# DO-333 Certification Case Studies

Darren Cofer and Steven Miller

Rockwell Collins Advanced Technology Center
{ddcofer, spmiller}@rockwellcollins.com

**Abstract.** RTCA DO-333, *Formal Methods Supplement to DO-178C and DO-278A*, provides guidance for software developers wishing to use formal methods in the certification of airborne systems and air traffic management systems. This paper presents three case studies describing the use of different classes of formal methods to satisfy DO-178C certification objectives. The case studies examine different aspects of a common avionics example, a dual-channel Flight Guidance System (FGS), which is representative of the issues encountered in actual developments. The three case studies illustrate the use of theorem proving, model checking, and abstract interpretation. Each of these techniques has strengths and weaknesses and each could be applied to different life cycle data items and different objectives than those described here. Our purpose is to illustrate a reasonable application of each of these techniques to produce the evidence needed to satisfy certification objectives in a realistic avionics application. We hope that these case studies will be useful to industry and government personnel in understanding formal methods and the benefits they can provide.

**Keywords:** Formal methods, certification, model checking, theorem proving, abstract interpretation

## 1 Introduction

Certification can be defined as legal recognition by a government authority that a product, service, organization, or person complies with specified requirements. In the context of commercial aircraft, certification consists primarily of convincing the relevant certification authority (the FAA in the U.S. or EASA in Europe) that all required steps have been taken to ensure the safety, reliability, and integrity of the aircraft. Software itself is not certified in isolation, but only as part of an aircraft design. Certification differs from verification in that it focuses on evidence provided to a third party to demonstrate that the required activities were performed completely and correctly, rather on performance of the activities themselves.

For software in commercial aircraft, the relevant certification guidance is found in DO-178C, "Software Considerations in Airborne Systems and Equipment Certification" (known in Europe as ED-12C) [10]. Certification authorities in North American and Europe have agreed that an applicant (aircraft manufacturer) can use this guidance as a means of compliance with the regulations governing aircraft certification.

Its predecessor, DO-178B, allowed for the use of formal methods to satisfy certification objectives, but did so only as an “Alternative Method.” DO-178C now provides guidance specific to newer software technologies including formal methods, model-based development, and object-oriented software. This technology-specific guidance is contained in supplemental documents which may add, modify, or replace objectives in the core document. With the publication of DO-333, *Formal Methods Supplement to DO-178C and DO-278A* [12], the use of formal methods has become a recognized means of compliance (rather than an alternative method), streamlining the process for aircraft manufacturers to obtain certification credit through the use of formal verification techniques.

This paper presents three case studies describing the use of different classes of formal methods to satisfy DO-178C certification objectives using the guidance in DO-333. The three case studies illustrate the use of theorem proving, model checking, and abstract interpretation. Each of these techniques has strengths and weaknesses, and each could be applied to different life cycle data items and different objectives than those described here. The material presented is not intended to represent a complete certification effort. Rather, the purpose is to show how formal methods can be used in a realistic avionics software development, focusing on the evidence produced that could be used to satisfy the verification objectives found in DO-178C. The complete version of the case studies along with all the associated models, code, and verification artifacts will be available as a NASA contractor report in 2014.

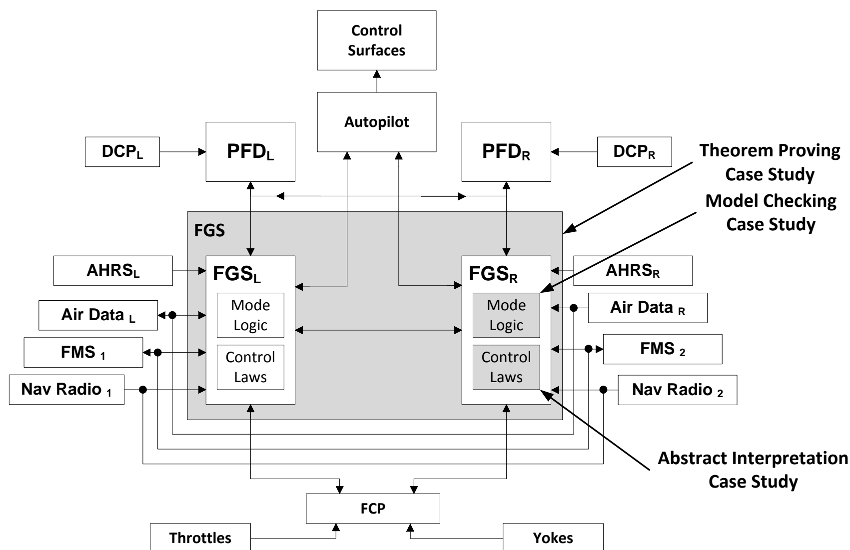


**Fig. 1**. Formal Methods applications in the Flight Guidance System example.

The case studies examine different aspects of a common avionics system, a dual-channel Flight Guidance System (FGS), shown in Fig. 1. While not intended as a complete example, it is representative of the issues encountered in actual development

projects and includes design artifacts specified using PVS, MATLAB Simulink/Stateflow®, and C source code.

An FGS is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The pilots interact with the FGS via the Flight Control Panel (FCP), Primary Flight Display (PFD), and the Display Control Panel (DCP).

The FGS subsystem accepts input about the aircraft's state from the Attitude Heading Reference System (AHRS), the Air Data System (ADS), the Flight Management System (FMS), and the Navigation Radios. Using this information, it computes pitch and roll guidance commands that are provided to the autopilot (AP). When engaged, the AP translates these commands into movement of the aircraft's control surfaces necessary to achieve the commanded changes about the lateral and vertical axes.

The FGS has two physical sides, or channels – one on the left side and one on the right side of the aircraft. These provide redundant implementations that communicate with each other over a cross-channel bus. Each channel of the FGS can be further broken down into the mode logic and the flight control laws. The flight control laws accept information about the aircraft's current and desired state, and compute the pitch and roll guidance commands. A flight control law is active if its guidance commands are being used to control the aircraft or to provide visual cues to the flight crew. A flight control law that is operational but that is not yet active is armed. The mode logic determines which lateral and vertical modes of operation are active (e.g. controlling the aircraft or providing visual guidance cues to the flight crew) and armed (e.g. operational but not yet active) at any given time. These in turn determine which flight control laws are active and armed.

## 2 Certification and DO-333

General guidance is provided in DO-333 that is applicable to the overall verification process when formal methods are used. This includes the following requirements:

- All formal notations used must have unambiguous, mathematically defined syntax and semantics.
- The soundness of each formal analysis method should be documented. A sound method never asserts that a property is true when it may not be true. Soundness here refers to the underlying analysis method, not soundness of the tool implementation. Tool soundness issues are addressed separately as part of the tool qualification process described in DO-330 [11].
- All assumptions related to the formal analysis should be described and justified (e.g. assumptions about execution semantics on the target computer, or assumptions about data range limits).

Beyond these general requirements, specific guidance is provided to describe how formal methods can be applied within each of the verification activities and objectives defined in DO-178C. This is illustrated in Fig. 2 for Level A software, the highest criticality level defined in DO-178C. These include compliance with requirements,

accuracy and consistency of requirements, compatibility with the target computer, verifiability of requirements, conformance to standards, traceability between life cycle data items, and algorithmic correctness.

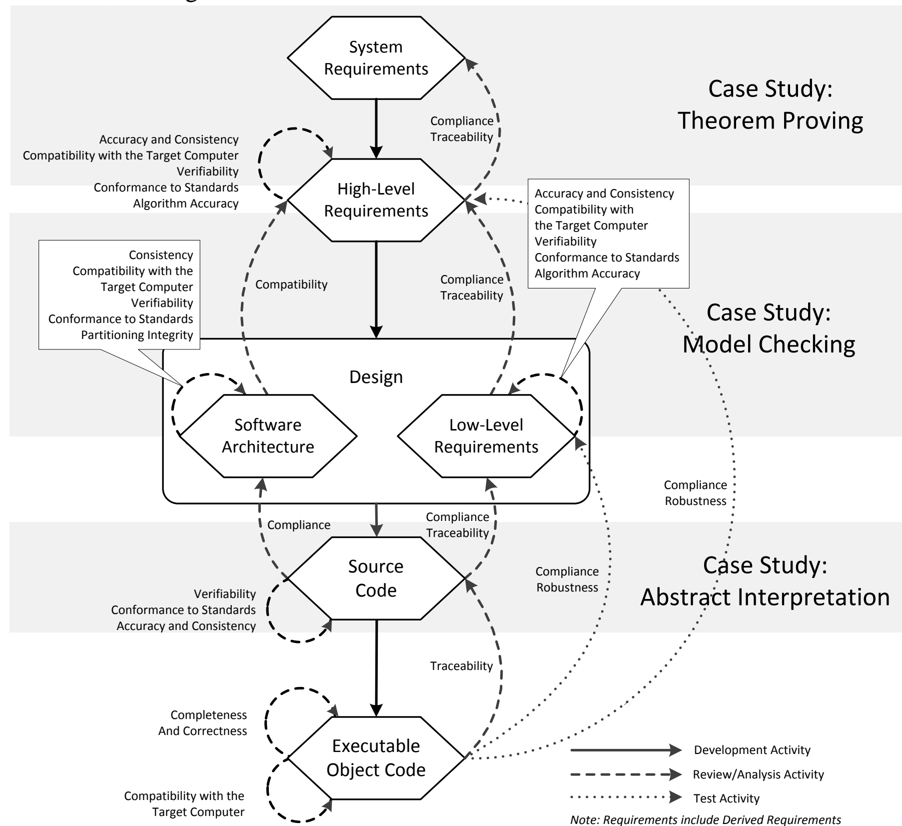


**Fig. 2**. Relationship of Case Studies to DO-178C Level A Objectives (adapted from DO-333).

Fig. 2 also shows the relationship between the three case studies and DO-178C objectives. Theorem proving was applied to the verification of the High-Level Requirements (HLR) for the Pilot Flying synchronization logic of the two channels of the FGS, focusing on the objectives of DO-333 Table FM.A-3. Theorem proving is generally considered the most powerful and versatile class of formal methods, but it is also the least automated, and usually requires the significant expertise and user training. This case study is described in Section 3.

Model checking has been applied to the verification of the Low-Level Requirements (LLR) for the mode logic of a single FGS channel, focusing on the objectives of DO-333 Table FM.A-4. Current model checking tools are very powerful and provide much more automation than theorem provers. In general, less user expertise is required, but the user must be able to specify requirements to be analyzed in a formal language. These tools are relatively mature and (in our opinion)

the benefits of using formal methods are greatest at this level. This case study is described in Section 4.

Abstract interpretation has been applied to the Source Code implementing one of the control laws of the FGS, focusing on the objectives of DO-333 Table FM.A-5. Abstract interpretation is the most automated of the three techniques, at least as used in currently available commercial tools, and typically requires the least expertise from users. Part of this is due to the use of abstract interpretation to check non-functional requirements, eliminating the need to formally specify requirements. We should note, however, that more powerful versions of abstract interpretation tools exist which require much more expertise to specify and check user-defined abstract domains. This case study is described in Section 5.

Another issue we address in each case study is *tool qualification*. Tool qualification is the process necessary to obtain certification credit for the use of a software tool within the context of a specific airborne system. The purpose of qualification is to ensure that the tool provides confidence at least equivalent to that of any process which is eliminated, reduced, or automated. DO-178C specifies that tool qualification should be performed in accordance with *DO-330, Software Tool Qualification Considerations* [11].

Each case study includes:

- The objectives to be satisfied and the evidence produced
- A general description of the portion of the example system to be verified
- A description of the verification approach, including the life cycle data items produced and the tools used, corresponding to some of the information that should be included in a Software Verification Plan
- Tool qualification issues for the formal methods tools used
- A detailed description of the verification effort that was performed

There are some parts of DO-333 that are not covered in these case studies. In particular, we do not address the verification of Executable Object Code (DO-333 Table FM.A-6), nor do we address the replacement of coverage testing by formal analysis (DO-333 Table FM.A-7).

## 3 Theorem Proving Case Study

This case study illustrates the use of the PVS [9] and the HOL4 [7] theorem proving systems to verify the outputs of the software requirements process (DO-178C Section 5.1) focusing on the objectives of Table A-3 in DO-178C and Table FM.A-3 in DO-333. The purpose of these verification activities is to detect any errors that may have been introduced during the software requirements process. The DO-178C and DO-333 objectives satisfied through theorem proving are summarized in Table 1. The table indicates whether an objective was satisfied (fully or partially) in the case study for each software level, A through D. Some objectives do not need to be satisfied for the less critical Level C or Level D software and are indicated by shaded boxes in the corresponding columns of the table.

**Table 1.** Summary of Objectives Satisfied by Theorem Proving

| Obj | Description | A | B | C | D | Notes |
|---|---|---|---|---|---|---|
| A-3.1 | High-level requirements comply with system requirements. | ■ | ■ | ■ | ■ | Established by proof the system requirements are implemented by the high-level requirements and the system architecture. |
| A-3.2 | High-level requirements are accurate and consistent. | ■ | ■ | ■ | ■ | Accuracy is established by formalization of the high-level requirements. Consistency is established by proving the absence of logical conflicts. |
| A-3.3 | High-level requirements are compatible with target computer. | | | | | Not addressed |
| A-3.4 | High-level requirements are verifiable. | ■ | ■ | ■ | | Established by formalizing the requirements and completion of the proof. |
| A-3.5 | High-level requirements conform to standards. | □ | □ | □ | | Partially established by specifying the high-level requirements as formal properties. |
| A-3.6 | High-level requirements are traceable to system requirements. | ■ | ■ | ■ | ■ | Established by verification of the system requirements, and by demonstrating the necessity of each high-level requirement for satisfying some system requirement. |
| A-3.7 | Algorithms are accurate. | ■ | ■ | ■ | | Correctness of the pilot flying selection logic is established by proof. |
| FM.A-3.8 | Formal analysis cases and procedures are correct. | ■ | ■ | ■ | | Established by review. |
| FM.A-3.9 | Formal analysis results are correct and discrepancies explained. | ■ | ■ | ■ | | Established by review. |
| FM.A-3.10 | Requirements formalization is correct. | ■ | ■ | ■ | | Established by review. |
| FM.A-3.11 | Formal method is correctly defined, justified, and appropriate. | ■ | ■ | ■ | ■ | Established by review. |

■ *Full credit claimed* □ *Partial credit claimed* *Satisfaction of objective is at applicant's discretion*

Consider Objective A-3.1 in Table 1 (high-level requirements comply with system requirements). The system architecture is captured in the PVS theory *Pilot_Flying_System*. This theory describes how the system components interact in the overall system. The system requirements are stated formally as theorems in the PVS theory *Pilot_Flying_System_Requirements*. Machine checked proofs are developed in PVS to prove that these requirements are satisfied by the system architecture and the high-level requirements for the system components. The high-level software requirements are specified for each FGS side in the *Side_HLR* theory. This theory uses axioms and uninterpreted types, constants, and functions to eliminate design detail from the requirements. The axioms are proven consistent by demonstrating that at least one concrete implementation exists that satisfies the axioms. The objective is satisfied by proving with the PVS theorem prover that the system level requirements specified as theorems in theory *Pilot_Flying_-System_Requirements* are implemented by the system architecture defined in theory *Pilot_Flying_System*, the high-level software requirements specified as axioms in theory *Side_HLR* and the high-level hardware requirements specified as axioms in theory *Bus_HLR*. A more detailed discussion of how each objective is satisfied is provided in the full contractor report available from NASA.

The specific example used in the theorem proving case study is the synchronization of the Pilot Flying side of the aircraft. The overall FGS system has two physical sides, or channels, one on the left side and one on the right side of the aircraft. These provide redundant implementations that communicate with each other over a cross-

channel bus as shown in Fig. 3. Bidirectional communication between the left and right sides is modeled separately as LR_Bus and RL_Bus.

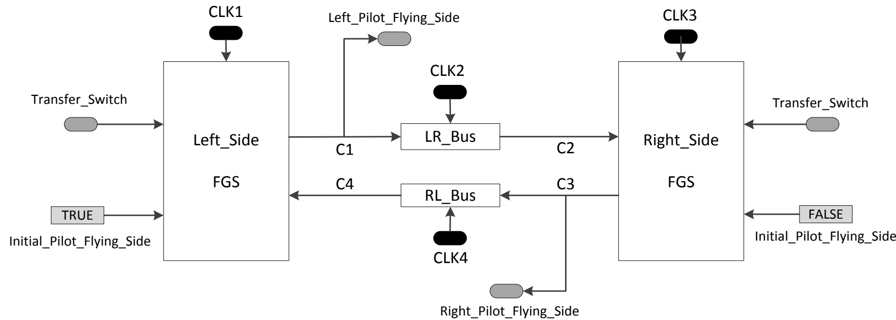


**Fig. 3.** Overview of the Dual FGS System

Most of the time, the FGS operates in *dependent* mode where only one FGS channel is *active* and provides guidance to the AP. In this mode, the flight crew can choose whether the left or the right FGS is the active, or *pilot flying*, side by pressing the Transfer Switch. The other side serves as a hot spare and sets its modes to agree with those of the active side. In this example, there are five system-level requirements related to the synchronization of the pilot flying side. Stated informally, these are:

- R1. At least one side shall be the pilot flying side.
- R2. At most one side shall be the pilot flying side.
- R3. Pressing the Transfer Switch shall always change the pilot flying side.
- R4. The system shall start with the Primary Side as the pilot flying side.
- R5. The system shall not change the pilot flying side unless the Transfer Switch is pressed.

The case study formalizes these system-level requirements in PVS and HOL4, develops high-level software and hardware requirements for each side and the cross-channel bus, and proves that that the system architecture, the high-level software requirements, and the high-level hardware requirements comply with the system requirements. This is done in both PVS and HOL4 for a synchronous design in which all components are driven from single master clock. The example was repeated in PVS for an asynchronous design in which the components are driven by separate clocks.

For example, the PVS specification of the requirements R1 and R2 for the synchronous design is shown in Fig. 4. Note that formalizing these requirements required a precise statement of what it means for the system to be switching sides.

The case study then develops a set of high-level requirements for each subcomponent, i.e., the FGS sides and the buses of Fig. 3, that are completely free of design detail by using uninterpreted PVS types and axioms specifying the relationship of their outputs to their inputs. These high-level requirements are then proven to be consistent (i.e. to not contradict each other) by creating a concrete implementation using interpreted PVS types and functions and showing that the concrete implementation is a PVS theory interpretation of the high-level component

requirements. Finally, we prove that the system architecture and the high-level requirements of the components comply with the system requirements by proving that the system requirements are satisfied by the synchronous design instantiated with any components that satisfy the high-level component requirements.

```
%-------------------------------------------------------------------
% The system is switching sides when either side has become the
% pilot flying side and that change has not reached the other side
%-------------------------------------------------------------------
switching_sides(s) : bool =
   pilot_flying(Left_Side(s))  AND NOT output(LR_Bus(s)) OR
   pilot_flying(Right_Side(s)) AND NOT output(RL_Bus(s))

%-------------------------------------------------------------------
% R1. At least one side shall be the pilot flying side.
%-------------------------------------------------------------------
  R1: THEOREM
      Reachable_State(s) =>
        Left_Pilot_Flying_Side(s) or Right_Pilot_Flying_Side(s)
%-------------------------------------------------------------------
% R2. At most one side shall be the pilot flying side
%     except while the system is switching sides.
%-------------------------------------------------------------------
  R2: THEOREM
      Reachable_State(s) AND NOT switching_sides(s) =>
         Left_Pilot_Flying_Side(s) /= Right_Pilot_Flying_Side(s)
```

**Fig. 4.** Example of FGS System Requirements in PVS

The verification was then repeated for an asynchronous design in which each side and each bus is driven by its own independent clock (CLK1-4 in Fig. 3). We followed the same process for the asynchronous case. However, the sides and the buses needed to be modified to allow an acknowledgement signal to be exchanged between the two sides to implement a hand-shaking protocol to synchronize on the pilot flying side. Aside from changing the definition of what it meant to be switching sides, the system level requirements did not need to be modified.

The synchronous Dual FGS example was also verified using HOL4. HOL4 proofs were developed using both the next-state approach used with PVS and a stream approach similar to that used in synchronous data flow languages such as Lustre [4]. In the next-state approach, the evolution of each component and the overall system was specified by defining a next-state function that returns the next state given its current state and inputs as arguments. In the stream-based approach, each system variable is specified as a mapping from a natural number representing the system step to the variable's value on that step. The evolution of each component and the overall system is then specified by defining the value of each system variable for each step.

Qualification of a theorem prover may be a difficult task. The largest part of a normal qualification effort is focused on defining operational requirements for the tool (what the tool claims to do – the processes eliminated, reduced, or automated), and then developing a comprehensive test suite to show that those requirements are satisfied over an appropriate range of tool inputs. An alternative approach is to avoid the need to qualify the theorem prover itself by providing an independent check of the proof it produces. This may be feasible depending on the nature of the proof artifacts generated by a particular theorem prover.

PVS is based on a classical strongly-typed higher-order logic and the theorem prover itself is a based on a sequent calculus for this logic. PVS does not normally emit a proof that could be checked by a separate (qualified) proof checking tool, though this option is available. Depending upon the nature of the proof rules used, this expansion could in principle be independently checked by a separate tool. However, we are not aware of this having been done in practice and development of an appropriate independent checker for PVS is still a research topic.

**Table 2.** Summary of Objectives Satisfied by Model Checking

| Objective | Description | A | B | C | D | Notes |
|---|---|---|---|---|---|---|
| A-4.1 | Low-level requirements comply with high-level requirements. | ■ | ■ | ■ | | Established by proof that the high-level requirements are implemented by the low-level requirements and the software architecture. |
| A-4.2 | Low-level requirements are accurate and consistent. | ■ | ■ | ■ | | Established by modeling using an executable language and translation to a formal specification language. |
| A-4.3 | Low-level requirements are compatible with target computer. | | | | | Not addressed |
| A-4.4 | Low-level requirements are verifiable. | ■ | ■ | | | Established by modeling using an executable language and translation to a formal specification language. |
| A-4.5 | Low-level requirements conform to standards. | □ | □ | □ | | Established by use of Simulink/Stateflow design language. |
| A-4.6 | Low-level requirements are traceable to high-level requirements. | □ | □ | □ | | Established by verification of the high-level requirements. |
| A-4.7 | Algorithms are accurate. | ■ | ■ | ■ | | The accuracy of the mode logic is established by model checking. |
| A-4.8 | Software architecture is compatible with high-level requirements. | ■ | ■ | ■ | | Established by proof that the high-level requirements are implemented by the low-level requirements and the software architecture. |
| A-4.9 | Software architecture is consistent | ■ | ■ | ■ | | Established by modeling using an executable language and translation to a formal specification language. |
| A-4.10 | Software architecture is compatible with target computer. | | | | | Not addressed |
| A-4.11 | Software architecture is verifiable. | ■ | ■ | | | Established by modeling using an executable language and translation to a formal specification language. |
| A-4.12 | Software architecture conforms to standards. | □ | □ | □ | | Partially established by use of Simulink/Stateflow. |
| A-4.13 | Software partitioning integrity is confirmed. | | | | | Partitioning integrity has been established using formal methods for several commercial operating systems. This is not addressed in the current case study. |
| FM.A-4.14 | Formal analysis cases and procedures are correct. | ■ | ■ | ■ | | Established by review |
| FM.A-4.15 | Formal analysis results are correct and discrepancies explained. | ■ | ■ | ■ | | Established by review |
| FM.A-4.16 | Requirements formalization is correct. | ■ | ■ | ■ | | Established by review |
| FM.A-4.17 | Formal method is correctly defined, justified, and appropriate. | ■ | ■ | ■ | ■ | Established by review |

■ *Full credit claimed* □ *Partial credit claimed* *Satisfaction of objective is at applicant's discretion*

The HOL4 implementation is based on a small trusted kernel, which encapsulates just the primitive inference rules, axioms, and definition mechanisms of the logic. The logic kernel is an abstract data type, having the property that the only way a theorem can be obtained is ultimately by making primitive inference steps, which are very close in granularity to those in the mathematical definition of the logic. As a

consequence, it is straightforward to instrument HOL kernels so that they emit formal proofs. This has been done in a variety of research projects [8][5]. Programs that check the correctness of such proofs are small and relatively easy to verify.

# 4 Model Checking Case Study

This case study illustrates the use of the Kind [3] and MathWork's Design Verifier model checkers to perform verification activities associated with the outputs of the software design process, focusing on the objectives of Table A-4 in DO-178C and Table FM.A-4 in DO-333. The purpose of these verification activities is to detect any errors that may have been introduced during the software design process (DO-178C Section 5.2). The DO-178C and DO-333 objectives satisfied through model checking are summarized in Table 2.

The specific example used in the model checking case study is the verification of the mode logic of one side of the FGS. Specifically as it relates to the FGS, FAA Advisory Circular AC/ACJ 25.1329 defines a mode as *a system configuration that corresponds to a single (or set of) FGS behavior(s)* [2]. In the FGS, the modes are actually abstractions of their associated flight control law and reflect the current state of the flight control law. The FGS modes are organized into the lateral modes, which control the behavior of the aircraft about the roll and yaw axes of the aircraft and the vertical modes, which control the behavior of the aircraft about the pitch axis of the aircraft. The lateral modes in the example include *Roll Hold*, *Lateral Navigation*, *Lateral Approach,* and *Lateral Go Around*. The vertical modes include *Pitch Hold*, *Vertical Speed*, *Flight Level Change*, *Altitude Hold*, *Altitude Select*, *Vertical Approach,* and *Vertical Go Around*.

In the case study, the mode logic is viewed as the software low-level requirements and is specified using MATLAB Simulink and Stateflow. For example, the Stateflow diagram for the Lateral Navigation (NAV) mode is shown in Fig. 5. Details of the transition guards are specified as Stateflow truth tables.

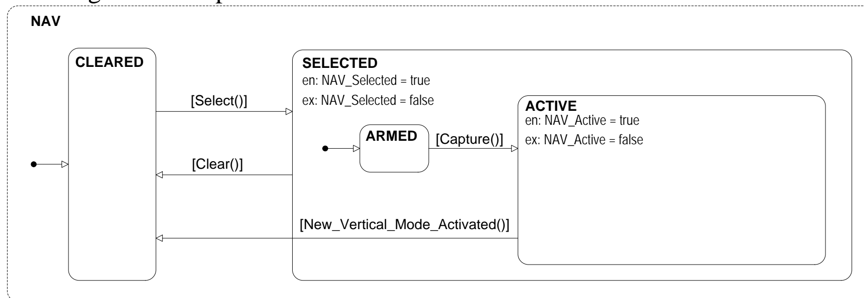


**Fig. 5. Lateral Navigation (NAV) Mode Low-Level Requirements in Stateflow**

The mode logic of the FGS specifies these individual modes and the rules for transitioning between them. To provide proper guidance of the aircraft, these modes are tightly synchronized so that only a small portion of their total state space is actually reachable. For example, since at least one lateral and one vertical mode must

be active and providing guidance whenever the AP is engaged, one mode is designated as the *basic* mode for each axis. The basic mode is automatically activated if no other mode is active for that axis. In this example, the basic modes are *Roll Hold* and *Pitch Hold*. In similar fashion, only one lateral mode and one vertical mode can provide guidance to the AP at the same time, so the mode logic must ensure that at most one lateral and one vertical mode are ever active at the same time.

Other constraints enforce relationships between the modes that are dictated by the characteristics of the aircraft and the airspace. For example, *Vertical Approach* mode is not allowed to become active until *Lateral Approach* mode has become active to ensure that the aircraft is horizontally centered on the localizer before tracking the glideslope. These constraints constitute the high-level software requirements for the mode logic and are captured as 118 high-level properties written in the Lustre specification language. For example, the requirement that at least one lateral mode is always active is specified in Lustre as

```
At_Least_One_Lateral_Mode_Active =
    ROLL_Active or HDG_Active or NAV_Active or
    LAPPR_Active or LGA_Active;
```

To verify that the Simulink and Stateflow low-level requirements for the mode logic satisfy these high-level requirements, the Simlink/Stateflow model of the mode logic is translated into Lustre, the input language of the Kind model checker, using the Rockwell Collins formal translation framework [6] and merged with the high-level requirements written in Lustre. This file can then be analyzed by the Kind model checker. Sixteen errors were discovered in the mode logic using the Kind model checker and three errors are discussed in detail in the full report.

Once these error were corrected, the model checker showed that the Simulink/Stateflow model (the software LLR) complies with the Lustre specifications (the software HLR). This corresponds to Objective A-4.1 in Table 2.

The mode logic was also verified using MATLAB Design Verifier. Properties can be specified either textually as MATLAB function blocks or graphically as Simulink/Stateflow models. For example, the requirement that at least one vertical mode is active is specified textually as a MATLAB function block.

```
function At_Least_One_Vertical_Mode_Active(PITCH_Active, VS_Active,
    FLC_Active, ALT_Active, ALTSEL_Active, VAPPR_Active, VGA_Active)
    % At least one vertical mode shall be active.
    P = ( PITCH_Active  || FLC_Active   || ALT_Active    ||
          ALTSEL_Active || VAPPR_Active || VGA_Active);
sldv.prove(P);
```

The command *sldv.prove(P)* instructs Design Verifier to attempt to prove that P is true for all combinations of inputs and outputs.

Model checkers do not (in general) produce independently checkable output. This means that a model checker must be qualified if its outputs are to be used for certification credit. In addition to the development artifacts that must be provided, tool qualification requires that Tool Operational Requirements (TOR) be defined. The TORs describe what the tool claims to do relative to the certification objectives. Then a comprehensive test suite must be developed to show that those requirements are satisfied over an appropriate range of tool inputs. For a model checker, this would

mean producing a collection of models and properties that span the full range of constructs found in the model and property specification language(s) of the tool. These example models would need to contain property errors which the model checker would have to be shown to identify correctly. We are not aware of any existing efforts to qualify an academic open source model checker like Kind. For commercial tools like Simulink Design Verifier, some support from the tool vendor may be needed to achieve qualification.

**Table 3.** Summary of Objectives Satisfied by Abstract Interpretation

| Objective | Description | A | B | C | D | Notes |
|---|---|---|---|---|---|---|
| A-5.1 | Source Code complies with low level requirements. | | | | | Not addressed |
| A-5.2 | Source Code complies with software architecture. | | | | | Not addressed |
| A-5.3 | Source Code is verifiable. | □ | □ | | | This may be partially satisfied by demonstrating that the code conforms to input restrictions for the tool. |
| A-5.4 | Source Code conforms to standards | □ | □ | □ | | This may be partially or fully satisfied by different analysis tools, depending upon the coding standards and tool qualification |
| A-5.5 | Source Code is traceable to low-level requirements. | | | | | Not addressed |
| A-5.6 | Source Code is accurate and consistent. | □ | □ | □ | | The absence of some classes of run-time errors is established through analysis with abstract interpretation tools. |
| A-5.7 | Output of software integration process is complete and correct. | | | | | Not addressed |
| A-5.8 | Parametric Data Item File is correct and complete. | | | | | Not addressed |
| A-5.9 | Verification of Parametric Data Item File is achieved. | | | | | Not addressed |
| FM.A-5.10 | Formal analysis cases and procedures are correct. | ■ | ■ | ■ | | Established by review |
| FM.A-5.11 | Formal analysis results are correct and discrepancies explained. | ■ | ■ | ■ | | Established by review |
| FM.A-5.12 | Requirements formalization is correct. | ■ | ■ | ■ | | Established by review |
| FM.A-5.13 | Formal method is correctly defined, justified, and appropriate. | ■ | ■ | ■ | ■ | Established by review |

■ *Full credit claimed* □ *Partial credit claimed* *Satisfaction of objective is at applicant's discretion*

## 5 Abstract Interpretation Case Study

This case study illustrates the use of two commercial static analysis tools (AbsInt's Astrée and MathWorks' Polyspace) to perform verification activities associated with the outputs of the software coding process, focusing on the objectives of Table A-5 in DO-178C and Table FM.A-5 in DO-333. The purpose of these verification activities is to detect any errors that may have been introduced during the software coding process (DO-178C Section 5.3). The DO-178C and DO-333 objectives satisfied through abstract interpretation are summarized in Table 3.

The Heading Control Law (Fig. 6) is one of the flight modes in the FGS that is selected by the mode logic. It computes aileron, elevator, rudder, and throttle

commands based on sensor inputs and commanded aircraft heading, altitude, and speed. For this case study, we are using a publicly available model provided by researchers at the University of Minnesota (UMN) [1]. The complete flight software implemented by UMN consists of a sensor data acquisition module, a navigation module, a guidance law, a main control law, and a number of other modules associated with sensor faults and system identification. The heading control law that we are using is one mode available in the main control law. It is comparable in many ways to flight control laws that would be found in commercial aircraft. The other functions of the UMN flight test platform would be carried out by other parts of our FGS example system.

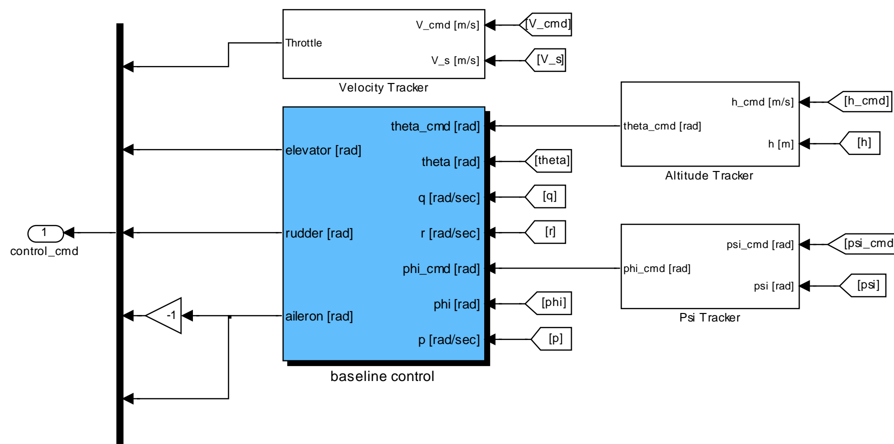


**Fig. 6.** Heading Control Law Model.

In this case study, we have used abstract interpretation to verify the outputs of the software coding and integration process. In the example, this corresponds to verification that the source code implementing the Heading Control Law is correct. Current abstract interpretation tools are best suited to detecting run-time errors in the code rather than satisfaction of behavioral requirements. Verification was performed on C source code generated from the Simulink control law model. Our primary objective was to check the code for accuracy and consistency (DO-333 Section 6.3.4.f and Objective A-5.6 in Table 3). We can also check for unreachable code. We assume that the code will be tested against high and low level requirements–based test cases as part of a traditional test-based verification process.

Astrée can be used to prove that no floating-point overflow errors can occur during the execution of the control code, but this is only possible if the user does some fine-tuning in order to eliminate false alarms. This fine-tuning is done by indicating to Astrée that at certain points in the program, different cases need to be distinguished, which is called *partitioning* in the terminology of Astrée. In order to find the places in the code where partitioning needs to be done, and to determine the conditions which distinguish the different cases in the partitioning, the user needs to have some understanding of the implemented system.

Astrée initially reported four potential issues in the source code, corresponding to C statements which might cause floating-point overflow errors. The code of the

control law implements four integrators, which are protected from overflow by *anti-windup* mechanisms. However, the abstraction made by Astrée keeps the tool from detecting the effectiveness of the overflow prevention. To enable Astrée to prove that these mechanisms are effective, the analysis needs to be guided by some partitioning information provided by the user.

Astrée is, in general, not able to provide direct user feedback to show where the case partitions must be done. However, an experienced user can find the necessary fine-tuning relatively easily. Also, there is some hope that future versions of Astrée will be able to treat this kind of program completely automatically using new partitioning heuristics currently under development.

We also analyzed the example source code using Polyspace and obtained similar results. Polyspace identified unreachable code which was determined to be caused by branch conditions in the anti-windup logic which always evaluate to false. The unused branch of the logic can be optimized away by either the code generator or the compiler, eliminating the unreachable code.

Polyspace also identified several floating-point overflow errors. Polyspace provides a Data Range Specification (DRS) mechanism to specify range limits on inputs to the system. These limits can then be used to more precisely compute the actual range of the variables whose values are computed from these inputs. Once a DRS is setup for each of the system inputs, the potential overflow errors are eliminated.

A DO-178C/DO-330 tool qualification kit is available for Polyspace from the vendor. The qualification kit includes development artifacts and an extensive list of TORs. Test cases are defined with input code for the errors that the tool is intended to detect. For Astrée, a Qualification Support Kit (QSK) is available from its vendor, AbsInt. The currently available QSK can be used for qualification up to level A under DO-178B.

## 6 Conclusion

We have provided an overview of three case studies illustrating the use of different formal methods tools to satisfy the certification objectives defined in DO-178C and its accompanying formal methods supplement, DO-333. These case studies provide a practical demonstration of theorem proving, model checking, and abstract interpretation applied to a Flight Guidance System design that is representative of systems deployed in commercial aircraft. The case studies show how the evidence produced by these three techniques might be used in an actual certification effort. Each technique has strengths and weaknesses and each could be applied to different life cycle data items and different objectives from those described here.

Formal methods and tools have already been used to a limited extent in several actual aircraft certification efforts. However, due to the proprietary nature of the models, code, and other artifacts, it has not been possible to make these results public. We hope that by providing a collection of publicly available examples, our case studies will be useful to industry and government personnel in understanding both the

new certification guidance in DO-333 and the benefits that can be realized through the use of formal methods.

The complete version of the case studies along with all the associated models, code, and verification artifacts will be available as a NASA contractor report.

**Acknowledgements.** This work was sponsored by NASA under contract NNL12AB85T, under subcontract from The Boeing Company. The authors thank Hugh Taylor at Boeing, and Konrad Slind, Jennifer Davis, Siddhartha Bhattacharrya, and Michael Dierkes at Rockwell Collins for their contributions to the case studies.

## References

[1] A. Dorobantu, W. Johnson, FAP. Lie, A. Murch, YC. Paw, D. Gebre-Egziabher, and G.J. Balas, "An Airborne Experimental Test Platform: From Theory to Flight," in *Proceedings of the 2013 American Control Conference*, Washington DC, June 2013.

[2] Federal Aviation Administration, Joint Advisory Circular: Flight Guidance System Appraisal, AC/ACJ 25.1329, 2001.

[3] George Hagen, Cesare Tinelli, Scaling up the formal verification of Lustre programs with SMT-based techniques, in *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design (FMCAD'08)*, Portland, Oregon. IEEE, 2008.

[4] N. Halbwachs , P. Caspi , P. Raymond , D. Pilaud, The Synchronous Dataflow Programming Language LUSTRE, Proceedings of the IEEE, 1991.

[5] Joe Hurd, Composable packages for higher order logic theories, in *Proceedings of the 6th International Verification Workshop (VERIFY 2010)* (M. Aderhold, S. Autexier, and H. Mantel, eds.), July 2010, http://gilith.com/research/papers.

[6] Steven P. Miller, Michael W. Whalen, and Darren D. Cofer, Software Model Checking Takes Off, *Communications of the ACM*, Vol. 33, ISS 2, February, 2010.

[7] M. Norrish, K. Slind, HOL-4 Manual, 1998-2013, http://hol.sourceforge.net/.

[8] Steven Obua and Sebastian Skalberg, Importing HOL into Isabelle/HOL, IJCAR (Ulrich Furbach and Natarajan Shankar, eds.), Lecture Notes in Computer Science, vol. 4130, Springer, 2006.

[9] S. Owre and N. Shankar, *The Formal Semantics of PVS,* NASA Technical Report CS-1999-209321, May, 1999.

[10] RTCA DO-178C, Software Considerations in Airborne Software, December 2011.

[11] RTCA DO-330, Software Tool Qualification Considerations, December 2011.

[12] RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A, December 2011.