Applying the SCR Requirements Specification Method to Practical Systems: A Case Study[§]

Presented at *The 21st Software Engineering Workshop*
NASA Goddard Space Flight Center, Greenbelt MD, USA Dec. 1996

Ramesh Bharadwaj and Connie Heitmeyer
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC 20375-5320
E-mail: {ramesh,heitmeyer}@itd.nrl.navy.mil

1 Introduction

Studies have shown that the majority of errors in software systems are due to incorrect requirements specifications. The root cause of many requirements errors is the imprecision and ambiguity that arise because the software requirements are expressed in natural language. An effective way to reduce such errors is to express requirements in a formal notation. For a number of years, researchers at the Naval Research Laboratory (NRL) have been working on a formal method based on *tables* to specify the requirements of practical systems [2, 11]. Known as the Software Cost Reduction (SCR) method, this approach was originally formulated to document the requirements of the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft [2]. Since SCR's introduction more than a decade ago, many industrial organizations, including Lockheed, Grumman, and Ontario Hydro, have used SCR to specify requirements. Recently, NRL has developed both a formal state machine model [12, 14] to define the SCR semantics and a set of software tools to support analysis and validation of SCR requirements specifications [10]. The tools support consistency and completeness checking, simulation, and model checking.

To evalute the SCR method and toolset, we recently used SCR to produce a black box requirements specification of a simplified mode control panel for the Boeing 737 autopilot. Beginning with the English language description of the system presented in [4], we represented the environmental quantities that the computer system monitors (e.g., the pilot switches, dials, and sensors) and the environmental quantities that the computer system controls (i.e., the individual displays) as monitored and controlled variables. We then used these variables and the SCR tabular notation to specify the requirements of the mode control panel. The heart of the specification is the relation REQ, the required relation between the monitored and controlled variables [20].

In this paper, we use the autopilot mode control panel as an example for comparing and contrasting the SCR approach to requirements specification and analysis with the approach used in [4]. The latter approach uses the formal language of SRI's Prototype Verification System (PVS)

[§]This work was supported by the Office of Naval Research.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1996		**00-00-1996 to 00-00-1996**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Applying the SCR Requirements Speification Method to Practical Systems: A Case Study	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**15**	

[17] to represent the requirements of the mode control panel and then applies the automated reasoning provided by PVS to analyze the specification. Formulating the requirements specification for the mode control panel in SCR exposed a number of problems, including a missing input event, an incorrect assumption about the environment, and a misinterpretation of the prose description. We also discovered that because parts of the PVS specification are highly abstract, certain key aspects of the system's requirements are omitted. In contrast, the SCR approach makes explicit many important questions about the required behavior of the mode control panel. We conclude with a discussion of general issues such as the appropriate level of abstraction for documenting requirements, the choice of notation, the kinds of analyses that can be done on the specification, the relation between different kinds of analyses, and the role of tool support. Appendix B contains the complete SCR requirements specification of the mode control panel.

2 Motivation and Background

It is widely acknowledged that requirements are a major source of errors during the development of large software systems [1, 9, 16]. For example, studies by Lutz [16] have shown that functional and interface requirements were the source of a majority of safety-related software errors in NASA's Voyager and Galileo spacecrafts. There is no doubt that getting a complete and consistent characterization of software requirements is inherently hard. However, there are failings in the software development process, including the requirements process, that can be rectified by improved practice [8]. A disciplined and rigorous approach to the analysis and specification of software requirements can address many difficulties that result from such failings.

The goal of the requirements phase is to create a document, the Software Requirements Specification (SRS), to precisely describe the problem to be solved and to accurately characterize the set of acceptable solutions to the problem. The effectiveness of the requirements phase is determined by the extent to which the SRS is precise, unambiguous and consistent (i.e., its correctness), whether it captures all the results of the analysis (i.e., its completeness), and its usability. The usability criteria are ease of change (i.e., its modifiability), whether the notation is understandable both by customers as well as the developers (i.e., its readability), its organization for easy reference and review (for instance, one should quickly be able to find answers to specific questions about the requirements), and organization for ease of change. In addition, the underlying conceptual model and notation of the SRS should support formal analyses such as *validation* (to ensure that the specification describes the intended requirements), and *verification* (which establishes that the specification satisfies critical properties of interest). Finally, the method should provide guidelines that support decisions on organization and modification of the SRS. By sufficiently constraining the underlying semantic model, these guidelines ensure that the quality of the SRS does not depend too much on the level of expertise of its writer(s).

2.1 The SCR Method

Unlike traditional research on requirements, which concentrates on the *requirements analysis process*, the focus of the SCR work at the Naval Research Laboratory is on issues that influence the creation and maintenance of the SRS. By identifying desirable properties of an SRS, the SCR

project has developed a set of guidelines for writing the SRS [11, 8]. These guidelines include *separation of concerns*, *information hiding*, and the use of a readable yet *formal notation*. For example, the guideline *separation of concerns* supports usability, modifiability, and verifiability of the SRS. Moreover, the notation supported by the SCR method is designed to be understandable both by customers as well as software developers. Underlying the notation is a mathematical model which supports completeness and consistency checking, validation, test case generation, and formal verification.

To support the SCR method, NRL has developed a set of software tools for analysis and validation of SCR requirements specifications [10, 13]. The tools include a *specification editor* for creating and modifying the specifications, a *simulator* for symbolic execution, and tools for formal analysis. The latter include a *consistency checker* which uncovers application-independent errors such as syntax and type errors, missing cases, and unwanted nondeterminism, and a *verifier* which checks a specification for critical application-specific properties.

2.2 PVS

PVS (Prototype Verification System) [17] is an environment for specification and verification developed at SRI International. The PVS system is built around a highly expressive specification language. The system has a number of predefined theories, and comes with a very effective interactive theorem prover in which most of the low-level proof steps are automated. The PVS specification language is based on higher-order logic with a richly expressive type system. The PVS prover consists of a powerful collection of inference steps which include arithmetic and equality decision procedures, automatic rewriting, and boolean simiplification. PVS has been applied to a number of practical problems [4, 5, 21]. Many organizations, including NASA, have used the PVS specification language for documenting software requirements.

3 Comparison of PVS with the SCR method

In this section, we address some of the strengths and limitations of using PVS, and compare the PVS approach to the SCR method. We base our comparison on the assumption that a notation (and associated tools) should support the following process, which may be thought of as an idealization of a real-world process for requirements analysis [19].

1. **SRS Creation:** The results of problem analysis are captured in the SRS, using a formal notation.

2. **SRS Checking:** The SRS is checked for proper syntax, type correctness, consistency, completeness, and other application-independent properties, using an automated checker.

3. **SRS Validation:** The goal of this phase is to ensure that the SRS captures the customers' intent. This is achieved by symbolically executing the SRS using a simulator.

4. **SRS Verification:** This phase verifies that certain crucial application specific properties, such as safety and security properties, hold for the SRS. Verification is carried out by using an interactive theorem prover or by "lightweight" analysis tools such as model checkers.

3.1 SRS Creation

The choice of notation, and availability of guidelines to support decisions on SRS organization and modification, are factors which influence this phase. A simpler, more restrictive notation is preferable to a more powerful, expressive one. In addition to ease of use, a restricted semantic model can provide guidelines for creating and organizing the SRS. A well-designed notation will help even novices create good specifications.

The PVS system is built around a highly expressive specification language. However, most developers, being unfamiliar with higher-order logic (the underlying formalism of the PVS specification notation), lambda expressions, higher-order functions and quantification, etc, find the notation hard to use. It has also been our experience that the expressive power of higher-order logic is seldom required for requirements specification of most practical systems. The organizing unit for PVS specifications is the "Theory". The PVS language lacks structures to support readability and ease of change. It is very hard for novices to create good PVS specifications. For example, it has been observed by Young [22] that the quality of specifications in PVS depends to a large extent on the expertise of the specification writer.

The SCR method is suitable for embedded, real-time systems, i.e., for systems that sense and control quantities in their environment [20]. The SCR method includes a systematic approach for capturing requirements [11, 15, 6], and is based on a tabular notation which has a formal mathematical basis [12, 13, 14]. The SCR notation, having been tailored to a specific class of problems, sacrifices generality for ease of use and improved support for analysis. Most engineers find the tabular notation easy to use and understand. Also, tables afford a natural organization which permits independent construction, review, modification, and analysis of smaller parts of a large requirements specification.

It has been observed that in comparison to graphical notations and (structured) text, tabular notations scale very well to large problems. According to Parnas, the specification of the shutdown system for the Darlington Nuclear Power Plant [18] weighed more than 20 kilograms on paper. In our own experience, we have come across examples of SCR requirements specifications for practical systems (e.g., the OFP for the C-130J aircraft [7]) containing more than a thousand tables.

3.2 SRS Checking

In addition to checks for incorrect syntax, the PVS language has a rich type system which supports rigorous typechecking. The type system of PVS is undecidable, which means that typechecking cannot be completely automated. In most situations, the PVS typechecker will generate proof obligations which have to be proved using the interactive prover. Such proofs amount to a very strong consistency check on some aspects of the specification.

The consistency and completeness checker of the SCR toolset verifies application-independent properties derived automatically from the requirements model. These checks ensure that a specification is well-formed by identifying syntax and type errors, incompleteness, missing initial values, unreachable modes, and circular definitions. The tool also identifies missing cases and undesirable nondeterminism. All these checks are carried out automatically.

3.3 SRS Validation

PVS does not support validation.

The tabular notation of SCR supports validation by inspection and simulation. Most domain experts find this notation easy to read and review. For example, Parnas [18] observes that the utility of the tabular notation was evident during the formal review of the Darlington specification. During the review, each "case" and its associated subcases could be reviewed individually and independently of other "cases". The tabular notation also forces one to consider all possible scenarios. Further, we show in [3] that theorems that are true of certain fragments of an SCR requirements specification also hold for the whole specification.

The simulator in the SCR toolset performs symbolic execution of the underlying state machine model, which allows users to assess system behavior in specific "use cases" directly from the requirements specification. The simulator can expose problems — such as missing requirements and incorrectly stated requirements — that cannot be detected by verification techniques.

3.4 SRS Verification

Using PVS, one can establish, by interactive theorem proving, properties that are deemed to be true of a requirements specification. However, few practitioners have the mathematical sophistication required to carry out such proofs. The state-of-the-art theorem prover of PVS does ameliorate the problem by including powerful decision procedures that automate parts of a proof that would otherwise require user guidance. Very often, a property *will not* hold for a requirements specification. In such a case, either the formulation of the property is incorrect, or the specification is wrong (or both). Proper diagnosis and user feedback are therefore very important to help correct the problem. Theorem provers provide very little help in such situations because theorem proving is incomplete; i.e., if one is unable to prove a theorem using a theorem prover, then all one can conclude is that the theorem prover failed to find a proof (the theorem may be true). On the other hand, methods such as model checking are complete — if a model checker reports that a theorem is false, it is false. Additionally, most model checkers will provide a *counterexample* that falsifies the theorem. PVS does support model checking for a limited subset of the language, but provides no counterexample.

The SCR toolset supports proof of safety properties of a requirements specification using state exploration based model checking [3]. One of the main design goals of our toolset is to provide proper error diagnosis by generating understandable counterexamples for user feedback. Future plans include support for other forms of model checking and automatic theorem proving. Since the underlying model of the SCR notation is a state machine, several other verification activities can be supported. For instance, we plan to automatically generate test-cases from an SCR specification, to assist in black-box testing of implementations. In certain limited contexts, it should also be possible to automatically generate code directly from an SCR requirements specification.

4 The Autopilot Requirements Specification

To illustrate the SCR method, we consider a simplified mode control panel for the Boeing 737 autopilot as discussed in [4]. The mode control panel for the autopilot is shown in *Figure 1*.

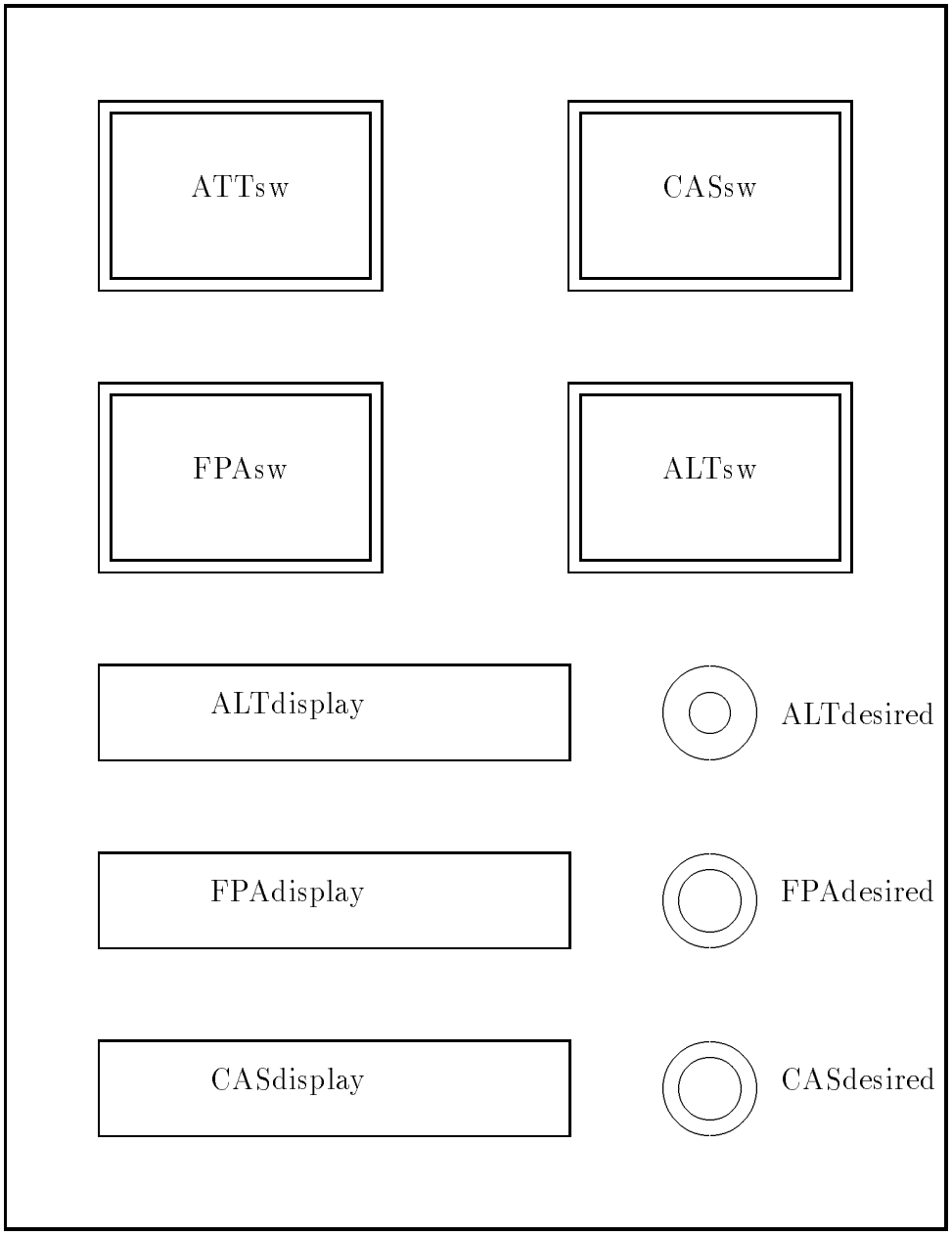

Figure 1: Mode Control Panel

The system monitors the aircraft's altitude (ALT), flight path angle (FPA) and calibrated air speed (CAS). The panel includes three displays which show the current values for altitude, flight path angle, and airspeed of the aircraft. The pilot may enter a new value into a display by "dialing-in" the value using one of three knobs next to the displays. The pilot engages or disengages the autopilot by pressing one of four buttons on the panel. Appendix A contains a description of the system in English prose (adapted from [4]). Below, we informally present the steps taken to document the requirements using the SCR notation.

In SCR, the required system behavior is described by REQ, the required relation between *monitored variables*, environmental quantities that the system monitors, and *controlled variables*, environmental quantities that the system controls [20]. To specify this relation concisely, the SCR approach uses four constructs – modes, terms, conditions, and events. A *mode class* is a variable whose values are *system modes* (or simply *modes*), while a *term* is any function of monitored variables, modes, or other terms. A *condition* is a predicate defined on one or more system entities (an *entity* is a monitored or controlled variable, mode class, or term). An *event* occurs when the value of any system entity changes. The notation "`@T(c) WHEN d`" denotes a *conditioned event*, defined as

$$\texttt{@T(c) WHEN d} \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed condition c is evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. The notation "`@F(c)`" denotes the event `@T(NOT c)`. The environment may change a monitored quantity, causing an *input event*. In response, the system changes controlled quantities and updates terms and mode classes.

We begin by identifying the monitored quantities, i.e., the environmental quantities that the autopilot system monitors, and denote them by corresponding *monitored variables*. We use the prefix "`m`" for all monitored variable names. Each monitored variable is of a certain *type*, which specifies the range of values that may be assigned to that variable. The autopilot system monitors the actual altitude (denoted by monitored variable `mALTactual`), the actual flight path angle (`mFPAactual`), and the actual calibrated air speed (`mCASactual`). We assume these variables to range over the integers. Switches `ALTsw`, `ATTsw`, `CASsw`, and `FPAsw` are denoted respectively by `mALTsw`, `mATTsw`, `mCASsw`, and `mFPAsw`. These monitored variables may take on one of the values from the set {`on`, `off`}. Finally, knobs `ALTdesired`, `CASdesired`, and `FPAdesired` are denoted by monitored variables `mALTdesired`, `mCASdesired`, and `mFPAdesired` respectively, which range over the integers.

We then identify the controlled quantities, i.e., the environmental quantities that the autopilot system controls, and denote them by corresponding *controlled variables*. We use the prefix "`c`" for all controlled variable names. Just as for monitored variables, we assign a *type* to each controlled variable. For simplicity of exposition we shall, as in [4], only model the mode-control panel itself, and not the commands that will be sent out to the flight-control computer. The three controlled quantities of the mode control panel are `ALTdisplay`, `FPAdisplay`, and `CASdisplay`, which we denote respectively by `cALTdisplay`, `cFPAdisplay`, and `cCASdisplay`. We assume these values to range over the integers.

We model the primary modes of the mode-control panel by the *modeclass* `Status`, denoted by variable `mcStatus`. The variable can take on any value in the set {`ALTmode`, `ATTmode`, `FPAmode`}. The altitude engaged mode being "armed" is denoted by a boolean *term* variable `tARMED` (we use

the prefix "t" for terms). If tARMED is *true*, then mcStatus should be FPAmode. The previous sentence is an example of a property of the specification which we may later want to prove. We also define a boolean valued *term* tCASmode, to model the system being in the *calibrated air speed* mode. By describing the status of the mode-control panel in this manner, we have ensured that the following sentences in the prose requirements are trivially satisfied:

1. Only one of the three modes ALTmode, ATTmode, or FPAmode can be engaged at any time.
2. One of the three modes, ATTmode, FPAmode, or ALTmode should be engaged at all times.
3. Engaging any of the three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.
4. The mode CASmode can be engaged at the same time as any of the other modes.

We define three boolean valued terms tALTpresel, tCASpresel, and tFPApresel to denote whether the corresponding quantity has been pre-selected by dialing in a new value using one of the three knobs. Finally, we define a boolean term tNear to denote the predicate mALTdesired − mALTactual ≤ 1200.

The behavior of mode class mcStatus is specified in a *mode transition table*. In the following, the expression CHANGED(x) denotes the event "variable x has changed". The table defines all events that change the value of the mode class mcStatus. For example, the first row of the table states, "If mcStatus is ALTmode, and mATTsw is switched on, or the setting of knob mALTdesired is changed, then mcStatus changes to ATTmode." Events that do not change the value of the mode class are omitted from the table.

Source Mode	Events	Destination Mode
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mFPAsw = on) OR @T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAmode
FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear) OR @T(tNear) WHEN tARMED	ALTmode
FPAmode	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) WHEN tARMED	ATTmode

Each row in the mode transition table above corresponds to certain sentences in the prose requirements. We describe this correspondence below. Here, "paragraph x" refers to the numbered paragraph x of the prose requirements in Appendix A.

Row 1. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., pressing ATTsw should engage ATTmode **OR** *If the pilot dials in a new altitude while* ALTmode *is engaged, then* ALTmode *is disengaged and* ATTmode *is engaged* (paragraph 7).

Row 2. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., by pressing FPAsw the pilot engages FPAmode.

Row 3. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., pressing `ALTsw` engages `ALTmode`. *However, the altitude must be pre-selected before* `ALTsw` *is pressed* (paragraph 4). *If the pilot dials an altitude that is more than* 1,200 *feet above* `ALTactual` *and then presses* `ALTsw`, *then* `ALTmode` *will not directly engage* (paragraph 3).

Row 4. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., by pressing `FPAsw` the pilot engages `FPAmode` **OR** *If the pilot dials into* `ALTdesired` *an altitude that is more than* 1,200 *feet above* `ALTactual` *and then presses* `ALTsw`, *then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged* (paragraph 3).

Row 5. The situation described for row (3) above **OR** *Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged. [...]* `FPAmode` *will remain engaged until the aircraft is within* 1,200 *feet of* `ALTactual`, *then* `ALTmode` *is automatically engaged* (paragraph 3).

Row 6. *The pilot engages a mode by pressing the corresponding button on the panel* (paragraph 1) i.e., by pressing `mATTsw` the system enters `ATTmode` **OR** `FPAsw` *toggles on and off every time it is pressed.* (paragraph 5) **OR** *If the pilot dials in a new altitude while the altitude engage mode is "armed" then* `ATTmode` *is engaged. [...]* `FPAmode` *should be disengaged as well.* (paragraph 7).

The behavior of term `tARMED` is specified in the *event table* below. Like mode transition tables, event tables make explicit only those events that cause the variable defined by the table to change. For example, the first entry in the first row states, "If `mcStatus` is `ATTmode` *or* `FPAmode` and `mALTsw` is turned `on` when `tALTpresel` is *true* and `tNear` is *false*, then `tARMED` becomes *true*." The entry "`NEVER`" in an event table means that no event can cause the variable defined by the table to assume the value in the same column as the entry; thus, the entry "`NEVER`" in row 2 of the table means that when `mcStatus` is `ALTmode` no event can cause `tARMED` to become *true*. An entry "`@T(Inmode)`" in a row of a mode transition table or an event table denotes the event "system entered the corresponding mode".

Modes	Events	
ATTmode, FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	@F(mcStatus = FPAmode)
ALTmode	NEVER	@F(mcStatus = FPAmode)
tARMED =	true	false

We finally present the behavior of the display `cCASdisplay` using the *condition table* below. This table states that "If `tCASpresel` is *true* then `cCASdisplay` has the value `mCASdesired`; otherwise, it has the value `mCASactual`". The complete autopilot specification is in Appendix B.

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCASactual

5 Discussion of General Issues

In [3] we present a verification technique for proving properties of SCR requirements specifications. This technique proved to be valuable in detecting and correcting bugs in the autopilot specification. For example, an initial formulation of the specification violated the property "*the altitude engage mode will be* `ARMED` *only when the flight path angle select mode is engaged*". The counterexample generated by the tool helped diagnose the error (we were setting `tARMED` to *true* when `mcStatus` is `ALTmode`, and `mALTsw` is turned `on` when `tALTpresel` is *true* and `tNear` is *false*).

We found that the PVS model does not clearly distinguish a system's environmental quantities from the dependent quantities. Also, by not clearly identifying environmental quantities the system monitors, and environmental quantities the system controls, it was very hard to find an answer to the question "What is the required behavior of the system?" by examining the PVS model. During the process of creating the SCR requirements specification, we came up with several questions for which we could not find answers from the PVS model. This is because the PVS description is not at the appropriate level of abstraction.

5.1 Appropriate Level of Abstraction

The PVS model of the autopilot in [4] is too *abstract* to serve as a requirements specification, i.e., as a black box description of all acceptable system implementations. Rather than specifying the required relationship between environmental quantities of the autopilot mode control panel, the PVS description is an abstract model of the mode control panel. Therefore, it is not a requirements specification. For example, the monitored quantity `ALTactual` is denoted abstractly by two boolean variables `alt_reached` and `alt_gets_near`; boolean variable `input_alt` abstractly denotes the pilot "dialing-in" the desired altitude using knob `ALTdesired`; etc. It is usual to make such abstractions during verification, because existing methods cannot be directly applied to requirements specifications, which are too detailed. However, the right approach is to begin by formulating the requirements specification, and later to describe formally the relationship between the specification and the abstract verification models. If the correspondence between the abstract models and the requirements specification is informal (or if the requirements specification is never created), it leaves room for misinterpretation.

5.2 Kinds of Analyses

In our experience, the first three phases of our idealized process for requirements analysis, viz., SRS Creation, SRS Checking, and SRS Validation, are the most crucial ones. It is very likely that a large proportion of activities of requirements analysis will be in support of these phases. It is also safe to assume that for a majority of projects (barring a small number of projects developing safety or mission critical applications) the last phase, i.e., SRS Verification, will be completely skipped. Since PVS concentrates exclusively on this phase of analysis, and provides poor support for the initial three phases, it is unlikely to be very effective as a tool to support requirements analysis. However, PVS has been effective in the analysis of critical algorithms and architectures for fault-tolerance, such as the correctness of distributed agreement protocols for a hybrid fault model, and in the verification of crucial subsystems, such as a commercial avionics microprocessor.

5.3 Role of Tool Support

In our experience, tools that support a limited analysis domain, with a specific conceptual model, tend to be more effective than general purpose tools. If a method lacks a strong underlying conceptual model, the benefits of automation are likely to be minimal ([8] provides more details). If a method does not adequately constrain the problem, the corresponding support tools cannot guide the developer when making difficult decisions. Since the SCR method standardizes the problem domain, the conceptual model, the notation, and the process, significant automated tool support is possible. For example, by using information about the current state of a specification, and knowledge of the process, a tool can guide developers in making the next step. Also, by providing standard templates, a tool can automate the routine activities of SRS creation. By applying the SCR method to several industrial problems, we plan to exploit the full potential of such tools.

6 Acknowledgements

We thank Jim Kirby for many helpful discussions on the autopilot specification. We gratefully acknowledge Ricky Butler for providing helpful insights and for his prompt answers to all our questions about the autopilot mode control panel.

References

[1] M. Alford. Software Requirements Engineering Methodology (Development). *RADC-TR-79-168*, U.S. Air Force Rome Air Development Center, June 1979.

[2] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. Software Requirements for the A-7E Aircraft. Technical Report NRL-9194, NRL, Washington DC, 1992.

[3] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. Submitted for publication.

[4] Ricky W. Butler. An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot. NASA Technical Memorandum 110255. NASA Langley Research Center, May 1996.

[5] B. L. DiVito and L. W. Roberts. Using Formal Methods to Assist in the Requirements Analysis of the Space Shuttle GPS Change Request. NASA Contractor Report 4752. NASA Langley Research Center, Hampton VA 23681, August 1996.

[6] S. R. Faulk, et al. The CoRE method for real-time requirements. *IEEE Software*, 9(5), September 1992.

[7] S. R. Faulk, et al. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conference on Computer Assurance,* Gaithersburg MD, June 1994.

[8] S. R. Faulk. Software Requirements: A Tutorial. Technical Report *NRL/MR/5546–95-7775*, Naval Research Laboratory, Washington DC, 1995.

[9] General Accounting Office (US). Mission Critical Systems: Defense Attempting to Address Major Software Challenges. *GAO/IMTEC-93-12*, December 1992.

[10] Constance Heitmeyer, et al. SCR*: A toolset for specifying and analyzing requirements. In *Proc.* 10^{th} *Annual Conference on Computer Assurance*, NIST, Gaithersburg MD, June 1995.

[11] K. L. Heninger. "Specifying software requirements for complex systems: New techniques and their applications". *IEEE Transactions on Software Engineering* SE-6(1), Jan 1980.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for Analyzing SCR-style Requirements Specifications: A Formal Foundation. Technical Report NRL-7499, NRL, Wash. DC, 1995. In preparation.

[13] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Trans. on Software Engg. and Methodology*, 5(3)231–261, July 1996.

[14] Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. Consistency checking of SCR-style requirements specifications. In *Proc. 1995 Int'l Symposium on Requirements Engg.*, York, England, March 1995.

[15] C. L. Heitmeyer and J. McLean. "Abstract requirements specifications: A new approach and its application". *IEEE Transactions on Software Engineering*, SE-9(5), Sep 1983.

[16] R. Lutz. Analyzing software requirements errors in safety-critical embedded systems. In *Proc. IEEE Int'l Symp. on Requirements Engg.*, pp. 126–133, Jan 1993.

[17] Sam Owre, John Rushby, and Natarajan Shankar. PVS: A prototype verification system. In 11^{th} *International Conference on Automated Deduction*, LNCS-607, pp 748–752, 1992.

[18] D. L. Parnas, G. J. K. Asmis and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety,* 32(2), 1991.

[19] D. L. Parnas and P. Clements. A rational design process: how and why to fake it. *IEEE Trans. on Software Engg.*, 12(2), February 1986.

[20] D. L. Parnas and J. Madey. Functional documents for computer systems. *Science of Computer Programming,* 25(1), pp 41–62, Oct 1995.

[21] M. K. Srivas and S. P. Miller. Formal Verification of an Avionics Microprocessor. NASA Contractor Report 4682, NASA Langley Research Center, July 1995.

[22] W. D. Young. Comparing verification systems: interactive consistency in ACL2. In *Proc. COMPASS'96*, Gaithersburg MD, 1996.

A Description of the autopilot

1. *The mode-control panel contains four buttons for selecting modes and three displays for dialing in or displaying values, as shown in Figure 1. The system supports the following four modes: attitude control wheel steering (*`ATTmode`*), flight path angle selected (*`FPAmode`*), altitude engage (*`ALTmode`*), and calibrated air speed (*`CASmode`*).*

   Only one of the first three modes can be engaged at any time. The mode `CASmode` *can be engaged at the same time as any of the other modes. The pilot engages a mode by pressing the corresponding button on the panel. One of the three modes,* `ATTmode`, `FPAmode`, *or* `ALTmode` *should be engaged at all times. Engaging any of the first three modes will automatically cause the other two to be disengaged since only one of these three modes can be engaged at a time.*

2. *There are three displays on the panel: altitude (*`ALTdisplay`*), flight path angle (*`FPAdisplay`*), and calibrated air speed (*`CASdisplay`*). The displays usually show the current values of altitude (*`ALTactual`*), flight path angle (*`FPAactual`*), and air speed (*`CASactual`*) of the aircraft. However, the pilot can enter a new value into a display by dialing in the value using the knob next to the display (*`ALTdesired`, `FPAdesired`, *or* `CASdesired`*). This is the target or "pre-selected" value that the pilot wishes the aircraft to attain. For example, if the pilot wishes to climb to* $25,000$ *feet, he will dial* $25,000$ *(using the knob* `ALTdesired`*) into* `ALTdisplay` *and then press* `ALTsw` *to engage* `ALTmode`*. Once the target value is achieved or the mode is disengaged, the display reverts to showing the "current" value.*

3. *If the pilot dials into* `ALTdesired` *an altitude that is more than* $1,200$ *feet above the current altitude (*`ALTactual`*) and then presses* `ALTsw`*, then* `ALTmode` *will not directly engage. Instead, the altitude engage mode will change to "armed" and* `FPAmode` *is engaged. The pilot must then dial in, using the knob* `FPAdesired`*, the desired flight-path angle into* `FPAdisplay`*, which will be followed by the flight-control system until the aircraft attains the desired altitude.* `FPAmode` *will remain engaged until the aircraft is within* $1,200$ *feet of* `ALTactual`*, then* `ALTmode` *is automatically engaged.*

4. `CASdesired` *and* `FPAdesired` *need not be pre-selected before the corresponding modes are engaged — the current values displayed will be used. The pilot can dial-in a different target value after the mode is engaged. However, the altitude must be pre-selected before* `ALTsw` *is pressed. Otherwise, the command is ignored.*

5. `CASsw` *and* `FPAsw` *toggle on and off every time they are pressed. For example, if* `CASsw` *is pressed while the system is already in* `CASmode`*, that mode will be disengaged. However, if* `ATTsw` *is pressed while* `ATTmode` *is already engaged, the command is ignored. Likewise, pressing* `ALTsw` *while the system is already in* `ALTmode` *has no effect.*

6. *Whenever a mode other than* `CASmode` *is engaged, all other pre-selected displays should return to current.*

7. *If the pilot dials in a new altitude while* `ALTmode` *is engaged or the altitude engage mode is "armed", then* `ALTmode` *is disengaged and* `ATTmode` *is engaged. If the altitude engage mode is "armed" then* `FPAmode` *should be disengaged as well.*

B SCR Specification of the autopilot

Monitored Variables:

mALTactual, mCASactual, mFPAactual : Integer **initially** all 0;
mALTsw, mATTsw, mCASsw, mFPAsw : {on, off} **initially** all off;
mALTdesired, mCASdesired, mFPAdesired : Integer **initially** all 0;

Controlled Variables:

cALTdisplay, cCASdisplay, cFPAdisplay : Integer **initially** all 0;

Mode Class:

mcStatus : {ALTmode, ATTmode, FPAmode} **initially** ATTmode;

Terms:

tARMED : Boolean **initially** false;
tCASmode : Boolean **initially** false;
tALTpresel, tCASpresel, tFPApresel : Boolean **initially** all false;
tNear $\stackrel{\text{def}}{=}$ mALTdesired − mALTactual $\leq$ 1200;

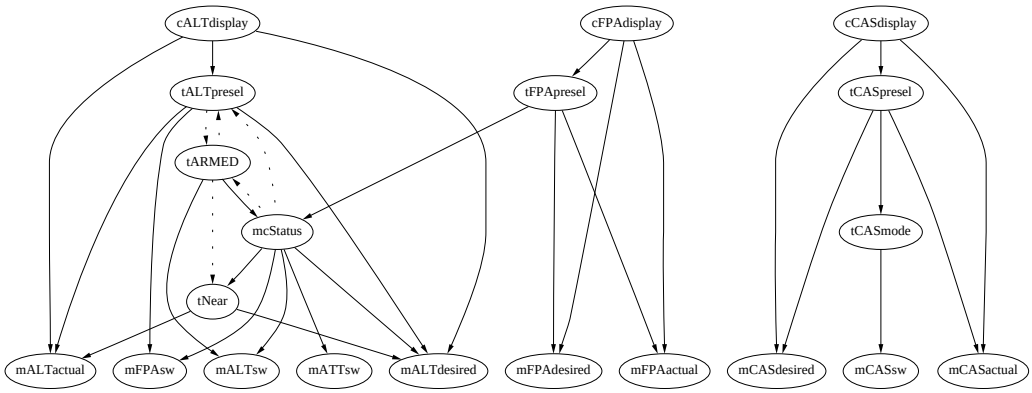

Figure 2: Variable Dependency Graph

Mode Transition Table for mcStatus		
Source Mode	Events	Destination Mode
ALTmode	@T(mATTsw = on) OR CHANGED(mALTdesired)	ATTmode
ALTmode	@T(mFPAsw = on)	FPAmode
ATTmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear)	ALTmode
ATTmode	@T(mFPAsw = on) OR @T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	FPAmode
FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND tNear) OR @T(tNear) WHEN tARMED	ALTmode
FPAmode	@T(mATTsw = on) OR @T(mFPAsw = on) OR CHANGED(mALTdesired) WHEN tARMED	ATTmode

Modes	Events	
ATTmode, FPAmode	@T(mALTsw = on) WHEN (tALTpresel AND NOT tNear)	@F(mcStatus = FPAmode)
ALTmode	NEVER	@F(mcStatus = FPAmode)
tARMED =	true	false

	Events	
	@T(mCASsw = on) WHEN NOT tCASmode	@T(mCASsw = on) WHEN tCASmode
tCASmode =	true	false

Modes	Events	
ALTmode	NEVER	@T(mALTdesired = mALTactual) OR @F(INMODE)
FPAmode	CHANGED(mALTdesired) WHEN NOT tARMED	NEVER
ATTmode	CHANGED(mALTdesired)	@T(INMODE) OR @T(mFPAsw = on)
tALTpresel =	true	false

	Events	
	CHANGED(mCASdesired)	@F(tCASmode) OR @T(mCASdesired = mCASactual) WHEN tCASmode
tCASpresel =	true	false

	Events	
	CHANGED(mFPAdesired)	@T(mcStatus = ATTmode) OR @T(mcStatus = ALTmode) OR @T(mFPAdesired = mFPAactual) WHEN (mcStatus = FPAmode)
tFPApresel =	true	false

	Conditions	
	tALTpresel	NOT tALTpresel
cALTdisplay =	mALTdesired	mALTactual

	Conditions	
	tCASpresel	NOT tCASpresel
cCASdisplay =	mCASdesired	mCASactual

	Conditions	
	tFPApresel	NOT tFPApresel
cFPAdisplay =	mFPAdesired	mFPAactual