Applying a Formal Requirements Method to Three NASA Systems: Lessons Learned

Constance L. Heitmeyer and Ralph D. Jeffords
Naval Research Laboratory (Code 5546)
Washington, DC 20375, USA
{heitmeyer, jeffords}@itd.nrl.navy.mil

Abstract— Recently, a formal requirements method called SCR (Software Cost Reduction) was used to specify software requirements of mission-critical components of three NASA systems. The components included a fault protection engine, which determines how a spacecraft should respond to a detected fault; a fault detection, isolation and recovery component, which, in response to an undesirable event, outputs a failure notification and raises one or more alarms; and a display system, which allows a space crew to monitor and control on-orbit scientific experiments. This paper demonstrates how significant and complex requirements of one of the components can be translated into an SCR specification and describes the errors detected when the authors formulated the requirements in SCR. It also discusses lessons learned in using formal methods to document the software requirements of the three components. Based on the authors' experiences, the paper presents several recommendations for improving the quality of requirements specifications of safety-critical aerospace software.

1. Introduction

In a landmark article published in 1987 [4], Fred Brooks states that

The hardest single part of building a software system is deciding what the requirements are . . . No other part of the work so cripples the resulting system if done wrong . . . [or] is as difficult to produce and hard to fix later on.

Almost twenty years later, eliciting, representing, and organizing requirements remains one of the most challenging problems in software development. Recently, a group of internationally known software experts met at a workshop, sponsored by NSF and the EU, whose goal was to define the most important problems in software development. One of the most difficult problems discussed at the workshop was that of eliciting and representing software requirements.

One method that has been formulated to specify the required externally visible behavior of safety-critical and mission-critical software systems is the SCR (Software Cost Reduction) requirements method. Formulated in the late 1970s to specify the requirements of the Operational Flight Program of the U.S. Navy's A-7 aircraft, the SCR method was designed to improve the quality of software requirements by documenting them in a manner that is unambiguous, precise, and readable. During the 1980s and the early 1990s, many companies, including Bell Laboratories, Grumman, Ontario Hydro, and Lockheed, used the SCR method to develop real-world systems. However, the application of SCR was limited because either no tools or only weak tools were available to support the method.

To provide powerful, robust tool support customized for SCR, NRL began developing the SCR toolset in 1994. The current toolset includes an editor for constructing the specification, a dependency graph browser for displaying variable dependencies, a consistency checker to automatically detect well-formedness errors (such as missing cases), a simulator for validating the specification, an invariant generator for deriving invariant properties from the specification [17], a model checker and a theorem prover for checking application properties [10], an automatic code generator for constructing efficient source code from the SCR specification [20], and an automatic test case generator for constructing tests from the SCR specification [7].

To provide formal underpinnings for the method, NRL has developed a formal model which defines the semantics of SCR requirements specifications [12]. SCR's underlying computational model is a synchronous variation of the classical state machine. Because SCR specifications are expressed in a user-friendly tabular notation rather than more complex notations (e.g., a higher-order or temporal logic) or a specialized computational model (such as CSP), SCR's start-up cost is lower than that of many other "formal methods" tools. Moreover, applying SCR enables detection and removal of requirements errors early in the software lifecycle when errors are much cheaper to fix than errors detected later, for example, during testing. Another benefit is that the SCR method and tools can help developers construct a specification that is unambiguous, concise, readable, and organized as a reference document. Such a specification facilitates both human and mechanical error detection.

Currently, several sites of Lockheed Martin are using the SCR tools to specify and analyze a number of avionics functions, such as flight navigation, flight control and management, and airborne traffic and collision avoidance. Many of these sites

IEEEAC paper # 1601

use the SCR tools in conjunction with an automatic, test case generator called T-VEC [3]. Lockheed is also using the SCR tools in developing software for the Joint Strike Fighter.

The utility of formal methods and their support tools in analyzing SCR and other formally represented requirements of safety-critical systems has been described in many previous papers and reports. For example, in 1998, Easterbrook et al. described the utility of formal tools, including SCR and PVS, for detecting ambiguity, missing assumptions, and other defects in requirements specifications of spacecraft fault protection systems [5]. Also in 1998, our group described the use of model checking to detect a serious defect in the contractor specification of a safety-critical military system [13]. More recently, we described how both the TAME front-end to PVS and SCR were used in the formal specification and verification of a security-critical embedded system as part of a Common Criteria evaluation [11].

Rather than focus on the utility of advanced verification techniques, this paper describes the construction of a formal requirements specification in SCR and how practitioners can produce such a specification. Once available, a formal specification can be analyzed with both light-weight tools, such as the SCR consistency checker, or with advanced verification tools, such as model checkers and theorem provers, to weed out errors, and thus a high-quality requirements specification can be constructed.

In most previous efforts, formal methods experts have formulated the requirements specifications, and, in most cases, have discarded these specifications after the analysis tools were applied. Unlike these earlier efforts, our goal is to produce a requirements specification that is useful, not only for purposes of formal verification, but throughout the system life-cycle. Such a requirements specification should be the basis for both software design and implementation. In addition, when the requirements change, either because the original requirements were incorrect or because the required system behavior needs to be modified or extended, the original requirements specification should be revised to reflect the needed changes. Once the system has been deployed, the requirements specification can be used as the basis for developing new versions of the system—many parts of the original specification should be reusable without change or with minor changes in the new requirements specification.

A major difference between our approach and previous approaches is that our goal is to teach practitioners the skills needed to produce a rigorous software requirements specification. Hence, although for all three projects, the interest of NASA personnel was in the SCR tools and their utility for improving NASA's software development process, an important question when we began these three independent efforts was whether software developers could learn to produce an acceptable requirements specification using the SCR method. This paper presents a preliminary answer to this question.

The paper is organized as follows. Section 2 introduces the three NASA systems. Section 3 presents a brief review of the SCR method and tools. Section 4 presents excerpts from the SCR specification of the requirements of one of the NASA systems, the FDIR (Fault Detection, Isolation and Recovery) software module, and describes two problems detected in the requirements document for this module using SCR. Section 5 describes the lessons learned in applying SCR to the three systems, Section 6 presents several recommendations, and Section 7 presents some concluding remarks.

2. THREE NASA SYSTEMS

Recently, the SCR requirements method was applied to software components of three NASA systems: a Fault Protection Engine, which detects faults in the operation of a spacecraft and determines how the system should respond to each fault; a Fault Detection, Isolation and Recovery system, which, in response to some undesirable event, outputs a failure notification and raises one or more alarms; and an Incubator Display, which allows a space crew to monitor and control on-orbit science experiments. This section briefly describes each component and the plan for applying the SCR method and tools to that component. In each case, we developed, in conjunction with NASA personnel, an SCR specification of a complex portion of the required behavior of the component. The effort we expended in producing the SCR specification was quite modest: less than one week in the case of FDIR, less than two weeks in the case of the Incubator Display, and about three weeks in the case of the FPE.

Fault Protection Engine

The function of the Fault Protection Engine (FPE), the most complex component of the Deep Impact Fault Protection System, is to detect faults in a spacecraft's software and hardware and to schedule, coordinate, and track responses to those faults [6]. The FPE uses a relatively complex algorithm to decide how the system is required to respond to a fault, e.g., when it is responding to a previously detected fault. The objective of applying SCR to the FPE was to construct a complete, formal SCR specification of the required external behavior of the FPE and, based on the specification, to automatically generate a set of test cases satisfying a given coverage criterion using the technique described in [7]. The overall goal of the project was to assess the utility of the generated test cases for evaluating a software implementation of the FPE. Figure 1 illustrates the FPE's required behavior.

Fault Detection, Isolation and Recovery

The Fault Detection, Isolation, and Recovery (FDIR) module is part of the Thermal Radiator Rotary Joint Manager (TRRJ_M) software, whose function is to position the thermal radiators of the International Space Station in an on-edge orientation to the sun to maximize heat dissipation from the radiators. FDIR processing for the TRRJ_M subsystem is safety-critical because failure to detect, isolate, and recover from faults could lead to serious damage to the radiators. The

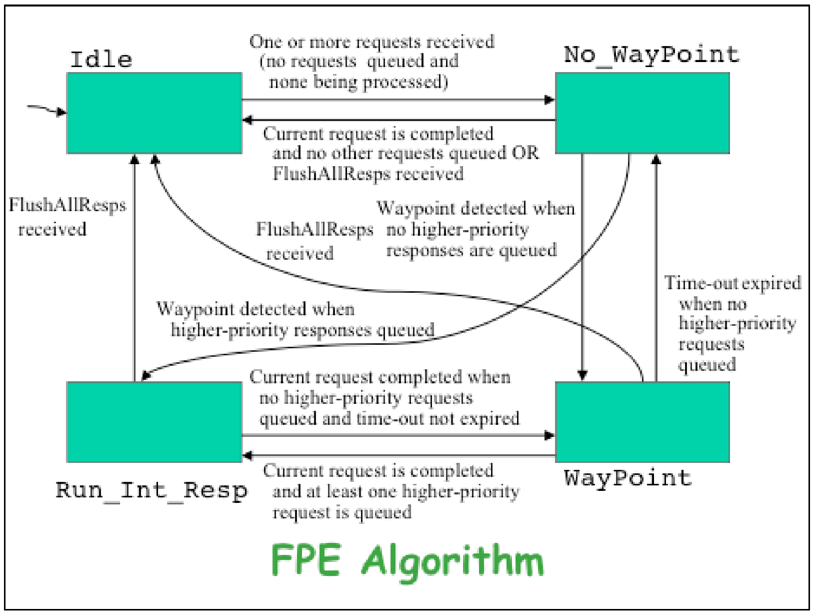
Idle
No_WayPoint
Run_Int_Resp
WayPoint
One or more requests received (no requests queued and none being processed)
Current request is completed and no other requests queued OR FlushAllResps received
FlushAllResps received
FlushAllResps received
Waypoint detected when no higher-priority responses are queued
Waypoint detected when higher-priority responses queued
Time-out expired when no higher-priority requests queued
Current request completed when no higher-priority requests queued and time-out not expired
Current request is completed and at least one higher-priority request is queued
FPE Algorithm

Figure 1. Required behavior of the Fault Protection Engine.

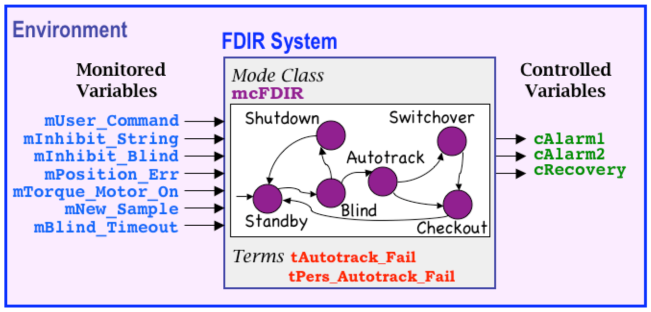

Figure 2. Monitored and controlled variables, modes, and terms in the SCR specification of FDIR

objective of applying SCR to FDIR was to evaluate the utility of the SCR tools in detecting errors in the FDIR requirements documentation.

Incubator Display

The Incubator Display is a component of the biological laboratory developed for use on-orbit on the International Space Station. It provides the human interface for controlling the Incubator, a habitat drawer whose purpose is to provide temperature-controlled environments for scientific experiments, and to monitor operations of the experiments contained in the Incubator Experiment Chamber. The objective of applying SCR to the Incubator Display was to evaluate the utility of SCR for representing the display's requirements and for simulating the required behavior of the display software using a graphical user interface. Figure 3 illustrates one screen of the Incubator Display's interface.

3. Overview of SCR

In an SCR specification of a system's requirements [10], [12], *monitored* and *controlled variables* represent the quantities in the system environment that the system monitors and controls. The required system behavior is specified as relations that the system must maintain between the monitored and controlled variables. To specify these relations concisely, the SCR language provides two classes of auxiliary variables—terms and *mode classes*—as well as conditions and events. A *condition* is a predicate defined on a system state. A basic *event*, represented as @T(c), indicates that condition c changes from false to true. The event @F(c) is defined by @T(¬c). If c's value in the current state is denoted c and its value in the next state as c', then the semantics of @T(c) is defined by $\neg c \wedge c'$ and the semantics of @F(c) by $c \wedge \neg c'$. A *conditioned event*, denoted @T(c) WHEN d, adds a qualifying condition d to an event and has the semantics $\neg c \wedge c' \wedge d$.

In SCR specifications, the monitored variables represent *independent variables*, and the mode classes, terms, and controlled variables represent *dependent variables*. SCR specifications define the values of the dependent variables using three types of tables: *condition*, *event*, and *mode transition tables*. The value of each term and controlled variable is defined by either a condition or an event table. To indicate a mode change, a mode transition table maps the current mode and a conditioned event to the new mode.

4. SCR Specification of FDIR

To illustrate the SCR method and notation, this section gives more details of the required behavior of FDIR and then presents an excerpt from the SCR specification of the FDIR requirements. It also describes two problems detected in the original FDIR requirements documentation when the SCR method was applied.

Figure 2 shows the inputs, outputs, and modes of FDIR and how they could be represented using SCR constructs. Shown

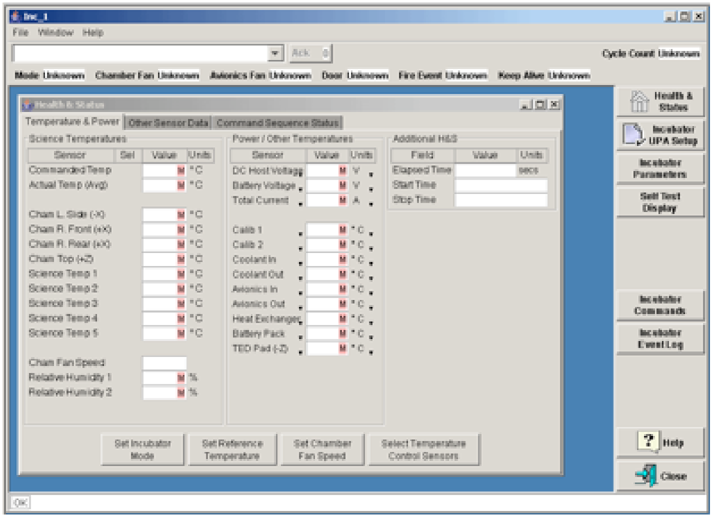

Figure 3. User interface of the Incubator Display.

in the figure are seven of FDIR's inputs represented as monitored variables; for example, `mPosition_Err` is a boolean indicating whether a position error has been detected, while `mUser_Command` indicates one of several user commands. At any given time, the system is in one of seven modes, either `Standby`, `Shutdown`, `Blind`, `Checkout`, `Switchover`, `Auto_Track`, or `DirectedPosition`. (Figure 2 omits one FDIR mode and excludes some of the FDIR mode transitions.) The controlled variables, `cAlarm1`, `cAlarm2`, and `cRecovery`, represent three system outputs, i.e., two alarms and the recovery action. To make it concise, the SCR specification also includes two terms, `tAutotrack_Fail` and `tPers_Autotrack_Fail`.

Example: Required response to a failure condition

The FDIR requirements documentation that we received from NASA consisted of 1) a table listing the set of failure conditions and the required response to each condition, and 2) a finite state diagram showing the FDIR modes, mode transitions, and input events triggering the transitions. Table 1 contains excerpts from item 1). According to the NASA contractor who provided the table, each row of this table, such as the row with ID 5, is interpreted as follows:[1]

```
(*)  IF Failure Detection Phase
        AND IF Persistence Time=None
            THEN Failure Criteria holds
            ELSE Failure Criteria holds
                 for Persistence Time
            ENDIF
     THEN
            DO Failure Notifications AND
            IF NOT Inhibit THEN
                 DO Recovery Response
            ENDIF
     ENDIF
```

[1]A variant of these semantics applies to the pairs of rows (ID 1a and ID 1b) and (ID 7 and ID 8). See the Appendix for details.

Consider the failure condition named "Blind Ops Timeout Exceeded" with ID 5 in Table 1. Given the above interpretation, the rule for responding to this failure condition is defined as follows: If a) the FDIR system is in the mode `Blind` and `Torque_Motor` is on and b) the time period that the system has been in mode `Blind` exceeds "Limit + 1", then issue the failure notification `Time_Limit_Blind` and, if the inhibit condition `Inhibit_Blind` is false, make a transition to `Shutdown_Mode`.

Translating the example into SCR

Table 1 describes the required FDIR response to each input event. The response may consist of one or more of the following: a mode transition, the generation of one or more alarms, or a recovery action. Alternatively, the response may be null. This section describes how the requirements in Table 1 may be expressed in three SCR tables, a mode transition table describing the conditions that cause the system to make a transition to a new mode, and two event tables, one defining the value of a recovery action and the other the value of the failure notification, a caution warning alarm.

The entries in the third, fourth, fifth, seventh, and eighth columns of In Table 1 specify the FDIR mode transitions: The entry in the third column indicates the current mode and (possibly) some condition that must hold, the entry in the fourth column indicates another condition (possibly persistent as indicated in the fifth column), the entry in the eighth column indicates the condition which inhibits a response to the failure condition, and the entry in the seventh column indicates the mode change if any. Table 2 shows an excerpt from the mode transition table defining the FDIR mode transitions based on the information in Table 1. For the row in Table 1 with ID 5, the corresponding row in Table 2 is the first row. This row states that if the current mode is `Blind` and the `mBlind_Timeout` expires when the condition `Torque_Motor_On` is true and the inhibit condition `mInhibit_Blind` is false, then the system makes a transi-

1	2	3	4	5	6	7	8
ID	**Failure Condition**	**Failure Detection Phase**	**Failure Criteria**	**Persistence Time**	**Failure Notifications**	**Recovery Response**	**Inhibit**
1a	Failure to Autotrack: response not inhibited	Autotrack Mode	Position_Err $\geq$ Autotrack_Error	Pers_Autotrack _Failure	Autotrack_ Failure, Joint_ Failure	Transition to Switchover Mode	Inhibit_ String
1b	Failure to Autotrack: response inhibited	Autotrack Mode	Position_Err $\geq$ Autotrack_Error	Pers_Autotrack _Failure	Autotrack_ Failure, Joint_ Failure	Device_ Power_Off, Transition to Checkout Mode	Inhibit_ String
	...	...	...	...	...	...	...
5	Blind Ops timeout exceeded	Blind Mode and Torque Motor On	Blind duration $>$ Limit + 1	None	Time_Limit_ Blind	Transition to Shutdown Mode	Inhibit_ Blind
	...	...	...	...	...	...	...
7	String failure: response inhibited not	Autotrack Mode	Receive CWA_Str ing_Failure	Pers_String _Failure	Joint_ Failure	Transition to Switchover Mode	Inhibit_ String
8	String failure: response inhibited	Autotrack Mode	Receive CWA_Str ing_Failure	Pers_String _Failure	Joint_ _Failure	Device_ Power_Off, Transition to Checkout Mode	Inhibit_ String
	...	...	...	...	...	...	...

Table 1. Excerpts from NASA's original requirements document for FDIR

Current Mode	Events	New Mode
`Blind`	`@T(mBlind_Timeout) WHEN (mTorque_Motor_On AND NOT mInhibit_Blind)`	`Shutdown`
`Autotrack`	`@T(tPers_Autotrack_Fail) WHEN (NOT Inhibit_String)`	`Switchover`
`Autotrack`	`@T(tPers_Autotrack_Fail) WHEN Inhibit_String`	`Checkout`
...	...	...

Table 2. Excerpt from the mode transition table for FDIR

tion to mode Shutdown. Note that mBlind_Timeout is an abstract monitored variable that becomes true when the time that the system has been in mode Blind exceeds Limit + 1.[2]

The information in the third, fourth, fifth, and sixth columns of Table 1 is also used to compute the value of the Caution Warning Alarm, which we model as cAlarm1. The third column indicates the current mode and possibly some condition, the fourth column specifies a condition (possibly persistent as indicated by the fifth column), and the sixth column indicates the value of the alarm. For example, for failure condition with ID 5 in Table 1, the corresponding row in Table 3, the event table defining the value of cAlarm1, is the second row. This row states that if the current mode is Blind and the mBlind_Timeout expires when the condition Torque_Motor_On is true, then the value of cAlarm1 is Time_Limit_Blind. The entry Never in the second row means that if the current mode is Blind, there is no event that can change cAlarm1 to have the value Autotrack_Failure.

In Table 1, the third, fourth, fifth, seventh, and eighth columns contain the information that defines the value of the recovery action. The entry in the fifth column indicates the *persistence*, the time that the condition defined in columns 3 and 4 must persist. In computing the recovery action, any entry in the seventh column other than the destination mode is significant. For failure condition with ID 1b in Table 1, the significant entry in the seventh column is Device_Power_Off, while for ID 1a there is no significant entry besides the mode transition. For failure condition with ID 1b in Table 1, the value of the recovery action cRecovery is defined in the first row and the last row of Table 4. These rows state that, if the system is in Autotrack mode and the failure to autotrack has persisted for some specified time[3] when Inhibit_String is false, then the value of cRecovery is None; if the system is in Autotrack mode and the failure to autotrack has persisted for some specified time when Inhibit_String is true, then the value of cRecovery is Device_Power_Off.

Two Problems in the Original FDIR Requirements Document

The objective of a requirements document is to specify the set of all acceptable system implementations. Hence, the document should be free of implementation bias since any design or implementation decision in a requirements document excludes some acceptable implementations. One serious example of implementation bias in the original FDIR requirements information was an internal continuation indicator for the failure condition ID 1. The function of this internal indicator was to "trigger" two alternative failure conditions to obtain an additional response indirectly. We eliminated this implementation bias by splitting the failure condition with ID 1 (in the original FDIR requirements document from which Table 1 was extracted) into two failure conditions and including the additional responses directly. The resulting failure conditions are those with the IDs 1a and 1b in Table 1 (see the

[2]This operation could have been expressed more concretely using SCR's duration operator.

[3]The details of persistence are expressed in the table defining tPers_Autotrack_Fail, which has been omitted from this paper.

Mode	Events	
Autotrack	@T(tPers_Autotrack_Fail)	Never
Blind	Never	@T(mBlind_Timeout) WHEN mTorque_Motor_On
cAlarm1′ =	Autotrack_Failure	Time_Limit_Blind

Table 3. Excerpt from event table defining Caution Warning Alarm cAlarm1

Mode	Events	
Autotrack	@T(tPers_Autotrack_Fail) WHEN NOT mInhibit_String	@T(tPers_Autotrack_Fail) WHEN mInhibit_String
Blind	@T(mBlind_Timeout) WHEN mTorque_Motor_On	Never
cRecovery′ =	None	Device_Power_Off

Table 4. Excerpt from event table defining cRecovery

Appendix for the full details). This elimination took place in the original FDIR requirements document *before* translating these requirements into SCR. Thus elimination of implementation bias from the original semi-formal tabular requirements was independent of the formal method chosen to represent the requirements.

Another error was detected when we used the information in Table 1 to specify Table 4, the event table defining the value of the recovery action cRecovery. In examining Table 4, one notes that a response of Device_Power_Off is required when mInhibit_String is true, while the response is None when mInhibit_String is false. This seemed non-intuitive to us since the objective of an Inhibitor is to inhibit a response, while this does the reverse. In fact, this was a problem with the stated requirements that was later fixed. However, this case is not reflected in the nominal semantics (*) but rather requires a detailed explanation from the domain experts.

5. Lessons Learned

General Lessons Learned

Quality of the Requirements Document. The original requirements documentation of each of the three components was, in varying degrees, ambiguous and imprecise. In the case of the FPE, the required software behavior was described in both natural language and in a Statecharts-like graphical notation. Although the FPE requirements document provided us with some intuition about the required software behavior, many details of the behavior were difficult to understand, and some of the documented requirements were ambiguous or inconsistent. In addition, the FPE requirements document omitted some of the required behavior. The requirements document for the Incubator Display was written solely in natural language, and, as a result, this document also suffered from imprecision and some incompleteness. The highest quality requirements documentation was the tabular requirements information and the finite state diagram describing the required behavior of the FDIR component. The tabular format distinguished important aspects of the required software behavior, such as the details of the component's inputs and outputs, and the finite state diagram identified other significant requirements information, such as the system modes, the mode transitions, and the input events that could trigger mode transitions.

Given the ambiguity, imprecision, and incompleteness of the requirements documents of the three components, we relied on application experts to clarify the required software behavior and to provide the details of missing requirements information. Access to these application experts was critical to preparing a precise, unambiguous SCR requirements specification. Such a requirements specification makes instances of incompleteness obvious. Application experts can often provide the missing requirements information.

Utility of the SCR Tools. As mentioned in Section 1, the SCR tools include various tools for validating and verifying the documented requirements. Those SCR tools with the most utility in formulating the requirements of the three NASA systems were the SCR consistency checker and the SCR simulator. The consistency checker was valuable in detecting instances of incompleteness and inconsistency in the tabular representation of the SCR-style requirements. It also exposed less serious problems, such as syntax and type errors, undefined and unused variables, and other errors and warnings analogous to those a compiler detects in a program. The simulator was useful in validation of the SCR specification by application experts. Running the simulator, for example, helped expose a serious error in the SCR specification of the FPE requirements (in this case, the error was in the translation to SCR). The simulator was also valuable for exposing aspects of the requirements that needed further clarification, e.g., the assumptions on which the requirements are based, the required response of a given system to a sequence of input events, etc.

The more advanced verification tools made available by the SCR toolset, e.g., the SPIN model checker [16], the theorem provers (the TAME interface [1] to PVS [19] and SALSA [2], and the invariant generator [17]) were not used in analyzing the three SCR specifications since the scope of each of the three efforts did not include advanced verification. The goal in the FPE effort was to demonstrate the feasibility of test case generation, while the goal in FDIR was to evaluate the utility of the SCR method and tools for detecting specification errors. A major goal for the Incubator Display effort was to demonstrate the utility of SCR's Graphical User Interface builder. The utility of SCR's advanced verification techniques have already been demonstrated in other efforts (see, e.g., [13], [18], and [11]).

Utility of the SCR-Style Requirements Method. In each of the three cases, the NASA personnel and contractors with whom we interacted were able to understand and evaluate the tabular and graphical notation that encode the SCR representation of the required software behavior. They were also able to apply the SCR consistency checker and simulator. In each case, these tools helped to detect various problems with the SCR requirements specification. However, in none of the three cases were the NASA personnel able to construct the initial SCR representation of the required component behavior. How to represent the required externally visible behavior of a component in the SCR notation can be challenging for practitioners, especially at the beginning. In each case, we generated the initial SCR specification of the some significant aspect of the required component behavior and presented it to the NASA personnel, who were able to provide feedback on behavior that we had captured incorrectly. Clearly, more guidance is needed to help software developers construct a state machine model of the required component behavior and to represent that behavior in the form of tables.

Fault Protection Engine: Lessons Learned

Utility of Automatic Test Case Generation. Most of the time and effort expended in the FPE effort was spent in constructing the SCR specification of the FPE requirements. Once the SCR specification was complete, applying the SCR test case generator [7] was relatively straightforward. Applying the test case generator to the SCR specification of the FPE requirements produced a small set of test cases for evaluating an FPE software implementation. These test cases were designed to satisfy the Branch Coverage criterion, which guarantees that software corresponding to every statement in the SCR specification is tested at least once. Unfortunately, due to limited funding, the set of test cases were never used to evaluate an FPE implementation.

Lack of Data Structures in the SCR Language. The lack of support in the SCR language for data structures, such as arrays and queues, made it difficult to specify some important aspects of the required behavior of the FPE requirements. For example, one major input to the FPE was a three-element vector containing the IDs of three different priority requests for responses to detected faults. The highest priority requests were ground requests for responses, the next highest priority requests were for interrupting responses, and the lowest priority requests were for non-interrupting responses. In addition, if it is already processing a request, the FPE is required to queue any new requests. Because the implementation of the SCR toolset available at the time did not support either arrays or queues, the SCR specification of the FPE requirements represented these data structures using integers and booleans. This made the FPE specification harder to understand and the generation of test cases more complex than necessary. A new prototype version of the SCR toolset supports both arrays and queues and hence this shortcoming of the toolset has been reduced.

FDIR: Lessons Learned

Exposing Implementation Bias. As described in Section 4, the SCR method was useful in exposing and removing implementation bias from the documented FDIR requirements. Interestingly, this problem was exposed, not by executing the SCR tools, but by organizing the FDIR requirements information in a different manner (see Table 1). The process of defining the SCR variables led us to discover the problem of implementation bias. The original requirement expressed the need to "Trigger String Failure," which was identified as an internal event, not an external input. The lesson here is that organizing the requirements with the SCR method can expose undesirable aspects of the requirements document that other organizations may not expose. Unlike many alternative specification languages, such as Statecharts [8] and Stateflow, SCR requires a black-box representation of the requirements that focuses on the externally visible behavior of the software—i.e., The system inputs and outputs and the required relation between them.

Importance of Explicit Semantics. When we first received NASA's tabular representation of the FDIR requirements, we were not sure how to interpret the information in the table. How to connect this information to the entities in an SCR-style requirements document—e.g., to monitored and controlled variables—was unclear. Translating the original requirements information from the original NASA table to the SCR representation removed this ambiguity. We were fortunate to have an application expert available who could explain the semantics of the NASA table. Without his expertise, the translation to SCR would no doubt have been incorrect.

Incubator Display: Lessons Learned

Utility of a Graphical User Interface. Once an SCR requirements specification has been constructed, the SCR toolset allows a user to design a graphical user interface to the SCR simulator. Such an interface is extremely valuable because it allows users to use the simulator to evaluate the correctness of the underlying SCR requirements specification. By running scenarios (i.e., sequences of monitored variable changes) through the simulator, an application expert can check that the specified behavior captures the intended behavior. The existence of the graphical interface means that the application expert can evaluate the specified requirements without understanding the SCR tables or the underlying state machine specification of the required behavior. The graphical front-end that we built for the Incubator Display with a small effort (approximately two days) was especially valuable.

Suitability of SCR for Specifying Reactive Systems. SCR is customized for specifying *reactive systems*, systems required to produce certain outputs in response to certain inputs. In reactive systems, there is a complex interaction between inputs, outputs, and system states (relevant parts of which are often modeled by modes) that may be conveniently captured in a state machine model. Although it was required to respond to inputs, the Incubator Display was not the ideal system to model in SCR since the nature of its responses mostly followed the same straightforward (and rather uninteresting) pattern of "In response to a user command to change some incubator parameter (such as the temperature, lighting, or fan speed), display the desired new value of the parameter." Further, the required response to each user command was independent of the Incubator Display's response to each prior command, so there was no concern about complex interactions. This lack of complexity meant that the underlying SCR state machine model was of limited value in representing the requirements of the Incubator Display.

6. RECOMMENDATIONS

Based on our experience in specifying the requirements of the three NASA components and, in particular, the lessons learned in doing so, we make three recommendations for improving the quality of requirements documents for safety-critical aerospace systems.

Need for an Updated SCR-Style Requirements Document. In 1978, the SCR requirements method was introduced with the publication of the requirements document for the Operational Flight Program for the A-7 aircraft [14]. In 1980, a journal paper was published describing the principles used in designing the requirements document and providing numerous examples [15]. The A-7 requirements document was carefully organized as a reference document: The goal of the document was to make requirements information easy to find, to remove redundancy, and to make the requirements information as precise, unambiguous, and complete as possible. Since 1980, some new information could be added to the individual chapters of an SCR requirement document; for example, a chapter listing a set of abstract properties, e.g., the high level "shall" statements in many NASA requirements documents, could be added.

There is an urgent need to revise and extend the organization of an SCR-style requirements document, such as the document published in 1978. A fully worked out version of such a document for an existing system, such as one of the components described in this paper, would be valuable as an example of how a rigorous requirements document should be organized. It would also provide examples of how tables can be used to specify the values of the system's outputs, i.e., its controlled variables, and other components of the software requirements. Such an example requirements document will serve two functions. First, it will provide an example of how software practitioners can specify and organize requirements information. Second, it will provide a repository for specifications that have been checked using tools. For example, the SCR consistency checker can be used to check the tables in the document for consistency and completeness. In addition, the more advanced tools in the SCR toolset, e.g., SPIN and TAME, could be used to check that the SCR specification satisfies the set of abstract properties included in the document.

Need to Involve Requirements Experts. Based on our experience in applying the SCR method to many practical systems, including mission-critical Navy systems and the three NASA components described above, our conclusion is that, for the foreseeable future, software developers will need requirements experts to help in formulating requirements documents in formal languages such as SCR. We believe it should be sufficient for requirements experts to construct an initial version of the requirements specification for some important aspect of the required component behavior. Once practitioners have a good example of how to specify and organize requirements documents in the SCR style, they should be able to extend and modify the requirements document.

Need to Develop and Apply More Advanced Technology. In eliciting and documenting requirements, applying some of the more advanced technology, such as advanced verification techniques, should help improve the quality of software requirements specification. In addition, once a software requirements specification has been validated (e.g., by application experts using a simulator) and verified (e.g., using a model checker or a theorem prover), it can be used as the basis for automatically generating test cases or for automatically generating efficient source code in a high level language such as C or Java (see, e.g., [20]). The use of this advanced technology should lead to significant improvements in the quality of both requirements specifications and software; it should also decrease the cost of producing reliable software.

7. CONCLUSIONS

Although this paper has focused on the SCR method for requirements specification, other rigorous requirements methods, such as RSML and its variants SpecTRM [21] and RSML^{-e} [9], should also be considered for formally specifying and formally analyzing software requirements. By weeding out errors early in the development process when they are cheap to correct, using more rigorous requirements methods and tools designed to improve the quality of requirements specifications (e.g., consistency checkers, simulators, and advanced verification tools) should produce both higher quality requirements specifications and higher quality, more reliable software. Moreover, the existence of high quality requirements specifications can help lower the cost of both producing test cases for evaluating hand-coded software and constructing efficient code by constructing the code automatically from the specifications.

8. Acknowledgments

The authors thank Allen Nikora of JPL, Randy Copeland of Titan, Tom Presburger of NASA Ames, and Susanne Moran of Intrinsix for answering our questions and providing documentation of the three NASA systems. This research was supported by NASA and the U.S. Office of Naval Research.

9. Appendix

This appendix presents the details of the implementation bias detected in the FDIR requirements document and how the bias was eliminated. To simplify the explanation, we define a number of abbreviations, each of which (except i) describes externally visible behavior. A, B, C, and D represent FDIR inputs treated essentially as boolean expressions, while X, Y, Q, and R represent FDIR responses.

A	"In Autotrack Mode"
D	"Position_Err $\geq$ Autotrack_Error holds for Pers_Autotrack_Failure"
i	"String_Failure"
X	"Autotrack_Failure"
B	"CWA_String_Failure holds for Pers_String"
C	"Inhibit_String"
Y	"Joint_Failure"
Q	"Transition to Switchover Mode"
R	"Device_Power_Off, Transition to Checkout Mode"

There are two exceptions to the nominal semantics (*) for interpreting the rows of the original FDIR requirements shown in Table 1. The semantics of ID 7, which indicates "response not inhibited" in Failure Condition is defined by:

```
IF Failure Detection Phase
   AND Failure Criteria holds for
       Persistence Time
   AND NOT Inhibit
THEN
      DO Failure Notifications AND
      DO Recovery Response
ENDIF
```

The semantics of ID 8, which indicates "response inhibited" in Failure Condition, is defined by:

```
IF Failure Detection Phase
   AND Failure Criteria holds for
       Persistence Time
   AND Inhibit
THEN
      DO Failure Notifications AND
      DO Recovery Response
ENDIF
```

Using (1) the original table expressing the semantics of FDIR processing, (2) the above abbreviations, (3) the semantics for ID 1 given by (*), and (4) the special case semantics for ID 7 and ID 8 given immediately above, the semantics for these three cases may be expressed as shown below ("DO x,y,..." abbreviates "DO x AND DO y AND ..."):

```
ID 1:  IF $A$ AND $D$ DO $X$, $i$:=true
ID 7:  IF $A$ AND ($B$ OR $i$) AND NOT $C$ DO $Y$, $Q$
ID 8:  IF $A$ AND ($B$ OR $i$) AND $C$ DO $Y$, $R$
```

The internal continuation indicator i is treated in a special way: if both A and D hold, the system should perform X and set i to *true*. Setting i in turn triggers processing via either ID 7 or ID 8 (depending upon whether C is *false* or *true*) to provide additional responses Y and Q or Y and R, respectively. Not only is the indicator i an artifact of "reusing" the requirements, but the sequencing of responses Y and Q or Y and R after response X is also artificial.

To remove the implementation bias, we eliminated the internal trigger i and sequencing of responses by splitting ID 1 into two cases ID 1a and ID 1b that now directly incorporate the respective additional responses Y, Q or Y, R as shown below:

```
ID 1a:  IF $A$ AND $D$ AND NOT $C$ DO $X$, $Y$, $Q$
ID 1b:  IF $A$ AND $D$ AND $C$ DO $X$, $Y$, $R$
ID 7:   IF $A$ AND $B$ AND NOT $C$ DO $Y$, $Q$
ID 8:   IF $A$ AND $B$ AND $C$ DO $Y$, $R$
```

Translating back to the tabular form Table 1 shows the complete version of rows ID 1a and ID 1b corresponding to the abbreviated forms above. Note that ID 1a and ID 1b now have the same special case semantics as the respective ID 7 and ID 8.

References

[1] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9:201–232, 2002.

[2] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Berlin, March 2000.

[3] M. Blackburn, R. Busser, A. Nauman, R. Knickerbocker, and R. Kasuda. Mars polar lander fault identification using model-based testing. In *Proceedings, 26th Annual NASA Goddard Software Engineering Workshop*, 2001.

[4] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[5] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Trans. on Softw. Eng.*, 24(1), January 1998.

[6] M. Feather, S. Fickas, and N. A. Razermera-Marny. Model-checking for validation of a Fault Protection System. In *Proc., 6th International Symposium on High Assurance Systems Engineering (HASE 2001)*, 2001.

[7] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Proc., 7th European Engineering Conf. and 7th ACM SIGSOFT Symp. on the Foundations of Software Engineering*, volume 1687 of *LNCS*, pages 6–10, Sep 1999.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[9] M. P. E. Heimdahl, M. W. Whalen, and J. M. Thompson. Nimbus: A tool for specification centered development. In *Proceedings of the 11th IEEE International Requirements Engineering Conf. (RE '03)*, September 2003.

[10] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for contructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Software and Systems Engineering*, 20(1):19–35, January 2005.

[11] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings, 13th ACM Conf. on Computer and Communications Security (CCS 2006)*, 2006.

[12] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5:231–261, July 1996.

[13] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), November 1998.

[14] K. Heninger, D. L. Parnas, J. E. Shore, and J. W. Kallander. Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

[15] K. L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Trans. Softw. Eng.*, SE-6(1):2–13, January 1980.

[16] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley, 2003.

[17] R. D. Jeffords and C. L. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, November 1998.

[18] J. Kirby, M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proceedings, 15th Annual Computer Security Applications Conference (ACSAC '99)*, pages 109–118, Phoenix, AZ, December 1999. IEEE Computer Society.

[19] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[20] T. Rothamel, C. Heitmeyer, E. Leonard, and A. Liu. Generating optimized code from scr specifications. In *Proceedings, ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2006)*, June 2006.

[21] M. Zimmerman, M. Rodriguez, B. Ingram, M. Katahira, M. de Villepin, and N. Leveson. Making formal methods practical. In *Proceedings, Digital Avionics System Conference (DASC)*, October 2000.

Constance Heitmeyer *heads the Software Engineering Section of NRL's Center for High Assurance Computer Systems. Her research interests include formal methods, requirements specification, and automatic test case generation. She is the co-editor of one book and the author of over 100 articles. She has MAs in mathematics and history from the University of Michigan.*

Ralph Jeffords *is a researcher in the Software Engineering Section of NRL's Center for High Assurance Systems. His research interests include formal methods, requirements specification, and invariant generation. He is the author of a number of articles in the areas mentioned above. Dr. Jeffords received his Ph.D. in computer science from Washington State University. He is a member of the Association for Computing Machinery.*