

Applying Formal Methods to an Information Security Device: An Experience Report ^{*}

Presented at HASE '99, Washington, DC, November 17-19, 1999

James Kirby, Jr. Myla Archer Constance Heitmeyer
Code 5546, Naval Research Laboratory, Washington, DC 20375
{kirby, archer, heitmeyer}@itd.nrl.navy.mil

Abstract

SCR (Software Cost Reduction) is a formal method for specifying and analyzing system requirements that has previously been applied to control systems. This paper describes a case study in which the SCR method was used to specify and analyze a different class of system, a cryptographic system called CD, which must satisfy a large set of security properties. The paper describes how a suite of tools supporting SCR—a consistency checker, simulator, model checker, invariant generator, theorem prover, and validity checker—were used to detect errors in the SCR specification of CD and to verify that the specification satisfies seven security properties. The paper also describes issues of concern to software developers about formal methods—e.g., ease of use, cost-effectiveness, scalability, how to translate a prose specification into a formal notation, and what process to follow in applying a formal method—and discusses these issues based on our experience with CD. Also described are some unexpected results of our case study.

1 Introduction

Although scores of formal methods have been proposed by researchers, the methods are rarely used by software practitioners. The common perception among practitioners is that formal methods require too much effort (e.g., specification using temporal logic) and too much detail early in the development process, do not scale to industrial projects, are not cost-effective, and provide no clear benefits. The following excerpt from the February, 1997 Call for Papers for the First IEEE International Conference on Formal Engineering Methods expresses this negative view:

Formal methods have been extensively researched in academia, but their application in industry is very limited. A critical barrier is the poor understanding of how to merge academic advances in formal methods into how industry actually builds software. Practitioners also feel that existing formal methods are difficult to use and their application consumes prohibitive amounts of resources, particularly at the start-up.

Motivated in part by our desire to evaluate these perceptions, we recently conducted a case study in which a formal requirements method called SCR (Software Cost Reduction) and a set of tools called SCR* were applied to a requirements specification for a communication security device (CD). Most of our information about CD, which is based upon the PEIP (Programmable Embeddable INFOSEC Product) technology, was obtained from a prose document describing CD's requirements.

At the start of the study, we formulated a set of questions that the study would address. These questions were designed either to evaluate the above perceptions or to address the suitability and the capabilities of the SCR method for CD. They included the following:

- **Ease of Use.** How hard was SCR to use? What aspects of the method presented the most difficulty? How could the level of effort be reduced?
- **Benefits.** What was learned about CD's requirements that was not previously known? Given the skepticism about formal methods, what aspects of our method were attractive to the CD program manager? What aspects were viewed negatively?
- **Cost-Effectiveness.** Was the effort of applying SCR to CD worth the benefits gained?
- **Scalability.** How well did SCR scale? Can the scalability of SCR be improved?
- **Applying SCR to a Secure System.** In the past, the SCR method has been applied to control systems, not to secure systems. How suitable is SCR for specifying and analyzing systems like CD that must satisfy critical security properties?
- **Starting with Prose.** How difficult is the translation from a prose requirements document to the SCR tabular notation? What characteristics are needed in a prose specification to facilitate translation into SCR?
- **Error Detection vs. Verification.** In many projects, formal methods are used to detect specification errors. Can one use the SCR method to go beyond error detection, to *verify*, that is, prove correct, critical system properties?

^{*}This work is funded by the Office of Naval Research and SPAWAR.

- **Process.** Reference [8] suggests a process for applying the various techniques of SCR* (i.e., specification followed in turn by consistency checking, simulation, model checking, and mechanical theorem proving). How effective is this process? Can it be improved?

This paper is organized as follows. Section 2 provides background information about SCR, the SCR* toolset, and CD. Section 3 describes the process we followed in applying the SCR method to CD. Section 4 describes what we learned from this exercise about the above questions, and Section 5 discusses some unexpected results. Finally, Section 6 discusses related work, and Section 7 presents our conclusions.

2 Background

SCR and the SCR* Toolset. The SCR requirements method [8] offers a tabular notation for specifying system requirements. The approach to specification is black box: SCR specifies the required system behavior in terms of *monitored and controlled variables*. These variables represent physical quantities in the system environment (such as airspeed, aircraft position, switch settings, or actuator positions) that the system monitors or controls. An SCR specification also uses additional variables, called *auxiliary variables*, to capture system history and to make the specification more concise. In an SCR specification, each dependent variable (each variable other than a monitored variable) is defined either by a *condition table* relating its value to those of other variables or by an *event table* describing the value of the variable when one of the variables on which it depends changes value. Besides tables, an SCR specification includes dictionaries of variables, constants, types, assumptions about the system environment, and application properties, e.g., safety and security properties that the system is intended to satisfy. The formal basis for the SCR* toolset is a formal model [8] which defines the semantics of SCR requirements specifications. In the formal model, a system is represented as an automaton whose states are determined by the values of the monitored and dependent variables, and whose transitions are determined from the tables in the specification.

SCR* [7, 6] is a set of software tools developed by NRL to provide mechanized support for the SCR method. SCR* includes a specification editor, a consistency checker, a simulator, a dependency graph browser, and an invariant generator [11]. To help the user check that the operational specification is consistent with the properties in the property dictionary, SCR* supports several analysis tools. Among these are the explicit state model checker Spin [9], which has been integrated into the toolset, and several tools which are partially integrated, including the TAME (Timed Automata Modeling Environment) interface

[1] to PVS [14], and a validity checker [3]. An additional SCR* tool is a test case generator [4] which constructs sequences of system inputs and expected outputs for testing the conformance of an implementation with an SCR specification. Automatic code generation of Java and C source code from SCR specifications is in the planning stage.

CD. The communications security device CD (COMSEC Device) provides cryptographic processing for a US Navy radio receiver. CD generates keystreams compatible with another cryptographic device and supports multiple radio channels. In addition, CD can download algorithms and associated keys into working storage, assign them to designated channels, and clear an algorithm and its keys from memory. Unlike most other COMSEC devices, which are implemented solely in hardware, CD is partially implemented in software.

The SCR specification of CD is based on a 48-page prose document, the CD System Requirements document (SRD). The SRD was designed to satisfy a large number of security requirements compiled by the DoD organization that evaluates COMSEC devices and certifies them for use. The SCR specification of CD captures a significant subset of the system behavior described in the SRD. While much of the SRD is consistent with the SCR’s black-box requirements model, some of the CD behavior described in the SRD does not follow this model. The SCR specification captures some of this behavior, also. In a few cases, the SCR specification captures additional behavior missing in the SRD; this additional behavior makes the SCR specification more complete. Like the SRD, the SCR specification deliberately omits some sensitive CD behavior.

3 Applying SCR to CD

To apply the SCR requirements method to CD, we first translated a subset of the prose SRD into SCR’s tabular notation. Then, we applied the SCR* analysis tools—the consistency checker, simulator, invariant generator, Spin, TAME, and the validity checker—to the SCR specification. The consistency checker, simulator, and invariant generator were used to check the well-formedness of the specification and to perform “sanity checks”. Spin, TAME, and the validity checker were used to analyze the CD specification for eight security properties.

3.1 From Prose to SCR

The CD Systems Requirement Document (SRD) is a traditional 2167A-style prose document. To capture the CD’s required behavior, the SRD describes the system modes, the mode transitions, and the system functions (e.g., key load function, reset function, report status function).

Unlike most prose requirements documents, the SRD captures many aspects of the required system behavior precisely and completely. We were able to ex-

tract most of CD’s required behavior directly from the SRD. We obtained security properties by examining our SCR specification, surmising the goals of the required behavior, and interpreting descriptions of functions in the CD SRD as security requirements. The CD project manager has reviewed the security properties that we formulated and confirmed that, except for one, they are reasonable security properties of CD.

Our SCR specification describes the behavior of CD that is consistent with the SCR’s black-box model of requirements. In SCR, the CD behavior is described in terms of inputs (the status of primary and backup power, data provided by the host, and positions of switches), outputs (indicator lights and status messages), and modes. The SCR specification also describes some memory management behavior that goes beyond SCR’s usual modeling of black box requirements. Because the SRD intentionally omits the rules for cryptographic synchronization and generating keystreams, we were unable to capture some required behavior that would be relevant and useful to reason about in our SCR specification.

The CD SRD assumes an unbounded number of algorithms and keys and an unspecified number of algorithm and key storage locations. Because the toolset does not provide a convenient means of specifying many identical entities, we assume two key banks, each with two key storage locations, and at most 1,000 different algorithms and 1,000 different keys. So that the system is always in exactly one mode, the SCR CD specification includes an additional Off mode not included in the SRD.

Another feature of the CD specification that does not fit SCR’s black box model is the Built-in Test (BIT), which represents a design decision, not a requirement. The SCR CD specification replaces the full and background BITs by the monitored variables mHealthyFull and mHealthyBackground, which each denote the operational “health” of CD’s security-critical components.

Most of the effort spent in building the SCR CD specification took place over a nine-month period as a background activity. The initial build of the specification took approximately one person-week. About one additional person-week was devoted to refining and completing the specification, with frequent use of the consistency checker as described in Section 3.2.

3.2 Applying the Consistency Checker

The consistency checker uses static analysis to check a specification for consistency with the SCR requirements model. The checks expose syntax and type errors, variable name discrepancies, unwanted instances of non-determinism (called *disjointness errors*) missing cases (called *coverage errors*), and circular definitions. Since the consistency checker is designed for use by engineers developing high assurance systems, the checks

are implemented as fully automatic push-button analyses that require no user input or guidance. When an error is detected, the consistency checker facilitates error correction by providing detailed feedback. For some types of errors, the checker, in addition to describing the error and highlighting where in the specification the error occurs, displays an example that demonstrates the error.

Aside from disjointness and coverage checks, all the checks execute quickly, so we invoked the checks many times during an editing session. We invoked the more computationally expensive disjointness and coverage checks less frequently.

3.3 Simulating the CD Specification

The SCR* simulator is a tool for *validation*, i.e., for checking that the specification captures the intended behavior of the system. Validating a system before it is built is difficult because people who have a deep understanding of the system’s intended behavior often do not have the time or the skills needed to read and analyze the specifications. The SCR* simulator allows users to evaluate the behavior of the specified system before it is built and without reading the SCR specification.

To facilitate validation of a specification, the simulator supports the rapid construction of front-ends customized for particular applications. Thus, application experts can interact with a representation of the system that closely resembles the actual system to be built. For example, to indicate to the user that the variable `cAlarmIndicator` has value `on`, the simulator displays a graphical representation of a red light labeled ALARM and may sound an alarm. By interacting with such front-ends, the user moves out of the world of requirements specification and into the world of the application.

We found an application-specific front-end for CD useful in interacting with the CD project manager. After viewing a simulation of CD using the CD-specific front-end (built in less than a day), the CD project manager provided us with useful feedback on the SCR specification of CD. Evaluation of the CD specification through this front-end to the simulator made effective use of a scarce commodity, the project manager’s time.

3.4 Automatic Invariant Generation

The algorithm for invariant generation constructs state invariants from the functions defining the dependent variables. For a dependent variable v taking values in a finite set $\{a_1, a_2, \dots, a_n\}$, the algorithm examines the conditions that can cause the value of v to change and generates for each a_i an invariant of the form $(v = a_i) \Rightarrow C_i$, where C_i is a predicate on the variables on which v depends. For dependent variables of numeric type, the hypotheses $v = a_i$ are replaced by predicates restricting v to intervals. Often, such inter-

vals can be computed automatically from values with which v is compared in the specification.

Applying the implemented algorithm to the mode transition table for the CD mode class `smOperation` initially yielded invariants that, for some modes, were unexpectedly weak. This led us to examine the mode transition table more closely and to correct the formulation of several events leading to a mode transition.

The invariants generated from the corrected mode transition table proved significant in an additional respect: three invariants provided auxiliary lemmas needed in the verification of security properties (see Sections 3.6 and 3.7). As noted in Section 3.6 below, two additional auxiliary invariants were also needed in verifying security properties. Applying a fuller (not yet implemented) version of the invariant generation algorithm by hand generated these two additional invariants.

3.5 Model Checking Properties

When a software specification describes an automaton, as in SCR, one can use a model checker to check its properties. Model checking performs an exhaustive search of some representation of the state space of the automaton. When the number of state variables is large, and particularly when—as is common in software specifications—the individual variables take values in a large (even infinite) set, the state space can become extremely large, making exhaustive search of the entire state space difficult or impossible. This is referred to as the *state explosion* problem. The problem can often be alleviated by applying *abstraction*.

For SCR*, we have developed automatable abstraction methods that reduce the state space either by eliminating variables (*variable restriction*) or by reducing the sizes of their type sets (*variable abstraction*) [6]. However, even with the use of abstraction, the state space to be searched often remains too large to search exhaustively. As a result, model checkers are seldom able to verify that a particular property holds. Nevertheless, when a property is not an invariant for the automaton, a partial search of the state space can often find states that violate the property. In addition to finding states in violation, most model checkers produce counterexamples in the form of scenarios—sequences of inputs—that lead to the bad state. Such counterexamples help users understand the reasons for property violations, and how to fix the specification to eliminate them.

We used Spin several times to analyze each property, adjusting Spin’s parameters in an attempt to explore the largest possible subset of the reachable state space. The discovery of a few property violations led to corrections in the formulation of some properties. Model checking failed to detect any violations of the eight properties we investigated. But because we were unable to search the complete state space of any of

the abstract specifications (the model checker ran out of memory before the analysis was complete), theorem proving was required to establish the properties as invariants. The importance of the theorem proving phase was demonstrated when we were able to establish with the help of a theorem prover that one of the eight properties is *not* an invariant (see Sections 3.6 and 3.7). In other words, sometimes theorem provers can find property violations that model checkers cannot, because for theorem provers, state explosion is not usually a problem.

3.6 Checking Properties with TAME

The tool TAME is a specialized interface to PVS whose goal is to reduce the human effort required in using PVS to specify, and to prove properties of, automata models. TAME was originally designed to specify and reason about Lynch-Vaandrager (LV) timed automata [12] but has been recently adapted to two other automaton models, I/O automata and the automata model that underlies SCR specifications (see [1]). TAME provides a template for specifying automata models, and approximately twenty specialized PVS strategies that mimic the high-level proof steps typically used by humans in proving invariant properties. Experience has shown that for automata models whose state variables have simple types (such as numerical, boolean, or enumerated types), nearly all state invariants can be proved using the TAME steps exclusively.

TAME has been partially integrated into the SCR* toolset. A prototype translator transforms SCR specifications to TAME specifications. Further, additional PVS strategies, appropriate for use in proofs of properties of SCR specifications, have been developed [1]. In practice, most properties of interest for an SCR automaton are either state invariants (properties of each single reachable state) or transition invariants (properties of all reachable transition pairs of states). State invariants, which are one-state properties, must either be proved by induction or by appealing to other state invariants. Although induction can also be used to prove transition invariants (which are two-state properties), this approach is seldom appropriate, since the transitions possible from any given state seldom have any connection to the transitions possible from one of its successor states. Rather, transition invariants are normally proved by reasoning directly about the next-state relation of the SCR automaton. To analyze these two kinds of properties, TAME provides two SCR-specific strategies: `SCR_INDUCT_PROOF`, which performs the standard parts of an induction proof for a state invariant and `SCR_DIRECT_PROOF`, which does the same for the direct proof of a transition invariant. These strategies combine into a single TAME invariant strategy.

In many cases, TAME’s invariant strategy is suffi-

cient in itself to prove an invariant. When the invariant strategy fails to complete a proof, there are two possible reasons: either the invariant does not hold, or additional invariants are needed in the proof. Associated with every “dead-end” in the proof is a problem transition. TAME supplies an analysis strategy ANALYZE that causes PVS to display the details of a problem transition to the user.

Applying the automatic invariant strategy of TAME to our eight proposed invariants for CD resulted in the automatic proof of four of these invariants. Three of the remaining invariants were proved by proposing and proving in TAME one or more auxiliary invariants. Studying these auxiliary invariants revealed that all of them were direct consequences of invariants generated by the invariant generation algorithm (see Section 3.4).

TAME also detected 14 problem transitions for the eighth proposed CD property, `smOperation = sAlarm \Rightarrow cKeyBank1Key1 = 0` (informally, “if CD is in Alarm mode then key 1 in keybank 1 is 0”). Some intelligent exploration using the SCR* simulator led to the discovery of a counterexample leading to one of these transitions, thus establishing that the property does not hold in our SCR specification of CD.

3.7 Applying the Validity Checker

VC, the SCR* validity checker [3], checks the validity of first-order one-state or two-state properties directly by using an initial term-rewriting phase followed by application of a decision procedure. The decision procedure uses BDDs (binary decision diagrams) to evaluate propositional formulae and a constraint solver to reduce simple integer arithmetic formulae (Presburger formulae). The variable ordering heuristic for the BDDs has been refined to be particularly efficient for SCR specifications. VC can also perform induction proofs by first applying a preprocessor to generate the appropriate base and induction cases and then applying the direct method to the generated cases. A prototype translator of SCR specifications into input for VC has been built.

VC is not a general-purpose theorem prover but can be applied to properties whose proofs do not require higher-order reasoning, reasoning about nonlinear numerical constraints, or interactive proof. Our eight security properties were all suitable for analysis by VC. VC was able to prove directly all of the properties not requiring auxiliary lemmas. Those properties that did require auxiliary lemmas had to be reformulated for VC with the auxiliary lemmas included as assumptions. Once this was done, VC was able to verify the remaining true properties of the SCR specification. For the false property (see Section 3.6), VC produced a single problem transition that is a special case of one of the 14 problem transitions reported by TAME. It is also the problem transition for which a corresponding counterexample was discovered using experimentation

in the simulator. Thus, like TAME, VC can also be used to detect property violations.

3.8 Generating Test Sets for CD

Applying the above techniques can lead to very high-quality requirements specifications. Although high-quality requirements specifications are valuable, the ultimate objective of the software development process is to produce high-quality *software*—software that satisfies its requirements. To weed out software errors and to convince customers that the software performance is acceptable, the software needs to be tested. An enormous problem, however, is that software testing, especially of safety-critical systems, is extremely costly and time-consuming. It has been estimated that current testing methods consume between 40% and 70% of the software development effort [2].

One benefit of SCR is that the high-quality specification produced by the method can play a valuable role in software testing. We have developed an automated technique [4] that constructs a suite of *test cases* from an SCR requirements specification. A test case is a sequence of system inputs, each coupled with a set of system outputs. We have built a prototype tool in Java based on this technique which automatically constructs a suite of test cases from an SCR requirements specification. Given the Java tool’s early success in constructing test cases [4], we expect that using the tool to generate test cases from the CD specification should be equally successful. The CD project manager has expressed significant interest in using test cases generated by our Java tool to test the CD software and other related software. Hence, the next crucial step in applying our method is to use the Java tool to generate test cases from the CD requirements specification.

4 Lessons Learned

This section answers the questions posed in the introduction, based on our experience with CD.

Ease of Use. We are experts in SCR, and thus did not need to learn the SCR specification method. Although we are novices in the area of secure systems, we nevertheless were able to produce a well-formed SCR specification for CD in just two weeks and to analyze a set of security properties for CD in less than three weeks. The brief time required to specify and to analyze the CD requirements using SCR* is evidence that SCR is easy to use.

The direct use of formal methods such as model checkers or theorem provers requires the user to invent an appropriate model of a system for analysis by these tools. Much of the advantage we experienced with SCR arises because SCR provides the user with a well thought-out organization for a specification and an editor for creating well-structured specifications. Further, SCR specifications can be automatically transformed into representations suitable for analysis by Spin, PVS,

and other formal analysis tools. This greatly simplified our application of model checking and theorem proving to the SCR specification of CD.

There were some difficulties. First, while Spin is well integrated into the toolset, there are inherent difficulties with Spin itself. For example, the effects of different settings of the Spin options are not obvious, so using Spin often involves much trial and error. Also, because a full state space search is usually infeasible, what conclusion to draw when Spin fails to find a counterexample is unclear. Second, the interpretation of proof dead-ends in TAME (was an auxiliary invariant needed, or did they correspond to counterexamples?) requires thought and creativity. Finally, some of the tools, including the invariant generator, VC, and TAME, are not fully integrated into the toolset. Hence, they cannot conveniently exchange information. Moreover, some invariant generation needs to be done by hand since the full invariant generation algorithm is not implemented, and some rewriting of the SCR specification is sometimes needed before the automatic translator into TAME can be applied.

Benefits. As Section 3 makes clear, we obtained several benefits from applying SCR to CD. First, we have a complete and consistent requirements specification for CD. Second, this SCR specification has been verified to satisfy seven security properties. Third, we have demonstrated that one particular property does not hold, and we have a counterexample that demonstrates the violation. Besides having a “good” SCR specification for CD, we discovered a few instances of incompleteness in the prose requirements document: a missing mode (as stated above) and some missing mode transitions.

In contrast to our positive view of the benefits obtained, the CD program manager viewed the formalization of the CD specification to be valuable mainly as a basis for automatic test case generation and automatic code generation. He was not especially interested in our formalization of CD or our formal analysis of its properties.

Cost-Effectiveness. As discussed above, it took us two person-weeks to specify a real system and less than three to analyze the specification. Thus, the effort required was low-cost. Moreover, we consider the benefits—a precise “build-to” specification that we know 1) is well-formed, 2) satisfies seven security properties, and 3) fails in a well-understood way to satisfy an eighth property—to be well worth that effort.

Scalability. We note that CD is a real system, and not a toy. The following statistics show that the CD specification is moderately complex. Our SCR specification of CD has 39 variables—17 monitored variables, 3 auxiliary variables, and 19 controlled variables. Both the relationship among these variables and their individual

tables are complex. In any state after the initial state, the monitored variable `mHostCommand` can take one of 17 values, and therefore, in any state of the CD, there are 16 possible input events involving changes in this variable. In addition, there are 16 other input variables. As a result, the mode transition table is large, involving 55 events to define 25 mode transitions. Many event tables in the specification are also large: the average number of events per table is 8, with the largest table containing 16 events.

SCR and its analysis techniques are not extraordinarily strained by a specification of this complexity. Full consistency checking of the CD specification takes around eight minutes. Individual Spin runs to analyze the specified properties require only a few—between one and thirteen—minutes. The proofs of the (true) state invariants require an average of six to seven minutes in TAME; the proofs of transition invariants require one minute or less.

Where will the problems arise as SCR is applied to larger and more complex specifications? In connection with the larger example of the WCP specification studied in [6], we have already encountered difficulty with the PVS typechecker. We developed abstractions of the WCP specification for six different properties, and then applied TAME to the properties in these abstractions. Only three of the abstractions could be type-checked in PVS, one after some hand optimization. That the PVS prover, unlike the typechecker, did not run out of memory for this last property suggests that the typechecker may be the biggest bottleneck in applying TAME to properties of more complex systems. A second problem, described above, is that often, model checkers do not run to completion on large examples. As a result, counterexamples may not be found, even when they exist. A third problem we have encountered is the slowness of the current SCR editor in displaying certain information, such as the variable dictionary, when the number of variables is large.

How to solve the PVS typechecking problem is an open question; the nature of the problem is somewhat subtle, since, for example, there is not an exact correlation of typechecking complexity with the number of state variables. An improvement that would address the problem with model checking is the development and implementation of better abstraction methods. The problem with display speed in the SCR editor has been solved in a new Java implementation of the SCR* toolset that is currently in progress.

Applying SCR to a Secure System. NRL originally developed SCR to specify requirements for control systems (e.g., avionics). CD, in contrast, is an information system that must satisfy a set of security properties. While most of CD’s behavioral requirements were easily captured by SCR, capturing others in the SCR black box model was problematic. The

SCR specification puts portions of CD’s memory on the interface of the black box. It describes the rules CD’s implementation must follow when loading keys and algorithms into memory and the rules for clearing memory when certain undesired events (e.g., power failure) occur.

As noted above, using SCR, we were able to verify the seven correct security properties we identified for CD, and to find a counterexample for the incorrect one. Since the actual security requirements are sensitive, we did not use them in verifying CD. Although we did not have access to the actual security properties, the CD project manager confirms that all but one of the properties that we identified are reasonable security properties. This gives us confidence that we would have similar results with the actual properties.

Because the authors of the prose requirements decided to omit details of some sensitive requirements (e.g., cryptographic synchronization, keystream generation), the SCR specification does not capture them. How well SCR could capture these requirements remains an open question.

Starting with Prose. Many consider the task of creating a formal specification from prose to be daunting. Yet, the two person-weeks of effort required to translate the prose CD requirements into the SCR tabular notation was relatively modest. The effort compares favorably with the approximately one person-week required to translate the larger, semi-formal WCP specification to the SCR notation using a partially automated process [6].

While the CD and WCP requirements documents differed in several important ways (e.g., prose vs. semi-formal specification, requirements for communication security device vs. control system), both captured behavioral requirements relatively completely and precisely. In both translation efforts we did not make (and could not have made correctly) decisions about the required behavior of each system; we only had to decide how to capture that behavior using SCR notation. In our experience, this does not require major effort.

The effort to create the CD SCR specification could have been reduced. Much of the translation effort involved mapping prose requirements organized by system function to SCR requirements, which are organized by the different variables. Organizing the prose requirements in a way that better supports hand translation to SCR (e.g., describing in one place all rules that determine the value of a particular output) would reduce the translation effort. We believe that eliminating much of the prose and capturing decisions about required behavior directly with the SCR toolset would save time and effort. Such an approach would rely on simulation to communicate the specified system behavior to those without the time or skills to read the specification.

Error Detection vs. Verification. The CD case study has demonstrated that it is possible to verify properties of the requirements specification of a real system, and, moreover, that it is possible to do so without extraordinary effort. Our experience suggests that verification may be feasible in other real-world applications.

Our case study has also shown that by applying verification, one can uncover errors that tools such as model checkers miss. This occurs in part because theorem provers can reason about abstract values, and as a result are much less dependent upon model abstraction than model checkers.

Process. In previous work with the SCR* Toolset [6], model checking before mechanical theorem proving eliminated the effort of trying to verify properties that do not hold. By guiding improvements in the formulation of some properties, model checking reduced the verification effort for CD as well. Unlike our experience with the WCP [6], however, model checking failed to find a violation of an invalid property. Both TAME and VC uncovered a violation of this property. Moreover, the tools in the toolset were complementary in additional ways: the results of consistency checking were useful to later analysis tools (e.g., the results of disjointness checking influenced the TAME representation of CD), and invariant generation proved sufficient to find all auxiliary lemmas needed in theorem proving. More automated communication between the tools would improve the process.

5 Some Unanticipated Results

We were surprised that, as noted above, Spin failed to uncover any counterexample for the false property of CD (in contrast to our experience with the WCP in [6]). This may be because the property in this case is “nearly true”, i.e., the density of counterexamples is small. In any case, it shows that the attempt to prove properties true can be useful in the search for errors.

Another unexpected result was that automatically generated invariants provided *all* of the auxiliary lemmas needed in the proofs of the true properties. How often this will happen in other analyses is unclear. However, the fact that it happened in this case demonstrates the potential power of automatic invariant generation and the importance of better information sharing between the different tools.

Although we had previously encountered similar skepticism about formal methods, we were surprised at the CD project manager’s negative view of our formalization of the CD requirements. The project manager’s view may partly be a result of the phase of development of CD when we undertook this study—implementation was underway—and may change when, as planned, we apply SCR in the initial stages of development of the CD transmitter (which has not yet been designed). Ap-

plying SCR in the initial phase of a project is likely to increase the cost-effectiveness of doing so, since the most costly errors in software development tend to occur during the requirements phase. However, for selling practitioners on the utility of formal methods, it may be effective to emphasize that a high-quality formal specification provides the foundation for the automatic generation of both source code and test cases.

6 Related Work

Other tabular specification methods with supporting tools include Tablewise [10] and RSML [5]. Tablewise was developed at Odyssey Research Associates, Inc. as part of a NASA-funded project in formal methods for the development of high assurance avionics software. RSML has been successfully applied to finding errors in the specification of a complex avionics system: the Traffic alert and Collision Avoidance System II (TCAS II).

Tablewise specifications consist of *decision tables*, which are similar to SCR condition tables, but use a more complex format to permit very simple (atomic) table entries. Like SCR, Tablewise provides disjointness and coverage checks. It also supports the generation of Ada code from the decision tables. Tablewise is not being further developed.

The AND/OR tables in RSML specify details of *transitions*, while SCR tables specify how *dependent state variables* are updated. As a result, RSML specifications require many more tables than SCR specifications. In contrast with SCR, RSML explicitly supports specification features such as hierarchical states and local variables. Like SCR, RSML provides automated support for checking consistency and a version of completeness (called d-completeness). Automated support for the analysis of specification properties beyond consistency and completeness is not yet extensive.

7 Conclusion

The CD example shows that the SCR method can be applied to systems other than embedded control systems. In addition, our experience with CD has underscored the importance of supporting a formal method with a suite of analysis tools, up to and including mechanical theorem proving. In particular, it has shown that the individual tools can both complement and support one another.

SCR has now been applied to many real systems, e.g., the Operational Flight Program of the A-7 aircraft [8], a flight guidance program [13], a weapons control panel (WCP) of a Navy submarine [6], and now, CD. In both the A-7 and flight guidance applications, after a best effort to eliminate specification errors using inspection, the SCR consistency checker still revealed a significant number of errors. Consistency checking also revealed errors in the WCP specification, and further analysis by model checking revealed several prop-

erty violations. The CD example has illustrated the usefulness of all the SCR analysis tools from consistency checking through theorem proving. Together, the above examples show that it is both feasible and beneficial to apply SCR to formalize and analyze real-world systems.

Acknowledgements

We wish to thank Stanley Chincheck and Thomas Sasala for providing us with their prose specification for CD, and Stan, Tom and our colleague Bruce Labaw for many helpful discussions. We also thank our colleagues Ralph Jeffords, Ramesh Bharadwaj and Steven Sims for performing some of the analyses of CD.

References

- [1] M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers 1998*, Eindhoven, Netherlands, July 1998.
- [2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.
- [3] R. Bharadwaj and S. Sims. Salsa: Combining decision procedures for fully automatic verification. Draft.
- [4] A. Gargantini and C. Heitmeyer. Automatic generation of tests from requirements specifications. In *Proc. ACM 7th Eur. Software Eng. Conf. and 7th ACM SIGSOFT Symp. on the Foundations of Software Eng. (ESEC/FSE99)*, Toulouse, FR, Sept. 1999.
- [5] M. P. E. Heimdahl and N. G. Leveson. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
- [6] 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), Nov. 1998.
- [7] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *10th Intl. Conf. on Computer Aided Verification (CAV'98)*, Lect. Notes in Comp. Sci., pages 526–531. Springer-Verlag, 1998.
- [8] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.
- [9] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997.
- [10] D. N. Hoover and Z. Chen. Tablewise, a decision table tool. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 97–108, Gaithersburg, MD, June 1995. IEEE.
- [11] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. 6th International Symposium on the Foundations of Software Engineering (FSE-6)*, Orlando, FL, November 1998.
- [12] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.
- [13] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.
- [14] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.