Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Proving the Shalls

Early Validation of Requirements Through Formal Methods

Steven P. Miller[1], **Alan C. Tribble**[1], **Michael W. Whalen**[1], **and Mats P.E. Heimdahl**[2]

[1]Rockwell Collins Inc, 400 Collins Road NE,Cedar Rapids, Iowa, 52498, USA,
e-mail: spmiller@rockwellcollins.com, phone: (319) 295-2055, fax: (319) 295-2005

[2]Department of Computer Science and Engineering, University of Minnesota, 4-192 EE/CSci Bldg, 200 Union Street S.E., Minneapolis, Minnesota, 55455, USA

Received: date / Revised version: date

Abstract. Incomplete, inaccurate, ambiguous, and volatile requirements have plagued the software industry since its inception. The convergence of model-based development and formal methods offers developers of safety-critical systems a powerful new approach for the early validation of requirements. This paper describes a case study conducted to determine if formal methods could be used to validate system requirements early in the lifecycle at reasonable cost. Several hundred functional and safety requirements for the mode logic of a typical Flight Guidance System were captured as natural language "shall" statements. A formal model of the mode logic was written in the RSML^{-e} language and translated into the NuSMV model checker and the PVS theorem prover using translators developed as part of the project. Each "shall" statement was manually translated into a NuSMV or PVS property and proven using these tools. Numerous errors were found in both the original requirements and the RSML^{-e} model. This demonstrates that formal models can be written for realistic systems and that formal analysis tools have matured to the point where they can be effectively used to find errors before implementation.

Key words: Software requirements, formal verification, model-based development

1 Introduction

Incomplete, inaccurate, ambiguous, and volatile requirements have plagued the software industry since its inception. In a 1987 article [10], Fred Brooks wrote

The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements... No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.

Studies have shown that the majority of software errors are made during requirements analysis, and that most of these errors are not found until the later phases of a project. Other studies have shown that the cost of correcting a requirements error grows dramatically the later in the product life cycle it is corrected [9,16,50,39]. Researchers have also found that requirements errors are more likely to affect the safety of a system than errors introduced during design or implementation [25,30].

The avionics industry has long recognized the need for better requirements, and has spearheaded the development of several methodologies for requirements specification. The Software Cost Reduction (SCR) methodology [23,22] was originally developed to specify the requirements for the A-7 aircraft [38]. It was later extended to the CoRE methodology by the Software Productivity Consortium [19] and used to specify the avionics requirements on the Lockheed C-130J [20]. The Requirements State Machine Language (RSML) notation was developed to specify the requirements for TCAS-II, a collision avoidance system installed on all commercial aircraft seating more than 30 passengers [27]. Even Statecharts [21], whose various derivatives make up one of the most widely accepted modeling notations in use today, has its roots in the avionics industry.

Despite this legacy, the requirements for most avionics systems are still specified using a combination of natural language and informal diagrams. In fact, in some ways, these efforts have actually increased the confusion about what requirements are and how they should be stated. Should requirements be captured as a list of

[1] *This project was partially funded by the NASA Langley Research Center under contract NCC1-01001 of the Aviation Safety Program*

"shall" statements written in a natural language? Or should requirements be expressed as mathematical models defining the relationship between the inputs and outputs as is done in SCR, CoRE, and RSML? Can the requirements of a system be completely stated with use cases? When does one cross the line between requirements analysis and design, and why does that matter?

This paper describes a case study conducted by the Advanced Technology Center of Rockwell Collins, the Critical Systems Research Group at the University of Minnesota, and the NASA Langley Research Center to determine how far formal analysis could be pushed in an industrial example. In this exercise, a model of the mode logic of a typical Flight Guidance System was specified in the Requirements State Machine Language without Events (RSML^{-e}) notation developed by the Critical Systems group at the University of Minnesota. Translators were developed from RSML^{-e} to the NuSMV model checker and the PVS theorem prover. These tools were then used to verify several hundred properties of the RSML^{-e} model. In the process, several errors were discovered and corrected in the original RSML^{-e} model.

The results of this exercise are significant for several reasons. While the model of the FGS does not describe a fielded product, it was specifically created to be representative in size and complexity of an typical system and has been used to determine if several proposed methods or tools would scale to industrial use. As a result, the success of this exercise demonstrates that formal models can be written for real problems using notations acceptable to practicing engineers, and that formal analysis tools have matured to the point where they can be efficiently used to find errors before implementation.

In previous studies conducted by the authors using this example, only limited formal analysis was done on the model, or one model was used for specification and simulation while another model was created by hand for formal verification. For example, [31] describes the authors' experiences modeling the mode logic informally using the CoRE methodology [19] and the benefits that were gained from entering this model into the SCR* tool and using the consistency and completeness checks provided by SCR* [23,22]. In [11], a portion of the mode logic was modeled by hand in PVS and several properties were proven using the PVS theorem prover. In contrast, in this effort, the same model was used for specification, review, and simulation, and automatically translated into other notations for formal verification. In addition, all of the functional and safety requirements were formally verified in a clearly cost-effective manner. Perhaps just as important, this study clarifies the relationship between requirements stated informally as shall statements, formal properties stated in notations such as predicate calculus and temporal logic, and requirements models written in notations such as RSML, SCR, or Statecharts. This is discussed further in Section 4.

2 Background Information

This section describes the role of a Flight Guidance System in a modern aircraft and provides a brief overview of the RSML^{-e} notation, the NIMBUS toolset, the NuSMV model checker, and the PVS theorem proving system.

2.1 Overview of a Flight Guidance System

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 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 (attempting to lock on to a navigation source) and active (providing guidance to the aircraft) at any given time.

The overall FGS system consists of two identical subsystems, one asssociated with the left side of the aircraft and one with the right side. In most modes of operation, only one side is active and responds to pilot inputs and produces outputs. The inactive side simply copies its internal state from the active side, serving as a hot backup. In a few critical modes such as Approach and Go Around, both sides of the FGS are active and generate outputs that are compared before they are used.

We have used the mode logic of a FGS as an example in several previous studies [11,24,49]. It is an excellent example because it is complex and representative of a class of problems frequently encountered in the design of embedded control systems.

2.2 The RSML^{-e} Specification Language

For this exercise, we specified the FGS mode logic using the Requirements State Machine Language without Events (RSML^{-e}) notation. A number of different specification languages were considered at the start of the project. The main criteria used in selecting a language included an emphasis on the specification of requirements as opposed to design, a precise formal semantics, a clear path to integration with formal verification tools, and the likelihood of future commerical tool support. The emphasis on requirements specification and formal semantics reduced the field to RSML and SCR, both of which were expressly designed for the modeling of system requirements. Of the two, the ongoing of development of RSML into a commercial tool, SpecTRM-RL, by the Safeware Engineering Corporation, was a key factor in the choice of the closely related RSML^{-e} .

RSML^{-e} is based on the Requirements State Machine Language (RSML) developed by Nancy Leveson's group at the University of California at Irvine as a language for specifying the behavior of process control systems [27]. One of the main design goals of RSML was readability and understandability by non-computer professionals such as end-users, engineers in the application domain, managers, and representatives from regulatory agencies. RSML was used to specify TCAS-II and this specification was ultimately adopted by the FAA as the official specification for TCAS-II [26].

RSML was in turn heavily influenced by Statecharts [21] and uses a similar notion of explicit event propagation. In the course of developing the TCAS-II specification and its independent verification and validation experiment, it became clear that the most common source of errors was this dependence on explicit events [28]. To reduce this problem, the Critical Systems group at the University of Minnesota developed the Requirements State Machine Language without Events (RSML^{-e}) [47,48]. As its name implies, RSML^{-e} eliminates the use of explicit events and is a fully formal synchronous language [7]. RSML^{-e} is similar to another derivative of RSML, SpecTRM-RL, developed by Safeware Engineering Corporation, but has a slightly different syntax and semantics and a different underlying philosophy of how the language should be used in modeling. The full specification of the FGS Mode Logic in RSML^{-e} can be obtained at [34].

An RSML^{-e} specification consists of a collection of input variables, state variables, input/output interfaces, functions, macros, and constants; input variables are used to record the values observed in the environment, state variables are organized in a hierarchical fashion and are used to model various states of the control model, interfaces act as communication gateways to the external environment, and functions and macros encapsulate computations providing increased readability and ease of use.

Figure 1 shows the specification of two macros, Select_ROLL and Deselect_ROLL, and the specification of the ROLL state variable in the RSML^{-e} specification of the Flight Guidance System.

Macros are simply named predicates used to improve the the readability and maintainability of the specification. Conditions in the macro definitions are represented in the AND/OR table format developed for the original RSML notation to make large predicates easy for pilots and other reviewers to read. Each column of truth values represents a conjunction of the propositions in the leftmost column (a "*" represents a "don't care" condition). If a table contains several columns, we take the disjunction of the columns; thus, the table is a way of expressing conditions in a disjunctive normal form. For example, the Select_ROLL macro returns the value (Is_No_Nonbasic_Lateral_Mode_Active() AND Modes = On) while the Deslect_ROLL macro returns the value (When_Nonbasic_Lateral_Mode_Activated() OR Modes = Off).

The state variable ROLL is the default lateral mode in the FGS mode logic, and is declared as a child state of the variable Modes. The ROLL state variable only has meaning when Modes has the value On. This notion of hierarchical variables provides the same abstractions and structuring mechanism as the AND and OR states in Statecharts, but the semantics are simpler [51].

The conditions under which the state variable changes value are defined by either direct assignments or by TRANSITION clauses. For example, if this side of the FGS is not the active side, ROLL will take on the value Offside_ROLL provided from the other side of the FGS. If this side of the FGS is the active side, its value is determined by the four transition clauses. The first two transitions in the definition of the ROLL variable define what value ROLL is to take on when it becomes defined, i.e., when its parent variable Modes takes on the value On. If Select_ROLL is false, ROLL will take on the value Cleared. If Select_ROLL is true, ROLL takes on the value Selected. The remaining transitions describe how ROLL changes back and forth from Cleared to Selected.

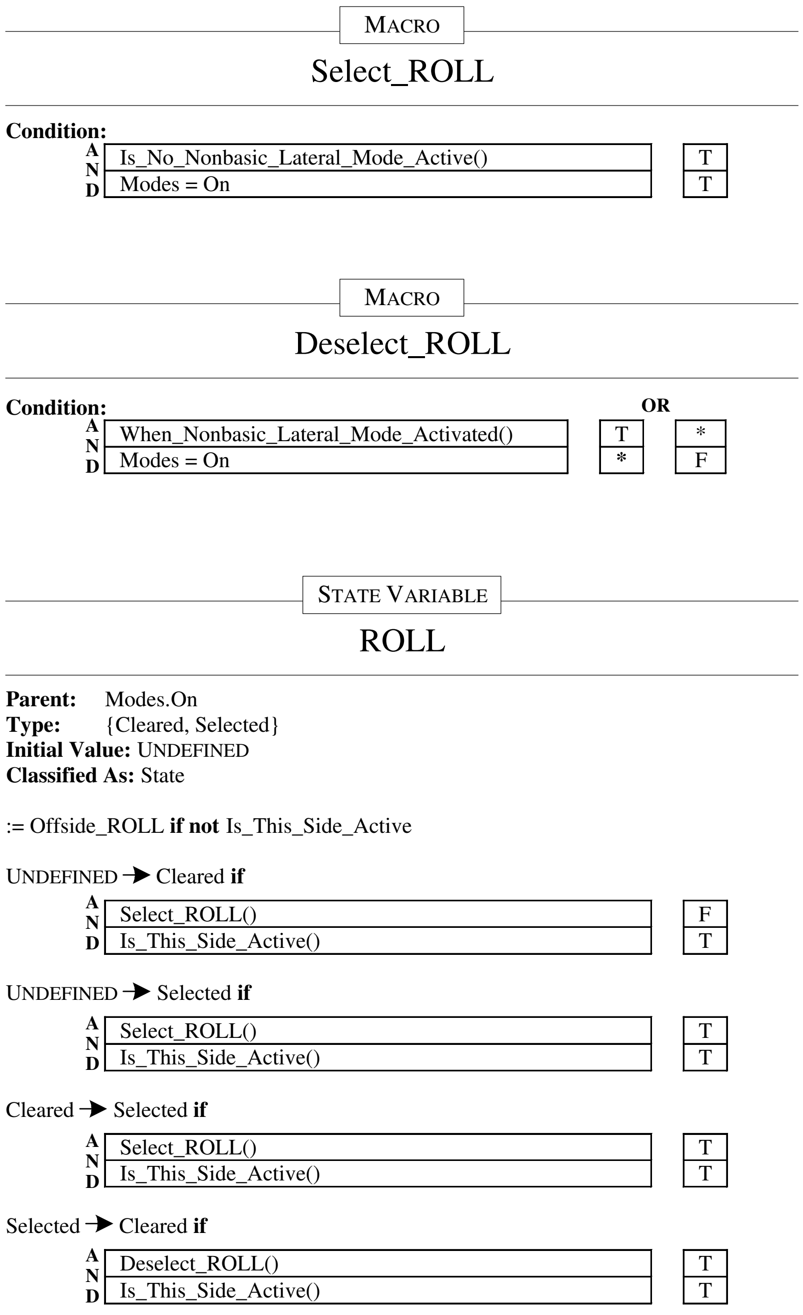
Often one needs to refer to values of the variables at a certain point in the variable's history. RSML^{-e} provides a construct for doing this, as shown in the following example.

```
MACRO Were_Modes_Off() :
        PREV_STEP(Modes) = Off
END MACRO
```

In the above example, PREV_STEP(Modes) refers to the previous value of the state variable Modes.

RSML^{-e} transitions are purely condition-based and free of internal events. As soon as the guards in a variable definition can be evaluated, it will take on its new value. The variables are partially ordered based on the data dependency induced by the guard conditions. Similar semantics are adopted in the programming language Lustre [6]. Data-flow semantics removes complex issues caused by internal events, such as infinite triggering events (akin to infinite loops in a programming language) or analysis of micro-steps [12], from the language.

Startup behavior and behavior in the face of sensor failures pose particular challenges when specifying control systems. Under these circumstances we simply do not know what the state of the environment might be. RSML^{-e} supports modeling of this uncertainty by providing the concept of "undefinedness". One can explicitly specify the initial value of variables at startup to be UNDEFINED, such as ROLL = UNDEFINED in Figure 1. Also, when a parent variable takes on a new



MACRO

Select_ROLL

Condition:

AND		
	Is_No_Nonbasic_Lateral_Mode_Active()	T
	Modes = On	T

MACRO

Deselect_ROLL

Condition:

AND		OR	
	When_Nonbasic_Lateral_Mode_Activated()	T	*
	Modes = On	*	F

STATE VARIABLE

ROLL

Parent: Modes.On
Type: {Cleared, Selected}
Initial Value: UNDEFINED
Classified As: State

:= Offside_ROLL **if not** Is_This_Side_Active

UNDEFINED ➔ Cleared **if**

AND		
	Select_ROLL()	F
	Is_This_Side_Active()	T

UNDEFINED ➔ Selected **if**

AND		
	Select_ROLL()	T
	Is_This_Side_Active()	T

Cleared ➔ Selected **if**

AND		
	Select_ROLL()	T
	Is_This_Side_Active()	T

Selected ➔ Cleared **if**

AND		
	Deselect_ROLL()	T
	Is_This_Side_Active()	T

Fig. 1. A Fragment of the FGS Specification in RSML^{-e}

value, each child variable of the parent value that was just changed is no longer relevant and must not be used. RSML^{-e} handles this by implicitly assigning these variables the value UNDEFINED.

2.3 The NuSMV Model Checker

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 reimplementation and extension of SMV [15], 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 testbed for formal verification techniques, and applied to other research areas [3]. Properties to be verified in NuSMV are specified using either Computation Tree Logic (CTL) or Linear Time logic (LTL) [15]. Since RSML^{-e} is a synchronous specification language and the FGS mode logic can be completely specified using only Boolean and enumerated types, we believed that a symbolic model checker such as NuSMV would be well suited for analysis of the FGS mode logic.

2.4 The PVS Theorem Prover

PVS is an environment for specification and verification 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 a highly effective 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 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 [37,4]. Since successful use of a theorem prover requires considerable expertise, our familiarity with the PVS environment was a key factor in its selection.

2.5 The NIMBUS *Environment*

NIMBUS is a requirements engineering environment for RSML^{-e} models built by the Critical Systems Research Group at the University of Minnesota. The capabilities of NIMBUS, shown in Figure 2, support a variant of model-based development we call *specification-based prototyping* [47,48] that allows an engineer to evolve a formal requirements model from a prototype to a production system through the use of simulation, inspection, static analysis, and code generation.

Specification-based prototyping combines the advantages of traditional formal specifications (e.g., preciseness and analyzability) with the advantages of rapid prototyping (e.g., risk management and early end-user involvement). The approach lets an engineer refine a formal executable model of the system requirements to a detailed model of the software requirements. Throughout this refinement process, the specification is used as an early prototype of the proposed software. By using the specification as the prototype, most of the problems that plague traditional code-based prototyping disappear. First, the formal specification will always be consistent with the behavior of the prototype (excluding real-time response) and the specification is, by definition, updated as the prototype evolves. Second, the risk of evolving the prototype into a poorly designed production system is largely eliminated.

NIMBUS supports rich simulation capabilities, that allow an engineer to simulate an RSML^{-e} specification while interacting with (1) user input or text file input scripts, (2) RSML^{-e} models of the components in the environment, (3) software simulations of the components, or (4) the physical components themselves (i.e., hardware-in-the-loop simulations). It is possible to run the simulator in real-time or step-time to allow closer examination of particular scenarios. The simulation capabilities of the NIMBUS environment are designed to make it easy to create simple simulations using input files or user input early in the project lifecycle that can be refined to accurate simulations with complex environmental models as the project progresses.

NIMBUS also contains a flexible automated translation framework, allowing RSML^{-e} specifications and properties to be automatically translated to a wide range of formal analysis tools, including theorem provers (PVS [4]), model checkers (NuSMV [3] and the SRI SAL Toolset [17]), and random search tools (Lurch [35]). It is important to target multiple analysis tools for two reasons: first, no single tool meets all of our analysis needs, and each tool is best suited to certain specification and property styles; second, it allows multiple independent verification paths for analysis, lessening the chance that a bug in a tool will affect the correctness of our analysis process. The translation framework allows us to quickly target an array of analysis tools with minimum effort.

Using this framework, models are developed in tandem with formal requirements and safety properties that are expected to hold over the model. We then translate the models and properties to a particular back-end tool, which determines whether or not the property holds on the specification. These tools, with the exception of Lurch, can provide a guarantee that the property of interest holds on the specification, which is a much stronger claim than is possible with testing. These kinds of analysis are integrated into the specification-based prototyping approach: many critical safety properties can be checked even on early iterations of models.

Finally, NIMBUS supports a sophisticated document generation capability that is suitable for generating high-quality, cross-referenced (and hyperlinked) documents that are suitable for formal inspections [18,2] of RSML^{-e} models. This capability was used, with no additional hand-tuning, to create the final NASA contractor report for the Flight Guidance System [34].

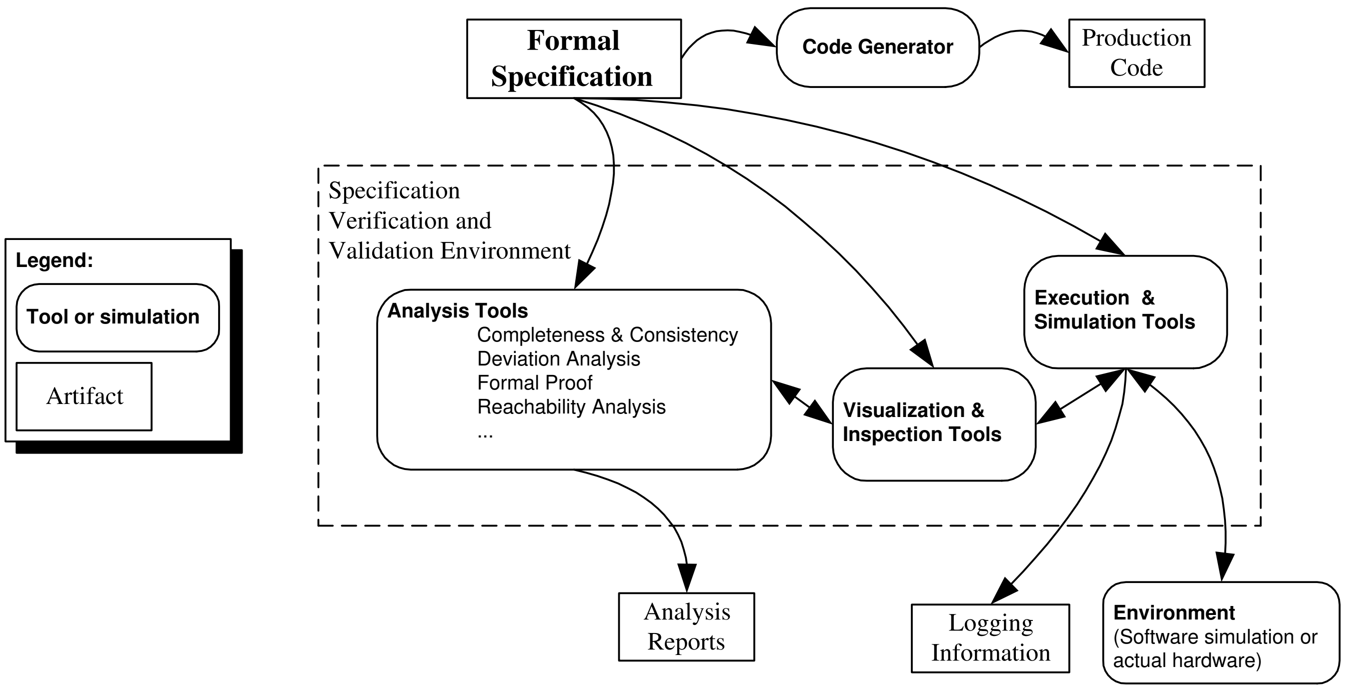


Fig. 2. The NIMBUS Toolset

2.5.1 Translation to Analysis Tools

The NIMBUS toolset translates to a wide variety of analysis tools. Although there is a great deal of overlap in terms of what the tools are theoretically able to verify, they vary widely in terms of how they are used and how much guidance must be provided by the engineer. Thus, the requirements for creating a “good” translation vary depending on the capabilities of the target tool.

When translating to a completely automated tool, such as a model-checker or random search tool, we want to transform the specification to minimize the state space of the specification. Human readability of the resultant model-checker code is not that important for the actual task of verifying the property of interest. Thus, when translating to the model-checker, we support options to build in conservative abstractions that make model-checking feasible by sacrificing some accuracy and expressiveness of the original RSML^{-e} specification.

On the other hand, theorem-proving with PVS is an interactive process. In this case, readability of the translated output and maintaining a close correspondence with the source specification are of prime importance. Further, since PVS supports more expressive logics and reasoning procedures, there is not as great a need to abstract away details in the source specification. Thus, the requirements for the translation were that it should fully capture the semantics of the source language in an elegant way, producing readable PVS specifications. Below we provide a brief overview of the translation approaches. The reader interested in the details is referred to reports dedicated to the translation details [13,40].

2.5.2 NuSMV Translation

The model checking project was designed with the following goals: (1) the translation must to the largest extent possible preserve the full power of the source language RSML^{-e}, (2) the translation must require little-or-no user interaction, (3) counterexamples must be easy to interpret without detailed knowledge of the translation process, and, finally, (4) any abstractions must be fully automated. Although much work remains to be done—especially with respect to abstraction—the model checking translation has been used very successfully on several large case examples. An example of a portion of the translation to SMV is shown in Figure 3.

Our translation scheme from RSML^{-e} to NuSMV is described in [13] and is similar to the one selected in [12]. As mentioned earlier, the basic constructs in RSML^{-e} are variables and next-state relations for variables. Disregarding the complicating factor of UNDEFINED values, RSML^{-e} variables share a one-to-one correspondence with NuSMV variables. The assignment relations for RSML^{-e} variables can be immediately translated to SMV variable assignments: each RSML^{-e} transition becomes an assignment. To describe the situation in which no transition occurs (so the variable value remains the same), we add a default case to the SMV assignment that assigns the variable the value it had in the previous step. The remaining RSML^{-e} constructs (macros, functions,

```
MODULE Select_ROLL(Modes,m_Is_No_Nonbasic_Lateral_Mode_Active)
VAR result : boolean;

ASSIGN
   init(result):=0 ;
   next(result):=
      next(m_Is_No_Nonbasic_Lateral_Mode_Active.result) & next(Modes)=On ;


MODULE Deselect_ROLL(Modes,m_When_Nonbasic_Lateral_Mode_Activated)
VAR
 result : boolean;

ASSIGN
   init(result):=0 ;
   next(result):=
      next(m_When_Nonbasic_Lateral_Mode_Activated.result) | (!(next(Modes)=On)) ;


...

VAR
   ROLL: {Cleared,Selected,Un_defined } ;

ASSIGN
   init(ROLL):=  Un_defined ;
   next(ROLL):=
      case
         !(next(Modes)= On)               :  Un_defined  ;

         !(next(Is_This_Side_Active)=1)   :  next(Offside_ROLL) ;

         !(next(m_Select_ROLL.result)) &
         next(Is_This_Side_Active)=1 &
         ROLL= Un_defined                 :  Cleared ;

         next(m_Select_ROLL.result) &
         next(Is_This_Side_Active)=1 &
         ROLL= Un_defined                 :  Selected ;

         next(m_Select_ROLL.result) &
         next(Is_This_Side_Active)=1 &
         ROLL=Cleared                     :  Selected ;

         next(m_Deselect_ROLL.result) &
         next(Is_This_Side_Active)=1 &
         ROLL=Selected                    :  Cleared ;

         TRUE                             :  ROLL ;
      esac;
```

Fig. 3. A Fragment of an RSML^{-e} Specification Translated to SMV

and interfaces) can all be straightforwardly mapped to SMV modules.

Undefinedness can be simulated in NuSMV by adding additional state. Depending on the type of the variable in question, we simulate undefinedness in one of two ways. For Boolean and enumerated types, we simply extend the range of the type with an additional value, UNDEFINED, that represents the value of the variable when it is in an undefined state. For numeric quantities, we add an additional Boolean variable that represents the undefined-status of the variable. We then use this additional state to determine whether or not a variable is UNDEFINED.

Figure 3 shows the SMV translation corresponding to the RSML^{-e} fragment in Figure 1. In this fragment, the Select_ROLL and Deselect_ROLL macros are implemented as SMV Modules. These modules are very similar to the RSML^{-e} macros, but require additional parameters. These parameters are necessary because SMV does not support nested module definitions, so any variables and macros that are used within the body of the macro must be passed as parameters.

The translation for the ROLL state variable is slightly more involved. First, we create a SMV variable with the same name. To support UNDEFINED values, we extend the type of this SMV variable with `Un_defined`. Next, the transitions for ROLL are rewritten into a set of SMV cases. As described in Section 2.2, if the *Modes* state variable is not in state *On*, then the value of ROLL is UNDEFINED. This behavior is handled by the first case in the SMV assignment for variable ROLL. The next five cases correspond to the conditions and transitions of the ROLL state variable. Finally, if none of the transitions apply, we want the value of ROLL to be left unchanged. This behavior is implemented by the final (TRUE) case in the SMV assignment.

One aspect of the translation that is of theoretical interest involves domain abstraction on numeric variables. If RSML^{-e} specifications contain numeric variables over large domains, then a naïve translation will yield an intractable model-checking problem. In [14], we describe a theoretically sound domain abstraction technique for model checking software systems that reduces the state space of specifications with large-domain variables. This abstraction technique is integrated into the Nimbus NuSMV translator, so that counterexamples on the abstracted specification can be understood on the full specification. It is a conservative technique, so certain classes of properties may yield spurious counterexamples on the abstract model but not the full model. However, it is accurate enough to prove many properties on models that are otherwise intractable.

2.5.3 PVS Translation

When starting the project, we considered two competing approaches for representing RSML^{-e} specifications in PVS. The first approach is to view the *state-space as a cross product of the domains* of system variables. The specification is viewed as a collection of constraints determining the set of possible initial states and the set of possible transitions between states. Then, the transitive closure of the initial states under the transition relation constitutes the reachable state-space of the system. The system will satisfy a certain property of interest if it can be established that every reachable state satisfies this property. This view is usually adopted when one is verifying state-based specifications using model-checkers like SMV or the μ-calculus model-checker of PVS. Owre *et al.* [36] discuss such an approach to translate requirement specifications written in SCR to PVS.

The second approach is to consider *state as a point of observation of certain quantities of interest* in the system. The system variables represent quantities of interest, i.e., they are mappings from states (the observation points) to values of those quantities (at those observation points). When the system responds to changes in its environment, it moves to a new observation point, i.e., to a new state. So each state has an associated (finite) *history* or *stream* of observations up to that point. In this view, the system specification is a set of constraints on the histories of observations at each state. If we think of constraints as Boolean valued quantities constructed using system variables, then the specification lists a set of such quantities that are given to be true in every state. Properties of interest that one wants to prove are also similar to constraints but one has to establish that these are true. Bensalem *et al.* [6] adopt such an approach for proving properties of control system specified in Lustre using PVS.

In an earlier version of our translation we adopted the former approach to translate RSML^{-e} specifications to PVS. However, we found that it was difficult to construct proofs of properties in PVS for large systems using such an approach. Part of the difficulty arose from the fact that one had to carry around the complete state construct in proofs, even though much of the reasoning and proof steps involved only a few variables at any given time. Also, the translated output was quite difficult to comprehend. The latter approach, which we adopted subsequently, overcomes these shortcomings.

Our current formalization of RSML^{-e} in PVS is built the second approach. Verification in this context is checking whether the set of possible histories as constrained by the specification satisfy a given predicate. This formalization has the advantage of retaining both the structure and the semantics of the RSML^{-e} source specification in the translated PVS output. Also, inductive streams are easy to formalize in PVS, allowing for a fairly straightforward reasoning process when constructing proofs.

3 The Requirements Analysis Process

In this section we describe the process we followed in eliciting, modeling, and analyzing the requirements of the FGS mode logic. This process is depicted in Figure 4.

In the first phase of elicitation, we collected the system requirements as informal “shall” statements. The next phase modeling, consisted of constructing by hand an executable RSML^{-e}model that we believed exhibited the behavior informally stated in the shall statements. Throughout creation of the model, we continually used the simulation capabilities of the Nimbus environment execute the model and informally confirm that it behaved as we expected. In the formal verification phase, we manually translated the shall statements into formal properties stated over the model in either CTL or PVS, and merged these formal properties with the appropriate translation of the RSML^{-e}model into NuSMV or PVS created using the translators developed by the University of Minnesota. The NuSMV model checker or the PVS theorem prover was then used to confirm whether the property held over the model or not. If a property was found to not hold, the necessary changes were made to either the model or property. At each stage of this process, corrections were fed back into the previous stage, ultimately resulting in greatly improved set of requirements. This process is described in greater detail in the following subsections.

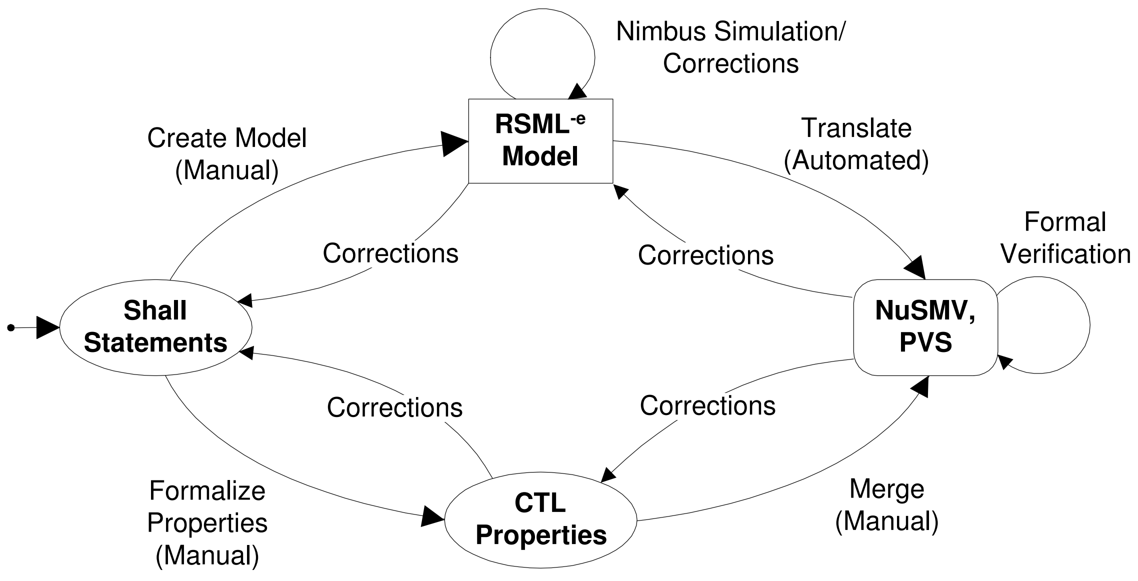


Fig. 4. Overview of Requirements Analysis Process

3.1 Requirements Elicitation

As in most projects, one of our first tasks was to develop an informal understanding of what the system was to do. A variety of techniques have been advocated for eliciting requirements, ranging from the traditional listing of shall statements to writing a concepts of operation document to the development of use cases. Since we were interested in injecting formal modeling into existing practices, we chose to start with the lowest common denominator, simply capturing the requirements as informal shall statements stored in a DOORS [1] database. Examples of a few such requirements for the FGS mode logic are shown in the left hand column of Table 1 on page 13.

3.2 Requirements Modeling

Our next step was to create a formal statement of the black box behavior of the system. We were guided in this by a methodology developed at Rockwell Collins that was heavily based on the CoRE methodology developed by the Software Productivity Consortium [19], which is in turn based on the SCR methodology [23,22,38].

This model was written in the RSML^{-e} language. One of the great advantages of executable specification languages such as RSML^{-e} or SCR is that they can be connected to a mockup of their environment, provided inputs, and their behavior studied. This provides a very easy way for the developer to get immediate feedback about the model being created. We used this approach to continuously review the model under construction. Using this capability and simplified models of the environment, we were able to simulate a large portion of the flight deck, including two FGS systems, primary flight displays, and the flight control panel, as shown in Figure 5.

When completed, the RSML^{-e} model of the FGS mode logic consisted of 41 input variables, 16 small, tightly synchronized hierarchical finite state machines, 122 macro or function definitions, 29 output values, and was roughly 160 pages long. A detailed description of the model and its simulation environment is available in [34].

In the course of building the RSML^{-e} model, we often found ourselves going back and modifying the original shall statements as shown in Figure 4 on page 9. Sometimes, they were just wrong. More often, their organization needed to be changed to provide clear traceability to the model. For example, in the original statement of the requirements, the conditions under which the mode annunciations and the flight director guidance cues would be turned on were combined in several shall statements. We found that the requirements were clearer if we broke these out as distinct groups of statements. Gradually, we realized that the revised shall statements were a clearer and improved description of the system. Maintaining even a coarse mapping between the shall statements and the RSML^{-e} model forced us to be more precise in writing down the shall statements.

3.3 Model Checking

As the model neared completion, the University of Minnesota team completed the first RSML^{-e} to NuSMV translator. This translator, described in [13], automati-

[1] DOORS (Dynamic Object Oriented Requirements System) by Telelogic is a commercial requirements management tool.

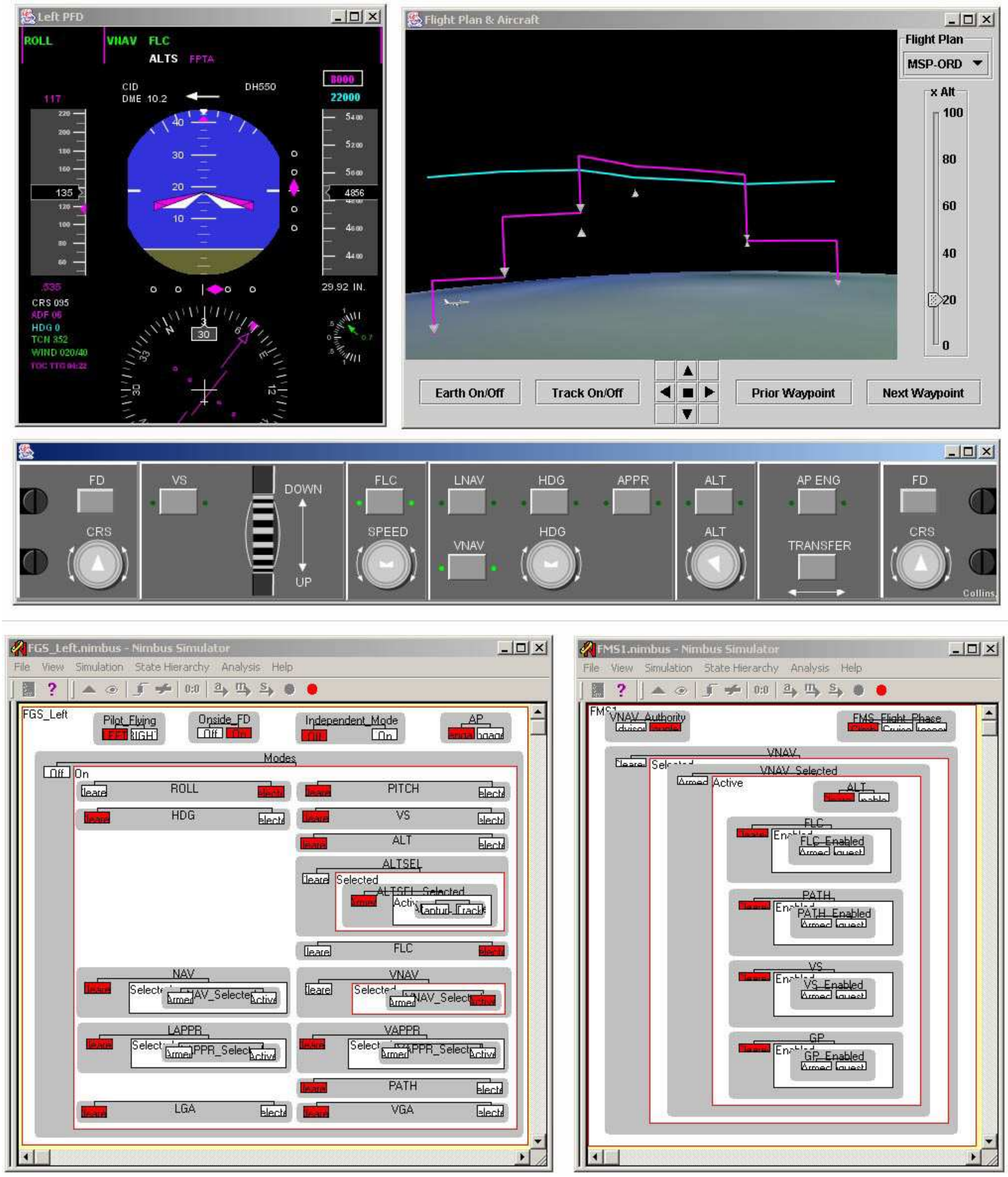


Fig. 5. FGS Simulation using Nimbus.

cally converted the RSML^{-e} model to the specification language of the NuSMV model checker. While the automated translation of the model to NuSMV is shown as a single dataflow in Figure 4 on page 9, automating this process was a key step in the entire process. Automating this process greatly increased our confidence that properties proven over the translated model would also hold over the original RSML^{-e}model.

There were a few issues that had to be dealt with for model checking to be feasible. We knew state space explosion would be a problem since we had included in the RSML^{-e} model a few integer input variables, such as the aircraft's altitude, and a few comparisons that depended on time. The state space explosion resulting from these few variables was indeed enough to make the verification of most properties infeasible using the earliest translators. While the University of Minnesota team was planning to develop algorithms that would reduce the size of the translated model through a variety of abstraction techniques, these extensions were not yet ready.

Fortunately, algorithms to deal with the time dependencies proved straightforward and were quickly implemented in the translator. Details of these algorithms can be found in [13]. To deal with the few remaining integer variables, we abstracted the model by hand by replacing comparisons involving these variables (e.g., $Altitude > PreSelectAlt + AltCapBias$) with Boolean inputs representing the result of the comparison (e.g., Altitude_Gt_PreSelect_Plus_AltCapBias), eliminating all integers from the model. [2] Since there were only a few such computations, this took only a few hours to implement and did not significantly alter the specification. These changes reduced the state space of the model enough that we could check almost any property of the mode logic with the NuSMV model checker in a matter of minutes.

It was feasible to make these abstractions manually in this particular case because they were so few and so straightforward. While we have been able to use this same technique in several other examples, in some domains the number of abstractions that would be needed would be too large to do reliably by hand. Ideally, these abstractions would be identified and made automatically during the translation process. As mentioned earlier, work is underway to add these capabilities to the RSML^{-e} to NuSMV translator [14].

At first, we focused on showing that our model satisfied the safety properties we had identified through a hazard analysis and fault tree analysis [49]. However, it quickly became apparent that all of the original requirements, not just the safety properties, could be stated in CTL. As a result, we extended our verification to include all the shall statements captured during elicitation.

Our approach was to state each requirement as a CTL property over the translated model. Since there was a close correspondence between names in the RSML^{-e} model and the NuSMV model, this quickly became routine and most of the requirements could be translated by hand into CTL in a few minutes. A desirable future enhancement would be the development of a property specification language in RSML^{-e} so that the translator could translate the CTL properties automatically along with the NuSMV model.

All of the requirements could be specified with only two CTL formats. The first was simply a safety constraint that had to be maintained by all reachable states. For example, the requirement

If this side is active, the mode annunciations shall be on if and only if the onside FD cues are displayed, or the offside FD cues are displayed, or the AP is engaged

was translated into the CTL property

```
AG(Is_This_Side_Active ->
     (Mode_Annunciations_On <->
      (Onside_FD_On | Offside_FD_On = TRUE |
       Is_AP_Engaged)))
```

where the AG operator states that the property must hold for all globally reachable states and the operators $->$ and $<->$ have their usual meaning of “implies” and “iff”.

Occasionally, the semantics of RSML^{-e} and CTL interacted in inelegant ways. For example the RSML^{-e} input variable Offside_FD_On in the above example could take on the values TRUE, FALSE, or UNDEFINED and had to be explicitly compared with the value TRUE in CTL. In contrast, the variable Onside_FD_On was a computed internal RSML^{-e} Boolean variable that could only take on the values TRUE and and FALSE and be used in a CTL expression without explicit comparison. While it would have been possible to adopt a single specification style by also comparing true Boolean variables to TRUE and FALSE, using two different styles was actually a useful way to emphasize in the CTL expressions themselves which variables could take on the value UNDEFINED.

The second format was a constraint over a state and all possible next states. For example, the requirement

If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged.

was translated into the CTL property

```
AG((!Onside_FD_On & !Is_AP_Engaged)->
    AX(Is_AP_Engaged -> Onside_FD_On))
```

[2] For model checking, the actual values of these expressions are immaterial as the model checker will evaluate all possible combinations. During simulation, these Boolean inputs are computed in a small model executing in parallel with the the abstracted specification. There are no complex timing relationships between these expressions, so the order and timing of their evaluation is not an issue.

where the AX operator states the enclosed property must hold for all states reachable in the next step.

Only these two formats were needed, largely because RSML^{-e} is a synchronous language in which each transition to the next system state is computed in a single atomic step. All the properties we were interested in could be stated as simple safety properties over a single state, or as a relationship describing how the system changed in a single step. If we had wanted to verify liveness properties, or if portions of the model had been allowed to evolve asynchronously, other temporal logic operators such as eventually (F), until (U), or release (R) would also have been needed [15].

Ultimately, all 281 properties originally stated informally in English were translated into CTL and checked using the NuSMV model checker. All 281 properties could be verified on a 2GHz Pentium 4 processor running Linux in less than an hour. To track the CTL properties, we modified the DOORS database to maintain both the informal and CTL versions of the requirements and to export a file that could be passed directly into the NuSMV model. This made it very easy to recheck the properties after the model was changed, though a simple "include" statement in the NuSMV language would have been very helpful. A few of the shall statements and their CTL properties are shown in Table 1.

These properties are organized by a functional decomposition of the FGS that closely reflect how the FGS requirements have traditionally been organized. First, the ways in which a function can be selected are specified, followed by the ways in which the function can be deselected, finally followed by any invariants that must be maintained during the function's operation. Functions that can only be active when a "parent" function is active are nested in a natural outline structure.

The rationale for selecting this organization was to provide a clear bridge from the traditional specification of requirements to the formal statement of the properties. Practicing engineers accept this structure very well, and are usually intrigued by the clear mapping of informal shall statements to their formal properties.

3.4 Errors Found Through Model Checking

Use of the model checker produced counter examples revealing several errors in the RSML^{-e} model of the mode logic that had not been discovered through simulation. Sometimes, this required a correction to the RSML^{-e}model as shown in Figure 4 on page 9. Other times, it was the property itself that needed to be corrected. In either case, corrections were also propagated back to the original shall statement.

For example, in trying to prove the requirement

> *If Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the Flight Control Panel.*

we discovered two ways in which this property was not true. First, if another event arrived at the same time as the HDG switch was pressed, that event could preempt the HDG switch event. Second, if this side of the FGS was not active, the HDG switch event was completely ignored by this side of the FGS. This led us to modify the requirement to state

> *If this side is active and Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the FCP (providing no higher priority event occurs at the same time).*

While longer and more difficult to read than the original statement, it has the advantage of being a more accurate description of the system's behavior. Of course, we also had to clearly define what a "higher priority"' event was.

Stating that the FGS needs to be active is a constraint well understood by the engineers and the actual value of this clarification is probably minimal. However, we also discovered several ways in which important safety properties, such as having more than one mode active or having no mode active when a mode must be active, could be violated in our model. The model checker was relentless in tracking these scenarios down and presenting us with a counter example. Practicing engineers are well aware of the difficulty of identifying all such scenarios and have evolved a series of defensive coding practices to ensure that the safety properties are not violated. Model checking of the specification allowed us to provide a rigorous analysis that the specification cannot violate these properties in the first place.

As one example, an entire class of errors was discovered that involved more than one input event arriving at the same time. This could occur for a variety of reasons. For example, the pilot might press a switch at the same time as the system captured a navigation source. Occasionally, these combinations would drive the model into an unsafe state.

There are several ways to deal with such simultaneous input events. SCR [23,22] makes the "one input assumption" mandating that only one input variable can change in any step. This makes reasoning about the specification simpler, but requires that the developer implement the system in such a way as to guarantee that only one input variable can change in each step. In a polling system, where all the inputs are sampled at periodic intervals, this can only be done by adding additional logic outside the specification that prioritizes multiple events and discards lower priority events or queues them for processing in subsequent steps.

RSML^{-e} normally makes a similar "one input message" assumption in which only one message is processed in each step, but any number of fields within the message are allowed to change in a single step. Since we were uncertain how communication with the outside

Table 1. Sample of English Requirements and CTL Translation from DOORS Database

English Requirement	**CTL Property**
1. Mode Annunciations	
1.1 Mode Annunciation Selection	
If this side is active and the mode annunciations are off, the mode annunciations shall be turned on when the onside FD is turned on.	$AG((!Mode\_Annunciations\_On \ \& \ !Onside\_FD\_On) \rightarrow AX((Is\_This\_Side\_Active = 1 \ \& \ Onside\_FD\_On) \rightarrow Mode\_Annunciations\_On))$
If this side is active and the mode annunciations are off, the mode annunciations shall be turned on when the offside FD is turned on.	$AG((!Mode\_Annunciations\_On \& Offside\_FD\_On = FALSE) \rightarrow AX((Is\_This\_Side\_Active = 1 \ \& \ Offside\_FD\_On = TRUE) \rightarrow Mode\_Annunciations\_On))$
If this side is active and the mode annunciations are off, the mode annunciations shall be turned on when the AP is engaged.	$AG((!Mode\_Annunciations\_On \ \& \ !Is\_AP\_Engaged) \rightarrow AX((Is\_This\_Side\_Active = 1 \ \& \ Is\_AP\_Engaged) \rightarrow Mode\_Annunciations\_On))$
1.2 Mode Annunciation Deselection	
If this side is active and the mode annunciations are on, the mode annunciations shall be turned off if the onside FD is off, the offside FD is off, and the AP is disengaged.	$AG(Mode\_Annunciations\_On \rightarrow AX((Is\_This\_Side\_Active = 1 \ \& \ !Onside\_FD\_On \ \& \ Offside\_FD\_On = FALSE \ \& \ !Is\_AP\_Engaged) \rightarrow !Mode\_Annunciations\_On))$
1.3 Mode Annunciation Operation	
The mode annunciations shall not be on at system power up.	$(!Mode\_Annunciations\_On)$
If this side is active the mode annunciations shall be on if and only if the onside FD cues are displayed, or the offside FD cues are displayed, or the AP is engaged.	$AG(Is\_This\_Side\_Active = 1 \rightarrow (Mode\_Annunciations\_On \leftrightarrow (Onside\_FD\_On \mid Offside\_FD\_On = TRUE \mid Is\_AP\_Engaged)))$

world would ultimately be implemented, we selected an RSML^{-e} option in which all input messages (and hence all input variables) were read once on each step. This allowed for the possibility that all 41 input variables could change in the same step.

The problem was simplified somewhat in that only 21 of these input variables were of concern. The other 20 input variables provide state information from the other FGS used to set the state of the current FGS when it is the inactive (backup) side and had no impact on the system state when the current side was active. However, this still left 21 input variables that could change in a single step. To deal with this, we assigned a priority to each input event and only used the highest priority event in each step, ignoring the lower priority events.

The logic to do this was localized in one part of the model. In the original specification, each event was defined as a macro with a name of the format "When_Event_X" which took on the value TRUE for one step when the Boolean variable X changed from FALSE to TRUE. For each such macro, we defined a new macro named "When_Event_X_Seen" that behaved in the same way unless a higher priority event occured at the same time, in which case it remained FALSE. All references in the specification to "When_Event_X" were then replaced by references to "When_Event_X_Seen". In this way, the prioritization of events could easily be modified without changing the main body of the specification.

In a few cases, such as the acquisition of a navigation signal, it was undesirable to simply ignore the event. In these cases, the macro "When_Event_X_Seen"was changed to return TRUE whenever the variable X was TRUE (rather than when the variable changed from FALSE to TRUE) unless a higher priority event occured in that step. [3] In this way, the desired action would be taken in the first step in which no higher priority event occurred. These changes effectively sequence the events so that only one can affect the specification at a time. This is similar to a "one input assumption", but the sequencing is defined within the specification, rather than outside of it.

In the course of developing this prioritization, we realized that it was possible for some combinations of events to be processed in the same step. For example, an input that changed the active lateral mode could often (but not always) be processed in the same step as an input that changed the active vertical mode. In

[3] In these cases, the macro probably should have been renamed to "When_Condition_X_Seen" to better reflect its behavior.

other words, a partial rather than a total order of the input events was acceptable. This partial order had three branches, with a maximum depth of eleven input events (i.e., eleven priorities) on a single branch. It was quite straightforward to understand, both by us and by the engineers who reviewed it for us. Since we could check both the safety and functional properties of the specification with NuSMV, we felt confident that the specified behavior was correct. However, without the power of formal verification, we would never have been able to convince ourselves that the safety properties of the system were still met when such multiple input events were allowed.

The handling of multiple input events has been a recurring issue in our examples, and appears to be a natural consequence of implementing a formal specification on an actual processor where system steps require a finite amount of time. On the one hand, it is impractical to ask human beings to reason about all possible combinations of inputs events in the main body of the specification. On the other hand, it is very difficult, if not impossible, to design systems that can guarantee that only one external input will change during a system step. Even interrupt driven systems must prioritize and queue external events that occur while a higher priority event is being handled. Our preference is to allow for the occurrence of multiple inputs, but to keep the logic that prioritizes the events separate from the logic that defines the processing of each individual event.

3.5 Theorem Proving

After verification of the mode logic with the NuSMV model checker was well underway, the University of Minnesota team completed the first version of the RSML^{-e} to PVS translator. This allowed us to start verifying properties using the PVS theorem prover.

In contrast to model checkers, theorem provers apply rules of inference to a specification in order to derive new properties of interest. Theorem provers are generally considered harder to use than model checkers, requiring more expertise on the part of the user. However, theorem provers are not limited by the size of the state space.

Even though we had been able to verify all the requirements against the RSML^{-e} model, we wanted to assess the use of PVS for a variety of reasons. First, we knew that not all problem domains would lend themselves to verification through model checking as well as the mode logic had. Models with very large or infinite state spaces would not be analyzable using model checking. We expected to encounter such problems when analyzing trajectories of aircraft relative to the flight plan. Also, the mode logic was already starting to strain the capabilities of NuSMV, and we were concerned that problems with larger state spaces would exceed its capabilities. For problems just at the limit of model checking, we speculated that theorem proving might even be more efficient than model checking. Finally, we had identified at least one class of properties, comparing the properties of two arbitrary states that were not temporally related to each other, that we were unable to state in CTL. An example of this was the property that any two arbitrary states with different mode configurations should have different annunciations (i.e., different mode indications on the Primary Flight Display) to the pilots.

We started by using PVS to verify some of the properties already confirmed using NuSMV. Since the same RSML^{-e} model was used to generate the PVS specification as was used to generate the NuSMV model, the same handful of manual abstractions were present in the PVS specification even though they were probably not necessary for PVS. In the course of completing the proofs, it became clear that we needed to define and prove many simple properties of the FGS that could be used as automatic rewrite rules by PVS. This automated and simplified the more complex proofs we were interested in. For example, we followed the RSML^{-e} convention of assigning input variables the initial value of UNDEFINED. This prevents the model from making use of an initial value that does not reflect the actual environment around it, a common cause of safety errors in automated systems. As a consequence, all internal variables and functions dependent on those input variables included UNDEFINED in their range, even though guards in their definitions ensured they could never take on the value UNDEFINED. By defining and proving properties stating that these variables and functions were always defined, PVS was able to automatically resolve large portions of the proofs. As these libraries evolved, we realized that many of these properties, as well as several useful PVS strategies (scripts defining sequences of prover commands) could have been automatically produced by the translator. These were identified as enhancements for future versions of the translator.

With this infrastructure in place, some of the proofs could be constructed in less than an hour. Others took several hours or even days, usually because they involved proving many other properties as intermediate lemmas. One surprise was that users proficient in PVS but unfamiliar with the FGS could usually complete a proof as quickly as someone familiar with the FGS. In fact, most of the proofs were completed by a graduate student with no avionics experience. The general process was to break the desired property down by case splits until a simple ASSERT or GRIND command could complete that branch of the proof tree. The structure of the proofs naturally reversed the dependency ordering defined in the RSML^{-e} specification. Many of the proofs could be simplified by introducing lemmas describing how intermediate values in the dependency graph changed, but identifying such lemmas seemed to require a sound understanding the FGS mode logic. As we gained experience, we started using the dependency map produced by the RSML^{-e} toolset to guide us in identifying these lemmas.

While the proofs might take hours to construct, they usually executed in less than twenty seconds. This was significant since the time taken to prove similar properties using the NuSMV model checker had grown steadily with the size of the model. If the model had grown much larger, it is possible that the time to verify a property using model checking might have become prohibitive. The time required to run the PVS proofs seemed much less sensitive to the size of the model.

Since we had already completed the safety analysis of the mode logic using NuSMV, we decided to focus on using PVS to study the mode logic for potential forms of mode confusion. Mode confusion occurs when the operators of an automated system believe they are in a mode different than the one they are actually in and make inappropriate responses to the automation. Mode confusion can also occur when the operators do not fully understand the behavior of the automation, i.e., when the operators have a poor "mental model" of the automation. Numerous studies have shown that mode confusion is an important safety concern in automated systems such as modern avionics systems [8,44–46].

In earlier work [32,33], we had extended a taxonomy of design patterns indicative of potential sources of mode confusion originally developed by Nancy Leveson [29]. Other researchers have described ways in which formal analysis tools can be used to search specifications for such patterns [11,42,41,43]. We decided to try using PVS to determine if there were patterns in our requirements model that might indicate potential sources of mode confusion. We were able to use PVS to search for ways that a system could enter and exit infrequently used (off normal) modes, ignore pilot inputs, introduce unintended side effects, enter and exit hidden modes of operation, and provide insufficient feedback to the pilots [33]. While space does not permit a complete description, we do present here an example of how PVS was used to detect ignored pilot inputs.

The basic approach is to prove that each pilot input provides some visible change in the system state. For example, to prove that pressing the Flight Director (FD) switch always causes a change in the visible state, we attempt to prove the theorem

```
FD_Switch_Never_Ignored : Theorem
  verify((When_FD_Switch_Pressed AND
            No_Higher_Event_Than_FD_Switch)
    IMPLIES
            (Onside_FD_On /= PREV(Onside_FD_On)))
```

This theorem asserts that if the FD switch is pressed, and no higher priority event occurs at the same time, the onside FD guidance cues toggle on and off. Trying to prove this lemma leads to the following sequent that must be discharged in PVS

```
[-1] *(Overspeed_Condition(s!1))
[-2] *(Onside_FD(s!1))=*(Onside_FD(s!1-1))
[-3] *(When_FD_Switch_Pressed(s!1))
[-4] *(No_Higher_Event_Than_FD_Switch(s!1))
  |-------
[1] *(Onside_FD(s!1-1)) = Off
[2] s!1 = 0
```

As with all PVS sequents, we are allowed to assume that properties above the turnstile (`|-------`) are true and that at least one property from below the turnstile must be proven true to discharge the proof obligation. The current state is s!1 and the previous state is s!1-1.

This sequent requires us to prove that either an overspeed condition cannot occur at arbitary time s!1 [-1], that the pilot cannot presss the FD switch at time s!1 [-3], that some higher priority event will always occur when the FD switch is pressed at time s!1 [-4], that the FD cues must always be off at time s!1-1 just before the pilot presses the FD switch [1], that s!1 is the inital state [2], or that the onside FD cues will change value value at time s!1. The first four are impossible constraints on the system inputs, the fifth is precluded since it implies the FD switch is pressed in the initial state (which is not possible given the definition of the macro FD_Switch_Pressed), and the last is what we started out to prove. From this we can conclude that the property we are trying to prove is probably false.

The sequent provides us with a clue of what is wrong in that one way to complete the proof would be to show that an overspeed condition [-1] cannot occur at time s!1. This is impossible, but review of the specification reveals that the FD switch is indeed ignored during an overspeed condition if the onside FD cues are on. To confirm that this is the problem, and to document this case of an ignored pilot input, we define a constraint

```
FD_Switch_Ignored_During_Overspeed: rCOND
   = (When_FD_Switch_Pressed AND
       Onside_FD_On AND Overspeed_Condition)
```

identifying the condition in which the FD switch is pressed, the onside FD is on, and an overspeed condition exists. We then use this to state an amended version of the theorem

```
FD_Switch_Never_Ignored : Theorem
  verify((When_FD_Switch_Pressed AND
            No_Higher_Event_Than_FD_Switch AND
            NOT FD_Switch_Ignored_During_Overspeed)
    IMPLIES
            (Onside_FD_On /= PREV(Onside_FD_On)))
```

stating that the FD switch is never ignored unless it is pressed during an overspeed condition while the FD cues are on. This proof completes without difficulty, taking a little under ten seconds to run.

In [11], we discuss how PVS was used to detect ignored pilot inputs in small, handcrafted models of the mode logic. We were not sure that we would be able to do similar proofs on PVS models translated from a much larger RSML^{-e} model of the mode logic. However, as this example shows, performing proofs over these models was no more difficult than doing them over the handcrafted models once the basic infrastructure was in place.

4 Observations on Specification Styles

There are at least two well-known styles of formal specification. In a *property*, or axiomatic, style of specification, one defines properties relating the operations of the type being specified without providing any information about the structure of the type itself. The common textbook example is the specification of a stack through equational specifications such as *top(push(s,e)) = e*.

In contrast, in a *constructive*, or model-based approach, one defines a new type in terms of primitive types and constructors provided by the specification language. For example, one might define a stack as a record consisting of an array *a* of the base type *e* and an integer *tos* representing the top of stack pointer. An operation such as *top* might then be defined as *top([a, tos]) = a(tos)*. That is, *top* returns the array element pointed at by *tos*.

The primary disadvantage of a constructive style of specification is that it biases the reader towards a particular implementation. In the example above, the specification strongly suggests that a stack *should* be implemented as a record containing an array and an integer. No such bias exists in the property style of specification since no information is provided about the structure of the type being defined. An advantage of a constructive style of specification is that it is used in common programming languages such as C and Ada and most engineers are immediately comfortable with it.

A property oriented specification can also be more difficult to understand and write. One also has to ensure that a property oriented specification is *consistent* and *complete*. A specification is consistent if it always defines a single value for each operation on the same inputs (i.e., each operation is a *function*). A specification is complete if a result is specified for every set of inputs to an operation (i.e., each operation is a *total* function).

Most constructive specification languages are designed so that *only* complete and consistent specifications can be written. In fact, the textbook method for showing that a property oriented specification is consistent is to create a constructive model of it and prove that all the properties hold over that model. This establishes that at least one implementation of the specification exists and its properties must therefore be consistent.

The analogies to our two styles of requirements specification are obvious. Requirements written as shall statements in a natural language are simply informal property oriented specifications. In addition to the usual problems of ensuring completeness and consistency, they are also encumbered by the ambiguity of natural language. This helps to explain why developers working from requirements captured as informal shall statements usually complain of problems with completeness, consistency, and ambiguity.

In contrast, requirements captured using notations such as SCR and RSML^{-e} actually are constructive models of the requirements. Due to the language constructs provided, they are inherently complete and consistent in the sense of defining a total function for each output. However, this also explains why a common reaction to such models is that they contain design decisions. In all honesty, they do suggest certain design decisions, even if nothing more than the names of internal variables that the customer does not care about.

These observations allow us to begin to address the questions raised in the introduction. The product lifecycle in the model-based development paradigm starts with informal techniques, such as writing shall statements in natural language or the development of use cases, to capture the requirements during the early, elicitation phase of the project. This is followed by the creation of a constructive model of the requirements that can be used to drive visualizations of the user interface so that the customer can simulate the requirements model and provide early feedback and validation. In the analysis phase, the informal statements are translated into properties over the model and proven to ensure their consistency and completeness. High-quality code generators and test case generators reduce much of the effort traditionally associated with coding and testing. Finally, since the models have been carefully developed so as to encapsulate key functions, selected components can be reused in the next project; in a sense closing a development cycle by providing assets for the next generation in a family of products. The use of property oriented and constructive specification styles are indicated as oval and rectangular artifacts in Figure 4 on page 9.

One of the questions posed in the introduction was whether requirements should be captured as a list of shall statements written in a natural language or whether they should be written as mathematical models defining the relationship between the inputs and outputs as is done in SCR, CoRE, and RSML^{-e}. The observation that shall statements are just informal statements of the system properties suggests these positions are not in conflict. The very commonality of us of shall statements indicates they are a natural and intuitive way for designers to put their first thoughts on paper. The problem with shall statements has always been that inconsistencies, incompleteness, and ambiguities are not found until the later phases of the project. However, by developing a formal, constructive model of the requirements against which the informal shall statements can be verified, identification of these problems can be forced into the early modeling, simulation, and analysis phases of the project.

Another question raised was whether a system's requirements can be completely specified with use cases. While more structured than shall statements, as practiced today use cases normally lack a precise formal semantics and suffer from the same problems of inconsistency, incompleteness, and ambiguity as shall statements. While not part of this exercise, it seems reasonable that it should be possible to express use cases as a sequence of properties describing how the system responds to its stimuli, and these sequences verified through simulation and formal analysis. In this way, the consistency and completeness of use cases could be improved in the same manner as was done for shall statements.

We also raised the question of when does one cross the line between requirements analysis and design, and why does that matter? The traditional answer is that requirements should not contain anything the customer does not require in order to avoid placing unnecessary constraints on the developers. For this reason, constructive models are often criticized for introducing design bias into the requirements. However, the reality is that for any real system, the requirements will be many and the models will be large and complex. Large and complex models need to be structured to be readable and robust in the face of change, and hopefully to be reused. This suggests that we *should* group portions of the model together that are logically related and likely to change together, and that requirements analysis *should* be driven by some of the same concerns that have traditionally been associated with the design process. Our preference is to define requirements analysis as the process of specifying the complete platform independent (logical) behavior of the system and to define design as the process of mapping of that behavior onto a specific (physical) platform. In this view, the requirements evolve from the informal definition gathered during elicitation to a formal, highly structured model suitable for the automatic generation of code and test cases.

Perhaps the most critical question is the cost effectiveness of formal verification. Traditional experiments in formal verification have often appeared too expensive because they included the cost of creating a formal specification (usually in addition to the informal specification actually used by the developers) and revolved around techniques such as theorem proving that do require significant expertise and time. Indeed, the largest cost in this exercise was creating the RSML^{-e}model of the FGS. Even though the problem domain was well understood and had been used in previous exercises, this still took several weeks for the authors to complete. As expected for this particular problem, proving properties using the PVS theorem prover did turn out to be more expensive than using the NuSMV model checker. However, manually translating the shall statements into CTL or PVS properties took only a few minutes, and proving these properties using NuSMV again took only a few minutes if the property was true. Of course, if the property was false, considerable time was sometimes required to correct the model, but this should be viewed as a development rather than as a verification cost (if the model had been correctly developed in the first place, the cost of correction would not have been incurred).

One of the great advantages of model-based development is that the formal model is created as an intrinsic step in the development process. By demonstrating that such models can be automatically translated into formal verification tools such as NuSMV and PVS, this exercise shows that it is feasible to eliminate the cost of developing a separate specification for the purpose of formal verification. In addition, this exercise shows that for at least some industrial systems, model checking can be used to very inexpensively verify properties of these models and find errors early in the life-cycle. Our experiences suggest that there are large portions of most systems that are suitable for verification by model checking.

As discussed in the introduction, the cost of finding errors late in the life-cycle is enormous. Our conclusions from this exercise are that if a formal model is created as an integral step in the development process, the benefit of using model checking to find errors early in the life-cycle is highly cost effective. In fact, the returns are so great that for critical properties, we believe the cost of theorem proving would also be well justified.

5 Conclusions and Future Directions

We have described how a model of the requirements for the mode logic of a Flight Guidance System was created in the RSML^{-e} language from an initial set of requirements stated as shall statements written in English. Translators were used to automatically generate equivalent models of the mode logic in the NuSMV model checker and the PVS theorem prover. The original shall statements were then hand translated into properties over these models and proven to hold over these models.

The process of creating the RSML^{-e} model improved the informal requirements, and the process of proving the formal properties found errors in both the original requirements and the RSML^{-e} model. Our concerns about the difficulty of proving properties in the NuSMV and PVS models that were automatically generated from the RSML^{-e} models turned out to be unfounded. In fact, the ease with which these properties were verified leads us to conclude that formal methods tools are finally maturing to the point where they can be profitably used on industrial sized problems.

Several directions exist for further work. We would like to explore translating use cases into sequences of properties that can be formally verified, just as was done with shall statements in this exercise. Stronger abstraction techniques are needed to increase the classes of problems that can be verified using model checkers. Better libraries and proof strategies are needed to make theorem proving less labor intensive. More work also needs to be done to identify proof strategies and properties that can be automatically generated from the model. Since many systems consist of synchronous components connected by asynchronous buses, work needs to be done to determine how properties that span models connected by asynchronous channels can be verified.

Over the last few years, a small number of commercial modeling languages with formal or nearly formal semantics, such as SCADE [1] or Simulink [5], have gained widespread acceptance in industry. While not specifically designed as requirement specification languages, restricted subsets of these notations can be used in virtually the same way as RSML^{-e} . Due to their growing acceptance by the engineering community, most of our recent work has been devoted to retargeting our translators to SCADE and Simulink. Using these tools, we starting to use model checking to validate the requirements of future products.

Acknowledgements

The authors wish to acknowledge the ongoing support of this work by Ricky Butler, Kelly Hayhurst, and Celeste Bellcastro of the NASA Langley Research Center, and the efforts of Anjali Joshi, Yunja Choi, Sanjai Rayadurgam, George Devaraj, and Dan O'Brien of the University of Minnesota in developing the technology described in this paper.

References

1. Anonymous. Esterel Technologies Home Page. Available at http://wwww.esterel-technologies.com.
2. Anonymous. NASA Software Assurance Technology Center Formal Inspections Page. Available at http://satc.gsfc.nasa.gov/fi/fipage.html.
3. Anonymous. NuSMV Home Page. Available at http://nusmv.irst.itc.it/.
4. Anonymous. PVS Home Page. Available at http://www.csl.sri.com/projects/pvs.
5. Anonymous. The MathWorks Home Page. Available at http://wwww.mathworks.com.
6. S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas (1999). A methodology for proving control systems with Lustre and PVS. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, pages 89–107, San Jose, CA, January 1999. IEEE Computer Society.
7. G. Berry and G. Gonthier (1992). The synchronous programming lanugage esterel: Design, semantics, and implementation. *Science of Computer Programming*, 19:87–152, 1992.
8. C. Billings (1997). *Aviation Automation: the Search for a Human-Centered Approach.* Lawrence Erlbaum Associates, Inc, Mahwah, NJ, 1997.
9. B. Boehm (1981). *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.
10. F. Brooks (1997). No silver bullet: : Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1997.
11. R. Butler, S. Miller, J. Potts, and V. Carreno (1998). A formal methods approach to the analysis of mode confusion. In *17st Digital Avionics Systems Conference (DASC'98)*, volume 1, pages C41/1 – C41/8, Belllevue, WA, October 1998.
12. W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese (1998). Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
13. Y. Choi and M. Heimdahl (2002). Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pages 109–118, Tokyo, Japan, October 2002.
14. Y. Choi, S. Rayadurgam, and M. Heimdahl (2002). Toward automation for model checking requirement specifications with numeric constraints. *Requirements Engineering Journal*, 7(4):225–242, December 2002.
15. E. Clark, O. Grumberg, and D. Peled (2001). *Model Checking.* The MIT Press, Cambridge, Massachusetts, 2001.
16. A. Davis (1993). *Software Requirements: Object, Function, and States.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
17. L. de Moura (2004). *SAL: Tutorial.* SRI International, Computer Science Laboratory. 333 Ravenswood Ave. Menlo Park, CA 94025, January 2004.
18. M. Fagan (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, April 1976.
19. S. Faulk, J. Brackett, P. Ward, and J. Kirby (1992). The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.
20. S. Faulk, L. Finneran, J. Kirby, S. Shah, and J. Sutton (1994). Experience applying the CoRE method to the Lockheed C-130J software requirements. In *Ninth Annual Conference on Computer Assurance*, pages 3–8, Gaithersburg, MD, June 1994.
21. D. Harel and A. Naamad (1996). The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, October 1996.
22. C. Heitmeyer, J. Kirby, and B. Labaw (1996). Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):231–261, July 1996.
23. C. Heitmeyer, B. Labaw, and D. Kiskis (1995). Consistency checking of SCR-style requirements specifications. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, pages 56–65, March 1995.

24. A. Joshi, S. Miller, and M. Heimdahl (2003). Mode confusion analysis of a flight guidance system using formal methods. In *22nd Digital Avionics Systems Conference DASC'03*, pages 2.D.1–1 – 2.D.1–11, October 2003.
25. N. Leveson. *Safeware: System Safety and Computer* (1995). Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
26. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese (1993). *TCAS II Collision Avoidance System (CAS) System Requirements Specification change 6.00.* Federal Aviation Administration, U.S. Department of Transportation, March 1993.
27. N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese (1994). Requirements specifications for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.
28. N. Leveson, M. Heimdahl, and J. Reese (1999). Designing specification languages for process control systems: Lessons learned and steps to the future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.
29. N. Leveson, D. Pinnel, S. Sandys, S. Koga, and J. Reese (1997). Analyzing software specifications for mode confusion potential. In *Workshop on Human Error and System Development*, Glascow, Scotland, March 1997.
30. R. Lutz (1993). Analyzing software requirements errors in safety-critical, embedded systems. In *IEEE Symposium on Requirements Engineering*, pages 126–133, San Diego, CA, 1993.
31. S. Miller (1998). Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Second Workshop on Formal Methods in Software Practice (FMSP98)*, pages 44–53, Clearwater Beach, Florida, 1998.
32. S. Miller (2001). Taxonomy of mode confusion sources - final report. In *NASA Contractor Report*, February 2001.
33. S. Miller and A. Tribble (2002). A methodology for improving mode awareness in flight guidance design. In *21st Digital Avionics Systems Conference (DASC'02)*, volume 2, pages 7D1–1 – 7D1–11, Irvine, CA, October 2002.
34. S. Miller, A. Tribble, T. Carlson, and E. Danielson (2003). Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA Langley Research Center, June 2003. Available at http://techreports.larc.nasa.gov/ltrs/refer/2003/cr/NASA-2003-cr212426.refer.html.
35. D. Owen and T. Menzies (2003). Lurch: a lightweight alternative to model checking. In *Proceedings of the 2003 Software Engineering and Knowledge Engineering Conference (SEKE'03)*, 2003.
36. S. Owre, J. Rushby, and N. Shankar (1995). Analyzing tabular and state-transition requirements specifications in PVS. Technical Report SRI-CSL-95-12, SRI International, June 1995.
37. S. Owre, J. Rushby, N. Shankar, and F. Henke (1995). Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
38. D. Parnas and J. Madey (1991). Functional documentation for computer systems engineering (Volume 2). Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1990.
39. C. Ramamoorthy, A. Prakesh, W. Tsai, and Y. Usuda (1984). Software engineering: Problems and perspectives. *IEEE Computer*, pages 191–209, October 1984.
40. S. Rayadurgam, A. Joshi, and M. Heimdahl (2003). Using PVS to prove properties of systems modelled in a synchronous dataflow language. In *Proceedigns of the 5th International Conference on Formal Engineering Methods (ICFEM 2003)*, pages 167–186, Singapore, November 2003.
41. J. Rushby (1999). Using model checking to help discover mode confusions and other automation surprises. In *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD'99)*, Liege, Belgium, June 1999.
42. J. Rushby (2001). Analyzing cockpit interfaces using formal models. *Electronic Notes in Theoretical Computer Science*, 43:1–14, 2001.
43. J. Rushby, J. Crow, and E. Palmer (1999). An automated method to detect potential mode confusion. In *Proceedings of the 18th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, volume 1, pages 4.B.2–1 – 4.B.2–6, St. Louis, MO, October 1999.
44. N. Sarter and D. Woods (1992). Pilot interaction with cockpit automation: Operational experiences with the flight management system. *The International Journal of Aviation Psychology*, 2(4):303 – 331, 1992.
45. N. Sarter and D. Woods (1994). Pilot interaction with cockpit automation II: An experimental study of pilots' model and awareness of the flight management system. *The International Journal of Aviation Psychology*, 4(1):1 – 28, 1994.
46. N. Sarter and D. Woods (1995). How in the world did I ever get into that mode?: Mode error and awareness in supervisory control. *Human Factors*, 37(1):5 – 19, 1995.
47. J. Thompson, M. Heimdahl, and S. Miller (1999). Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.
48. J. Thompson, M. Heimdahl, and S. Miller (1999). Specification based prototyping for embedded systems. Technical Report TR 99-006, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1999.
49. A. Tribble and S. Miller (2002). Safety analysis of a flight guidance system. In *21st Digital Avionics Systems Conference (DASC'02)*, volume 2, pages 13C1–1 – 13C1–10, Irvine, CA, October 2002.
50. A. van Schouwen (1990). The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report 90-276, Queens University, Hamilton, Ontario, 1990.
51. M. W. Whalen (2000). A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.