AIAA Guidance, Navigation and Control Conference and Exhibit, San Francisco, August 15-18, 2005.

Formal Verification of Flight Critical Software

Steven P. Miller[*], Elise A. Anderson[†], Lucas G. Wagner[‡], and Michael W. Whalen[§]
Rockwell Collins Inc, Cedar Rapids, IA, 52498, USA

Mats P. E. Heimdahl[**]
University of Minnesota, Minneapolis, MN, 55455, USA

Recent advances in modeling languages have made it feasible to formally specify and analyze the behavior of large system components. Synchronous data flow languages, such as Lustre, SCR, and RSML^{-e} are well suited to this task, and commercial versions of these tools such as SCADE and Simulink are growing rapidly in popularity among designers of safety critical systems, largely due to their ability to automatically generate code from the models. At the same time, advances in formal analysis tools have made it practical to formally verify important properties of these models to ensure that design defects are identified and corrected early in the lifecycle. This report describes how such formal verification tools have been applied to the FCS 5000, a new family of Flight Control Systems being developed by Rockwell Collins Inc.

I. Introduction

RECENT advances in modeling languages have made it feasible to formally specify the behavior of large system components. Synchronous data flow languages, such as Esterel[1], Lustre[2], SCR[3], and RSML^{-e} (Ref. 4) seem to be particularly well suited to this task, and commercial versions of these tools such as SCADE[5] and Simulink[6] are growing rapidly in popularity among designers of safety critical systems, largely due to their ability to automatically generate code from models. At the same time, advances in formal analysis tools have made it practical to formally verify important properties of these models to ensure that design defects are identified and corrected early in the software lifecycle.[7,8,9,10,11,12,13,14]

The FCS 5000 is a family of Flight Control Systems (FCS), developed by Rockwell Collins Inc (RCI), for use in business and regional jet aircraft. The mode logic of the Flight Guidance System (FGS) is an important component of the FCS 5000 architecture that determines which lateral and vertical flight modes are to be armed and active at any time. As discussed in several prior papers,[9,11,12,13,14,15] the mode logic of the FGS is inherently difficult to design and verify. To address this, the FCS 5000 team has made use of advanced verification technologies developed by RCI, the University of Minnesota, and the NASA Langley Research Center under NASA's Aviation Safety and Security Program (AvSSP). This paper describes how this technology has been used on the FCS 5000 to find and correct requirements and design errors early in the life cycle.

The remainder of this paper is organized as follows. Section II provides background information, including an overview of a FGS and descriptions of the modeling and analysis tools used in the project. Section III describes how the models are translated into the analysis tools for verification. Section IV shows how a single mode transition diagram is specified and how properties of it are verified. Section V discusses how two mode transition diagrams are composed and verified as a single entity. Section VI compares the scope of the full FCS 5000 effort to these examples and discusses the outcome of the verification to date. Finally, Section VII provides concluding remarks and directions for further work.

[*] Senior Principal Software Engineer, Advanced Technology Center, 400 Collins Road NE, MS 108-206, member AIAA.
[†] Software Engineer, Commercial Systems Flight Control, 400 Collins Road NE, MS 105-191.
[‡] Software Engineer, Advanced Technology Center, 400 Collins Road NE, MS 108-206.
[§] Senior Software Engineer, Advanced Technology Center, 400 Collins Road NE, MS 108-206.
[**] Associate Professor, Dept of Computer Science and Electrical Engineering, 4192 EE/CS Bldg, 200 Union Street, member AIAA.

II. Background

This section provides a brief overview of a Flight Guidance System and descriptions of the modeling and verification tools used in the project.

A. Overview of an FGS

A Flight Guidance System (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. These guidance commands are both displayed to the pilot as guidance cues on the Primary Flight Display (PFD) and sent to the Autopilot (AP) that moves the control surfaces of the aircraft to achieve the commanded pitch and roll.

The internal structure of the FGS can be 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. The mode logic determines which lateral and vertical modes are armed and active at any given time. These in turn determine which flight control laws are generating guidance commands. While very complex, the mode logic consists almost entirely of Boolean and enumerated types. This makes it well suited for analysis using symbolic model checkers. We have used the mode logic of a FGS as an example in several previous studies.[9,11,12,13,14] It is an excellent example because it is complex and representative of a class of problems (control logic consisting of Booleans and enumerated types) frequently encountered in the design of embedded control systems.

B. Modeling and Verification Tools

This section provides an overview of the modeling and verification tools used in the verification of the FCS 5000 mode logic. The mode logic was modeled in Simulink® and analyzed using the NuSMV model checker. SCADE Suite™ and Reactis® were used in conjunction with software developed by RCI and the University of Minnesota to translate the Simulink models into NuSMV. The PVS and SAL tools from SRI International were also investigated as possible analysis tools.

1. Simulink

Simulink®, marketed by The Mathworks, is a popular platform for the modeling and simulation of dynamic systems.[16] It provides an interactive graphical environment and a customizable set of block libraries that can be used to design, simulate, debug, implement, and test reactive systems. Users assemble a system specification by dragging and dropping blocks onto a pallet and connecting the outputs of one block to the inputs of another block. Blocks can be composed hierarchically from simpler blocks, allowing designers to organize complex system designs. New blocks can be defined by the developer and added to a reusable library. Blocks can also be parameterized. Control logic for representing system states and state transitions can be modeled with the integrated StateFlow® add-on. Simulink and StateFlow are both integrated with the MATLAB® environment, also marketed by The Mathworks, providing access to several additional tools for algorithm development, data analysis, data visualization, and numerical computation. Executable code can generated from a Simulink model using the Real-Time Workshop® add-on. An advantage of Simulink is that it can be simulated with fixed or variable-step solvers, allowing both the control system and the plant model (for example, the airframe) to be modeled within the same framework.

2. SCADE

SCADE is an environment for the development of safety-critical systems similar to Simulink. Originally developed for the design of aircraft systems, similar but separate versions are now marketed by Esterel Technologies for the automotive industry (SCADE Drive™) and the avionics industry (SCADE Suite™).[5] SCADE also provides an interactive graphical environment that allows users to assemble system specifications by dragging and dropping blocks onto a pallet and connecting the outputs of one block to the inputs of another. Control logic for representing system states and state transitions can be modeled with the integrated Safe State Machine© (SSM) add-on. Since the SCADE tools were explicitly created for the development safety-critical software and hardware, SCADE supports only fixed step simulation. For the same reason, the features and blocks supported by SCADE and SSM are restricted to those with an unambiguous mathematical representation. An advantage of SCADE is that its models are translated into the Lustre language, a synchronous data flow language with a precise formal semantics.[1] C source code can be generated from SCADE using the KCG™ code generator which has been qualified as a Level A software development tool in accordance with DO178B.[17] The SCADE Suite also includes a gateway that can import Simulink models and a model checker called Design Verifier.

3. Reactis

Reactis® is an automated test-generation and property verification tool for Simulink/StateFlow models developed by Reactive Systems, Inc.[18] It uses random and heuristic search to try to exercise the behavior of models up to a defined level of structural coverage. Reactis supports several different coverage metrics including state-, condition-, branch-, boundary-, and MC/DC-level coverage. The result of the search process is a suite of tests which can be used both for structural testing and validation of the model.

Reactis allows properties to be specified either using a Reactis-specific textual notation or as additional StateFlow machines, and will check whether all tests within a test suite satisfy the properties of interest. Because Reactis uses random, rather than exhaustive, search, it can be used to generate tests and attempt to verify very large models that cannot be analyzed by exhaustive search tools such as model checkers. On the other hand, it is not guaranteed that Reactis will generate all tests necessary to reach a level of structural coverage. Furthermore, it is not possible to use the generated tests to prove whether a given property always holds of a model.

4. NuSMV

NuSMV is a symbolic model checker developed as a joint project between the Formal Methods group in the Automated Reasoning System Division at the Instituto Trintino di Cultura (ITC) - Center for Scientific and Technological Research (IRST), the Mechanized Reasoning Groups at the University of Genova and the University of Trento in Italy, and the Model Checking group at Carnegie Mellon University in the United States. NuSMV is a re-implementation and extension of SMV,[19] the first model checker based on Binary Decision Diagrams (BDDs). NuSMV has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a test bed for formal verification techniques, and applied to other research areas.[20] Properties to be verified in NuSMV are specified using either Computation Tree Logic (CTL) or Linear Time logic (LTL).[21]

The advantage of using a model checker such as NuSMV is that it will check all possible combinations of inputs and state to determine if a property is true. We have used the NuSMV model checker to verify properties of models with over 10^{120} reachable states.

5. PVS

PVS is a theorem prover that has been developed at SRI International's Computer Science Laboratory. In comparison to other widely used verification systems such as HOL and ACL2, the distinguishing characteristic of PVS is that it supports a highly expressive specification language with an interactive theorem prover in which most of the lower-level proof steps are automated. The system consists of a specification language, a parser, a type checker, and an interactive proof checker. The PVS specification language is based on higher-order logic with a richly expressive type system so that a number of semantic errors in a specification can be caught during type checking. The PVS prover consists of a powerful collection of inference steps that can be used to reduce a proof goal to simpler subgoals that can be discharged automatically by the primitive proof steps of the prover. The primitive proof steps involve, among other things, the use of arithmetic and equality decision procedures, automatic rewriting, and BDD-based Boolean simplification.[22,23]

6. SAL

SAL (Symbolic Analysis Laboratory) is a framework for combining different tools to analyze sequential and concurrent systems.[24] The heart of SAL is a language, developed in collaboration with Stanford, Berkeley, and Verimag, for specifying concurrent systems in a compositional way. The SAL framework contains tools for abstraction, invariant generation, program analysis (such as slicing), theorem proving, and model checking. These tools can be used to separate different analysis concerns and calculate properties (i.e., perform symbolic analysis) of sequential and concurrent systems. SAL includes an explicit-state model checker, a BDD-based symbolic model checker, a SAT-based bounded model checker, and a SAT-based infinite bounded model checker which can symbolically analyze systems containing real numbers. SAL can also be used as an interface to the PVS theorem prover.[25]

III. Translating from Simulink to NuSMV

The translation of Simulink models into NuSMV requires several steps. However, these are automated and normally completed without great difficulty. The translation process is illustrated in Fig. 1. Models are first created using MATLAB Simulink and/or StateFlow. These models then are translated into the Lustre formal specification language using one of two tool chains, depending on which path the user is more comfortable with. In one path, the Simulink/StateFlow models are imported into the SCADE Suite using the Simulink Gateway provided by Esterel Technologies. SCADE Suite is then used to translate the models into Lustre. In the second path, the

Simulink/StateFlow models are imported into Reactis and a Lustre file is generated using a translator developed by Rockwell Collins Inc and the University of Minnesota (hereafter RCI-UMN). From Lustre, the models are translated into NuSMV, PVS, or SAL using translators developed by RCI-UMN. The Lustre models can also be imported into Design Verifier, a model checker available in SCADE Suite.

The RCI-UMN translators produce highly optimized models most appropriate for the target language. For example, when translating to NuSMV, the translator produces a specification that is difficult for a human to read, but very efficient for proving properties. When translating to PVS, the resulting specification is optimized for readability and to support the development of proofs in PVS.

Since the FCS 5000 mode logic consists only of Boolean and enumerated types, it is very efficient to verify properties about the mode logic using a BDD-based model checker such as NuSMV. SAL, PVS, and Design Verifier were also investigated and found to be acceptable alternatives. However, due to the speed and ease of use of NuSMV, the bulk of the FCS 5000 verification was done using it. A more detailed comparison of the pros and cons of verification using NuSMV and PVS can be found in Ref. 12.

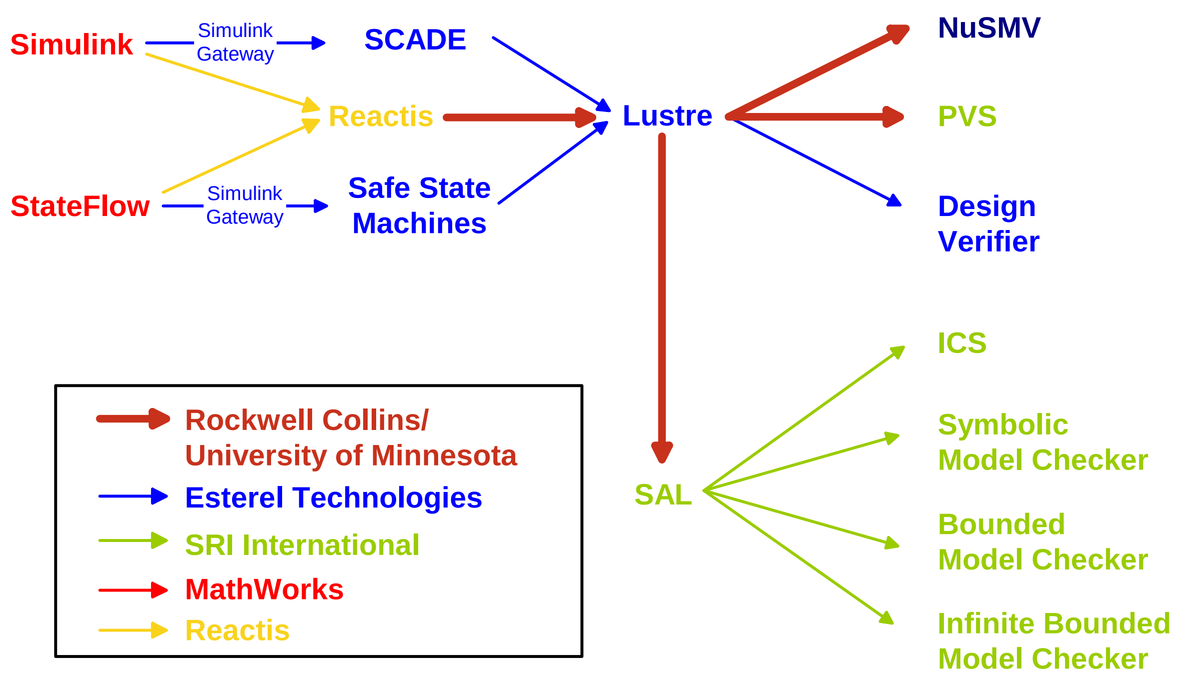

Figure 1. Translation from Simulink to NuSMV

IV. An Example of a Single Mode Transition Diagram

The mode logic of the FCS 5000 is specified using a format developed at RCI. These specifications are then modeled in Simulink, translated to NuSMV for analysis, and then implemented in C and C^{++} code. The actual specifications of the FCS 5000 are highly proprietary and cannot be described here. However, this section uses a simplified example to describe how a single mode transition diagram is specified and verified using NuSMV.

A. Specification of the Lateral Mode Logic

This section illustrates the RCI format for mode transition diagrams using a simple example, discusses some of the subtleties of how events are specified, and concludes with a brief discussion of the advantages and disadvantages of this style as compared to more traditional styles for specifying state transition diagrams.

1. Overview of the LAT Mode Transition Diagram

An example mode transition diagram for a simplified version of the lateral mode logic is shown in Fig. 2.[††] The lateral mode logic determines the active mode that is controlling the aircraft about its lateral axis. The lateral modes are listed across the bottom of the diagram. In this example, these are *ROLL* (hold current bank angle), *HDG* (hold selected heading), *LAPPR* (track a landing beacon during approach), and *LGA* (hold current heading during a go around).

The events that cause transitions are listed down the left hand side (the notation for specifying events is explained more fully in the next section). For ease of reference, all events other than initialization (Power Up) are numbered on the right hand side. If a transition from a state is possible for a particular event, it is indicated by an arrow at the intersection of the line running up from the state and the line crossing horizontally from the event. For example, the arrow at the intersection of the *ROLL* state and the *GA Switch* event (Event 7) indicates that a transition from the *ROLL* state will be made when the *GA Switch* event occurs. The destination state is found by traversing the horizontal line until a black dot is reached and following the associated arrow down to the destination state. For the example just described, the destination state is *LGA*.

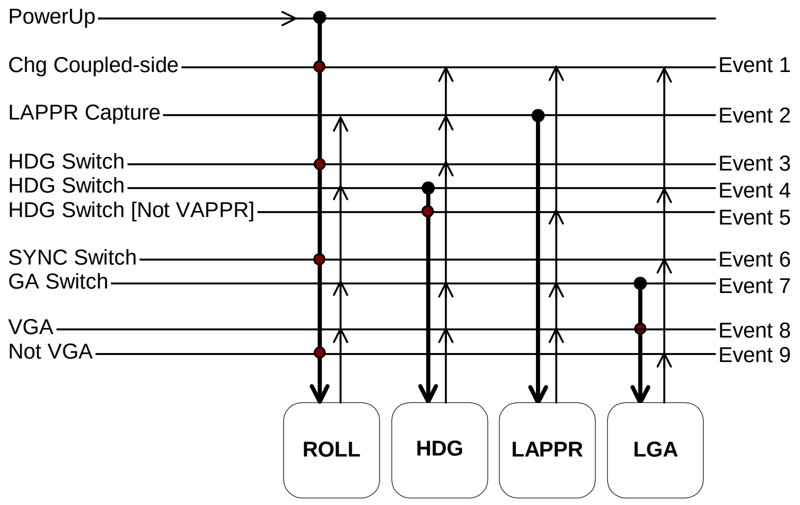

Figure 2. Example Lateral Mode Machine LAT

Sometimes an event can cause a transition to more than one destination state depending on the current state. In this case, the event is listed on the left once for each destination state. For example, the *HDG Switch* event (Event 3) causes a transition to the *ROLL* state when the system is in the *HDG* state, and a transition to the *HDG* state when in *ROLL* or *LGA* (Event 4). An event may also be listed more than once if it is associated with different guard conditions as illustrated by Event 5 (guard conditions will be described in the next section).

If more than one event can occur in the same step, priority is given to the first event that can cause a transition as one moves upward from the current state. Thus, if both the *HDG Switch* event (Event 4) and the *GA Switch* event (Event 7) occur while in *ROLL* mode, Fig. 2 specifies that the transition associated with the *GA Switch* event will be taken and the transition associated with the *HDG Switch* event ignored.

The initial system state is indicated by single horizontal arrow associated with the event of system initialization. In Fig. 2, this is the *Power Up* event and the initial system state is *ROLL.*

2. Specifying Events

Implied in Fig. 2 is a fairly rich notation for specifying events similar to that used in SCR.[3] Events occur at the point in time when a Boolean expression over the system inputs and state changes from false to true. For example, the *HDG Switch* "event" occurs when the value of the *HDG Switch* "input" changes from false to true. If we let `X represent the value of input X just before the event, and X´ represent the value of X just after the event, then the *HDG Switch* event can be more precisely defined as the predicate

[††] For purposes of illustration, this example has been considerably simplified and includes behavior that would not be implemented in an actual Flight Guidance System. For example, the logic here is always armed for Lateral Approach capture. In an actual system, this mode would first have to be enabled by the pilot.

NOT `HDG_Switch & HDG_Switch´

The Boolean expressions used in defining events can be arbitrarily complex. For example, if A and B are Boolean input signals, the event (A & B) can be rewritten, using distribution of the ` and ´ operators and De Morgan's law, as

NOT `(A & B) & (A & B)´ =
NOT (`A & `B) & (A´ & B´) =
(NOT `A OR NOT `B) & (A´ & B´)

Thus, if A is true and B is false, the event (A&B) occurs when B becomes true. In Fig. 2, events are written as simple Boolean expressions and the requirement for a change from false to true of the Boolean expression is implied.

Occasionally, it is necessary to refer not to the event of an input changing value, but to the actual value of the input. Such *guard* expressions are placed in square brackets and refer to the value of the expression just after the associated event[‡‡]. Thus, in Fig. 2, Event 2, *HDG Switch [Not VAPPR]*, is more precisely defined as

NOT `HDG Switch & HDG Switch´ & NOT VAPPR´

It is also possible to define an event that consists solely of an expression in square brackets. In that case, the event is interpreted as occurring at each point in time when the expression is true. Such an event occurs continuously until the expression becomes false.

3. Advantages and Disadvantages of this Format

The format illustrated in Fig. 2 has both advantages and disadvantages over more conventional representations of state machines such as Statecharts.[26] Its primary advantage is that it is easily understood and provides a very concise way of specifying a deceptively large number of transitions on a single diagram. This works particularly well when the states are highly connected by transitions. In fact, it was exactly this problem that prompted Leveson to introduce the concept of a transition bus.[27] The format of Fig. 2 goes beyond this by refining the transition bus into individual "event buses" so that the events associated with the transitions between any two states are clearly visible. This format is also equivalent to the mode tables of SCR[3] and CoRE[28]. However, its graphical presentation is more concise and easier to understand than the tabular presentations of SCR and CoRE.

This format also has some disadvantages over conventional representations of state machines. It does not represent state hierarchy well and discourages developers from thinking in terms of meta-states. As a consequence, meaningful groupings of states may not be named and introduced into the developer's lexicon. Another consequence is that transitions have to be repeated for each state in a meta-state, where in Statecharts the transitions could be specified once for the meta-state.

Finally, no commercial tools provide explicit support for this format. To cope with this problem, RCI has developed a proprietary method for encoding these diagrams as Simulink models. These Simulink models are then used for analysis, simulation, and possibly for automatic generation of source code.

B. Verifying Properties of LAT

This section discusses how properties of a single mode machine such as LAT can be verified using a model checker such as NuSMV. The specification of LAT given in Fig. 2 is essentially the requirements for the lateral mode logic, so there is very little that can be cross checked to verify the correctness of Fig. 2 itself. However, it is useful to verify that the Simulink model does correctly implement LAT. This can be checked by proving a set of properties that can be extracted directly from Fig. 2. These properties are also excellent specifications for test cases to verify that the generated code executing on the target platform performs correctly.

To do this, we first convert the Simulink model to NuSMV as described in Section III, then prove various CTL properties over this translated model. For example, to check that the *VGA* event (Event 8) causes the model to transition from *ROLL* mode to *LGA* mode, we prove the CTL property

$$\text{AX AG(ROLL} \rightarrow \text{AX(Event8} \rightarrow \text{LGA))}$$

This property states that for all globally reachable (AG) states for which the mode is *ROLL*, the mode in all next states (AX) will be *LGA* if Event 8 occurs. It is necessary to preface each CTL formula with the AX operator to step

[‡‡] This differs from the convention in SCR, where guard expressions are normally associated with the value just prior to the event. Using the value just after the event better matches the intention of the specifier in case the guard condition changes value at the exact moment of the event. It is also more naturally implemented in tools such as Simulink.

over the initial state as the values of the inputs and intermediate NuSMV variables are unspecified in the NuSMV representation of this state.

In like fashion, to prove that the system transitions from *ROLL* mode to *LGA* mode when the *GA Switch* event (Event 7) occurs, we prove the CTL property

$$\text{AX AG(ROLL} \rightarrow \text{AX((Event\_7 \& !(Event\_8))} \rightarrow \text{LGA))}$$

We must explicitly rule out the possibility of Event 8 since it has a higher priority than Event 7 and would mask the effect of Event 7. In the same way, to prove that the system transitions from *ROLL* mode to *HDG* mode when the *HDG Switch event* (Event 4) occurs, we check the CTL property

$$\text{AX AG(ROLL} \rightarrow \text{AX((Event\_4 \& !(Event\_7 | Event\_9))} \rightarrow \text{HDG))}$$

This process is repeated for each possible transition from *ROLL* mode. Finally, to verify that no other transitions from *ROLL* mode are possible, we check the CLT property

$$\text{AX AG(ROLL} \rightarrow \text{AX(!(Event\_2 | Event\_4 | Event\_7 | Event\_8)} \rightarrow \text{ROLL))}$$

stating that the system remains in *ROLL* mode if none of the events leading to a mode transition occur. A similar set of properties can then be checked for each of the other modes in LAT.

Proving these properties of LAT does provide a useful check on the correctness of the Simulink model of LAT, but very little insight into the correctness of Fig. 2 itself. More interesting properties are discussed in the next section in which LAT is composed with the vertical mode machine, VER.

V. An Example of Interacting Mode Transition Diagrams

It is often desirable to specify several mode transition diagrams separately, then compose them into a single system, often with strong interactions between the components. However, this raises several problems for formal verification. To illustrate this, this section introduces a second mode transition diagram for the vertical mode logic, composes it with the lateral mode logic, and discusses some of the problems raised during formal verification.

A. Specification of the Vertical Mode Logic

In this section we introduce a second mode machine VER representing the vertical mode logic (Fig. 3). VER is very similar to the mode transition diagram LAT discussed in Section IV. However, while the modes of LAT control the aircraft about the lateral axis, the modes of VER control the aircraft about the vertical axis. For example, PITCH mode holds the aircraft to a fixed pitch angle, AIRSPD holds the aircraft to a fixed airspeed by adjusting its pitch angle, VAPPR holds the aircraft to the vertical glide slope during approach, and VGA holds the aircraft to a fixed pitch angle during a go around.

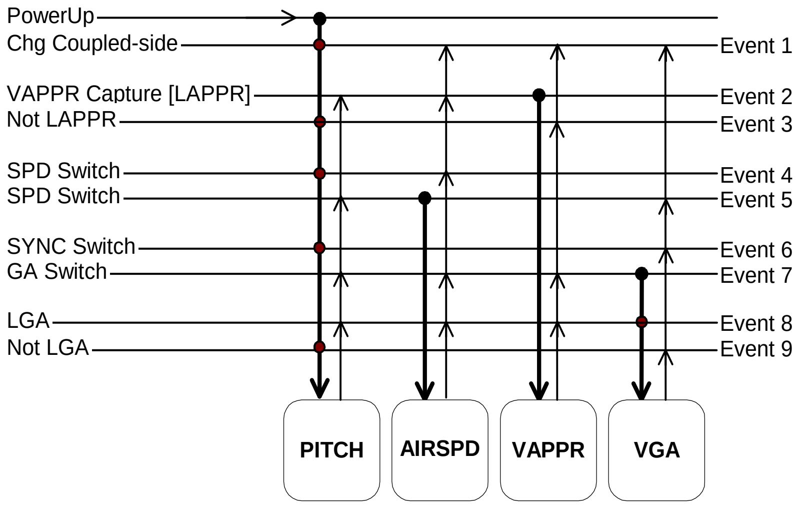

Figure 3. Example Vertical Mode Machine VER

B. Composing LAT and VER in Simulink

Examination of LAT (Fig. 2) and VER (Fig 3) reveals that some of the events of LAT refer to the state of VER and vice versa. For example, LAT transitions in and out of LGA mode as VER transitions in and out of VGA mode (Events 8 and 9 of Fig. 2), and VER exhibits similar behavior relative to LAT. Pressing the HDG switch has no effect on LAT when it is in LAPPR mode if VER is in VAPPR mode (Event 5 of Fig. 2), and VER cannot enter VAPPR mode unless LAT is in LAPPR mode (Event 2 of Fig. 3).

Clearly, there is an intent by the author to synchronize the states of LAT and VER. For example, it seems clear that the designer intends LAT to be in state LGA if and only if VER is in state VGA. Similarly, VER should be in VAPPR only if LAT is in LAPPR. These relationships can be stated more formally as the properties

$$\text{LGA} \leftrightarrow \text{VGA}$$
$$\text{VAPPR} \rightarrow \text{LAPPR}$$

However, our ability to enforce these properties depends critically on how LAT and VER interact. If they are composed synchronously, these relationships are much easier to maintain than if LAT and VER are composed asynchronously. When composed synchronously, both machines execute in lockstep and share a single clock. All inputs to the overall system are computed at the start of each step and held fixed until the step is completed. Communication between the two machines consists of providing the current mode of each machine as an input to the other machine. Of course, this leads to a cyclic dependency (algebraic loop) in which each machine defines its current state in terms of the current state of the other. This dependency must be broken by introducing a one step delay somewhere between the two machines.

An example of the synchronous composition of LAT and VER is shown in the Simulink model of Fig. 8. The block labeled *LAT* implements the Simulink model depicted in Fig. 2 and the block labeled *VER* implements the analogous Simulink model for VER specified in Fig. 4. Note that *LAT* and *VER* share three inputs, *Chg Coupled Side*, *SYNC Switch*, and *GA Switch*, which they are assumed to see simultaneously and at the same time as their other inputs. In each step, *LAT* computes its next mode which is used by *VER* in the same step in the computation of its next mode. The value of *VER Mode from the previous step* is in turn used by *LAT* in computing its next mode.

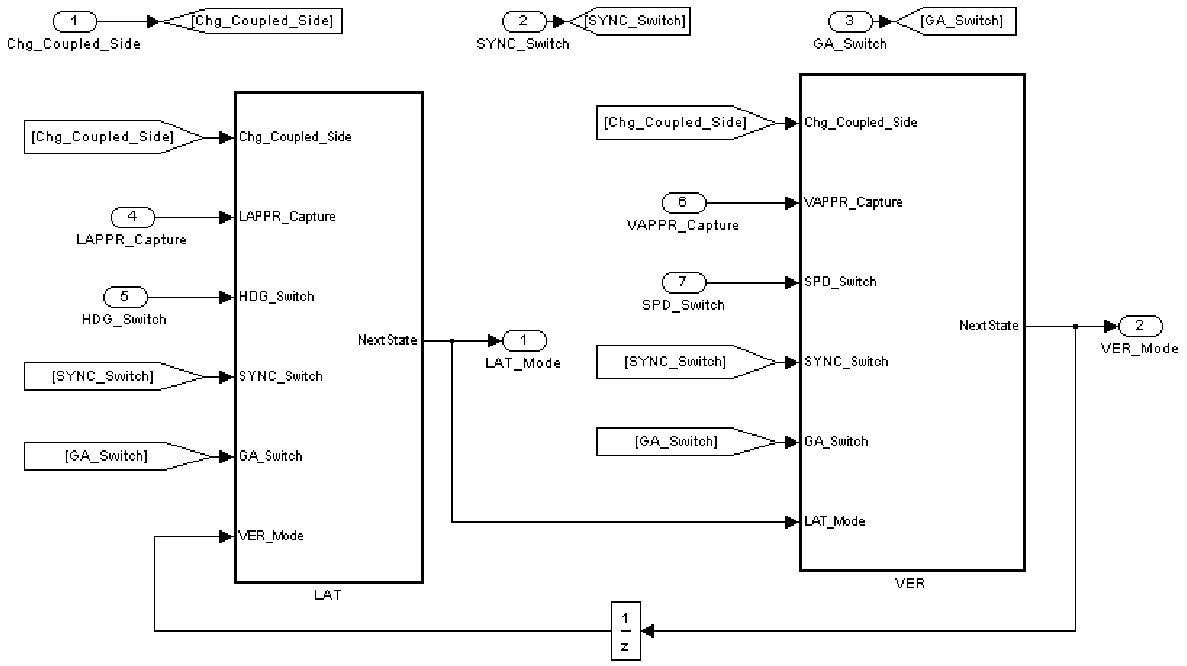

Figure 4. Synchronous Composition of LAT and VER

The synchronous composition of LAT and VER as a single Simulink model can easily be implemented as a single block of code executing on a single processor. A clear advantage of this is that the analysis of properties of the overall system (for example, showing that LGA $\leftrightarrow$ VGA and VAPPR $\rightarrow$ LAPPR) can be done using the same techniques and tools as were used when verifying LAT.

However, it may be necessary to implement LAT and VER as asynchronous processes executing on separate processing platforms, each with their own clock. This occurs when we need to meet system safety requirements by implementing redundant copies on isolated platforms, when we need to improve system throughput, or simply to take advantage of an existing system architecture. Such systems are often referred to as Globally Asynchronous/Locally Synchronous (GALS) architectures.

An approach for modeling and formally analyzing such systems based on that of Ref. 29 was explored in Ref. 30. While relatively straightforward to model, systems with such unbounded asynchrony are far more difficult to analyze. For one thing, the desired properties are usually more difficult to specify. For example, the property LGA $\leftrightarrow$ VGA will not hold at possible times since one machine may change state slightly before or after the other machine changes state. In addition, the number of reachable states grows dramatically compared to the synchronous case since all possible interleaving of the individual state machines have to be considered. While our simple example could be formally verified using this approach, analyzing the mode logic of a full FGS with unbounded asynchrony would likely exceed the capacity of current verification tools.

Another alternative is to implement LAT and VER as asynchronous processes, but force them to observe a logical synchrony. By doing this, an analysis of the synchronous composition shown in Fig. 4 would also apply to the asynchronous implementation. However, this places the burden of implementing a mechanism for enforcing logical synchrony on the developer.

Yet another approach is to compose the individual components synchronously with a single step delay in each direction between them. In Fig. 4 this would be accomplished by adding a one step delay (1/z) block on the signal from LAT to VER. Proof of a property of such a synchronous composition of two machines is not sufficient to prove that a property holds in the actual implementation as it does not consider the cases where the communication of state from one machine to the other takes two or more steps. However, it will find a very common class of errors where the property of interest fails to hold because one machine processes an external input before processing the state change from the other machine. This turns out to be a very effective debugging technique since the model checker will consider all possible combinations of inputs that might occur during the one step delay.

C. Verification of LAT and VER

There are several properties we wish to verify of LAT and VER using a model checker such as NuSMV. This section illustrates this process when LAT and VER are composed synchronously as shown in Fig. 4.

1. *Proving LGA $\leftrightarrow$ VGA*

One desirable property mentioned earlier is that the system should in Lateral Go Around mode if and only if it is in Vertical Go Around mode, i.e., LGA $\leftrightarrow$ VGA. To check this, we first try proving the CTL formula

$$\text{AX AG(LGA} \leftrightarrow \text{VGA)}$$

Unfortunately, this turns out to be false as show by the counterexample of Table 1.

Table 1. Counterexample to AX AG(LGA $\leftrightarrow$ VGA)

	STEPS	1	2	3	4
INPUTS					
	Chg_Coupled_Side	1	0	1	1
	SYNC_Switch	1	0	1	1
	GA_Switch	1	0	1	1
	LAPPR_Capture	1	1	0	1
	HDG_Switch	1	0	1	1
	VAPPR_Capture	1	1	1	1
	SPD_Switch	1	1	0	1
OUTPUTS					
	LAT_Mode	ROLL	ROLL	LGA	LGA
	VER_Mode	PITCH	PITCH	VGA	AIRSPD

In step 3, the GA switch is pressed, causing LAT to enter state LGA via Event 7 of Fig. 2. The change of LAT from ROLL to LGA causes VER to enter state VGA in step 3 due to Event 8 of Fig. 3 (Event 7 of Fig.. 3 could also cause VER to enter state VGA, but it is overridden by the higher priority Event 8). In step 4, the SPD switch is pressed, causing VER to enter state AIRSPD via Event 5. However, LAT will not change to state LGA via Event 9 of Fig. 2 until the next step due to the one step delay introduced between VER and LAT to break the cyclic dependency.

Clearly, our property is wrong since the counterexample does describe the correct behavior of the composition LAT and VER. In reality, it is possible for our property to be wrong for one step due to the one step delay that must be introduced between VER and LAT. However, it should never be the case that the property is wrong for two steps in a row. This can be stated as the CTL property

$$AX\ AG((LGA \leftrightarrow VGA) \mid AX(LGA \leftrightarrow VGA))$$

This states that the property LGA $\leftrightarrow$ VGA must either be true in the current state or it must be true in all possible next states. However, trying to prove this property results in yet another counterexample as shown in Table 2.

Table 2. Counterexample to AX AG((LGA $\leftrightarrow$ VGA) | AX(LGA $\leftrightarrow$ VGA))

	STEPS	1	2	3	4	5
INPUTS						
	Chg_Coupled_Side	1	0	1	1	1
	SYNC_Switch	1	0	1	1	1
	GA_Switch	1	0	1	0	1
	LAPPR_Capture	1	1	0	1	1
	HDG_Switch	1	0	1	1	1
	VAPPR_Capture	1	1	1	1	1
	SPD_Switch	1	1	0	1	1
OUTPUTS						
	LAT_Mode	ROLL	ROLL	LGA	LGA	ROLL
	VER_Mode	PITCH	PITCH	VGA	AIRSPD	VGA

This counterexample starts out the same as the previous one. In step 5, we even see LAT transition to state ROLL via Event 9 as it realizes that VER exited VGA mode in the previous step. However, at the same time, VER transitions back into VGA mode via Event 7 when the GA switch is pressed. This causes the property LGA $\leftrightarrow$ VGA to be false for both steps 4 and 5, but for very different reasons in each step.

A moment's reflection convinces us that if the pilot manages to press the GA button at precisely the right moments, he or she can keep the property LGA $\leftrightarrow$ VGA false indefinitely. However, this is highly unlikely and not what we are truly concerned about. What we really want to prove is that *in the absence of new inputs* the system will quickly stabilize to a state in which LGA $\leftrightarrow$ VGA holds. To do this, we define a predicate EVENT to NuSMV that is true whenever any of the events of Fig. 2 or Fig. 3 occur. We can then restate our CTL property as

$$AX\ AG((LGA \leftrightarrow VGA) \mid AX(!Event\ \rightarrow (LGA \leftrightarrow VGA)))$$

This states that if the property LGA $\leftrightarrow$ VGA isn't true in a given state, it will be true in all next states *providing no input event occurs in that next state*. This property is what we wish to prove as it guarantees that in the absence of new inputs, the longest that our property can be false is one step. This property turns out be true. It is worth noting how much complication even a single step delay can introduce in the externally visible behavior of the overall system and in the properties we wish to prove.

2. Proving VAPPR $\rightarrow$ LAPPR

Another desirable property is that the system should be in Lateral Approach mode whenever it is in Vertical Approach mode, i.e., VAPPR $\rightarrow$ LAPPR. To check this, we try proving the CTL formula

$$AX\ AG(VAPPR \rightarrow LAPPR)$$

which does prove successfully. Of course, the reason for this is that there is not a one step delay between LAT and VER and VER responds immediately when LAT exits the LAPPR state.

3. *Proving Stabilization of Communicating Mode Machines*

A concern with communicating state machines is ensuring that it is not possible to enter an infinite cycle in which a change in one machine causes a change in the other machine, which in turn causes a change in the first machine, and so on in an infinite cascade of events. It is possible to check for such cycles using the model checker, but we must first develop some additional infrastructure.

First, we are only interested in stabilization in the quiescent state in which no external inputs are driving the system. The definitions necessary for this have already been developed in our earlier example where we defined the Boolean predicate EVENT to be true if any external input event occurs in a given step. We next define stabilization to mean that if no input event occurs, the lateral and vertical modes will not change. If we expect this to occur in a single step, we would like to write this in CLT as

$$\text{AX AG}((!\text{EVENT} \ \&\ \text{LAT\_Mode} = l) \rightarrow \text{AX}(!\text{EVENT} \rightarrow \text{LAT\_Mode} = l))$$
$$\text{AX AG}((!\text{EVENT} \ \&\ \text{VER\_Mode} = v) \rightarrow \text{AX}(!\text{EVENT} \rightarrow \text{VER\_Mode} = v))$$

where *l* and *v* are logical variables representing the lateral and vertical modes in the two steps. Unfortunately, CLT does not support the use of such logical variables. However, we can achieve the same effect by defining *manifest constants l* and *v* by adding the following NuSMV statements to our model:

```
VAR      l: 0..4;      v: 0..4;
ASSIGN   next(l) := l;    next(v) := v;
```

These define NuSMV variables *l* and *v* that can take on the values 0 through 4 (the possible values of the state variables *LAT_Mode* and *VER_Mode*. The ASSIGN statements ensure that *l* and *v* cannot change their values from one step to the next. However, since we do not specify their initial value, they can take on any value in the initial state and can thus be used in a way similar to logical variables in a CTL statement. With this infrastructure in place, we can now check our two CLT properties and find that they do indeed hold, showing that our system does stabilize in a single step and does not have any infinite cycles.

Another way to check for stabilization would be to use the UNTIL operator U of CTL. Our two CTL stabilization properties would then be written as

$$\text{AX AG}((!\text{EVENT} \ \&\ \text{LAT\_Mode} = l) \rightarrow (\text{A}[\text{LAT\_Mode} = l \text{ U EVENT}]))$$
$$\text{AX AG}((!\text{EVENT} \ \&\ \text{VER\_Mode} = v) \rightarrow (\text{A}[\text{VER\_Mode} = v \text{ U EVENT}]))$$

These properties state that if no event occurs in a reachable state, then the lateral mode and vertical mode will remain unchanged in all following reachable states until an input event occurs. However, U is the *strong* UNTIL operator of CTL, which requires that the predicate following U *must* eventually hold for each possible path. Of course, there is a path in our system in which no additional input event ever occurs (e.g., the pilot never presses a button and a capture condition never occurs), and hence both properties fail. What is needed is the *weak* UNTIL operator (W), which does not require the predicate following it to eventually hold. Our CTL properties can then be written and proven as

$$\text{AX AG}((!\text{EVENT} \ \&\ \text{LAT\_Mode} = l) \rightarrow (\text{A}[\text{LAT\_Mode} = l \text{ W EVENT}]))$$
$$\text{AX AG}((!\text{EVENT} \ \&\ \text{VER\_Mode} = v) \rightarrow (\text{A}[\text{VER\_Mode} = v \text{ W EVENT}]))$$

Unfortunately, CTL does not support the weak UNTIL operator W. However, the same logical formula can be written by making use of the identity

$$\text{P W Q} \equiv (!\text{P} \ \&\ \text{Q}) \rightarrow (!\text{E}[!\text{P U } !(\text{P} \mid \text{Q})])$$

Our CTL properties for stabilization can then be written as

$$\text{AX AG}((!\text{EVENT} \ \&\ \text{LAT\_Mode} = l) \rightarrow (!\text{E}[!\text{EVENT U } !\ (\text{LAT\_Mode} = l \mid \text{EVENT})]))$$
$$\text{AX AG}((!\text{EVENT} \ \&\ \text{VER\_Mode} = v) \rightarrow (!\text{E}[!\text{EVENT U } !\ (\text{VER\_Mode} = v \mid \text{EVENT})]))$$

which are easily proven using the model checker.

VI. Verification of the FCS 5000 Mode Logic

In this section, we discuss the formal verification of the FCS 5000 mode logic currently underway. Details of the FCS 5000 design are proprietary and cannot be described here. However, it is possible to discuss the scope of the effort, what information is being collected, and the number and sorts of errors found to date.

A. Scope of the Effort

The FCS 5000 mode logic being analyzed consists of five mode transitions diagrams, each considerably larger than those shown in Fig. 2 and Fig. 3. There are 36 modes, 172 events, and 488 transitions. The events are typically guarded by Boolean expressions, and the guards of different events often share the same input values. Changes in the state of each mode diagram affect at least one, and often more than one, of the other mode diagrams. While each individual diagram is relatively clear and straightforward to understand, grasping all the possible interactions between them can be quite difficult.

The most interesting properties of the mode logic define relationships that must be maintained between the states of pairs of mode machines. For this reason, it is usually sufficient to analyze pairs of mode machines connected synchronously as discussed in Section V. The state of the other mode machines are treated as unconstrained inputs to the pair being analyzed, so the desired properties are shown to hold regardless of the state of the other mode machines.

Unfortunately, because of the size of the mode transition diagrams, it is not feasible to analyze such pairs of mode machines if they are composed as two synchronous models executing completely asynchronously, i.e., where each is driven by its own clock and the clocks are not synchronized. For this reason, they were modeled as two communicating synchronous models with a one step delay in *each* direction as discussed in Section V. Even in this constrained configuration, there are typically over 10^{20} reachable states (i.e., the cross product of the possible inputs and the states of the mode machines) that need to be checked in the verification. As discussed in Section V, model checking over this configuration is not sufficient to prove that the properties hold when the mode machines are implemented as asynchronous processes, but it is still a very useful debugging technique. For these reasons, the model checking was performed in addition to the traditional verification activities.

B. Information Collected

As errors were detected, they were logged and tagged with the date found, the individual that found the error, where the error was found, a description of the error, how the error was found, the severity of the error, and how the error was resolved. Most of these are straightforward, but the classification of how the error was found and the severity of the error merit more discussion.

Errors were not just found through model checking. Simply studying the mode transition diagrams prior to modeling found some errors, which were classified as being found through “Inspection”. Other errors were found while actually translating the mode transition diagrams into Simulink. These were classified as being found through “Modeling”. Errors were also found by simulating (executing) the models, and of course, through model checking. The complete list is given in Table 3.

Table 3. Classifications of Error Detection

Classification	Description
Inspection	Error found by manual review or inspection of the specification.
Modeling	Error found during the process of creating the Simulink model.
Simulation	Error found while executing the Simulink model.
Analysis	Error found through model checking or other analysis of the Simulink model.

Another desirable classification is some notion of the importance, or severity, of the error. However, this is surprisingly difficult to do in an objective way. For example, is an error in following documentation standards that requires thousands of hours to correct a trivial or major error? Is a coding error that could violate a system safety property, but is found in the first code review and requires only one line of code to be changed, a trivial or a major error? We have found it much easier to classify an error on how likely it is that it would have been detected through traditional verification techniques at some point during development. This classification scheme is shown in Table 4.

Table 4 – Classifications of Error Severity

Classification	Description
Trivial	The error is trivial – it does not matter if it is detected (i.e., spelling or punctuation errors).
Likely	It is very likely the error would have been detected by traditional verification techniques.
Possible	Is it possible the error would have been missed by traditional verification techniques.
Unlikely	It is very unlikely the error would have been detected by traditional verification techniques.

C. Errors Found

The mode transition diagrams are being analyzed individually and in various combinations. To date, a total of 26 errors have been recorded, which are summarized in Table 5 below.

Table 5. Summary of Errors Found

Dectected By	**Likelihood of Being Found by Traditional Methods**				
	Trivial	Likely	Possible	Unlikely	Total
Inspection			1	2	3
Modeling		5	1		6
Simulation					
Model Checking	2	1	13	1	17
Total	2	6	15	3	26

As Table 5 indicates, model checking of the mode transition logic has been quite effective at finding errors. This is even more remarkable since the project is still in the early stages of architectural design and requirements capture. It is well known that finding errors early in the lifecycle is far more valuable than finding them during integration testing. It's also worth noting that model checking and inspection tend to find the errors most likely to be missed by other techniques. This is not unexpected, since model checking allows us to apply a form of exhaustive testing. However, even more insight can be gained by considering the errors found when analyzing a single mode transition diagram versus the errors found when analyzing interacting mode transition diagrams. This is done in the following sections.

1. Errors Found by Analysis of a Single Mode Transition Diagram

The analysis of a single mode transition diagram is straightforward and is done as described in Section IV.B. Properties are extracted directly from the diagram for each transition and verified. As discussed in Section IV.B, these properties serve mainly to check that the manual translation of the mode transition diagrams into Simulink has been done correctly. In fact, these properties have found only one error, an incorrect implementation of the rising edge block in the initial system state (it defaulted to true if its input was true in the initial state). We view this as an indication of the ease with which the mode transition diagrams can be manually translated into Simulink.

However, unlike the simple examples of Section IV.B, it was also often possible to identify more general properties that the developer had embedded in the logic of a single machine. As these properties identify more general requirements, they were useful checks that did result in the identification of several errors.

An example was a property stating that when a particular event occurred, the system should always transition to ROLL mode. When this property was checked, it was discovered that the specifier had not defined a transition from ROLL mode back to itself when this event occurred. Since the objective was to be in ROLL mode immediately following the event, this seemed reasonable, even to the verifiers. However, the model checker revealed that if another lower priority event occurred at the same time, the system would take the lower priority event to a different state since there was no higher priority transition from ROLL back to itself to pre-empt it. The fix for this was simply to add another "arrow" to add the higher priority transition from ROLL back to itself.

This was a very subtle error that was easy to miss through reviews since humans tend to focus on the normal case behavior and are biased towards considering only one input event at a time. However, the model checker makes no such assumptions and considers all possible combinations of states and inputs. It's also worth pointing out that since implementation of the diagrams is now a routine, almost mechanical task, the implemented code would have very likely exhibited this same behavior if the error had not been caught during testing or inspections.

A summary of the errors found when analyzing a single mode transition diagram are shown in Table 6. It is worth noting that half of the errors were found through model checking, and that of these, two thirds could be missed by traditional verification activities, supporting our assertion that model checking tends to find errors missed by other techniques. It is also worth noting that simple inspection of the model found the two most serious errors. However, we believe that if these errors had not been found by inspection, they would have been found by model checking.

Table 6. Errors Found by Analysis of a Single Mode Transition Diagram

Dectected By	**Likelihood of Being Found by Traditional Methods**				
	Trivial	Llikely	Possible	Unlikely	Total
Inspection			1	2	3
Modeling		5	1		6
Simulation					
Model Checking	2	1	6		9
Total	2	6	8	2	18

2. Errors Found by Analysis of Interacting Mode Transition Diagrams

As discussed in Section V, verification of interacting mode transition diagrams is more difficult and is just getting underway. Even so, it has found several errors, which are summarized in Table 7.

Table 7. Errors Found by Analysis of Interacting Mode Transition

Dectected By	**Likelihood of Being Found by Traditional Methods**				
	Trivial	Likely	Possible	Unlikely	Total
Inspection					
Modeling					
Simulation					
Model Checking			7	1	8
Total			7	1	8

From Table 7 it can be seen that analysis of interacting mode transition diagrams tend to find errors that are unlikely to be detected by traditional methods. This occurs because these properties check for relationships that are to be maintained between the mode transition diagrams. This is a difficult task for human beings, but one for which a model-checker is admirably suited.

VII. Conclusions and Directions for Future Work

We have discussed how the mode logic of the FCS 5000 Flight Control System is specified, modeled in Simulink, and formally verified using the NuSMV model checker. The mode logic is specified in a format developed at RCI that is easily understood and provides a very concise way of specifying a large number of transitions on a single diagram. It is particularly well suited for state machines that exhibit little hierarchy and in which the states are highly interconnected by transitions. It is, in effect, an enhancement of the "transition bus" proposed several years ago by Leveson.[27]

To provide tool support for this format, RCI has developed a template for translating mode transition diagrams into Simulink. The primary advantage of this approach is that it is flexible and can be easily changed as the mode transition diagrams are changed. It also makes manual translation of the mode transition diagrams into Simulink a straightforward and simple task.

We have also shown how the Simulink models of the mode transition diagrams can be automatically translated into NuSMV and analyzed using the NuSMV model checker. Properties can be easily extracted from the mode transition diagrams to verify that the Simulink model correctly implements the mode transition diagram. Since the code for the FCS 5000 mode logic will be auto-generated or manually implemented directly from the Simulink models, these properties serve as an important check that mode transition logic has been correctly implemented. These same properties can also be used as test cases to ensure that the actual object code executing on the target platform is correct.

We also discussed the issues in composing interacting mode transition diagrams and formally verifying properties that must be maintained between them. Verification of such properties is important as they are particularly difficult to get right.

Finally, we discussed the application of model checking to the FCS 5000 mode logic. While not completed, this has already found 26 errors in the early versions of the mode logic, many of which might not have been discovered through traditional verification techniques. Finding and correcting these errors early in the development life cycle is particularly valuable, as it avoids the rework that would be introduced if the errors were instead discovered during integration or system test. The results so far suggests that verification of the properties that must be maintained between interacting mode transition diagrams are particularly valuable at finding errors.

There are several directions for further work. Obviously, the remainder of the FCS 5000 mode logic needs to be verified and this activity will continue in the following months.

The models that we are checking consist solely of Boolean and enumerated types in order to keep the reachable state space smaller than 10^{150} states. While every application we have looked at has large blocks of logic that meet these constraints (or can be manually abstracted to meet these constraints), many other domains intrinsically deal with integers or real numbers. Such *infinite state* systems cannot be analyzed with our existing tools. However, recent advances in model checkers that combine bounded model checking algorithms with decision procedures for integers and real numbers can deal with such infinite state system.[25,31]We would like to extend our translation framework to include such model checkers.

Finally, while we are pleased with the results of the FCS 5000 verification, we recognize that we are using our tools as powerful debugging tools, not for true proof of correctness. Ideally, we would like to be able to prove the correctness of properties even when our models are composed asynchronously. A possible answer to this problem is to exploit the fact that these systems exhibit bounded, rather than unbounded, asynchrony. That is, the clock of each model is known to run at specific rate and with a bounded drift. In theory, this information could be used to constrain the reachable state space to a manageable size. We are currently exploring approaches to this and are encouraged by the results so far.

Acknowledgments

The authors wish to thank Ricky Butler, Kelly Hayhurst, Celeste Bellcastro, and Carrie Walker of the NASA Langley Research Center, and Eric Danielson of Rockwell Collins Inc, for their ongoing support of this work. This work was supported in part by the NASA Langley Research Center under contract NCC-01001 of the Aviation Safety and Security Program.

References

[1] Berry, G. and Gonthier G., “The Synchronous Programming Language Esterel: Design, Semantics, and Implementation”, *Science of Computer Programming*, Vol. 19, 1992, pp. 87-152

[2]Benveniste A., Caspi P., Edwards S., Halbwachs N., Le Guernic P., and de Simone R., “The Synchronous Languages 12 Years Later”, *Proceedings of the IEEE*, Vol. 91, Issue 1, Jan 2003

[3]Heitmeyer C., Jeffords R., and Labaw B., “Automated Consistency Checking of Requirements Specification”, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 5, No. 3, July 1996, pp. 231-261

[4]Thompson J., Heimdahl M., and Miller S., “Specification Based Prototyping for Embedded Systems”, *Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, LNCS 1687, Sep. 1999

[5]Esterel Technologies, http://www.esterel-technologies.com

[6]Dabney J., and Harmon T., *Mastering Simulink*, Pearson Prentice Hall, Upper Saddle River NJ, 2004

[7]Bozzano M., Cavallo A., Cifaldi M., Valacca L., and Villafiorita A., “Improving Safety Assessment of Complex Systems : An Industrial Case Study”, *Proceedings of Formal Methods 2003 (LNCS 2805)*, Springer-Verlag, 2003, pp. 208-222

[8]Bozzano M. and Villafiorita A., “Improving System Reliability via Model Checking : the FSAP / NuSMV-SA Safety Analysis Platform”, *Proceedings of SAFECOMP 2003*, Sep. 23-26 2003, pp. 49-62

[9]Butler R., Miller S., Potts J., and Carreno V., “A Formal Methods Approach to the Analysis of Mode Confusion”, *Proceedings of the 17th AIAA/IEEE Digital Avionics Systems Conference*, Oct. 1998

[10] FSAP/NuSMV-SA, http://sra.itc.it/tools/FSAP

[11] Joshi A., Miller S., and Heimdahl M., “Mode Confusion Analysis of a Flight Guidance System Using Formal Methods”, *Proceedings of the 22nd Digital Avionics Systems Conference (DASC’03)*, Oct. 12-16 2003

[12] Miller S., Heimdahl M., and Tribble A., “Proving the Shalls”, *Proceedings of FM 2003 : the 12th International FME Symposium*, Sept. 8-14 2003

[13] Tribble A., Lempia D., and Miller S., “Software Safety Analysis of a Flight Guidance System”, *Proceedings of the 21st Digital Avionics Systems Conference (DASC’02)*, Oct. 27-31 2002

[14] Tribble A., Miller S., and Lempia D., “Software Safety Analysis of a Flight Guidance System” NASA/CR-2004-213004, March 2004

[15] Miller S., Tribble A., Carlson T., and Danielson E., “Flight Guidance System Requirements Specification”, NASA/CR-2003-212426, June 2003

[16] The Mathworks, http://www.mathworks.com

[17] RTCA, “Software Considerations in Airborne Systems and Equipment Certification”, RTCA/DO-178B, Washington, DC, December 1, 1992.

[18] Reactive Systems, Inc. http://www.reactive-systems.com

[19] Clarke E., Grumberg O., and Peled P., *Model Checking*, The MIT Press, Cambridge, Massachussetts, 2001

[20] IRST, http://nusmv.irst.itc.it/, The NuSMV Model Checker, Trento Italy

[21] Emerson E., "Temporal and Modal Logic", *Handbook of Theoretical Computer Science*, *Volume B: Formal Models and Semantics*, J. Van Leeuwen ed., North-Holland Pub. Co./MIT Press, 1990, pp. 995-1072

[22] Owre S., Rushby J., Shankar N., and Henke F., "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS", *IEEE Transactions on Software Engineering*, Feb. 1995, Vol. 21, No. 2, pp. 107-125

[23] Anonymous, PVS Home Page, http://www.csl.sri.com/projects/pvs/

[24] Bensalem S., et. Al, "An Overview of SAL", *Proceedings of LFM 2000: Fifth NASA Langley Formal Methods Workshop,* June 2000, pp. 187-196

[25] Anonymous, SAL Home Page, http://www.csl.sri.com/projects/sal/

[26] Harel D., "Statecharts : A Visual Formalism for Complex Systems", *Science of Computer Programming,* Vol 8., No. 3, June 1987, pp. 231-274

[27] Leveson N., Heimdahl M, Hildreth H., and Reese J., "Requirements Specifications for Process-Control Systems", *IEEE Transactions on Software Engineering*, Vol. 20, No. 9, Sept 1994, pp. 684-707

[28] Faulk S., Brackett J., Ward P., and Kirby J., "The CoRE Method for Real-Time Requirements,", *IEEE Software*, Vol 9. No. 5, Sept. 1992, pp. 22-33

[29] Halbwachs N., and Baghdadi S., "Synchronous Modeling of Asynchronous Systems", *Proceedings of EMSOFT'02*, LNCS 2491, Springer-Verlag, Grenoble, Oct. 2002

[30] Miller S., Whalen M., O'Brien D., Heimdahl M., and Joshi A., "A Methodology for the Design and Verification of Globally Asynchronous/Locally Synchronous Architectures", to be published as a *NASA Contractor Report.*

[31] Prover Technology, http://www.prover.com