Designing Control Logic for Cockpit Display Systems using Model-Based Design

Siddharth Sharma and Nishaat Vasi
The MathWorks Inc., 3 Apple Hill Drive, Natick, MA, 01701, USA

Jason Ghidella
The MathWorks Inc., 3 Apple Hill Drive, Natick, MA, 01701, USA

Cockpit display systems require logic for managing the modes of multiple components aboard the aircraft. Multiple competing criteria for display need to be applied and prioritized to ensure the most important and relevant information is displayed at all times. State machines provide a natural mechanism for representing system modes while flow charts make it easy to represent complex flow logic. This paper outlines the use of Model-Based Design to efficiently design the control logic for a cockpit display system. By conducting functional and structural verification on the design model, errors in the design can be quickly discovered and corrected before the design is implemented in software.

I. Introduction

The logic for generating pages for an aircraft's cockpit display system can be complicated and difficult to develop, test, and verify due to the large number of signals that must be processed and the various competing criteria for prioritizing what to show at any given time. Cockpit display software has three primary responsibilities: acquiring and conditioning data from the aircraft, warning management, and page control. Signals from various parts of the aircraft are first synchronized and filtered by the display software. For redundant signals, the signal values are compared before a signal value is selected. Then the selected signal values are passed to the warning management and page control logic for processing and display control.

Cockpit display design involves developing sophisticated logic to manage multiple pages and warnings for display on various display units. The display units are used to present critical information to the pilot, including the health of various aircraft systems, flight characteristics, and navigational information. The aim of the display system is to selectively present only the most relevant information in a clear and concise manner.

The display system software switches among different modes based on the operational status of the display system hardware, status of the aircraft equipment, phase of flight, and inputs from the user. The modes of the display system are used to generate different warnings and pages for display. When the display system hardware is initially switched on, the display units show the appropriate boot-up information. Once the startup is successful, a default page is shown. As an example, the System Display screen defaults to the auxiliary power unit (APU) page when it has just been started. The display units then switch pages based on the status of various systems in the aircraft. For example, when the aircraft engines are starting, the system display unit switches from the default page to the *Engines* page.

In each phase of flight a subset of signals is monitored to evaluate the status and performance of various systems. When a fault is detected for a monitored signal, the display unit changes modes, an appropriate page is populated on the screens, and a warning is generated for the Electronic Centralized Aircraft Monitor (ECAM) screen (see Fig. 1). Some faults may only cause warnings to be generated while others may only cause page switches on the displays. In the presence of multiple faults, a priority order is used to select the page to display and the order of the warnings. The pilot can also view a certain page by selecting from a set of buttons. The selection by the pilot causes a mode switch in the display unit software.

With Model-Based Design[1,2,3], system and verification engineers can efficiently design and test complex control logic early in the development process, where it is easier and less expensive to update if needed[4]. The design is specified using state machines and flow charts. Test harness models can also be constructed to test individual components against requirements. A system model that includes plant and environment models can be used to simulate and validate overall performance. The model is used throughout the design, first through desktop simulation, later for real-time simulation, and finally for implementation of the embedded software. This typically

means that the model is created and used by a number of different roles. For example, the design engineer is responsible for modeling the high-level requirements of the control logic, and the verification engineer is responsible for leading the verification and validation effort at a model and code level.

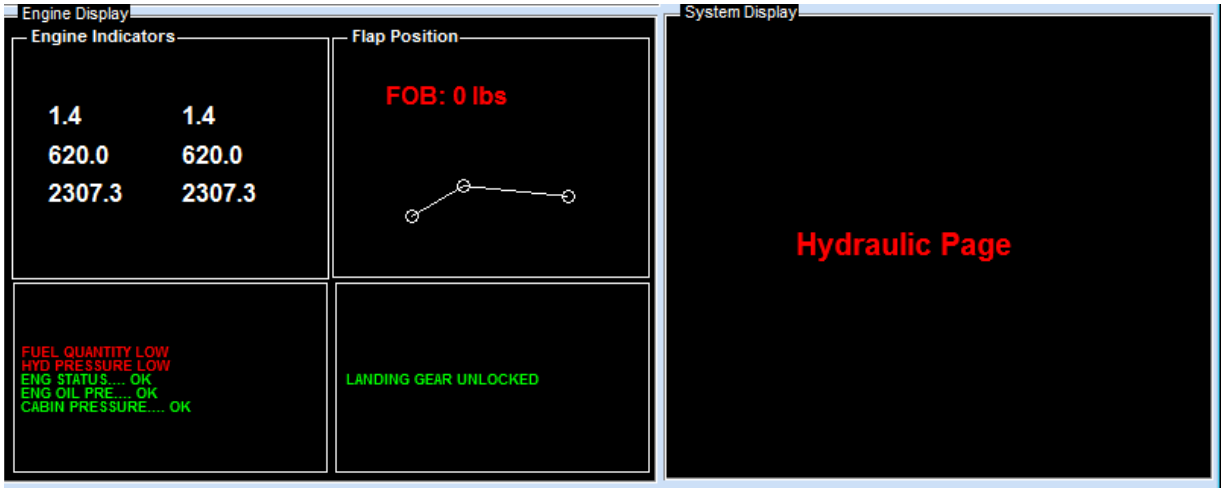

Figure 1. *ECAM screens: Engine Display (left) and System Display (right)*

In section II, the mode logic for this application is modeled using state machines and flow charts in the Stateflow®[5] design environment. State machines can clearly represent the modes of a system while flow charts provide a simple representation of the transitions between modes. During development of the logic, simulation is used extensively to help identify design and modeling errors early. The various components of the design are linked to the functional requirements.

In section III, the need to start design verification early is discussed using the landing gear warning component as an example. Here the interaction between the design and verification engineer is explored when using Model-Based Design. Section III also discusses the need for functional verification of the design against requirements, commonly known as requirements-based testing. Test cases were authored and linked to the same functional requirements document as the design. Simulation was used to run the test cases and verify the design.

In addition to functional verification, section III also covers the need for structural verification of the landing gear warning component. Model coverage metrics were collected during the simulation of the requirement-based test cases. A lack of complete model coverage can be due to undocumented requirements, missing test cases, or design errors. In this example, the analysis helped detect a design error that we could fix before the software was implemented.

II. Modeling the Cockpit Display System

The software for a cockpit display system (CDS) acquires and synchronizes data from various sensors and computers aboard the aircraft, generates flight warnings, and displays appropriate data and warnings to the pilot. Fig. 2 shows the relationship of the control logic with the rest of the application. The *Control Logic* block receives data and user input from the *Inputs* block. The *Inputs* block represents a model of the aircraft. It models various sensors and computers that send data to the CDS. The C*ontrol Logic* block models the CDS software. The *Display Code* block is responsible for integrating the numeric IDs from the *Control Logic* block with other text based sources and generating graphics on the display units (DUs). The component that integrates the IDs with other components is called glue code. Glue code is used to keep the design of the software for the warning system separate from the warning requirements.

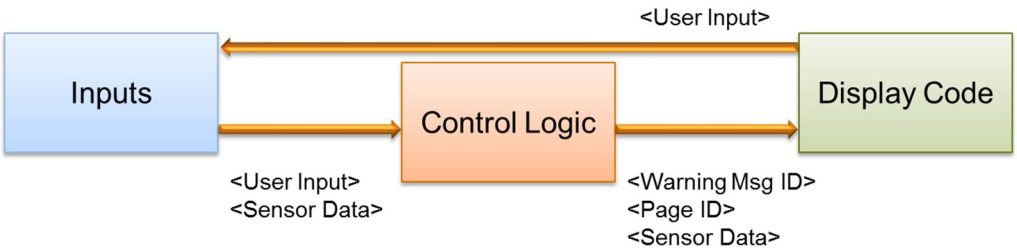

Figure 2. *Architecture of a Cockpit Display System*

The *Control Logic* block contains three components: signal conditioning, warning management, and page control. Data from the plant model is sent to the signal conditioning component (see Fig. 3a). The results are then used to compute warnings for the system. These warnings along with additional data from the signal conditioning component are sent to the page control component for selecting the pages and appropriate information to display on each DU. Information from the page control component is then sent to the *Display Code* component (see Fig. 3b). The next sections cover the structure, function, and development challenges of each control logic component.

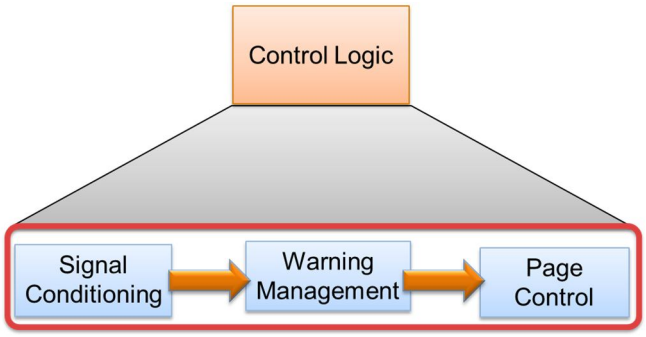

Figure 3a. *The components of Control Logic*

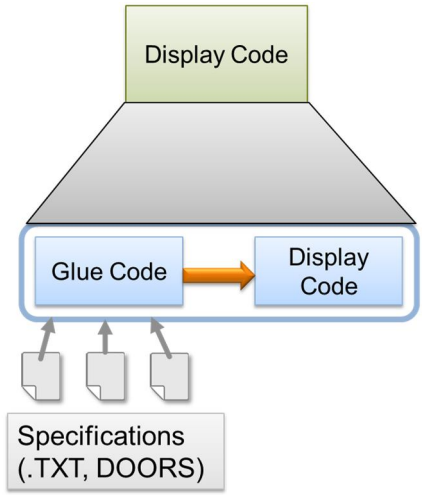

Figure 3b. *The components of Display Code*

A. Signal Conditioning

The C*ontrol Logic* component receives data from multiple sensors in the aircraft that are executing at different rates. The *Signal Conditioning* component (see Fig. 4) synchronizes the data by fetching data at the appropriate rates for the signals. To make the software components reusable, their units are normalized. The normalized data received from the sensors is filtered for noise. The filtered data is validated by checking that the values lie within certain limits. Since the above operations are computationally intensive, this component is best modeled using mathematical tools. Simulink was used in our model. Many critical systems on the aircraft have redundant sensors and a voting mechanism is used to select the final value of the data.

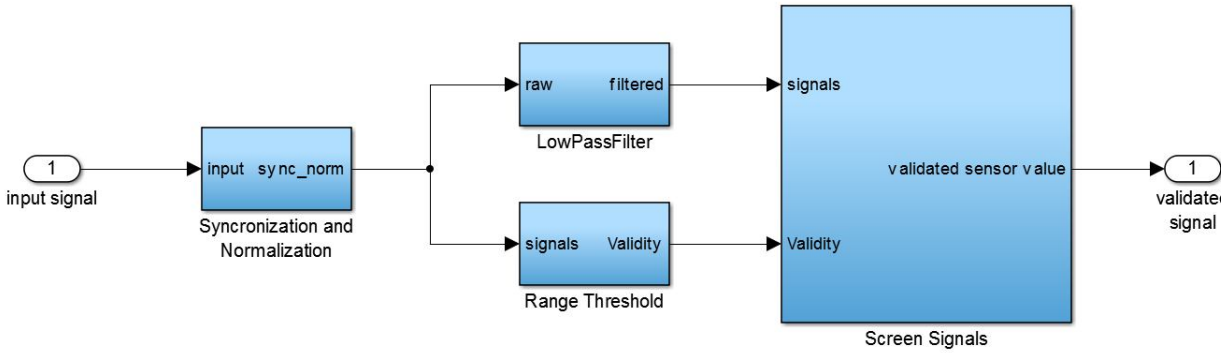

Figure 4. *Signal Conditioning model*

B. Warning Management

The warning management system is responsible for generating messages that are displayed on the various DUs. The messages indicate the health and status of various systems aboard the aircraft through memos and warnings. When in normal mode, the system will generate a memo to inform the pilot of the status of the component, but if a component is not behaving as expected, a warning is generated.

The behavior of the components can be modeled clearly as distinct modes of operation. As an example, the fuel pressure for the aircraft can be *normal*, *low*, and so on. Note that the *normal* fuel value ranges are different for each phase of flight since fuel is consumed over the course of the flight. Hence, warning management requires monitoring the phases of flight and the modes of the components. We use a state machine to represent the component modes and the phases of flight. The phases of flight and the modes of the components are represented using states that execute at each time step in the simulation, referred to as parallel states (see Fig. 5). Parallel states are drawn as rectangles with dashed borders. The state *FlightPhases* captures the phases of flight whereas *ComponentModes* encapsulate the modes for the components. Let us look at the content of these states in more detail.

Figure 5. *Interdependent components of warning management*

A complete flight is broken into phases based on critical system or flight characteristics (see Fig. 6). For example, the flight transitions from the first to the second phase of flight when the airspeed is above a predetermined threshold. Conditional logic based on the value of signals is used to transition between the phases of flight. The sequence of flight stages are represented with states, and transition lines are used to represent valid paths between those states. The transition from one state to another is taken when the condition on it is true. A numerical ID, *FlightMode*, is used to track the state of the system outside *FlightPhases*.

The second aspect of the design that affects message generation is the aircraft components' status. Below, we look at the design of the landing gear system component. A state machine is used to clearly represent the modes of the component. Flow charts and functions are used to define the conditions for transition between the modes. Later in the paper, we discuss how to link such a component to high-level requirements and test strategies for this component. Table 1 contains a subset of the high level requirements for the landing gear warning management system.

The landing gear warning management system receives Boolean command inputs, *LGLeftLock* and *LGRightLock*, which drive the state of the landing gear (see Fig. 7). The output *LGMode* is used to track the mode of operation outside of the state machine and display the appropriate warning message to the pilot. The aircraft has two landing gear units, one on the right and one on the left. The landing gear units can be locked when they are fully retracted (up) or fully extended (down). Both the landing gear units should work in tandem. In other words, they should both be locked at the end positions and unlocked for extending/retracting together. When both the units are locked, *LGMode* is 1; when only one is locked, *LGMode* is 2; and when both are unlocked, *LGMode* is 3. The output *LGMode* controls which warning message should be displayed to the pilot. If only one unit is locked for an extended period of time (10 seconds in the example), a fault is generated and a warning is displayed to the pilot.

It is important to note that all components do not generate warnings in each phase of flight. So, the warning generation system, *WarningsComputation*, needs to monitor both the phase of flight and the component modes, modeled in *FlightPhases* and *ComponentModes* respectively, as it generates the warning messages. To model such behavior we first use states to model the phases of flight (see Fig. 8a). The value of variable *FlightMode* is used to guard the transition between these states. Within each state, flow charts are used to check the modes of specific components.

Figure 6. *Phases of flight*

3.1	**Both landing gear units locked** If left landing gear unit and right landing gear unit are locked, set the warning display mode to 1
3.2	**Only one landing gear unit locked** When switching from locked to unlocked states and unlocked to locked states for each gear unit
3.2.1	If left landing gear unit is unlocked and the right landing gear unit is locked, set the warning display mode to 2
3.2.2	If right landing gear unit is unlocked and the left landing gear unit is locked, set the warning display mode to 2
3.2.3	If value of display mode is 2 for more than 10 seconds, indicate error by setting display mode to 4
3.2.4	If display mode is 4 and both the landing gear units are locked, set the warning display mode to 1. Else if, display mode is 4 and both the landing gear units are unlocked, set the warning display mode to 3.
3.3	**Both landing gear units unlocked** If left landing gear unit and right landing gear unit are unlocked, set the warning display mode to 3

Table 1. *Partial list of requirements for mode switching logic of landing gear warning system*

Using this approach, the designer can choose the components and component modes for which messages need to be displayed on the DU. Instead of flow charts, a text-based specification could also have been used for generating the message ID array with *switch-case* or *if-else* constructs.

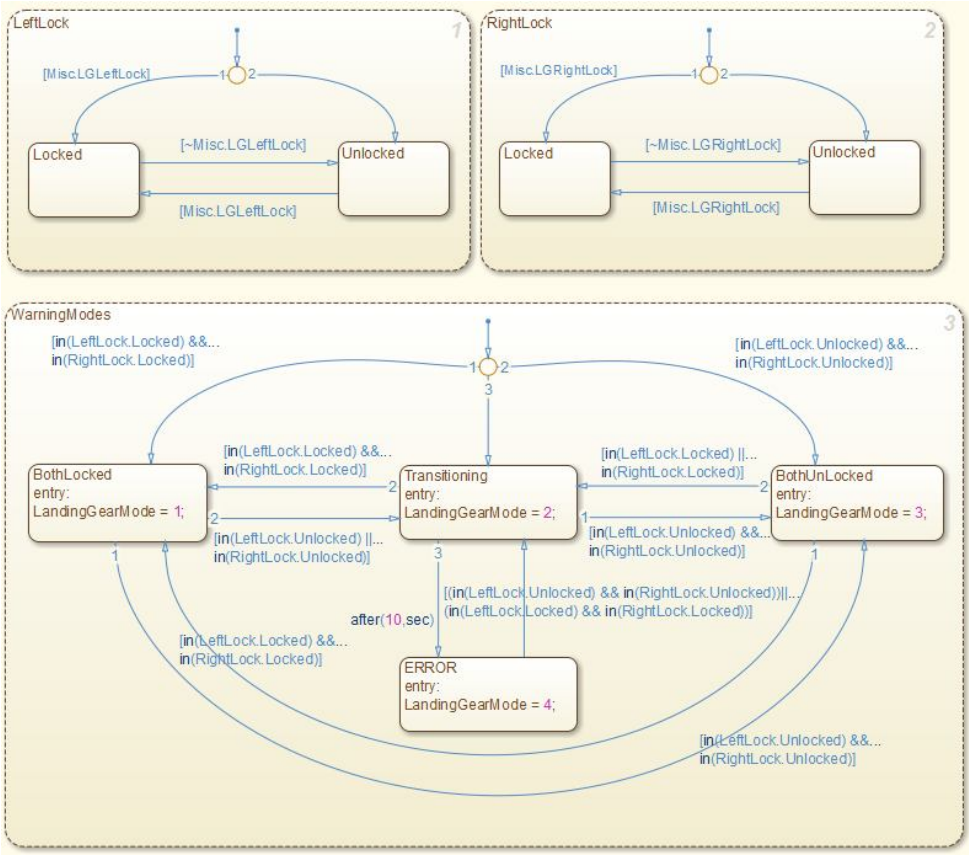

Figure 7. *Mode switching logic for landing gear warnings*

We have not discussed how the order of the warnings messages and their display colors are decided. That is because these details are handled by the glue code. Some messages might not be displayed because of the fixed number of error message lines on the display. The array of message IDs is sent from the warning message code to the glue code which in turn combines the message IDs with message priority, color code, and message text from external text specifications at run time. The messages are then sorted by priority and formatted before being displayed on the DUs. The message details are read in at run time as opposed to being hardcoded in the software to enable changes without regeneration of software. One drawback of loading error message details at run time is that it can be difficult to determine if issues are related to software or due to an incorrect specification. If the messages are defined completely in software, the debugging capabilities of the simulation environment can be used to understand issues. As an example, if the messages are completely defined in software, Stateflow® software highlights the paths taken by logic defined in the flow chart. You can set a breakpoint and step though the logic to inspect message details and correlate that to the modes of various components in the system. If the messages are loaded at run time by the glue code, the messages need to be mapped manually to message IDs and then correlated to the component modes.

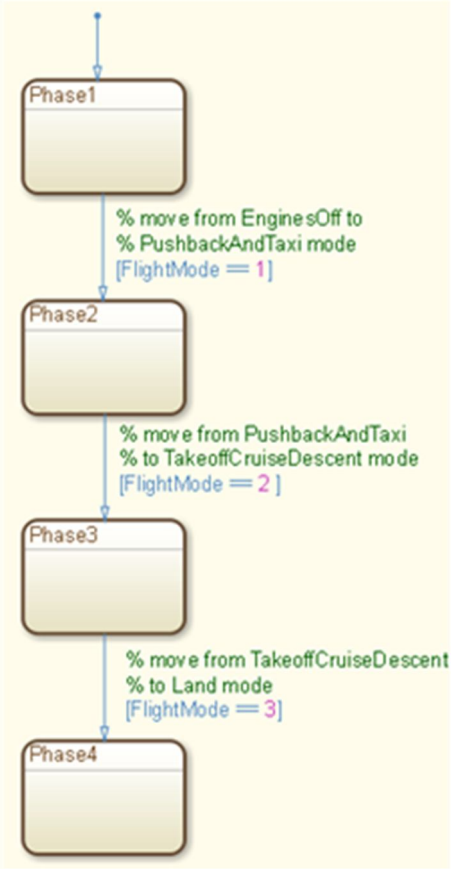

Figure 8a. *Representing flight phases for warning message generation*

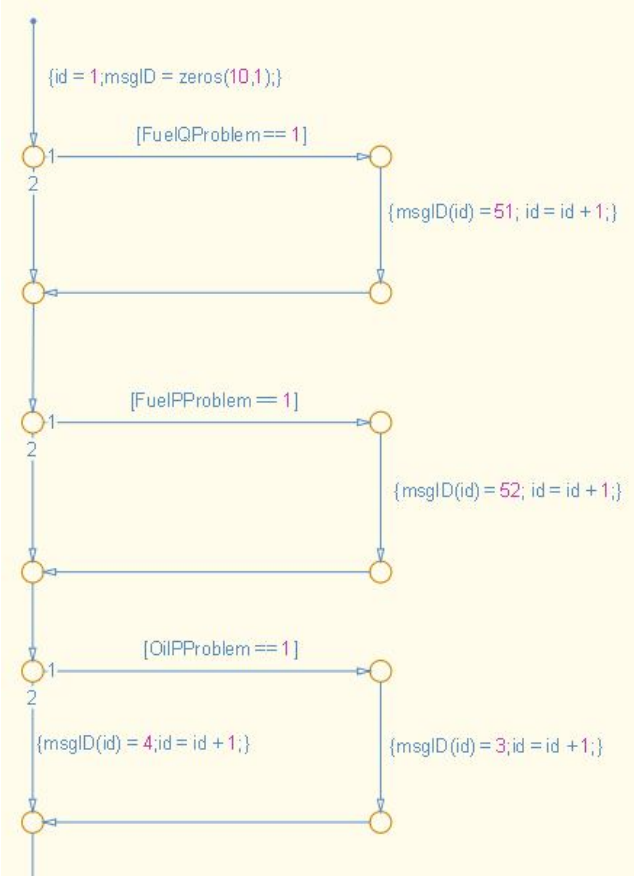

Figure 8b. *Appending Message IDs to an array*

C. Page Control

Software for display systems controls the switching of pages on the DU based on criteria including flight phase, system faults, and pilot input. Each page informs the pilot about a different aspect of the aircraft's equipment. The *APU* page, *Engines* page, and other pages can be modeled as modes of the display. These modes are represented using states in a state machine. The system stays in a particular state, or displays a specific page, until there is a change in the system or an event occurs. State machines are a clear representation of such event-based reactive systems.

CDS software drives multiple DUs, all of which are active simultaneously. Using state machines, systems that work independently but are active simultaneously are represented using parallel states. Each parallel state contains page logic for a different DU.

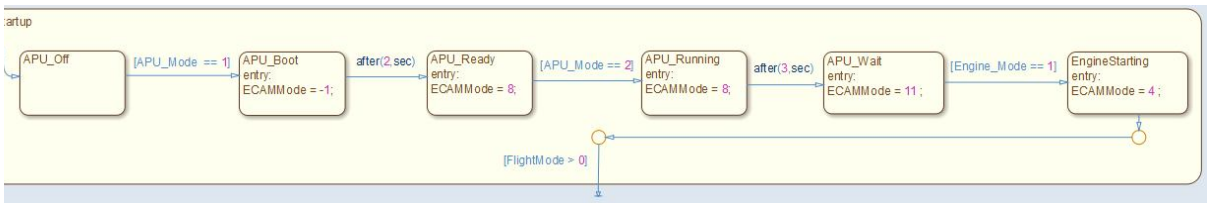

Figure 9. *Page sequence displayed on System Display screen during startup*

As an instructive example of page logic for the system display screen, consider the contents of the *SystemDisplay* state. As the pilot switches on the APUs aboard the aircraft, the display hardware boots up and the screen shows successive pages for startup including initialization, booting, the *APU* page, the *Engines* page, and so on based on the state of the systems and events generated by pilot input (see Fig. 9). After startup, the system displays pages for flight phases as the aircraft starts to move through phases of flight.

The phase of flight needs to be polled by the software to show the appropriate page on the screen. In state machines, polling is achieved by drawing a transition from the parent state (in this case, *NormalFlight*) to the junction inside the same state (see Fig. 10). Due to top-down semantics in Stateflow® software, the inner transition causes the software to exit the child states, recheck the conditions on the transitions at each time step, and then enter the appropriate child states representing the phases of flight.

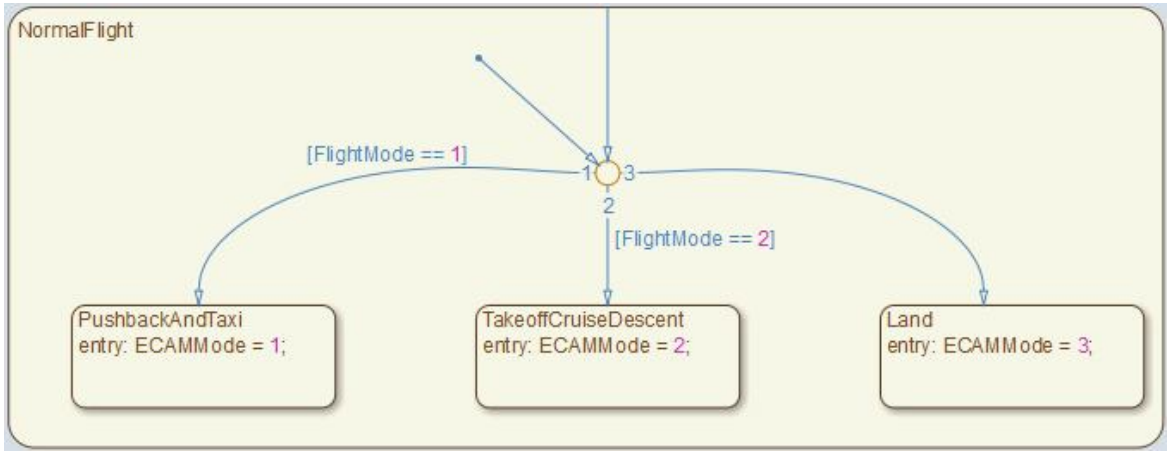

Figure 10. *Page logic for phases of flight*

System faults have higher priority than normal flight phase pages. User selection, in turn, has higher priority than system fault pages. Design engineers can use a combination of state machine semantics and flow charts to clearly model the priority order in software for page logic.

As shown in Fig. 11, a transition is drawn from the parent state *NormalFlight* to *UserSelection_Faults*. Due to the top-down (parent to children) semantics in Stateflow® software, this transition will be checked and executed irrespective of the child state of *NormalFlight* that the system is in. On this transition, the software checks if a fault exists or if a user selection has been made. This provides higher priority to the state *UserSelection_Faults* than the children of states *NormalFlight*. The system shows

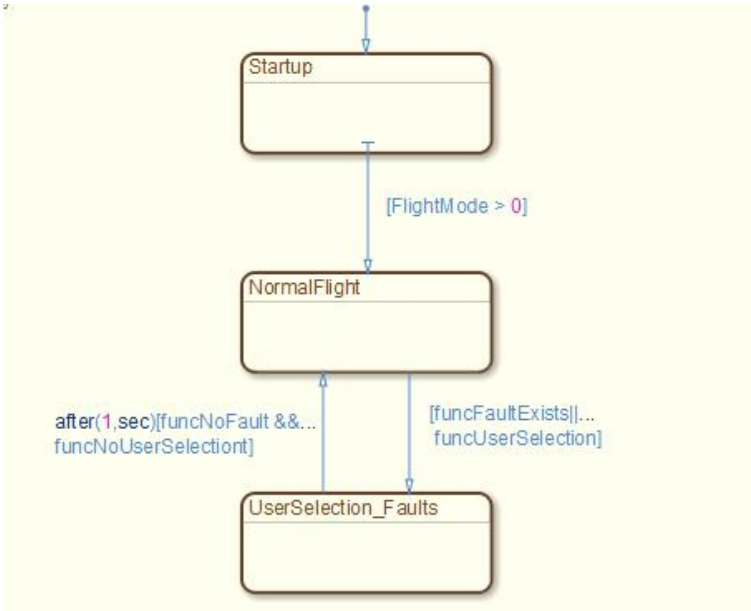

Figure 11. *Page sequences for the System Display screen*

default flight phase pages in the absence of faults and user selections modeled in *UserSelection_Faults*.

Within the *UserSelection_Faults* mode, user selection is given priority over system faults. As shown in Fig. 12, two transitions emanate from the junction at the top of the state. The transition labeled 1 has higher priority than the transition labeled 2 and will be checked first. The condition on the first transition checks if a user selection has been made. If yes, the logic traverses the graphical function to enter the appropriate state and set the value of the variable *ECAMMode*, which is used in the glue code to generate the appropriate page. In the absence of a user selection, the logic traverses down the right side and checks which component is in a fault mode. If there are multiple faults, the page located highest in the tree will be selected for display because the logic is moving down the tree in this case. So, for example, the *Engine*s page will be shown if the engine has a fault and the cabin pressure begins to drop at the same time.

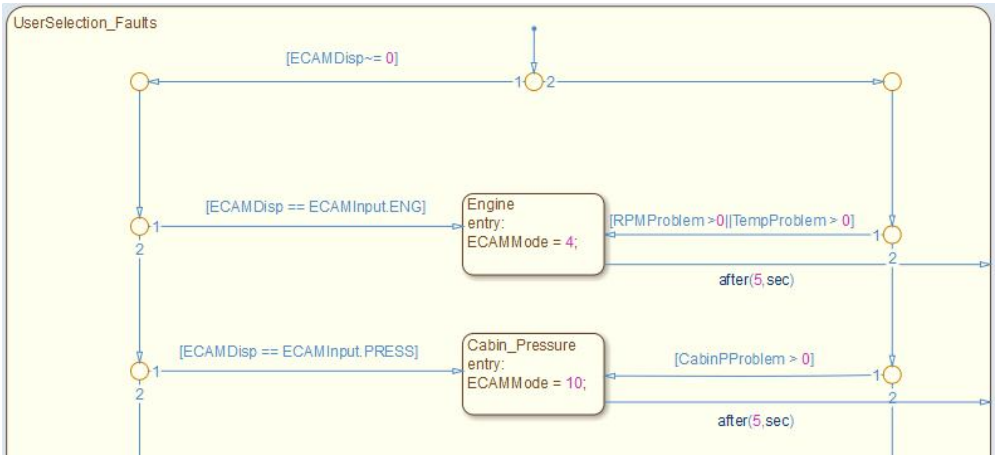

Figure 12. *Page priority for User Selections and Faults*

To show the most current page to the pilot, the page logic must be traversed at a defined time interval. This requires regular polling of component errors and user selections. Polling is done using the inner transitions from the child states, such as *Engine* and *Cabin_Pressure*, to the parent state *UserSelection_Faults*. The inner transition causes the logic to exit the active child state and traverse the hierarchy of decisions from the top to make sure the most important page is shown to the user every five seconds. The time delay was built into the condition for the inner transition to avoid rapid switching between pages. In the absence of faults and user selections, the pilot should not see the error pages any more (that will make the system appear unresponsive). So, as shown in Fig. 11, the transition from the parent state *UserSelection_Faults* to *NormalFlight* is checked at every instant after the system has been in the state *UserSelection_Faults* for a second.

III. Early Design Verification of the Landing Gear Warning Component

Creating an executable system specification in the form of a model enables continuous verification and validation during the design cycle. Using an incremental testing approach, wherein functional and structural completeness of each low level component within the controller is independently verified in an open loop configuration, the design and verification engineer validate whether the model meets its requirements before generating code and implementing the design on hardware. The primary motivating factor for early verification is identifying errors early in the design and test phases to save time and effort[6]. The cockpit display software is typically built one component at a time. This enables the design engineer to develop and functionally verify the behavior of each component in a modular fashion. Also, such an approach lends itself well to component-level verification and open-loop testing of each control module, typically performed by the verification engineer. In this

section, we outline techniques used to verify the landing gear warning component (refer to section II). With Model-Based Design the verification and design engineer can collaborate to verify software components. The verification methodology can be applied successively for each component in the cockpit display controller, before moving on to integration level testing.

A. Requirements Traceability

To determine if he has met high-level design requirements, the design engineer links model elements to specific requirements (or requirement objects) stored in a database or requirements lifecycle management tool. Creating these links enables traceability from a particular requirement to the design implementation and vice versa. Maintaining requirements traceability is particularly useful for design and verification reviews, and is mandated for DO-178 certification. If a particular requirement changes, the design engineer can quickly identify the parts of the model that need attention. In this work, traceability links are created to a set of high-level requirements stored and maintained in Microsoft Word. These requirements represent a small subset of the flight control application requirements for the *Landing Gear Warning Mode* control unit. A summary of the high-level requirements is shown in Table 1. The requirements are formulated in a natural language and defined in Word. One way of creating requirement associations is using the Simulink® Verification and Validation ™ software requirements traceability interface.

B. Functional Verification

Once the design engineer has a preliminary version of the landing gear warning mode component ready, and is confident that he has implemented the specified high-level requirements, he uses simulation to verify the functional behavior of the implementation. The verification engineer is tasked with creating test scenarios to verify design correctness, and uses the same high-level requirements (Table 1) to create test vectors for the Design-Under-Test (DUT). Typically, verification engineers combine empirical test input data with specific verification criteria for the landing gear design to construct tests. Instead of testing only the source or object code—which requires waiting for the design team to convert a floating-point model to a fixed-point design and generate code for a relevant embedded control unit—the design engineer can use test cases to simulate the functional correctness of the model.

Applying this early verification technique to test the requirements for a component of the system, specifically for the *Landing Gear Warning Modes* unit, a test harness is established (see Fig. 13) by extracting the DUT and setting up input and output ports to faithfully represent the interface of the DUT.

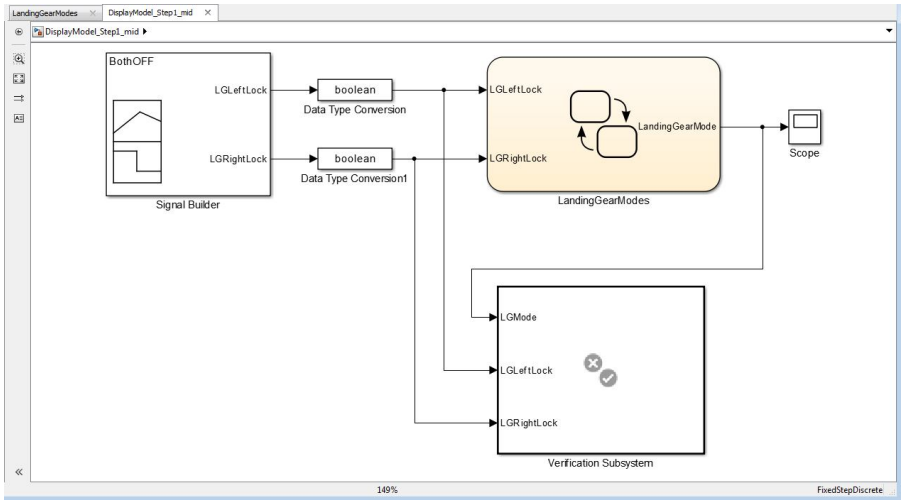

Figure 13. *Test harness setup in Simulink*

The inputs are driven by a Signal Builder block. The output is compared against the expected result (derived from requirements) by the verification subsystem. The *Verification Subsystem* contains logic and assertion blocks that evaluate whether the output modes of the DUT meet the output modes specified by the requirements.

Test cases are imported into Signal Builder and associated with specific verification blocks as well as the requirement that the test case is verifying. Associations of requirements and verification blocks with the test cases help validate the design for functional behavior. For example, in Fig. 14 the signals of test group *BothOFF* are associated with an Assert block within the *Verification Subsystem* and linked back to requirements.

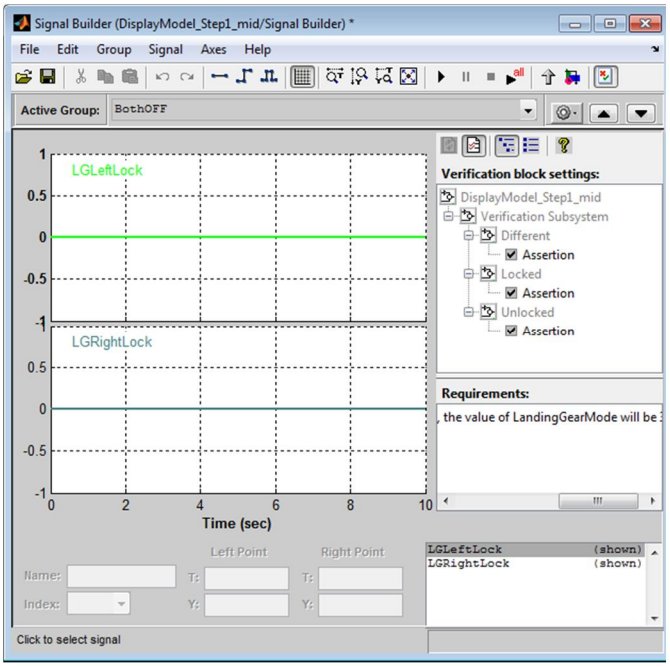

Figure 14. *Test cases in Signal Builder with associated verification mechanisms and requirements information*

When the test is executed, the simulation will stop if the condition specified within *Verification Subsystem* is invalidated (and thus the requirement is violated). If an assertion is detected, the engineer can trace the particular test case and design violation back to the root requirement for further review. Consider a case in which the design engineer encounters an assertion for the test *BothOFF* while simulating the tests (see Fig. 15).

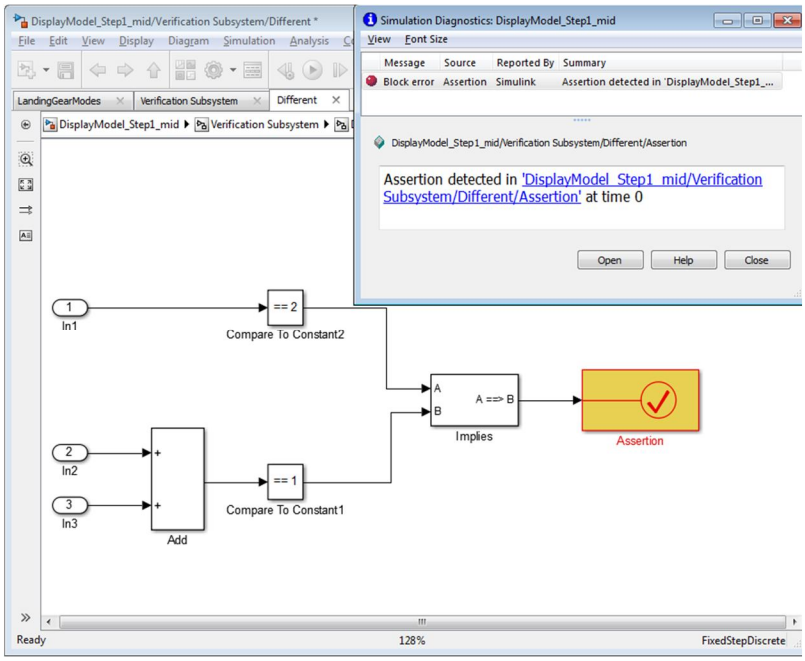

Figure 15. *A detected assertion*

The engineer finds that the landing gear assembly is incorrectly assigning a warning state (*LGMode = 2*) of only one gear being unlocked when both landing gears should be unlocked (*LGMode = 3*). He identifies an error in the state logic wherein the state *BothUnlocked* (that sets *LGMode = 3*) was always superseded by state *Transitioning* (that sets *LGMode = 2*). By changing the transition order for the above states and modifying the conditions for entry, the design engineer fixes the model and the *BothOFF* test passes (see Fig. 16).

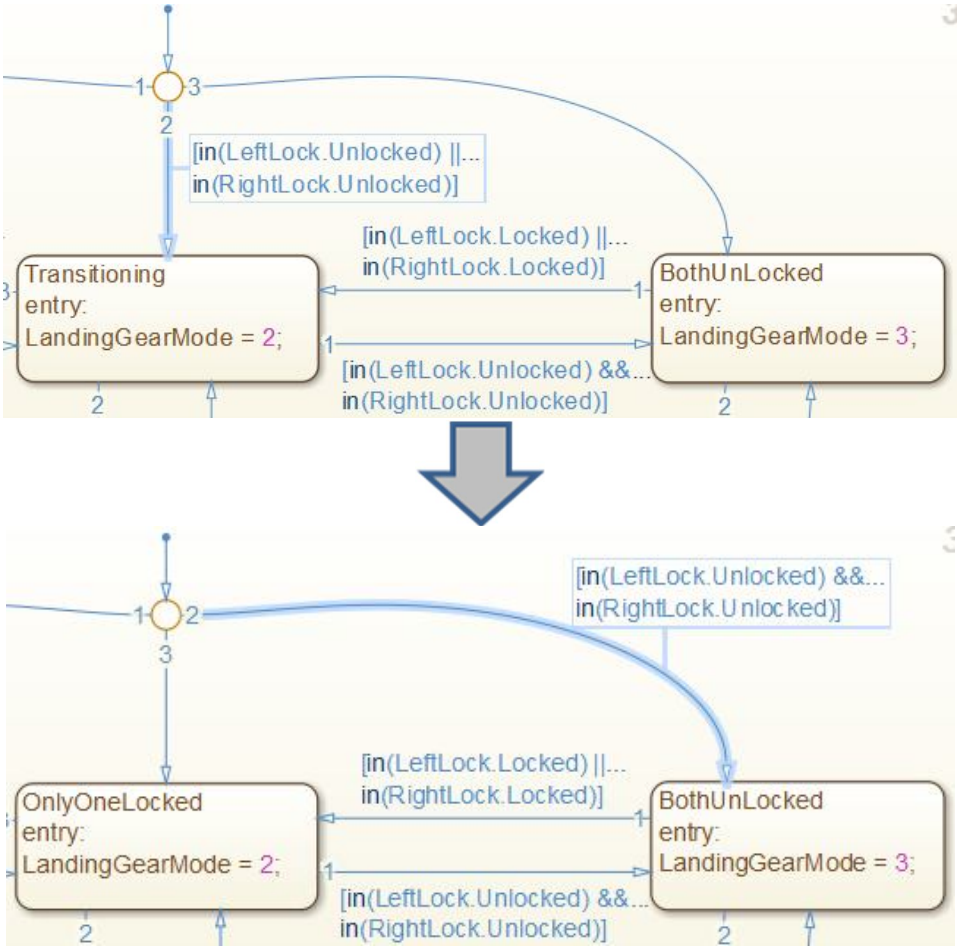

Figure 16. *The initial design (top) and modified design (bottom)*

C. Structural Verification

Executing requirements-based tests may not verify all the design elements and subcomponents. In practice, requirements could be lacking tests, the requirements could be ambiguous or incomplete, and the design could contain superfluous elements[6]. Model coverage metrics provided by Simulink® Verification and Validation ™ software[7] uncover untested portions of the design by displaying coverage information on the model. Model coverage metrics are analogous to code coverage metrics. They can be used to identify design elements that were not executed by the test cases and give an early indication of unforeseen modeling errors. Of the coverage metrics that can be recorded, for this work, the engineers capture decision coverage, condition coverage, and modified condition/decision coverage

Summary

Model Hierarchy/Complexity:

			D1		C1		MCDC	
1. DisplayModel_Step2_done	**34**	78%		55%		27%		
2. . . . LandingGearModes	**33**	78%		55%		27%		
3. . . . . SF: LandingGearModes	32	78%		55%		27%		
4. . . . . SF: LeftLock	4	100%		100%		100%		
5. . . . . SF: RightLock	4	100%		100%		100%		
6. . . . . SF: WarningModes	24	63%		50%		20%		

Figure 17. *Model coverage summary HTML report*

(MC/DC) metrics. Decision coverage examines items that represent decision points, such as states in Stateflow® software. Condition coverage examines blocks that output the logical combination of their input such as transitions in Stateflow® software. MC/DC determines if the logical inputs have independently changed the output.

After stimulating the DUT with the initial set of test cases, the engineers automatically generate a report (see Fig. 17) that documents decision coverage, condition coverage, and MC/DC for the controller. It is observed that the *WarningModes* state does not have full structural coverage and may warrant further investigation. The verification engineer now needs to determine why the test cases he provided did not completely exercise the DUT. He could either try to construct additional tests to meet full structural coverage or use Simulink® Design Verifier™ software, a formal methods analysis engine[8], to augment existing requirement-based tests with additional test vectors that meet all the coverage objectives for the DUT. To generate test vectors, he uses Simulink® Design Verifier™ software to choose MC/DC objectives. The analysis engine identifies a total of 128 coverage objectives for the DUT, for which it can create test cases that satisfy 112 objectives. The formal analysis engine proved that no test case for exercising the remaining 16 test objectives exists (see Fig. 18). The collective coverage for all the test cases is: condition coverage of 95%, decision coverage of 86%, and MC/DC of 82%.

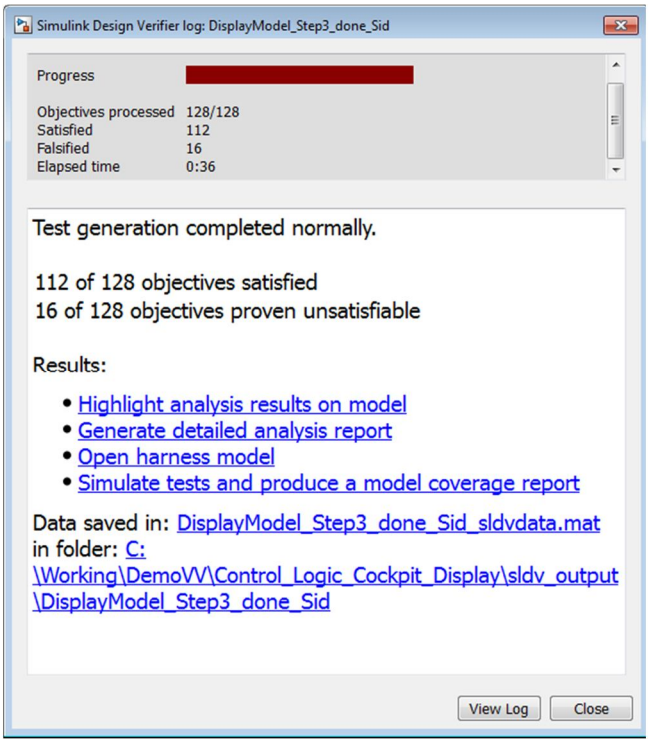

Figure 18. *Simulink Design Verifier analysis results*

From a verification perspective, the engineers are interested in the unsatisfied coverage objectives. Using the color-coded results of the model coverage analysis on the DUT (see Fig. 19a, 19b), the verification engineer identifies that transitions between the *BothLocked* and *BothUnlocked* states are never executed and there can be no test case that excites these two conditions sufficiently. In the model, these transitions are highlighted in red. Essentially, this logic is unused in the design and will lead to dead code or unreachable code in the implemented controller software. Coverage objectives (decision and condition points) highlighted in green within the model depicts parts of the design that are valid and can be exercised by a test case. The tool provides a link from each coverage objective to the test case that stimulates that model path during simulation. For dead logic, there can be no test scenario that stimulates the design path, and hence such logic needs further investigation by the verification and design engineers.

The goal of test generation is twofold. First, find dead logic in the design and, by extension, dead code. Second, automatically generate tests for coverage. Using requirements traceability, the engineers can navigate from the errant transition to the high-level requirement to investigate whether the design is achieving its purpose or is over-specified. The verification and design engineers review these results and decide on the next course of action – either fix the design, clarify the requirement, or modify the requirement. Regarding the automatically generated test vectors, the verification engineer needs to determine if these are part of a requirement that he may have overlooked and for which he did not create a manual test. For a flight control application, he must account for each test case that he creates and associate these to requirement documents. By performing structural and functional verification tasks on the model, the engineers identified erroneous design decisions and rectified these early in the development phase of the landing gear warning mode component.

The test vectors, a combination of functional and automatically generated tests, can be used to verify code running in software-in-the-loop (SIL) and processor-in-the-loop (PIL) test configurations. By frontloading the test-case generation, these engineers are able to reuse the tests for source code and object code verification steps later on.

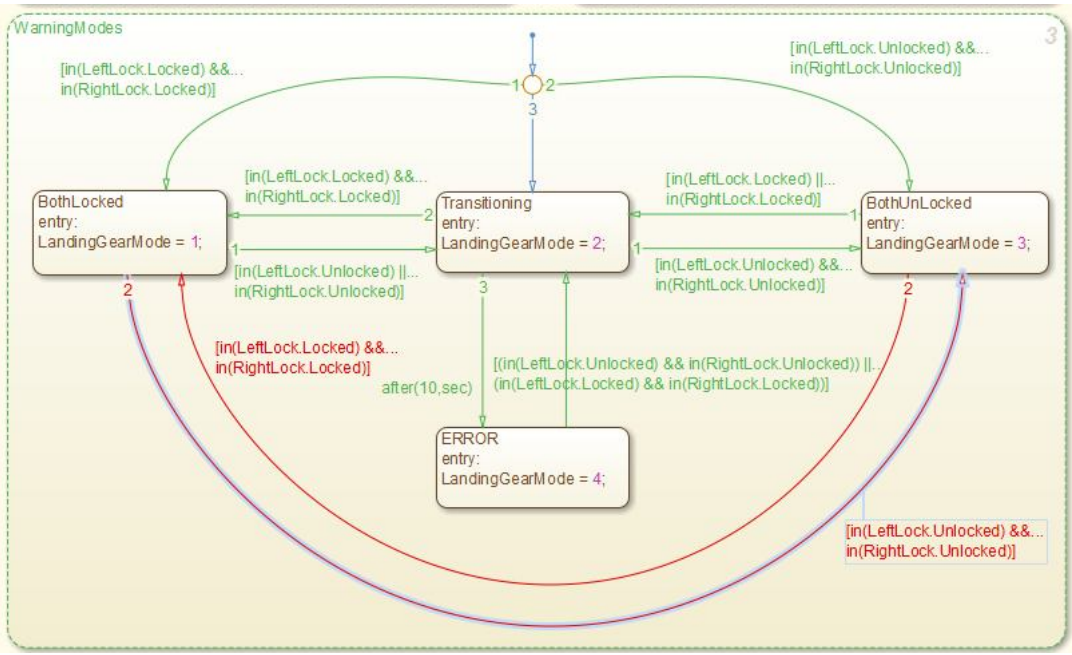

Figure 19a. *Dead logic in the model identified in red*

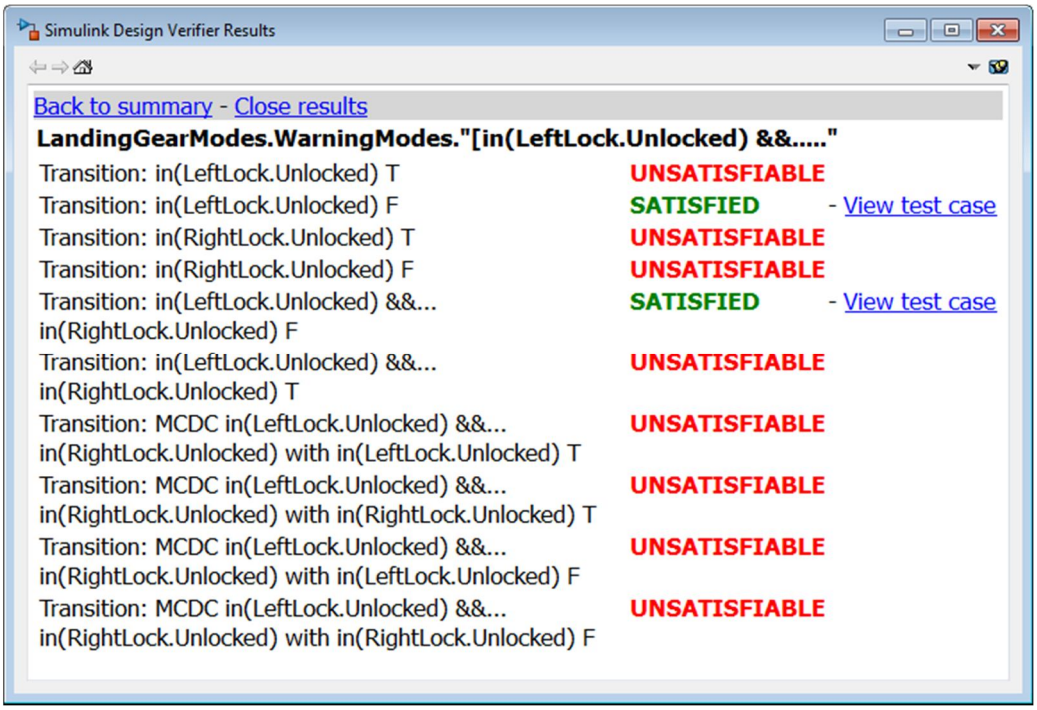

Figure 19b. *Simulink Design Verifier Results*

IV. Summary

This paper presents a workflow for developing complex logic in a cockpit display system efficiently using Model-Based Design. State machines and flow charts were used since they can clearly represent the modes of the aircraft components and the phases of flight, represent multiple display units, all of which are active simultaneously but work independently, and clearly model competing priorities for display.

Creating an executable system specification in the form of a model enabled continuous verification and validation during the design cycle. Simulation based tests were used to verify the functional behavior of components and uncovered an incorrect mode of operation. Additionally, by using automatic test generation and model coverage metrics, redundant design elements were detected that would result in dead code on the controller. Requirements traceability was used to navigate from the design model to its associated requirement to manage and track changes in both.

V. References

[1]Barnard, P., "Graphical Techniques for Aircraft Dynamic Model Development," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, Rhode Island, August 2004.

[2]Barnard, P., "Software Development Principles Applied to Graphical Model Development," *AIAA Modeling and Simulations Conference and Exhibit*, San Francisco, 2005.

[3]Turevskiy, A., Gage, S. and Buhr, C., "Model-Based Design of a New Light-weight Aircraft," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, South Carolina, 2007.

[4]Lin, J., "Measuring Return on Investment of Model-Based Design," *EE Times*. May 2011

[5]"Stateflow User's Guide", The MathWorks Inc., Natick, MA, March 2013

[6]Ghidella J., and Mosterman P., "Requirements-Based Testing in Aircraft Control Design", *AIAA modeling and Simulation Technologies Conference and Exhibit*, 2005

[7] "Simulink Verification and Validation User's Guide", The MathWorks Inc., Natick, MA, 2013

[8]"Simulink Design Verifier User's Guide", The MathWorks Inc., Natick, MA, March 2013