer from Centrale Paris and has an MS in theoretical computer science from the University of Paris VII. Contact him at emmanuel.ledinot@dassault-aviation.com.



**HERVÉ DELSENY** is an expert in avionic software aspects of certification at Airbus and was a member of the working group in charge of writing issue C of ED-12/DO-178. His professional interests include formal methods and promoting their use in avionics software verification. Delseny has an MS in industrial software from Tours University, France. Contact him at herve.delseny@airbus.com.



**VIRGINIE WIELS** is a research scientist at Onera. She previously worked for NASA on formal verification of the Space Shuttle's embedded software. Wiels received a PhD in formal system development and verification from Ecole Nationale Supérieure d'Aéronautique et d'Espace. Contact her at virginie.wiels@onera.fr.



**BENJAMIN MONATE** is a founder and director at TrustMySoft. He's the former leader of the Software Reliability Laboratory at CEA LIST and a senior expert in formal verification and validation. His research interests include application of formal methods to static and dynamic analysis of programs as well as their certification and methodologies of deployment. Monate has a PhD from Université Paris-Sud Orsay. Contact him at benjamin.monate@cea.fr.


relationship at the source-code level compared to the design model.

Airbus and Dassault-Aviation were early adopters of formal verification as a means to replace manually-intensive

testing, at a time where the applicable standard DO-178B didn't fully recognize it. New projects can expect to get the same benefits in contexts where the new standard DO-178C explicitly supports it.

**F**ormal methods technology has matured considerably in recent years, and it's attracting increasing interest in the domain of high-integrity systems. Airborne software is an obvious candidate, but DO-178B treated the use of formal methods for verification as an activity that could supplement but not necessarily replace the prescribed testing-based approach. The revision of DO-178B has changed this, and the new DO-178C standard together with its DO-333 supplement offer specific guidance on how formal techniques can replace, and not simply augment, testing.

Experience at Airbus and Dassault-Aviation shows that the use of formal methods in a DO-178 context isn't simply possible but also practical and cost-effective, especially when backed by automated tools. During the requirements formulation process, engineers can use formal notation to express requirements, thus avoiding the ambiguities of natural language, and formal analysis techniques can then be used to check for consistency. This is especially useful because, in practice, the errors that show up in fielded systems tend to be with requirements rather than with code. However, the correct capture of system-functional safety at the software level can't be addressed by formal methods. During the coding phase, formal verification techniques can determine that the source code complies with its requirements.

An interesting possibility that we didn't discuss here is to combine testing with formal verification. This has seen some promising research in recent years,<sup>11</sup> and further industrial experience in this area will no doubt prove useful. 

### Acknowledgments

We thank the anonymous reviewers and Benjamin Brosgol for their helpful comments on this article, as well as Cyrille Comar for inspiring us to write it.

## References

1. RTCA DO-178, "Software Considerations in Airborne Systems and Equipment Certification," RTCA and EUROCAE, 2011.
2. NASA ARMD Research Opportunities in Aeronautics 2011 (ROA-2011), research program System-Wide Safety and Assurance Technologies Project (SSAT2), subtopic AFCS-1.3 Software Intensive Systems, p. 77; <http://nspires.nasaprs.com/external/viewrepositorydocument/cmdocumentid=320108/solicitationId=%7B2344F7C4-8CF5-D17B-DB86-018B0B184C63%7D/viewSolicitationDocument=1/ROA-2011%20Amendment%208%2002May12.pdf>.
3. J. Rushby, "New Challenges in Certification for Aircraft Software," *Proc. 9th ACM Int'l Conf. Embedded Software*, ACM, 2011; [www.csl.sri.com/users/rushby/papers/emsoft11.pdf](http://www.csl.sri.com/users/rushby/papers/emsoft11.pdf).
4. RTCA DO-333, *Formal Methods Supplement to DO-178C and DO-278A*, RTCA and EUROCAE, 2011.
5. J. Souyris et al., "Formal Verification of Avionics Software Products," *Proc. Formal Methods*, Springer, 2009; [http://link.springer.com/chapter/10.1007%2F978-3-642-05089-3\\_34?LI=true](http://link.springer.com/chapter/10.1007%2F978-3-642-05089-3_34?LI=true).
6. E. Ledinot and D. Pariente, "Formal Methods and Compliance to the DO-178C/ED-12C Standard in Aeronautics," *Static Analysis of Software*, J.-L. Boulanger, ed., John Wiley & Sons, 2012, pp. 207-272.
7. D. Brown et al., "Guidance for Using Formal Methods in a Certification Context," *Proc. Embedded Real-Time Systems and Software*, 2010; [www.open-do.org/wp-content/uploads/2013/03/ERTS2010\\_0038\\_final.pdf](http://www.open-do.org/wp-content/uploads/2013/03/ERTS2010_0038_final.pdf).
8. P. Cuoq et al., "Frama-C, A Software Analysis Perspective," *Proc. Int'l Conf. Software Eng. and Formal Methods*, Springer, 2012; [www.springer.com/computer/swe/book/978-3-642-33825-0](http://www.springer.com/computer/swe/book/978-3-642-33825-0).
9. J. Barnes, *SPARK, the Proven Approach to High Integrity Software*, Altran Praxis, 2012.
10. J. Souyris and D. Favre-Félix, "Proof of Properties in Avionics," *Building the Information Society*, IFIP Int'l Federation for Information Processing, René Jacquart, ed., vol. 156, 2004, pp. 527-535.
11. C. Comar, J. Kanig, and Y. Moy, "Integrating Formal Program Verification with Testing," *Proc. Embedded Real-Time Systems and Software*, 2012; [www.adacore.com/uploads\\_gems/Hi-Lite\\_ERTS-2012.pdf](http://www.adacore.com/uploads_gems/Hi-Lite_ERTS-2012.pdf).



See [www.computer.org/software-multimedia](http://www.computer.org/software-multimedia) for multimedia content related to this article.



## Richard E. Merwin Student Leadership Scholarship

IEEE Computer Society is offering \$40,000 in student scholarships, from \$1,000 and up, to recognize and reward active student volunteer leaders who show promise in their academic and professional efforts.

Graduate students and undergraduate students in their final two years, enrolled in a program in electrical or computer engineering, computer science, information technology, or a well-defined computer-related field, are eligible. IEEE Computer Society student membership is required.

Apply now! Application deadline is 30 April 2013. For more information, go to [www.computer.org/scholarships](http://www.computer.org/scholarships), or email [chuffman@computer.org](mailto:chuffman@computer.org).

To join IEEE Computer Society, visit [www.computer.org/membership](http://www.computer.org/membership).

IEEE  computer society

