This document has been publically released

2016-01-2044

Practical Uses of Formal Methods in Development of Airborne Software

Jeffrey Joyce[1], Scott Beecher[2], Laurent Fabre[1] and Ramesh Rajagopalan[2]

1 – Critical Systems Labs Inc., 2 – Pratt & Whitney

Abstract

Over the past few decades, advanced methods have been developed for the analysis of digital systems using mathematical reasoning, i.e., formal logic. These methods are supported by sophisticated software tools that can be used to perform analysis far beyond what is practically achievable using "paper and pencil" analysis. In December 2011, RTCA published RTCA DO-178C [1] along with a set of technical supplements including RTCA DO-333 [2] which provides guidance on the use of formal methods towards the certification of airborne software. Such methods have the potential to reduce the cost of verification by using formal analysis instead of conventional test-based methods to produce a portion of the verification evidence required for certification. Formal methods can also be used to find problems earlier in the development process – for example, while the requirements are being developed rather than during system integration when the cost of re-working the requirements and design is much higher. This paper provides an introduction to the practical use of formal methods in the development and certification of airborne software. The paper includes an illustrative example of using formal methods motivated by experience with the application of formal methods to engine control software.

Introduction

This paper provides an introduction to "formal methods" and their potential use in the development of airborne software. The paper begins with brief introduction to the use of formal methods in the development of software, followed by a summary of key points in RTCA DO-178C concerning the use of verification results obtained by means of formal methods for the purposes of certification. Next, the paper presented a simple example of using of formal methods. This is followed by the discussion of the example and finally, a summary of the paper and conclusions.

Formal Methods

The term "formal methods" in this context refers to a variety of methods based on discrete mathematics that can be used to specify and analyze digital systems including software.

The development of formal methods including software tools have been the subject of research since at least the 1960's. These methods typically make use of fundamental elements of discrete mathematics such as deductive logic, set theory and arithmetic. The use of formal methods in the development of software is motivated by the expectation that, as in other engineering disciplines, performing appropriate mathematical analyses can contribute to establishing the correctness and robustness of a design.

A formal method always involves the use of a notation or language that has both a formally defined syntax and formal semantics. While often textual, the notation or language of a formal method could also include graphical elements.

Formal methods can be used to unambiguously specify the functional behavior of software systems. They can also be used to model the design or implementation of a software system at various levels of abstraction. In some cases, the source code that implements the system may itself be a formal model that can analyzed directly using formal analysis.

There is a wide variety of different kinds of formal analysis that may be used for various purposes. For example, formal analysis may be used to show that a model of the design or implementation of a software system is correct with respect to a formal specification of its functional behavior. Other kinds of analyses may be used to verify that a functional specification satisfies certain properties such as consistency and completeness.

Some kinds of formal analysis can be performed automatically by a software tool. Other kinds of formal analyses generally require some degree of interaction between a skilled user and a software tool that automates certain aspects of the formal analysis.

When used properly, formal analysis can be used to produce verification evidence whose thoroughness is equivalent to exhaustive testing. Verification evidence produced by formal analysis can find defects and other problems that have a high likelihood of being missed when relying on conventional test-based verification.

Aside from the possibility of being used to produce verification results, formal analysis can also be a very effective way to understand complex details of a specification or model. Especially when the capability to represent behavior at a higher level of abstraction is used, formal methods can be very effective as a means of exploring design options before committing resources to the development of a detailed design. The use of formal analysis can also result in improvements to the specification of requirements, avoiding re-work later in development that could be very costly.

The use of formal methods has become well established in the development of digital hardware such as integrated circuits. For various reasons, the adoption of formal methods by industry in the

This is a pre-print version of SAE Technical Paper 2016-01-2044, 2016, doi:10.4271/2016-01-2044.

development of software systems has been slower. However, there is an increasing number of publically documented instances of industry using formal methods in the development of complex software systems. This includes, for example, the verification of some elements of the Airbus 380 software [3]. This gradual adoption of formal methods as a means of verifying software has motivated updates to guidance material such as RTCA DO-178C with respect to the use of verification evidence produced by means of formal methods for the purposes of certification.

RTCA DO-178C

RTCA DO-178B, issued in December 1992, includes a forward-looking reference to the possibility of using formal methods as an "alternate method". While recognizing this possibility, this guidance left the onus entirely on the applicant to convince the certification authority that the alternate method can be used for credit towards certification.

Providing better guidance for the use of verification results obtained by means of formal methods was among the tasks given to RTCA SC-205 when this special committee was established in 2005. This resulted in the 2011 publication of a technical supplement to RTCA DO-178C that provides specific guidance for how the results of formal analysis may be used towards certification credit. This technical supplement is published by RTCA as a separate document, RTCA DO-333. It is generally known as the "formal methods supplement" of RTCA DO-178C.

A key theme of RTCA DO-333 is that formal analysis can be used selectively to (partially) address particular objectives of RTCA DO-178C, in combination with conventional methods. For example, Section 6.3.2 of the formal methods supplement provides guidance on how formal analysis may be used to satisfy Objective 1 in Table A-4 of RTCA DO-178C, namely, that "low-level requirements comply with high-level requirements". This possibility is illustrated by the example presented in the next section of this paper.

Example

Figure 1 shows a model of a software design which implements a variant of "2 out of 3" voting logic as part of a fault tolerance strategy for a critical software system. Such a model could be used to specify this functionality in a graphical modeling language such as Simulink. The software generated from this model periodically updates the value of the output signal, *out*, as a function of current and previous inputs to the voting logic.

The three inputs to this voting logic are critical signals that, for example, report the state of a switch in the cockpit of an aircraft. In the absence of faults, the value of *out* will be identical to the common value of the three inputs. When a fault causes one of the input signals to disagree with the other two input signals, the voting logic will select the majority value – with one exception. If the values of *a* and *b* have agreed for the previous *N* cycles and the current value of input *a* is 1 (true), then the voting logic will select the value of *a* as the output even if the value of *a* is the minority value. This exception is intended to mask the effect of a single cycle fault on input *b* that would otherwise cause a falling edge on the output of the voting logic.

The rectangle in the diagram below represents a latch that becomes set when a mismatch between the values of *a* and *b* is detected. The latch will remain set until the two signals have agreed continuously for at least *N* update cycles. The value of *N* is known as the fault recovery time. Setting *N* to a value greater than 0 is intended to partially mitigate the possibility of an intermittent fault affecting *b*.

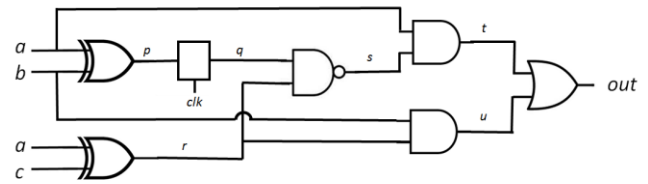

Figure 1. Model for "2 out of 3" Voting Logic.

To appreciate the potential benefit of this design, it may be assumed that it is used in a system context where a single cycle delay in the transition of the output from 1 (true) to 0 (false) is more tolerable than allowing a single cycle fault on signal b to cause the output to transition from 1 (true) to 0 (false). For example, a transition of the output from 1 (true) to 0 (false) might trigger an action that degrades performance. If the new value of *b* persists beyond a single cycle, the result of delaying this action by a single cycle may be of little or no consequence. However, triggering this action unnecessarily in response to a single cycle "glitch" on input *b* might have undesirable consequences with respect to performance.

As part of a disciplined approach to software development using model-based development, the above model should be reviewed while the model is being developed. Ideally, this review is performed before testing the software generated from the model and integrated into the rest of the system. This review might simply be a "paper and pencil" review that relies on the skill of the reviewers to spot potential flaws. The reviewers might also have access to model animation tools, i.e., a software tool that simulates the behavior of the software by determining the output of a circuit for a particular sequence of inputs.

The behavior of the individual logic gates should be clear to any qualified reviewer. However, the behavior of the clocked latch is a potential source of ambiguity. One possibility is that the output of the clocked latch immediately shows when a mismatch between *a* and *b* is detected, i.e., a rising edge of internal signal *p* will propagate to *q* before the latch is clocked. Another possibility is that the output of the clocked latch is delayed by one update cycle, i.e., a rising edge of internal signal *p* does not result in a rising edge of *q* until the latch is clocked.

The difference between these two behaviors is illustrated by the timing diagram shown in Figure 2. In this diagram, q_1 depicts the output of the latch upon a rising edge of *p* for the first of the two possible behaviors described above. This contrasts with the behavior of q_2 for which the effect of the rising edge of *p* is delayed until the next update.

This is a pre-print version of SAE Technical Paper 2016-01-2044, 2016, doi:10.4271/2016-01-2044.

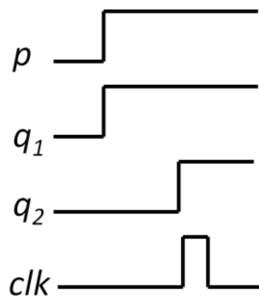

Figure 2. Two different possible behaviors of the clocked latch.

It is easy to imagine that a reviewer might base his or her review of the voting logic model on one of these two possibilities without confirming that their "mental model" of the latch behavior is in fact accurate. Choosing the wrong possibility in this regard is an instance of a general class of errors made by software developers known as "off by one errors". The clocked latch might be a commonly used element of a library of "parts" shared by multiple projects. It is possible that the behavior of this part is not clearly documented, or that its behavior has been recently modified to accommodate the needs of another project without updating the documentation to reflect this modification. In some designs that incorporate this clocked latch, the difference between these two behaviors might be unimportant with respect to overall functionality of the software. For this reason, the reviewer might not see any value in taking the trouble to clarify which of the two possibilities most accurately represents the behavior of the latch. Alternatively, the reviewer might conclude that they are using the right mental model of the latch for their review if the paper and pencil simulation and/or the model animation yield the expected results for a variety of test cases.

In fact, the difference between the two possible behaviors of the clocked latch is crucial for the correct implementation of the voting logic software. If the rising edge of internal signal *p* propagates to *q* immediately, the model of the voting logic will yield a software function that always selects the majority value of its three inputs. On the other hand, if the output of the clocked latch is delayed by one update cycle, the software generated from this model will tolerate certain intermittent faults.

The effect of the difference between the two different possible latch behaviors on the overall function of the design is only revealed by one of 16 possible test cases needed to exhaustively cover all possible combinations of values of the three input signals *a*, *b* and *c*, and the one internal state variable, *q*. In particular, the behaviors differ only in the case where the input signals *a*, *b* and *c* are set to 1 (true), 0 (false) and 0 (false) respectively when the internal state variable, *q*, is 0 (false), i.e., not in the recovering state.

This difference is depicted in Figure 3 where out_1 is the result for the case in which the rising edge of internal signal *p* propagates to *q* immediately and out_2 is the result for the case in which output of the clocked latch is delayed by one update cycle. As represented by out_1, this fault propagates to the output even for a single cycle fault. This contrasts with the result of using a latch in which the output of the clocked latch is delayed by one update cycle. In this second case, the single cycle fault does not propagate to the output, i.e., out_2 remains stable at 1 (true). Hence, use of a latch in which the output of the clocked latch is delayed by one update cycle is essential to tolerating a fault on input signal *b* if it clears after one cycle.

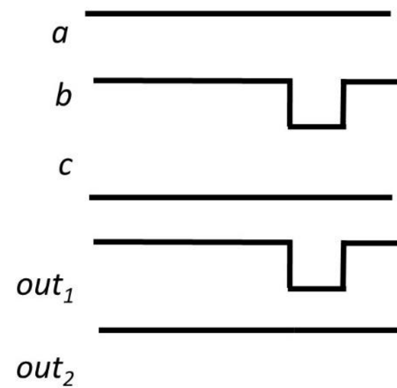

Figure 3. Effect of a 1-cycle fault on input signal *b*.

Formal Analysis

Formal analysis is an alternative to "paper and pencil" review or test-based verification of the model. When properly used, it has the potential to automatically discover problems with the model.

To demonstrate how formal analysis may be used, the voting logic model has been translated into a textual description using the notation of a well-established formal analysis tool called NuSMV [4]. This textual description is shown below in Figure 4.

```
MODULE main
  VAR
    a : boolean;
    b : boolean;
    c : boolean;
    q : boolean;
    n : 0..4;
    l : LATCH(p,n,q);
  DEFINE
    p := a xor b;
    r := a xor c;
    s := !(q & r);
    t := a & s;
    u := b & r;
    out := t | u;
    exception := a & !b & !c & (n = 0);
  SPEC
    AG(
      (exception -> (out = TRUE)) &
      (!exception -> (out = ((a & b) | (a & c) | (b & c))))
    )

MODULE LATCH (p,n,q)
  DEFINE
    N := 4;
  ASSIGN
    init(n) := N;
    q := p | (n > 0);
    next(n) := case p : N; (n < 1) : 0; TRUE : (n - 1); esac;
```

Figure 4. NuSMV code for "2 out of 3" voting logic model.

The above NuSMV code includes a model of the clocked latch in which the rising edge of internal signal *p* propagates to *q* immediately.

The recovery period has been set to four cycles, i.e., $N = 4$.

In addition to describing the structure of the model, the above NuSMV code includes a formal specification of the intended behavior of the voting logic. This specification appears in the SPEC part of the above NuSMV code. In particular, the specification asserts that the following two properties hold:

This is a pre-print version of SAE Technical Paper 2016-01-2044, 2016, doi:10.4271/2016-01-2044.

1. The value of *out* is 1 (true) for the exceptional case when the current values of *a*, *b* and *c* are 1 (true), 0 (false) and 0 (false) and the values of *a* and *b* have agreed for the previous four cycles.

```
exception -> (out = TRUE)
```

2. Otherwise, the value of *out* is the simple "2 out of 3" majority of the current values of the three input signals.

```
!exception -> (out = ((a & b) | (a & c) | (b & c)))
```

The condition that determines when the output should be set to 1 (true) instead of using the simple "2 out of 3" majority is defined earlier in the NuSMV code as follows:

```
exception := a & !b & !c & (n = 0);
```

When the NuSMV tool is applied to the above NuSMV code, this formal analysis tool automatically discovers the problem with this design that occurs when rising edge of internal signal *p* propagates to *q* immediately. In addition to reporting that the design fails to fully satisfy its specification, the tool generates a counter-example which is described by the following output of NuSMV.

```
Trace Description: CTL Counterexample
Trace Type: Counterexample
  -> State: 1.1 <-
    a = FALSE
    b = FALSE
    c = FALSE
    q = TRUE
    n = 4
    exception = FALSE
    out = FALSE
    u = FALSE
    t = FALSE
    s = TRUE
    r = FALSE
    p = FALSE
    l.N = 4
  -> State: 1.2 <-
    a = TRUE
    b = TRUE
    n = 3
    out = TRUE
    u = TRUE
    s = FALSE
    r = TRUE
  -> State: 1.3 <-
    n = 2
  -> State: 1.4 <-
    n = 1
  -> State: 1.5 <-
    b = FALSE
    n = 0
    exception = TRUE
    out = FALSE
    u = FALSE
    p = TRUE
```

The above counter-example is a trace of the values of signals in the design over an interval of five update cycles. Each state in the above trace corresponds to a single update cycle. State 1.1 of the trace shows the initial value of each signal in the trace. In State 1.2, the values of two input signals, *a* and *b*, spontaneously change from 0 (false) to 1 (true). This change affects other signals including the output, *out*, which transitions from 0 (false) to 1 (true). Three update cycles later in State 1.5, the value of *b* spontaneously transitions from 1 (true) to 0 (false). This causes the output to also transition to 0 (false), as the majority value of the three inputs is 0 (false). However, this result contradicts the specification which asserts that the output value should remain 1 (true) when the values of *a* and *b* have agreed for the previous four cycles and the current values of a, b and are 1 (true), 0 (false) and 0 (false). That is, the above counter-example contradicts the following assertion:

```
exception -> (out = TRUE)
```

The above NuSMV code can be modified to correct the problem made evident by the automatically generated counter-example. In particular, the model can be modified so that the output of the clocked latch is delayed by one update cycle. The necessary modification consists of adding a new line of NuSMV code that initializes the value of *q* to TRUE and modifying the latch specification to delay the propagation of its input to its output until the next update cycle. These two changes are highlighted by underlined bold text in the revised NuSMV code shown below.

```
MODULE main
  VAR
    a : boolean;
    b : boolean;
    c : boolean;
    q : boolean;
    n : 0..4;
    l : LATCH(p,n,q);
  DEFINE
    p := a xor b;
    r := a xor c;
    s := !(q & r);
    t := a & s;
    u := b & r;
    out := t | u;
    exception := a & !b & !c & (n = 0);
  ASSIGN
    init(q) = TRUE;
  SPEC
    AG(
      (exception -> (out = TRUE)) &
      (!exception -> (out = ((a & b) | (a & c) | (b & c))))
    )

MODULE LATCH (p,n,q)
  DEFINE
    N := 4;
  ASSIGN
    init(n) := N;
    next(q) := p | (n > 1);
    next(n) := case p : N; (n < 1) : 0; TRUE : (n - 1); esac;
```

Applying NuSMV to the modified NuSMV code yields a result which confirms that the model of the voting logic fully satisfies the formal specification of its intended behavior, as given in the SPEC part of the NuSMV code. This proves that the correct choice of latch for this design is the latch that delays the propagation of the output by one update cycle.

Discussion

The simple example presented in this paper is intended to be an introduction to the application of formal methods in the development of airborne software. The current state of the art is capable of modeling and analyzing substantially more complex systems. For example, Miller et al. [5] refer to several example uses of formal methods including formal analysis of a Rockwell Collins product, ADGS-2100, that provides heads-down and heads-up displays, and display management software for next-generation commercial aircraft. The five main components of this system, which were each analyzed independently, reportedly contain a total of 16,117 primitive Simulink blocks.

The example given in this paper is intentionally simple to ease its presentation with enough detail that it could be easily reproduced by readers without further elaboration. Part of its simplicity is due to the fact that the model for the voting logic is presented in isolation from other elements of a model for a complex system. If the voting logic model had instead been presented as an element of a more substantial model, the use of conventional methods to analyze this model would have been both more challenging and more representative of a situation in which formal analysis offers substantial advantages over conventional methods.

This is a pre-print version of SAE Technical Paper 2016-01-2044, 2016, doi:10.4271/2016-01-2044.

The most obvious benefit of using formal methods in the example presented in this paper is the discovery of the problem that results from the use of a latch for which the rising edge of internal signal p propagates to q immediately. However, this is not the only benefit of using formal analysis.

The use of formal methods to create this example also improved the specification of its functional behavior. In a preliminary draft of this paper, the specification (in English) of the functional behavior of the voting logic was incomplete. The process of applying formal analysis to the model revealed gaps in the functional specification. This feedback was used to improve the specification of the functional behavior of the voting logic. In addition to finding problems such as omissions and inconsistencies in the requirements, the formal representation of a set of requirements can be a very effective filter that screens out design details that should not be present at the requirements level. In this regard, the use of formal methods can be used to discourage a "bad habit" of burdening requirements with design detail. Even if the results of formal analysis were not used towards certification credit, improvements to the formal specification of the requirements might be well worth the effort of using formal analysis – especially, this exercise avoids problems that would otherwise have caused re-work later in development.

An interesting question is whether formal analysis should be applied directly to the model used to generate the software, or alternatively, to a more abstract representation of this model.

Some tool suites for model based development of software include built-in support for formal analysis – for example, the MATLAB Simulink Design Verifier allows formal analysis to be applied directly to a Simulink model.

There are some obvious benefits of applying formal analysis directly to the model from which the software will be generated. However, there might be situations in which it is advantageous to create a separate representation of the model for the purposes of formal analysis. For example, it may be beneficial to use formal analysis at an early stage in the development of the model before much of the level low detail has been fleshed out.

As well, a separate model might be created for the purposes of formal analysis as a means of screening out details that are judged to be not relevant to the specific properties of interest. For example, the model created for the purpose of generating the software might include complicated timing constraints that are not relevant to a property concerned with the function implemented by a block of combinational logic. The formal analysis of this combination logic may be unnecessarily complicated by irrelevant details of the timing constraints. Such details might also limit the extent to which the formal analysis will yield useful results in a reasonable amount of time. Effective use of formal analysis often depends on the analyst using strategies to filter out irrelevant details. Creating a separate model for the purpose of formal analysis allows this freedom to use such strategies.

When a separate model is created for the purposes of formal analysis, there is always a possibility that the separate model yields results that are not true for the model that will be used to generate the software. This possibility is recognized by RTCA DO-333 with guidance that requires the use of verification results obtained by means of formal analysis for the purposes of certification to be supported by an informal argument that the formal representation of the model is a "conservative representation".

Summary/Conclusions

The selective application of formal methods in the development of airborne software can find problems that may otherwise be missed by a methodology that relies on review and conventional test-based methods.

The most commonly cited benefit of using formal methods is the potential to increase the thoroughness of verification. It is important to appreciate that the use of formal methods offers additional benefits beyond increased thoroughness of the results. As demonstrated by the simple example in this paper, the use of formal methods can yield improvements to the specification of requirements and re-usable library elements. Other studies also reported cost benefits [3,5], particularly when the cost of using formal methods for the purposes of verification is compared to the cost of test-based methods over the lifecycle of a system.

References

1. RTCA DO-178, "Software Considerations in Airborne Systems and Equipment Certification," RTCA and EUROCAE, 2011.
2. RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A, RTCA and EUROCAE, 2011.
3. Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate, "Testing or Formal Verification: DO-178C Alternatives and Industrial Experience," IEEE Software, vol. 30, no. 3, pp. 50-57, May-June, 2013.
4. NuSMV homepage, http://nusmv.fbk.eu/.
5. S. Miller, M. Whalen, and D. Cofer. "Software model checking takes off", *Commun. ACM*, vol. 53, no. 2, pp. 58-64, February 2010.

Contact Information

Jeffrey Joyce jeff.joyce@cslabs.com, http://www.cslabs.com.

This is a pre-print version of SAE Technical Paper 2016-01-2044, 2016, doi:10.4271/2016-01-2044.