NASA/CR–2014-218244

Formal Methods Case Studies for DO-333

Darren Cofer and Steven P. Miller
Rockwell Collins, Inc., Cedar Rapids, Iowa

April 2014

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NASA Aeronautics and Space Database and its public interface, the NASA Technical Report Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

- Access the NASA STI program home page at *http://www.sti.nasa.gov*

- E-mail your question to help@sti.nasa.gov

- Fax your question to the NASA STI Information Desk at 443-757-5803

- Phone the NASA STI Information Desk at 443-757-5802

- Write to:
  STI Information Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR–2014-218244

Formal Methods Case Studies for DO-333

Darren Cofer and Steven P. Miller
Rockwell Collins, Inc., Cedar Rapids, Iowa

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NNL06AA04B

April 2014

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320
443-757-5802

Table of Contents

Acknowledgements 11

Abstract 12

1 Introduction 13

2 Example: Dual-Channel Flight Guidance System 17

3 Case Study: Theorem Proving 21

3.1 Overview of FGS System 21

3.2 Software Verification Plan 23

3.2.1 Formal Specification and Verification Tools 23

3.2.2 Life Cycle Data Items 23

3.2.3 Objectives to Be Satisfied 24

3.2.4 Tool Qualification Issues 28

3.3 The Synchronous Pilot Flying Example 31

3.3.1 High-Level Requirements for the Synchronous Bus 31

3.3.2 Low-Level Requirements for the Synchronous Bus 35

3.3.3 High-level Requirements for the Synchronous FGS Side 37

3.3.4 Low-level Requirements for the Synchronous FGS Side 41

3.3.5 PVS Specification of the Synchronous Pilot Flying System 45

3.3.6 Formal Verification of the Synchronous Pilot Flying Example 48

3.4 The Asynchronous Pilot Flying Example 58

3.4.1 Specification of the Asynchronous Bus Messages 59

3.4.2 High-Level Requirements for the Asynchronous Bus 59

3.4.3 Low-Level Requirements for the Asynchronous Bus 60

3.4.4 High-Level Requirements for the Asynchronous FGS Side 62

3.4.5 Low-Level Requirements for the Asynchronous FGS Side 67
3.4.6 PVS Specification of the Asynchronous Pilot Flying Example 70
3.4.7 Formal Verification of the Asynchronous Pilot Flying Example 73
3.5 The Synchronous Pilot Flying Example in HOL 84
3.5.1 Specification in HOL4 using a Next-State Approach 84
3.5.2 Formal Verification of the Next-State Approach in HOL4 87
3.5.3 Specification in HOL4 using a Streams Approach 90
3.5.4 Formal Verification of the Streams Approach in HOL4 93
4 Case Study: Model Checking 95
4.1 Mode Logic Overview 95
4.2 Software Verification Plan 98
4.2.1 Formal Specification and Verification Tools 98
4.2.2 Life Cycle Data Items 99
4.2.3 Objectives to Be Satisfied 99
4.2.4 Tool Qualification Issues 103
4.3 Specification of the Mode Logic 104
4.3.1 Flight Modes 106
4.3.2 Event Processing 132
4.4 Formal Verification of the Mode Logic 136
4.4.1 Heuristics for Writing Formal Properties 136
4.4.2 Verification of the Mode Logic Using the Kind Model Checker 138
4.4.3 Verification of the Mode Logic Using MATLAB Design Verifier 153
5 Case Study: Abstract Interpretation 158
5.1 Overview of the Heading Control Model 158

5.1.1 Heading Control Model and Code 158
5.1.2 Properties to Be Checked 161
5.2 Software Verification Plan 162
5.2.1 Life Cycle Data Items 164
5.2.2 Objectives to Be Satisfied 164
5.2.3 Tool Qualification Issues 167
5.3 Analysis of the Heading Control Law Source Code with Astrée 167
5.4 Analysis of the Heading Control Law Source Code with Polyspace 169
6 Conclusion 172
7 References 173
Appendix A Acronyms 175
Appendix B Mode Logic Properties 177
Appendix C Mode Logic Error Log 198

Figures

Figure 1 – Relationship of Elements of the Dual-channel FGS Example 14
Figure 2 – Relationship of Case Studies to DO-178C Objectives 15
Figure 3 – Overview of the Flight Guidance System 17
Figure 4 – Flight Control Panel 18
Figure 5 – Primary Flight Display 19
Figure 6 – FGS System Function 21
Figure 7 – Synchronous Pilot Flying System 31
Figure 8 – High-Level Requirements for the Synchronous Bus 32
Figure 9 – Low-Level Requirements for the Synchronous Bus 35
Figure 10 – Theory Interpretation for Synchronous Bus 36
Figure 11 – TCCs Generated from Theory Interpretation for the Synchronous Bus 37
Figure 12 – Synchronous Pilot Flying Side Logic 38
Figure 13 – High-Level Requirements for the Synchronous Side (Part 1) 39
Figure 14 – High-Level Requirements for the Synchronous Side (Part 2) 40
Figure 15 – Low-Level Requirements for the Synchronous Side (Part 1) 42
Figure 16 – Low-Level Requirements for the Synchronous Side (Part 2) 43
Figure 17 – Theory Interpretation for the Synchronous Side 44
Figure 18 – PVS Specification of the Synchronous Pilot Flying System (Part 1) 45
Figure 19 – PVS Specification of the Synchronous Pilot Flying System (Part 2) 46
Figure 20 – Incorrect Statement of Synchronous Requirement R1 in PVS 48
Figure 21 – Synchronous Pilot Flying System Requirements (Part 1) 49
Figure 22 – Correct Statement of Requirement R1 in PVS 50
Figure 23 – PVS Proof of Synchronous Reachable States Valid Theorem 51
Figure 24 – Inductive Sequent for Reachable States 51
Figure 25 – Synchronous Pilot Flying System Requirements (Part 2) 52
Figure 26 – PVS Proof of Requirement R1 for Synchronous Pilot Flying System 53
Figure 27 – PVS Specification of the Pilot Flying System Requirements (Part 3) 54
Figure 28 – PVS Proof of Requirement R3a/b for Synchronous Pilot Flying System 55
Figure 29 – Pilot Flying System Requirements 2 56
Figure 30 – PVS Proof of Switching Transient 57

Figure 31 – Asynchronous Pilot Flying System 58
Figure 32 – PVS Specification of a Bus Message 59
Figure 33 – High-Level Requirements for the Asynchronous Bus..... 60
Figure 34 – Low-Level Requirements for the Asynchronous Bus 61
Figure 35 – Theory Interpretation for Asynchronous Bus..... 62
Figure 36 – Asynchronous Pilot Flying Side Logic..... 63
Figure 37 – High-Level Requirements for the Asynchronous Side (Part 1)..... 64
Figure 38 – High-Level Requirements for the Asynchronous Side (Part 2)..... 65
Figure 39 – High-Level Requirements for the Asynchronous Side (Part 3)..... 66
Figure 40 – High-Level Requirements for the Asynchronous Side (Part 4)..... 67
Figure 41 – Low-Level Requirements for the Asynchronous Side (Part 1) 68
Figure 42 – Low-Level Requirements for the Asynchronous Side (Part 2) 69
Figure 43 – Theory Interpretation for the Asynchronous Side 70
Figure 44 – PVS Specification of the Asynchronous Pilot Flying Example – Part 1..... 71
Figure 45 – PVS Specification of the Asynchronous Pilot Flying Example – Part 2..... 72
Figure 46 – Asynchronous Pilot Flying System Requirements (Part 1)..... 74
Figure 47 – Asynchronous Pilot Flying System Requirements (Part 2)..... 75
Figure 48 – Asynchronous Pilot Flying System Requirements (Part 3)..... 76
Figure 49 – Lemmas Added to Side_HLR to Support Theorem Proving..... 78
Figure 50 – Grind-use-grind PVS Proof Strategy..... 79
Figure 51 – PVS Proof of Asynchronous Reachable States Valid Theorem..... 79
Figure 52 – Asynchronous Pilot Flying System Requirements (Part 4)..... 80
Figure 53 – Incorrect Statement of Asynchronous Requirement R3..... 80
Figure 54 – Asynchronous Pilot Flying System Requirements (Part 5)..... 81
Figure 55 – Incorrect Statement of Asynchronous Requirement R5..... 82
Figure 56 – Asynchronous Pilot Flying System Requirements (Part 6)..... 83
Figure 57 – Rise and Data Type Definitions 85
Figure 58 – Initial State Definitions..... 86
Figure 59 – Next State Definitions 87
Figure 60 – Valid State Definition..... 88
Figure 61 – Custom Simplification Set *sys_ss*..... 89

Figure 62 – HOL4 Proof that all Reachable States are Valid 89
Figure 63 – HOL4 Proof that All Valid States Satisfy R1 90
Figure 64 – HOL4 Proof that All Valid States Satisfy R1 90
Figure 65 – Bus Specification with Streams Approach 90
Figure 66 – Two-Delay Bus Specification with Streams Approach 91
Figure 67 – 'Rise' Definition with Streams Approach 91
Figure 68 – Alternate 'Rise' Definition with Streams Approach 91
Figure 69 – System Specification with Streams Approach 92
Figure 70 – Side Specification with Streams Approach 92
Figure 71 – Final Bus Specification for Streams Approach 93
Figure 72 – Statement of R4 Property 93
Figure 73 – Statement of R1 Property 93
Figure 74 – Proof of R4 Property for Streams Approach 94
Figure 75 – Proof of R1 Property for Streams Approach 94
Figure 76 – A Non-Arming Mode 96
Figure 77 – An Arming Mode 96
Figure 78 – A Capture/Track Mode 97
Figure 79 – Mode Logic Top Level 105
Figure 80 – Flight Modes Subsystem 106
Figure 81 – FD Mode Logic 107
Figure 82 – ANNUNCIATIONS Mode Logic 109
Figure 83 – LATERAL Modes 110
Figure 84 – Heading Select (HDG) Mode 112
Figure 85 – Lateral Navigation (NAV) Mode 113
Figure 86 – Lateral Approach (LAPPR) Mode 115
Figure 87 – Lateral Go Around (LGA) Mode 117
Figure 88 – Roll Hold (ROLL) 118
Figure 89 – VERTICAL Modes 120
Figure 90 – Vertical Speed (VS) Mode 121
Figure 91 – Flight Level Change (FLC) Mode 123
Figure 92 – Altitude Hold (ALT) Mode 125

Figure 93 – Altitude Select (ALTSEL) Mode 126
Figure 94 – Vertical Approach (VAPPR) Mode..... 128
Figure 95 – Vertical Go Around (VGA) Mode 130
Figure 96 – Pitch Hold (PITCH) Mode 132
Figure 97 – Event Processing 133
Figure 98 – Seen Logic 134
Figure 99 – At Least One Lateral Mode Active (Lustre)..... 138
Figure 100 – Weaker Version of at Least One Lateral Mode Active (Lustre) 139
Figure 101 – At Most One Lateral Mode Active (Lustre) 140
Figure 102 – VAPPR Active Only If LAPPR Active (Lustre)..... 141
Figure 103 – LGA Active If and Only If VGA Active (Lustre)..... 141
Figure 104 – Overspeed Implies FLC, ALT, ALTSEL, or PITCH Active (Lustre)..... 142
Figure 105 – Overspeed and PITCH Transitory (Lustre) 142
Figure 106 – HDG Switch Pressed Selects HDG (Lustre) 143
Figure 107 – Definition of RISING (Lustre) 143
Figure 108 – HDG Switch Pressed Selects HDG Using Internal Variables (Lustre)..... 144
Figure 109 – No Higher Event Than HDG Switch Pressed (Lustre)..... 145
Figure 110 – Functional Requirements for Clearing HDG Mode (Lustre) 145
Figure 111 – Initial Functional Requirements for Activating NAV Mode (Lustre)..... 146
Figure 112 – Correct Functional Requirements for Activating NAV Mode (Lustre) 146
Figure 113 – VGA Clear Error 147
Figure 114 –Counterexample for Clearing VGA Error 148
Figure 115 – Counterexample for FLC Select Error 150
Figure 116 – ALTSEL Select Error (Lustre) 151
Figure 117 – Counterexample for ALTSEL Select Error..... 152
Figure 118 – At Least One Lateral Mode Active (Design Verifier)..... 153
Figure 119 – NAV Active When Capture Cond Met (Design Verifier)..... 154
Figure 120 – At Least One Vertical Mode Active (Design Verifier) 154
Figure 121 – Properties Subsystem (Design Verifier)..... 155
Figure 122 – Top-Most Model (Design Verifier)..... 157
Figure 123 – Heading Control Law Model..... 159

Figure 124 – Fragment of Autogenerated C Code for Heading Control Model 160
Figure 125 – Astrée Analysis Results 168
Figure 126 – Astrée Directives to Define Partitions 169
Figure 127 – Anti-Windup Logic 170

Tables

Table 1 – Summary of Objectives Satisfied by Theorem Proving 25
Table 2 – Summary of Objectives Satisfied by Model Checking..... 101
Table 3 – Turn FD On..... 107
Table 4 – Lateral Mode Manually Selected..... 108
Table 5 – Vertical Mode Manually Selected 108
Table 6 – Turn FD Off..... 108
Table 7 – Turn Annunciations On 109
Table 8 – Turn Annunciations Off..... 109
Table 9 – New Lateral Mode Activated..... 111
Table 10 – HDG Select..... 112
Table 11 – HDG Clear 113
Table 12 – HDG Will Be Activated..... 113
Table 13 – NAV Select..... 114
Table 14 – NAV Capture 114
Table 15 – NAV Clear 114
Table 16 – NAV Will Be Activated..... 115
Table 17 – LAPPR Select 115
Table 18 – LAPPR Capture 116
Table 19 – LAPPR Clear 116
Table 20 – LAPPR Will Be Activated..... 116
Table 21 – LGA Select 117
Table 22 – LGA Clear..... 118
Table 23 – LGA Will Be Activated 118
Table 24 – Lateral Mode Active 119
Table 25 – New Vertical Mode Activated 121
Table 26 – VS Select..... 122
Table 27 – VS Clear..... 122
Table 28 – VS Will Be Activated 122
Table 29 – FLC Select 123
Table 30 – FLC Clear 124

Table 31 – FLC Will Be Activated 124
Table 32 – ALT Select 125
Table 33 – ALT Clear 126
Table 34 – ALT Will Be Activated 126
Table 35 – ALTSEL Select 127
Table 36 – ALTSEL Capture 127
Table 37 – ALTSEL Track 127
Table 38 – ALTSEL Clear 128
Table 39 – ALTSEL Will Be Activated 128
Table 40 – VAPPR Select 129
Table 41 – VAPPR Capture 129
Table 42 – VAPPR Clear 129
Table 43 – VAPPR Will Be Activated 130
Table 44 – VGA Select 131
Table 45 – VGA Clear 131
Table 46 – LGA Will Be Activated 131
Table 47 – Vertical Mode Active 132
Table 48 – FLC Select Error 149
Table 49 – Summary of Objectives Satisfied by Abstract Interpretation 165
Table 50 – Initial Polyspace Analysis Results 169
Table 51 – Unproven Runtime Checks 171
Table 52 – Coding Rules Analysis 171

Acknowledgements

Several individuals contributed to the case studies described in this report. Darren Cofer of Rockwell Collins provided general oversight and developed the sections on meeting DO-333 objectives and tool qualification. Steven P. Miller and Jennifer Davis of Rockwell Collins developed the Pilot Flying theorem proving case study based on PVS with assistance from Konrad Slind, also of Rockwell Collins. Jennifer Davis and Konrad Slind provided the HOL4 version of the Pilot Flying case study. Cesar Munoz of the NASA Langley Research Center provided valuable input regarding tool qualification and soundness for the PVS theorem prover. Sam Owre of SRI International and Mike Whalen of the University of Minnesota provided assistance on the use of PVS.

Steven P. Miller developed the Mode Logic model checking case study based on MATLAB Simulink/Stateflow® and the Kind and MATLAB Design Verifier™ model checkers with assistance from Andrew Gacek and Daren Cofer of Rockwell Collins. Cesare Tinelli of the University of Iowa provided valuable input regarding tool qualification and soundness for the Kind model checker.

Michael Dierkes of Rockwell Collins France developed the Heading Control abstract interpretation case study based on Astrée. Sidhartha Battacharyya of Rockwell Collins developed the Polyspace® version of the same case study. Gary Balas of the University of Minnesota provided the Heading Control flight control law.

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. The supplement identifies the modifications and additions to DO-178C and DO-278A objectives, activities, and software life cycle data that should be addressed when formal methods are used as part of the software development process. This report presents three case studies describing the use of different classes of formal methods to satisfy certification objectives for a common avionics example – a dual-channel Flight Guidance System. The three case studies illustrate the use of theorem proving, model checking, and abstract interpretation. The material presented is not intended to represent a complete certification effort. Rather, the purpose is to illustrate how formal methods can be used in a realistic avionics software development project, with a focus on the evidence produced that could be used to satisfy the verification objectives found in Section 6 of DO-178C.

1 Introduction

RTCA DO-333, Formal Methods Supplement to DO-178C and DO-278A [35] provides guidance for software developers wishing to use formal methods in the certification of airborne systems and air traffic management systems. The supplement identifies the modifications and additions to DO-178C [33] objectives, activities, and software life cycle data that should be addressed when formal methods are used as part of the software development process. This includes artifacts that would be expressed using some formal notation and the verification evidence that could be derived from them.

This report presents three case studies describing the use of different classes of formal methods to satisfy DO-178C certification objectives. The material presented is not intended to represent a complete certification effort. Rather, the purpose is to illustrate how formal methods can be used in a realistic avionics software development project, with a focus on the evidence produced that could be used to satisfy the verification objectives found in Section 6 of DO-178C.

The case studies examine different aspects of a common avionics example – a dual-channel Flight Guidance System (FGS) shown in Figure 1. While not intended as a complete example, it is representative of the issues encountered in actual avionics development projects and includes design artifacts specified using PVS, MATLAB Simulink/Stateflow®, and C source code. These files are available for download and use without restriction from the same site where this report is posted. A description of this example is provided in Section 2.

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 purpose here is to illustrate a reasonable application of each of these techniques for satisfying certification objectives.

DO-333 provides general guidance that is applicable to the overall verification process when formal methods are used. This includes requirements for the use of formal notations with unambiguous, mathematically defined syntax and semantics, soundness of the formal analysis methods used, and justification of all assumptions used in each formal analysis. 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 Figure 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. Some of the objectives do not need to be satisfied for the less critical Level C or Level D software.

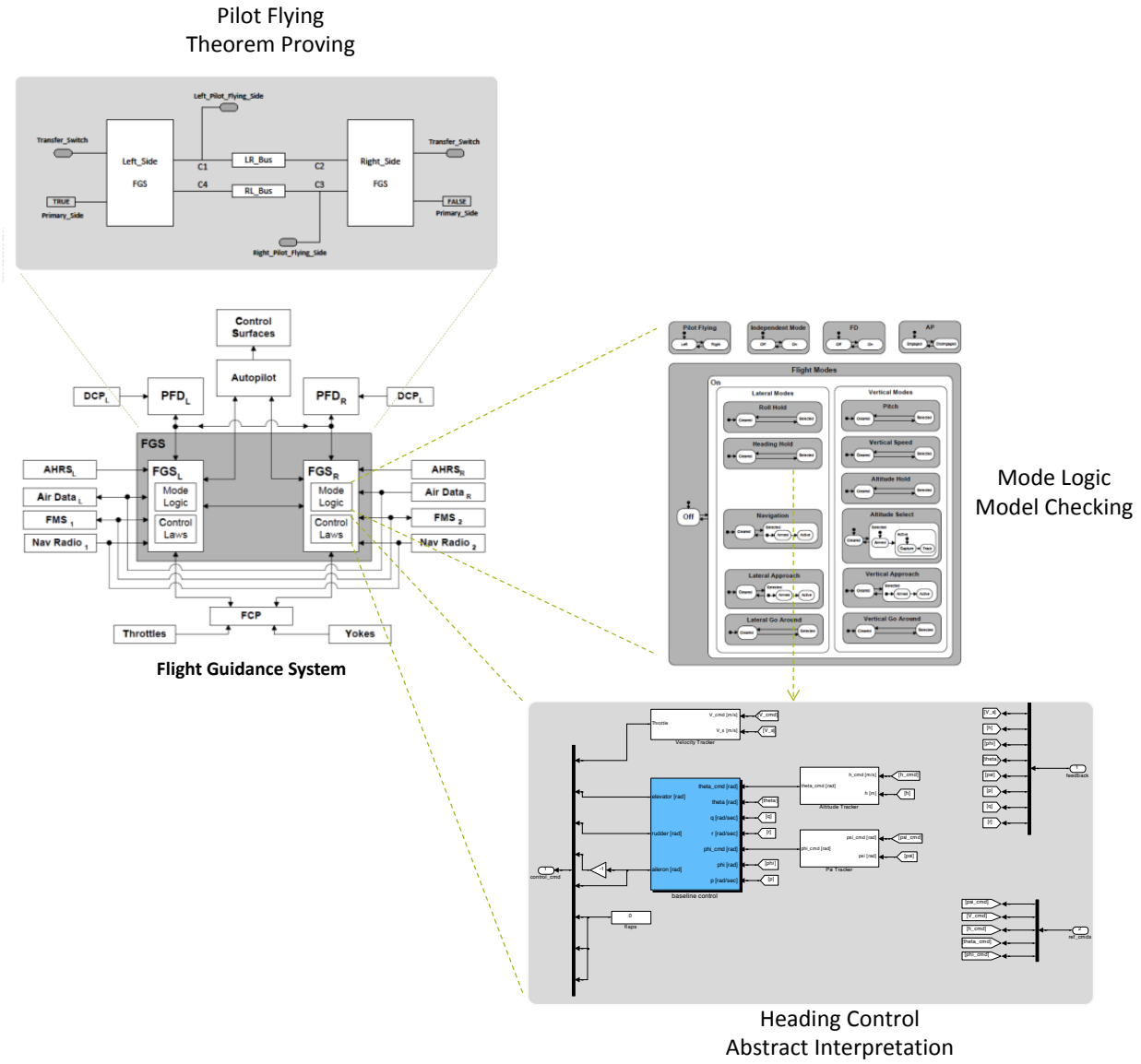

Figure 1 – Relationship of Elements of the Dual-channel FGS Example

As shown in Figure 2, theorem proving was applied to the verification of the High-Level Requirements, model-checking was applied to verification of the Low-Level Requirements and Software Architecture, and abstract interpretation was applied to verification of the Source Code.

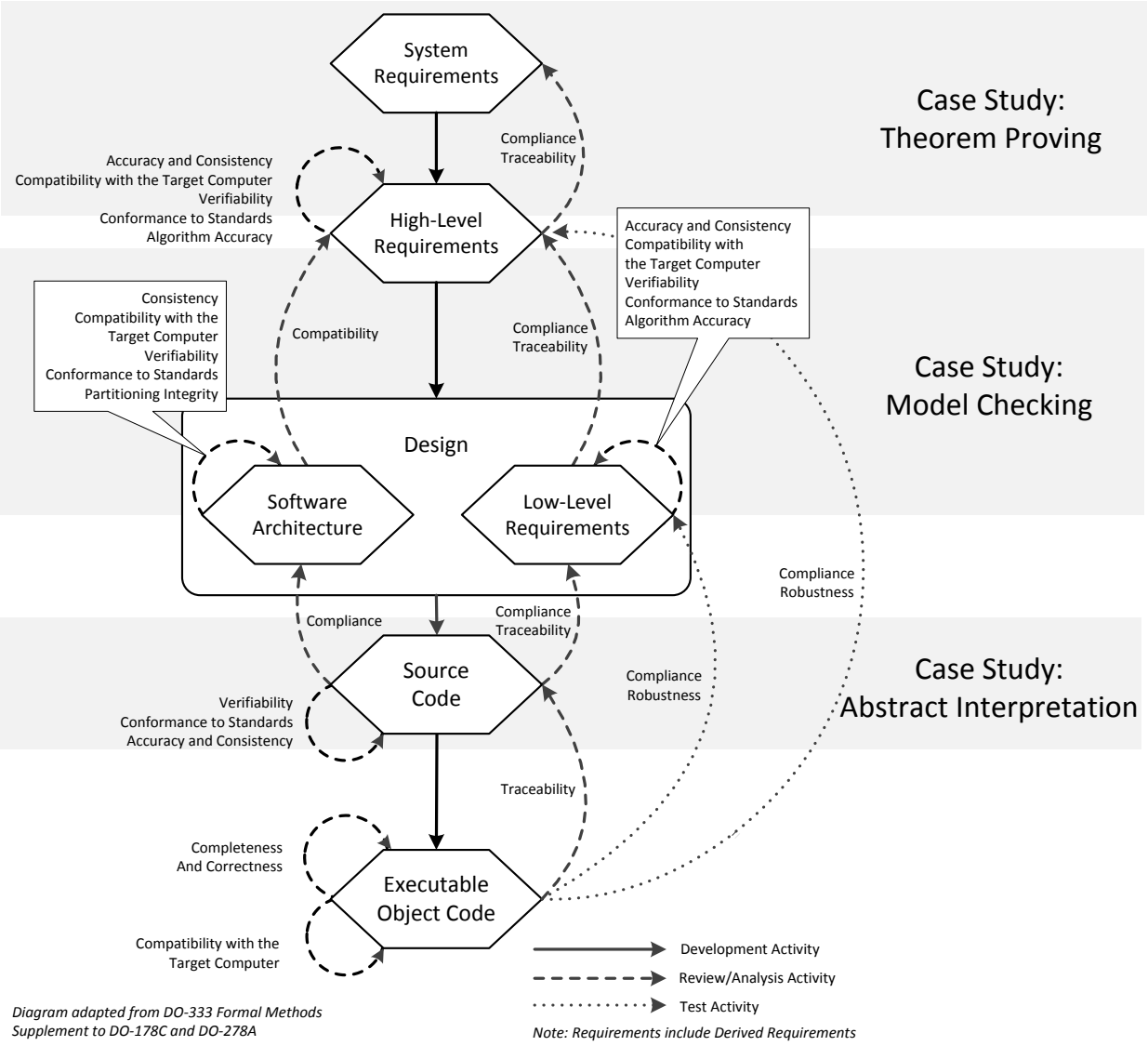

Figure 2 – Relationship of Case Studies to DO-178C Objectives

Theorem proving was applied to the verification of the High-Level Requirements for the synchronization 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 was applied to the verification of the Low-Level Requirements 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 was 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 require 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.

Each case study includes:

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

As a result of the tools and objectives we have chosen for these case studies, there are some parts of DO-333 that are not covered. In particular, we do not address the verification of Executable Object Code (Table FM.A-6 objectives), nor do we address the replacement of coverage testing by formal analysis (Table FM.A-7 objectives).

2 Example: Dual-Channel Flight Guidance System

We will illustrate the use of formal methods to satisfy DO-178C objectives with three case studies built around a common avionics example, a dual-channel Flight Guidance System (FGS). In this chapter, we provide a high level description of the FGS and how it interacts with the other avionics systems and with the flight crew.

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. An overview of an FCS that emphasizes the role of the FGS is shown in Figure 3.

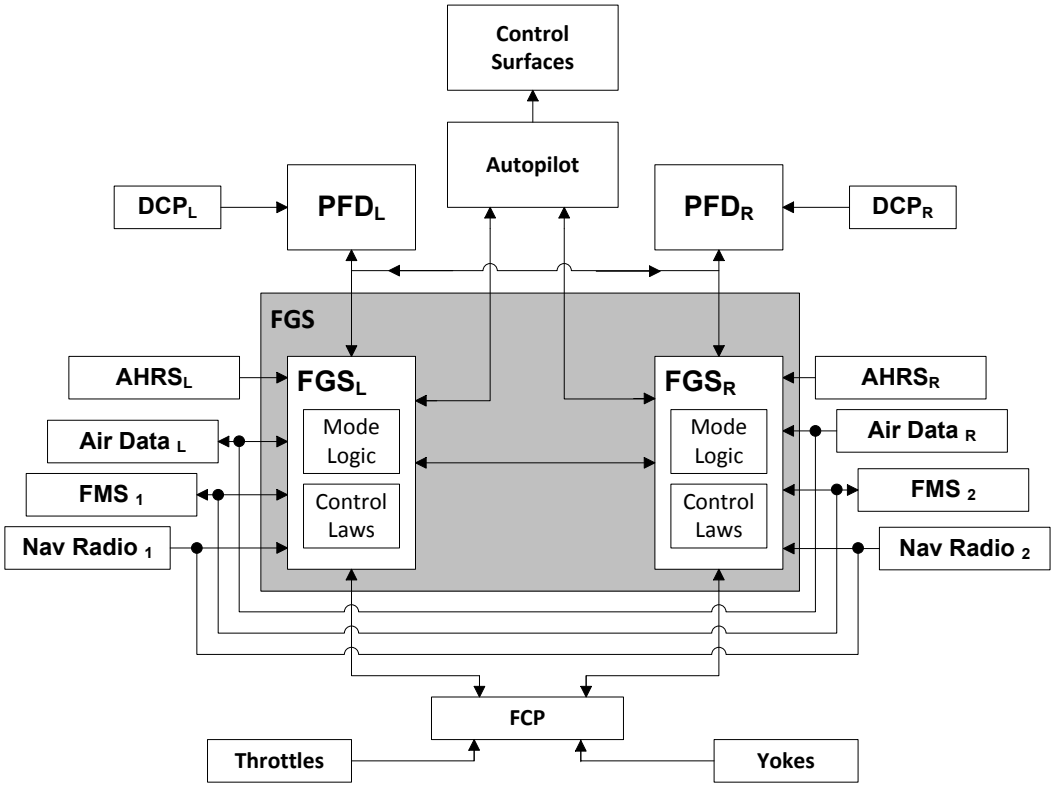

Figure 3 – Overview of the Flight Guidance System

As shown in Figure 3 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 flight crew interacts with the FGS primarily through the Flight Control Panel (FCP), shown in Figure 4. The FCP includes switches for turning the Flight Director (FD) on and off, switches for selecting the different flight modes such as vertical speed (VS), lateral navigation (NAV), heading select (HDG), altitude hold (ALT), and approach (APPR), the Vertical Speed/Pitch Wheel, and the AP disconnect bar. The FCP also supplies feedback to the crew, indicating selected modes by lighting lamps on either side of a selected mode's button.

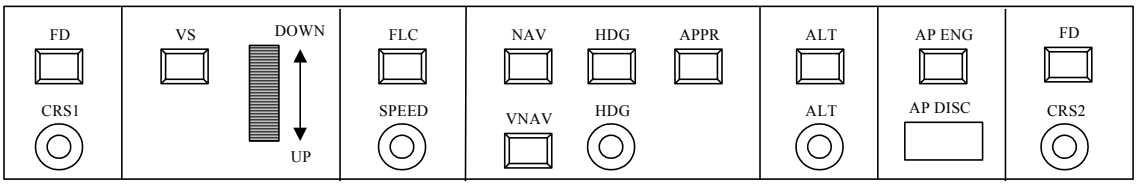

Figure 4 – Flight Control Panel

A few key controls, such as the Go Around button, the AP Disengage switch, and the SYNC switch are provided on the control yokes and throttles and routed through the FCP to the FGS. Navigation sources are selected through the Display Control Panel (DCP), with the selected navigation source routed through the PFD to the FGS.

As shown in Figure 3, 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. These are annunciated, or displayed, on the Primary Flight Displays (PFD) along with a graphical depiction of the flight guidance commands generated by the FGS.

A simplified image of a Primary Flight Display (PFD) is shown in Figure 5. The PFDs display essential information about the aircraft, such as airspeed, vertical speed, attitude, the horizon, and heading. The active lateral and vertical modes are displayed (annunciated) at the top of the display. The annunciations in Figure 5 indicate that the current active lateral mode is Roll Hold (ROLL), the active vertical mode is Pitch Hold (PITCH), and that the Altitude Select (ALTSEL) mode is armed.

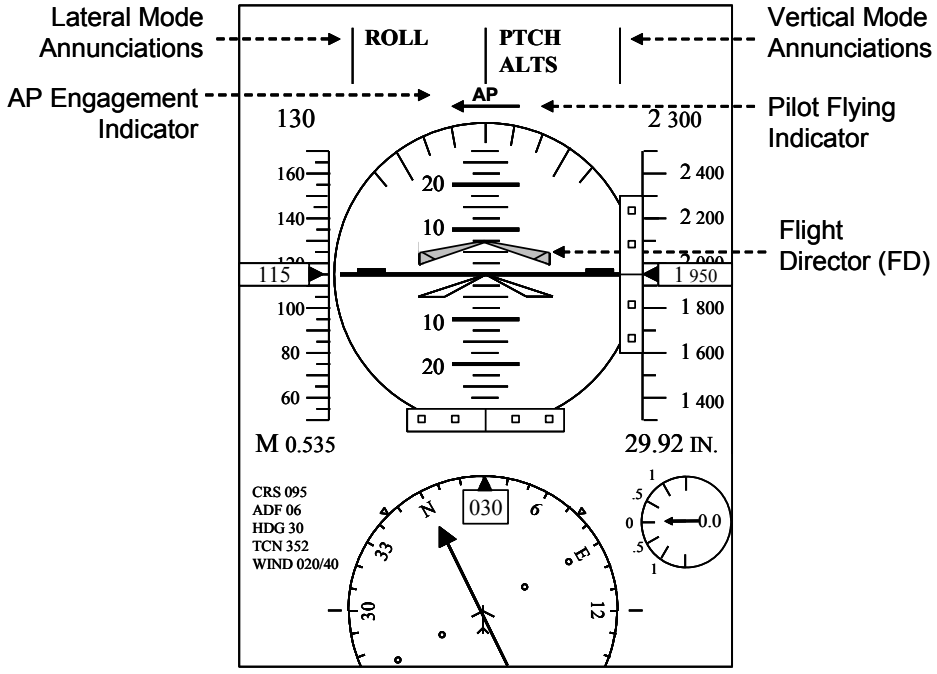

Figure 5 – Primary Flight Display

The large sphere in the center of the PFD is the attitude indicator. The horizontal line across its middle is the artificial horizon. The current pitch and roll of the aircraft is indicated by a white wedge representing the aircraft in the middle of the attitude indicator. Figure 5 depicts an aircraft in level flight with zero degrees of roll and zero degrees of pitch.

The graphical presentation of the pitch and roll guidance commands on the PFD are referred to as the Flight Director (FD)[1]. The pitch and roll guidance commands are shown as a grey wedge in the sky/ground ball. When the AP is not engaged, these are interpreted as guidance to the pilot. When the AP is engaged, these indicate the direction the aircraft is being steered by the

[1] The term Flight Director is also commonly used to refer to the logic that computes the pitch and roll guidance commands.

AP. Engagement of the AP is indicated by the letters AP displayed directly under the mode annunciations. Figure 5 depicts an aircraft in which the AP is engaged and the FD is commanding the pilot to pitch up 7.5 degrees.

In most flight modes, the FGS operates in *dependent* mode where only one FGS channel is *active* and provides guidance to the FD and the AP. The other side serves as a hot spare and sets its modes to match those of the active side. This is indicated by the Pilot Flying indicator displayed directly below the mode annunciations which points to the pilot flying (active) side. In some of the more critical modes such as Approach, Takeoff, and Go Around, the FGS operates in *independent* mode where both channels are active and independently computing guidance commands that must agree for the AP to be engaged.

3 Case Study: Theorem Proving

This section illustrates the use of theorem proving to verify important system properties of the dual-channel FGS. We demonstrate the use of two different interactive theorem proving tools: PVS and HOL4. The majority of the case study is carried out using PVS, and then repeated (on a part of the example) using HOL4 to illustrate the differences in the approaches.

The rest of this section is organized as follows. Section 3.1 provides an overview of the FGS system and the functionality that we will be verifying in this case study. Section 3.2 describes the software verification plan, identifying the life-cycle data items to be produced, the DO-178C objectives to be satisfied, and tool qualification issues. Section 3.3 describes the synchronous example, including how the components are specified in PVS, how the entire system is composed from the components, and how the system requirements are verified. Section 3.4 extends the synchronous example to the asynchronous case by introducing independent clocks for each component and repeats the verification. Finally, Section 3.5 presents an alternative specification and verification of the synchronous example using HOL4.

3.1 Overview of FGS System

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 Figure 6.

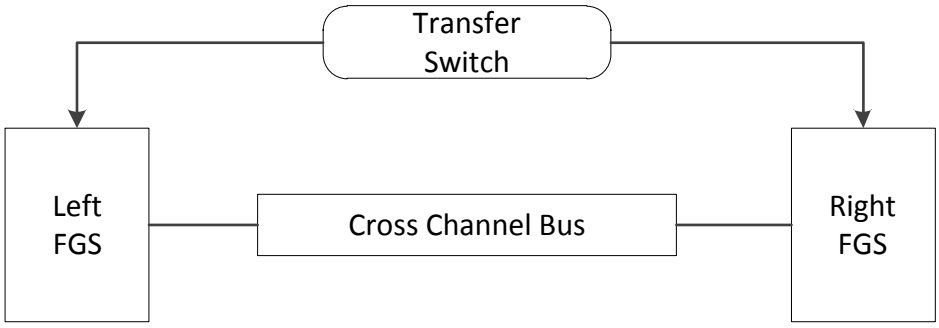

Figure 6 – FGS System Function

Most of the time, the FGS operates in *dependent* mode where only one FGS channel is *active* and provides guidance to the FD and 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 located

above the FCP. The other side serves as a hot spare and sets its modes to agree with those of the active side. In some of the more critical modes such as Approach, Takeoff, and Go Around, the FGS operates in *independent* mode where both channels are active and independently compute guidance commands that must agree for the AP to be engaged, regardless of which side is the current pilot flying 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.

Note that these requirements are system-level requirements that encompass the two sides of the FGS and the cross-channel bus between them.

The overview of Figure 6 provides no indication whether the two FGS execute synchronously or asynchronously. In some designs, such as in a Time-Triggered Architecture (TTA), all components are driven off of a single master clock and execute synchronously. In other architectures, such as Avionics Full-Duplex Switched Ethernet (AFDX), each component is driven by its own local clock and the components execute asynchronously relative to each other.

In this case study we first develop a synchronous design in which all components are driven from single master clock. We develop a set of high-level requirements for each FGS side and the cross-channel bus that are completely free of design detail. Using theorem proving, we show that these high-level requirements are consistent (i.e. do not contradict each other) by proving that there is at least one concrete implementation that satisfies the high-level component requirements. We then show 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. We then extend this example to the more complex asynchronous case by introducing independent clocks for each component.

3.2 Software Verification Plan

In this case study, we will use theorem proving 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. Specifically, this case study will verify the high-level software requirements for the synchronization of the pilot flying side of the FGS and show that the system architecture, the high-level software requirements, and the high-level hardware requirements comply with the system requirements.

3.2.1 Formal Specification and Verification Tools

The PVS formal specification language will be used to specify the system architecture, system requirements, high-level software and hardware requirements of the system components, and candidate low-level software and hardware requirements of the system components. Verification of all formal properties will be performed using the PVS theorem proving system.

3.2.2 Life Cycle Data Items

Life cycle data items are provided for both the synchronous and asynchronous examples using the PVS formal specification language. To facilitate comparison of the two examples, the same names are used for comparable data items in each example.

System Architecture The system architecture is captured in the PVS theory *Pilot_Flying_System*. This theory describes how the system components interact in the overall system.

System Requirements 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.

High-Level Software Requirements 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.

Low-Level Software Requirements Candidate low-level requirements are specified for an FGS side in the *Side_LLR* theory. This theory uses interpreted types, constants, and functions to define a specification that is consistent-by-construction. The *Side_Interpretation* theory is used to prove that the low-level requirements comply with (i.e. implement) the high-level requirements, proving that the high-level requirements are consistent.

High-Level Hardware Requirements The high-level hardware requirements are specified for the cross-channel bus in the *Bus_HLR* theory. This theory uses axioms and uninterpreted types, constants, and functions to eliminate all design detail from the requirements. The axioms are proven consistent by demonstrating that at least one concrete implementation exists that satisfies the axioms.

Low-Level Hardware Requirements Candidate low-level requirements are specified for the cross-channel bus in the *Bus_LLR* theory. This theory uses interpreted types, constants, and functions to define a specification that is consistent-by-construction. The *Bus_Interpretation* theory is used to prove that the low-level requirements comply with (i.e. implement) the high-level requirements, proving that the high-level requirements are consistent.

The low-level requirements for each FGS side and the cross-channel bus exist primarily to prove that the high-level requirements are consistent. While they could be used as the actual low-level requirements for a development, a more likely scenario is that more detailed low-level requirements would be developed and proven to comply with (i.e. implement) the high-level requirements. So long as the more detailed requirements maintain the same interface and are proven to satisfy the high-level requirements, they can be substituted for the low-level requirements without invalidating the verification of the system requirements.

3.2.3 Objectives to Be Satisfied

The DO-178C and DO-333 objectives to be satisfied through theorem proving are summarized in Table 1. Columns A through D in the table indicate for each DO-178C software level whether that objective must be satisfied, and if the objective has been fully or partially satisfied in the case study using formal methods. A more detailed discussion of how each objective is satisfied

is provided in this section. The discussion here is focused on the PVS version of the specification.

Table 1 – Summary of Objectives Satisfied by Theorem Proving

Objective	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*

Objective A-3.1 – High-level requirements comply with system requirements. This objective is demonstrated 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*.

Objective A-3.2 High-level requirements are accurate and consistent. Accuracy is demonstrated by formalizing the high-level software requirements as axioms in the PVS theory *Side_HLR*.

Consistency is demonstrated by proving that the concrete implementation defined in *Side_LLR* implements the axioms of *Side_HLR*. This is done by mapping in theory *Side_Interpretation* each uninterpreted type, constant, and function in *Side_HLR* to its interpreted counterpart in *Side_LLR* and proving that the axioms of *Side_HLR* are implemented by *Side_LLR*.

Objective A-3.4 High-level requirements are verifiable. This objective is demonstrated by formalizing the high-level software requirements as axioms in the PVS theory *Side_HLR* and proving that axioms are satisfied by the concrete implementation defined in *Side_LLR*.

Objective A-3.5 High-level requirements conform to standards. This objective is partially demonstrated by formalizing the high-level software requirements as axioms in the PVS theory *Side_HLR,* establishing that the requirements conform to the PVS specification language. Additional standards, such as naming standards or the presence of comments providing rationale for the requirement, are verified by review.

Objective A-3.6 High-level requirements are traceable to system requirements. This objective is demonstrated by the proof that the high-level requirements comply with the system requirements (objective A-3.1) and by commenting out the high-level requirements one at a time and showing that the proof of the system requirements fail without each high-level requirement.

Objective A-3.7 Algorithms are accurate. This objective is demonstrated by proving that the system-level requirements specified in theory *Pilot_Flying_System_Requirements* are implemented by the system architecture specified in the theory *Pilot_Flying_System* when instantiated with the high-level requirements specified in theory *Side_HLR* and *Bus_HLR*.

Objective FM.A-3.8 Formal analysis cases and procedures are correct. This objective is met through review to ensure that the analyses and procedures satisfy the objectives A-3.1 through A-3.7 for which credit is claimed. The soundness of the proofs is ensured by the PVS verification system once the consistency of all axioms is verified and all types are shown to be non-empty. The remaining reviews consist of validating any assumptions. This includes:

- Confirming that the resolution of cyclic dependencies in the theory *Pilot_Flying_System* is correct and can be implemented.
- Confirming that the addition of asynchronous clocks conservatively models the behavior of the asynchronous system.

- Confirming that assumption that the Transfer Switch is observed by both sides in the same step is acceptable.

Objective FM.A-3.9 Formal analysis results are correct and discrepancies explained. This objective is met through review to ensure that all theorems or lemmas are proven. The PVS verification system will identify any proofs that cannot be completed or that depend on a proof that cannot be completed. Many of the properties had to be revised before they could be proved. Typically, these were due to omissions in the original requirements or oversights introduced by the informality of textual requirements. For example, the requirement R2 "*At most one side shall be the pilot flying side"* had to be changed to "*At most one side shall be the pilot flying side except while the system is switching sides."* Each such discrepancy was explained and fed back into the safety assessment process for review.

Objective FM.A-3.10 Requirements formalization is correct. This objective is met through review to ensure that the formal statement of a requirement is a conservative representation of the informal requirement.

Objective FM.A-3.11 Formal method is correctly defined, justified, and appropriate. This objective is met through a review to ensure:

a. All notations used for formal analysis are verified to have precise, unambiguous, mathematically defined syntax and semantics. Only the PVS language was used which provides a formal syntax and semantics.
b. The soundness of each formal analysis method is normally demonstrated by citing research papers that discuss the soundness of the method(s) implemented in a given tool. In the case of PVS, it may be necessary to restrict the use of certain proof strategies for which evidence of soundness is not available. At a minimum, it must be shown that all axioms are consistent and all types are non-empty. Soundness of both PVS and HOL4 is discussed further in the next section in the context of tool qualification.
c. Assumptions related to each formal analysis are described and justified. The one assumption described above that both sides observe the Transfer Switch in the same step is shown to be acceptable since only the *pilot not flying* side listens for the Transfer Switch. Since this example contains only Boolean and enumerated types and no other

relationships are assumed about its inputs, no assumptions related to the formal analysis (e.g., approximating floating-point numbers as reals) were necessary.

3.2.4 Tool Qualification Issues

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. It is likely that any formal methods tool used for verification as described in DO-333 will require qualification.

According to DO-178C, qualification of a tool is needed when:

1. DO-178C processes are eliminated, reduced, or automated through the use of the tool, and
2. The output of the tool is used without being verified.

The purpose of qualification is to ensure that the tool provides confidence at least equivalent to that of the 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* [34]. DO-330 specifies five different Tool Qualification Levels (TQLs) that define what activities must be performed to qualify a particular software tool. DO-178C, in turn defines three criteria to determine to determine which TQL should apply to a particular tool in a given context.

Criteria 1: A tool whose output is part of the airborne software and thus could insert an error.

Criteria 2: A tool that automates verification processes and thus could fail to detect an error, *and* whose output is used to justify the elimination or reduction of either verification processes *other than* those automated by the tool, or development processes that could have an impact on the airborne software.

Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error (in other words, a tool that automates verification processes).

Once the criteria are applied, DO-178C provides a table in section 12.2.2 to map software level and qualification criteria to the required TQL. For the certification objectives and tool use that we are considering in this case study, Criteria 3 applies. This means that for all airborne

software levels the theorem provers would need to be qualified to TQL-5 (if their results are not independently checked).

Depending on the tool, qualification of a theorem prover may be a difficult task. Even at the lowest qualification level (TQL-5) there are a number of development artifacts that must be produced as part of the qualification process. The largest part of the 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 more or less feasible depending on the nature of the proof artifacts generated by a particular theorem prover.

The PVS proof engine is built from a small set of primitive inference steps. Most of the inference steps are small, but a few involve deep combinations of decision procedures and rewriting. Larger proof strategies can be defined using the primitive ones. Several external proof tools for Binary Decision Diagram (BDD)-based simplification and model checking, monadic second-order reasoning, nonlinear arithmetic, and predicate abstraction have been added to PVS.

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. Standard references for the PVS language are [29], [30], [31] and [32]. However, not all aspects of the language are described. For example, the formal semantics of recursive functions and their termination is not described. This would likely present some challenges to qualification and the requirement to demonstrate soundness of the analysis method.

Unlike Isabelle/HOL or Coq, PVS does not normally emit a proof that could be checked by a separate (qualified) proof checking tool, though this option is available. We have expanded one of our proofs down to the primitive proof rules to demonstrate that this is feasible. 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.

Extant implementations of HOL follow the so-called "LCF approach" to designing theorem provers. This methodology implements the logic by 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. For example, the introduction and elimination rules for the logical connectives constitute the bulk of the implementation of the HOL4 kernel, and these are very simple to implement. In contrast, basic inference steps in systems like PVS tend to be much larger, amounting to invocation of complex combinations of simplifiers and powerful decision procedures.

As a consequence, it is straightforward to instrument HOL kernels so that they produce formal proofs. This has been done in a variety of research projects [28], [15]. Programs that check the correctness of such proofs are small and relatively easy to verify.

Another approach to theorem prover correctness is to verify the kernel once and for all. This has been done for the implementations of the primitive inferences of HOL Light, with respect to an abstract, set-theoretic specification of the logic.

A more extensive overview of the issues for PVS and HOL4, as well as other popular theorem proving environments, can be found in the summary document for the "Trusted Extensions of Interactive Theorem Provers" workshop [36].

3.3 The Synchronous Pilot Flying Example

The synchronous Pilot Flying example consists of four main components, the *Left_Side* FGS, the *Right_Side* FGS, an *LR_Bus* and an *RL_Bus* connecting the two sides as shown in Figure 7.

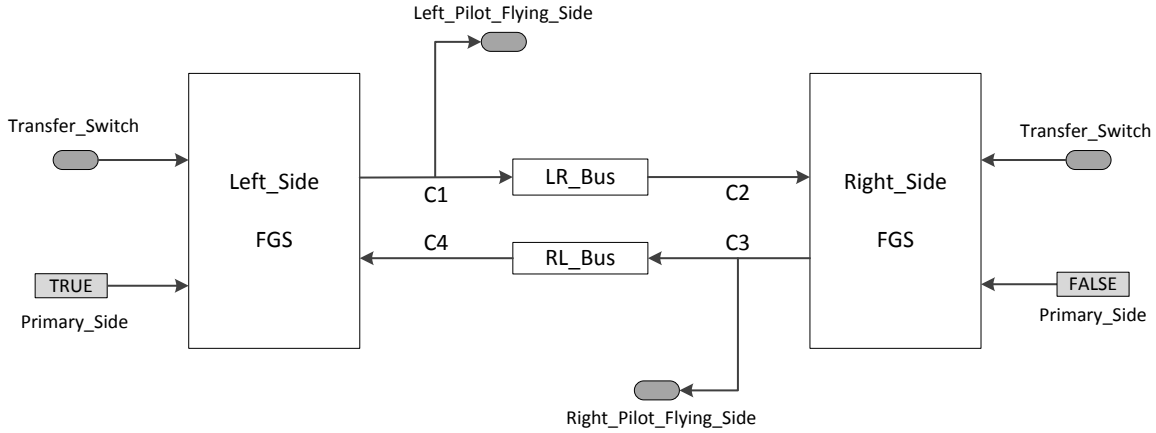

Figure 7 – Synchronous Pilot Flying System

All four components are assumed to be driven by the same master clock and to execute synchronously. Each FGS produces a *Pilot_Flying_Side* Boolean output indicating if it believes itself to be the current pilot flying side. The two buses pass the value produced by one side to the other side, introducing a one-step delay in the process. Each FGS accepts as inputs a Boolean value representing the current value of the *Transfer_Switch*[2] and the *Pilot_Flying_Side* value passed across the bus from the other side. Each FGS also accepts a single Boolean constant indicating if it is the *Primary_Side* that is to be the initial pilot flying side. The constant for the *Left_Side* is set to true to make it the initial pilot flying side while the input to the *Right_Side* is set to false. The intermediate connections C1, C2, C3, and C4 are also labeled on the diagram to facilitate discussion of the formal specification and the proofs.

3.3.1 High-Level Requirements for the Synchronous Bus

Each bus in the synchronous Pilot Flying example transfers its input to its output with a one-step delay. The PVS theory specifying this behavior is shown in Figure 8.

---

[2] Note that there is an implied assumption in this model that the *Transfer Switch* arrives at both sides on the same step. The subsequent discussion shows that this assumption can be safely made since only the pilot not flying side listens for the *Transfer Switch* and no more than one side is ever *the pilot not flying side*.

```
Bus_HLR[Init: bool]: THEORY
BEGIN

    %-------------------------------------------------------------------------
    % State defined as an uninterpreted, non-empty type
    %-------------------------------------------------------------------------
    State: TYPE+

    Initial_State: State

    %-------------------------------------------------------------------------
    % Next state defined as an uninterpreted function
    %-------------------------------------------------------------------------
    next_state: [State, bool -> State]

    %-------------------------------------------------------------------------
    % Output of the bus
    %-------------------------------------------------------------------------
    output: [State -> bool]

    %-------------------------------------------------------------------------
    % High level requirements
    %-------------------------------------------------------------------------
    HLR1: AXIOM
       output(Initial_State) = Init

    HLR2: AXIOM
       forall (s: State, i: bool) :
          output(next_state(s, i)) = i


 END Bus_HLR
```

Figure 8 – High-Level Requirements for the Synchronous Bus

Since the bus introduces a one-step delay in conveying its inputs to its outputs, the initial output value of the bus must be specified. This is done by parameterizing the theory with the *Init* Boolean value. To be able to compose all the components of the Pilot Flying System in a consistent fashion, we adopt the convention of defining a *State* type for each component representing its internal state. Since we want the high-level requirements to be as free of design decisions as possible, we define the type *State* for the synchronous bus to be an uninterpreted type that provides no information about how the state is implemented. Since an empty type can lead to an unsound proof system, we do specify that the type must have at least one value by defining it to PVS as a TYPE+. We also define an uninterpreted constant *Initial_State* to be an element of this type. As its name implies, *Initial_State* will represent the initial state of the bus at start-up.

While the type of the internal state of a component can be defined in PVS, PVS does not provide a mechanism for persisting state as might be done in an object-oriented system. Instead, the current state is passed in as an argument of each function that manipulates it. We adopt the convention of defining for each component a *next_state* function that compute the component's next state from its current state and its inputs and an *output* function that returns the outputs of the component from its current state. For the synchronous bus, the *next_state* function takes a state *s* of the bus and a Boolean *input*, and returns the next internal state of the bus. The *output* function takes a state *s* and returns a Boolean value.

The high-level requirements for the synchronous bus are specified as two PVS axioms. The first requirement, HLR1, simply states that the *output* of the *Initial_State* is the Boolean value *Init* specified as the theory parameter. The second requirement, HLR2, states that the output of the *next_state* of any state *s* and any input *i* is *i*.

These axioms were defined through use of a well-known heuristic for writing axiomatic, or declarative, specifications. We first identify all *constructors* that generate new values of the type *State* and all *extractors* that extract values from elements of the type *State*. For the synchronous bus, the constructors are *Initial_State* and *next_state* and the only extractor is *output*. We then write one or more axioms that describe the result of applying each extractor to each constructor. For the synchronous bus, this results in the two axioms shown in Figure 8.

Writing requirements as axioms allows us to specify the required behavior of a component without committing to a specific design or implementation. Note that these axioms only specify the functionality of a component. In particular, we have not stated whether they are system, software, or hardware requirements. They could be viewed as the high-level functional system requirements for the bus. If the bus is to be implemented primarily in software, they could also be used as the high-level functional software requirements. If the bus is to be implemented primarily in hardware, they could be used as the high-level functional hardware requirements. Such axiomatic or declarative specifications are formal versions of the informal "shall" statements typically written in requirements documents. In fact, they share two common concerns with such requirements, *completeness* and *consistency*.

Completeness refers to whether the axioms (requirements) fully specify what needs to be built. We address this issue in two ways. First, we used the heuristic described earlier to define the

effect of each extractor on each constructor, giving us a systematic way of developing the axioms. More importantly, we prove in Section 3.3.6 that the requirements for the overall Pilot Flying system will be satisfied using any bus that satisfies these axioms, showing that the axioms are sufficiently complete to ensure the required system behavior.

Consistency refers to whether the axioms contradict each other. In an informal development process, inconsistent requirements are typically discovered during development when it becomes apparent that the specified system cannot be built. In the same way, inconsistent axioms cannot be implemented. Even worse, inconsistent axioms introduce unsoundness into a proof system allowing false theorems to be proven. For this reason, whenever axioms are used in a specification, it is important to prove that they are consistent. The standard way of doing this is to demonstrate that an implementation exists that satisfies all the axioms. In the next section, we define a set of low-level requirements for the bus that are consistent by construction and prove that the implementation satisfies the high-level axioms.

3.3.2 Low-Level Requirements for the Synchronous Bus

A PVS theory for an implementation of the synchronous bus is shown in Figure 9.

```
Bus_LLR[Init:bool]: THEORY
BEGIN

    %-------------------------------------------------------------------------
    % Define the bus state and initial state
    %-------------------------------------------------------------------------
    State: TYPE = bool

    Initial_State: State = Init

    %-------------------------------------------------------------------------
    % Next state function
    %-------------------------------------------------------------------------
    next_state(s: State, i: bool): State = i

    %-------------------------------------------------------------------------
    % Output of the bus
    %-------------------------------------------------------------------------
    output(s: State): bool = s

 END Bus_LLR
```

Figure 9 – Low-Level Requirements for the Synchronous Bus

As with the high-level requirements, the theory is parameterized with the initial output value of the bus. We also introduce a type *State* and a constant *Initial_State* of that type. However, this time we give each of these a concrete interpretation. We define the *State* of the bus to be of type Boolean and we assign the theory parameter *Init* to be the value of *Initial_State*. We also provide a concrete interpretation for each function. We define the *next_state* function to return the value of its input *i* and we define the *output* of the bus to be its current state *s*.

Note that this specification does not include any axioms. It is constructed using only the base types of PVS and functions. The PVS type system will ensure that such a *constructive* specification is consistent by generating type-correctness conditions (TCCs) that must be proven using the PVS theorem prover.

This specification is notable in that is actually shorter than the high-level bus requirements. However, it is also more concrete in that it makes the design decision to represent the state of the channel as a single Boolean variable. We could have also chosen to implement the bus using an integer rather than a Boolean or using a queue of size one providing the constructor and extractor

functions were defined appropriately. So long as an implementation provides the same types, constants, and functions and satisfies the two axioms of the high-level requirements of Figure 8, it would be an acceptable implementation of the synchronous bus.

To prove that the low-level bus requirements of Figure 9 implement, or comply with, the high-level bus requirements of Figure 8, we use the theory interpretation capability of PVS. This is shown in Figure 10. The *Bus_Interpretation* theory first imports the *Bus_LLR* theory using the same Boolean theory parameter, *Init*. It then imports the *Bus_HLR* theory, but provides an interpretation for each uninterpreted type, constant, and function based on the *Bus_LLR* specification. For example, it defines the *State* type of the *Bus_HLR* theory to be the *State* type of the *Bus_LLR* theory. In fact, it defines each uninterpreted type, constant and function in *Bus_HLR* to be the corresponding type, constant, or function in *Bus_LLR*. When this theory is typechecked using PVS, it generates the two TCCs shown in Figure 11.

```
Bus_Interpretation[Init : bool] : THEORY

BEGIN

  %-------------------------------------------------------------------------
  % Import low-level requirements (LLR) for a bus
  %-------------------------------------------------------------------------
  IMPORTING Bus_LLR[Init]

  %-------------------------------------------------------------------------
  % Import high-level (HLR) requirements for a bus and
  % define the LLR as an interpretation of the HLR
  %-------------------------------------------------------------------------
  IMPORTING Bus_HLR[Init] {{
                    State         := Bus_LLR.State,
                    Initial_State := Bus_LLR.Initial_State,
                    next_state    := Bus_LLR.next_state,
                    output        := Bus_LLR.output
    }}

  END Bus_Interpretation
```

Figure 10 – Theory Interpretation for Synchronous Bus

A careful examination of these TCCs reveals that they are the axioms of the *Bus_HLR* theory instantiated with the types, constants, and functions defined in the *Bus_LLR* theory. These TCCs can easily be proved using the *typecheck-prove* (M-x tcp) command of PVS, proving that the *Bus_LLR* theory implements the axioms of the *Bus_HLR* theory.

```
% Mapped-axiom TCC generated (at line 14, column 12) for
    % Bus_HLR[Init]
    %     {{ State := Bus_LLR.State,
    %        Initial_State := Bus_LLR.Initial_State,
    %        next_state := Bus_LLR.next_state,
    %        output := Bus_LLR.output }}
  % proved - complete
IMP_Bus_HLR_HLR1_TCC1: OBLIGATION
  Bus_LLR[Init].output(Bus_LLR[Init].Initial_State) = Init;

% Mapped-axiom TCC generated (at line 14, column 12) for
    % Bus_HLR[Init]
    %     {{ State := Bus_LLR.State,
    %        Initial_State := Bus_LLR.Initial_State,
    %        next_state := Bus_LLR.next_state,
    %        output := Bus_LLR.output }}
  % proved - complete
IMP_Bus_HLR_HLR2_TCC1: OBLIGATION
  FORALL (s: State[Init], i: bool):
    Bus_LLR[Init].output(Bus_LLR[Init].next_state(s, i)) = i;
```

Figure 11 – TCCs Generated from Theory Interpretation for the Synchronous Bus

In this way, we have shown that at least one concrete implementation of the *Bus_HLR* theory exists and that the axioms of the *Bus_HLR* are consistent, allowing us to safely use it as the high-level functional requirements for the synchronous bus. The *Bus_LLR* theory can be used as the low-level requirements for the bus or a more detailed set of low-level requirements can be developed and proven to implement the high-level requirements.

3.3.3 High-level Requirements for the Synchronous FGS Side

In this section we develop the high-level requirements for the synchronous FGS side just as we did for the synchronous bus. For the synchronous system, each side executes the simple state machine shown in Figure 12 to determine which side is the current pilot flying side.

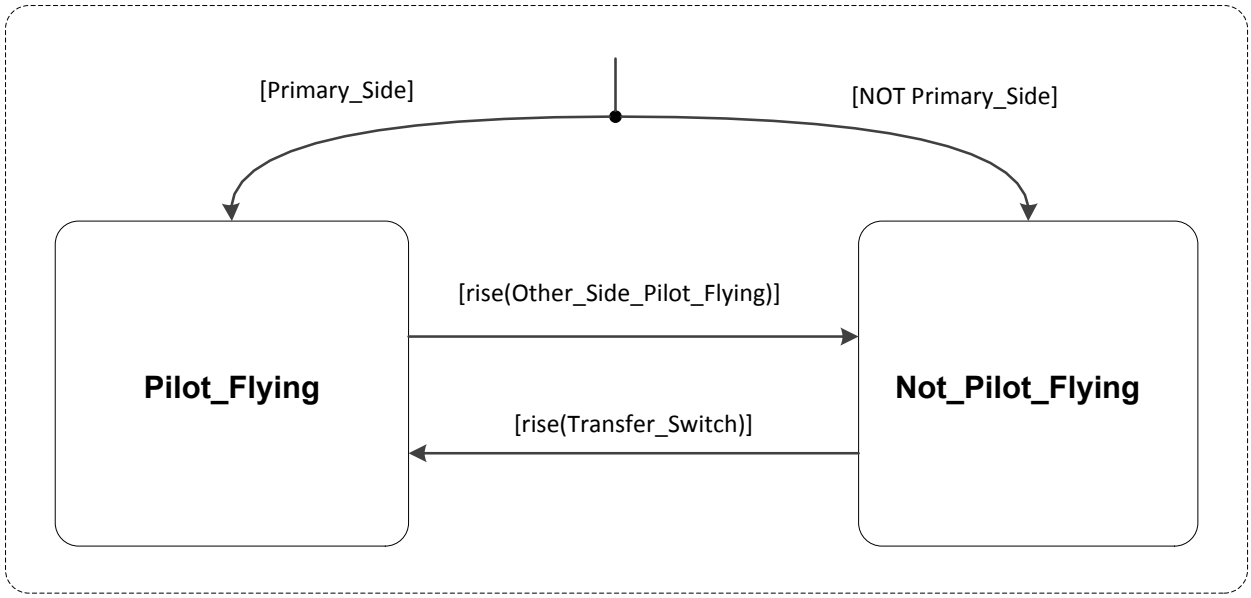

Figure 12 – Synchronous Pilot Flying Side Logic

If a side believes itself to be the *Not_Pilot_Flying* side, it will become the *Pilot_Flying* side when it sees the *Transfer_Switch* pressed (i.e. a rising edge). If a side is the *Pilot_Flying* side, it will become the *Not_Pilot_Flying* side when it sees the other side become the pilot flying side. Thus, it is always the *Not_Pilot_Flying* side that responds to the Transfer Switch, and the current *Pilot_Flying* side always yields when it sees the other side become the *Pilot_Flying* side. The PVS specification of the high-level requirements for a synchronous FGS side is shown in Figure 13 and Figure 14.

The theory is parameterized with whether this side is the *Primary_Side*. Just as with the bus, we introduce an uninterpreted *State* type, an uninterpreted *Initial_State* constant and an uninterpreted *next_state* function. Since each side needs to be able to observe a rising edge of the *Transfer_Switch* and of the other side's *Pilot_Flying* value, each side saves the previous value of the *Transfer_Switch* and the other side's *Pilot_Flying* value passed across the bus.

```
Side_HLR[Primary_Side : bool] : THEORY

BEGIN

    %-------------------------------------------------------------------------
    % State defined as an uninterpreted, non-empty type
    %-------------------------------------------------------------------------
    State: TYPE+

    Initial_State: State

    %-------------------------------------------------------------------------
    % Next state defined as an uninterpreted function
    %-------------------------------------------------------------------------
    next_state: [State, bool, bool -> State]

    %-------------------------------------------------------------------------
    % Extractor functions for this side
    %-------------------------------------------------------------------------
    pre_TS   : [State -> bool]
    pre_OSPF : [State -> bool]

    %-------------------------------------------------------------------------
    % Pilot flying output for this side
    %-------------------------------------------------------------------------
    pilot_flying : [State -> bool]

    %-------------------------------------------------------------------------
    % Auxiliary definitions for stating axioms
    %-------------------------------------------------------------------------
    rise_ospf(s:State, ospf:bool) : bool = NOT pre_OSPF(s) AND ospf
    rise_ts  (s:State, ts  :bool) : bool = NOT pre_TS  (s) AND ts
```

Figure 13 – High-Level Requirements for the Synchronous Side (Part 1)

We specify uninterpreted extractor functions *pre_TS* and *pre_OSPF* that extract these values from the side's state. We also define an extractor *pilot_flying* for the single output of the side. To provide a convenient shorthand for writing axioms, we also define (interpreted) auxiliary functions *rise_ospf* and *rise_ts* as shown. The completion of the theory is shown in Figure 14 where the axioms specifying the side's required behavior are given.

```
    %-------------------------------------------------------------------------
    % High level requirements
    %-------------------------------------------------------------------------
    HLR1: AXIOM
       pilot_flying(Initial_State) = Primary_Side

    HLR2: AXIOM
       pre_TS(Initial_State)

    HLR3: AXIOM
       pre_OSPF(Initial_State) = NOT Primary_Side

    HLR4: AXIOM
       forall (s:State, ts, ospf: bool) :
         pilot_flying(s) AND rise_ospf(s, ospf) =>
            NOT pilot_flying(next_state(s, ts, ospf))

    HLR5: AXIOM
       forall (s:State, ts, ospf: bool) :
          pilot_flying(s) AND NOT rise_ospf(s, ospf) =>
             pilot_flying(next_state(s, ts, ospf))

    HLR6: AXIOM
       forall (s:State, ts, ospf: bool) :
          NOT pilot_flying(s) AND rise_ts(s, ts) =>
             pilot_flying(next_state(s, ts, ospf))

    HLR7: AXIOM
       forall (s:State, ts, ospf: bool) :
          NOT pilot_flying(s) AND NOT rise_ts(s, ts) =>
             NOT pilot_flying(next_state(s, ts, ospf))

    HLR8: AXIOM
       forall (s:State, ts, ospf: bool) :
          pre_TS(next_state(s, ts, ospf)) = ts

    HLR9: AXIOM
       forall (s:State, ts, ospf: bool) :
          pre_OSPF(next_state(s, ts, ospf)) = ospf

END Side_HLR
```

Figure 14 – High-Level Requirements for the Synchronous Side (Part 2)

Axioms HLR1 through HLR3 define the result of applying each of the three extractors to the *Initial_State* constructor. HLR4 through HLR7 define the result of applying the *pilot_flying* extractor to the *next_state* constructor. HLR4 defines the case when this side is the pilot flying side and the other side is observed to become the pilot flying side.[3] HLR5 defines the case where

[3] The "=>" operator is the implication operator (read "A implies B"), where A => B states that if A is true, B must also be true. It is logically equivalent to not A or B.

this side is the pilot flying side and the other side is *not* observed to become the pilot flying side. HLR 6 and HLR7 define the two cases when this side is not the pilot flying side. Note that axioms HLR4 through HLR7 are disjoint in that only one of their antecedents can be true at a time. Finally, HLR8 and HLR9 define the effect of applying the *pre_TS* and *pre_OSPF* extractors to the *next_state* constructor.

3.3.4 Low-level Requirements for the Synchronous FGS Side

In this section, we show that the axioms of Section 3.3.3 are consistent by developing a constructive implementation of the synchronous FGS side and proving that it satisfies these axioms. The PVS theory *Side_LLR* shown in Figure 15 and Figure 16 defines a constructive implementation of a synchronous *Side* by providing a concrete interpretation for each type, constant, and function.

Just as for the high-level requirements, the theory is parameterized with whether it is the *Primary_Side*. The two possible values of the state machine of Figure 12, *Pilot_Flying* and *Not_Pilot_Flying*, are specified as the PVS enumeration type *Pilot_Flying_Side*. The *Initial_Pilot_Flying_Side* is a constant whose value is determined by the *Primary_Side* theory parameter

The *State* of a side is stored in a PVS record structure consisting of three fields – the current value of the *Pilot_Flying_Side* state machine (*st*), the previous value of *Transfer_Switch* (*pre_ts*), and the previous value of *Other_Side_Pilot_Flying* input *(pre_ospf)*. The last two values are used in the next state function to determine whether a rising edge of the Transfer Switch or the other side's pilot flying output has occurred. The initial value of *pre_ts* is set to true to ensure that a rising edge of the Transfer Switch is not detected in the initial state. The initial value of *pre_ospf* is set to match the initial value of the pilot flying output from the other side.

```
Side_LLR[Primary_Side : bool] : THEORY

BEGIN

    %-------------------------------------------------------------------------
    % Pilot flying state machine values
    %-------------------------------------------------------------------------
    Pilot_Flying_Side : TYPE =  {PilotFlying, NotPilotFlying}

    Initial_Pilot_Flying_Side : Pilot_Flying_Side =
            IF Primary_Side THEN  PilotFlying
            ELSE NotPilotFlying ENDIF

    %-------------------------------------------------------------------------
    % Definition of state and initial state for this side
    %-------------------------------------------------------------------------
    State :TYPE+          = [# st       : Pilot_Flying_Side,
                               pre_ts   : bool,
                               pre_ospf : bool              #]

    Initial_State: State  = (# st       := Initial_Pilot_Flying_Side,
                               pre_ts   := TRUE,
                               pre_ospf := NOT Primary_Side  #)

    %-------------------------------------------------------------------------
    % Extractor functions for this side
    %-------------------------------------------------------------------------
    pre_TS  (s: State) : bool = pre_ts(s)
    pre_OSPF(s: State) : bool = pre_ospf(s)

    %-------------------------------------------------------------------------
    % Auxiliary functions for defining next state function
    %-------------------------------------------------------------------------
    rise_ts  (s:State, ts  : bool) : bool = NOT pre_ts  (s) AND ts
    rise_ospf(s:State, ospf: bool) : bool = NOT pre_ospf(s) AND ospf
```

Figure 15 – Low-Level Requirements for the Synchronous Side (Part 1)

To maintain consistency with the high-level requirements, extractor functions *pre_TS* and *pre_OSPF* are defined that simply extract the relevant fields from the state record. Auxiliary functions *rise_ts* and *rise_ospf* are also defined to simplify defining the *next_state* function. The *next_state* function (shown in Figure 16) computes the next state of a side given its current state and inputs.

The next state is computed by first computing the next *Pilot_Flying_Side* value *next_pfs*. For example, transition 1 is taken if the *Pilot_Flying_Side* of the current state, *st(s)*, is *Pilot_Flying* and the other side is observed to become the pilot flying side, setting *next_pfs* to *Not_Pilot_Flying*. In similar fashion, if the *Transfer_Switch* is pressed while this side is the

Not_Pilot_Flying side, transition 2 is taken setting *next_pfs* to *Pilot_Flying*. If neither transition is taken, *next_pfs* is left unchanged as *st(s)*. The next state is then composed from the computed value of *next_pfs* and current input values of the *Transfer_Switch* and *Other_Side_Pilot_Flying*.

```
    %------------------------------------------------------------------------
    % Next state function
    %------------------------------------------------------------------------
    next_state(s: State, ts:bool, ospf:bool) : State =
     LET
        next_pfs =
          %-------------------------------------------------------------------
          % Transition 1 - Other side becomes the pilot flying side
          %-------------------------------------------------------------------
          IF PilotFlying?(st(s)) AND rise_ospf(s, ospf)
             THEN NotPilotFlying
          %-------------------------------------------------------------------
          % Transition 2 - Transfer switch is pressed
          %-------------------------------------------------------------------
          ELSIF NotPilotFlying?(st(s)) AND rise_ts(s,ts)
             THEN PilotFlying
          %-------------------------------------------------------------------
          % No transition taken
          %-------------------------------------------------------------------
          ELSE
             st(s)
          ENDIF
       IN (# st:= next_pfs, pre_ts := ts, pre_ospf := ospf #)

    %------------------------------------------------------------------------
    % Pilot flying output
    %------------------------------------------------------------------------
    pilot_flying(s: State):bool = PilotFlying?(st(s))


  END Side_LLR
```

Figure 16 – Low-Level Requirements for the Synchronous Side (Part 2)

Finally, the *output* function extracts a Boolean value from the current state indicating if this side believes it is the pilot flying side.

Just as for the synchronous bus, we prove that the low-level requirements of Figure 15 and Figure 16 implement the high-level requirements of Figure 13 and Figure 14 by defining a theory interpretation of the high-level requirements based on the low-level requirements. This is shown in Figure 17.

```
Side_Interpretation[Primary_Side : bool] : THEORY

BEGIN

  %-------------------------------------------------------------------------
  % Import low-level requirements (LLR) for a side
  %-------------------------------------------------------------------------
  IMPORTING Side_LLR[Primary_Side]

  %-------------------------------------------------------------------------
  % Import high-level (HLR) requirements for a side and
  % define the LLR as an interpretation of the HLR
  %-------------------------------------------------------------------------
  IMPORTING Side_HLR[Primary_Side] {{
                   State         := Side_LLR.State,
                   Initial_State := Side_LLR.Initial_State,
                   next_state    := Side_LLR.next_state,
                   pre_TS        := Side_LLR.pre_TS,
                   pre_OSPF      := Side_LLR.pre_OSPF,
                   pilot_flying  := Side_LLR.pilot_flying
    }}

  END Side_Interpretation
```

Figure 17 – Theory Interpretation for the Synchronous Side

When this theory is typechecked using PVS, it generates nine TCCs, one for each axiom specified in the high-level requirements shown in Figure 14. These TCCS are easily proven using the *typecheck-prove* (M-x tcp) command of PVS, proving that the *Side_LLR* requirements are an implementation of the *Side_HLR* requirements and that the axioms of the *Side_HLR* requirements are consistent. Just as for the synchronous bus, the *Side_LLR* theory can be used as the low-level requirements for the side or a more detailed set of low-level requirements can be developed and proven to satisfy the *Side_HLR* requirements.

3.3.5 PVS Specification of the Synchronous Pilot Flying System

The PVS specification for the entire synchronous *Pilot_Flying_System* of Figure 7 is shown in Figure 18 and Figure 19.

```
Pilot_Flying_System : THEORY
BEGIN

  %-------------------------------------------------------------------------
  % Importing the sytem componenets
  %-------------------------------------------------------------------------
  IMPORTING Side_HLR [TRUE]  AS Left_Side;
  IMPORTING Bus_HLR  [TRUE]  AS LR_Bus;
  IMPORTING Side_HLR [FALSE] AS Right_Side;
  IMPORTING Bus_HLR  [FALSE] AS RL_Bus;

  %-------------------------------------------------------------------------
  % Defining the system state
  %-------------------------------------------------------------------------
  State : Type         = [# Left_Side  : Left_Side.State,
                            LR_Bus     : LR_Bus.State,
                            Right_Side : Right_Side.State,
                            RL_Bus     : RL_Bus.State,
                            pre_TS     : bool              #]

  %-------------------------------------------------------------------------
  % Defining the initial system state
  %-------------------------------------------------------------------------
  Initial_State: State = (# Left_Side  := Left_Side.Initial_State,
                            LR_Bus     := LR_Bus.Initial_State,
                            Right_Side := Right_Side.Initial_State,
                            RL_Bus     := RL_Bus.Initial_State,
                            pre_TS     := TRUE             #)
```

Figure 18 – PVS Specification of the Synchronous Pilot Flying System (Part 1)

The theory begins by importing the four components instantiated with the appropriate parameters. The *Left_Side* is instantiated with the *Primary_Side* parameter set to true and the *Right_Side* set to false. The *LR_Bus* is initiated with its initial output set to true to match the initial output of the *Left_Side* and the *RL_Bus* is instantiated with its output set to false to match the initial output *Right_Side*. The internal *State* of the system is defined to be a record structure containing the internal state of each of its four components and the previous value of the Transfer Switch. The *Initial_State* of the system is simply a record containing the initial state of each component with the previous value of the Transfer Switch initialized to true.

The *next_state* function for the system shown in Figure 19 merits careful examination. As shown in Figure 7, the *Left_Side* provides an input to the *LR_Bus* which provides an input to the

Right_Side which provides an input to the *RL_Bus* which provides an input to *Left_Side*. This circular dependency (also known as an algebraic loop) is only possible if some component depends only on the *previous* value of its predecessor. Like an Escher print (e.g. Relativity) [4], a circular dependency in which every component depends on the current value of its predecessor is an illusion that cannot be implemented.

```
%-------------------------------------------------------------------------
 % Next state function
 %-------------------------------------------------------------------------
 next_state(s: State, TS: bool): State =
   LET
       %-------------------------------------------------------------------
       % Compute next state of LR Bus and Right Side
       %-------------------------------------------------------------------
       C1      = Left_Side.Pilot_flying(Left_Side(s)),
       next_LR = LR_Bus.next_state(LR_Bus(s), C1),
       C2      = LR_Bus.output(next_LR),
       next_RS = Right_Side.next_state(Right_Side(s), TS, C2),

       %-------------------------------------------------------------------
       % Compute next state of RL Bus and Left Side
       %-------------------------------------------------------------------
       C3      = Right_Side. Pilot_flying(Right_Side(s)),
       next_RL = RL_Bus.next_state(RL_Bus(s), C3),
       C4      = RL_Bus.output(next_RL),
       next_LS = Left_Side.next_state(Left_Side(s), TS, C4)
   IN
      (# Left_Side  := next_LS,
         LR_Bus     := next_LR,
         Right_Side := next_RS,
         RL_Bus     := next_RL,
         pre_TS     := TS       #)

 %-------------------------------------------------------------------------
 % Outputs of the system
 %-------------------------------------------------------------------------
 Left_Pilot_Flying_Side(s: State) : bool = pilot_flying(Left_Side(s))

 Right_Pilot_Flying_Side(s: State): bool = pilot_flying(Right_Side(s))

 END Pilot_Flying_System
```

Figure 19 – PVS Specification of the Synchronous Pilot Flying System (Part 2)

We choose to resolve this cyclic dependency by having each bus read the current value of its input (i.e. the previous value form the side inputting to the bus) and have each side read the new value computed by its inputting bus. The *next_state* function of Figure 19 defines this precisely. It states that output of the *Left_Side* in the current state (C1) is used as an input in computing the

next state of the *LR_Bus*. The output of the *LR_Bus* after its state is updated (C2) is then used in computing the next state of the *Left_Side*. In similar fashion, the output of the *Right_Side* in the current state (C3) is used as an input in computing the next state of the *RL_Bus*. The output of the *RL_Bus* after its state is updated (C4) is then used in computing the next state of the *Right_Side*. The next state of the entire system is composed from the new state computed for each component and by updating the previous value of the Transfer Switch.

Finally, the theory defines the two outputs of the Pilot Flying system, the Boolean functions *Left_Pilot_Flying_Side* and *Right_Pilot_Flying_Side* indicating whether each side believes it is the pilot flying side.

3.3.6 Formal Verification of the Synchronous Pilot Flying Example

This section shows that the system architecture and high-level component requirements comply with the system requirements by proving that the five system requirements are satisfied by the system design and high-level requirements for each component. Stated informally, the five system requirements 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 first requirement R1 can be stated formally in PVS as shown in Figure 20.

```
s:  VAR Pilot_Flying_System.State

R1: THEOREM
    Left_Pilot_Flying_Side(s) or Right_Pilot_Flying_Side(s)
```

Figure 20 – Incorrect Statement of Synchronous Requirement R1 in PVS

Unfortunately, this property cannot be proven. This is because the type *State* does not explicitly exclude system states where neither side is the pilot flying side. However, for our system the only states of interest are those that can be reached in some number of steps from the *Initial State*. The step in proving requirement R1 is restating it so that it only applies to reachable states. The first part of the *Pilot_Flying_System_Requirements* theory shown in Figure 21 defines what it means for a state to be reachable.

```
Pilot_Flying_System_Requirements: THEORY

  BEGIN

    IMPORTING Pilot_Flying_System

    s:  VAR Pilot_Flying_System.State
    ts: VAR bool

    %-------------------------------------------------------------------------
    % Definition of a reachable state
    %-------------------------------------------------------------------------
    Reachable_State(s): INDUCTIVE bool =
        s = Initial_State OR
            (EXISTS (r: Pilot_Flying_System.State, t: bool) :
                        Reachable_State(r) AND s = next_state(r,t))

    %-------------------------------------------------------------------------
    % Definition of a valid state
    %-------------------------------------------------------------------------
    Pre_TS_Consistency(s): bool =
        pre_TS(Left_Side(s))  = pre_TS(s) and
        pre_TS(Right_Side(s)) = pre_TS(s)

    Pre_OSPF_Consistency(s): bool =
        pre_OSPF(Right_Side(s)) = LR_Bus.output(LR_Bus(s)) and
        pre_OSPF(Left_Side(s))  = RL_Bus.output(RL_Bus(s))

    At_Least_One_Side_Flying(s): bool =
        Left_Pilot_Flying_Side(s) OR Right_Pilot_Flying_Side(s)

    Buses_Differ_When_Sides_Same(s): bool =
        pilot_flying(Left_Side(s)) = pilot_flying(Right_Side(s)) =>
            LR_Bus.output(LR_Bus(s)) /= RL_Bus.output(RL_Bus(s))

    Valid_State(s): bool =
        At_Least_One_Side_Flying(s) AND
        Pre_TS_Consistency(s) AND
        Pre_OSPF_Consistency(s) AND
        Buses_Differ_When_Sides_Same(s)

    %-------------------------------------------------------------------------
    % Proof that every reachable state is a valid state
    %-------------------------------------------------------------------------
    Reachable_States_Valid: THEOREM
        Reachable_State(s) => Valid_State(s)
```

Figure 21 – Synchronous Pilot Flying System Requirements (Part 1)

The predicate *Reachable_States* in Figure 21 inductively defines a reachable state to be either the *Initial_State* or any state that can be reached through application of the *next_state* function from a reachable state. With this definition, we can restate the theorem for requirement R1 as shown in Figure 22.

```
R1: THEOREM
    Reachable_State(s) =>
       Left_Pilot_Flying_Side(s) or Right_Pilot_Flying_Side(s)
```

Figure 22 – Correct Statement of Requirement R1 in PVS

This theorem can be proven, but the proof is complicated and it needs to be repeated in proving the other requirements. To keep our proofs manageable, we first shall define a set of predicates describing the relationships between the system components that the system will maintain during its execution and then prove that those predicates are true of every reachable state.

The *Pre_TS_Consistency* predicate of Figure 21 states that the previous value of the Transfer Switch stored in the left and right sides must be the same as that stored in the system state. The *Pre_OSPF_Consistency* predicate states that the previous value of the other side's pilot flying indication stored in each side (*pre_OSPF*) agrees with the output of the bus that passed it that value. The *At_Least_One_Side_Flying* predicate states that at least one side is the pilot flying side. The *Buses_Differ_When_Both_Sides_Flying* predicate states that when both sides are the pilot flying side, the buses must contain different Boolean values. These predicates are collected together as a single conjunction in the *Valid_State* predicate shown in Figure 21.

Since *Valid_State* contains many of the relationships between system components necessary to prove the system requirements and since it is not defined recursively, it is much easier to use in the proofs than *Reachable_States*. Unfortunately, development of the *Valid_State* predicate is not always obvious. Some of the constraints, such as *At_Least_One_Side_Flying* and *Pre_TS_Consistency* are intuitive. The others were developed by trying to prove the system requirements and carefully studying the proof obligations produced by PVS. For example, the *Buses_Differ_When_Both_Sides_Flying* predicate is not obvious and was added in the process of proving requirements R1 through R5. However, in retrospect it is clear that if it were not true, the system would either deadlock with both sides as the pilot flying side or immediately transition to a state where neither side is the pilot flying side.

The next step is to prove that every reachable state is also a valid state. The *Reachable_States_Valid* theorem is stated at the bottom of Figure 21 and its proof is shown in Figure 23.

```
;;; Proof Reachable_States_Valid-2 for formula
Pilot_Flying_System_Requirements.Reachable_States_Valid
;;; developed with shostak decision procedures
(""
 (auto-rewrite-theories "Left_Side" "LR_Bus" "Right_Side" "RL_Bus")
 (rule-induct "Reachable_State")
 (skosimp)
 (case "s!2 = Initial_State")
 (("1" (grind))
  ("2"
   (assert)
   (hide 1)
   (grind)
   (("1"
     (case "Right_Side.pilot_flying(Right_Side(r!1))")
     (("1" (grind)) ("2" (grind))))
    ("2"
     (case "Right_Side.pilot_flying(Right_Side(r!1))")
     (("1" (grind)) ("2" (grind))))
    ("3"
     (case "Left_Side.pilot_flying(Left_Side(r!1))")
     (("1" (grind)) ("2" (grind))))
    ("4"
     (case "Left_Side.pilot_flying(Left_Side(r!1))")
     (("1" (grind)) ("2" (grind)))))))))
```

Figure 23 – PVS Proof of Synchronous Reachable States Valid Theorem

The *auto-rewrite-theories* command installs the axioms stated in the high-level requirements for each side and each bus as automatic rewrite rules. The *rule-induct* command adds the inductive definition of *Reachable_States* to the proof tree as the rule shown in Figure 24.

```
FORALL (s):
        (s = Initial_State OR
          (EXISTS (r: Pilot_Flying_System.State, t: bool):
             Valid_State(r) AND s = next_state(r, t)))
         IMPLIES Valid_State(s)
```

Figure 24 – Inductive Sequent for Reachable States

The *skosimp* command replaces the universal quantification over *s* with an unspecified constant *s!2* and simplifies the proof obligation. The *case* command splits the proof into two sub-goals. The first sub-goal, in which *s!2* is the *Initial_State*, is easily discharged with the PVS *grind* command. The second sub-goal, in which *s!2* is any state other than the *Initial_State*, is reduced through application of the *assert*, *hide*, and *grind* commands to four sub-goals. Each of these can be discharged by case splitting on the value of the *Pilot_Flying* output of either the left or right

side of state *r!1* (introduced by the *grind* command to instantiate the existential quantification over *r*) followed by application of *grind* to each of the two resulting sub-goals.

With the proof that every reachable state is a valid state, the proof of requirements R1 through R5 is straightforward. The formal statement of requirements R1 and R2 are shown in Figure 25.

To make the requirements more readable, we define two predicates over the system state. The predicate *switching_sides* identifies the system states in which the system is in the process of changing the pilot flying side, i.e. where one side has become the pilot flying side but that change has not reached the other side. The predicate *pressed* provides a convenient way of identifying a rising edge of the Transfer Switch.

```
%-----------------------------------------------------------------------
% The Transfer Switch is pressed in state s when its value rises.
%-----------------------------------------------------------------------
pressed(ts, s) : bool = not pre_TS(s) and ts

%-----------------------------------------------------------------------
% 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)
```

Figure 25 – Synchronous Pilot Flying System Requirements (Part 2)

The proof of requirement R1 is shown in Figure 26.

```
;;; Proof R1-1 for formula Pilot_Flying_System_Requirements.R1
;;; developed with shostak decision procedures
("" (use "Reachable_States_Valid") (grind))
```

Figure 26 – PVS Proof of Requirement R1 for Synchronous Pilot Flying System

The proof of R1 is immediate since all of the real work was done in proving that every reachable state is a valid state. The proof invokes the *Reachable_States_Valid* theorem with the PVS *use* command followed by a PVS *grind* command.

Trying to prove requirement R2 (at most one side shall be the pilot flying side) reveals that this requirement cannot hold for all reachable states. In particular, in a state in which one side has just become the pilot flying side but this information has not been transmitted to the other side, both sides will be the pilot flying side for one step. However, a safety analysis shows that it is acceptable to have both sides be the pilot flying side for a single step and we modify R2 to be required for only reachable states while the system is not switching sides. Requirement R2 can be proven with the same proof shown in Figure 26 used to prove requirement R1.

```
  %-----------------------------------------------------------------------
  % R3. Pressing the Transfer Switch shall always change pilot
  %     pilot flying side.
  %-----------------------------------------------------------------------
  R3a: THEOREM
      Reachable_State(s)  =>
         (not Left_Pilot_Flying_Side(s) and pressed(ts,s) =>
              Left_Pilot_Flying_Side(next_state(s,ts)))

  R3b: THEOREM
       Reachable_State(s) =>
          (not Right_Pilot_Flying_Side(s) and pressed(ts,s) =>
               Right_Pilot_Flying_Side(next_state(s,ts)))

  %-----------------------------------------------------------------------
  % R4. The system shall start with the Primary Side as the pilot
  %     flying side.
  %-----------------------------------------------------------------------
  R4: THEOREM
      Left_Pilot_Flying_Side(Initial_State)

  %-----------------------------------------------------------------------
  % R5. The system shall not change the pilot flying side if it is
  %     not switching sides and the Transfer Switch is not pressed.
  %-----------------------------------------------------------------------
  R5a: THEOREM
      Reachable_State(s) AND
         NOT switching_sides(s) AND NOT pressed(ts, s) =>
             (Left_Pilot_Flying_Side(next_state(s, ts)) =
                 Left_Pilot_Flying_Side(s))

  R5b: THEOREM
     Reachable_State(s) AND
        NOT switching_sides(s) AND NOT pressed(ts, s) =>
            (Right_Pilot_Flying_Side(next_state(s, ts)) =
                 Right_Pilot_Flying_Side(s))

END Pilot_Flying_System_Requirements
```

Figure 27 – PVS Specification of the Pilot Flying System Requirements (Part 3)

Requirement R3 (pressing the Transfer Switch shall always change the pilot flying side) is broken down into two smaller requirements. R3a states that for a reachable system state, if the left side is not the pilot flying side and the Transfer Switch is pressed, then the left side will become the pilot flying side in the next state. R3b states the same property for the right side. Both of these requirements can be proven with the proof shown in Figure 28.

```
;;; Proof R3a-1 for formula Pilot_Flying_System_Requirements.R3a
;;; developed with shostak decision procedures
(""
 (use "Reachable_States_Valid")
 (auto-rewrite-theories "Left_Side" "LR_Bus" "Right_Side" "RL_Bus")
 (grind))
```

Figure 28 – PVS Proof of Requirement R3a/b for Synchronous Pilot Flying System

This proof first invokes the *Reachable_States_Valid* theorem with the PVS *use* command, then installs the axioms stated in the high-level requirements for each side and each bus as automatic rewrite rules using the PVS *auto-rewrite-theories* command, and finally completes the proof with a PVS *grind* command.

The proof of requirement R4 (the system shall start with the Primary Side as the pilot flying side) is equally straightforward. It is discharged by using the *auto-rewrite-theories* command to install the axioms for the high-level requirements as automatic rewrite rules followed by a *grind* command.

Finally, the proof of requirement R5 (the system shall not change the pilot flying side unless the Transfer Switch is pressed) requires a slight modification. First, it is broken down into two smaller requirements, R5a and R5b, as was done for requirement R3. Trying to prove these reveal that they only hold if the system is not already in the process of switching sides. We revise the original requirement to include this caveat as shown in Figure 27. Both requirements can then be proven using the same proof used to prove requirements R3a and R3b (Figure 28).

While this completes the proof of the original five requirements, the current definition of *Valid_State* may not be sufficient to prove additional useful properties about the Pilot Flying system. This is because while we proved that every reachable state is a valid state, we have not proven that every valid state is reachable. In other words, there may be states that are valid but that cannot be reached. In fact this is exactly the situation. The definition of *Valid_State* does not include all the relationships between system components that the system will maintain, ensuring that there are valid states that are not reachable. This is demonstrated in the theory *Pilot_Flying_System_Requirements2* shown in Figure 29.

```
Pilot_Flying_System_Requirements2: THEORY

  BEGIN

    IMPORTING Pilot_Flying_System_Requirements

    s:  VAR Pilot_Flying_System.State
    ts: VAR bool

    %-------------------------------------------------------------------
    % Enhanced definition of a valid state
    %-------------------------------------------------------------------
    At_Least_One_Bus_High(s): bool =
       output(LR_Bus(s)) OR output(RL_Bus(s))

    Quiescent(s): bool =
        (output(LR_Bus(s)) /= output(RL_Bus(s)) AND
         pilot_flying(Left_Side(s)) /= pilot_flying(Right_Side(s))) =>
        (output(LR_Bus(s)) = pilot_flying(Left_Side(s)) AND
         output(RL_Bus(s)) = pilot_flying(Right_Side(s)))

    Valid_State2(s) : bool =
        Valid_State(s) AND
        At_Least_One_Bus_High(s) AND
        Quiescent(s)

    %-------------------------------------------------------------------
    % Every reachable state is a valid (2) state
    %-------------------------------------------------------------------
    Reachable_States_Valid2: THEOREM
        Reachable_State(s) => Valid_State2(s)

    %-------------------------------------------------------------------
    % The system only switches sides for one step
    %-------------------------------------------------------------------
    Switching_Transient: THEOREM
        Reachable_State(s) AND switching_sides(s) =>
           NOT switching_sides(next_state(s, ts))

  END Pilot_Flying_System_Requirements2
```

Figure 29 – Pilot Flying System Requirements 2

This theory imports the *Pilot_Flying_System_Requirements* theory so that it contains the current definition of *Valid_State*. It also includes the *Switching_Transient* theorem which states that if the system is switching sides in one state, it is not switching sides in the next state. This theorem cannot be easily proven using the definition of *Valid_State*, but it can be proven using the definition *Valid_State2* that adds two additional constraints to *Valid_State*. The first, *At_Least_One_Bus_High*, states that the output of at least one of the two buses is always true. The second, *Quiescent*, states that in a quiescent state where the output of both sides and both

buses differ, the output of the left bus will be equal to the output of the left side and the output of the right bus will be equal to the output of the right side.

It is easily proven that all reachable states are *Valid_State2* states using a proof similar to that of Figure 23. The *Switching_Transient* theorem is then easily proven using the proof of Figure 30 which invokes *Reachable_States_Valid2* rather than *Reachable_States_Valid*.

```
;;; Proof Switching_Transient-2 for formula
Pilot_Flying_System_Requirements2.Switching_Transient
;;; developed with shostak decision procedures
(""
 (auto-rewrite-theories "Left_Side" "LR_Bus" "Right_Side" "RL_Bus")
 (use "Reachable_States_Valid2")
 (grind))
```

Figure 30 – PVS Proof of Switching Transient

3.4 The Asynchronous Pilot Flying Example

Designing and verifying the Pilot Flying System is considerably more difficult in the asynchronous case when the components are not driven by a single master clock. Values may be missed entirely by a component if they arrive while it is not executing, leading to race and deadlock conditions. If no assumptions are made about the individual component clocks, the Pilot Flying System can be implemented correctly only through the use of a hand-shaking protocol. This section describes how the fully asynchronous case can be specified and verified in PVS.

To model asynchrony, we introduce for each component a single Boolean valued clock signal. When its clock is true, a component will take a step just as in the synchronous case. When its clock is false, the component makes no change to its internal state or outputs. While this model assumes an underlying discrete model of time where each component clock can tick only when the global clock ticks, we make no other assumptions about the clocks. The global clock may tick at any rate and the component clocks may tick or not tick at any time the global clock ticks. This model of time is sufficient to generate the conditions we are interested in verifying. The top level diagram for the asynchronous Pilot Flying example is shown in Figure 31.

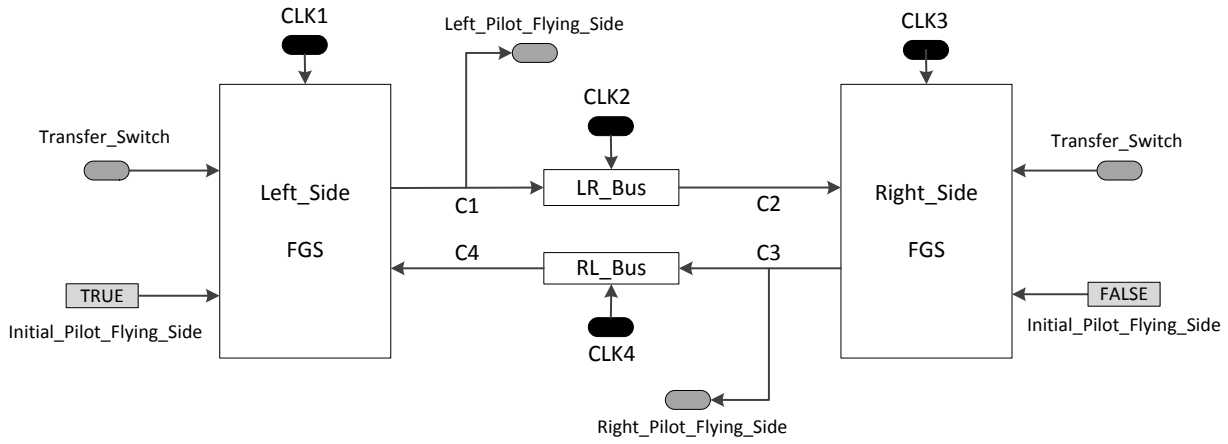

Figure 31 – Asynchronous Pilot Flying System

The asynchronous system diagram differs from the synchronous diagram only in the addition of the four clocks, *CLK1* through *CLK4*. However, there are several other changes needed to the underlying components to produce a correct implementation. These are described in the following sections.

3.4.1 Specification of the Asynchronous Bus Messages

To implement the hand-shaking protocol necessary in the asynchronous case, each FGS will generate both its pilot flying status and a Boolean acknowledgement. Since these will be produced and conveyed across the bus together we define a single *Message* type as shown Figure 32. A bus *Message* is a record type with two fields, *pfs* for the pilot flying status and *ack* for the acknowledgement. We also define a *Msg* constructor function which constructs a new message from values for *pfs* and *ack*.

```
Message: THEORY
BEGIN

    %-------------------------------------------------------------------
    % A bus message consists of the Pilot Flying Status and an Ack
    %-------------------------------------------------------------------
    Message: TYPE = [# pfs: bool, ack :bool #]

    %-------------------------------------------------------------------
    % Message constructor
    %-------------------------------------------------------------------
    Msg(pfs, ack : bool) : Message = (# pfs := pfs, ack := ack #)

 END Message
```

Figure 32 – PVS Specification of a Bus Message

3.4.2 High-Level Requirements for the Asynchronous Bus

The PVS theory for the high-level requirements for the asynchronous bus is shown in Figure 33.

```
Bus_HLR[INIT_PFS: bool, INIT_ACK: bool]: THEORY
BEGIN

    Importing Message

    %-----------------------------------------------------------------------
    % Define the state and initial state
    %-----------------------------------------------------------------------
    State: TYPE+

    Initial_State: State

    %-----------------------------------------------------------------------
    % Next state function
    %-----------------------------------------------------------------------
    next_state: [State, bool, Message -> State]

    %-----------------------------------------------------------------------
    % Output of the bus
    %-----------------------------------------------------------------------
    output: [State -> Message]

    %-----------------------------------------------------------------------
    % High level requirements
    %-----------------------------------------------------------------------
    HLR1: AXIOM
      output(Initial_State) = Msg(INIT_PFS, INIT_ACK)

    HLR2: AXIOM
      forall (s: State, clk: bool, input: Message) :
          output(next_state(s, clk, input)) =
              IF NOT clk THEN output(s) ELSE input ENDIF

 END Bus_HLR
```

Figure 33 – High-Level Requirements for the Asynchronous Bus

The theory is parameterized with the *INIT_PFS* and *INIT_ACK* values specifying the initial output values of the bus. Just as with the high-level requirements for the synchronous bus, uninterpreted values are provided for the type *State*, the *Initial_State*, and the *next_state* and *output* functions. Two axioms, HR1 and HR2, specify the high level requirements of the bus. The other major change from the synchronous case is that HR2 specifies that the output of the bus changes only when its clock value is true.

3.4.3 Low-Level Requirements for the Asynchronous Bus

The low-level requirements for the asynchronous bus are shown in Figure 34.

```
Bus_LLR[INIT_PFS: bool, INIT_ACK: bool]: THEORY
BEGIN

    Importing Message

    %-------------------------------------------------------------------
    % Define the state and initial state
    %-------------------------------------------------------------------
    State: TYPE+ = Message

    Initial_State: State = Msg(INIT_PFS, INIT_ACK)

    %-------------------------------------------------------------------
    % Next state function
    %-------------------------------------------------------------------
    next_state(s: State, clk: bool, input: Message): State =
       IF NOT clk THEN s ELSE input ENDIF

    %-------------------------------------------------------------------
    % Output of the bus
    %-------------------------------------------------------------------
    output(s: State): Message = s

 END Bus_LLR
```

Figure 34 – Low-Level Requirements for the Asynchronous Bus

In the low-level requirements, a concrete interpretation has been assigned to each uninterpreted type, constant, and function of the high-level requirements. The type *State* has been defined to be a record structure of the type *Message*. The *Initial_State* is a message constructed from the theory parameters. The *next_state* function is defined to return the current state when its clock is false and its *input* message when its clock is true. The *output* function returns the current state of the bus.

Just as with the synchronous bus, we demonstrate that the axioms of the high-level bus requirements are consistent with a theory interpretation as shown in Figure 35.

```
Bus_Interpretation[INIT_PFS: bool, INIT_ACK: bool]: THEORY

BEGIN

  %-------------------------------------------------------------------
  % Import low-level requirements (LLR) for a bus
  %-------------------------------------------------------------------
  IMPORTING Bus_LLR[INIT_PFS, INIT_ACK]

  %-------------------------------------------------------------------
  % Import the high-level (HLR) requirements for a bus and
  % define the LLR as an interpretation of the HLR
  %-------------------------------------------------------------------
  IMPORTING Bus_HLR[INIT_PFS, INIT_ACK] {{
                    State         := Bus_LLR.State,
                    Initial_State := Bus_LLR.Initial_State,
                    next_state    := Bus_LLR.next_state,
                    output        := Bus_LLR.output
    }}

  END Bus_Interpretation
```

Figure 35 – Theory Interpretation for Asynchronous Bus

Type checking this theory generates two TCCs, both of which are easily proven using the *typecheck-prove* (M-x tcp) command of PVS, proving that the *Bus_LLR* requirements are an implementation of the *Bus_HLR* requirements and that the axioms of the *Bus_HLR* theory are consistent.

3.4.4 High-Level Requirements for the Asynchronous FGS Side

The PVS specification for an FGS *Side* also needs to be changed to input and output values of type *Message* rather than just a simple Boolean. Since we make no assumptions about the component clocks, a correct implementation of the synchronization logic requires a hand-shaking protocol as illustrated in Figure 36.

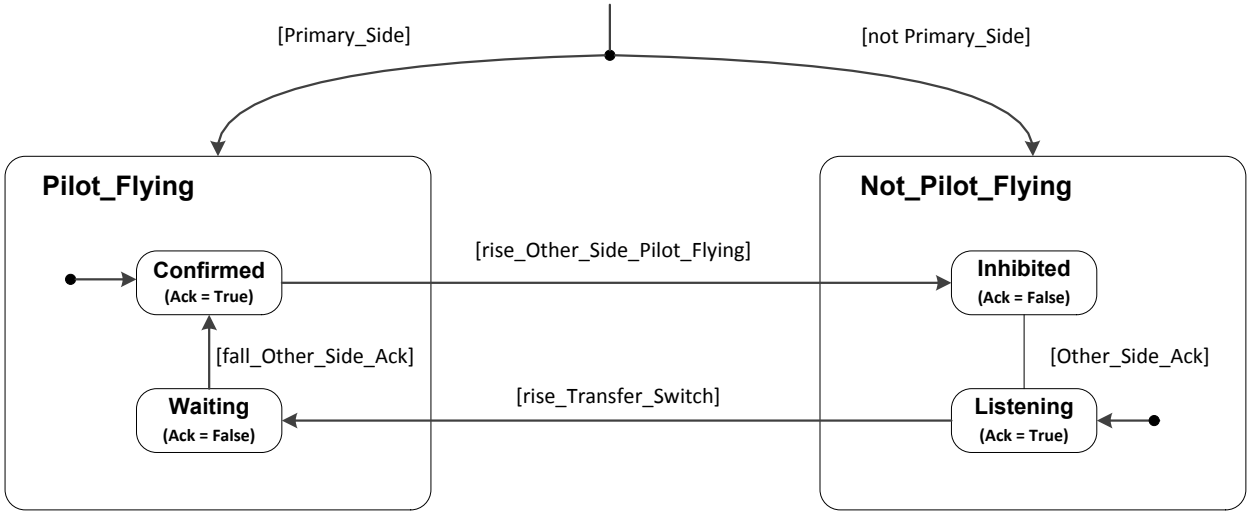

Figure 36 – Asynchronous Pilot Flying Side Logic

The *Ack* value is used to communicate to the other side when a side has reached a stable state. The *Primary_Side* starts in the *Confirmed* sub-state of the *Pilot_Flying* state with its *Ack* set to true. The other side starts in the *Listening* sub-state of the *Not_Pilot_Flying* state with its *Ack* also set to true. When the *Primary_Side* sees the other side become the pilot flying side, it transitions to the *Inhibited* sub-state of the *Not_Pilot_Flying* state and sets its *Ack* to false. While in the *Inhibited* state, a side does not respond to the flight crew pressing the Transfer Switch.[4] The *Not_Pilot_Flying_Side* resumes listening for the Transfer Switch when it receives an *Ack* from the other side indicating that the other side has reached the *Confirmed* state. When the Transfer Switch is pressed, the *Not_Pilot_Flying* side transitions to the *Waiting* sub-state of the *Pilot_Flying* state and remains in this sub-state until it sees the other side's *Ack* fall, indicating that the other side has yielded control. The PVS specification for the high-level requirements for an asynchronous FGS side is shown in Figure 37, Figure 44, and Figure 45.

[4] In an actual system, it could be remedied by ensuring the Transfer Switch remains high longer than the time needed for the *Ack* to be received from the new pilot flying side.

```
Side_HLR[Primary_Side : bool] : THEORY

BEGIN

   IMPORTING Message

    %-------------------------------------------------------------------
    % State defined an an uninterpreted, non-empty type
    %-------------------------------------------------------------------
    State: TYPE+

    Initial_State: State

    %-------------------------------------------------------------------
    % Next state defined as an uninterpreted function
    %-------------------------------------------------------------------
    next_state: [State, bool, bool, Message-> State]

    %-------------------------------------------------------------------
    % Extractor functions for this side
    %-------------------------------------------------------------------
    pre_TS : [State -> bool]
    pre_MSG: [State -> Message]

    %-------------------------------------------------------------------
    % Output is a message containing the pilot flying status and ack
    %-------------------------------------------------------------------
    output : [State -> Message]

    %-------------------------------------------------------------------
    % Auxiliary functions for stating properties
    %-------------------------------------------------------------------
    rise_ts  (s: State, ts:bool  ): bool = NOT pre_TS(s) AND ts
    rise_ospf(s: State, m:Message): bool = NOT pfs(pre_MSG(s)) AND pfs(m)
    fall_ack (s: State, m:Message): bool = ack(pre_MSG(s)) AND NOT ack(m)

    confirmed(s: State): bool =     pfs(output(s)) AND ack(output(s))
    inhibited(s: State): bool = NOT pfs(output(s)) AND NOT ack(output(s))
    listening(s: State): bool = NOT pfs(output(s)) AND     ack(output(s))
    waiting  (s: State): bool =     pfs(output(s)) and NOT ack(output(s))
```

Figure 37 – High-Level Requirements for the Asynchronous Side (Part 1)

The *Side_HLR* theory is parameterized with whether this side is the Primary Side. Uninterpreted values are provided for the type *Side*, the constant *Initial_State*, the *next_state* function, and extractor functions for the previous value of the Transfer Switch *(pre_TS*), the previous value of the other side's message, (*pre_MSG*), and the *output* of this side. We also specify several convenient auxiliary functions. The functions *rise_ts*, *rise_ospf*, and *fall_ack* define precisely how a rising edge of the Transfer Switch, a rising edge of the other side's pilot flying value, and

a falling edge of the other side's ack are to be identified. The functions *confirmed*, *inhibited*, *listening*, and *waiting* provide a convenient way identifying the sub-state of a side.

```
%-----------------------------------------------------------------------
% High level requirements
%-----------------------------------------------------------------------
HLR1: AXIOM
   output(Initial_State) = Msg(Primary_Side, TRUE)

HLR2: AXIOM
   pre_TS(Initial_State)

HLR3: AXIOM
   pre_MSG(Initial_State) = Msg(NOT Primary_Side, TRUE)
```

Figure 38 – High-Level Requirements for the Asynchronous Side (Part 2)

Axioms HLR1 through HLR3 in Figure 38 define the result of applying each of the three extractors to the *Initial_State* constructor. The *output* of the *Initial_State* is set to either *confirmed* or *listening* depending on the *Primary_Side* theory parameter. The stored value of the Transfer Switch is initialized to true so that it cannot be pressed in the initial step. The stored value of the other side's message is initialized to agree with the initial output of the bus from the other side

```
HLR4: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m:Message) :
      NOT clk => output(next_state(s, clk, ts, m)) = output(s)

HLR5: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND confirmed(s) AND rise_ospf(s, m) =>
         inhibited(next_state(s, clk, ts, m))

HLR6: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND confirmed(s) AND NOT rise_ospf(s, m) =>
         confirmed(next_state(s, clk, ts, m))

HLR7: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND inhibited(s) AND ack(m) =>
         listening(next_state(s, clk, ts, m))

HLR8: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND inhibited(s) AND NOT ack(m) =>
         inhibited(next_state(s, clk, ts, m))

HLR9: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND listening(s) AND rise_ts(s, ts) =>
         waiting(next_state(s, clk, ts, m))

HLR10: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND listening(s) AND NOT rise_ts(s, ts) =>
         listening(next_state(s, clk, ts, m))

HLR11: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND waiting(s) AND fall_ack(s, m) =>
         confirmed(next_state(s, clk, ts, m))

HLR12: AXIOM
   FORALL (s:State, clk:bool, ts:bool, m: Message) :
      clk AND waiting(s) AND NOT fall_ack(s, m) =>
         waiting(next_state(s, clk, ts, m))
```

Figure 39 – High-Level Requirements for the Asynchronous Side (Part 3)

HLR4 through HLR12 in Figure 39 define the result of applying the *output* extractor to the *next_state* constructor in accordance with the protocol of Figure 36 (recall that *confirmed*, *inhibited*, *listening*, and *waiting* are defined in terms of the *output* extractor). Note that HLR4 asserts that if the clock for a side is false, no change occurs in the output of the side.

```
    HLR13: AXIOM
       FORALL (s:State, clk:bool, ts: bool, m: Message) :
          NOT clk => pre_TS(next_state(s, clk, ts, m)) = pre_TS(s)

    HLR14: AXIOM
       FORALL (s:State, clk:bool, ts: bool, m: Message) :
          clk => pre_TS(next_state(s, clk, ts, m)) = ts

    HLR15: AXIOM
       FORALL (s:State, clk:bool, ts: bool, m: Message) :
          NOT clk => pre_MSG(next_state(s, clk, ts, m)) = pre_MSG(s)

    HLR16: AXIOM
       forall (s:State, clk:bool, ts: bool, m: Message) :
          clk => pre_MSG(next_state(s, clk, ts, m)) = m

END Side_HLR
```

Figure 40 – High-Level Requirements for the Asynchronous Side (Part 4)

Finally, HLR13 through HLR16 define the result of applying the extractors *pre_TS* and *pre_MSG* to the constructor *next_state*. Again, if the clock for the side if false, the value returned by each extractors is not changed by the *next_state* function.

3.4.5 Low-Level Requirements for the Asynchronous FGS Side

The PVS theory *Side_LLR* shown in Figure 41 and Figure 42 defines a constructive implementation of a synchronous *Side* by providing a concrete interpretation for each type, constant, and function. The four possible values of the state machine of Figure 36, *Confirmed*, *Inhibited*, *Listenin*g, and *Waiting*, are defined as the PVS enumeration type *Pilot_Flying_Side*. The initial value of the state machine is defined to be either *Confirmed* or *Listening* depending on the *Primary_Side* theory parameter.

The *State* of a side is defined as a PVS record structure consisting of the three fields – the current value of the *Pilot_Flying_Side* state machine (*st*), the previous value of the Transfer Switch (*pre_ts*), and the previous value of the bus message from the other side (*pre_msg*). The *Initial_State* constant consists of a record in which the *st* component is set the *Initial_Pilot_Flying_Side*, the *pre_ts* component is initialized to true to ensure the Transfer Switch cannot be pressed in the initial state, and the *pre_msg* component is set to agree with the message provided by the bus from the other side.

```
Side_LLR[Primary_Side : bool] : THEORY
BEGIN

    IMPORTING Message

    %-------------------------------------------------------------------
    % Pilot flying state machine values
    %-------------------------------------------------------------------
    Pilot_Flying_Side: TYPE = {Confirmed, Inhibited, Listening, Waiting}

    Initial_Pilot_Flying_Side : Pilot_Flying_Side =
      IF Primary_Side THEN Confirmed ELSE Listening ENDIF

    %-------------------------------------------------------------------
    % Definition of state and initial state for this side
    %-------------------------------------------------------------------
    State: TYPE          = [# st      : Pilot_Flying_Side,
                            pre_ts  : bool,
                            pre_msg : Message #]

    Initial_State: State = (# st      := Initial_Pilot_Flying_Side,
                            pre_ts  := TRUE,
                            pre_msg := Msg(NOT Primary_Side, TRUE) #)

    %-------------------------------------------------------------------
    % Extractor functions for this side
    %-------------------------------------------------------------------
    pre_TS (s: State): bool    = pre_ts(s)
    pre_MSG(s: State): Message = pre_msg(s)

    %-------------------------------------------------------------------
    % Auxiliary functions for defining next state function
    %-------------------------------------------------------------------
    rise_ospf(s:State, m:Message): bool = NOT pfs(pre_MSG(s)) AND pfs(m)
    rise_ts  (s:State, ts:bool  ): bool = NOT pre_TS(s) AND ts
    fall_ack (s:State, m:Message): bool = ack(pre_MSG(s)) AND NOT ack(m)
```

Figure 41 – Low-Level Requirements for the Asynchronous Side (Part 1)

To maintain consistency with the high-level requirements, extractor functions *pre_TS* and *pre_MSG* are defined. Auxiliary functions are also defined to simplify specification of the *next_state* function.

```
  %-------------------------------------------------------------------
  % Next state function
  %-------------------------------------------------------------------
  next_state(s: State, clk: bool, ts: bool, m: Message): State =
    %-----------------------------------------------------------------
    % No change when clock is false
    %-----------------------------------------------------------------
    IF (NOT clk) THEN
        s
    ELSE LET
       next_st =
        %-----------------------------------------------------------
        % Transition 1 - Rise of Other Side Pilot Flying
        %-----------------------------------------------------------
        IF (Confirmed?(st(s)) AND rise_ospf(s, m))
          THEN Inhibited
        %-----------------------------------------------------------
        % Transition 2 - Rise of Other Side Pilot Flying
        %-----------------------------------------------------------
        ELSIF (Inhibited?(st(s)) AND ack(m))
          THEN Listening
        %-----------------------------------------------------------
        % Transition 3 - Rise of Transfer Switch
        %-----------------------------------------------------------
        ELSIF (Listening?(st(s)) AND rise_ts(s, ts))
          THEN Waiting
        %-----------------------------------------------------------
        % Transition 4 - Fall of Other Side Ack
        %-----------------------------------------------------------
        ELSIF (Waiting?(st(s)) AND fall_ack(s, m))
          THEN Confirmed
        %-----------------------------------------------------------
        % No transition taken
        %-----------------------------------------------------------
        ELSE
         st(s)
        ENDIF
      IN
        (# st := next_st, pre_ts := ts, pre_msg := m #)
    ENDIF

  %-------------------------------------------------------------------
  % Output is a message containing the pilot flying status and the ack
  %-------------------------------------------------------------------
  output(s:State) : Message =
           (# pfs := Confirmed?(st(s)) OR Waiting?(st(s)),
              ack := Confirmed?(st(s)) OR Listening?(st(s)) #)

END Side_LLR
```

Figure 42 – Low-Level Requirements for the Asynchronous Side (Part 2)

The *next_state* function (Figure 42) computes each of the three components of the next system state from the current state and inputs, where the inputs consist of the side's clock, the Transfer

Switch, and the bus message from the other side. If the side's clock is false, no change occurs in the side's state. Otherwise, the next sub-state is computed in accordance with the logic of Figure 36 and composed with the current value of the Transfer Switch and the message from the other side.

To show that the low-level requirements of Figure 41 and Figure 42 implement the high-level requirements of Figure 37 through Figure 40, we define a theory interpretation as shown in Figure 43.

```
Side_Interpretation[Primary_Side: bool]: THEORY

BEGIN

  %-------------------------------------------------------------------
  % Import the low-level requirements (LLR) for a side
  %-------------------------------------------------------------------
  IMPORTING Side_LLR[Primary_Side]

  %-------------------------------------------------------------------
  % Import the high-level (HLR) requirements for a side and
  % define the LLR as an interpretation of the HLR
  %-------------------------------------------------------------------
  IMPORTING Side_HLR[Primary_Side] {{
                    State         := Side_LLR.State,
                    Initial_State := Side_LLR.Initial_State,
                    next_state    := Side_LLR.next_state,
                    pre_TS        := Side_LLR.pre_TS,
                    pre_MSG       := Side_LLR.pre_MSG,
                    output        := Side_LLR.output
            }}

  END Side_Interpretation
```

Figure 43 – Theory Interpretation for the Asynchronous Side

Type checking this theory with PVS generates 16 TCCs, one for each axiom specified in the high-level requirements of Figure 37 through Figure 40. These TCCS are easily proven using the *typecheck-prove* (M-x tcp) command of PVS, proving that the *Side_LLR* requirements are an implementation of the *Side_HLR* requirements and that the axioms of the *Side_HLR* requirements are consistent.

3.4.6 PVS Specification of the Asynchronous Pilot Flying Example

The PVS specification for the entire asynchronous *Pilot_Flying_System* depicted in Figure 31 is shown in Figure 44 and Figure 45.

```
Pilot_Flying_System: THEORY
BEGIN

  %-------------------------------------------------------------------
  % Import the system components
  %-------------------------------------------------------------------
  IMPORTING Side_HLR[TRUE]        AS Left_Side;
  IMPORTING Bus_HLR [TRUE, TRUE]  AS LR_Bus;
  IMPORTING Side_HLR[FALSE]       AS Right_Side;
  IMPORTING Bus_HLR [FALSE, TRUE] AS RL_Bus;

  %-------------------------------------------------------------------
  % Define the system state
  %-------------------------------------------------------------------
  State: Type          = [# Left_Side  :  Left_Side.State,
                            LR_Bus     :  LR_Bus.State,
                            Right_Side :  Right_Side.State,
                            RL_Bus     :  RL_Bus.State,
                            pre_TS     :  bool                #]

  %-------------------------------------------------------------------
  % Define the initial system state
  %-------------------------------------------------------------------
  Initial_State: State = (# Left_Side  := Left_Side.Initial_State,
                            LR_Bus     := LR_Bus.Initial_State,
                            Right_Side := Right_Side.Initial_State,
                            RL_Bus     := RL_Bus.Initial_State,
                            pre_TS     := TRUE                #)
```

Figure 44 – PVS Specification of the Asynchronous Pilot Flying Example – Part 1

The specification of the asynchronous system is very similar to the synchronous specification given in Figure 18 and Figure 19. The main difference is that the *next_state* function takes each of the four clocks as inputs and passes the appropriate clock value to each component.

```
  %-------------------------------------------------------------------
  % Next state function
  %-------------------------------------------------------------------
  next_state(s: State, CLK1, CLK2, CLK3, CLK4, TS: bool): State =
    LET
        %-------------------------------------------------------------
        % Compute the next state of LR Bus and the Right Side
        %-------------------------------------------------------------
        C1      = Left_Side.output(Left_Side(s)),
        next_LR = LR_Bus.next_state(LR_Bus(s), CLK2, C1),
        C2      = LR_Bus.output(next_LR),
        next_RS = Right_Side.next_state(Right_Side(s), CLK3, TS, C2),

        %-------------------------------------------------------------
        % Compute the next state of the RL Bus and the Left Side
        %-------------------------------------------------------------
        C3      = Right_Side.output(Right_Side(s)),
        next_RL = RL_Bus.next_state(RL_Bus(s), CLK4, C3),
        C4      = RL_Bus.output(next_RL),
        next_LS = Left_Side.next_state(Left_Side(s), CLK1, TS, C4)
    IN
       (# Left_Side  := next_LS,
          LR_Bus     := next_LR,
          Right_Side := next_RS,
          RL_Bus     := next_RL,
          pre_TS     := TS        #)

  %-------------------------------------------------------------------
  % Outputs
  %-------------------------------------------------------------------
  Left_Pilot_Flying_Side(s: State) : bool = pfs(output(Left_Side(s)))

  Right_Pilot_Flying_Side(s: State): bool = pfs(output(Right_Side(s)))

END Pilot_Flying_System
```

Figure 45 – PVS Specification of the Asynchronous Pilot Flying Example – Part 2

3.4.7 Formal Verification of the Asynchronous Pilot Flying Example

Formal verification of the asynchronous Pilot Flying system proceeds in much the same was as for verification of the synchronous example. The informal requirements for the asynchronous system are identical to those for the synchronous system:

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 formal specification of these requirements is shown in Figure 46 through Figure 56.

The definition of a *Reachable_State* is similar to that given for the synchronous example except that the *next_state* function now depends on the clock of each component. Since each component only takes a step if its clock is true, this means that there are more reachable states than for the synchronous system. This is to be expected since it is the varying execution rates of the components that are the source of potential race conditions and deadlocks.

To simplify the formal specification of the requirements, we define a *stable_state* to be one in which the *ack* of each side is true. This corresponds to the states where one side is in the *Confirmed* sub-state and the other side is in the *Listening* sub-state (see Figure 36). Of course, it also includes states where both sides are in the *Confirmed* or *Listening* sub-state, but these will be excluded in the formal specification of the requirements as unreachable states.

Since the two sides may now execute on different steps, it is no longer true that if one side sees the Transfer Switch pressed, the other side will also see the Transfer Switch pressed on the same step. To more precisely specify requirements involving the Transfer Switch, we introduce three definitions for when the Transfer Switch is pressed. The predicates *pressed_seen_left* and *pressed_seen_right* are true when the Transfer Switch is observed to be pressed by the left or right side respectively, i.e. when the Transfer Switch has risen from the previous value observed by that side. The predicate *pressed* is true when the Transfer Switch has risen from its previous value regardless of the value of each side's clock.

```
Pilot_Flying_System_Requirements: THEORY
  BEGIN

    IMPORTING Pilot_Flying_System

    s:                          VAR Pilot_Flying_System.State
    m1, m2, m3:                 VAR Message
    ts, clk1, clk2, clk3, clk4: VAR bool

    %-------------------------------------------------------------------
    % Definition of a reachable state
    %-------------------------------------------------------------------
    Reachable_State(s): INDUCTIVE bool =
      s = Initial_State OR
          (EXISTS (r: Pilot_Flying_System.State, c1, c2, c3, c4, t: bool):
                      Reachable_State(r) AND
                         s = next_state(r, c1, c2, c3, c4, t))

    %-------------------------------------------------------------------
    % The system is stable when both side's acks are true
    %-------------------------------------------------------------------
    stable_state(s): bool =
      (ack(output(Left_Side(s))) and ack(output(Right_Side(s))))

    %-------------------------------------------------------------------
    % Definitions for Transfer Switch pressed
    %-------------------------------------------------------------------
    pressed(ts, s) : bool = NOT pre_TS(s) AND ts
    pressed_seen_left(ts, s) : bool = not pre_TS(Left_Side(s)) and ts
    pressed_seen_right(ts, s): bool = not pre_TS(Right_Side(s)) and ts;
```

Figure 46 – Asynchronous Pilot Flying System Requirements (Part 1)

Verification of the system properties is based on definition of a *Valid_State* predicate just as was done for the synchronous example. However, definition of the *Valid_State* predicate is more complex in the asynchronous system and makes use of several auxiliary definitions shown in Figure 47.

```
%------------------------------------------------------------------------
% Predicates used to define Valid State
%------------------------------------------------------------------------
Confirmed(m1) : bool =     pfs(m1) AND     ack(m1);
Inhibited(m1) : bool = NOT pfs(m1) AND NOT ack(m1);
Listening(m1) : bool = NOT pfs(m1) AND     ack(m1);
Waiting  (m1) : bool =     pfs(m1) AND NOT ack(m1);

%------------------------------------------------------------------------
% Equality of messages
%------------------------------------------------------------------------
== : [Message, Message -> bool] =
     LAMBDA (m1, m2) : pfs(m1) = pfs(m2) AND ack(m1) = ack(m2);

%------------------------------------------------------------------------
% Ordering of messages
%------------------------------------------------------------------------
>> : [Message, Message -> bool] =  LAMBDA (m1, m2) :
     Inhibited(m1) AND Confirmed(m2) OR
     Listening(m1) AND Inhibited(m2) OR
     Waiting(m1)   AND Listening(m2) OR
     Confirmed(m1) AND Waiting(m2);
```

Figure 47 – Asynchronous Pilot Flying System Requirements (Part 2)

Since each side generates an output message consisting of a pilot flying indication (*pfs*) and an acknowledgement (*ack*), it is convenient to name each of the four possible messages to match the sub-state of the side from which it is generated. This is done by defining predicates *Confirmed*, *Inhibited*, *Listening* and *Waiting* over the type *Message* as shown in Figure 47. We also define equality over *Messages* and an ordering of *Messages*. The intuition behind the ordering relation is that a message *m1* follows a message *m2* (*m1 >> m2*) if *m1* is the next state after *m2* in the state transition diagram of Figure 36, i.e. *Inhibited >> Confirmed*, *Listening >> Inhibited* and so forth.

```
%-----------------------------------------------------------------------
% Valid state constraints over component states
%-----------------------------------------------------------------------
 Side_Bus_Side_Consistency(m1, m2, m3) : bool =
     m1 == m2 and m2 == m3 OR
     m1 >> m2 and m2 == m3 OR
     m1 == m2 and m2 >> m3 OR
     Waiting(m1) AND Inhibited(m2) AND Inhibited(m3) OR
     Waiting(m1) AND Listening(m2) AND Inhibited(m3) OR
     Waiting(m1) AND Waiting  (m2) AND Inhibited(m3)

%-----------------------------------------------------------------------
% Valid state constraints on side components
%-----------------------------------------------------------------------
 Side_Consistency(m1, m2) : bool =
     Listening(m1) AND Confirmed(m2) OR
     Waiting  (m1) AND Inhibited(m2) OR
     Confirmed(m1) AND Listening(m2) OR
     Confirmed(m1) AND Waiting  (m2) OR
     Inhibited(m1) AND Confirmed(m2)

%-----------------------------------------------------------------------
% Full definition of a valid state
%-----------------------------------------------------------------------
Valid_State: [Pilot_Flying_System.State -> bool] =
  {s | LET
          LS =  output(Left_Side(s)),
          LR =  LR_Bus.output(LR_Bus(s)),
          RP =  pre_MSG(Right_Side(s)),
          RS =  output(Right_Side(s)),
          RL =  RL_Bus.output(RL_Bus(s)),
          LP =  pre_MSG(Left_Side(s))
       IN
          Side_Bus_Side_Consistency(LS, LR, RP) AND
          Side_Bus_Side_Consistency(RS, RL, LP) AND
          Side_Consistency(LP, LS)              AND
          Side_Consistency(RP, RS)
  }

%-----------------------------------------------------------------------
% Proof that every reachable state is a valid state
%-----------------------------------------------------------------------
 Reachable_States_Valid: THEOREM
    Reachable_State(s) => Valid_State(s)
```

Figure 48 – Asynchronous Pilot Flying System Requirements (Part 3)

There are two types of consistency defined in Figure 48 that we wish to enforce in the definition of *Valid_State*. *Side_Bus_Side_Consistency* captures the constraint that if a change in sub-state occurs in a side, it will transfer across the bus and eventually be stored in the other side as the previous value of the first side's message. To illustrate, consider the case where *m1* is the output

message of the *Left_Side*, *m2* is the output message of the *LR_Bus*, and *m3* is the *pre_MSG* of the *Right_Side* (an analogous situation holds for the *Right_Side*, *RL_Bus*, and *Left_Side*). One possibility is that the sub-state of the *Left_Side* has been transmitted to the *Right_Side* and *m1*, *m2* and *m3* are identical (*m1 == m2* and *m2 == m3*). Another possibility is that the *Left_Side* has just changed to a following sub-state, but the *LR_Bus* and the *Right_Side* have not yet seen the change (*m1 >> m2* and *m2 == m3*). Another possibility is that the *LR_Bus* has been updated but the change has not yet reached the *Right_Side* (*m1 == m2* and *m2 >> m3*).

There are three other possible relationships that are explicitly enumerated in *Side_Bus_Side_Consistency*. Let *m1*, *m2*, and *m3* be as just described. Consider the situation where the *Left_Side* has just entered the *Listening* sub-state, but that information has not yet been communicated to the *LR_Bus* so that the system is in the state *Listening(m1)*, *Inhibited(m2)* and *Inhibited(m3)*. The first case occurs when the Transfer Switch is pressed and the system enters the state *Waiting(m1)*, *Inhibited(m2)* and *Inhibited(m3)*. The second case occurs when the Transfer Switch is pressed while in the state *Listening(m1)*, *Listening(m2)* and *Inhibited(m3)*, putting the system into the state *Waiting(m1)* and *Listening(m2)* and *Inhibited(m3)*. The third case evolves directly from the second case when the *LR_Bus* is updated with the *Left_Side*'s output, putting the system into the *Waiting(m1)* and *Waiting(m2)* and *Inhibited(m3)* state. Analogous situations exist for the *Right_Side*, *RL_Bus*, and the *Left_Side*.

The *Side_Consistency* predicate identifies relationships that must be maintained between the sub-state of a side and its copy of the previous message from the other side. For example, if a side is in the *Confirmed* sub-state, then the previous message from the other side must be either *Listening* or *Inhibited*.

All of these constraints on the valid system states are collected in the definition of *Valid_State* in Figure 48. The *Reachable_States_Valid* theorem states that all reachable states are also valid states for the asynchronous Pilot Flying system. The proof of this theorem is similar to the proof for the synchronous example shown in Figure 23, but due to the larger number of reachable states, more work is needed to keep the proof tractable. For example, simply applying the PVS *grind* command to the inductive branch of the proof as was done in the synchronous case generates 544 sub-goals. While each of these can be easily dispatched with a few PVS commands, the proof can be almost fully automated with two additional steps.

When proving the 544 sub-goals, the PVS theorem prover often requires human assistance to case split a sub-goal in order to determine which of the *Side_HLR* axioms should be invoked. It is actually easier for the theorem prover to use a constructive definition more like that found in the *Side_LLR* theory. However, we prefer to maintain the declarative style using axioms found in *Side_HLR* since this is closer to the traditional "shall" statements software engineers expect to receive as requirements. To do this while still facilitating theorem proving, we add to the *Side_HLR* theory the two lemmas as shown in Figure 49.

```
%-----------------------------------------------------------------
% Lemmas used to simplify system level proofs
%-----------------------------------------------------------------
L1: LEMMA
   FORALL (s: State, clk: bool, ts:bool, m: Message) :
      output(next_state(s, clk, ts, m))  =
        IF (NOT clk) THEN
           output(s)
        ELSIF (confirmed(s) AND rise_ospf(s, m)) THEN
           (# pfs := FALSE, ack := FALSE #)
        ELSIF (inhibited(s) AND ack(m)) THEN
           (# pfs := FALSE, ack := TRUE #)
        ELSIF (listening(s) AND rise_ts(s, ts)) THEN
           (# pfs := TRUE, ack := FALSE #)
        ELSIF (waiting(s) AND fall_ack(s, m)) THEN
           (# pfs := TRUE, ack := TRUE #)
        ELSE
           output(s)
      ENDIF

L2: LEMMA
   forall (s: State, clk: bool, ts:bool, m: Message) :
      pre_MSG(next_state(s, clk, ts, m))  =
        IF (NOT clk) THEN pre_MSG(s) ELSE m  ENDIF
```

Figure 49 – Lemmas Added to Side_HLR to Support Theorem Proving

These lemmas define the effect of applying the *output* and *pre_MSG* extractors to the *next_state* constructor in a constructive style that defines the result for all possible conditions. These lemmas can easily be proven using the axioms of *Side_HLR* and the PVS theorem prover can be instructed to invoke these lemmas as automatic rewrite rules using the PVS *use* command. This eliminates the need for manual intervention to case split a sub-goal.

The second step for proof automation is to define a custom PVS strategy. PVS strategies can be thought of as user-defined proof commands constructed from the basic proof commands of PVS. The strategy needed here is shown in Figure 50.

```
(defstep grind-use-grind (&rest lemmas)
  (let ((uselems (cons 'use* lemmas)))
   (then
     (grind) uselems (grind)
   ))
    "Applies grind, then uses lemmas followed by grind on on each subgoal."
    "Applying grind - use - grind."
 )
```

Figure 50 – Grind-use-grind PVS Proof Strategy

This strategy defines a PVS proof command called *grind-use-grind* that accepts a sequence of lemma names as an argument. Its effect is to apply the PVS *grind* command to the current sub-goal, possibly generating one or more new sub-goals. It then invokes the named lemmas on each new sub-goal with the PVS *use* command, and finally applies the *grind* command to that sub-goal. An illustration of its use is shown in the last step of the proof for the asynchronous *Reachable_States_Valid* theorem shown in Figure 51.

```
;;; Proof Reachable_States_Valid-1 for formula
Pilot_Flying_System_Requirements.Reachable_States_Valid
;;; developed with shostak decision procedures
(""
 (rule-induct "Reachable_State")
 (skosimp)
 (case "s!2 = Initial_State")
 (("1"
   (auto-rewrite-theories "Message" "Left_Side" "LR_Bus" "Right_Side"
    "RL_Bus")
   (grind))
  ("2"
   (assert)
   (hide 1)
   (grind-use-grind "LR_Bus.HLR2" "RL_Bus.HLR2" "Left_Side.L1"
    "Left_Side.L2" "Right_Side.L1" "Right_Side.L2"))))
```

Figure 51 – PVS Proof of Asynchronous Reachable States Valid Theorem

Successful completion of the proof of Figure 51 establishes that every reachable state of the asynchronous *Pilot_Flying_System* is also a valid state. The *Reachable_States_Valid* theorem can then be used to prove requirements R1 and R2 shown in Figure 52, both of which can be discharged with a PVS *(use "Reachable_States_Valid)* command followed by a *grind* command.

```
%------------------------------------------------------------------
% R1. At least one side shall always be the pilot flying side.
%------------------------------------------------------------------
R1: THEOREM
  Reachable_State(s) =>
    Left_Pilot_Flying_Side(s) OR Right_Pilot_Flying_Side(s)

%------------------------------------------------------------------
% R2. Both sides shall agree on the pilot flying side
%     except while the system is switching sides.
%------------------------------------------------------------------
R2: THEOREM
  Reachable_State(s) AND stable_state(s) =>
    (Left_Pilot_Flying_Side(s) = NOT Right_Pilot_Flying_Side(s))
```

Figure 52 – Asynchronous Pilot Flying System Requirements (Part 4)

However, the proof of requirement R3 (pressing the Transfer Switch shall always change the pilot flying side) reveals a problem. To formalize R3, we break it into two smaller requirements R3a and R3b as shown in Figure 53.

```
%------------------------------------------------------------------
% R3. Pressing the transfer switch shall always change the pilot
%     flying side except when the system is switching sides.
%------------------------------------------------------------------
R3a: THEOREM
  Reachable_State(s) AND stable_state(s) =>
    (NOT Left_Pilot_Flying_Side(s) AND pressed(ts, s) =>
         Left_Pilot_Flying_Side(next_state(s,TRUE,clk2,clk3,clk4,ts)))

R3b: THEOREM
  Reachable_State(s) AND stable_state(s) =>
     (NOT Right_Pilot_Flying_Side(s) AND pressed(ts, s) =>
          Right_Pilot_Flying_Side(next_state(s,clk1,clk2,TRUE,clk4,ts)))
```

Figure 53 – Incorrect Statement of Asynchronous Requirement R3

These requirements state that if the Transfer Switch is pressed while the system is in a stable state (i.e. is not switching sides) the side that is not the pilot flying side shall become the pilot flying side providing its clock is true. However, trying to prove either theorem generates four sub-goals that cannot be proven. These correspond to cases in which the Transfer Switch was true the last time the side's clock was true. In other words, even if the Transfer Switch is pressed on this step and the side's clock is true, the side does not see a rising edge of the Transfer Switch because the last time its clock was true the Transfer Switch was also true.

There is no easy fix for this problem. Instead, the system must be designed to ensure that the pilot not flying side sees the Transfer Switch being pressed. In an actual system, it would be possible to place additional requirements on the length of time the Transfer Switch must remain high and the time it must remain low to ensure that the pilot not flying side would see a rising edge of the Transfer Switch, but since we are making no assumptions about the system clock that will not work for our system. To make this explicit, we replace the *pressed* predicate in R3a and R3b with *pressed_seen_left* and *pressed_seen_right* as shown in Figure 54. This makes it clear the requirements are satisfied only if the pilot not flying side actually observes the Transfer Switch being pressed. Both of these theorems can be proven with a PVS *(use "Reachable_States_Valid")* command followed by a *grind-use-grind* command invoking the appropriate lemmas.

```
%-----------------------------------------------------------------
% R3. Pressing the transfer switch shall always change the pilot
%     flying side except when the system is switching sides.
%-----------------------------------------------------------------
R3a: THEOREM
  Reachable_State(s) AND stable_state(s) =>
    (NOT Left_Pilot_Flying_Side(s) AND pressed_seen_left(ts, s) =>
         Left_Pilot_Flying_Side(next_state(s,TRUE,clk2,clk3,clk4,ts)))

R3b: THEOREM
  Reachable_State(s) AND stable_state(s) =>
     (NOT Right_Pilot_Flying_Side(s) AND pressed_seen_right(ts, s) =>
          Right_Pilot_Flying_Side(next_state(s,clk1,clk2,TRUE,clk4,ts)))

%-----------------------------------------------------------------
% R4. The system shall start with the left side as the pilot
%     flying side.
%-----------------------------------------------------------------
R4: THEOREM
  Left_Pilot_Flying_Side(Initial_State)
```

Figure 54 – Asynchronous Pilot Flying System Requirements (Part 5)

Requirement R4 (the system shall start with the left side as the pilot flying side) also shown in Figure 54 is easily proven by installing the axioms of *Side_HLR* and *Bus_HLR* as automatic rewrite rules followed by a *grind* command. However, requirement R5 (the system shall not change the pilot flying side unless the Transfer Switch is pressed) turns out to not be true as formulated in Figure 55

```
%-----------------------------------------------------------------
% R5. The system shall not change the pilot flying side while it
%     is in a stable state unless the transfer switch is pressed.
%-----------------------------------------------------------------
R5a: THEOREM
   Valid_State(s) AND stable_state(s) AND NOT pressed(ts, s)  =>
        Left_Pilot_Flying_Side(next_state(s,clk1,clk2,clk3,clk4,ts)) =
            Left_Pilot_Flying_Side(s)

R5b: THEOREM
   Valid_State(s) AND stable_state(s) AND NOT pressed(ts, s) =>
        Right_Pilot_Flying_Side(next_state(s,clk1,clk2,clk3,clk4,ts)) =
            Right_Pilot_Flying_Side(s)
```

Figure 55 – Incorrect Statement of Asynchronous Requirement R5

The counter example for this occurs when the clock for the *Not_Pilot_Flying* side becomes false while the Transfer Switch is false. At a later time, the Transfer Switch becomes true, but the *Not_Pilot_Flying* side does not observe this rising edge because its clock is still false. The Transfer Switch remains true for several steps and during this period, the *Not_Pilot_Flying* side's clock becomes true. At this point, the *Not_Pilot_Flying* side observes a rising edge of the Transfer Switch and becomes the *Pilot_Flying_Side*. However, the predicate *pressed* is not true on that step since the Transfer Switch has been true for several steps.

Requirement R5 is false due to a subtle interaction between the component clocks and their stored value of the Transfer Switch. The intent of this requirement was that the system should not spontaneously change state while in a stable state unless there is some external stimulation. Interestingly enough, it is possible to prove R5 if it is restated as shown in Figure 56.

```
   %-----------------------------------------------------------------
   % R5. The system shall not change the pilot flying side while it
   %     is in a stable state unless the transfer switch is high.
   %-----------------------------------------------------------------
   R5a: THEOREM
      Valid_State(s) AND stable_state(s) AND NOT ts =>
           Left_Pilot_Flying_Side(next_state(s,clk1,clk2,clk3,clk4,ts)) =
               Left_Pilot_Flying_Side(s)

   R5b: THEOREM
      Valid_State(s) AND stable_state(s) AND NOT ts =>
           Right_Pilot_Flying_Side(next_state(s,clk1,clk2,clk3,clk4,ts)) =
               Right_Pilot_Flying_Side(s)

END Pilot_Flying_System_Requirements
```

Figure 56 – Asynchronous Pilot Flying System Requirements (Part 6)

Here, we have replaced the *NOT pressed(ts, s)* with *NOT ts*, so that the requirement states that the system shall not change the pilot flying side while it is in a stable state unless the Transfer Switch is high. While technically a slightly weaker requirement, this formulation still satisfies the original intent of the requirement and is easily proven to be true.

3.5 The Synchronous Pilot Flying Example in HOL

Higher Order Logic (HOL) is a formal system originally adopted and implemented by Mike Gordon [7], [8]. Subsequently, other implementations of HOL have been developed [24], [25], [13], [27], [26], [17]. In general, these systems agree on the formal system implemented [24], but their interfaces and proof infrastructure can be quite different. In our example we will work with HOL4 [25], [12].

In HOL4 notation, '~' is the 'not' operator, '/\' is conjunction, '\/' is disjunction, 'T' is True, and 'F' is False. The '!' symbol is 'for all', and the '?' symbol is 'there exists'. Finally, '<|' and '|>' are used to indicate a record structure.

3.5.1 Specification in HOL4 using a Next-State Approach

This section describes a HOL4 specification of the synchronous pilot flying example using a "this state, next state" approach similar to what was done in PVS. We start by defining the *RISE* function in terms of a Boolean signal *s*. We then define data types for *PFS* (cf. *Pilot_Flying_Side* type in Figure 15) and *Side_State* (cf. *State* type in Figure 15). We also define *System_State* (cf. *State* type in Figure 18). These HOL4 definitions are shown in Figure 57.

```
numLib.prefer_num();

(*------------------------------------------------------------------------*)
(* Rise definition                                                        *)
(*------------------------------------------------------------------------*)

val RISE_def =
 Define
  `RISE(pre_s : bool,
        s     : bool)
   =
   ~pre_s /\ s`;

(*------------------------------------------------------------------------*)
(* Data type definitions                                                  *)
(*------------------------------------------------------------------------*)

Hol_datatype
  `PFS = Pilot_Flying
       | Not_Pilot_Flying`;

Hol_datatype
  `Side_State =
      <| st: PFS;
         pre_ts: bool;
         pre_ospf: bool
      |>`;

Hol_datatype
  `System_State =
      <| stateLS: Side_State;
         stateLR: bool;
         stateRS: Side_State;
         stateRL: bool;
         pre_ts: bool
       |>`;
```

Figure 57 – Rise and Data Type Definitions

Next we define the initial states for the sides (both the primary side and the non-primary side), the buses, and the system. These are shown in Figure 58 and correspond with the initial state definitions in PVS.

```
val Initial_Side_State_Primary_def =
 Define
  `Initial_Side_State_Primary =   <| st:= Pilot_Flying;
                                     pre_ts:= T;
                                     pre_ospf:= F |>`;

val Initial_Side_State_Not_Primary_def =
 Define
  `Initial_Side_State_Not_Primary = <| st:= Not_Pilot_Flying;
                                       pre_ts:= T;
                                       pre_ospf:= T |>`;

val Initial_Bus_State_Primary_def =
 Define
  `Initial_Bus_State_Primary = T`;

val Initial_Bus_State_Not_Primary_def =
 Define
  `Initial_Bus_State_Not_Primary = F`;

val Initial_System_State_def =
 Define
  `Initial_System_State = <| stateLS := Initial_Side_State_Primary;
                             stateLR := Initial_Bus_State_Primary;
                             stateRS := Initial_Side_State_Not_Primary;
                             stateRL := Initial_Bus_State_Not_Primary;
                             pre_ts  := T |>`;
```

Figure 58 – Initial State Definitions

To complete the specification, we have left to define the next state functions for Side, Bus, and System. These functions are called *Side_ns*, *Bus_ns*, and *System_ns* in the code shown in Figure 59. Note that the *Bus_ns* function simply returns the input value. The delay is implemented at the top level in the *System_ns* function.

```
val Side_ns_def =
 Define
  `Side_ns (state : Side_State,
            ts    : bool,
            ospf  : bool) =
   let
    nextst =
     if (state.st = Pilot_Flying) /\ RISE(state.pre_ospf, ospf)
         then Not_Pilot_Flying
     else if (state.st = Not_Pilot_Flying) /\ RISE(state.pre_ts, ts)
       then Pilot_Flying
     else state.st
   in
    state with <| st:= nextst; pre_ts:= ts; pre_ospf:= ospf |>`;

val Bus_ns_def =
 Define
  `Bus_ns (input : bool) = input`;

val System_ns_def =
 Define
  `System_ns (state : System_State,
              ts    : bool) =
   let
     nextLR = Bus_ns (state.stateLS.st = Pilot_Flying) and
     nextRL = Bus_ns (state.stateRS.st = Pilot_Flying)
   in
    state with <| stateLS := Side_ns(state.stateLS, ts, nextRL);
                  stateLR := nextLR;
                  stateRS := Side_ns(state.stateRS, ts, nextLR);
                  stateRL := nextRL;
                  pre_ts  := ts |>`;
```

Figure 59 – Next State Definitions

3.5.2 Formal Verification of the Next-State Approach in HOL4

In this section we discuss formal verification in HOL4 of the specification described in the previous section. We prove one property ("At Least One Side Flying") as an example.

Proving properties of the system proceeds in a manner very similar to that used in PVS. We will walk through the proof of the first property, R1: "At least one side shall be the pilot flying side." We first define *Reachable_State.* We then define *Valid_State*, which contains all system states satisfying the following four predicates: *Pre_TS_Consistency, Pre_OSPF_Consistency, At_Least_One_Side_Flying,* and *Buses_Differ_When_Both_Sides_Flying.* These definitions are shown in Figure 60.

```
(*-------------------------------------------------------------------------*)
(* Reachable_State                                                         *)
(*-------------------------------------------------------------------------*)

val reachable = Hol_reln
 `Reachable Initial_System_State
  /\
  (!s1 s2 ts. Reachable s1 /\ (System_ns(s1,ts) = s2) ==> Reachable s2)`;

val (Reachable_rules,Reachable_induction,Reachable_cases) = reachable;

(*-------------------------------------------------------------------------*)
(* Definitions required by Valid_State                                     *)
(*-------------------------------------------------------------------------*)

val Pre_TS_Consistency_def =
 Define
  `Pre_TS_Consistency s =
      (s.stateLS.pre_ts = s.pre_ts) /\ (s.stateRS.pre_ts = s.pre_ts)`;

val Pre_OSPF_Consistency_def =
 Define
 `Pre_OSPF_Consistency s =
     (s.stateLS.pre_ospf = s.stateRL) /\ (s.stateRS.pre_ospf = s.stateLR)`;

val At_Least_One_Side_Flying_def =
 Define
  `At_Least_One_Side_Flying s =
        (s.stateLS.st = Pilot_Flying) \/ (s.stateRS.st = Pilot_Flying)`;

val Buses_Differ_When_Both_Sides_Flying_def =
 Define
  `Buses_Differ_When_Both_Sides_Flying s =
       (s.stateLS.st = Pilot_Flying) /\ (s.stateRS.st = Pilot_Flying)
       ==> ~(s.stateLR = s.stateRL)`;

(*-------------------------------------------------------------------------*)
(* Valid_State                                                             *)
(*-------------------------------------------------------------------------*)

val Valid_State_def =
 Define
  `Valid_State s = Pre_TS_Consistency s /\
                   Pre_OSPF_Consistency s /\
                   At_Least_One_Side_Flying s /\
                   Buses_Differ_When_Both_Sides_Flying s`;
```

Figure 60 – Valid State Definition

We wish to show the first property, R1: “At least one side shall be the pilot flying side.” We do this in two steps. We show that all reachable states are valid, and then we show that all valid states satisfy R1. First, we define a custom simplification set (shown in Figure 61), which we

will use in nearly all of our proofs. This simplification set expands the definitions in our specification as needed and includes LET_THM to simplify complicated LET expressions.

```
val sys_ss = srw_ss() ++ rewrites
    [Pre_TS_Consistency_def,
     Pre_OSPF_Consistency_def,
     At_Least_One_Side_Flying_def,
     Buses_Differ_When_Both_Sides_Flying_def,
     Valid_State_def,
     Side_ns_def,
     Bus_ns_def,
     System_ns_def,
     RISE_def,
     Initial_System_State_def,
     Initial_Side_State_Primary_def,
     Initial_Side_State_Not_Primary_def,
     Initial_Bus_State_Primary_def,
     Initial_Bus_State_Not_Primary_def,
     LET_THM];
```

Figure 61 – Custom Simplification Set *sys_ss*

Using the *sys_ss* simplification set just defined, we can show that the initial state is valid and that the next state of a valid state is valid. These two theorems are then used to show that all reachable states are valid (see Figure 62).

```
val Valid_State_Base = ``Valid_State Initial_System_State``;


val Valid_State_Base_Thm = prove
(Valid_State_Base,
 RW_TAC sys_ss [] THEN METIS_TAC[]);

val Valid_State_Inductive =
  ``!s ts. Valid_State s ==> Valid_State (System_ns(s,ts))``;

val Valid_State_Inductive_Thm = prove
(Valid_State_Inductive,
 RW_TAC sys_ss [] THEN METIS_TAC[]);

val Reachable_States_Are_Valid = prove
(``!s. Reachable s ==> Valid_State s``,
 Induct_on `Reachable s`
   THEN METIS_TAC [Valid_State_Base_Thm,
                   Valid_State_Inductive_Thm]);
```

Figure 62 – HOL4 Proof that all Reachable States are Valid

Next we show that all valid states satisfy R1: “At least one side shall be the pilot flying side.” The proof is done using rewriting with the custom simplification set *sys_ss* as shown in

```
val R1_Thm = store_thm
 ("R1_Thm",
 ``!s. Valid_State s ==>
    (s.stateLS.st = Pilot_Flying) \/ (s.stateRS.st = Pilot_Flying)``,
 RW_TAC sys_ss []);
```

Figure 63 – HOL4 Proof that All Valid States Satisfy R1

We now put the two pieces together to prove that all reachable states satisfy R1.

```
val R1_Thm_Reachable = store_thm
 ("R1_Thm",
 ``!s. Reachable s ==>
    (s.stateLS.st = Pilot_Flying) \/ (s.stateRS.st = Pilot_Flying)``,
 METIS_TAC [R1_Thm, Reachable_States_Are_Valid]);
```

Figure 64 – HOL4 Proof that All Valid States Satisfy R1

3.5.3 Specification in HOL4 using a Streams Approach

Modeling components that operate over infinite streams of data can be approached in a variety of ways. An approach that is well-suited to higher order logic represents a stream of elements, where each element has type τ, by a function of type num→ τ. A device, or component, having a number of ports is modeled as a predicate on the possible values of the ports. Since a port is a stream, the behavior of the device is represented by a predicate that takes a bundle of streams and returns true or false.

For example, a bus that acts as a unit delay can be represented as

```
  Bus (init,input,output) =
     (output 0 = init) /\
     (output (n+1) = input n)
```

Figure 65 – Bus Specification with Streams Approach

where *input* and *output* have type num→α, and *init*, the initial value of the stream, has type α. The type variable α shows that the definition is polymorphic, and thus can be instantiated to yield a bus over streams of any type.

The parallel composition of devices is achieved by conjunction of the corresponding predicates. Combining devices by connecting them together is modeled by relational composition: the resulting device is a predicate on the external ports, and internal connections are hidden by existential quantification. Thus, we may obtain a bus that delays data by two clock ticks by composing two buses:

```
Two_Delay_Bus (init1,init2,input,output) =
  ?c. Bus (init1,input,c) /\
      Bus (init2,c,output)
```

Figure 66 – Two-Delay Bus Specification with Streams Approach

Suppose we want to say that there has been a "rise" at time *t* on a Boolean stream, written "RISE stream t". At time zero there can be no rise. At any other time, a rise occurs at time *t* if the line is low at time *t-1* and high at time *t*. This can be directly modeled as follows:

```
(RISE stream 0 = F) /\
 (RISE stream (t+1) = ~(stream t) /\ stream (t+1)
```

Figure 67 – 'Rise' Definition with Streams Approach

The *RISE* predicate has been defined by explicit case analysis on whether a number is zero or a successor. An equivalent definition is the following:

```
RISE stream t = if t = 0 then F else  ~(stream (t-1)) /\ stream t
```

Figure 68 – Alternate 'Rise' Definition with Streams Approach

This modeling style has been used successfully on a wide variety of system verifications [20]. It is well suited to systems where all components share the same clock, but it can also be extended to deal with asynchronous systems with handshakes, for example. It can express circuits with feedback loops without having to solve tricky recursive equations.

3.5.3.1 Properties

Expressing properties of systems modeled in this style involves asserting that the behaviors of a device, or implementation, are contained within the behaviors allowed by a specification. Using the fact that subset is defined by implication, we obtain the general recipe:

implementation ==> specification

This can also be read as "anything that models the implementation is also a model of the specification". Note that there is a residual question to be dealt with, namely one would want to show that there is indeed a model of the implementation. However, when working with implementations imported into logic from an existing design, this step is often skipped over.

3.5.3.2 System Model

We will proceed in a top-down fashion. The flight guidance system can be directly modeled as follows:

```
System (L,R,Transfer_Switch) =
     ?w1 w2.
       Side(T,Transfer_Switch,w2,L) /\    (* Left Side *)
       Side(F,Transfer_Switch,w1,R) /\    (* Right Side *)
       Bus(L,w1) /\                       (* LR_Bus *)

       Bus(R,w2)                          (* RL_Bus *)
```

Figure 69 – System Specification with Streams Approach

The external ports of the system are *L*, *R,* and *Transfer_Switch*. There are also two internal buses *w1*, *w2* that connect the two sides. The left side is initialized with **T** and the right side with **F**, setting the initial pilot flying side. Both sides take the transfer switch as input and connect to the buses and the external ports *L* and *R*.

Each *Side* component implements some decision-making logic. This is defined as follows:

```
     Side (Initial_Pilot_Flying_Side : bool,
         Transfer_Switch           : num->bool,
         Other_Side_Pilot_Flying   : num->bool,
         Pilot_Flying_Side         : num->bool)
     =
     (Pilot_Flying_Side 0 = Initial_Pilot_Flying_Side) /\
     (!t. Pilot_Flying_Side (SUC t) =
          if (Pilot_Flying_Side t = T) /\
             RISE Other_Side_Pilot_Flying (SUC t)
              then F
          else if (Pilot_Flying_Side t = F) /\
                  RISE Transfer_Switch (SUC t)
              then T
          else Pilot_Flying_Side t)
```

Figure 70 – Side Specification with Streams Approach

This tiny state machine is defined by case analysis on whether the system is at its initial step (time 0) or otherwise.

We have specified the buses as pure unit delays, omitting to specify the output at time zero:

```
Bus (instream:num->bool, outstream:num->bool)
     =
    !t. outstream (SUC t) = instream t
```

Figure 71 – Final Bus Specification for Streams Approach

In an earlier formalization of the system, initial values were specified for the buses, but the proofs revealed that the proof succeeded no matter what the initial values were set to. By omitting initial values, the statements and proofs about the system become more general, since any possible initial values will be acceptable for the purposes of proof.

3.5.4 Formal Verification of the Streams Approach in HOL4

We proved two facts about the system in HOL4. First, we proved that the *L* port is high at time zero. The statement of this is

```
System(L,R,Transfer_Switch) ==> L(0)
```

Figure 72 – Statement of R4 Property

and the proof is quite straightforward, as we discuss below. Next, we proved that at least one side is the pilot flying side:

```
System(L,R,Transfer_Switch) ==> !t. L(t) \/ R(t)
```

Figure 73 – Statement of R1 Property

This proof is more challenging but still relatively small.

3.5.4.1 Left at Time Zero

This proof is straightforward. We merely simplify with the system definitions and then reduce.

```
val Left_at_Time_Zero = prove
(``System(L,R,Transfer_Switch) ==> L(0)``,
 RW_TAC std_ss [System_def, Side_def,Bus_def]);
```

Figure 74 – Proof of R4 Property for Streams Approach

3.5.4.2 At Least One Side Flying

This property is more challenging. It requires induction and the system state at time t can depend on the state at times *t*-1 and *t*-2, which means that complete induction is needed. The proof then proceeds by case analysis on whether the system is at step 0, 1, or some arbitrary number greater than one. The base cases are simple to prove by simplification with the definitions of the system components. The inductive hypothesis yields 4 possible combinations for values of the two Pilot_Flying_Side ports at the two previous steps in the computation. Some basic reasoning finishes the proof.

The full proof of this property, once packaged up, is shown in Figure 75.

```
val At_Least_One_Side_Flying =
Count.apply prove
 (``System(L,R,Transfer_Switch) ==> !t. L(t) \/ R(t)``,
  DISCH_TAC
    THEN completeInduct_on `t`
    THEN `(t = 0) \/ (t = SUC 0) \/ ?k. t = SUC (SUC k)`
           by METIS_TAC [arithmeticTheory.num_CASES]
    THEN FULL_SIMP_TAC kstd_ss [System_def,Bus_def,Side_def,RISE_def]
    THEN `(L k \/ R k) /\ (L (SUC k) \/ R (SUC k))`
           by METIS_TAC [prim_recTheory.LESS_SUC_SUC]
    THEN METIS_TAC[]);
```

Figure 75 – Proof of R1 Property for Streams Approach

Elapsed time was about a fifth of a second. In the course of the proof, no axioms were declared, no definitions were made, and no theories were brought in from disk. Approximately 41,000 primitive inference steps were required to achieve the proof.

4 Case Study: Model Checking

This chapter illustrates the use of model checking to verify the correctness of the mode logic of a single side of the FGS. The FGS mode logic, while quite complex, consists only of Boolean inputs and outputs. This makes it ideally suited for formal verification with a wide range of model checkers, including implicit state BDD-based model checkers such as NuSMV as well as Satisfiability Modulo Theories (SMT)-based model checkers such as Kind. While models consisting only of Boolean logic are well-suited for model checking, most model checkers can also handle models with enumerated types and small integers. SMT-based model checkers such as Kind can also handle models with real numbers if they do not involve nonlinear arithmetic.

The rest of this chapter is organized as follows. Section 4.1 provides an overview of the FGS mode logic. Section 4.2 describes the software verification plan for the mode logic, identifying the life-cycle data items to be produced, the DO-178C objectives to be satisfied, and tool qualification issues. Section 4.3 provides a detailed specification of the mode logic as a MATLAB Simulink/Stateflow model. Section 4.4 discusses the formal verification of the mode logic using the Kind model checker and Simulink Design Verifier™.

4.1 Mode Logic Overview

Modes are defined by Leveson as *mutually exclusive sets of system behaviors* [18]. Specifically as it relates to the FGS, Advisory Circular AC/ACJ 25.1329 defines a mode as *a system configuration that corresponds to a single (or set of) FGS behavior(s)* [5]. In the FGS, the modes are actually abstractions of their associated flight control law and reflect the current state of the flight control law. There are three different types of modes in the FGS mode logic.

The simplest modes (non-arming modes) have only two actual states, *CLEARED* and *SELECTED*, as shown in Figure 76. A mode is said to be *selected* if it has been manually requested by the flight crew or if it has been automatically requested by a subsystem such as the FMS, otherwise it is said to be *cleared*. Non-arming modes become *active* immediately upon selection with its associated flight control law providing guidance commands to the FD and, if engaged, the AP. When cleared, the mode's associated flight control law is non-operational, i.e., it does not generate any outputs.

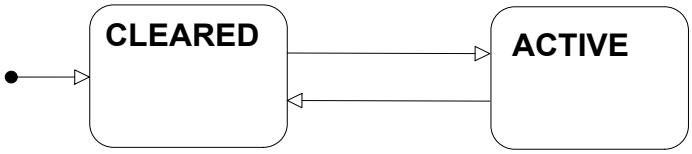

Figure 76 – A Non-Arming Mode

Some modes must first be armed to become active when a capture condition is met, such as the acquisition of a navigation source or proximity to a target reference such as a desired altitude. Such modes have three states as shown in Figure 77. The two states *ARMED* and *ACTIVE* are sub-states of the *SELECTED* state, i.e., when the mode is armed or active, it is also selected. While in the *ARMED* state, the mode's flight control law is not generating guidance commands for the FD or AP, but it may be accepting inputs, accumulating state information, and helping to determine if the capture condition is met. Once the capture condition is met, the mode transitions to the *ACTIVE* state and its flight control law begins generating guidance for the FD and AP. Note that in most arming modes the only way to exit the *ACTIVE* state is to deselect the mode, i.e., it is not usually possible to revert directly from the *ACTIVE* state to the *ARMED* state.

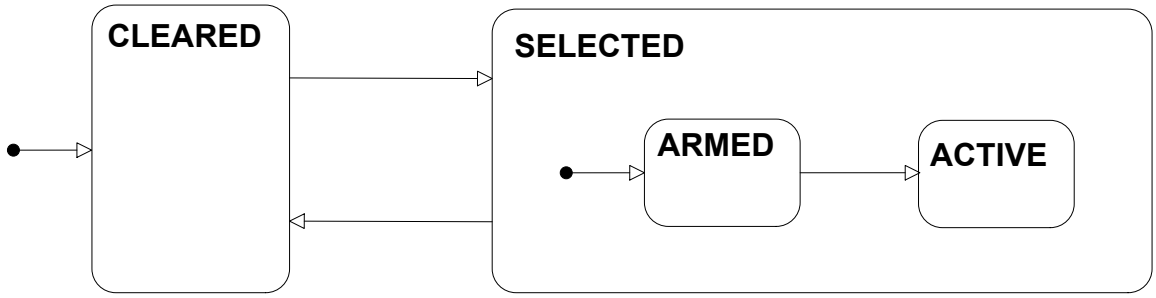

Figure 77 – An Arming Mode

Some modes also distinguish between capturing and tracking the target. Such a mode is shown in Figure 78. Once in the *ACTIVE* state, such a mode's flight control law first captures the target by maneuvering the aircraft to align it with the navigation source or reference. Once correctly aligned, the mode transitions to the tracking state in which it holds the aircraft on the target. Both the *CAPTURE* and *TRACK* states are sub-states of the *ACTIVE* state and the mode's flight control law is active in both states, i.e., generating guidance commands for the FD and AP. Note that the only way to exit the *ACTIVE* state is to deselect the mode, i.e., it is not usually possible to revert directly from the *TRACK* state to the *CAPTURE* state or from the *ACTIVE* state to the *ARMED* state.

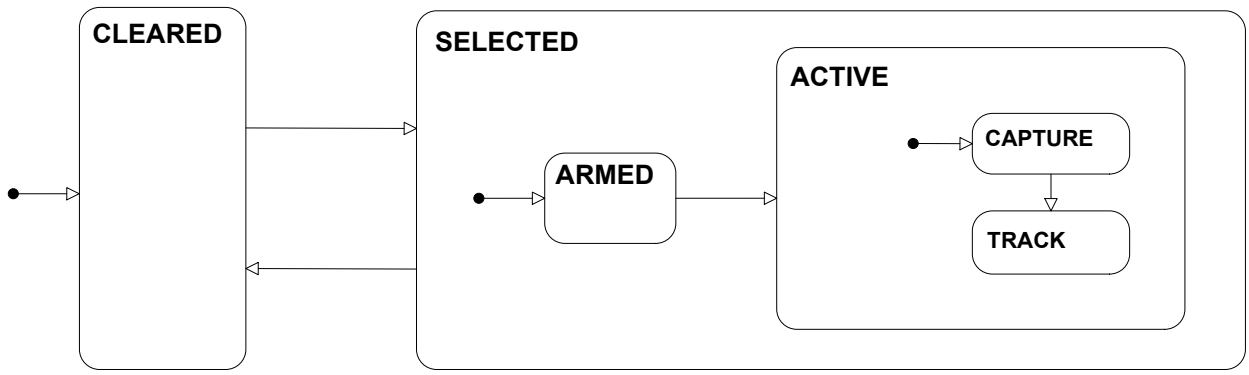

Figure 78 – A Capture/Track Mode

The FGS modes are organized into the lateral modes, which control the behavior of the aircraft about the longitudinal, or roll, axis of the aircraft and the vertical modes, which control the behavior of the aircraft about the lateral, or pitch, axis of the aircraft. The lateral modes in this 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*.

The mode logic of the FGS consists of the specification of 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 FD is displayed or 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 FD and 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. As another example, if the pilot cancels a landing and performs a *go around*, *Lateral* and *Vertical Go Around* modes must both be active.

Still other relationships must be maintained between the mode logic and the surrounding systems. For example, the AP should never be engaged when *Lateral* and *Vertical Go Around* modes are active since the pilot should be manually flying the aircraft during a go around.

While verifying the behavior of a single mode is straightforward, ensuring that all the relationships between the modes and the aircraft are always maintained can be very complex. Moreover, since many of these relationships are of the form "the mode logic shall always ensure relationship x" they cannot be fully verified with testing since testing can only verify a small sample of the total inputs and states. In contrast, model checking is an ideal way to verify these relationships. All of the relationships described above and many others are formally stated and verified in Section 4.4.

4.2 Software Verification Plan

In this case study, we will use model checking 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). Specifically, this case study will verify the low-level software requirements for the mode logic of a side of the FGS and show that the software architecture and the low-level software requirements comply with the high-level software requirements.

4.2.1 Formal Specification and Verification Tools

The low-level software requirements and software architecture will be specified as MATLAB Simulink and Stateflow models. These models will be translated into the Lustre formal specification language using a proprietary Rockwell Collins tool. Lustre is the input language of the JKind SMT-based model checker [14]. JKind is a Java implementation of the Kind model checker developed by the University of Iowa [9].

Kind and JKind make use of SMT (Satisfiability Modulo Theories) solvers and the k-induction inference principle. SMT solvers are tools for determining the satisfiability of logical expressions containing a finite number of terms. A bounded model checking problem (one that considers only a finite number of steps) can be mapped to a satisfiability problem. The k-induction principle is then used to extend the analysis to traces of infinite length.

The high-level software requirements will be specified as Lustre predicates and merged with the Lustre specification of the low-level requirements. The low-level software requirements and software architecture will be shown to comply with the high-level requirements using JKind.

To provide a commercially available example, the MATLAB Design Verifier model checker will also be used to show that the low-level software requirements and the software architecture comply with a subset of the high-level requirements. These high-level requirements will be specified using both the MATLAB Simulink graphical and textual notations.

4.2.2 Life Cycle Data Items

Life cycle data items for this example are specified using a variety of notations and tools.

High-Level Software Requirements The high-level software requirements are specified in two different ways. For verification with the JKind SMT-based model checker, they are specified as predicates in the Lustre formal specification language in the file *Mode_Logic.lustre-props*. For verification with MATLAB Simulink Design Verifier, they are specified using both the Simulink graphical and textual notation in the file *Mode_Logic_Props.mdl*. *Mode_Logic_Props.mdl* also refers to other modeling files, including *Mode_Logic_Props_lib.mdl* which contains Simulink blocks frequently used by *Mode_Logic_Props.mdl*, and *mode_logic_inputs.mat*, *mode_logic_outputs.mat* and *no_higher_event.mat* which contain Simulink bus definitions for the system inputs, outputs and prioritized events.

Low-Level Software Requirements The low-level software requirements are specified as MATLAB Simulink/Stateflow models *Mode_Logic.mdl* and *Mode_Logic_lib.mdl.*

Software Architecture The software architecture is specified as MATLAB Simulink/Stateflow models in the files *Mode_Logic.mdl* and *Mode_Logic_lib.mdl.*

4.2.3 Objectives to Be Satisfied

The DO-178C and DO-333 objectives to be satisfied through model checking are summarized in Table 2. A more detailed discussion of how each objective is satisfied is provided in this section.

Objective A-4.1 – Low-level requirements comply with high-level requirements. This objective is demonstrated by proving with the JKind model checker or Design Verifier that the high-level

software requirements are implemented by the low-level software requirements and the software architecture.

Objective A-4.2 – Low-level requirements are accurate and consistent. This objective is met by modeling the low-level requirements and the software architecture in the executable model language Simulink/Stateflow and by translating into the formal specification language Lustre.

Objective A-4.4 – Low-level requirements are verifiable. This objective is met by modeling the low-level requirements and the software architecture in the executable model language Simulink/Stateflow and by translating that model into the formal specification language Lustre.

Objective A-4.5 – Low-level requirements conform to standards. This objective is partially met by modeling the low-level requirements and the software architecture in the Simulink/Stateflow design language. Commonly used blocks are provided in a library of blocks approved for use on the project. Models can be automatically checked with the MATLAB Model Advisor for conformance with some project defined standards. Conformance to any remaining standards can be shown by manual review of the graphical models.

Objective A-4.6 – Low-level requirements are traceable to high-level requirements. This objective is partially met by proving with the JKind model checker or Design Verifier that the high-level software requirements are implemented by the low-level software requirements and the software architecture, demonstrating that all high-level requirements have been developed into low-level requirements. Demonstrating that all low-level requirements can be traced to high-level requirements is accomplished through manual review.

Objective A-4.7 – Algorithms are accurate. This objective is met by modeling the low-level requirements in Simulink/Stateflow and proving that the high-level software requirements are implemented by the low-level software requirements and the software architecture.

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*

Objective A-4.8 – Software architecture is compatible with the high-level requirements. This objective is demonstrated by proving with the JKind model checker or Design Verifier that the high-level software requirements are implemented by the low-level software requirements and the software architecture.

Objective A-4.9 – Software architecture is consistent. This objective is met by modeling the low-level requirements and the software architecture in the executable modeling language Simulink/Stateflow and by translating that model into the formal specification language Lustre.

Objective A-4.11 – Software architecture is verifiable. This objective is met by modeling the low-level requirements and the software architecture in the executable model language Simulink/Stateflow and by translating that model into the formal specification language Lustre.

Objective A-4.12 – Software architecture conforms to standards. This objective is partially met by modeling the low-level requirements and the software architecture in the Simulink/Stateflow design language. Commonly used blocks are provided in a library of blocks approved for use on the project. Models can be automatically checked with the MATLAB Model Advisor for conformance with some project defined standards. Conformance to any remaining standards can be shown by manual review of the graphical models.

Objective FM.A-4.14 Formal analysis cases and procedures are correct. This objective is met through review to ensure that the analyses and procedures satisfy the objectives A-4.1 through A-4.12 for which credit is claimed. There are no assumptions associated with the Simulink/Stateflow models to be checked. The model contains only Boolean and enumerated types, has only Boolean inputs and outputs, and assumes no constraints on its inputs.

Objective FM.A-4.15 Formal analysis results are correct and discrepancies explained. This objective is met through review to ensure that all formal properties are proven. Many of the properties had to be revised before they could be proved. Typically, these were due to omissions in the original requirements or oversights introduced by the informality of textual requirements. For example, the requirement that "vertical approach mode shall be active only if lateral approach mode is active" could not be implemented with the chosen software architecture and had to be relaxed to allow lateral approach mode to be inactive for a single step before vertical approach mode became inactive. Each such requirement change or discrepancy was explained and fed back into the safety assessment process for review.

Objective FM.A-4.16 Requirements formalization is correct. This objective is met through review to ensure that the formal statement of a requirement is a conservative representation of the informal requirement. In the case where the JKind model checker is used for verification, the translation of the Simulink/Stateflow model to Lustre must either be checked by review or by qualification of the translation tool.

Objective FM.A-4.17 Formal method is correctly defined, justified, and appropriate. This objective is met through a review to ensure:

a. All notations used for formal analysis are verified to have precise, unambiguous, mathematically defined syntax and semantics. The formal notation used was a subset of Simulink/Stateflow that was automatically translated to the Lustre formal specification language.
b. The soundness of each formal analysis method is justified. The JKind model checker is based k-induction, as is the Kind model checker upon which it is based. Soundness of k-induction is straightforward and is discussed in [9] and its references. Since the Design Verifier model checker is a commercial product, less information is available about the underlying methodology. Soundness concerns would have to be addressed by the vendor as part of a qualification support kit.
c. Assumptions related to each formal analysis are described and justified. Since this example contains only Boolean and enumerated types, no assumptions related to the formal analysis (e.g., approximating floating-point numbers as reals) were necessary.

4.2.4 Tool Qualification Issues

As was the case for the theorem proving case study, for the certification objectives and mode of tool use that we are considering in this case study, Criteria 3 applies. This means that for all airborne software levels the model checkers would need to be qualified to TQL-5. Model checking does not (in general) produce independently checkable output. This means that the 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. However, there is no reason that this could not be accomplished following the process outlined above, in a manner similar to that carried out for any other TQL-5 verification tool.

Our use of Kind relied upon the Rockwell Collins translation framework to generate Lustre input from the Simulink/Stateflow model. There are two ways that this might be handled with respect to certification/qualification concerns. The first approach would be to consider the translation tool and the model checker to be a single tool that acts directly on the Simulink/Stateflow model and doesn't directly expose the intermediate Lustre translation. The TORs and qualification test suite would be written to be consistent with this interface. The second approach would be to consider the translation and model checking steps separately. The model checker would be qualified on its own based on the Lustre input language. The translation step would then be treated as part of the Low-Level Requirements formalization process (Objective FM.A-4.16). The objective would be to show that the Lustre output is a conservative representation of the Simulink/Stateflow input model. This could be satisfied either through a manual review of the input and output, or by qualification of the translation tool to automate this function.

For commercial tools like Simulink Design Verifier, some support from the tool vendor may be needed to achieve qualification. As of this time, MathWorks has not provided a qualification kit for the Design Verifier. However, there is no reason in principle, that this could not be done. The general outline of qualification should be similar to that of the Polyspace abstract interpretation tool (described in section 5.2.3 and [19]).

4.3 Specification of the Mode Logic

This section describes the mode logic in detail as a MATLAB Simulink/Stateflow model. The top level Simulink diagram is shown in Figure 79.

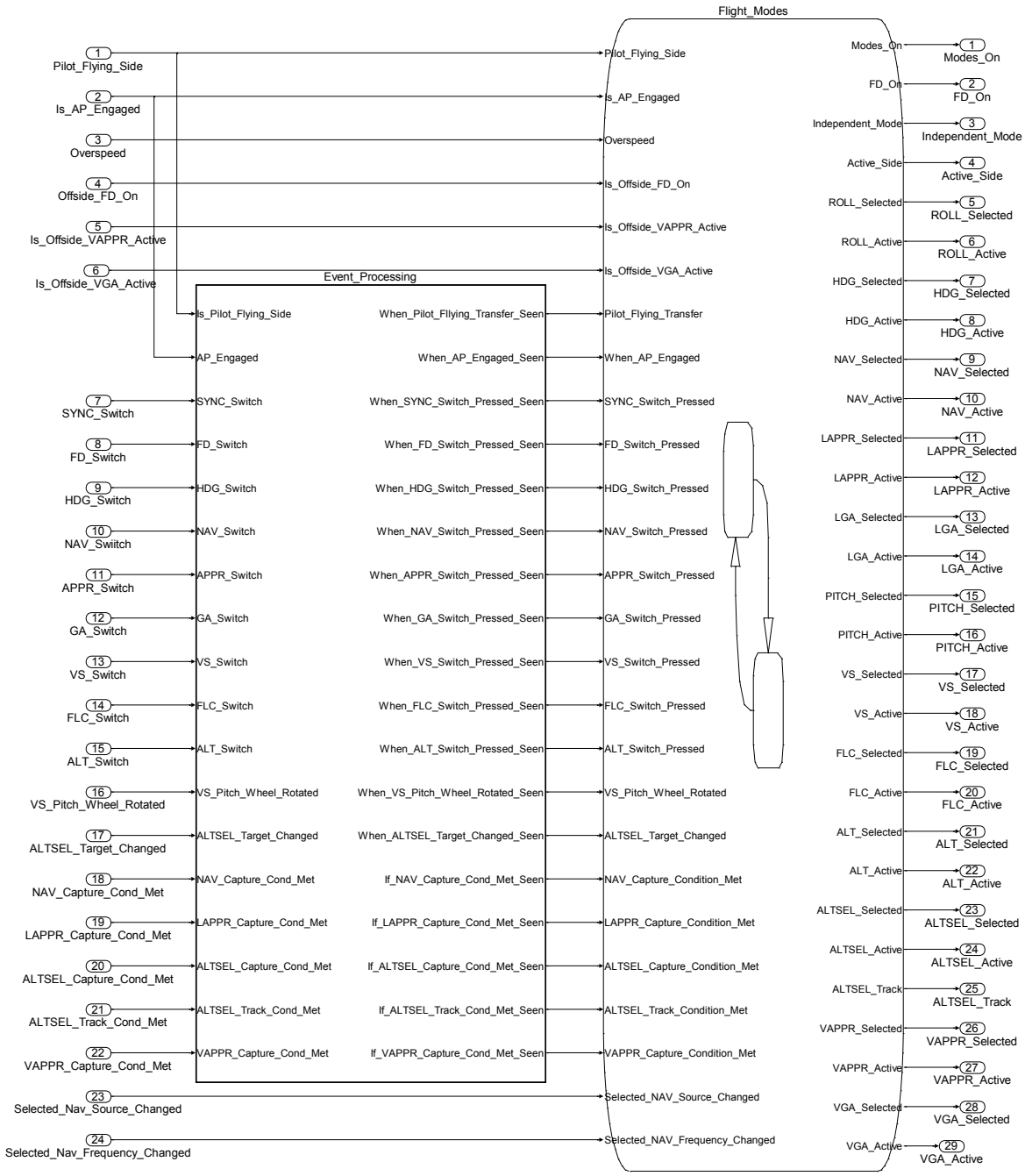

Figure 79 – Mode Logic Top Level

As can be seen in Figure 79, the mode logic takes several Boolean valued inputs and outputs several Boolean values summarizing the status of the system modes. The mode logic has two subsystems, *Event Processing* and *Flight Modes*. *Event Processing* outputs Boolean events (signals that are true for at most one step) and Boolean conditions (signals that can be true for several steps). It establishes a priority among the incoming events and conditions and ensures

that if multiple events or conditions occur on the same step, only the higher priority events and conditions are output to the Flight Modes. Event Processing is discussed in more detail in Section 4.3.2.

4.3.1 Flight Modes

The *Flight Modes* subsystem of Figure 79 is the heart of the mode logic. As shown in Figure 80, it is organized into four parallel state machines, FD, ANNUNCIATIONS, LATERAL, and VERTICAL,

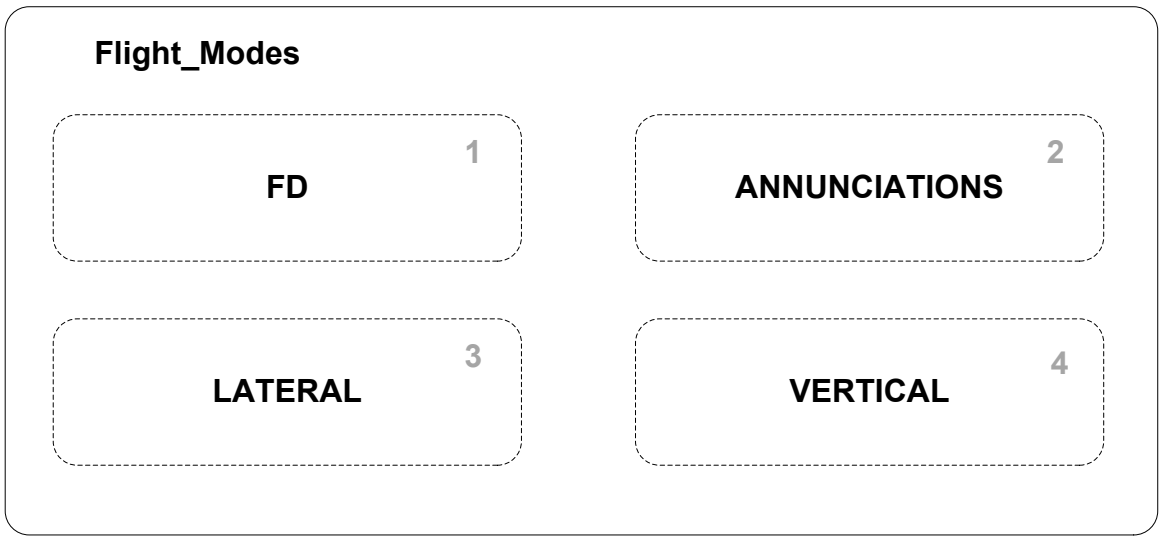

Figure 80 – Flight Modes Subsystem

These four state machines execute in the order indicated by the number in each state machine in Figure 80. The FD state machine determines whether the FD is displayed on the PFD, while the ANNUNCIATIONS state machine determines whether the mode annunciations are displayed on the PFD. The LATERAL and VERTICAL state machines are further decomposed into the state machines for the lateral and vertical modes of the FGS.

4.3.1.1 FD

The FD state machine determines whether the FD associated with this FGS channel is displayed on the PFD. Its logic is shown in Figure 81.

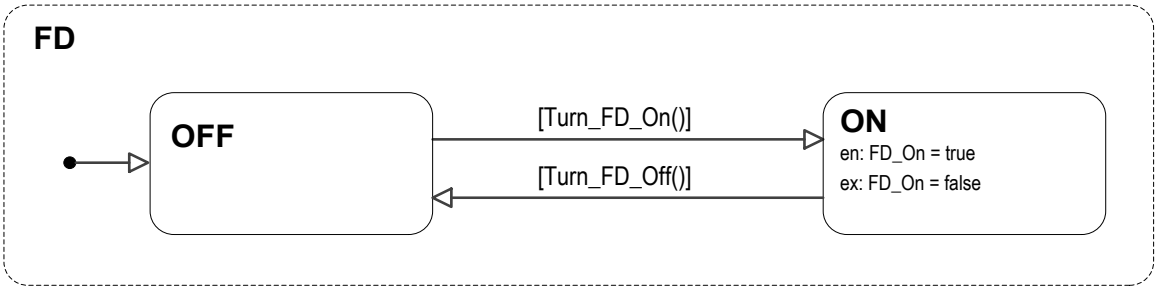

Figure 81 – FD Mode Logic

The FD is either OFF or ON and always starts in the OFF state. Whether the FD transitions from the OFF state to the ON state is determined by the *Turn_FD_On* truth table shown in Table 3. As specified in that table, the FD should be turned on if the FD switch is pressed, the AP is engaged, an overspeed condition exists, a lateral mode is manually selected, a vertical mode is manually selected, or there is a pilot flying transfer to this side of the aircraft while the mode annunciations are on.

Table 3 – Turn FD On

Condition	1	2	3	4	5	6	7
FD_Switch_Pressed	T	-	-	-	-	-	-
When_AP_Engaged	-	T	-	-	-	-	-
Overspeed	-	-	T	-	-	-	-
Lateral_Mode_Manually_Selected()	-	-	-	T	-	-	-
Vertical_Mode_Manually_Selected()	-	-	-	-	T	-	-
Pilot_Flying_Transfer	-	-	-	-	-	T	-
Pilot_Flying_Side	-	-	-	-	-	T	-
Modes_On	-	-	-	-	-	T	-
	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

Note that the FD state machine sets the mode logic output *FD_On* of the mode logic to true when it enters the ON state and to false when it exits the ON state. All of the FGS outputs are turned on and off in similar fashion by other state machines.

The truth table for when a lateral mode is manually selected is shown in Table 4. A lateral mode is manually selected when the HDG, NAV, APPR, or GA switch is pressed.

Table 4 – Lateral Mode Manually Selected

Condition	1	2	3	4	5
HDG_Switch_Pressed	T	-	-	-	-
NAV_Switch_Pressed	-	T	-	-	-
APPR_Switch_Pressed	-	-	T	-	-
GA_Switch_Pressed	-	-	-	T	-
	TRUE	TRUE	TRUE	TRUE	FALSE

The truth table for when a vertical mode is manually selected is shown in Table 5. A vertical mode is manually selected when the VS, FLC, ALT, APPR, or GA switch is pressed or when the VS Pitch Wheel is rotated while Vertical Speed (VS) and Vertical Approach (VAPPR) modes are not active and an overspeed condition does not exist.

Table 5 – Vertical Mode Manually Selected

Condition	1	2	3	4	5	6	7
VS_Switch_Pressed	T	-	-	-	-	-	-
FLC_Switch_Pressed	-	T	-	-	-	-	-
ALT_Switch_Pressed	-	-	T	-	-	-	-
APPR_Switch_Pressed	-	-	-	T	-	-	-
GA_Switch_Pressed	-	-	-	-	T	-	-
VS_Pitch_Wheel_Rotated	-	-	-	-	-	T	-
VS_Active	-	-	-	-	-	F	-
VAPPR_Active	-	-	-	-	-	F	-
Overspeed	-	-	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

The conditions for turning the FD off are show in Table 6. The FD should be turned off when the FD switch is pressed provided an overspeed condition does not exist.

Table 6 – Turn FD Off

Condition	1	2
FD_Switch_Pressed	T	-
Overspeed	F	-
	TRUE	FALSE

4.3.1.2 ANNUNCIATIONS

The ANNUNCIATIONS state machine determines whether the mode annunciations are displayed on the PFD. Its logic is shown in Figure 82.

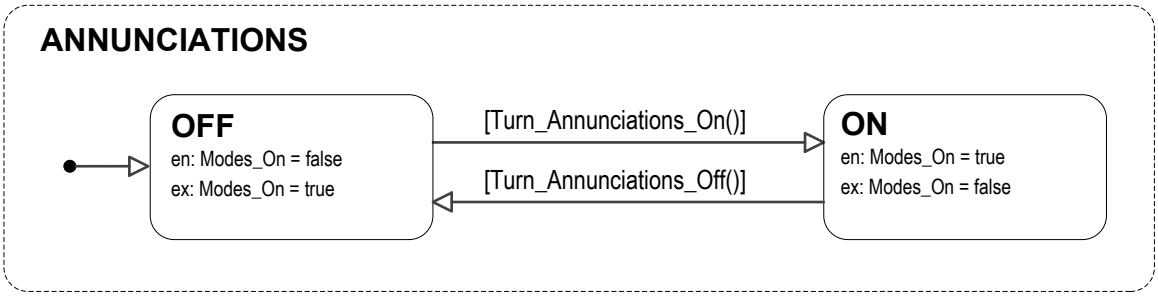

Figure 82 – ANNUNCIATIONS Mode Logic

The mode annunciations are either OFF or ON and always start in the OFF state. Whether the annunciations are displayed is determined by the *Turn_Annunciations_On* truth table shown in Table 7. The mode annunciations are turned on whenever the AP is engaged, the offside FD (i.e., the FD on the other side of the aircraft) is turned on, or the FD on this side of the aircraft is turned on.

Table 7 – Turn Annunciations On

Condition	1	2	3	4
Is_AP_Engaged	T	-	-	-
Is_Offside_FD_On	-	T	-	-
FD_On	-	-	T	-
	TRUE	TRUE	TRUE	FALSE

The logic for turning the mode annunciations off is given in the truth table of Table 8. The mode annunciations are turned off when the AP is not engaged, the offside FD is not displayed, and the onside FD is not displayed.

Table 8 – Turn Annunciations Off

Condition	1	2
Is_AP_Engaged	F	-
Is_Offside_FD_On	F	-
FD_On	F	-

Actions	TRUE	FALSE

4.3.1.3 LATERAL

The lateral modes control the behavior of the aircraft about the longitudinal, or roll, axis. The organization of the LATERAL state machine is shown in Figure 83.

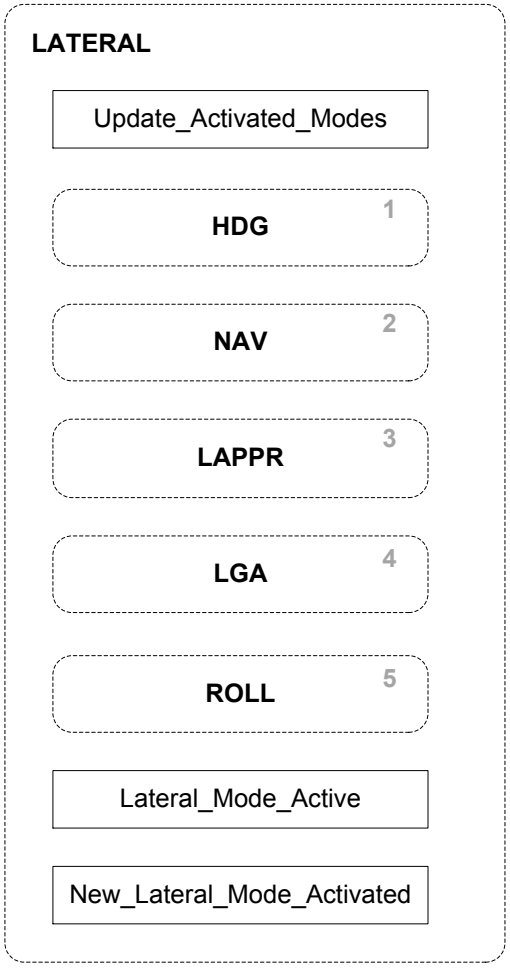

Figure 83 – LATERAL Modes

The individual lateral modes of HDG, NAV, LAPPR, LGA, and ROLL are implemented as parallel state machines that execute in the order shown in Figure 83. Since exactly one lateral mode should be active and providing guidance to the FD and AP at a time, the LATERAL state machine defines two truth tables, *Lateral_Mode_Active* and *New_Lateral_Mode_Activated*, and a function, *Update_Activated_Modes*, to support this synchronization.

The *Lateral_Mode_Active* truth table is used to make the default lateral mode Roll Hold (ROLL) active if no other lateral mode is active and to deactivate ROLL mode when another lateral mode becomes active. Its specification is given in the description of ROLL mode in Section 4.3.1.3.5.

The *New_Lateral_Mode_Activated* truth table is used to deactivate the active lateral mode when a new lateral mode becomes active. Its logic is shown in Table 9.

Table 9 – New Lateral Mode Activated

Condition	1	2	3	4	5
HDG_Will_Be_Activated	T	-	-	-	-
NAV_Will_Be_Activated	-	T	-	-	-
LAPPR_Will_Be_Activated	-	-	T	-	-
LGA_Will_Be_Activated	-	-	-	T	-
	TRUE	TRUE	TRUE	TRUE	FALSE

While the intent of *New_Lateral_Mode_Activated* is straightforward – it should return true if a new lateral mode will be activated during this step - its implementation is actually quite subtle. Invoking it during the execution of the currently active mode to determine if that mode should deactivate may depend on whether a mode that has not yet executed will become active. The recommended way to synchronize parallel state machines in Stateflow is to "wake-up" a mode machine that executed earlier through the use of directed broadcast events. Unfortunately, this leads to a model that is not well-suited for model checking.[5]

Instead, we require that each mode implements a truth table *Will_Be_Activated* that predicts if that mode will become active based on its current state and its inputs. At the start of each step of the LATERAL mode machine, these values are computed and stored in local state variables by the *Update_Activated_Modes* function. These stored values are then used in the *New_Lateral_Mode_Activated* truth table shown in Table 9 so that the current active lateral mode knows whether to deactivate itself regardless of its order of execution.

[5] For model checking, the state transition relation must be "unwound" into a static description of all possible system transitions. Each directed broadcast significantly increases the complexity of this description and the interleaving of all possible sequences of directed broadcasts results in a combinatorial explosion in the size of the state transition relation.

4.3.1.3.1 Heading Select (HDG)

Heading Select (HDG) mode turns the aircraft to the selected heading displayed on the PFD and then holds the aircraft to that heading. It is a non-arming mode that can be selected to become the active lateral mode at any time. Its logic is shown in Figure 84.

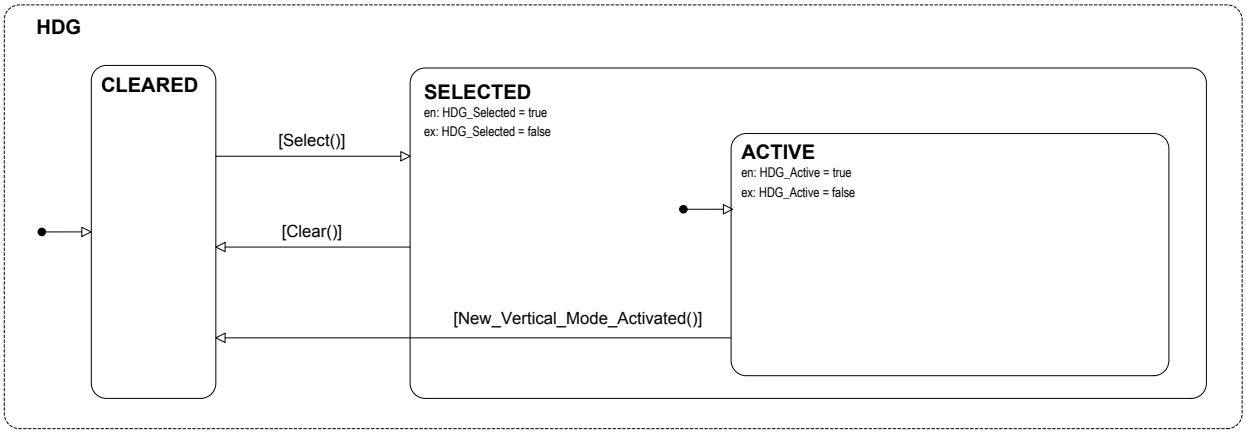

Figure 84 – Heading Select (HDG) Mode

HDG mode starts in the CLEARED state. It transitions into the SELECTED state when the *HDG_Select* truth table shown in Table 10 evaluates to true, i.e., when the HDG switch is pressed. On entry to the SELECTED state it sets the *HDG_Selected* output to true.

Table 10 – HDG Select

Condition	1	2
HDG_Switch_Pressed	T	-
	TRUE	FALSE

Since it is not an arming mode, it immediately transitions into the ACTIVE state. On entry to the ACTIVE state it sets the *HDG_Active* output to true. HDG mode returns to the CLEARED state when the *HDG_Clear* truth table shown in Table 11 evaluates to true, i.e., when the HDG switch is pressed, when there is a pilot flying transfer, or when the mode annunciations are turned off. Note that on exit from the ACTIVE state it sets the *HDG_Active* output to false and on exit from the SELECTED state it sets the *HDG_Selected* output to false.

Table 11 – HDG Clear

Condition	1	2	3	4
HDG_Switch_Pressed	T	-	-	-
Pilot_Flying_Transfer	-	T	-	-
Modes_On	-	-	F	-
	TRUE	TRUE	TRUE	FALSE

HDG mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Lateral_Mode_Activated* is taken). In similar fashion, if HDG mode becomes active, the current active lateral mode must deactivate itself. The *Will_Be_Activated* truth table that supports this synchronization for HDG mode is shown in Table 12.

Table 12 – HDG Will Be Activated

Condition	1	2
in(CLEARED)	T	-
Select()	T	-
	TRUE	FALSE

4.3.1.3.2 Lateral Navigation (NAV)

Lateral Navigation (NAV) mode captures and tracks lateral guidance for en route navigation and non-precision approaches. It is an arming mode that must be armed before it can become active. Its logic is shown in Figure 85.

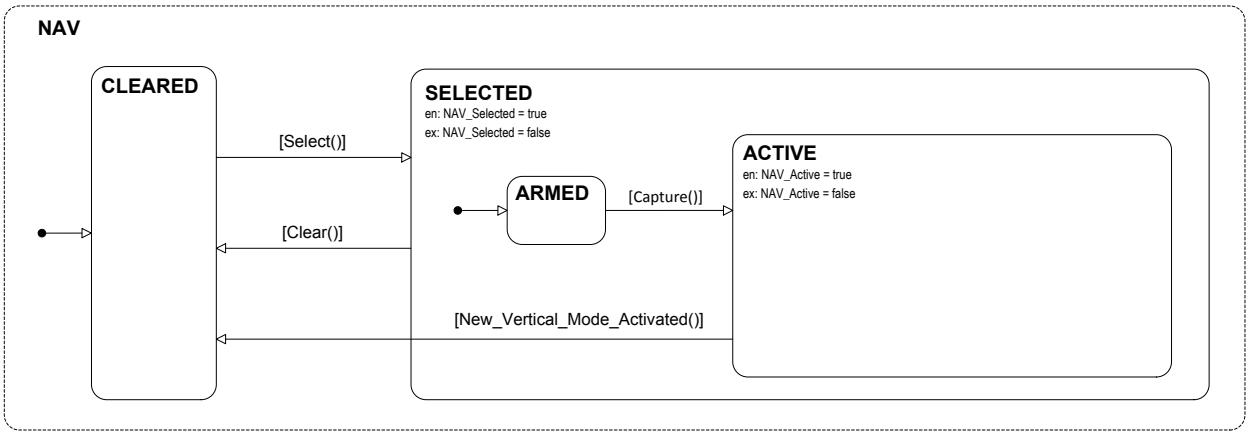

Figure 85 – Lateral Navigation (NAV) Mode

NAV mode starts in the CLEARED state. It transitions into the SELECTED state when the *NAV_Select* truth table shown in Table 13 evaluates to true, i.e., when the NAV switch is pressed.

Table 13 – NAV Select

Condition	1	2
NAV_Switch_Pressed	T	-
	TRUE	FALSE

Since it is an arming mode, it then enters the ARMED state. From the ARMED state it will transition to the ACTIVE state when the *NAV_Capture* truth table shown in Table 14 evaluates to true, i.e. when the *NAV_Capture_Condition_Met* input is true.

Table 14 – NAV Capture

Condition	1	2
NAV_Capture_Condition_Met	T	-
	TRUE	FALSE

NAV mode returns to the CLEARED state when the *NAV_Clear* truth table shown in Table 15 evaluates to true, i.e., when the NAV switch is pressed, when the navigation source or frequency changes, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 15 – NAV Clear

Condition	1	2	3	4	5	6
NAV_Switch_Pressed	T	-	-	-	-	-
Selected_NAV_Source_Changed	-	T	-	-	-	-
Selected_NAV_Frequency_Changed	-	-	T	-	-	-
Pilot_Flying_Transfer	-	-	-	T	-	-
Modes_On	-	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

NAV mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Lateral_Mode_Activated* is taken). Note that it is not possible to directly transition back to the ARMED state from the ACTIVE state. The *Will_Be_Activated* truth table for NAV mode is shown in Table 16.

Table 16 – NAV Will Be Activated

Condition	1	2
in(SELECTED.ARMED)	T	-
Capture()	T	-
Clear()	F	-
	TRUE	FALSE

Note that since NAV is an arming mode, Table 16 must guard against the situation in which NAV mode receives a command to clear itself at the same time that its capture condition becomes true.

4.3.1.3.3 Lateral Approach (LAPPR)

Lateral Approach (LAPPR) mode captures and tracks lateral guidance for precision and non-precision approaches. It is an arming mode that must be armed before it can become active. Its logic is shown in Figure 86.

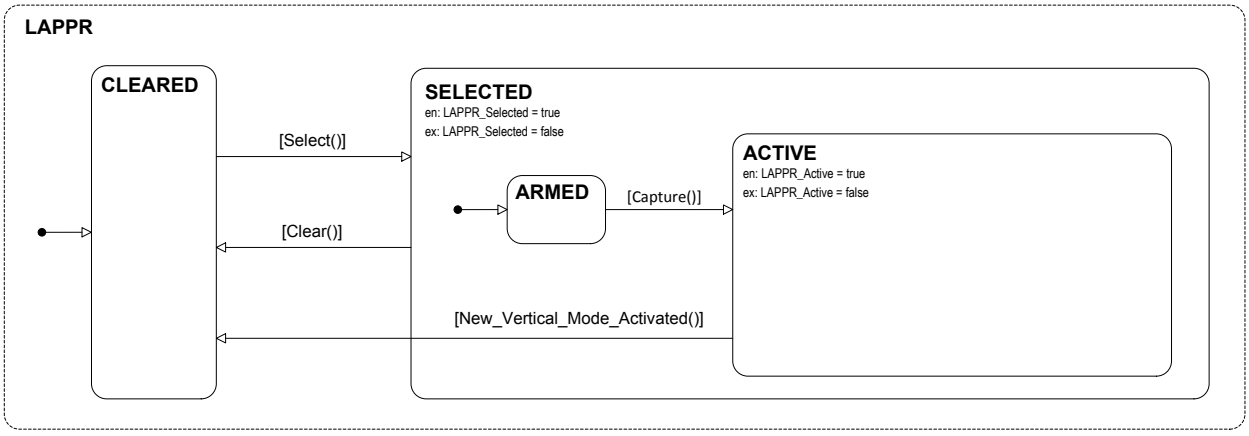

Figure 86 – Lateral Approach (LAPPR) Mode

LAPPR mode starts in the CLEARED state. It transitions into the SELECTED state when the *LAPPR_Select* truth table shown in Table 17 evaluates to true, i.e., when the APPR switch is pressed.

Table 17 – LAPPR Select

Condition	1	2
APPR_Switch_Pressed	T	-
	TRUE	FALSE

Since it is an arming mode, it then enters the ARMED state. From the ARMED state it will transition to the ACTIVE state when the *LAPPR_Capture* truth table shown in Table 18 evaluates to true, i.e. when the *LAPPR_Capture_Condition_Met* input is true.

Table 18 – LAPPR Capture

Condition	1	2
LAPPR_Capture_Condition_Met	T	-
	TRUE	FALSE

LAPPR mode returns to the CLEARED state when the *LAPPR_Clear* truth table shown in Table 19 evaluates to true, i.e., when the APPR switch is pressed, when the navigation source or frequency is changed, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 19 – LAPPR Clear

Condition	1	2	3	4	5	6
APPR_Switch_Pressed	T	-	-	-	-	-
Selected_NAV_Source_Changed	-	T	-	-	-	-
Selected_NAV_Frequency_Changed	-	-	T	-	-	-
Pilot_Flying_Transfer	-	-	-	T	-	-
Modes_On	-	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

LAPPR mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Lateral_Mode_Activated* is taken). Note that it is not possible to directly transition back to the ARMED state from the ACTIVE state. The *Will_Be_Activated* truth table for LAPPR mode is shown in Table 20.

Table 20 – LAPPR Will Be Activated

Condition	1	2
in(SELECTED.ARMED)	T	-
Capture()	T	-
Clear()	F	-
	TRUE	FALSE

4.3.1.3.4 Lateral Go Around (LGA)

Lateral Go Around (LGA) mode maintains the current heading when the pilot aborts a landing. It is a non-arming mode that can become the active lateral mode at any time. Its logic is shown in Figure 87.

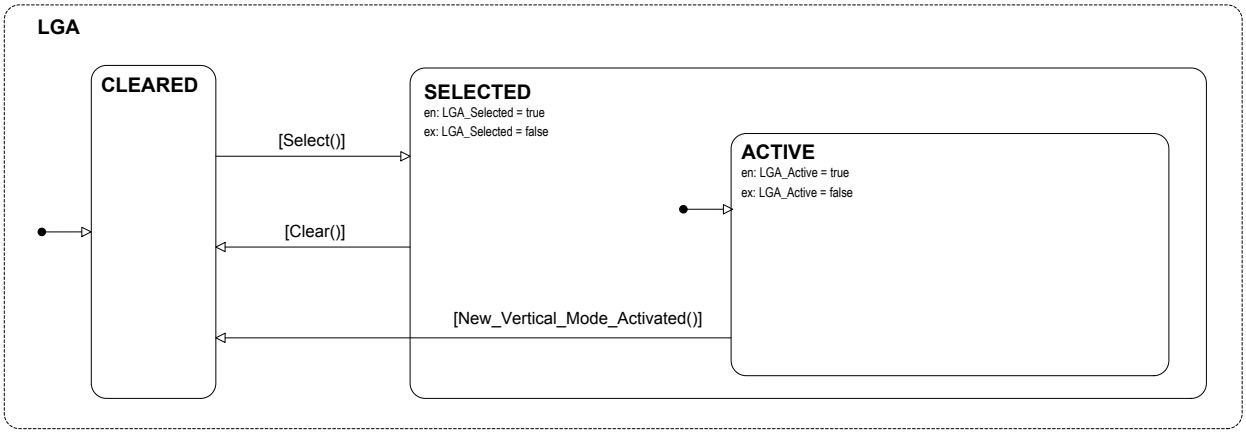

Figure 87 – Lateral Go Around (LGA) Mode

LGA mode starts in the CLEARED state. It transitions into the SELECTED state when the *LGA_Select* truth table shown in Table 21 evaluates to true, i.e., when the GA switch is pressed while an overspeed condition does not exist.

Table 21 – LGA Select

Condition	1	2
GA_Switch_Pressed	T	-
Overspeed	F	-
	TRUE	FALSE

Since it is not an arming mode, it immediately transitions into the ACTIVE state. LGA mode returns to the CLEARED state when the *LGA_Clear* truth table shown in Table 22 evaluates to true, i.e., when the AP is engaged, when the SYNC switch is pressed, when Vertical Go Around (VGA) mode becomes inactive, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 22 – LGA Clear

Condition	1	2	3	4	5	6
When_AP_Engaged	T	-	-	-	-	-
SYNC_Switch_Pressed	-	T	-	-	-	-
VGA_Active	-	-	F	-	-	-
Pilot_Flying_Transfer	-	-	-	T	-	-
Modes_On	-	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

LGA mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Lateral_Mode_Activated* is taken). The *Will_Be_Activated* truth table for LGA mode is shown in Table 23.

Table 23 – LGA Will Be Activated

Condition	1	2
in(CLEARED)	T	-
Select()	T	-
	TRUE	FALSE

4.3.1.3.5 Roll Hold (ROLL)

Roll Hold (ROLL) mode holds the aircraft at the fixed bank angle it is in when the mode becomes active or when the SYNC switch is pressed. ROLL mode is the basic lateral mode and is always active when no other lateral mode is active and the mode annunciations are on. Since it may need to become active at any time, it a non-arming mode. Its logic is shown in Figure 88.

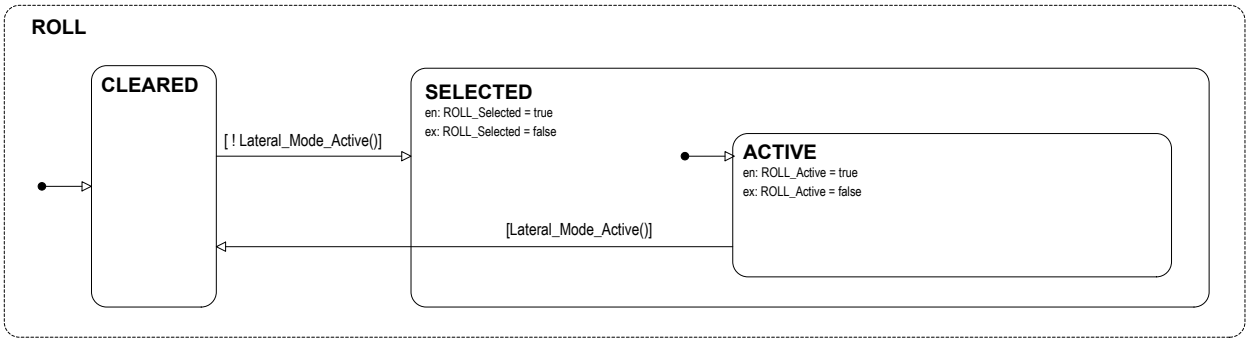

Figure 88 – Roll Hold (ROLL)

ROLL mode starts in the Active state. It transitions into the CLEARED state when the Lateral Mode Active truth table shown in Table 24 evaluates to true, i.e., when another lateral mode is

active. ROLL mode transitions back to the ACTIVE state when no other lateral mode is active, i.e. when *Lateral_Mode_Active* evaluates to false.

Table 24 – Lateral Mode Active

Condition	1	2	3	4	5
HDG_Active	T	-	-	-	-
NAV_Active	-	T	-	-	-
LAPPR_Active	-	-	T	-	-
LGA_Active	-	-	-	T	-
	TRUE	TRUE	TRUE	TRUE	FALSE

4.3.1.4 VERTICAL

The vertical modes control the behavior of the aircraft about the lateral, or pitch, axis. The organization of the VERTICAL state machine is shown in Figure 89.

The individual vertical modes of VS, FLC, ALT, ALTSEL, VAPPR, VGA and PITCH are implemented as parallel state machines that execute in that order. Just as with the lateral modes, the VERTICAL state machine defines two truth tables, *Vertical_Mode_Active* and *New_Vertical_Mode_Activated*, and a function, *Update_Activated_Modes*, to support the synchronization between the vertical modes.

The *Vertical_Mode_Active* truth table is used to make the default vertical mode Pitch Hold (PITCH) active if no other vertical mode is active and to deactivate PITCH mode when another vertical mode becomes active. Its specification is given in the description of PITCH mode in Section 4.3.1.4.7.

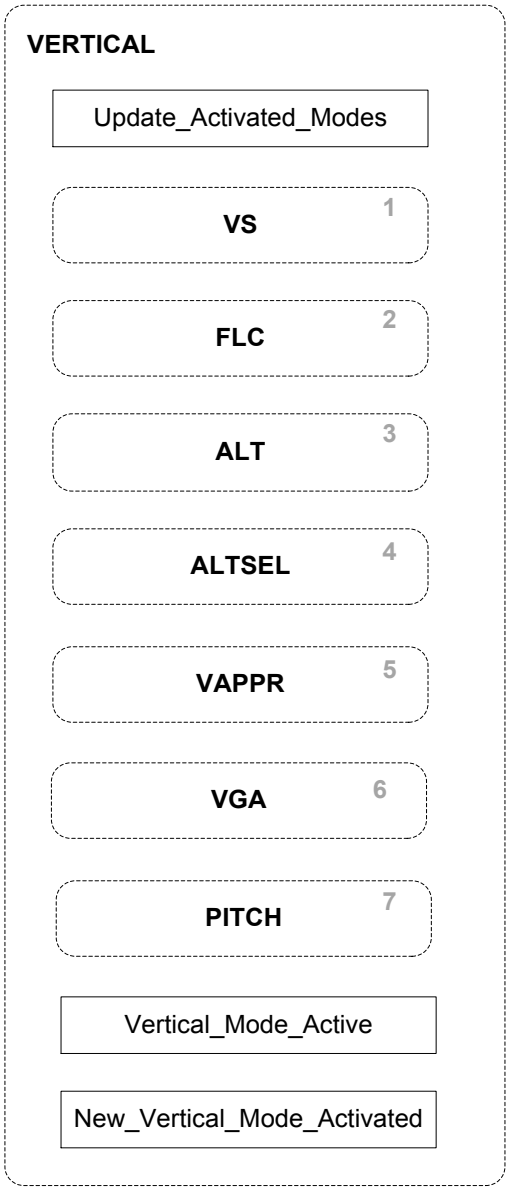

Figure 89 – VERTICAL Modes

The *New_Vertical_Mode_Activated* truth table is used to deactivate the current active vertical mode when a new vertical mode becomes active. Its logic is shown in Table 25.

Table 25 – New Vertical Mode Activated

Condition	1	2	3	4	5	6	7
VS_Will_Be_Activated	T	-	-	-	-	-	-
FLC_Will_Be_Activated	-	T	-	-	-	-	-
ALT_Will_Be_Activated	-	-	T	-	-	-	-
ALTSEL_Will_Be_Activated	-	-	-	T	-	-	-
VAPPR_Will_Be_Activated	-	-	-	-	T	-	-
VGA_Will_Be_Activated	-	-	-	-	-	T	-
	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

As with the lateral modes, we require that each vertical mode implements a truth table *Will_Be_Activated* that predicts if that mode will become active based on its current state and its inputs. At the start of each step of the VERTICAL mode machine, these values are computed and stored in local state variables by the *Update_Activated_Modes* function. These stored values are then used in the *New_Vertical_Mode_Activated* truth table shown in Table 25 so that the current active vertical mode knows whether to deactivate itself if another vertical mode will become active.

4.3.1.4.1 Vertical Speed (VS)

Vertical Speed (VS) mode holds the aircraft to the Vertical Speed (VS) reference displayed on the PFD. It is a non-arming mode that can become active at any time. Its logic is shown in Figure 90.

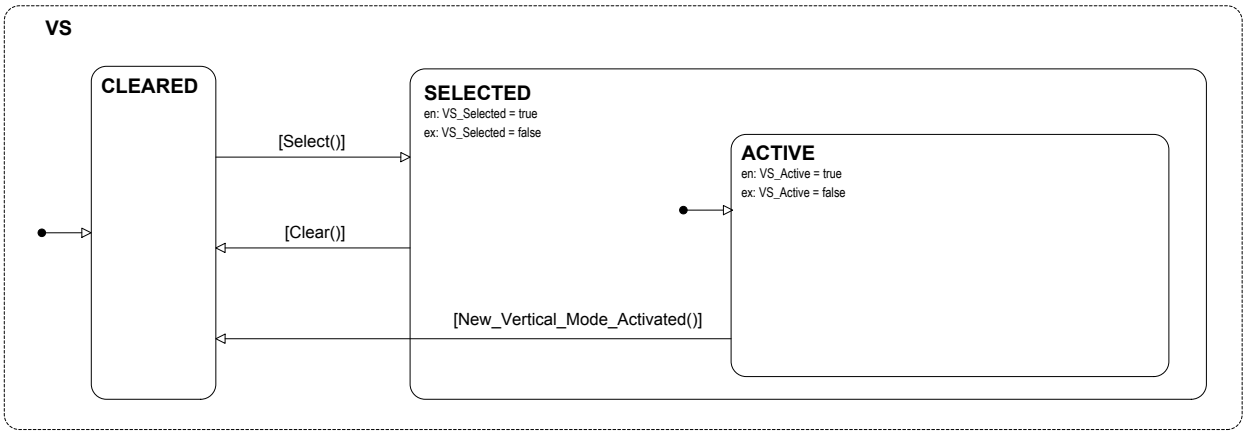

Figure 90 – Vertical Speed (VS) Mode

VS mode starts in the CLEARED state. It transitions into the SELECTED state when the *VS_Select* truth table shown in Table 26 evaluates to true, i.e., when the VS switch is pressed while an overspeed condition does not exist and Vertical Approach (VAPPR) mode is not active.

Table 26 – VS Select

Condition	1	2
VS_Switch_Pressed	T	-
Overspeed	F	-
VAPPR_Active	F	-
	TRUE	FALSE

Since it is not an arming mode, it immediately transitions into the ACTIVE state. VS mode returns to the CLEARED state when the *VS_Clear* truth table shown in Table 27 evaluates to true, i.e., when the VS switch is pressed, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 27 – VS Clear

Condition	1	2	3	4
VS_Switch_Pressed	T	-	-	-
Pilot_Flying_Transfer	-	T	-	-
Modes_On	-	-	F	-
	TRUE	TRUE	TRUE	FALSE

VS mode will also transition to the CLEARED state if another vertical mode becomes active on this step (i.e., the transition guarded by *New_Vertical_Mode_Activated* is taken). The *Will_Be_Activated* truth table for VS is shown in Table 28.

Table 28 – VS Will Be Activated

Condition	1	2
in(CLEARED)	T	-
Select()	T	-
	TRUE	FALSE

4.3.1.4.2 Flight Level Change (FLC)

Flight Level Change (FLC) mode acquires and tracks an Indicated Airspeed (IAS) or Mach Reference Airspeed while also climbing or descending to bring the aircraft to the Preselected Altitude. It is a non-arming mode that can become active at any time. Its logic is shown in Figure 91.

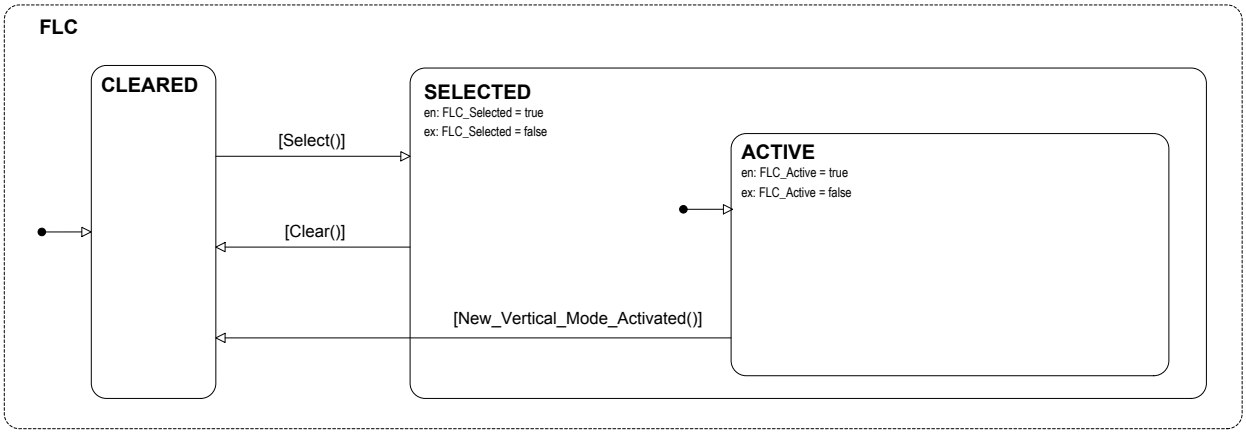

Figure 91 – Flight Level Change (FLC) Mode

FLC mode starts in the CLEARED state. It transitions into the SELECTED state when the *FLC_Select* truth table shown in Table 29 evaluates to true, i.e., when the FLC switch is pressed while Vertical Approach (VAPPR) mode is not active, or when an overspeed condition exists while Altitude Hold (ALT) mode and Altitude Select (ALTSEL) mode are not active and are not about to become active.

Table 29 – FLC Select

Condition	1	2	3
FLC_Switch_Pressed	T	-	-
VAPPR_Active	F	-	-
Overspeed	-	T	-
ALT_Active	-	F	-
ALT_Will_Be_Activated	-	F	-
ALTSEL_Active	-	F	-
ALTSEL_Will_Be_Activated	-	F	-
	TRUE	TRUE	FALSE

Since it is not an arming mode, it immediately transitions into the ACTIVE state. FLC mode returns to the CLEARED state when the *FLC_Clear* truth table shown in Table 30 evaluates to

true, i.e., when the FLC switch is pressed while there is not an overspeed condition, when the VS Pitch Wheel is rotated while there is not an overspeed condition, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 30 – FLC Clear

Condition	1	2	3	4	5
FLC_Switch_Pressed	T	-	-	-	-
Overspeed	F	F	-	-	-
VS_Pitch_Wheel_Rotated	-	T	-	-	-
Pilot_Flying_Transfer	-	-	T	-	-
Modes_On	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	FALSE

FLC mode will also transition to the CLEARED state if another vertical mode becomes active on this step (i.e., the transition guarded by *New_Vertical_Mode_Activated* is taken). The *Will_Be_Activated* truth table for FLC is shown in Table 31.

Table 31 – FLC Will Be Activated

Condition	1	2
in(CLEARED)	T	-
Select()	T	-
	TRUE	FALSE

4.3.1.4.3 Altitude Hold (ALT)

Altitude Hold (ALT) mode acquires and tracks the altitude reference, which is set to the current altitude when the mode is activated or upon a SYNC request by the flight crew. It is a non-arming mode that can become active at any time. Its logic is shown in Figure 92.

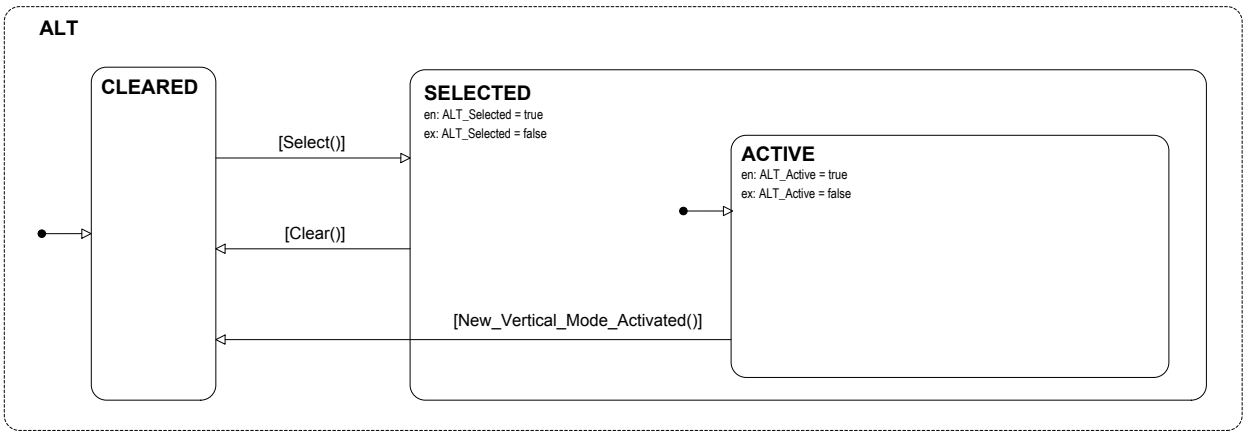

Figure 92 – Altitude Hold (ALT) Mode

ALT mode starts in the CLEARED state. It transitions into the SELECTED state when the *ALT_Select* truth table shown in Table 32 evaluates to true, i.e., when the ALT switch is pressed while Vertical Approach (VAPPR) mode is not active or when the Altitude Select (ALTSEL) target altitude is changed while ALTSEL mode is in the TRACK state and VAPPR mode is not active.

Table 32 – ALT Select

Condition	1	2	3
ALT_Switch_Pressed	T	-	-
VAPPR_Active	F	F	-
ALTSEL_Target_Changed	-	T	-
ALTSEL_Track	-	T	-
	TRUE	TRUE	FALSE

Since it is not an arming mode, it immediately transitions into the ACTIVE state. ALT mode returns to the CLEARED state when the *ALT_Clear* truth table shown in Table 33 evaluates to true, i.e., when the ALT switch is pressed, when the VS Pitch Wheel is rotated, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 33 – ALT Clear

Condition	1	2	3	4	5
ALT_Switch_Pressed	T	-	-	-	-
VS_Pitch_Wheel_Rotated	-	T	-	-	-
Pilot_Flying_Transfer	-	-	T	-	-
Modes_On	-	-	-	F	-
Actions	TRUE	TRUE	TRUE	TRUE	FALSE

ALT mode will also transition to the CLEARED state if another vertical mode becomes active on this step (i.e., the transition guarded by *New_Vertical_Mode_Activated* is taken). The *Will_Be_Activated* truth table for ALT is shown in Table 34.

Table 34 – ALT Will Be Activated

Condition	1	2
in(CLEARED)	T	-
Select()	T	-
	TRUE	FALSE

4.3.1.4.4 Altitude Select (ALTSEL)

Altitude Select (ALTSEL) mode captures and tracks the Preselected Altitude. It is a capture/track mode that must be armed before it can become active and that has both capture and track sub-states of its active state. Its logic is shown in Figure 93.

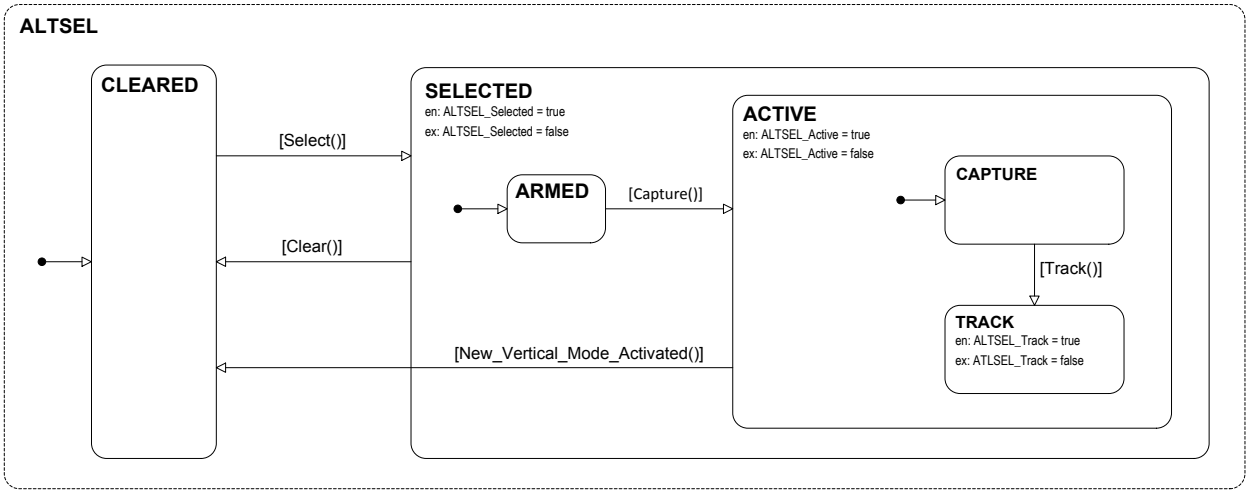

Figure 93 – Altitude Select (ALTSEL) Mode

ALTSEL starts in the CLEARED state. It transitions to the SELECTED state when the *ALTSEL_Select* truth table shown in Table 35 evaluates to true, i.e., when none of Vertical Approach (VAPPR), Vertical Go Around (VGA), or Altitude Hold (ALT) mode are active.

Table 35 – ALTSEL Select

Condition	1	2
VAPPR_Active	F	-
VGA_Active	F	-
ALT_Active	F	-
	TRUE	FALSE

It also immediately enters the ARMED state in which the FGS monitors the aircraft closure rate towards the target altitude and determines the optimum capture point to transition to the capture state. When the ALTSEL capture condition is met (Table 36) it enters the ACTIVE state.

Table 36 – ALTSEL Capture

Condition	1	2
ALTSEL_Capture_Condition_Met	T	-
Actions	TRUE	FALSE

It also immediately enters the CAPTURE state in which the FGS generates vertical guidance commands to perform a smooth capture of the target altitude. Once the target altitude is reached and the ALTSEL track condition is met (Table 37) it transitions to the TRACK state.

Table 37 – ALTSEL Track

Condition	1	2
ALTSEL_Track_Condition_Met	T	-
	TRUE	FALSE

ALTSEL mode returns to the CLEARED state when the *ALTSEL_Clear* truth table shown in Table 38 evaluates to true, i.e., when one of Vertical Approach (VAPPR), Vertical Go Around (VGA), or Altitude Hold (ALT) modes becomes active, or when the mode annunciations are turned off.

Table 38 – ALTSEL Clear

Condition	1	2	3	4	5
VAPPR_Active	T	-	-	-	-
VGA_Active	-	T	-	-	-
ALT_Active	-	-	T	-	-
Modes_On	-	-	-	F	-
Actions	TRUE	TRUE	TRUE	TRUE	FALSE

ALTSEL mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Vertical_Mode_Activated* is taken). The *Will_Be_Activated* truth table for ALTSEL mode is shown in Table 39.

Table 39 – ALTSEL Will Be Activated

Condition	1	2
in(SELECTED.ARMED)	T	-
Capture()	T	-
Clear()	F	-
	TRUE	FALSE

4.3.1.4.5 Vertical Approach (VAPPR)

Vertical Approach (VAPPR) mode captures and tracks the vertical guidance for Instrument Landing System (ILS) precision glideslope approaches. It is an arming mode that must be armed before it can become active. Its logic is shown in Figure 94.

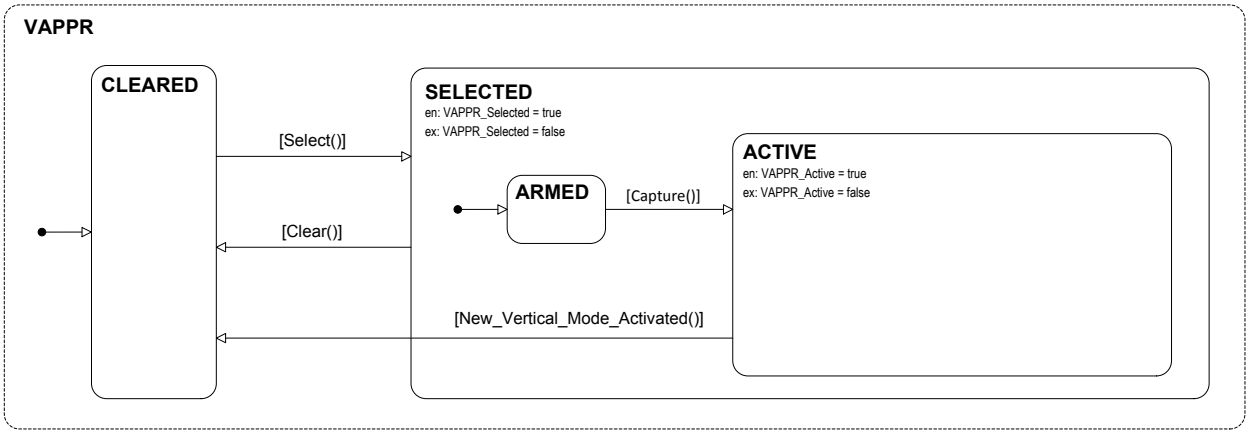

Figure 94 – Vertical Approach (VAPPR) Mode

VAPPR mode starts in the CLEARED state. It transitions into the SELECTED state when the *VAPPR_Select* truth table shown in Table 40 evaluates to true, i.e., when the APPR switch is pressed.

Table 40 – VAPPR Select

Condition	1	2
APPR_Switch_Pressed	T	-
	TRUE	FALSE

It also immediately enters the ARMED state. From the ARMED state it will transition to the ACTIVE state when the VAPPR Capture truth table shown in Table 41 evaluates to true, i.e., when the vertical approach capture condition is met and Lateral Approach (LAPPR) mode is active and an overspeed condition does not exist.

Table 41 – VAPPR Capture

Condition	1	2
VAPPR_Capture_Condition_Met	T	-
LAPPR_Active	T	-
Overspeed	F	-
	TRUE	FALSE

VAPPR mode returns to the CLEARED state when the *VAPPR_Clear* truth table shown in Table 42 evaluates to true, i.e., when the APPR switch is pressed, when Lateral Approach (LAPPR) mode is not selected, when the navigation source or frequency is changed, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 42 – VAPPR Clear

Condition	1	2	3	4	5	6	7
APPR_Switch_Pressed	T	-	-	-	-	-	-
LAPPR_Selected	-	F	-	-	-	-	-
Selected_NAV_Source_Changed	-	-	T	-	-	-	-
Selected_NAV_Frequency_Changed	-	-	-	T	-	-	-
Pilot_Flying_Transfer	-	-	-	-	T	-	-
Modes_On	-	-	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

VAPPR mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Vertical_Mode_Activated* is taken). The *Will_Be_Activated* truth table for VAPPR mode is shown in Table 43.

Table 43 – VAPPR Will Be Activated

Condition	1	2
in(SELECTED.ARMED)	T	-
Capture()	T	-
Clear()	F	-
	TRUE	FALSE

4.3.1.4.6 Vertical Go Around (VGA)

Vertical Go Around (VGA) mode maintains a fixed pitch angle when the pilot aborts a landing. It is a non-arming mode that can become the active vertical mode at any time. Its logic is shown in Figure 95.

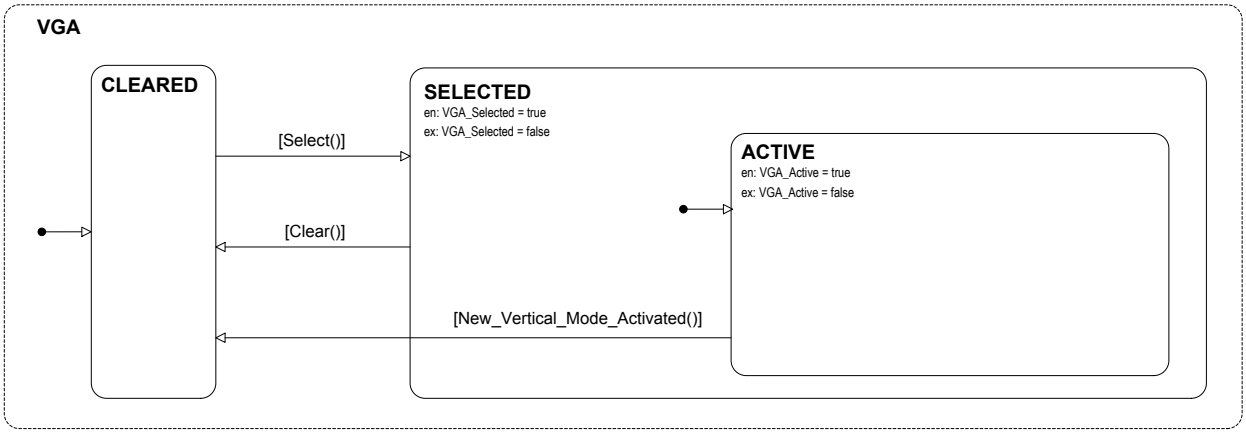

Figure 95 – Vertical Go Around (VGA) Mode

VGA mode starts in the CLEARED state. It transitions into the SELECTED state when the *VGA_Select* truth table shown in Table 44 evaluates to true, i.e., when the GA switch is pressed while an overspeed condition does not exist.

Table 44 – VGA Select

Condition	1	2
GA_Switch_Pressed	T	-
Overspeed	F	-
Actions	TRUE	FALSE

It immediately transitions into the ACTIVE state. VGA mode returns to the CLEARED state when the *VGA_Clear* truth table shown in Table 45 evaluates to true, i.e., when the AP is engaged, when the SYNC switch is pressed, when the VS Pitch Wheel is rotated, when there is a pilot flying transfer, or when the mode annunciations are turned off.

Table 45 – VGA Clear

Condition	1	2	3	4	5	6
When_AP_Engaged	T	-	-	-	-	-
SYNC_Switch_Pressed	-	T	-	-	-	-
VS_Pitch_Wheel_Rotated	-	-	T	-	-	-
Pilot_Flying_Transfer	-	-	-	T	-	-
Modes_On	-	-	-	-	F	-
	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

LGA mode will also transition to the CLEARED state if another mode becomes active on this step (i.e., the transition guarded by *New_Vertical_Mode_Activated* is taken). The *Will_Be_Activated* truth table for LGA mode is shown in Table 23.

Table 46 – LGA Will Be Activated

Condition	1	2
in(CLEARED)	T	-
Select()	T	-
	TRUE	FALSE

4.3.1.4.7 Pitch Hold (PITCH)

Pitch Hold (PITCH) mode holds the aircraft at the fixed pitch angle it is in when the mode becomes active or when the SYNC switch is pressed. PITCH is the basic vertical mode and is always active when no other vertical mode is active. Since it may need to become active at any time, it is a non-arming mode. Its mode logic is shown in Figure 96.

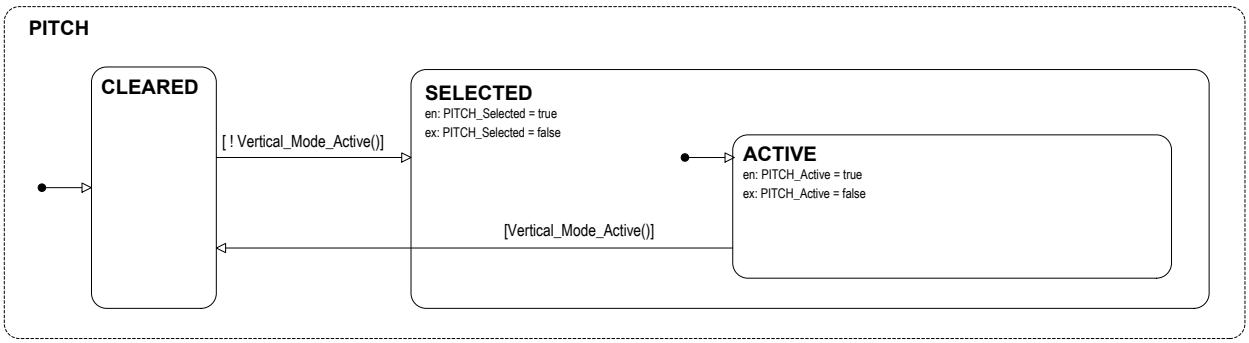

Figure 96 – Pitch Hold (PITCH) Mode

PITCH mode starts in the Active sub-state of the SELECTED state. It transitions into the CLEARED state when the *Vertical_Mode_Active* truth table shown in Table 47 evaluates to true, i.e., when another vertical mode is active. PITCH mode transitions back to the ACTIVE state when no other vertical mode is active, i.e. when *Vertical_Mode_Active* evaluates to false.

Table 47 – Vertical Mode Active

Condition	1	2	3	4	5	6	7
VS_Active	T	-	-	-	-	-	-
FLC_Active	-	T	-	-	-	-	-
ALT_Active	-	-	T	-	-	-	-
ALTSEL_Active	-	-	-	T	-	-	-
VAPPR_Active	-	-	-	-	T	-	-
VGA_Active	-	-	-	-	-	T	-
	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

4.3.2 Event Processing

While the Flight Modes subsystem of Section 4.3.1 constitutes most of the mode logic, the Event Processing subsystem also plays an important role. The LATERAL and VERTICAL state diagrams ensure that there is always at least one lateral and one vertical mode active, and that the current active lateral or vertical mode is always deactivated if a new mode becomes active. However, they do not ensure that only one lateral and one vertical mode is active at the same time. In fact, without Event Processing there are several situations in which more than one lateral or vertical mode can become active. The next section describes the approach taken here - prioritization of input events - to keep this from happening. Other approaches and the reasons for not selecting them are discussed in Section 4.3.2.1.

4.3.2.1 Event Prioritization

The logic of the Event Processing subsystem is shown in Figure 97. Event processing establishes a ranking of input events so that higher priority events supersede lower priority events.

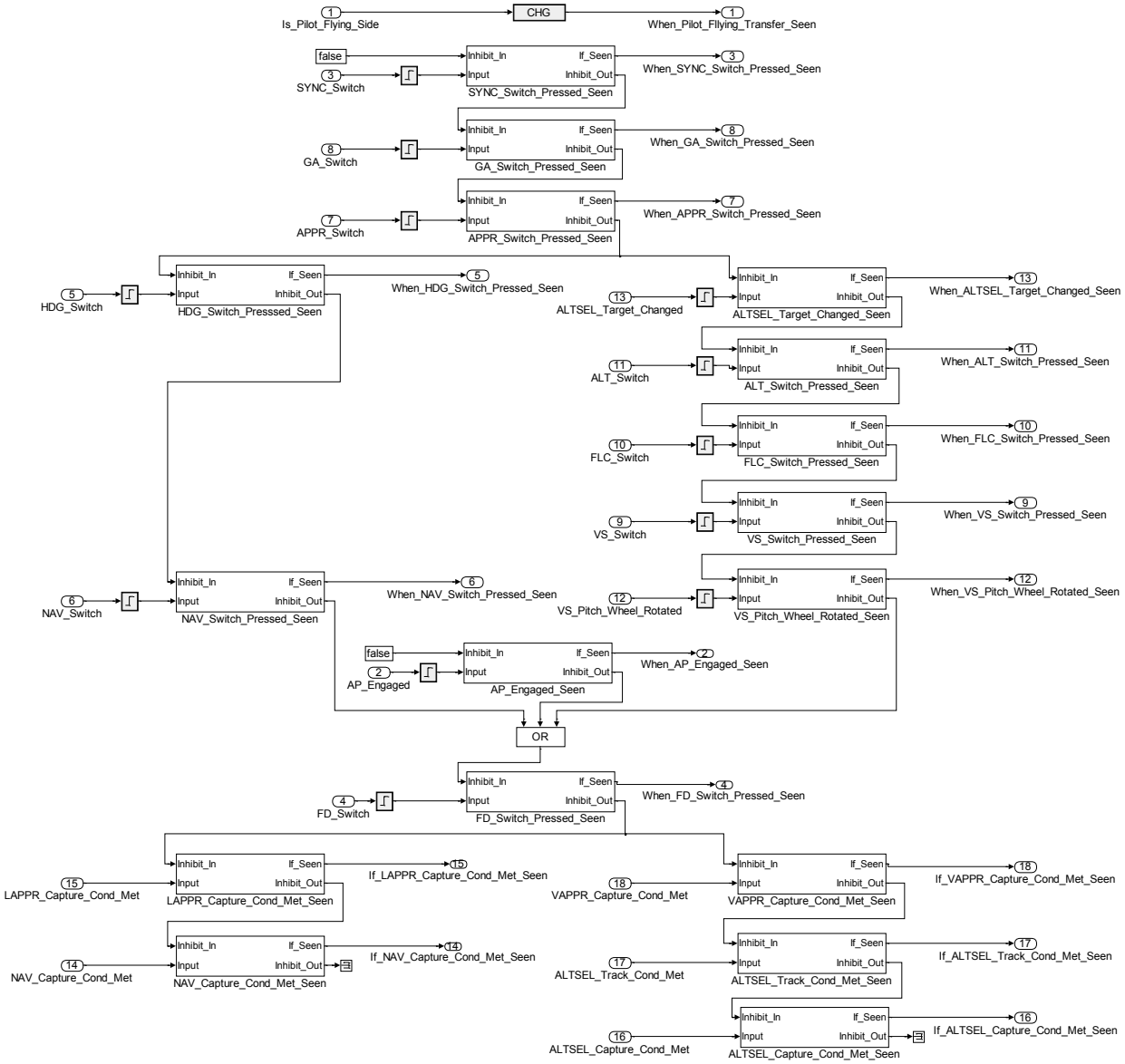

Figure 97 – Event Processing

Blocks such as *SYNC_Switch_Pressed_Seen* instantiate the *Seen* logic shown in Figure 97, which outputs *Seen* as true if its *Input* signal is true while the *Inhibit_In* signal is false. This block also generates an *Inhibit_Out* signal when either the *Inhibit_In* input or the *Seen* output is true.

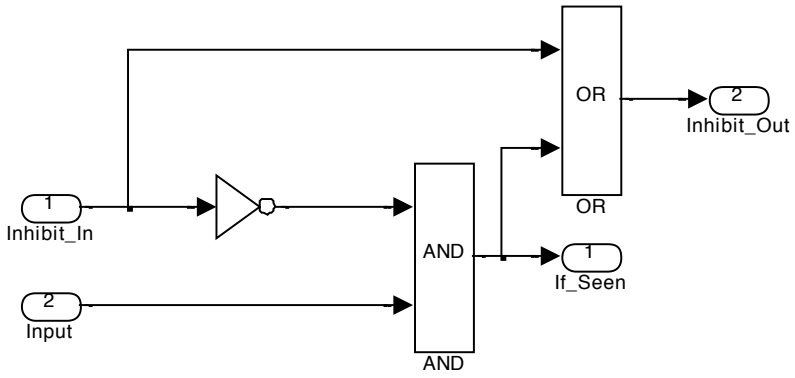

Figure 98 – Seen Logic

These blocks can be cascaded as shown in Figure 97 to inhibit generation of low priority events when a higher priority event occurs at the same time. In this way, simultaneous input events are resolved in favor of the more important events, providing a mechanism to ensure that only one lateral and one vertical mode becomes active on each step.

It is not necessary to preempt all simultaneous events. For example, the *HDG_Switch_Pressed* event can occur simultaneously with the *VS_Switch_Pressed* event without conflict. For this reason, the prioritization of events is organized in a lattice that allows some simultaneous events. This helps to ensure that as few input events as possible are inhibited. Identifying which events can safely occur simultaneously is done through model checking.

Most of the blocks are guarded by a rising edge detector, which is true for the one step on which the input transitions from false to true. These blocks generate signals that correspond to events, which can only be true for one step. However, some blocks, such as *LAPPR_Capture_Cond_Met_Seen*, are not preceded by rising edge detectors. These blocks establish a priority on input conditions, which can be true for several steps. Input conditions are given a lower priority than input events. This allows the mode logic to respond to a high priority input event and still respond to an input condition (that will probably still be true) on the subsequent step.

Also, note that the *When_Pilot_Flying_Transfer_Seen* event is set true whenever the *Is_Pilot_Flying* input (indicating if this is the pilot flying side of the aircraft) changes value. This event does not conflict with any other inputs or conditions and is neither inhibited nor inhibits the other outputs of Event Processing.

The advantage of this approach is that it isolates the handling of conflicting input events into one place where it is easier to reason about their priority. While discarding some events might seem unsafe, a simple thought exercise shows that it is safe and preferable to discard the lower priority event than it is to try to process both events. Consider the situation in the FGS in which the lower priority event occurs immediately after the higher priority event. In that situation, the lower priority event effectively overrides the higher priority event. Now consider the situation in which the higher priority event occurs immediately after the lower priority event. In that situation, the higher priority event effectively overrides the lower priority event. Both sequences can happen in the FGS and both result in safe flight. So discarding the lower priority event is acceptable since it corresponds to the situation in which the lower priority event occurs immediately before the higher priority event. Allowing some events to occur simultaneously because they do not conflict with each other is a modest advantage made feasible through formal verification.

4.3.2.2 Alternatives to Event Prioritization

There are other approaches to resolving conflicting simultaneous events. One approach is to resolve the conflicts by adding constraints in the tables for mode activation that directly consider all possible combinations of inputs. However, this can quickly become overwhelming and obscures the most important cases when there aren't simultaneous inputs. It also unnecessarily entangles the logic of the different modes.

Another approach is to add an Active state machine that keeps track of which mode is currently active and use it to coordinate the individual mode machines. While appealing on the surface, it has several disadvantages. First, unlike the individual mode machines, it does not represent an abstraction of the underlying system state; its purpose is to enforce the constraint that only one lateral mode and one vertical mode can be active at a time. An indication of this is the fact that there must be a transition from every active mode to every other active mode, since any mode can become active at any time. However, the more important issue is that it separates the logic for making a mode active from the logic for arming and clearing the modes. This leads to a model of the mode logic that is distributed across several state machines and is difficult to understand.

The approach taken here is to represent each mode as a small, relatively independent, state machine that represents an abstraction of the underlying flight control law. Rather than building

complex structures to enforce constraints required by the physical aircraft, we use formal analysis to ensure those constraints are met.

4.4 Formal Verification of the Mode Logic

This section discusses how model checking can be used to formally verify that the Mode Logic model meets is requirements. Since model checking is so highly automated, more of the emphasis shifts to writing good properties than in theorem proving. In Section 4.4.1, we discuss informal heuristics for writing good properties. In Section 4.4.2 we use these heuristics to develop formal properties for the mode logic and verify them using the Kind SMT-based model checker. In Section 4.4.3 we discuss verifying a subset of these properties using MATLAB Design Verifier.

4.4.1 Heuristics for Writing Formal Properties

Writing good formal properties shares many similarities with writing good requirements and is as much art as science. Fortunately, most organizations tend to build variations of the same systems and will develop libraries of good properties over time. However, writing that first set of properties can be challenging. Formal properties do have the advantage over requirements that they can be mathematically checked against a model of the system and even a single property can find many errors. In many ways, it is more important to get started than it is to write the ideal set of properties. This section describes several strategies, heuristics, and rules of thumb for writing good formal properties. While certainly incomplete and informal, many of these were followed in developing the formal properties for the mode logic discussed in the next section.

One of best sources of formal properties is often found in the safety-related[6] requirements for the system. Not only are these requirements inherently important, but their implementation typically touches on several parts of the system. Properties that cut across an entire system in this way often find the most errors. For example, in the mode logic, the properties that found the most errors were those that checked that at least one lateral and vertical mode was active and that at most one lateral and vertical mode was active. These properties depended on the correct interaction of all the modes and cut across the entire model.

---

[6] Here "safety-related requirements" are those can affect the safe operation of the aircraft as used in DO-178C rather than the formal methods notion of a predicate over a finite number of successive states.

If the system developers or domain experts are available, another good strategy is to simply ask them what things they are the most worried about in the system. Refining their concerns into a set of formal properties often requires an ongoing dialogue[7], but their intuition and knowledge of the system can be invaluable.

Another excellent source of formal properties is the system or software requirements. Informal requirements can take on many different forms, from text to block diagrams to state-machines to use cases to pseudo-code, but their intent is always to describe what the system should do without specifying how it should do it. Often, an informal requirement actually encompasses several properties. A strategy that we have used successfully is to transcribe the informal requirement into one or more textual statements and then writing a formal property for each textual statement.

User manuals can also be an excellent source of formal properties. Often, user manuals are written with less design information than the system requirements. Most of the requirements for the mode logic were originally developed from user manuals describing the system for pilots.

Once the above sources of properties have been exhausted, another option is to carefully review the model itself looking for anything that is not checked by a property. Often, this will identify requirements (and hence properties) that have been completely omitted. Surprisingly, even writing properties directly from the model itself will often expose errors in understanding the semantics of the model and should not be ruled out as a strategy.

Another heuristic that can be helpful in developing properties is to consider each input and try to identify all the ways that changing that input can affect one or more outputs and then write properties describing each such change. This provides a systematic way of breaking the problem down into several small problems and identifying properties. Ideally, the behavior of the system expressed in the requirement is gleaned from requirements or user manuals.

A final guideline concerns the process of writing properties and their verification. When a property is found to be false, this only means that there is a discrepancy between the formal property and the model. As often as not, such discrepancies expose errors in the property rather

---

[7] For example, when asked this question one developer replied simply "whether it's right." Further discussion led to the development of several important safety-related properties.

than in the model, and these take the form of missing assumptions. For example, the original requirement may be written as "the system shall arm for vertical approach mode when the APPR button is pressed" but formal verification reveals that this is true unless an overspeed condition exists. In this situation, it is probably the formal property, not the model, which needs to be corrected. Since many requirements will share the same undocumented assumptions, it is generally best to check properties incrementally as they are developed rather than developing all the properties and then checking them. In this way, insights gained from verifying the first properties can be incorporated into the development of later properties.

4.4.2 Verification of the Mode Logic Using the Kind Model Checker

This section describes how the Kind Model Checker can be used to formally verify that the MATLAB Simulink/Stateflow model described in Section 4.2 meets its requirements. To do this, the model must be translated into the Lustre language accepted by the Kind model checker and the requirements stated formally in Lustre. The translation into Lustre has been performed using the Rockwell Collins Formal Verification Framework. The resulting file is provided along with this report. Section 4.4.2.1 describes how the safety-related requirements and functional requirements are stated formally in the Lustre language. Section 4.4.2.2 discusses the process of understanding the counterexamples produced from three false properties and correcting the Mode Logic model.

4.4.2.1 Writing Properties for the Kind Model Checker

There are 118 properties that have been formally verified for the mode logic. These are listed in Appendix B – Mode Logic Properties. The reader may wish to refer to that appendix while reading this section. We specify properties in Lustre by defining a unique Boolean variable for each requirement and assigning to this variable the formal specification of the requirement. For example, the requirement that at least one lateral mode shall always be active is specified as

```
--------------------------------------------------------------------------------
-- At least one lateral mode shall always be active
-- when the FD is displayed or the AP is engaged.
--------------------------------------------------------------------------------
 At_Least_One_Lateral_Mode_Active =
    ROLL_Active or HDG_Active or NAV_Active or LAPPR_Active or LGA_Active;

 check At_Least_One_Lateral_Mode_Active;
```

Figure 99 – At Least One Lateral Mode Active (Lustre)

We have chosen to use descriptive names such as *At_Least_One_Lateral_Mode_Active* for the Boolean values assigned to requirements. Any unique valid Lustre name, for example, R0015a, would also be acceptable. We will also follow the convention of embedding the informal, textual statement of the requirement in Lustre comments (lines starting with "—") immediately before the formal statement of the requirement. The requirement is then formally specified by assigning a predicate defining the requirement to the Boolean variable. In the case of Figure 99 this is just a Boolean expression stating that at least one of the system outputs specifying the active status of the lateral modes must be true. Finally, the "check" statement instructs the Kind model checker to attempt to prove the property is always true for all possible combination of inputs and states.

Note that we have actually verified a stronger requirement than the one stated in the informal textual requirement which requires that a lateral mode must be active when the FD is displayed or the AP is engaged. Its proof demonstrates at least one lateral mode is *always* active, not just when the FD is displayed or the AP is engaged. It is always acceptable to prove a stronger property than is actually required, though we may wish to keep track of the original requirement in case changes to the model invalidate the stronger property. If this stronger property was not true of our model, the weaker actual property could be stated using an implication as shown in Figure 100.

```
--------------------------------------------------------------------------------------
-- At least one lateral mode shall always be active
-- when the FD is displayed or the AP is engaged.
--------------------------------------------------------------------------------------
 At_Least_One_Lateral_Mode_Active =
    FD_On or Is_AP_Engaged =>
       ROLL_Active or HDG_Active or NAV_Active or LAPPR_Active or LGA_Active;

 check At_Least_One_Lateral_Mode_Active;
```

Figure 100 – Weaker Version of at Least One Lateral Mode Active (Lustre)

The requirement that at least one lateral mode is always active when the FD is on or the AP is engaged is an example of a safety-related software requirement that traces to a system level safety requirement that the FGS shall provide valid guidance when the FD is on or the AP is engaged. As discussed in 4.4.1, safety-related requirements are excellent candidates for formal verification. This is partly because of their explicit relationship to system safety, but it is also due to the inherent difficulty of testing them. Safety-related requirements are often of the form "bad things shall never happen" or conversely, "good things shall always happen." Such requirements are difficult to test since testing can demonstrate their truth only for states and inputs actually

tested. In contrast, formal verification proves them to be true for all possible combinations of inputs and system states. Safety-related requirements are also excellent properties for finding design errors since the entire system often contributes to maintaining safety-related requirements and an error anywhere in the system will often falsify the requirement.

Another safety-related requirement is the requirement that no more than one lateral mode can ever be active at the same time. This is important since having two lateral flight control laws active at the same time would generate conflicting guidance commands to the FD and the AP. This requirement is formally stated in Figure 101.

```
--------------------------------------------------------------------------------------
-- At most one lateral mode shall be active
-- when the FD is displayed or the AP is engaged.
--------------------------------------------------------------------------------------
 At_Most_One_Lateral_Mode_Active =
  (ROLL_Active =>
    not (               HDG_Active or NAV_Active or LAPPR_Active or LGA_Active)) and
  (HDG_Active =>
    not (ROLL_Active or               NAV_Active or LAPPR_Active or LGA_Active)) and
  (NAV_Active =>
    not (ROLL_Active or HDG_Active or               LAPPR_Active or LGA_Active)) and
  (LAPPR_Active =>
    not (ROLL_Active or HDG_Active or NAV_Active or                 LGA_Active)) and
  (LGA_Active =>
    not (ROLL_Active or HDG_Active or NAV_Active or LAPPR_Active              ));

 check At_Most_One_Lateral_Mode_Active;
```

Figure 101 – At Most One Lateral Mode Active (Lustre)

The requirements for at least one and at most one lateral mode active could be combined into a single requirement that exactly one lateral mode shall be active. We have chosen to state them as separate requirements as a matter of preference. In similar fashion, there are requirements that at least one *vertical* mode shall be active and at most one vertical mode shall be active.

Another safety-related requirement is that VAPPR (vertical approach) mode shall be active only if LAPPR (lateral approach) mode is active. This is important since the aircraft should not be making a vertical descent to land when it isn't aligned with the runway. It turns out to be difficult to deactivate VAPPR mode on the exact step in which LAPPR mode is deactivated, but easy to deactivate VAPPR mode on the next step after LAPPR is deactivated. Since a delay of one step is insignificant given the inertia of the aircraft, this requirement can be relaxed to allow a one-step delay in clearing VAPPR mode when LAPPR mode is deactivated. This can be formally stated as "if VAPPR mode is active on two successive steps, LAPPR mode must be

active on the first step." Since this is true for any two successive steps, it is equivalent to requiring that LAPPR must be active when VAPPR is active except on the last step in which LAPPR is active. This is shown in Figure 102 using the Lustre "pre" operator which returns the value of a variable on the previous step.

```
--------------------------------------------------------------------------------
-- VAPPR mode shall be active only if LAPPR mode is active (except on the last step).
--------------------------------------------------------------------------------
 VAPPR_Active_Implies_LAPPR_Active =
   (pre VAPPR_Active and VAPPR_Active) => pre LAPPR_Active;

 check VAPPR_Active_Implies_LAPPR_Active;
```

Figure 102 – VAPPR Active Only If LAPPR Active (Lustre)

Of course, the weakening of the original requirement to accommodate the design decision allowing a one-step delay needs to be fed back into the system safety process for review as specified in DO-178B/C to ensure that it does not violate the system safety requirements.

Another safety-related requirement is that LGA (lateral go around) mode shall be active if and only if VGA (vertical go around) mode is active. These modes are active only during takeoff or during a go around following an aborted landing. The LGA flight control law maintains a fixed heading while the VGA flight control law maintains a fixed pitch and both modes should be active at the same time. Similar to the requirement that VAPPR mode active implies LAPPR mode active, this requirement is much simpler to implement if it can be relaxed for one step. For this reason, the requirement is specified as two implications as shown in Figure 103.

```
--------------------------------------------------------------------------------
-- VGA mode shall be active if LGA mode is active (except for one step).
--------------------------------------------------------------------------------
   LGA_Active_Implies_VGA_Active =
       (pre LGA_Active and LGA_Active) => pre VGA_Active;

   check LGA_Active_Implies_VGA_Active;

--------------------------------------------------------------------------------
-- LGA mode shall be active if VGA mode is active (except for one step).
--------------------------------------------------------------------------------
   VGA_Active_Implies_LGA_Active =
       pre VGA_Active and VGA_Active => pre LGA_Active;

   check VGA_Active_Implies_LGA_Active;
```

Figure 103 – LGA Active If and Only If VGA Active (Lustre)

Another safety-related requirement is that when an overspeed condition occurs, either FLC, ALT, or ALTSEL mode shall be active. The normal response of the FGS to an overspeed

condition is to enter FLC mode to pitch the aircraft up to reduce speed. However, to avoid moving outside of the aircraft's assigned flight level, the pilot may select either ALT mode to hold the aircraft at its current altitude or the aircraft may activate ALTSEL mode by capturing and tracking the preselected altitude. However, trying to prove this requirements reveals that there are several ways in which either ALT or ALTSEL mode can be deactivated (e.g., deselection by the pilot, rotating the VS Pitch Wheel, or a pilot flying transfer) causing the aircraft to enter basic PITCH mode in which it holds the current pitch angle. However, we can prove that FLC, ALT, ALTSEL, or PITCH mode must be active when an overspeed condition exists as shown in Figure 104.

```
--------------------------------------------------------------------------------
-- FLC, ALT, ALTSEL, or PITCH mode shall be active
-- while an overspeed condition exists.
--------------------------------------------------------------------------------
 Overspeed_Implies_FLC_ALT_ALTSEL_PITCH =
   Overspeed => FLC_Active or ALT_Active or ALTSEL_Active or PITCH_Active;

 check Overspeed_Implies_FLC_ALT_ALTSEL_PITCH;
```

Figure 104 – Overspeed Implies FLC, ALT, ALTSEL, or PITCH Active (Lustre)

Once in PITCH mode, FLC will immediately be selected due to the overspeed condition, so PITCH mode can be active for only one step while an overspeed condition exists. We confirm this by proving the property shown in Figure 105 that if PITCH is active in one step and an overspeed condition exists in the next step, the system shall exit PITCH mode.

```
--------------------------------------------------------------------------------
-- PITCH mode shall be active for only one step while an overspeed condition exists.
--------------------------------------------------------------------------------
 Overspeed_and_PITCH_Transitory = true ->
   pre PITCH_Active and Overspeed => not PITCH_Active;

 check Overspeed_and_PITCH_Transitory;
```

Figure 105 – Overspeed and PITCH Transitory (Lustre)

The -> (followed by) operator of Lustre (not to be confused with the => implies operator) is used in Figure 105 to exclude the initial system state from the proof. The -> operator replaces the value of the Boolean predicate in the first step with the value true (its left hand operand) and uses the value of the predicate (its right hand operand) for all subsequent steps. This is necessary since as it is possible for the Overspeed input to be true in the initial step before the system has had time to respond to it. The -> operator is a convenient way to exclude the initial system state from proofs in which the validity of the property in the initial step does not matter.

When combined with the proof of Figure 104 this proves that FLC, ALT, or ALTSEL mode must be active during an overspeed condition except for one transitory step during which PITCH mode can be active. As before, these requirement changes must be fed back into the system safety process for review.

As discussed in Section 4.4.1, another useful source of properties is the functional requirements for the mode logic or the user's manual. For example, HDG is simple lateral mode in which the aircraft acquires and tracks a heading reference (i.e. a compass direction). Both the system requirements and the user's manual for the FGS state that this mode should be selected whenever the pilot presses the HDG button on the FCP if HDG mode is not already selected. This is formally stated as shown in Figure 106.

```
--------------------------------------------------------------------------------
-- HDG mode shall be selected if the HDG switch is pressed while HDG mode is cleared.
--------------------------------------------------------------------------------
 HDG_Switch_Pressed_Selects_HDG =
   not pre HDG_Selected and RISING(HDG_Switch)
   and No_Higher_Event_Than_HDG_Switch_Pressed => HDG_Selected;

 check HDG_Switch_Pressed_Selects_HDG;
```

Figure 106 – HDG Switch Pressed Selects HDG (Lustre)

Formally stating the requirement of Figure 106 requires the introduction of two auxiliary definitions in Lustre. The first of these, the function *RISING*, is quite simple. Its definition is shown in Figure 107.

```
--------------------------------------------------------------------------------
-- RISING - returns true when signal s changes from false to true
--------------------------------------------------------------------------------
node RISING (s : bool) returns (p : bool);
let
   p = false -> (not pre s and s);
tel;
```

Figure 107 – Definition of RISING (Lustre)

RISING takes a single Boolean valued input and returns true if its value has changed from false to true. Note that its value in the initial step is always false. The function *RISING* must be used in the formal statement of Figure 106 since the mode logic only responds a rising value of the input *HDG_Switch* as shown in the *Event_Prioitization* logic of Figure 97.

The second auxiliary definition, *No_Higher_Event_Than_HDG_Switch_Pressed*, is more complicated. Recall that the *Event_Prioritization* logic described in Section 4.3.2.1 masks some

input events when a higher priority event occurs at the same time. Since the input *HDG_Switch* refers to the system level input rather than the possibly masked value *HDG_Switch_Pressed_Seen* passed into the *Flight_Modes* specification (Section 4.3.1), formally specifying the conditions under which HDG mode is selected must incorporate the behavior in the event prioritization logic.

One way to do this would be to use the internal value *HDG_Switch_Pressed_Seen* computed in the model itself to formally state the requirement. A practical difficulty in doing this is that the intermediate value's name in the Lustre specification may be quite different from the name used by the Simulink designer due to the translation process. In fact, the intermediate value may even have been optimized away during translation. However, if both of these obstacles were overcome, the requirement of Figure 106 could be restated as shown in Figure 108.

```
--------------------------------------------------------------------------------
-- HDG mode shall be selected if the HDG switch is pressed while HDG mode is cleared.
--------------------------------------------------------------------------------
 HDG_Switch_Pressed_Selects_HDG =
   not pre HDG_Selected and HDG_Switch_Pressed_Seen => HDG_Selected;

 check HDG_Switch_Pressed_Selects_HDG;
```

Figure 108 – HDG Switch Pressed Selects HDG Using Internal Variables (Lustre)

Despite its intuitive appeal, there is a more insidious danger in using internal values to specify properties. The problem is that the validity of the proof now depends on the correctness of the model itself. For example, imagine that the portion of the model that computes *HDG_Switch_Pressed_Seen* is incorrect and always returns the value false. The property is then trivially true since false => p is always true for any predicate p. The effect would be that an error in the portion of the model defining *HDG_Switch_Pressed_Seen* could mislead us into believing the property of Figure 108 was true when it was actually false.

One solution to this problem would be to prove the correctness of the internal variable with its own set of properties. Another solution, and the one that we have used here, is to only use input and output variables (i.e. no internal variables) in our properties. However, restricting the variables in properties to only system input and output variables leads to verbose properties unless we first introduce auxiliary definitions that independently specify portions of the model. The function *RISING* is one example of such an auxiliary definition. Another is the Lustre

variable *No_Higher_Event_Than_Heading_Switch_Pressed*. Its definition is shown in Figure 109.

```
No_Higher_Event_Than_HDG_Switch_Pressed =
  (not RISING(APPR_Switch) and No_Higher_Event_Than_APPR_Switch_Pressed);
```

Figure 109 – No Higher Event Than HDG Switch Pressed (Lustre)

The purpose of this variable is to independently specify the event prioritization logic relevant to the HDG switch. It is set to the value true if the next higher priority event, the pressing of the APPR switch, does not occur and no even higher priority event than the pressing of the APPR switch occurs. Of course, the Lustre variable *No_Higher_Event_Than_APPR_Switch_Pressed* must also be defined, but the resulting collection of recursive definitions nicely captures the event prioritization logic in a form that supports succinct specification of mode logic properties. For example, formalizing the requirement that HDG mode should be selected if the HDG switch is pressed while HDG mode is cleared can be written as shown in Figure 106.

Since HDG mode is a non-arming mode, to complete the verification of its functional behavior we need to specify all ways in which HDG mode can be selected and all ways in which HDG mode can be cleared. HDG mode can only be selected by the pilot pressing the HDG switch, but there are three ways in which HDG mode can be cleared. These are shown in Figure 110.

```
--------------------------------------------------------------------------------------
-- HDG mode shall be cleared if the HDG switch is pressed while HDG mode is selected.
--------------------------------------------------------------------------------------
 HDG_Switch_Pressed_Clears_HDG =
   pre HDG_Selected and RISING(HDG_Switch)
   and No_Higher_Event_Than_HDG_Switch_Pressed => not HDG_Selected;

 check HDG_Switch_Pressed_Clears_HDG;

--------------------------------------------------------------------------------------
-- HDG mode shall be cleared when there is a pilot flying transfer.
--------------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_HDG =
   pre HDG_Selected and CHANGED(Pilot_Flying_Side)  => not HDG_Selected;

 check Pilot_Flying_Transfer_Clears_HDG;

--------------------------------------------------------------------------------------
-- HDG mode shall be cleared when the mode annunciations are turned off.
--------------------------------------------------------------------------------------
 Modes_Off_Clears_HDG =
   pre HDG_Selected and not Modes_On  => not HDG_Selected;

 check Modes_Off_Clears_HDG;
```

Figure 110 – Functional Requirements for Clearing HDG Mode (Lustre)

The functional requirements for arming modes such as NAV, LAPPR, ALTSEL, and VAPPR must include properties describing how an armed mode becomes active. Our first attempt at formalizing a property specifying how NAV mode can become active is shown in Figure 111.

```
--------------------------------------------------------------------------------------
-- NAV mode shall become active if the NAV capture condition is met
-- while NAV mode is armed.
--------------------------------------------------------------------------------------
 NAV_Active_When_Capture_Cond_Met = true ->
   pre NAV_Selected and not pre NAV_Active
   and NAV_Capture_Cond_Met
   and No_Higher_Event_Than_NAV_Capture_Cond_Met => NAV_Active;

 check NAV_Active_When_Capture_Cond_Met;
```

Figure 111 – Initial Functional Requirements for Activating NAV Mode (Lustre)

This property specifies that NAV mode is armed by stating that it is selected but not active in the previous step. There is also no need to look for the rising edge of *NAV_Capture_Cond_Met* since the mode logic transitions from armed to active whenever the NAV capture condition is true, not just on its rising edge. However, attempting to prove this property identifies several conditions under which it is not true. The correct property is shown in Figure 112.

```
--------------------------------------------------------------------------------------
-- NAV mode shall become active if the NAV capture condition is met
-- while NAV mode is armed.
--------------------------------------------------------------------------------------
 NAV_Active_When_Capture_Cond_Met = true ->
   pre NAV_Selected and not pre NAV_Active
   and NAV_Capture_Cond_Met
   and not Selected_NAV_Source_Changed
   and not Selected_NAV_Frequency_Changed
   and not CHANGED(Pilot_Flying_Side)
   and Modes_On
   and No_Higher_Event_Than_NAV_Capture_Cond_Met => NAV_Active;

 check NAV_Active_When_Capture_Cond_Met;
```

Figure 112 – Correct Functional Requirements for Activating NAV Mode (Lustre)

For most arming modes it is possible for the mode to be either activated or cleared while armed, depending on external events. NAV mode will be cleared if the selected navigation source is changed (e.g. selecting a different type of navigation beacon), the frequency is changed, the Transfer switch is pressed (causing a change in the pilot flying side) or the mode annunciations are turned off. All of these conditions must be incorporated into the antecedent of the property.

This illustrates one of the most important benefits of formal specification – it forces a precise enumeration of the exceptions to the normal case. Note that we have not gone back and revised

the textual informal specification of the requirement to include these exceptions. Instead, we have chosen to leave the informal specification the way it was originally written since this captures the requirement's original intent. The exceptions are then documented in the formal specification.

4.4.2.2 Debugging False Properties in Kind

One of the most important benefits of formal verification is its ability to find errors that traditional verification approaches such as reviews or testing would miss. For example, sixteen errors were found when verifying the MATLAB Simulink/Stateflow model of the mode logic using the Kind model checker (see Appendix C). Even though this example had been specified previously in RSML^{-e} [4] and formally verified using the NuSMV model checker, the process of rewriting it in Simulink/Stateflow was sufficient to introduce new errors. This section discusses a few of the errors listed in Appendix C and describes the process of understanding why a property is false and correcting the problem.

4.4.2.2.1 VGA Clear Error

Our first example is a simple naming error that was found by the Kind model checker in a few seconds. While this error would probably have been found through testing, it was simpler and faster to find it through model checking. It is also a good first example because of its simplicity. The error was made in the specification of the VGA mode and is shown in Figure 113.

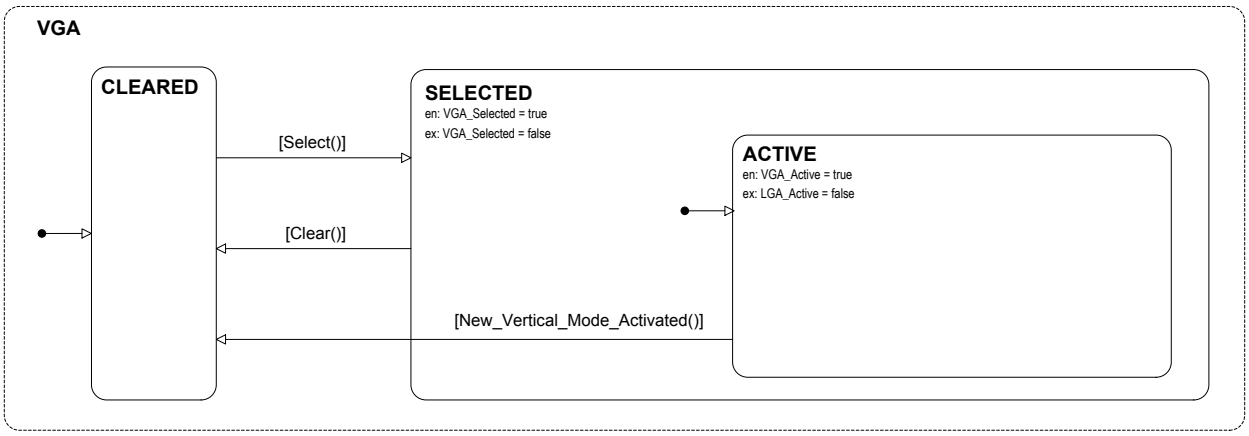

Figure 113 – VGA Clear Error

The output variable *LGA_Active*, rather than *VGA_Active*, was being set to false on exiting ACTIVE mode. This error was detected while trying to prove that at least one lateral mode is

always active (Figure 99). The counterexample produced by the Kind model checker is shown in Figure 114.

	Step 1	Step 2	Step 3
Input Signals			
ALTSEL_Capture_Cond_Met	false	false	false
ALTSEL_Target_Changed	false	false	false
ALTSEL_Track_Cond_Met	false	false	false
ALT_Switch	false	false	false
APPR_Switch	false	false	false
FD_Switch	true	false	true
FLC_Switch	false	false	false
GA_Switch	false	true	false
HDG_Switch	false	true	false
Is_AP_Engaged	true	false	false
Is_Offside_VAPPR_Active	false	false	false
Is_Offside_VGA_Active	false	false	true
LAPPR_Capture_Cond_Met	false	false	false
NAV_Capture_Cond_Met	false	false	false
NAV_Switch	true	false	false
Offside_FD_On	false	false	false
Overspeed	false	false	false
Pilot_Flying_Side	true	false	false
SYNC_Switch	true	true	false
Selected_NAV_Frequency_Changed	false	false	false
Selected_NAV_Source_Changed	false	false	false
VAPPR_Capture_Cond_Met	false	false	false
VS_Pitch_Wheel_Rotated	false	false	true
VS_Switch	false	false	false
Output Signals			
HDG_Active	false	false	false
LAPPR_Active	false	false	false
LGA_Active	false	true	false
NAV_Active	false	false	false
ROLL_Active	true	false	false

Figure 114 –Counterexample for Clearing VGA Error

The counterexample is three steps long with the values of the relevant inputs and outputs shown for each step (Kind does not generate values for inputs and outputs that do not affect the property). Values that have not changed from the previous step are shown in grey text. The most

significant values have had their enclosing box shaded in grey by the authors to help in understanding the counterexample.

In the initial step ROLL mode is active as expected. In the second step, the GA switch is pressed, activating LGA mode. This also activates VGA mode, though *VGA_Active* is not included in the counterexample since its value is not directly relevant to the property. In step 3, the VS Pitch Wheel is rotated, which clears VGA mode. This does not actually clear LGA mode in step 3 (though LGA mode would be cleared in step 4), but due to the naming error, the output variable *LGA_Active* is incorrectly set to false, making it appear that LGA mode has been cleared.

This example illustrates several important points about model checking. First, the model checker will produce a counterexample if it can invalidate a property, but it may not be the best counterexample for human comprehension. In understanding a counterexample, changes in values are often important clues (for example the GA switch being pressed in step 2), but the model checker may also change values that have no impact on outputs. For example, the HDG switch is pressed in step 2, but this is not relevant since it is masked due to event prioritization by the pressing of the GA switch. Finally, though not demonstrated here, if the model can be simulated it may be even more helpful to step through the simulation using the input values provided by the counterexample.

4.4.2.2.2 FLC Select Error

Our second example is a much more subtle design error that probably would not have been found through testing. It also would have allowed two vertical modes to be active while an overspeed condition existed. The error was made in the specification of the selection logic for FLC mode and is shown in Table 48.

Table 48 – FLC Select Error

Condition	1	2	3
FLC_Switch_Pressed	T	-	-
VAPPR_Active	F	-	-
Overspeed	-	T	-
ALT_Active	-	F	-
ALTSEL_Active	-	F	-
	TRUE	TRUE	FALSE

The error was detected while trying to prove that no more than one vertical mode is ever active. The counterexample produced by the Kind model checker is shown in Figure 115.

	Step 1	Step 2
Input Signals		
ALTSEL_Capture_Cond_Met	false	false
ALTSEL_Target_Changed	false	false
ALTSEL_Track_Cond_Met	false	false
ALT_Switch	false	true
APPR_Switch	false	false
FD_Switch	false	true
FLC_Switch	false	true
GA_Switch	true	false
HDG_Switch	false	false
Is_AP_Engaged	false	true
Is_Offside_VAPPR_Active	false	false
Is_Offside_VGA_Active	false	false
LAPPR_Capture_Cond_Met	false	false
NAV_Capture_Cond_Met	false	false
NAV_Switch	false	false
Offside_FD_On	false	false
Overspeed	false	true
Pilot_Flying_Side	true	true
SYNC_Switch	true	false
Selected_NAV_Frequency_Changed	false	false
Selected_NAV_Source_Changed	false	false
VAPPR_Capture_Cond_Met	false	false
VS_Pitch_Wheel_Rotated	false	true
VS_Switch	false	true
Output Signals		
ALTSEL_Active	false	false
ALT_Active	false	true
FLC_Active	false	true
PITCH_Active	true	false
VAPPR_Active	false	false
VGA_Active	false	false
VS_Active	false	false

Figure 115 – Counterexample for FLC Select Error

In the initial step PITCH mode is active as expected. In the second step, the ALT switch is pressed, activating ALT mode. However, an overspeed condition also occurs in the second step, activating FLC mode. This occurred because of a design decision to never mask an overspeed condition in the *Event_Prioritization* logic of Section 4.3.2.1.

In this situation, precedence should be given to the pilot's selection of ALT mode to hold the aircraft at the current altitude. To enforce this, the FLC selection logic of Table 48 must be modified to return false if ALT will become active in this step and not just if it is already active. A similar change must be made if ALTSEL will become active in this step. The correct logic for FLC Select is shown in Table 29 on page 123.

It is worth noting that this error would have been very difficult to detect through testing since it depends on two events (the pilot pressing the ALT switch and the start of an overspeed condition) on the exact same step. It also would have been very difficult to find through inspection since very few reviewers would catch a corner condition such as this. The error also has an unknown impact on safety since it's not clear what the behavior of the aircraft would be with two flight control laws active at the same time.

4.4.2.2.3 ALTSEL Select Error

The last error illustrates how execution order can affect the behavior of a Stateflow model in subtle ways. As discussed in Section 4.3.1.4.4, ALTSEL mode is to be cleared when ALT, VAPPR or VGA are active and selected when none of them are active. This requirement is captured in the property of Figure 116 below.

```
--------------------------------------------------------------------------------------
-- If the mode annunciations are on, ALTSEL mode shall be selected if
-- none of ALT, VAPPR, or VGA mode are active.
--------------------------------------------------------------------------------------
 ALTSEL_Selected_If_Not_ALT_VAPPR_VGA_Active = true ->
   Modes_On and not (ALT_Active or VAPPR_Active or VGA_Active) => ALTSEL_Selected;

 check ALTSEL_Selected_If_Not_ALT_VAPPR_VGA_Active;
```

Figure 116 – ALTSEL Select Error (Lustre)

However, the Kind model checker was able to falsify this property in a few seconds, producing the counterexample shown in Figure 117.

Input Signals			
ALTSEL_Capture_Cond_Met	true	false	false
ALTSEL_Target_Changed	false	false	true
ALTSEL_Track_Cond_Met	false	false	false
ALT_Switch	false	false	true
APPR_Switch	false	false	true
FD_Switch	false	false	true
FLC_Switch	false	false	true
GA_Switch	false	true	false
HDG_Switch	false	false	true
Is_AP_Engaged	false	false	false
Is_Offside_VAPPR_Active	false	false	false
Is_Offside_VGA_Active	false	false	true
LAPPR_Capture_Cond_Met	true	false	false
NAV_Capture_Cond_Met	false	false	false
NAV_Switch	false	false	true
Offside_FD_On	true	false	false
Overspeed	false	false	false
Pilot_Flying_Side	false	false	false
SYNC_Switch	false	false	true
Selected_NAV_Frequency_Changed	false	false	false
Selected_NAV_Source_Changed	false	false	false
VAPPR_Capture_Cond_Met	false	false	false
VS_Pitch_Wheel_Rotated	false	true	false
VS_Switch	false	false	true
Output Signals			
ALTSEL_Selected	true	true	false
ALT_Active	false	false	false
Modes_On	false	true	true
VAPPR_Active	false	false	false
VGA_Active	false	true	false

Figure 117 – Counterexample for ALTSEL Select Error

In step 1 ALTSEL mode is selected with ALT, VAPPR, and VGA cleared as expected. In step 2, the GA Switch is pressed, causing VGA mode to become active. However, ALTSEL mode does not clear as expected. In step 3, the SYNC switch is pressed, clearing VGA mode. However, ALTSEL mode now clears.

This unexpected behavior occurred because in the original model, ALTSEL mode was assigned to execute immediately after ALT mode and before VAPPR and VGA mode. As a result, the ALTSEL selection logic of Table 35 on page 127 referred to the value of *ALT_Active* after ALT mode had executed and the values of *VAPPR_Active* and *VGA_Active* before VAPPR and VGA mode had executed. As a result, there was a one-step delay in the reaction of ALTSEL mode to

changes in VAPPR and VGA mode while changes in ALT mode were processed in the same step.

While this was not a particularly serious error, it does violate the original requirement. It could lead to considerable confusion during debugging, and it would also be very difficult to find using testing or reviews.

Fortunately, this error was easily fixed by assigning ALTSEL mode to execute after ALT, VAPPR, and VGA modes but before PITCH mode. To make this clear, the position of ALTSEL mode was changed so it was positioned immediately before PITCH mode.

4.4.3 Verification of the Mode Logic Using MATLAB Design Verifier

It is also possible to formally state and verify properties using MATLAB Design Verifier. Properties can be specified either graphically as Simulink/Stateflow models or textually as MATLAB function blocks. For example, the graphical specification of the requirement that at least one lateral mode shall be active is shown in Figure 118.

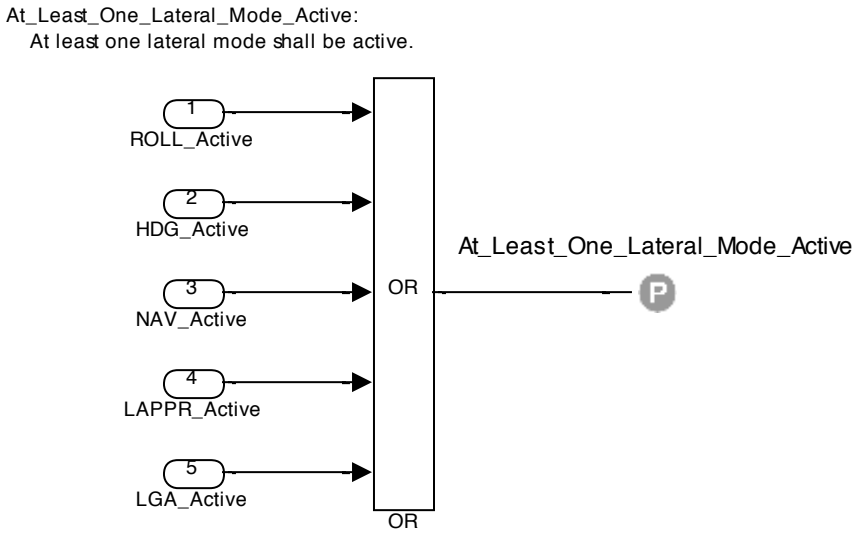

Figure 118 – At Least One Lateral Mode Active (Design Verifier)

This Simulink block simply computes the OR of the output signals for each active mode. The circular "P" icon is a proof objective block from the Design Verifier library. When Design Verifier is invoked on the model, it will attempt to prove that its value is always true (or whatever value has been specified in its dialog box).

The specification of the slightly more complex requirement that NAV mode shall become active when the NAV capture condition is met is shown Figure 119.

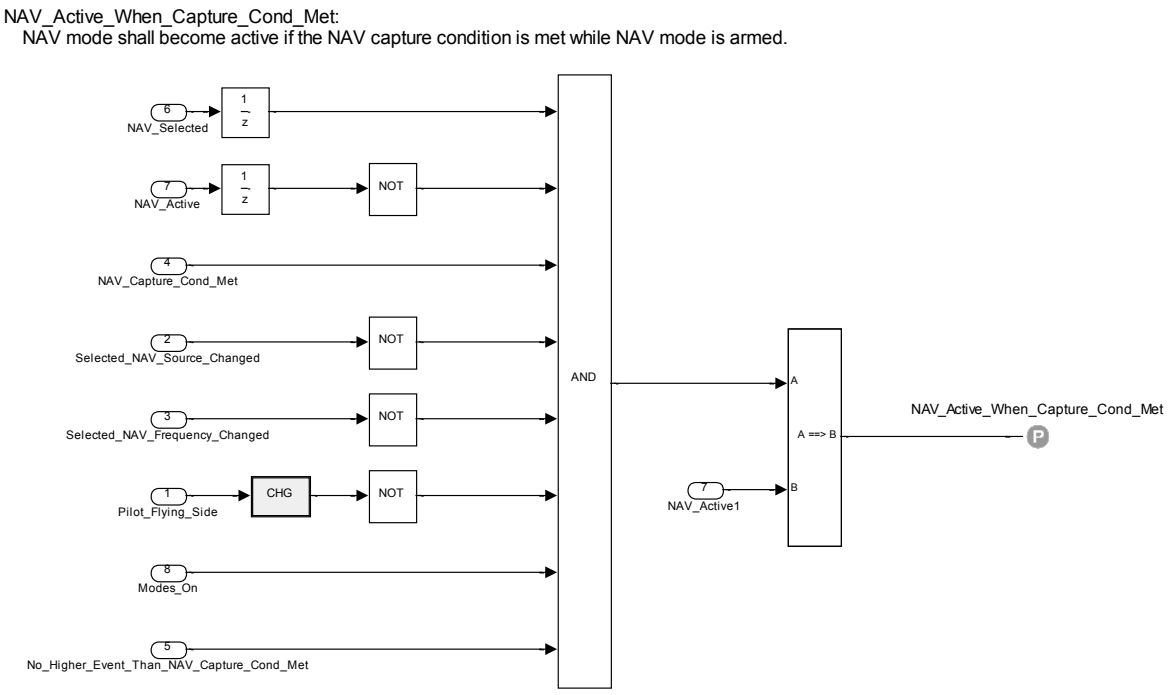

Figure 119 – NAV Active When Capture Cond Met (Design Verifier)

The implication (A ==> B) block used in Figure 119 is also part of the Design Verifier library. It returns true if the antecedent A is false or the consequent B is true. Note this property is true only when the selected NAV source and frequency are not being changed and a pilot flying transfer is not occurring and the mode annunciations are on and no higher priority event is masking the capture condition.

To illustrate the textual specification of properties, the requirement that at least one vertical mode is active is specified as a MATLAB function block in Figure 120.

```
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);
```

Figure 120 – At Least One Vertical Mode Active (Design Verifier)

The command *sldv.prove(P)* behaves similar to the Proof Objective of Figure 118 and instructs Design Verifier to attempt to prove that P is true for all combinations of inputs and outputs.

For convenience, we group both the graphical and textual properties into one or more subsystem blocks as shown in Figure 121.

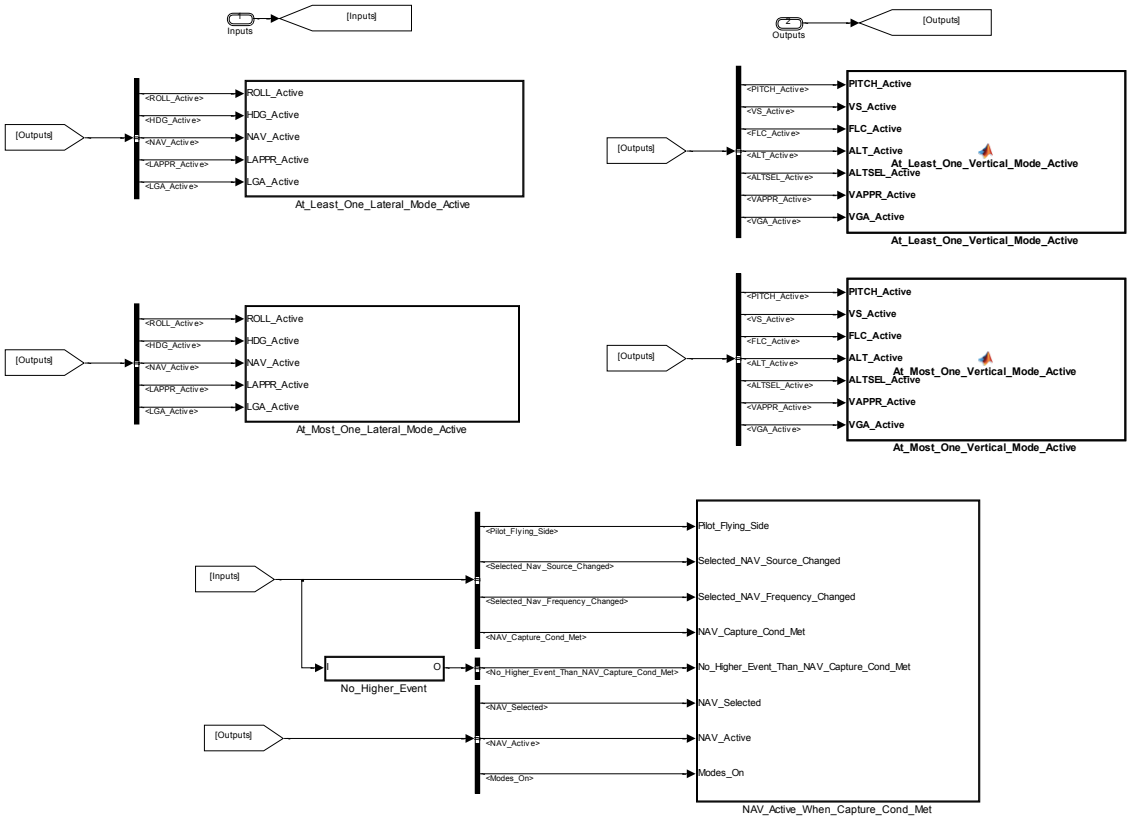

Figure 121 – Properties Subsystem (Design Verifier)

This block has bus input signal (Inputs) that contains all the inputs to the Mode Logic and a bus output signal (Outputs) that contains all the outputs from the mode logic. The bus signals are defined using the Simulink Bus Editor and bus selectors are used to extract individual signals that are used to define each property. Note that either inputs or outputs for the Mode Logic can be used as inputs for a specific property.

The *No_Higher_Event* block outputs a bus signal that contains signals such as *No_Higher_Event_Than_HDG_Switch_Pressed*. These signals serve the same purpose they did in the Lustre specification - to independently specify the event prioritization logic in a form that

supports the succinct specification of mode logic properties. In this example, the signal *No_Higher_Event_Than_NAV_Capture_Cond_Met* is being selected from the bus signal and input to the property specified in Figure 119.

The actual model that is analyzed with Design Verifier is shown in Figure 122. This model contains the Mode Logic model itself and the property subsystem block of Figure 121. It only has one input, the bus signal for the Mode Logic inputs. The individual inputs for the Mode Logic are extracted using bus selector blocks and input to the Mode Logic. The outputs from the Mode Logic are collected into a bus signal using a bus creator block. The Mode Logic input and output bus signals are then fed into the property subsystem block. If desired, several property subsystem blocks with different subsets of the Mode Logic properties could be created.

When the "Prove Properties" function of Design Verifier is invoked on this model, Design Verifier attempts to prove each proof obligation identified with a Proof Objective block or a *sldv.prove* command. Design Verifier will create a detailed analysis report of the results. The user can also request that a harness model be created to support the more detailed analysis of counterexamples.

The examples available with this report include Simulink specifications of ten of the 118 Mode Logic properties. Development and verification of the other 108 properties are left as an exercise for the reader.

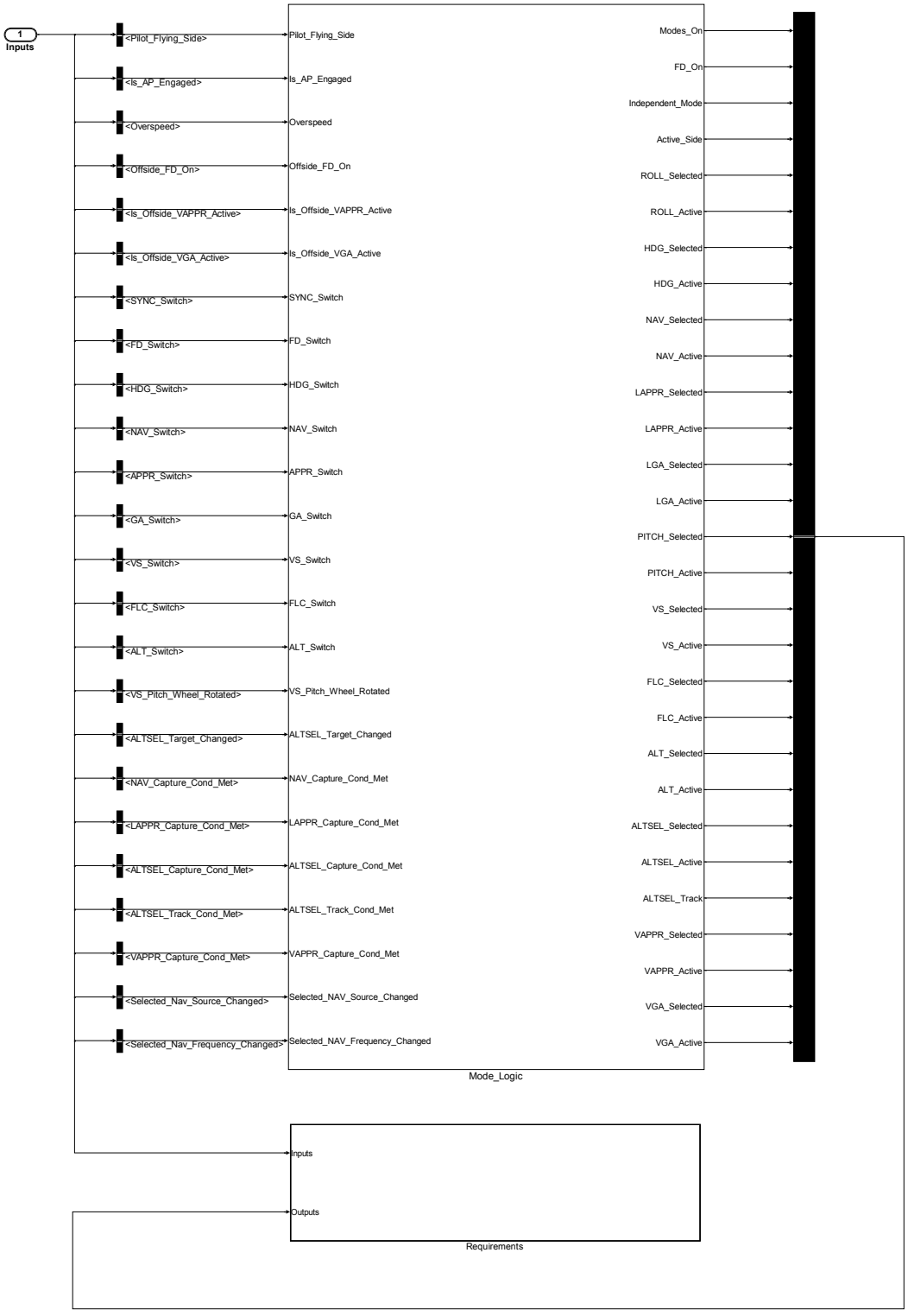

Figure 122 – Top-Most Model (Design Verifier)

5 Case Study: Abstract Interpretation

This section illustrates the use of abstract interpretation to verify the correctness of the source code implementation of one of the flight control modes for the example system. The Heading Control Law source code consists of C code generated from a Simulink model of the controller. We will check a variety of non-functional properties related to the run-time behavior of the code using the Polyspace and Astrée abstract interpretation tools.

The rest of the chapter is organized as follows. Section 5.1 provides an overview of the Heading Control model and the source code generated from it. It also describes the kinds of properties that will be checked by abstract interpretation. Section 5.2 describes the software verification plan for the Heading Control source code, identifying the life-cycle data items to be produced, the DO-178C objectives to be satisfied, and tool qualification issues. Sections 5.3 and 5.4 describe the formal verification results using Astrée and Polyspace, respectively.

5.1 Overview of the Heading Control Model

The Heading Control Law 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 have used a publicly available model provided by researchers at the University of Minnesota (UMN). A detailed description of the model and its use in an Unmanned Aerial Vehicle (UAV) flight test platform can be found in [3]. 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 example system.

5.1.1 Heading Control Model and Code

The heading control software is implemented as a single thread that executes at 50Hz. The design is implemented as a two-tiered structure with inner and outer control loops (Figure 123). The inner loop controller (shown in blue) tracks the desired pitch (theta) and roll (phi) angles of the aircraft while damping out oscillations present in the open-loop dynamics. It responds to

inputs produced by the three outer loop controllers. The Altitude Tracker produces a pitch angle reference command, and the Velocity Tracker produces a throttle command. Both the Altitude Tracker and Velocity Tracker use proportional-integral control and implement integrator anti-windup logic to safely limit the commands provided to the inner loop control system. The throttle command is constrained between 0 and 1, and the pitch angle reference is constrained to ±20º. The heading controller (Psi Tracker) uses proportional gain, and the roll angle reference is constrained directly at ±45º. This limiting is required to prevent the aircraft from rolling over due to large ground track angle step commands.

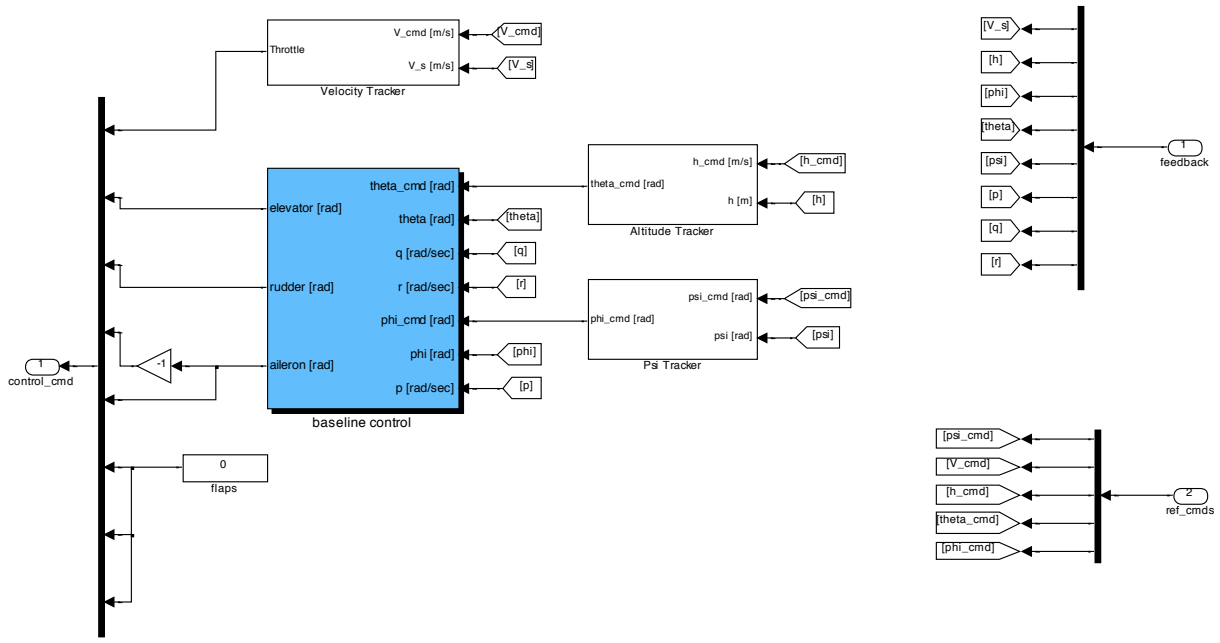

Figure 123 – Heading Control Law Model

The MATLAB Real-Time Workbench (RTW) was used to generate C code for the heading control model. The target platform selected to generate the source code for the Simulink model was the Embedded Real Time (ERT) target environment. RTW generated library files, header files, the *heading_control.c* and *ert_main.c*. The *heading_control.c* and the *ert_main.c* were included as the source files in the Polyspace project. The C header files also had to be included in the project to run the verification successfully. RTW generates functions for each block in the Simulink model. There are 856 lines of code in *heading_control.c* and 101 lines of code in *ert_main.c*. A fragment of the C code for *heading_control.c* is shown in Figure 124.

```
/*
 * File: heading_control.c
 *
 * Code generated for Simulink model 'heading_control'.
 *
 * Model version                  : 1.153
 * Simulink Coder version         : 8.1 (R2011b) 08-Jul-2011
 * TLC version                    : 8.1 (Jul  9 2011)
 * C/C++ source code generated on : Fri Mar 01 16:24:21 2013
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Generic->32-bit x86 compatible
 * Code generation objectives: Unspecified
 * Validation result: Not run
 */

#include "heading_control.h"
#include "heading_control_private.h"

/* user code (top of source file) */
#include "../../../Software/FlightCode/control/rtw_grt_control.c"

/* Block signals (auto storage) */
BlockIO_heading_control heading_control_B;

/* Block states (auto storage) */
D_Work_heading_control heading_control_DWork;

/* External inputs (root inport signals with auto storage) */
ExternalInputs_heading_control heading_control_U;

/* External outputs (root outports fed by signals with auto storage) */
ExternalOutputs_heading_control heading_control_Y;

/* Real-time model */
RT_MODEL_heading_control heading_control_M_;
RT_MODEL_heading_control *const heading_control_M = &heading_control_M_;

/* Model step function */
void heading_control_step(void)
{
  real_T denAccum;
  real_T u;
  real_T u_0;

  /* DiscreteIntegrator: '<S3>/Discrete-Time Integrator' */
  heading_control_B.DiscreteTimeIntegrator =
    heading_control_DWork.DiscreteTimeIntegrator_DSTATE;

  /* Gain: '<S3>/Gain1' */
  heading_control_B.Gain1 = 0.04 * heading_control_B.DiscreteTimeIntegrator;

  /* DiscreteTransferFcn: '<S3>/Discrete Transfer Fcn' */
  denAccum = 0.0392 * heading_control_DWork.DiscreteTransferFcn_DSTATE;
  heading_control_B.DiscreteTransferFcn = denAccum;
```

Figure 124 – Fragment of Autogenerated C Code for Heading Control Model

5.1.2 Properties to Be Checked

Abstract interpretation is a technique which can be applied at the source code level in order to prove the absence of runtime errors including division by zero, arithmetic overflow, and out-of-bounds array indexing. This kind of non-functional requirement is in general difficult to verify using testing, since runtime errors might only occur under certain very special circumstances that are not exposed in any test case. In contrast, abstract interpretation can guarantee that certain classes of runtime errors cannot occur under any circumstances.

Typical run-time errors that can be detected by abstract interpretation include:

- Unreachable code
- Out of bounds array index
- Division by zero
- Non-initialized variables
- Scalar and float overflows
- Uninitialized return value
- Shift operation errors
- Illegal pointer dereferencing
- Non-initialized pointers
- User assertions
- Non-termination of call
- Non-termination of loop
- Standard library function call error

Roughly speaking, abstract interpretation tools compute over-approximations of the set of all possible executions of a given program. This means that the tool does not only consider program executions which can really occur, but also executions which cannot occur because they would not respect the relations which exist between certain variables of a program. This is often referred to as the *precision* of an analysis: higher precision is more difficult to achieve, but means that there are fewer impossible executions which are included in the analysis.

A low precision analysis is easier to achieve. However, in this case the analyzer might indicate that a runtime error could happen at a certain point in a program, but none of the program

executions which can really occur will provoke this error. Such errors are called *false alarms*, and if they occur, the precision of the analysis must be increased by user interaction. A tool that produces a high number of false alarms not only results in an increased workload, but is annoying to use and is likely to be rejected by developers.

5.2 Software Verification Plan

In this case study, we will use 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 checking for run-time errors in the code rather than satisfaction of behavioral requirements, so this is where we will focus our effort. The purpose of these verification activities is to detect any errors that may have been introduced during the software coding process.

Verification will be performed on source code generated from the Simulink control law model. Our primary objective is to check the code for accuracy and consistency (DO-333 Section 6.3.4.f). 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 is a C code analysis tool which has been developed by the team of Professor Patrick Cousot at the Ecole Normale Supérieure in Paris [2] in close cooperation with Airbus. Therefore, Astrée provides some analysis capabilities which have been designed especially for real-time control software such as the code generated from SCADE models. In recent years, Astrée has been commercialized by the German company AbsInt, and features for the analysis of more general programs have been added, as well as a powerful GUI [1].

An example of a feature which enables the proof of absence of overflow errors in control applications is the so-called *filter domain*, which is able to express invariants of first and second order filters implemented by the analyzed code. Typically, filters have invariants in the form of ellipsoids, which cannot be described by linear expressions. Astrée tries to find patterns in the code which correspond to filters, and then uses the filter domain to compute invariants, taking into account that floating-point rounding errors can occur.

Polyspace is a commercial static analysis tool based on abstract interpretation and sold by MathWorks. It verifies both C and C++ code. Polyspace identifies the potential for overflow, divide by zero, out of bound array access, and other runtime errors. Polyspace provides some support for automated analysis to check compliance with the Software Architecture. It also supports compliance checking of the software against coding standards.

Since some runtime errors are dependent on the target CPU and operating system, the user must specify the type of CPU and operating system used in the target environment before running a verification. Other configuration parameters support tailoring of the precision of the analysis:

- Precision Level: This identifies the abstraction algorithm that is used to model the state of the program that is to be verified. It provides a trade-off between precision and analysis time.
- Verification Level: This indicates the Software Safety Analysis Level (0–4). This specifies how many times the abstract interpretation algorithm passes through the code. The deeper the verification goes the more precise it is. Each iteration results in a deeper level of propagation of calling and called context.

Polyspace can perform verification against some coding standards, such as MISRA C. This includes enforcing naming conventions checking for implicit type conversions, and detecting other error-prone coding practices. The tool has features to select only some rules among the set of all the available custom rules.

Polyspace provides some limited support to verify compliance with the software architecture. It generates a call tree that must be manually checked to see if it preserves the software architecture of the original model. It also determines the procedures and functions that have not been used.

There are also several open source abstract interpretation tools available, but which are outside the scope of this case study. Frama-C [6] is a suite of static analysis tools for software written in C. Its Value Analysis plugin uses abstract interpretation to compute a set of possible values for each variable in a program, and operates in both automatic and user-guided modes. IKOS is a C++ library designed to facilitate the development of sound static analyzers based on Abstract Interpretation [16]. IKOS provides a generic and efficient implementation of state-of-the-art Abstract Interpretation data structures and algorithms, such as control-flow graphs, fixpoint iterators, numerical abstract domains, etc.

5.2.1 Life Cycle Data Items

Low-Level Software Requirements The low-level software requirements are specified as a MATLAB Simulink model in the file *heading_control.mdl*, along with model libraries *Controller_Lib.mdl* and *Actuator_Lib.mdl*, and a collection of associated scripts.

Source Code The source code to be analyzed is C code autogenerated from the Simulink Low-Level Requirements using the MathWorks RTW. The code is contained in the files *ert_main.c*, *heading_control.c*, and *heading_control.h*, along with several other header files.

No separate behavioral requirements are verified in this case study. Abstract interpretation is being used to verify a standard set of run-time properties of C code.

5.2.2 Objectives to Be Satisfied

The DO-178C and DO-333 objectives to be satisfied through abstract interpretation are summarized in Table 49. A more detailed discussion of how each objective is satisfied is provided in this section.

Table 49 – 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 analysis 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 as part of tool qualification
FM.A-5.11	Formal analysis results are correct and discrepancies explained.	■	■	■		Established by review
FM.A-5.12	Requirements formalization is correct.	■	■	■		Established as part of tool qualification
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*

Objective A-5.3 – Source Code is verifiable. This objective is met by demonstrating that the code to be analyzed conforms to any input restrictions of the analysis tool, and that it was, in fact, accepted by the tool. Any portion of the code that does not conform to the tool restrictions or that is not processed by the tool for some other reason will have to be verified by some other method.

Objective A-5.4 – Source Code conforms to standards. Some abstract interpretation tools will check conformance to standard or user-defined coding rules. This may be used to fully or

partially satisfy the objective, depending upon the particular coding standards to be enforced. Tool qualification for checking conformance to standards would also be required.

Objective A-5.6 – Source Code is accurate and consistent. This objective is met by verifying the absence of run-time errors in the Source Code using an abstract interpretation tool. False alarms generated by the tool must be justified by separate analysis or testing. Any portion of the code for which the tool provides an "indeterminate" result must be verified through other methods.

Objective FM.A-5.10 Formal analysis cases and procedures are correct. This objective is met through review to ensure that the analyses and procedures satisfy the objectives for which credit is claimed. Since the properties to be checked are defined implicitly in the analysis tool, some aspects of tool qualification will be used to satisfy this objective.

Objective FM.A-5.11 Formal analysis results are correct and discrepancies explained. This objective is met through review of the analysis results to ensure that all of the code has been analyzed, and that any false alarms or indeterminate results from the tool have been justified through reviews or further analysis.

Objective FM.A-5.12 Requirements formalization is correct. This objective is met through tool qualification since the properties to be checked are implicit in the tool itself.

Objective FM.A-5.13 Formal method is correctly defined, justified, and appropriate. This objective is met through a review to ensure:

a. All notations used for formal analysis are verified to have precise, unambiguous, mathematically defined syntax and semantics. Abstract interpretation methods are based upon the formal semantics of the programming language to be analyzed.
b. The soundness of each formal analysis method is justified. Abstract interpretation tools may return an "indeterminate" result for some portions of the code, meaning that the tool was unable to conclusively determine that code to be error-free. Evidence of soundness should be provided through citations to publications addressing basis of soundness for the underlying analysis method.
c. Assumptions related to each formal analysis are described and justified. It is typical for proof of the absence of over/underflow to hold only if the program inputs stay within

defined ranges. These bounds are assumptions that must be documented, as described in the case study analysis.

In this case study, we have primarily demonstrated use of abstract interpretation for a specific objective, the absence of runtime errors as required by objective A-5.6, Source Code is accurate and consistent. The software tools we have used may also be used to partially satisfy a number of other objectives, including objectives related to Executable Object Code (EOC). In general, taking credit for EOC objectives in Table FM.A-6 based on activities performed on the source code requires that the equivalence of EOC and source code be established (see DO-333 paragraph 6.7.f).

Abstract interpretation tools have also been used to compute worst case execution time and stack usage. These analyses may be useful in satisfying other objectives such as those specified in FM.6.7.e and FM.A-6.5, Executable Object Code is compatible with target computer. These evaluations are outside the scope of the current case study.

5.2.3 Tool Qualification Issues

A DO-178C/DO-330 tool qualification kit is available for Polyspace from the vendor, MathWorks. MathWorks provides artifacts and evidence to support qualification under Criteria 2 since they suggest that the tool may be used to reduce object code verification processes in addition to automating source code verification processes. If this tool is being used to verify Level A or B software, this would map to TQL-4, while for Level C or D software this would map to TQL-5. The Polyspace 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.

5.3 Analysis of the Heading Control Law Source Code with Astrée

Astrée can be used to prove that no 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. To find

the places in the code where partitioning is needed, 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 (or *alarms*) in the source code corresponding to C statements which might cause floating-point overflow errors. The Astrée graphical user interface allows one to find the code line which corresponds to an alarm easily by clicking on the alarm message, as shown in Figure 125.

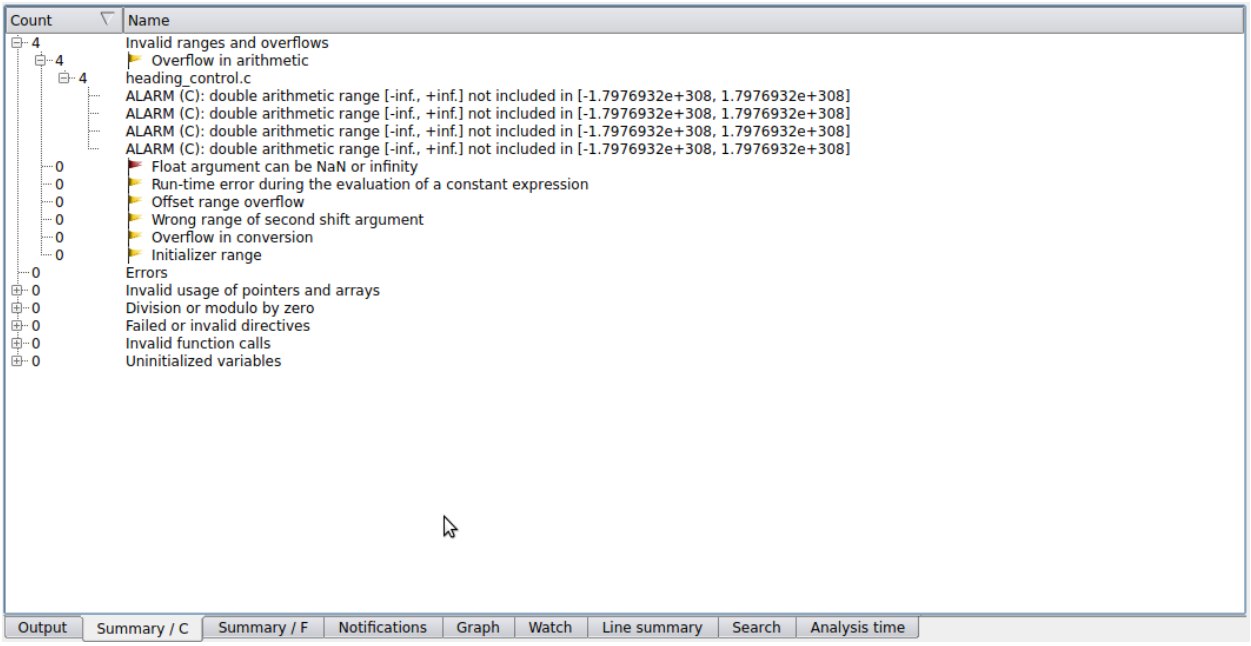

Figure 125 – Astrée Analysis Results

The alarms can be explained as follows: 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.

By analyzing the way the anti-windup-mechanism works, the necessary analysis partitions can be found. For every integrator, case distinctions need to be done according to two criteria: firstly, the case where the input value of the integrator is zero or positive must be distinguished from the case where the value is negative. Secondly, three cases must be distinguished depending on the

internal value of the integrator: the case where the value is within a certain interval must be distinguished from the case where it is above it and the case where it is below this interval.

```
__ASTREE_partition_control if (integrator >  25.01);
__ASTREE_partition_control if (integrator < -25.01);
__ASTREE_partition_control if (diff1 < 0.0);

sum1 = 0.04 * integrator + 0.15 * diff1;
```

Figure 126 – Astrée Directives to Define Partitions

Astrée is in general not able to provide direct feedback to show where the case partitions must be done, but this must be determined by the user. 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 handle this kind of program automatically, since new partitioning heuristics are being developed by AbsInt. However, it is not always possible to fine-tune the analysis in such a way that all false alarms disappear.

Astrée is also able to detect dead code, which is reported in the GUI by highlighting unreachable code lines. Astrée has some powerful features to detect dead code by determining for example that a given condition will always be evaluated to a constant value.

5.4 Analysis of the Heading Control Law Source Code with Polyspace

A summary of the results obtains from applying Polyspace to the Heading Control Law source code is shown in Table 50.

Table 50 – Initial Polyspace Analysis Results

Run-Time Checks	
Polyspace Verifier	Enabled
Number of Result Sets	x 1
Number of Red Run-Time Checks	0
Number of Gray Run-Time Checks	12
Number of Orange Run-Time Checks	13
Number of Green Run-Time Checks	478
Proven	97.4%
Pass/Fail	-

Possible runtime errors were detected in two categories indicated by color coding: Gray checks correspond to unreachable code and orange checks corresponding to unproven or potential runtime errors. No proven runtime errors (reported in red) were detected.

For the unreachable code, Polyspace reports an error such as:

```
if-condition always evaluates to false at line 779 (column 6)
      block ends at line 782 (column 2)
```

Upon further investigation, it was determined that all of the unreachable code was the result of branch conditions in the anti-windup logic for the integrators which always evaluate to false. The model constant 'gain_sign' is used to select a positive or negative value in the logic (see Figure 127). Therefore, once this constant is set in the application, the unused branch of the logic can be optimized away by either the code generator or the compiler, eliminating the unreachable code.

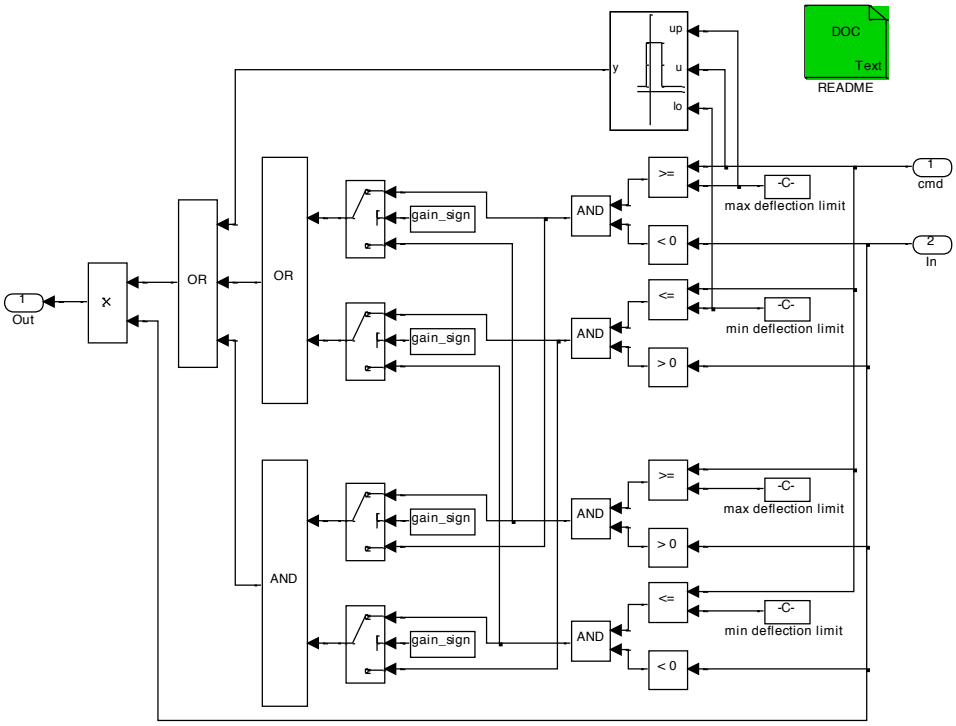

Figure 127 – Anti-Windup Logic

The unproven orange checks all corresponded to potential floating-point overflow errors (Table 51). Polyspace reports an error of the form:

```
heading_control_B.Sum1 = heading_control_U.ref_cmds[1]

OVFL.10 Unproven : operation [-] on float may overflow (on MIN or MAX bounds of
FLOAT64)
    If appropriate, applying Data Range Specification (DRS) to initialization of
    heading_control_U by the main generator, may remove this orange.
    If appropriate, applying DRS to initialization of heading_control_DWork by the
    main generator, may remove this orange.
    operator - on type float 64
         left:   full-range [-1.7977E+308 .. 1.7977E+308]
         right:  [-7.047E+306 .. 7.047E+306]
         result: full-range [-1.7977E+308 .. 1.7977E+308]
```

For the variables indicated, this means that it is not possible to guarantee that their values will not overflow unless some additional information about the system is provided. Polyspace provides a mechanism to specify range limits on inputs to the system (Data Range Specification, or DRS). 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 set up for each of the system inputs, the potential overflow errors are eliminated.

Table 51 – Unproven Runtime Checks

Check	Function	Line	Col.	Detail	Jus.	Class	Status
OVFL.10	heading_control_step()	65	57	Unproven : operation [-] on float may overflow (on MIN or MAX bounds of FLOAT64)	No	-	-

Polyspace can also be used to check code for conformance to standards. This can include standard coding rules (such as MISRA-C) or user-defined rules. This can be used to partially satisfy objective A-5.4, Source Code conforms to standards. To demonstrate, the heading control code was analyzed for conformance with the MISRA-C standard. The results are shown in Table 52.

Table 52 – Coding Rules Analysis

File	Warnings	Errors	Total
C:\rw_apps\polyspace_workspace\Heading_Control\source\heading_control.c	278	0	278
C:\rw_apps\polyspace_workspace\Heading_Control\include\rtwtypes.h	16	0	16
Total	**294**	**0**	**294**

6 Conclusion

The three case studies in this report illustrate 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.

7 References

[1] AbsInt, Astrée Run-Time Error Analyzer, http://www.absint.com/astree/index.htm.

[2] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter in *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, T. Mogensen and D.A. Schmidt and I.H. Sudborough (Editors). Lecture Notes in Computer Science, vol. 2566, pp. 85-108, Springer.

[3] 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.

[4] M. C. Escher, Relativity, http://en.wikipedia.org/wiki/Relativity_(M._C._Escher).

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

[6] Frama-C Software Analyzers, http://frama-c.com.

[7] Mike Gordon, Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware, Technical Report 77, Computer Laboratory, The University of Cambridge, 1985.

[8] Mike Gordon and Tom Melham, *Introduction to HOL: A theorem-proving environment for higher-order logic*, Cambridge University Press, 1993.

[9] George Hagen and 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.

[10] George Hagen. Verifying safety properties of Lustre programs: an SMT-based approach. PhD dissertation. Department of Computer Science. The University of Iowa. December 2008.

[11] John Harrison, HOL-Light: A Tutorial Introduction, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, LNCS, vol. 1166, pp. 265-269, Springer Verlag, 1996.

[12] HOL4, http://hol.sourceforge.net/.

[13] HOL Light, http://www.cl.cam.ac.uk/~jrh13/hol-light/.

[14] JKind, https://github.com/agacek/jkind.

[15] 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.

[16] IKOS: Inference Kernel for Open Static Analyzers, http://ti.arc.nasa.gov/opensource/ikos/.

[17] Isabelle/HOL, http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[18] Nancy Leveson, L. Denise Pinnel, Sean David Sandys, Shuichi Koga and Jon Damon Reese, Analyzing Software Specifications for Mode Confusion Potential, in *Proceedings of a Workshop on Human Error and System Development*, pages 132-146, 1997.

[19] MathWorks, DO Qualification Kit for Polyspace, http://www.mathworks.com/products/do-178/

[20] Thomas Melham, *Higher Order Logic and Hardware Verification*, Cambridge Tracts in Theoretical Computer Science, Number 31, Cambridge University Press, 1993.

[21] Steven P. Miller, Alan C. Tribble, Timothy M. Carlson, Eric J. Danielson, *Flight Guidance System Requirement Specification*, NASA Contractor Report CR-2003-212426, June 2003.

[22] Steven P. Miller, Elise A. Anderson, Lucas G. Wagner, and Michael W. Whalen. Mats P.E. Heimdahl, Formal Verification of Flight Critical Software, in *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, San Francisco, August 15-18, 2005.

[23] 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.

[24] M. Norrish and K. Slind, The HOL System: Logic, 1998-2013, http://hol.sourceforge.net/.

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

[26] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. Isabelle/HOL—A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283, Springer, 2002.

[27] ProofPower, http://www.lemma-one.com/ProofPower/index/index.html.

[28] 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.

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

[30] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert, *PVS Language System Guide 2.4*, SRI International, November 2001.

[31] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert, *PVS Language Reference Version 2.4*, SRI International, November 2001.

[32] S. Owre, N. Shankar, J. M. Rushby, D. W. J. Stringer-Calvert, *Prover Guide Version 2.4, SRI International*, November 2001.

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

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

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

[36] Trusted Extensions of Interactive Theorem Provers (TEITP) Workshop Summary Document. http://www.cs.utexas.edu/~kaufmann/itp-trusted-extensions-aug-2010/

Appendix A Acronyms

ADS	Air Data System
AFDX	Avionics Full-Duplex Switched Ethernet
AHRS	Attitude Heading Reference System
ALT	Altitude Hold Mode
ALTSEL	Altitude Select Mode
APPR	Approach
AP	Autopilot
BDD	Binary Decision Diagram
DCP	Display Control Panel
DRS	Data Range Specification
EOC	Executable Object Code
ERT	Embedded Real Time
FCP	Flight Control Panel
FCS	Flight Control System
FGS	Flight Guidance System
FD	Flight Director
FGS	Flight Guidance System
FLC	Flight Level Change Mode
FMS	Flight Management System
GA	Go Around
HDG	Heading Hold Mode
HOL	Higher Order Logic
IAS	Indicated Airspeed

ILS	Instrument Landing System
LAPPR	Lateral Approach Mode
LGA	Lateral Go Around Mode
NAV	Lateral Navigation Mode
PFD	Primary Flight Display
PITCH	Pitch Hold Mode
QSK	Qualification Support Kit
ROLL	Roll Hold Mode
RTW	Real-Time Workbench
SMT	Satisfiability Modulo Theories
TCC	Type-Correctness Condition
TOR	Tool Operational Requirements
TQL	Tool Qualification Level
TTA	Time-Triggered Architecture
UAV	Unmanned Aerial Vehicle
UMN	University of Minnesota
VAPPR	Vertical Approach Mode
VGA	Vertical Go Around Mode
VS	Vertical Speed Mode

Appendix B Mode Logic Properties

```
--=============================================================================================
-- SAFETY PROPERTIES
--=============================================================================================

-----------------------------------------------------------------------------------------------
-- At least one lateral mode shall always be active
-- when the FD is displayed or the AP is engaged.
-----------------------------------------------------------------------------------------------
 At_Least_One_Lateral_Mode_Active =
   ROLL_Active or HDG_Active or NAV_Active or LAPPR_Active or LGA_Active;

 check At_Least_One_Lateral_Mode_Active;

-----------------------------------------------------------------------------------------------
-- At most one lateral mode shall be active
-- when the FD is displayed or the AP is engaged.
-----------------------------------------------------------------------------------------------
 At_Most_One_Lateral_Mode_Active =
   (ROLL_Active   =>
     not (               HDG_Active or NAV_Active or LAPPR_Active or LGA_Active)) and
   (HDG_Active    =>
     not (ROLL_Active or               NAV_Active or LAPPR_Active or LGA_Active)) and
   (NAV_Active    =>
     not (ROLL_Active or HDG_Active or               LAPPR_Active or LGA_Active)) and
   (LAPPR_Active  =>
     not (ROLL_Active or HDG_Active or NAV_Active or                 LGA_Active)) and
   (LGA_Active    =>
     not (ROLL_Active or HDG_Active or NAV_Active or LAPPR_Active             ));

 check At_Most_One_Lateral_Mode_Active;

-----------------------------------------------------------------------------------------------
-- At least one vertical mode shall always be active
-- when the FD is displayed or the AP is engaged.
-----------------------------------------------------------------------------------------------
 At_Least_One_Vertical_Mode_Active =
   PITCH_Active or VS_Active or FLC_Active or ALT_Active or ALTSEL_Active or
   VAPPR_Active or VGA_Active;

 check At_Least_One_Vertical_Mode_Active;

-----------------------------------------------------------------------------------------------
-- At most one vertical mode shall be active
-- when the FD is displayed or the AP is engaged.
-----------------------------------------------------------------------------------------------
 At_Most_One_Vertical_Mode_Active =
   (PITCH_Active  =>
     not (                VS_Active or FLC_Active or ALT_Active or ALTSEL_Active or
          VAPPR_Active or VGA_Active)) and
   (VS_Active     =>
     not (PITCH_Active or              FLC_Active or ALT_Active or ALTSEL_Active or
          VAPPR_Active or VGA_Active)) and
   (FLC_Active    =>
     not (PITCH_Active or VS_Active or               ALT_Active or ALTSEL_Active or
          VAPPR_Active or VGA_Active)) and
   (ALT_Active    =>
     not (PITCH_Active or VS_Active or FLC_Active or               ALTSEL_Active or
          VAPPR_Active or VGA_Active)) and
   (ALTSEL_Active =>
```

```
    not (PITCH_Active or VS_Active or FLC_Active or ALT_Active or
         VAPPR_Active or VGA_Active)) and
  (VAPPR_Active  =>
    not (PITCH_Active or VS_Active or FLC_Active or ALT_Active or ALTSEL_Active or
                         VGA_Active)) and
  (VGA_Active    =>
    not (PITCH_Active or VS_Active or FLC_Active or ALT_Active or ALTSEL_Active or
         VAPPR_Active              ));

check At_Most_One_Vertical_Mode_Active;

-------------------------------------------------------------------------------
-- VGA mode shall be active if LGA mode is active (except for one step).
-------------------------------------------------------------------------------
LGA_Active_Implies_VGA_Active =
  pre LGA_Active and LGA_Active => pre VGA_Active;

check LGA_Active_Implies_VGA_Active;

-------------------------------------------------------------------------------
-- LGA mode shall be active if VGA mode is active (except for one step).
-------------------------------------------------------------------------------
VGA_Active_Implies_LGA_Active =
  pre VGA_Active and VGA_Active => pre LGA_Active;

check VGA_Active_Implies_LGA_Active;

-------------------------------------------------------------------------------
-- VAPPR mode shall be active only if LAPPR mode is active (except for one step).
-------------------------------------------------------------------------------
VAPPR_Active_Implies_LAPPR_Active =
  pre VAPPR_Active and VAPPR_Active => pre LAPPR_Active;

check VAPPR_Active_Implies_LAPPR_Active;

-------------------------------------------------------------------------------
-- FLC, ALT, ALTSEL, or PITCH mode shall be active
-- while an overspeed condition exists.
-------------------------------------------------------------------------------
Overspeed_Implies_FLC_ALT_ALTSEL_PITCH =
  Overspeed => FLC_Active or ALT_Active or ALTSEL_Active or PITCH_Active;

check Overspeed_Implies_FLC_ALT_ALTSEL_PITCH;

-------------------------------------------------------------------------------
-- PITCH mode shall be active for only one step while an overspeed condition exists.
-------------------------------------------------------------------------------
Overspeed_and_PITCH_Transitiory = true ->
  pre PITCH_Active and Overspeed => not PITCH_Active;

check Overspeed_and_PITCH_Transitiory;
```

```
--=============================================================================
-- FUNCTIONAL PROPERTIES
--=============================================================================

--=============================================================================
-- MODE ANNUNCIATIONS
--=============================================================================

-------------------------------------------------------------------------------
-- The mode annunciations shall be off at system start up.
-------------------------------------------------------------------------------
 Modes_Off_At_Startup =
   not Modes_On -> true;

 check Modes_Off_At_Startup;

-------------------------------------------------------------------------------
-- The mode annunciations shall be on if the AP is engaged.
-------------------------------------------------------------------------------
 AP_Engaged_Implies_Modes_On = true ->
   Is_AP_Engaged => Modes_On;

 check AP_Engaged_Implies_Modes_On;

 -------------------------------------------------------------------------------
-
-- The mode annunciations shall be on if the offside FD is on.
-------------------------------------------------------------------------------
 Offside_FD_On_Implies_Modes_On = true ->
   Offside_FD_On => Modes_On;

 check Offside_FD_On_Implies_Modes_On;

-------------------------------------------------------------------------------
-- The mode annunciations shall be on if the onside FD is on.
-------------------------------------------------------------------------------
 Onside_FD_On_Implies_Modes_On =
   FD_On => Modes_On;

 check Onside_FD_On_Implies_Modes_On;

-------------------------------------------------------------------------------
-- The mode annunciations shall be on if and only if the onside FD is on,
-- the offside FD is on, or the AP is engaged.
-------------------------------------------------------------------------------
 Modes_On_Iff_FD_On_or_AP_Engaged = true ->
   Modes_On = (FD_On or Offside_FD_On or Is_AP_Engaged);

 check Modes_On_Iff_FD_On_or_AP_Engaged;

--=============================================================================
-- FLIGHT DIRECTOR
--=============================================================================

-------------------------------------------------------------------------------
-- The onside FD shall be off at system start up
-------------------------------------------------------------------------------
 FD_Off_At_Startup = (not FD_On -> true);

 check FD_Off_At_Startup;

-------------------------------------------------------------------------------
-- The onside FD shall turn on when the FD switch is pressed.
```

```
--------------------------------------------------------------------------------
 FD_Switch_Turns_FD_On =
   not pre FD_On and RISING(FD_Switch)
   and No_Higher_Event_Than_FD_Switch_Pressed => FD_On;

 check FD_Switch_Turns_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall turn on when the AP is engaged.
--------------------------------------------------------------------------------
 AP_Engaged_Turns_FD_On =
   RISING(Is_AP_Engaged) => FD_On;

 check AP_Engaged_Turns_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall be on when an overspeed condition exists.
--------------------------------------------------------------------------------
 Overspeed_Implies_FD_On = true ->
   Overspeed => FD_On;

 check Overspeed_Implies_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall turn on when the HDG switch is pressed.
--------------------------------------------------------------------------------
 HDG_Switch_Turns_FD_On =
   RISING(HDG_Switch) and No_Higher_Event_Than_HDG_Switch_Pressed => FD_On;

 check HDG_Switch_Turns_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall turn on when the NAV switch is pressed.
--------------------------------------------------------------------------------
 NAV_Switch_Turns_FD_On =
   RISING(NAV_Switch) and No_Higher_Event_Than_NAV_Switch_Pressed => FD_On;

 check NAV_Switch_Turns_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall turn on when the APPR switch is pressed.
--------------------------------------------------------------------------------
 APPR_Switch_Turns_FD_On =
   RISING(APPR_Switch) and No_Higher_Event_Than_APPR_Switch_Pressed => FD_On;

 check APPR_Switch_Turns_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall turn on when the GA switch is pressed.
--------------------------------------------------------------------------------
 GA_Switch_Turns_FD_On =
   RISING(GA_Switch) and No_Higher_Event_Than_GA_Switch_Pressed => FD_On;

 check GA_Switch_Turns_FD_On;

--------------------------------------------------------------------------------
-- The onside FD shall turn on when the VS switch is pressed.
--------------------------------------------------------------------------------
 VS_Switch_Turns_FD_On =
   RISING(VS_Switch)and No_Higher_Event_Than_VS_Switch_Pressed => FD_On;

 check VS_Switch_Turns_FD_On;
```

```
-------------------------------------------------------------------------------
-- The onside FD shall turn on when the FLC switch is pressed.
-------------------------------------------------------------------------------
 FLC_Switch_Turns_FD_On =
   RISING(FLC_Switch) and No_Higher_Event_Than_FLC_Switch_Pressed => FD_On;

 check FLC_Switch_Turns_FD_On;

-------------------------------------------------------------------------------
-- The onside FD shall turn on when the ALT switch is pressed.
-------------------------------------------------------------------------------
 ALT_Switch_Turns_FD_On =
   RISING(ALT_Switch) and No_Higher_Event_Than_ALT_Switch_Pressed => FD_On;

 check ALT_Switch_Turns_FD_On;

-------------------------------------------------------------------------------
-- The onside FD shall turn on when the VS Pitch Wheel is rotated
-- while this side is active and VS and VAPPR mode are not active
-- and an overspeed condition does not exist.
-------------------------------------------------------------------------------
 VS_Pitch_Wheel_Rotated_Turns_FD_On =
   RISING(VS_Pitch_Wheel_Rotated)
   and pre Active_Side
   and not pre VS_Active and not pre VAPPR_Active
   and not Overspeed
   and No_Higher_Event_Than_VS_Pitch_Wheel_Rotated => FD_On;

 check VS_Pitch_Wheel_Rotated_Turns_FD_On;

-------------------------------------------------------------------------------
-- The onside FD shall turn on when the ALTSEL target altitude is changed
-- while this side is active and ALTSEL mode is active.
-------------------------------------------------------------------------------
 ALTSEL_Target_Changed_Turns_FD_On =
   RISING(ALTSEL_Target_Changed)and pre Active_Side and pre ALTSEL_Active
   and No_Higher_Event_Than_ALTSEL_Target_Changed => FD_On;

 check ALTSEL_Target_Changed_Turns_FD_On;

-------------------------------------------------------------------------------
-
-- The onside FD shall turn on when there is a pilot flying transfer
-- to this side of the aircraft while the mode annunciations are on.
-------------------------------------------------------------------------------
 Pilot_Flying_Transfer_While_Modes_On_Turns_FD_On =
   pre not FD_On and pre Modes_On and RISING(Pilot_Flying_Side) => FD_On;

 check Pilot_Flying_Transfer_While_Modes_On_Turns_FD_On;

-------------------------------------------------------------------------------
-
-- The onside FD shall turn off when the FD switch is pressed
-- while an overspeed condition does not exist.
-------------------------------------------------------------------------------
 FD_Switch_Turns_FD_Off =
   pre FD_On and RISING(FD_Switch) and not Overspeed
   and No_Higher_Event_Than_FD_Switch_Pressed => not FD_On;

 check FD_Switch_Turns_FD_Off;

-------------------------------------------------------------------------------
-
```

```
-- Disengaging the AP shall not turn off the FD unless the FD switch is pressed
-------------------------------------------------------------------------------
 FD_Stays_On_When_AP_Disengaged = true ->
   pre FD_On and not FALLING(Is_AP_Engaged) and not RISING(FD_Switch) => FD_On;

 check FD_Stays_On_When_AP_Disengaged;

-------------------------------------------------------------------------------
-
-- The onside FD shall not turn off unless the FD switch is pressed.
-------------------------------------------------------------------------------
 FD_Stays_On_Unless_FD_Switch_Pressed = true ->
   pre FD_On and not RISING(FD_Switch) =>  FD_On;

 check FD_Stays_On_Unless_FD_Switch_Pressed;

--=============================================================================
-- INDEPENDENT MODE
--=============================================================================

-------------------------------------------------------------------------------
-
-- Independent mode shall be active when VAPPR mode is active on both sides.
-------------------------------------------------------------------------------
 Both_Sides_VAPPR_Active_Implies_Independent_Mode = true ->
   pre VAPPR_Active and Is_Offside_VAPPR_Active => Independent_Mode;

 check Both_Sides_VAPPR_Active_Implies_Independent_Mode;

-------------------------------------------------------------------------------
-
-- Independent mode shall be active when VGA mode is active on both sides.
-------------------------------------------------------------------------------
 Both_Sides_VGA_Active_Implies_Independent_Mode = true ->
   pre VGA_Active and Is_Offside_VGA_Active => Independent_Mode;

 check Both_Sides_VGA_Active_Implies_Independent_Mode;

-------------------------------------------------------------------------------
-
-- Independent mode shall imply either VAPPR or VGA is active on both sides.
-------------------------------------------------------------------------------
 Independent_Mode_Implies_Both_VAPPR_or_VGA_Active =
   Independent_Mode =>
        (pre VAPPR_Active and Is_Offside_VAPPR_Active) or
        (pre VGA_Active   and Is_Offside_VGA_Active);

 check Independent_Mode_Implies_Both_VAPPR_or_VGA_Active;

--=============================================================================
-- ACTIVE SIDE
--=============================================================================

-------------------------------------------------------------------------------
-- This side shall be active iff it is in independent mode or the pilot flying side.
-------------------------------------------------------------------------------
 Active_iff_Independent_or_Pilot_Flying =
   Active_Side = Independent_Mode or Pilot_Flying_Side;

 check Active_iff_Independent_or_Pilot_Flying;

--=============================================================================
-- ROLL MODE
```

```
--=============================================================================

-------------------------------------------------------------------------------
-- ROLL mode shall be active if and only if ROLL mode is selected.
-------------------------------------------------------------------------------
 ROLL_Selected_Iff_ROLL_Active =
   (ROLL_Active = ROLL_Selected);

 check ROLL_Selected_Iff_ROLL_Active;

-------------------------------------------------------------------------------
-- ROLL mode shall be active iff no other lateral mode is active.
-------------------------------------------------------------------------------
 Default_Lateral_Mode_Is_ROLL = true ->
   ROLL_Active =
     not (HDG_Active or NAV_Active or LAPPR_Active or LGA_Active);

 check Default_Lateral_Mode_Is_ROLL;

-------------------------------------------------------------------------------
-- ROLL mode shall be active if the mode annunciations are off.
-------------------------------------------------------------------------------
 Modes_Off_Implies_ROLL_Active = true ->
    not Modes_On => ROLL_Active;

 check Modes_Off_Implies_ROLL_Active;

-------------------------------------------------------------------------------
-- ROLL mode shall be active if the FD switch is pressed
-- while the mode annunciations are off.
-------------------------------------------------------------------------------
 FD_Switch_Pressed_Modes_Off_Implies_ROLL_Active =
   not Modes_On and RISING(FD_Switch)
   and No_Higher_Event_Than_FD_Switch_Pressed => ROLL_Active;

 check FD_Switch_Pressed_Modes_Off_Implies_ROLL_Active;

-------------------------------------------------------------------------------
-- ROLL mode shall be active if the AP is engaged
-- while the mode annunciations are off.
-------------------------------------------------------------------------------
 AP_Engaged_Modes_Off_Implies_ROLL_Active =
   not Modes_On and RISING(Is_AP_Engaged) => ROLL_Active;

 check AP_Engaged_Modes_Off_Implies_ROLL_Active;

--=============================================================================
-- HDG MODE
--=============================================================================

-------------------------------------------------------------------------------
-- HDG mode shall be selected if and only if HDG mode is active.
-------------------------------------------------------------------------------
 HDG_Selected_Iff_HDG_Active =
     HDG_Active = HDG_Selected;

 check HDG_Selected_Iff_HDG_Active;

-------------------------------------------------------------------------------
-- HDG mode shall be selected if the HDG switch is pressed while HDG mode is cleared.
-------------------------------------------------------------------------------
 HDG_Switch_Pressed_Selects_HDG =
   not pre HDG_Selected and RISING(HDG_Switch)
```

```
   and No_Higher_Event_Than_HDG_Switch_Pressed => HDG_Selected;

 check HDG_Switch_Pressed_Selects_HDG;

-------------------------------------------------------------------------------------
-- HDG mode shall be cleared if the HDG switch is pressed while HDG mode is selected.
-------------------------------------------------------------------------------------
 HDG_Switch_Pressed_Clears_HDG =
   pre HDG_Selected and RISING(HDG_Switch)
   and No_Higher_Event_Than_HDG_Switch_Pressed => not HDG_Selected;

 check HDG_Switch_Pressed_Clears_HDG;

-------------------------------------------------------------------------------------
-- HDG mode shall be cleared when there is a pilot flying transfer.
-------------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_HDG =
   pre HDG_Selected and CHANGED(Pilot_Flying_Side)  => not HDG_Selected;

 check Pilot_Flying_Transfer_Clears_HDG;

-------------------------------------------------------------------------------------
-- HDG mode shall be cleared when the mode annunciations are turned off.
-------------------------------------------------------------------------------------
 Modes_Off_Clears_HDG =
   pre HDG_Selected and not Modes_On  => not HDG_Selected;

 check Modes_Off_Clears_HDG;

--=====================================================================================
-- NAV MODE
--=====================================================================================

-------------------------------------------------------------------------------------
-- NAV mode shall be selected if NAV mode is active.
-------------------------------------------------------------------------------------
 NAV_Selected_If_NAV_Active =
     NAV_Active => NAV_Selected;

 check NAV_Selected_If_NAV_Active;

-------------------------------------------------------------------------------------
-- NAV mode shall be selected if the NAV switch is pressed while NAV mode is cleared.
-------------------------------------------------------------------------------------
 NAV_Switch_Pressed_Selects_NAV =
   not pre NAV_Selected and RISING(NAV_Switch)
   and No_Higher_Event_Than_NAV_Switch_Pressed => NAV_Selected;

 check NAV_Switch_Pressed_Selects_NAV;

-------------------------------------------------------------------------------------
-- NAV mode shall become active if the NAV capture condition is met
--  while NAV mode is armed.
-------------------------------------------------------------------------------------
--   NAV_Active_When_Capture_Cond_Met = true ->
--     pre NAV_Selected and not pre NAV_Active
--     and NAV_Capture_Cond_Met
--     and not Selected_NAV_Source_Changed
--     and not Selected_NAV_Frequency_Changed
--     and not CHANGED(Pilot_Flying_Side)
--     and Modes_On
--     and No_Higher_Event_Than_NAV_Capture_Cond_Met => NAV_Active;
```

```
--   check NAV_Active_When_Capture_Cond_Met;

-------------------------------------------------------------------------------
-- NAV mode shall be cleared if the NAV switch is pressed while NAV mode is selected.
-------------------------------------------------------------------------------
 NAV_Switch_Pressed_Clears_NAV =
   pre NAV_Selected and RISING(NAV_Switch)
   and No_Higher_Event_Than_NAV_Switch_Pressed => not NAV_Selected;

 check NAV_Switch_Pressed_Clears_NAV;

-------------------------------------------------------------------------------
-- NAV mode shall be cleared if the selected NAV source is changed.
-------------------------------------------------------------------------------
 Selected_NAV_Source_Changed_Clears_NAV =
   pre NAV_Selected
   and Selected_NAV_Source_Changed => not NAV_Selected;

 check Selected_NAV_Source_Changed_Clears_NAV;

-------------------------------------------------------------------------------
-- NAV mode shall be cleared if the selected NAV frequency is changed.
-------------------------------------------------------------------------------
 Selected_NAV_Frequency_Changed_Clears_NAV =
   pre NAV_Selected
   and Selected_NAV_Frequency_Changed => not NAV_Selected;

 check Selected_NAV_Frequency_Changed_Clears_NAV;

-------------------------------------------------------------------------------
-- NAV mode shall be cleared when there is a pilot flying transfer.
-------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_NAV =
   pre NAV_Selected and CHANGED(Pilot_Flying_Side)  => not NAV_Selected;

 check Pilot_Flying_Transfer_Clears_NAV;

-------------------------------------------------------------------------------
-- NAV mode shall be cleared when the mode annunciations are turned off.
-------------------------------------------------------------------------------
 Modes_Off_Clears_NAV =
   pre NAV_Selected and not Modes_On  => not NAV_Selected;

 check Modes_Off_Clears_NAV;

--===============================================================================
-- LAPPR MODE
--===============================================================================

-------------------------------------------------------------------------------
-- LAPPR mode shall be selected if LAPPR mode is active.
-------------------------------------------------------------------------------
 LAPPR_Selected_If_LAPPR_Active =
     LAPPR_Active => LAPPR_Selected;

 check LAPPR_Selected_If_LAPPR_Active;

-------------------------------------------------------------------------------
-- LAPPR mode shall be selected if the APPR switch is pressed
-- while LAPPR mode is cleared.
-------------------------------------------------------------------------------
 APPR_Switch_Pressed_Selects_LAPPR =
   not pre LAPPR_Selected and RISING(APPR_Switch)
```

```
   and No_Higher_Event_Than_APPR_Switch_Pressed => LAPPR_Selected;

 check APPR_Switch_Pressed_Selects_LAPPR;

-------------------------------------------------------------------------------------
-- LAPPR mode shall become active if the LAPPR capture condition is met
-- while LAPPR mode is armed.
-------------------------------------------------------------------------------------
--   LAPPR_Active_When_Capture_Cond_Met = true ->
--     pre LAPPR_Selected and not pre LAPPR_Active
--     and LAPPR_Capture_Cond_Met
--     and not Selected_NAV_Source_Changed
--     and not Selected_NAV_Frequency_Changed
--     and not CHANGED(Pilot_Flying_Side)
--     and Modes_On
--     and No_Higher_Event_Than_LAPPR_Capture_Cond_Met => LAPPR_Active;

--   check LAPPR_Active_When_Capture_Cond_Met;

-------------------------------------------------------------------------------------
-- LAPPR mode shall be cleared if the APPR switch is pressed
-- while LAPPR mode is selected.
-------------------------------------------------------------------------------------
 APPR_Switch_Pressed_Clears_LAPPR =
   pre LAPPR_Selected and RISING(APPR_Switch)
   and No_Higher_Event_Than_APPR_Switch_Pressed => not LAPPR_Selected;

 check APPR_Switch_Pressed_Clears_LAPPR;

-------------------------------------------------------------------------------------
-- LAPPR mode shall be cleared if the selected NAV source is changed.
-------------------------------------------------------------------------------------
 Selected_NAV_Source_Changed_Clears_LAPPR =
   pre LAPPR_Selected
   and Selected_NAV_Source_Changed => not LAPPR_Selected;

 check Selected_NAV_Source_Changed_Clears_LAPPR;

-------------------------------------------------------------------------------------
-- LAPPR mode shall be cleared if the selected NAV frequency is changed.
-------------------------------------------------------------------------------------
 Selected_NAV_Frequency_Changed_Clears_LAPPR =
   pre LAPPR_Selected
   and Selected_NAV_Frequency_Changed => not LAPPR_Selected;

 check Selected_NAV_Frequency_Changed_Clears_LAPPR;

-------------------------------------------------------------------------------------
-- LAPPR mode shall be cleared when there is a pilot flying transfer.
-------------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_LAPPR =
   pre LAPPR_Selected and CHANGED(Pilot_Flying_Side) => not LAPPR_Selected;

 check Pilot_Flying_Transfer_Clears_LAPPR;

-------------------------------------------------------------------------------------
-- LAPPR mode shall be cleared when the mode annunciations are turned off.
-------------------------------------------------------------------------------------
 Modes_Off_Clears_LAPPR =
   pre LAPPR_Selected and not Modes_On => not LAPPR_Selected;

 check Modes_Off_Clears_LAPPR;
```

```
--=============================================================================================
-- LGA MODE
--=============================================================================================

---------------------------------------------------------------------------------------------
-- LGA mode shall be selected if and only if LGA mode is active.
---------------------------------------------------------------------------------------------
 LGA_Selected_Iff_LGA_Active =
     LGA_Active = LGA_Selected;

 check LGA_Selected_Iff_LGA_Active;

---------------------------------------------------------------------------------------------
-- LGA mode shall be selected if the GA switch is pressed while LGA mode is cleared
-- and an overspeed condition does not exist.
---------------------------------------------------------------------------------------------
 GA_Switch_Pressed_Selects_LGA =
   not pre LGA_Selected and RISING(GA_Switch) and not Overspeed
   and No_Higher_Event_Than_GA_Switch_Pressed => LGA_Selected;

 check GA_Switch_Pressed_Selects_LGA;

---------------------------------------------------------------------------------------------
-- LGA mode shall be cleared when the AP is engaged.
---------------------------------------------------------------------------------------------
 AP_Engaged_Clears_LGA =
   pre LGA_Selected and RISING(Is_AP_Engaged) => not LGA_Selected;

 check AP_Engaged_Clears_LGA;

---------------------------------------------------------------------------------------------
-- LGA mode shall be cleared when VGA mode is cleared.
---------------------------------------------------------------------------------------------
 VGA_Cleared_Clears_LGA =
   pre LGA_Selected and not pre VGA_Selected => not LGA_Selected;

 check VGA_Cleared_Clears_LGA;

---------------------------------------------------------------------------------------------
-- LGA mode shall be cleared when the SYNC switch is pressed
-- while LGA mode is selected.
---------------------------------------------------------------------------------------------
 SYNC_Switch_Pressed_Clears_LGA =
   pre LGA_Selected and RISING(SYNC_Switch) => not LGA_Selected;

 check SYNC_Switch_Pressed_Clears_LGA;

 ---------------------------------------------------------------------------------------------
-
-- LGA mode shall be cleared when there is a pilot flying transfer.
---------------------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_LGA =
   pre LGA_Selected and CHANGED(Pilot_Flying_Side) => not LGA_Selected;

 check Pilot_Flying_Transfer_Clears_LGA;

---------------------------------------------------------------------------------------------
-- LGA mode shall be cleared when the mode annunciations are turned off.
---------------------------------------------------------------------------------------------
 Modes_Off_Clears_LGA =
   pre LGA_Selected and not Modes_On  => not LGA_Selected;

 check Modes_Off_Clears_LGA;
```

```
--=============================================================================================
-- PITCH MODE
--=============================================================================================

-----------------------------------------------------------------------------------------------
-- PITCH mode shall be active iff no other vertical mode is active.
-----------------------------------------------------------------------------------------------
 Default_Vertical_Mode_Is_PITCH = true ->
   PITCH_Active = not (VS_Active or FLC_Active or ALT_Active or
                       ALTSEL_Active or VAPPR_Active or VGA_Active);

 check Default_Vertical_Mode_Is_PITCH;

-----------------------------------------------------------------------------------------------
-- PITCH mode shall be active if the mode annunciations are off.
-----------------------------------------------------------------------------------------------
 Modes_Off_Implies_PITCH_Active = true ->
   not Modes_On => PITCH_Active;

 check Modes_Off_Implies_PITCH_Active;

-----------------------------------------------------------------------------------------------
-- PITCH mode shall be active if the FD switch is pressed
-- while the mode annunciations are off.
-----------------------------------------------------------------------------------------------
 FD_Switch_Pressed_Modes_Off_Implies_PITCH_Active =
   not Modes_On and RISING(FD_Switch)
   and No_Higher_Event_Than_FD_Switch_Pressed => PITCH_Active;

 check FD_Switch_Pressed_Modes_Off_Implies_PITCH_Active;

-----------------------------------------------------------------------------------------------
-- PITCH mode shall be active if the AP is engaged
-- while the mode annunciations are off.
-----------------------------------------------------------------------------------------------
 AP_Engaged_Modes_Off_Implies_PITCH_Active =
   not Modes_On and RISING(Is_AP_Engaged) => PITCH_Active;

 check AP_Engaged_Modes_Off_Implies_PITCH_Active;

--=============================================================================================
-- VS MODE
--=============================================================================================

-----------------------------------------------------------------------------------------------
-- VS mode shall be selected if and only if VS mode is active.
-----------------------------------------------------------------------------------------------
 VS_Selected_Iff_VS_Active =
     (VS_Active = VS_Selected);

 check VS_Selected_Iff_VS_Active;

-----------------------------------------------------------------------------------------------
-- VS mode shall be selected if the VS switch is pressed while VS mode is cleared
-- if VAPPR mode is not active and an overspeed condition does not exist.
-----------------------------------------------------------------------------------------------
 VS_Switch_Pressed_Selects_VS =
   not pre VS_Selected and RISING(VS_Switch)
   and not pre VAPPR_Active and not Overspeed
   and No_Higher_Event_Than_VS_Switch_Pressed => VS_Selected;

 check VS_Switch_Pressed_Selects_VS;
```

```
-------------------------------------------------------------------------------
-- VS mode shall be cleared if the VS switch is pressed while VS mode is selected.
-------------------------------------------------------------------------------
 VS_Switch_Pressed_Clears_VS =
   pre VS_Selected and RISING(VS_Switch)
   and No_Higher_Event_Than_VS_Switch_Pressed => not VS_Selected;

 check VS_Switch_Pressed_Clears_VS;

-------------------------------------------------------------------------------
-- VS mode shall be cleared when there is a pilot flying transfer.
-------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_VS =
   pre VS_Selected and CHANGED(Pilot_Flying_Side)  => not VS_Selected;

 check Pilot_Flying_Transfer_Clears_VS;

-------------------------------------------------------------------------------
-- VS mode shall be cleared when the mode annunciations are turned off.
-------------------------------------------------------------------------------
 Modes_Off_Clears_VS =
   pre VS_Selected and not Modes_On  => not VS_Selected;

 check Modes_Off_Clears_VS;

--===============================================================================
-- FLC MODE
--===============================================================================

-------------------------------------------------------------------------------
-- FLC mode shall be selected if and only if FLC mode is active.
-------------------------------------------------------------------------------
 FLC_Selected_Iff_FLC_Active =
     (FLC_Active = FLC_Selected);

 check FLC_Selected_Iff_FLC_Active;

-------------------------------------------------------------------------------
-- FLC mode shall be selected if the FLC switch is pressed while FLC mode is cleared
-- if VAPPR mode is not active.
-------------------------------------------------------------------------------
 FLC_Switch_Pressed_Selects_FLC =
   not pre FLC_Selected and RISING(FLC_Switch)
   and not pre VAPPR_Active
   and No_Higher_Event_Than_FLC_Switch_Pressed => FLC_Selected;

 check FLC_Switch_Pressed_Selects_FLC;

-------------------------------------------------------------------------------
-- FLC mode shall be activated if an overspeed condition occurs
-- while neither ALT or ALTSEL are active.
-------------------------------------------------------------------------------
 Overspeed_Activates_FLC = true ->
   not pre FLC_Active
   and Overspeed
   and not pre ALT_Active and not ALT_Active
   and not pre ALTSEL_Active and not ALTSEL_Active => FLC_Selected;

 check Overspeed_Activates_FLC;

-------------------------------------------------------------------------------
-- FLC mode shall be cleared if the FLC switch is pressed while FLC mode is selected
```

```
-- and an overspeed condition does not exist.
-------------------------------------------------------------------------------
 FLC_Switch_Pressed_Clears_FLC =
   pre FLC_Selected and RISING(FLC_Switch)
   and not Overspeed
   and No_Higher_Event_Than_FLC_Switch_Pressed => not FLC_Selected;

 check FLC_Switch_Pressed_Clears_FLC;

-------------------------------------------------------------------------------
-- FLC mode shall be cleared if the VS Pitch Wheel is rotated while
-- an overspeed condition does not exist.
-------------------------------------------------------------------------------
 VS_Pitch_Wheel_Rotated_Clears_FLC =
   pre FLC_Selected
   and RISING(VS_Pitch_Wheel_Rotated)
   and not Overspeed
   and No_Higher_Event_Than_VS_Pitch_Wheel_Rotated => not FLC_Selected;

 check VS_Pitch_Wheel_Rotated_Clears_FLC;

-------------------------------------------------------------------------------
-- FLC mode shall be cleared when there is a pilot flying transfer.
-------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_FLC =
   pre FLC_Selected and CHANGED(Pilot_Flying_Side) => not FLC_Selected;

 check Pilot_Flying_Transfer_Clears_FLC;

-------------------------------------------------------------------------------
-- FLC mode shall be cleared when the mode annunciations are turned off.
-------------------------------------------------------------------------------
 Modes_Off_Clears_FLC =
   pre FLC_Selected and not Modes_On  => not FLC_Selected;

 check Modes_Off_Clears_FLC;

--===============================================================================
-- ALT MODE
--===============================================================================

-------------------------------------------------------------------------------
-- ALT mode shall be selected if and only if ALT mode is active.
-------------------------------------------------------------------------------
 ALT_Selected_Iff_ALT_Active =
   (ALT_Active = ALT_Selected);

 check ALT_Selected_Iff_ALT_Active;

-------------------------------------------------------------------------------
-- ALT mode shall be selected if the ALT switch is pressed while ALT mode is cleared
-- while VAPPR mode is not active.
-------------------------------------------------------------------------------
 ALT_Switch_Pressed_Selects_ALT =
   not pre ALT_Selected and RISING(ALT_Switch) and not pre VAPPR_Active
   and No_Higher_Event_Than_ALT_Switch_Pressed => ALT_Selected;

 check ALT_Switch_Pressed_Selects_ALT;

-------------------------------------------------------------------------------
-- ALT mode shall be selected if the ALTSEL target is changed
-- while in ALTSEL Track mode.
-------------------------------------------------------------------------------
```

```
 ALTSEL_Target_Changed_Selects_ALT =
   pre ALTSEL_Track and RISING(ALTSEL_Target_Changed)
   and No_Higher_Event_Than_ALTSEL_Target_Changed => ALT_Selected;

 check ALTSEL_Target_Changed_Selects_ALT;

-------------------------------------------------------------------------------
-- ALT mode shall be cleared if the ALT switch is pressed while ALT mode is selected.
-------------------------------------------------------------------------------
 ALT_Switch_Pressed_Clears_ALT =
   pre ALT_Selected and RISING(ALT_Switch)
   and No_Higher_Event_Than_ALT_Switch_Pressed => not ALT_Selected;

 check ALT_Switch_Pressed_Clears_ALT;

-------------------------------------------------------------------------------
-- ALT mode shall be cleared if the VS Pitch Wheel is rotated
-- while ALT mode is selected.
-------------------------------------------------------------------------------
 VS_Pitch_Wheel_Rotated_Clears_ALT =
   pre ALT_Selected and RISING(VS_Pitch_Wheel_Rotated)
   and No_Higher_Event_Than_VS_Pitch_Wheel_Rotated => not ALT_Selected;

 check VS_Pitch_Wheel_Rotated_Clears_ALT;

-------------------------------------------------------------------------------
-- ALT mode shall be cleared when there is a pilot flying transfer.
-------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_ALT =
   pre ALT_Selected and CHANGED(Pilot_Flying_Side)  => not ALT_Selected;

 check Pilot_Flying_Transfer_Clears_ALT;

-------------------------------------------------------------------------------
-- ALT mode shall be cleared when the mode annunciations are turned off.
-------------------------------------------------------------------------------
 Modes_Off_Clears_ALT =
   pre ALT_Selected and not Modes_On  => not ALT_Selected;

 check Modes_Off_Clears_ALT;

--===============================================================================
-- ALTSEL MODE
--===============================================================================

-------------------------------------------------------------------------------
-- ALTSEL mode shall be selected if ALTSEL mode is active.
-------------------------------------------------------------------------------
 ALTSEL_Selected_If_ALTSEL_Active =
     (ALTSEL_Active => ALTSEL_Selected);

 check ALTSEL_Selected_If_ALTSEL_Active;

-------------------------------------------------------------------------------
-- ALTSEL mode shall be active if ALTSEL is tracking the target altitude.
-------------------------------------------------------------------------------
 ALTSEL_Active_If_ALTSEL_Track =
   (ALTSEL_Track => ALTSEL_Active);

 check ALTSEL_Active_If_ALTSEL_Track;

-------------------------------------------------------------------------------
-- If the mode annunciations are on, ALTSEL mode shall be selected if
```

```
-- none of ALT, VAPPR, or VGA mode are active.
-------------------------------------------------------------------------------
 ALTSEL_Selected_If_Not_ALT_VAPPR_VGA_Active = true ->
   Modes_On and not (ALT_Active or VAPPR_Active or VGA_Active) => ALTSEL_Selected;

 check ALTSEL_Selected_If_Not_ALT_VAPPR_VGA_Active;

-------------------------------------------------------------------------------
-- ALTSEL mode shall become active if the ALTSEL capture condition is met
-- while ALTSEL mode is armed
-------------------------------------------------------------------------------
--   ALTSEL_Active_When_Capture_Cond_Met = true ->
--     pre ALTSEL_Selected and not pre ALTSEL_Active
--     and ALTSEL_Capture_Cond_Met
--     and not (ALT_Active or VAPPR_Active or VGA_Active)
--     and Modes_On
--     and No_Higher_Event_Than_ALTSEL_Capture_Cond_Met => ALTSEL_Active;

--   check ALTSEL_Active_When_Capture_Cond_Met;

-------------------------------------------------------------------------------
-- ALTSEL mode shall start tracking if the ALTSEL track condition is met
-- while ALTSEL
-- capturing the target altitude
-------------------------------------------------------------------------------
--   ALTSEL_Track_When_Track_Cond_Met = true ->
--     pre ALTSEL_Active and not pre ALTSEL_Track
--     and ALTSEL_Track_Cond_Met
--     and not ALTSEL_Target_Changed
--     and not VS_Pitch_Wheel_Rotated
--     and not CHANGED(Pilot_Flying_Side)
--     and ALTSEL_Active
--     and not (ALT_Active or VAPPR_Active or VGA_Active)
--     and Modes_On
--     and No_Higher_Event_Than_ALTSEL_Track_Cond_Met => ALTSEL_Track;

--   check ALTSEL_Track_When_Track_Cond_Met;

-------------------------------------------------------------------------------
-- ALTSEL mode shall revert to armed mode if the ALTSEL target is changed
-- while in capture mode.
-------------------------------------------------------------------------------
 ALTSEL_Deactivated_When_Target_Changed =
   pre ALTSEL_Active and not pre ALTSEL_Track
   and RISING(ALTSEL_Target_Changed)
   and not (ALT_Active or VAPPR_Active or VGA_Active)
   and Modes_On
   and No_Higher_Event_Than_ALTSEL_Target_Changed
       => (ALTSEL_Selected and not ALTSEL_Active);

 check ALTSEL_Deactivated_When_Target_Changed;

-------------------------------------------------------------------------------
-- ALTSEL mode shall revert to armed mode if the VS Pitch Wheel is rotated
-- while in capture mode.
-------------------------------------------------------------------------------
 ALTSEL_Deactivated_When_VS_Pitch_Wheel_Rotated =
   pre ALTSEL_Active and not pre ALTSEL_Track
   and RISING(VS_Pitch_Wheel_Rotated)
   and not (ALT_Active or VAPPR_Active or VGA_Active)
   and Modes_On
   and No_Higher_Event_Than_VS_Pitch_Wheel_Rotated
       => (ALTSEL_Selected and not ALTSEL_Active);
```

```
 check ALTSEL_Deactivated_When_VS_Pitch_Wheel_Rotated;

-------------------------------------------------------------------------------
-- ALTSEL mode shall revert to armed mode if there is a pilot flying transfer
-- while in capture mode.
-------------------------------------------------------------------------------
 ALTSEL_Deactivated_When_Pilot_Flying_Transfer =
   pre ALTSEL_Active and not pre ALTSEL_Track
   and CHANGED(Pilot_Flying_Side)
   and not (ALT_Active or VAPPR_Active or VGA_Active)
   and Modes_On => (ALTSEL_Selected and not ALTSEL_Active);

 check ALTSEL_Deactivated_When_Pilot_Flying_Transfer;

-------------------------------------------------------------------------------
-- ALTSEL mode shall revert to armed mode if a new vertical mode becomes active
-- while in capture mode.
-------------------------------------------------------------------------------
 ALTSEL_Deactivated_When_New_Active_Vertical_Mode =
   pre ALTSEL_Active and not pre ALTSEL_Track
   and not ALTSEL_Active
   and not (ALT_Active or VAPPR_Active or VGA_Active)
   and Modes_On => (ALTSEL_Selected and not ALTSEL_Active);

 check ALTSEL_Deactivated_When_New_Active_Vertical_Mode;

-------------------------------------------------------------------------------
-- If the mode annunciations are on, ALTSEL mode shall be cleared if
-- any of ALT, VAPPR, or VGA mode become active.
-------------------------------------------------------------------------------
 ALTSEL_Cleared_If_ALT_VAPPR_VGA_Active =
   Modes_On and (ALT_Active or VAPPR_Active or VGA_Active) => not ALTSEL_Selected;

 check ALTSEL_Cleared_If_ALT_VAPPR_VGA_Active;


--===============================================================================
-- VAPPR MODE
--===============================================================================

-------------------------------------------------------------------------------
-- VAPPR mode shall be selected if VAPPR mode is active.
-------------------------------------------------------------------------------
 VAPPR_Selected_If_VAPPR_Active =
   (VAPPR_Active => VAPPR_Selected);

 check VAPPR_Selected_If_VAPPR_Active;

-------------------------------------------------------------------------------
-- VAPPR mode shall be selected if the APPR switch is pressed
-- while VAPPR mode is cleared.
-------------------------------------------------------------------------------
 APPR_Switch_Pressed_Selects_VAPPR =
   not pre VAPPR_Selected and RISING(APPR_Switch)
   and No_Higher_Event_Than_APPR_Switch_Pressed => VAPPR_Selected;

 check APPR_Switch_Pressed_Selects_VAPPR;

-------------------------------------------------------------------------------
-- VAPPR mode shall become active if the VAPPR capture condition is met
-- while VAPPR mode is armed and LAPPR mode is active and
-- an overspeed condition does not exist.
```

```
--------------------------------------------------------------------------------
--   VAPPR_Active_When_Capture_Cond_Met = true ->
--     pre VAPPR_Selected and not pre VAPPR_Active
--     and VAPPR_Capture_Cond_Met
--     and LAPPR_Active and
--     and not Overspeed
--     and not Selected_NAV_Source_Changed
--     and not Selected_NAV_Frequency_Changed
--     and not CHANGED(Pilot_Flying_Side)
--     and Modes_On
--     and No_Higher_Event_Than_VAPPR_Capture_Cond_Met => VAPPR_Active;

--   check VAPPR_Active_When_Capture_Cond_Met;

--------------------------------------------------------------------------------
-- VAPPR mode shall be cleared if the APPR switch is pressed
-- while VAPPR mode is selected.
--------------------------------------------------------------------------------
 APPR_Switch_Pressed_Clears_VAPPR =
   pre VAPPR_Selected and RISING(APPR_Switch)
   and No_Higher_Event_Than_APPR_Switch_Pressed => not VAPPR_Selected;

 check APPR_Switch_Pressed_Clears_VAPPR;

--------------------------------------------------------------------------------
-- VAPPR mode shall be cleared if LAPPR mode is cleared.
--------------------------------------------------------------------------------
 LAPPR_Cleared_Clears_VAPPR =
   pre VAPPR_Selected and not pre LAPPR_Selected => not VAPPR_Selected;

 check LAPPR_Cleared_Clears_VAPPR;

--------------------------------------------------------------------------------
-- VAPPR mode shall be cleared if the selected NAV source is changed.
--------------------------------------------------------------------------------
 Selected_NAV_Source_Changed_Clears_VAPPR =
   pre VAPPR_Selected and Selected_NAV_Source_Changed => not VAPPR_Selected;

 check Selected_NAV_Source_Changed_Clears_VAPPR;

--------------------------------------------------------------------------------
-- VAPPR mode shall be cleared if the selected NAV frequency is changed.
--------------------------------------------------------------------------------
 Selected_NAV_Frequency_Changed_Clears_VAPPR =
   pre VAPPR_Selected and Selected_NAV_Frequency_Changed => not VAPPR_Selected;

 check Selected_NAV_Frequency_Changed_Clears_VAPPR;

--------------------------------------------------------------------------------
-- VAPPR mode shall be cleared when there is a pilot flying transfer.
--------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_VAPPR =
   pre VAPPR_Selected and CHANGED(Pilot_Flying_Side)  => not VAPPR_Selected;

 check Pilot_Flying_Transfer_Clears_VAPPR;

--------------------------------------------------------------------------------
-- VAPPR mode shall be cleared when the mode annunciations are turned off.
--------------------------------------------------------------------------------
 Modes_Off_Clears_VAPPR =
   pre VAPPR_Selected and not Modes_On  => not VAPPR_Selected;

 check Modes_Off_Clears_VAPPR;
```

```
--=============================================================================================
-- VGA MODE
--=============================================================================================

---------------------------------------------------------------------------------------------
-- VGA mode shall be selected if and only if VGA mode is active.
---------------------------------------------------------------------------------------------
 VGA_Selected_Iff_VGA_Active =
   (VGA_Active = VGA_Selected);

 check VGA_Selected_Iff_VGA_Active;

---------------------------------------------------------------------------------------------
-- VGA mode shall be selected if the GA switch is pressed
-- while VGA mode is cleared and an overspeed condition does not exist.
---------------------------------------------------------------------------------------------
 GA_Switch_Pressed_Selects_VGA =
   not pre VGA_Selected and RISING(GA_Switch) and not Overspeed
   and No_Higher_Event_Than_GA_Switch_Pressed => VGA_Selected;

 check GA_Switch_Pressed_Selects_VGA;

---------------------------------------------------------------------------------------------
-- VGA mode shall be cleared when the AP is engaged.
---------------------------------------------------------------------------------------------
 AP_Engaged_Clears_VGA =
   pre VGA_Selected and RISING(Is_AP_Engaged)  => not VGA_Selected;

 check AP_Engaged_Clears_VGA;

---------------------------------------------------------------------------------------------
-- VGA mode shall be cleared when LGA mode is cleared.
---------------------------------------------------------------------------------------------
 LGA_Cleared_Clears_VGA =
   pre VGA_Selected and not pre LGA_Selected  => not VGA_Selected;

 check LGA_Cleared_Clears_VGA;

---------------------------------------------------------------------------------------------
-- VGA mode shall be cleared when the SYNC switch is pressed
-- while VGA mode is selected.
---------------------------------------------------------------------------------------------
 SYNC_Switch_Pressed_Clears_VGA =
   pre VGA_Selected and RISING(SYNC_Switch) => not VGA_Selected;

 check SYNC_Switch_Pressed_Clears_VGA;

---------------------------------------------------------------------------------------------
-- VGA mode shall be cleared when the VS Pitch Wheel is rotated VGA mode is selected.
---------------------------------------------------------------------------------------------
 VS_Pitch_Wheel_Rotated_Clears_VGA =
   pre VGA_Selected and RISING(VS_Pitch_Wheel_Rotated)
   and No_Higher_Event_Than_VS_Pitch_Wheel_Rotated  => not VGA_Selected;

 check VS_Pitch_Wheel_Rotated_Clears_VGA;

---------------------------------------------------------------------------------------------
-- VGA mode shall be cleared when there is a pilot flying transfer.
---------------------------------------------------------------------------------------------
 Pilot_Flying_Transfer_Clears_VGA =
   pre VGA_Selected and CHANGED(Pilot_Flying_Side)  => not VGA_Selected;
```

```
 check Pilot_Flying_Transfer_Clears_VGA;

--------------------------------------------------------------------------------------
-- VGA mode shall be cleared when the mode annunciations are turned off.
--------------------------------------------------------------------------------------
 Modes_Off_Clears_VGA =
   pre VGA_Selected and not Modes_On  => not VGA_Selected;

 check Modes_Off_Clears_VGA;

--======================================================================================
-- Auxilliary signals used to simplify statement of properties.
--======================================================================================
 No_Higher_Event_Than_SYNC_Switch_Pressed = true;

 No_Higher_Event_Than_GA_Switch_Pressed =
  (not RISING(SYNC_Switch) and No_Higher_Event_Than_SYNC_Switch_Pressed);

 No_Higher_Event_Than_APPR_Switch_Pressed =
  (not RISING(GA_Switch) and No_Higher_Event_Than_GA_Switch_Pressed);

 No_Higher_Event_Than_HDG_Switch_Pressed =
  (not RISING(APPR_Switch) and No_Higher_Event_Than_APPR_Switch_Pressed);

 No_Higher_Event_Than_NAV_Switch_Pressed =
  (not RISING(HDG_Switch) and No_Higher_Event_Than_HDG_Switch_Pressed);

 No_Higher_Event_Than_AP_Engaged = true;

 No_Higher_Event_Than_ALTSEL_Target_Changed =
  (not RISING(APPR_Switch) and No_Higher_Event_Than_APPR_Switch_Pressed);

 No_Higher_Event_Than_ALT_Switch_Pressed =
  (not RISING(ALTSEL_Target_Changed) and  No_Higher_Event_Than_ALTSEL_Target_Changed);

 No_Higher_Event_Than_FLC_Switch_Pressed =
  (not RISING(ALT_Switch) and No_Higher_Event_Than_ALT_Switch_Pressed);

 No_Higher_Event_Than_VS_Switch_Pressed =
   (not RISING(FLC_Switch) and No_Higher_Event_Than_FLC_Switch_Pressed);

 No_Higher_Event_Than_VS_Pitch_Wheel_Rotated =
  (not RISING(VS_Switch) and No_Higher_Event_Than_VS_Switch_Pressed);

 No_Higher_Event_Than_FD_Switch_Pressed  =
  (    (not RISING(NAV_Switch) and No_Higher_Event_Than_NAV_Switch_Pressed)
   and (not RISING(Is_AP_Engaged) and No_Higher_Event_Than_AP_Engaged)
   and (not RISING(VS_Pitch_Wheel_Rotated) and
        No_Higher_Event_Than_VS_Pitch_Wheel_Rotated));

 No_Higher_Event_Than_LAPPR_Capture_Cond_Met =
  (not RISING(FD_Switch) and No_Higher_Event_Than_FD_Switch_Pressed);

 No_Higher_Event_Than_NAV_Capture_Cond_Met =
  (not LAPPR_Capture_Cond_Met and No_Higher_Event_Than_LAPPR_Capture_Cond_Met);

 No_Higher_Event_Than_VAPPR_Capture_Cond_Met =
  (not RISING(FD_Switch) and No_Higher_Event_Than_FD_Switch_Pressed);

 No_Higher_Event_Than_ALTSEL_Track_Cond_Met =
  (not VAPPR_Capture_Cond_Met and No_Higher_Event_Than_VAPPR_Capture_Cond_Met);

 No_Higher_Event_Than_ALTSEL_Capture_Cond_Met =
```

```
  (not ALTSEL_Track_Cond_Met and No_Higher_Event_Than_ALTSEL_Track_Cond_Met);

tel;

--------------------------------------------------------------------------------
-- RISING - returns true when signal s changes from false to true
--------------------------------------------------------------------------------
node RISING (s : bool) returns (p : bool);
let
   p = false -> (not pre s and s);
tel;

--------------------------------------------------------------------------------
-- FALLING - returns true when signal s changes from true to false
--------------------------------------------------------------------------------
node FALLING (s : bool) returns (p : bool);
let
   p = false -> (pre s and not s);
tel;

--------------------------------------------------------------------------------
-- CHANGED - returns true when signal s changes value
--------------------------------------------------------------------------------
node CHANGED (s : bool) returns (p : bool);
let
   p = false -> (not (s = pre s));
tel;
```

Appendix C Mode Logic Error Log

Date	Location	Classification	Description	How Resolved	Notes
10/4/2012	Event Processing	Minor	Incorrect inhibiting of input events. The inhibit_In of the When_Switch_Pressed_Seen block was wired to the output of the next higher priority block.	Modified the When_Switch_Pressed_Seen block to output an Inhbit_Out that was the OR of the Inhibit_In and the output indicating if the event was seen.	Checking that at least one lateral mode is active.
10/4/2012	Event Processing	Minor	LGA_Active being set to false when VGA exits the active state.	Corrected copy and paste error.	Checking that at least one lateral mode is active.
10/16/2012	Flight Modes	Moderate	Incorrectly predicting if a lateral or vertical mode would be activated on a step. Failed to account for the case where LNAV, LAPPR, ALTSEL, or VAPPR could be cleared at the same time the capture condition is met.	Updated Will_MODEX_Be_Activated to take into account the simultaneous occurrence of being cleared at the same time as the capture condition is met.	Checking that at least one lateral mode is active and that at least one vertical mode is active.
10/16/2012	Event Processing	Minor	Incorrect inhibiting of input conditions. The inhibit_In of the If_Condition_Seen block was wired to the output of the next higher priority block.	Modified the If_Condition_Seen block to output an Inhbit_Out that was the OR of the Inhibit_In and the output indicating if the condition was seen.	Checking that at most one vertical mode is active.
10/16/2012	Flight Modes	Major	Two vertical modes active at the same time caused by an Overspeed condition selecting FLC mode at the same time the ALT switch is pressed.	Modified the condition for selecting FLC mode so that FLC mode is not selected if ALT or ALTSEL are already active or will become active on this step.	Checking that at most one vertical mode is active.
10/16/2012	Flight Modes	Moderate	Two vertical modes active at the same time caused by pressing the FLC swtich while ALTSEL Capture is active.	Fixed the definition of Will_FLC_Be_Active by removing the condition that Clear_FLC be false.	Checking that at most one vertical mode is active.
11/20/2012	Mode Logic	Minor	NAV_Switch input incorrectly named NAV_Swiitch.	Changed the name of input to NAV_Switch.	Checking definition of No_Higher_Event_Than_FD_Switch_Pressed.
11/28/2012	Flight Modes	Moderate	Execution sequence of Independent and Active mode machines was incorrect - Independent was assigned an execution order of 5 and Active was assigned an exectuion order of 6. This placed their execution after Lateral and Vertical had executed.	Changed the execution order of Independent to 3 and the execution order of Active to 4.	Checking that onside and offside VAPPR active implies the FGS is in Independent mode.
11/29/2012	Flight Modes	Moderate	A pilot flying transfer did not make PITCH mode active while ALTSEL was active.	Added a deactivate transition from ALTSEL.Active to ALTSEL.Armed mode.	Checking that a Pilot Flying Transfer should make PITCH mode active.
11/29/2012	Flight Modes	Major	ROLL mode was not active while the mode annunciations were off.	Simplified definition of Lateral_Mode_Manually_Selected and Veritical_Mode_Manually_Selected to not test if mode annunciations were off in the previous step or if this side is active.	Checking that ROLL mode is active while the mode annunications are off.
11/29/2012	Flight Modes	Major	PITCH mode was not active while the mode annunciations were off.	Changed definition of Vertical_Mode_Manually_Selected to not if this side is active when the ALTSEL target altitude is changed while ALTSEL mode is tracking.	Checking that PITCH mode is active while the mode annunciations are off.
12/7/2012	Mode Logic	Minor	Selected_NAV_Source_Changed and Selected_NAV_Frequency_Changed incorrectly named Selected_Nav_Source_Changed and Selected_Nav_Frequency_Changed.	Changed spelling to Selected_NAV_Source_Changed and Selected_NAV_Frequency_Changed.	Checking that NAV mode becomes active when the NAV capture condition is met.
12/13/2012	Flight Modes	Moderate	Truth table for ALT Select was overspecified. Stated that ALT mode should be selected if the ALTSEL target was changed while in ALTSEL Track mode and not in VAPPR Active mode. The dependence on VAPPR Active mode was not needed since ALTSEL Track is an active mode and only one vertical mode can be active at a time.	Changed the dependence on VAPPR Active to a don't care in ALT Select truth table.	Checking that ALT mode is selected when the ALTSEL target is changed while in ALTSEL Track mode.
12/17/2012	Flight Modes	Major	Transitions between ALTSEL Clear and ALTSEL Selected would occur on the current step if ALT mode became active/inactive and on the next step if VAPPR or VGA became active/inactive. This occurred since the order of executing these state machines was ALT, ATLSEL, VAPRR, followed by VGA.	Changed the order of execution of ALTSEL to follow execution of ALT, VAPPR, and VGA modes.	Found checking that ALTSEL mode is selected iff none of ALT, VAPPR, or VGA modes are active. While the error was not serious (the necessary transition would occur in the current or next step), it would be difficult to detect through testing.
12/17/2012	Flight Modes	Moderate	Turning the mode annunciations off only deactivated ALTSEL mode (i.e. took it from ACTIVE to ARMED mode). Turning the mode annunciation off should clear ALTSEL mode (i.e. take it from SELECTED to CLEARED mode).	Moved column enabling a transition when the modes are turned off from the ALTSEL Deactivate truth table to the ATLSEL Clear truth table.	Found checking that ALTSEL mode is selected iff none of ALT, VAPPR, or VGA modes are active.
12/18/2012	Flight Modes	Major	Possible to have two modes active at the same time if an overspeed condition occurs (activating FLC) at the same time as the ALTSEL capture condition is met (activating ALTSEL Capture).	Strengthened the ALTSEL Capture() truth table to not capture ALTSEL mode while an overspeed condition exists.	Checking at most one vertical mode active while adding Active Side logic.

Classification		
Trivial	0	Spelling or Punctuation
Minor	5	Likely to be detected by traditional verification.
Moderate	6	Potential to not be detected by traditional verification.
Major	5	Unlikely to be detected by traditional verification.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 04 - 2014	Contractor Report	

4. TITLE AND SUBTITLE

Formal Methods Case Studies for DO-333

5a. CONTRACT NUMBER

NNL06AA04B

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Cofer, Darren; Miller, Steven P.

5d. PROJECT NUMBER

5e. TASK NUMBER

NNL12AB85T

5f. WORK UNIT NUMBER

534723.02.02.07.40

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, Virginia 23681

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/CR-2014-218244

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 64
Availability: NASA CASI (443) 757-5802

13. SUPPLEMENTARY NOTES

Work performed by Rockwell Collins, Inc., for The Boeing Company and NASA Langley Research Center, under Contract NNL06AA04B, NNL12AB85T.
Langley Technical Monitor: Benedetto L. Di Vito

14. 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. The supplement identifies the modifications and additions to DO-178C and DO-278A objectives, activities, and software life cycle data that should be addressed when formal methods are used as part of the software development process. This report presents three case studies describing the use of different classes of formal methods to satisfy certification objectives for a common avionics example – a dual-channel Flight Guidance System. The three case studies illustrate the use of theorem proving, model checking, and abstract interpretation. The material presented is not intended to represent a complete certification effort. Rather, the purpose is to illustrate how formal methods can be used in a realistic avionics software development project, with a focus on the evidence produced that could be used to satisfy the verification objectives found in Section 6 of DO-178C.

15. SUBJECT TERMS

Avionics software; Certification; Formal methods; Verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	203	(443) 757-5802