

## Proving the Shalls<sup>1</sup>

Steven P. Miller<sup>1</sup>, Alan C. Tribble<sup>1</sup>, and Mats P.E. Heimdahl<sup>2</sup>

<sup>1</sup> Rockwell Collins, 400 Collins Road NE,  
Cedar Rapids, Iowa, 52498, USA

{spmiller, actribbl}@rockwellcollins.com

<sup>2</sup> Department of Computer Science and Engineering, University of Minnesota,  
4-192 EE/CSci Bldg, 200 Union Street S.E., Minneapolis, Minnesota, 55455, USA  
heimdahl@cs.umn.edu

**Abstract.** This paper describes an experiment conducted to determine how effectively formal methods could be used to capture and validate the requirements of a typical embedded system. A model of the mode logic of a Flight Guidance System was specified in the RSML<sup>c</sup> notation and translated into the NuSMV model checker and the PVS theorem prover. These tools were then used to verify several hundred properties of the RSML<sup>c</sup> model. In the process, several errors were discovered and corrected in the original model. This demonstrates that formal requirements models can be written for real problems and that formal analysis tools have matured to the point where they can be used to find errors before implementation. It also points out a clear relationship between requirements stated informally as “shalls”, formal properties, and requirements models.

## 1 Introduction

Incomplete, inaccurate, ambiguous, and volatile requirements have plagued the software industry since its inception. In a 1987 article, Fred Brooks wrote [1]

*“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 fixing a requirements error grows dramatically the later in the product life cycle it is corrected [2], [3], [4], [5]. Re-

---

<sup>1</sup> This project was partially funded by the NASA Langley Research Center under contract NCC1-01001 of the Aviation Safety Program.

searchers have also found that requirements errors are more likely to affect the safety of a system than errors introduced during design or implementation [6], [7].

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 [8] was originally developed to specify the requirements for the A-7 aircraft [9]. It was later extended to the CoRE methodology by the Software Productivity Consortium [10] and used to specify the avionics requirements on the Lockheed C-130J [12]. 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 [13]. Even Statecharts, whose various derivatives make up one of the most widely accepted modeling notations in use today, has its roots in the avionics industry [13].

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 “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 an experiment 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 Flight Guidance System was specified in the RSML<sup>c</sup> notation. While this model was representative in size and complexity of an actual system, it did not describe a fielded product. Translators were developed from RSML<sup>c</sup> to the NuSMV model checker and the PVS theorem prover. These tools were then used to verify several hundred properties of the RSML<sup>c</sup> model. In the process, several errors were discovered and corrected in the original RSML<sup>c</sup> model.

The results of this experiment are dramatic. They demonstrate 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 experiments 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, [14] describes the authors’ experiences modeling the mode logic informally using the CoRE methodology [11] and the benefits that were gained from entering this model into the SCR\* tool and using the consistency and completeness checks provided by SCR\* [8]. In [16], 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 experiment, the same model was used for specification, review, and simulation, and automatically translated into other notations for

formal verification. Also, all of the functional and safety requirements were formally verified in a clearly cost-effective manner.

Perhaps just as important, this experiment 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 in detail in Section 4.

## 2 Background Information

This section provides useful background information, including a description of the role of a Flight Guidance System in a modern aircraft and provides a brief overview of the RSML<sup>c</sup> notation, 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 and active at any given time. These in turn determine which flight control laws are generating guidance commands.

Our model of the FGS function includes identical left and right sides. In most modes, 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 [14], [15], [16], [17]. 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<sup>c</sup> Specification Language

For this exercise, we specified the FGS mode logic using the RSML<sup>c</sup> notation, a derivative of RSML. RSML is a state-based specification language developed by Leve-

son's group at the University of California at Irvine as a language for specifying the behavior of process control systems [12]. 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.

RSML was heavily influenced by Statecharts [13] and uses a similar notion of explicit event propagation. In the course of developing the TCAS-II specification and the independent verification and validation effort, it became clear that the most common source of errors was this dependence on explicit events. To reduce this problem, the Critical Systems group at the University of Minnesota developed RSML<sup>e</sup> (RSML without events) [18]. As its name implies, RSML<sup>e</sup> eliminates the use of explicit events and is a synchronous language [19]. RSML<sup>e</sup> 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 the modeling tasks. An example of an RSML<sup>e</sup> specification can be found in [20].

### **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 Trinito 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 [21], 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 [22]. Properties to be verified in NuSMV are specified using either Computation Tree Logic (CTL) or Linear Time logic (LTL) [21].

### **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 the Boyer-Moore prover, 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. [23], [24].

### 3 The Requirements Analysis Process

In the next few sections, we describe the process we followed in eliciting, modeling, and analyzing the requirement of the FGS mode logic.

#### 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 database<sup>2</sup>. Examples of a few such requirements for the FGS mode logic are shown in the left hand column of Table 1.

#### 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 [10], which is in turn based on the SCR methodology [8], [9].

This model was written in the RSML<sup>c</sup> language. One of the great advantages of executable specification languages such as RSML<sup>c</sup> or SCR is that they can be connected to a mock-up 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.

When completed, the RSML<sup>c</sup> 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 [25].

In the course of building the RSML<sup>c</sup> model, we found ourselves going back and modifying the original shall statements. Sometimes, they were just wrong. More often,

---

<sup>2</sup> DOORS (Dynamic Object Oriented Requirements System) by Telelogic is a commercial requirements management tool.

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<sup>c</sup> 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<sup>c</sup> to NuSMV translator. This translator, described in [26], automatically converted the RSML<sup>c</sup> model to the specification language of the NuSMV model checker, a dialect of SMV. This made it possible for us to start formally checking our model.

We knew state space explosion would be a problem since we had included in the RSML<sup>c</sup> 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. To deal with the few remaining integer variables, we abstracted the model by hand by moving comparisons involving these variables (e.g.,  $\text{Altitude} > \text{PreSelectAlt} + \text{AltCapBias}$ ) into a different part of the specification and inputting the Boolean results directly into the model. 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 experiment because they were so few and so straightforward. In other domains, it is likely that 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. Work is underway to add these capabilities to the RSML<sup>c</sup> to NuSMV translator.

At first, we focused on showing that our model satisfied the safety properties we had identified through a hazard analysis and fault tree analysis [17], [27]. 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<sup>c</sup> 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<sup>∘</sup> 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  $\rightarrow$  and  $\leftrightarrow$  have their usual meaning of “implies” and “iff”. Occasionally, the semantics of RSML<sup>∘</sup> and CTL interacted in inelegant ways. For example the input variable `Offside_FD_On` in the above example could take on the values `TRUE`, `FALSE`, or `UNDEFINED` in RSML<sup>∘</sup> and had to be explicitly compared with the value `TRUE` in CTL.

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))
```

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

Only these two formats were needed because RSML<sup>∘</sup> 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. These were sufficient to describe all the safety properties and functional requirements. 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 [21].

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.

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

<b>English Requirement</b>	<b>CTL Property</b>
<b>1. Mode Annunciations</b>	
<b>1.1 Selection</b>	
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.	SPEC AG(!Mode_Annunciations_On & !Onside_FD_On) -> AX((Is_This_Side_Active = 1 & Onside_FD_On) -> 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.	SPEC AG(!Mode_Annunciations_On & Offside_FD_On = FALSE) -> AX((Is_This_Side_Active = 1 & Offside_FD_On = TRUE) -> 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.	SPEC AG(!Mode_Annunciations_On & !Is_AP_Engaged) -> AX((Is_This_Side_Active = 1 & Is_AP_Engaged) -> Mode_Annunciations_On))
<b>1.2 Deselection</b>	
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.	SPEC AG(Mode_Annunciations_On -> AX((Is_This_Side_Active = 1 & !Onside_FD_On & Offside_FD_On = FALSE & !Is_AP_Engaged) -> !Mode_Annunciations_On))
<b>1.3 Operation</b>	
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 -> (Mode_Annunciations_On <-> (Onside_FD_On   Offside_FD_On = TRUE   Is_AP_Engaged)))



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<sup>c</sup> model of the mode logic that had not been discovered through simulation. 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.

Clarifying whether the FGS needs to be active, while desirable, is a condition 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 allows 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 [8] 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<sup>c</sup> 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 world would ultimately be implemented, we selected an 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 specification so that the only change to the main body of the specification was to replace the references of the form “When\_Event\_X” with references of the form “When\_Event\_X\_Seen”. In a few cases, such as the acquisition of a navigation signal, it was undesirable to simply ignore the event. In these cases, the specification was changed to depend on the condition itself rather than on the event of the condition becoming true. In this way, the condition would be processed in the first step in which a higher priority event did not preempt it. These changes effectively implemented a “one input assumption” within the RSM<sup>c</sup> specification.

In course of developing this prioritization, we realized that it was possible for some combinations of events to be processed in the same step as the order in which the events were processed did not matter. 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 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.

The handling of multiple input events has been a recurring issue in our experiments, 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<sup>c</sup> 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<sup>c</sup> 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 to the pilots.

We started by using PVS to verify some of the properties already confirmed using NuSMV. Since the same RSML<sup>c</sup> 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<sup>c</sup> 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 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 proof's naturally reversed the dependency ordering defined in the RSML<sup>c</sup> 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<sup>c</sup> toolset to guide us in identifying these lemmas.

Another surprise was that 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 [28], [29], [30], [31].

In earlier work [32], [16], we had extended a taxonomy of design patterns indicative of potential sources of mode confusion originally developed by Nancy Leveson [33]. Other researchers have described ways in which formal analysis tools can be used to search specifications for such patterns [15], [34], [35], [36]. 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 off normal modes, ignore pilot inputs, intro-

duce unintended side effects, enter and exit hidden modes of operation, and provide insufficient feedback to the pilots [37]. 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))
[ -5 ] * (Onside_FD(s!1)) = * (Onside_FD(s!1-1))
|-----
[ 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 if the FD switch is pressed [-3] during an overspeed condition [-1] and no higher priority event occurs at the same time [-4] and the onside FD cues do not change value [-5] between state  $s!1-1$  and  $s!1$ , then it must be true that the onside FD cues were off before the FD switch was pressed [1] or that the current state is the initial system state [2]. This is impossible to prove, indicating that the property we are trying to prove must be 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] can not occur. This is also 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 [15], 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<sup>c</sup> 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 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

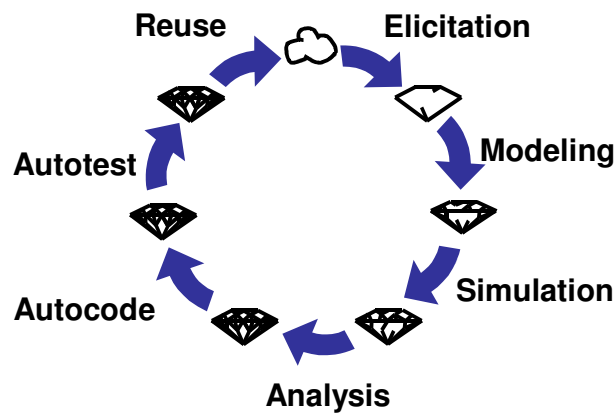
if a result is specified for every set of inputs to an operation (i.e., each operation is a *total* function).

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 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. Figure 1 illustrates a product life cycle paradigm often referred to as model-based development. This approach 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.



**Fig. 1. Model Based Development Process Lifecycle**

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.

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. The observation that shall statements are just informal statements of the system properties suggests that perhaps they aren't such a bad first step. The very commonality of their use 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 experiment, 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.

Finally, 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.



## 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<sup>c</sup> 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<sup>c</sup> model improved the informal requirements, and the process of proving the formal properties found errors in both the original requirements and the RSML<sup>c</sup> model. Our concerns about the difficulty of proving properties in the NuSMV and PVS models that were automatically generated from the RSML<sup>c</sup> 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, many of which are well known and have been proposed by others. We would like to explore translating use cases into sequences of properties than 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. Perhaps most important, these formal verification tools need to be used on real problems with commercially supported modeling tools such as SCADE, Esterel, and Simulink.

**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, the efforts of Mike Whalen, 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, and the insightful suggestions of the anonymous referees.

## References

1. Brooks, F.: No Silver Bullet: Essence and Accidents of Software Engineering, IEEE Computer, April, 1987.
2. Boehm, B.: Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.
3. Davis, A.: Software Requirements (Revised): Object, Functions, and States, Prentice-Hall, Englewood Cliffs, NJ, 1993.
4. van Schouwen, A.: 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.

5. Ramamoorthy, C., Prakesh, A., Tsai W., Usuda, Y.: Software Engineering: Problems and Perspectives, IEEE Computer, pages 191-209, October 1984.
6. Leveson, N.: Safeware: System Safety and Computers, Addison-Wesley Publishing Company: Reading, Massachusetts, 1995.
7. Lutz, R.: Analyzing Software Requirements Errors in Safety-Critical, Embedded, Systems, in IEEE International Symposium on Requirements Engineering, San Diego, CA, January 1993.
8. Heitmeyer, C., Jeffords, R., Labaw, B.: Automated Consistency Checking of Requirements Specification, ACM Transactions on Software Engineering and Methodology (TOSEM), 5(3):231-261, July 1996.
9. Parnas, D., Madey, J.: Functional Documentation for Computer Systems Engineering (Volume 2), Technical Report CRL 237, McMaster University, Hamilton, Ontario, September 1991.
10. Faulk, S., Brackett, J., Ward, P., Kirby, J.: The CoRE Method for Real-Time Requirements, IEEE Software, 9(5):22-33, September 1992.
11. Faulk, S., Finneran, L., Kirby, J., Shah, S., Sutton, J.: Experience Applying the CoRE Method to the Lockheed C-130J Software Requirements, in Proceedings of the Ninth Annual Conference on Computer Assurance, pages 3-8, Gaithersburg, MD, June 1994.
12. Leveson, N., Heimdahl, M., Hildreth, H., Reese, J.: Requirements Specifications for Process-Control Systems, IEEE Transactions on Software Engineering, 20(9):684-707, September 1994.
13. Harel, H., Naamad, A.: The STATEMATE Semantics of Statecharts, ACM Transactions on Software Engineering and Methodology, 5(4): 293-333, October, 1996.
14. Miller, S.: Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR, in Proceedings of The Second Annual Workshop on Formal Methods in Software Practice (FMSP'98), Clearwater Beach, Florida, March 4-5, 1998
15. Butler, R., Miller, S., Potts, J., Carreno, V.: A Formal Methods Approach to the Analysis of Mode Confusion, in Proceedings of the 17th AIAA/IEEE Digital Avionics Systems Conference, Bellevue, WA, October 1998.
16. Miller, S., Tribble, A.: A Methodology for Improving Mode Awareness in Flight Guidance Design, in Proceedings of the 21st Digital Avionics Systems Conference (DASC'02), Irvine, CA, Oct. 2002.
17. Tribble, A., Lempia, D., Miller, S.: Software Safety Analysis of a Flight Guidance System, in Proceedings of the 21st Digital Avionics Systems Conference (DASC'02), Irvine, CA, Oct. 2002.
18. Thompson, J., Heimdahl, M., Miller, S.: Specification Based Prototyping for Embedded Systems, in Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, LNCS, Number 1687, September 1999.
19. Berry, G., Gonthier, G.: The Synchronous Programming Language Esterel: Design, Semantics, and Implementation, Science of Computer Programming, 19:87-152, 1992.
20. Thompson, J., Heimdahl, M., Miller, S.: Specification Based Prototyping for Embedded Systems, in Proceedings of the Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering, LNCS, Number 1687, September 1999.
21. Clarke, E., Grumberg, O., Peled, P.: Model Checking, The MIT Press, Cambridge, Massachusetts, 2001.
22. Anonymous, NuSMV Home Page, <http://nusmv.irst.itc.it/>.
23. Owre, S., Rushby, J., Shankar, N., Henke, F.: Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS, IEEE Transactions on Software Engineering, Vol. 21, No. 2, pg. 107-125, February 1995.
24. Anonymous, PVS Home Page, <http://www.csl.sri.com/projects/pvs/>.

25. Miller, S., Tribble, A., Carlson, T., Danielson, E.: Flight Guidance System Requirements Specification Final Report, NASA Contractor Report, November 2001.
26. Heimdahl, M., Rayadurgam, S., Choi, Y., Joshi, A., Devaraj, G.: Proof and Model Checking Tools Final Report, NASA Contractor Report, November 2002.
27. Tribble, A.: FGS Safety Analysis Final Report, NASA Contractor Report, November 2002.
28. Billings, C.; Aviation Automation: the Search for a Human Centered Approach, Lawrence Erlbaum Associates, Inc., Mahwah, NJ, 1997.
29. Sarter, N., Woods, D.: Pilot Interaction with Cockpit Automation: Operational Experiences with the Flight Management System, *The International Journal of Aviation Psychology*, 2(4), pg. 303-31, 1992.
30. Sarter, N., Woods, D.: 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), pg. 1-28, 1994.
31. Sarter, N., Woods, D.: "How in the World Did I Ever Get Into That Mode?": Mode Error and Awareness in Supervisory Control, *Human Factors*, 37(1), pg. 5-19, 1995.
32. Miller, S.: Taxonomy of Mode Confusion Sources Final Report, NASA Contractor Report, February 2001.
33. Leveson, N., et al, Analyzing Software Specifications for Mode Confusion Potential, in *Proceedings of a Workshop on Human Error and System Development*, C.W. Johnson, Editor, pg. 132-146, Glasgow, Scotland, March 1997.
34. Rushby, J.: Analyzing Cockpit Interfaces Using Formal Methods, *Electronic Notes in Theoretical Computer Science*, 43, URL: <http://www.elsevier.nl/locate/entcs/volume43.html>, 2001.
35. Rushby, J.: Using Model Checking to Help Discover Mode Confusions and Other Automation Surprises, in the *Proceedings of the 3rd Workshop on Human Error, Safety, and System Development (HESSD'99)*, Liege, Belgium, June 7-8, 1999.
36. Rushby, J., Crow, J., Palmer, E.: An Automated Method to Detect Potential Mode Confusion, in the *Proceedings of the 18th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, St. Louis, MO, October 1999.
37. Miller, S., Joshi, A.: FGS Mode Awareness Final Report, NASA Contractor Report, November 2002.