SCR: A Practical Approach to Building a High Assurance COMSEC System*

James Kirby, Jr.   Myla Archer   Constance Heitmeyer

Code 5546, Naval Research Laboratory, Washington, DC 20375

{kirby, archer, heitmeyer}@itd.nrl.navy.mil

Abstract

To date, the tabular-based SCR (Software Cost Reduction) method has been applied mostly to the development of embedded control systems. This paper describes the successful application of the SCR method, including the SCR* toolset, to a different class of system, a COMSEC (Communications Security) device called CD that must correctly manage encrypted communications. The paper summarizes how the tools in SCR* were used to validate and to debug the SCR specification and to demonstrate that the specification satisfies a set of critical security properties. The development of the CD specification involved many tools in SCR*: a specification editor, a consistency checker, a simulator, the TAME interface to the theorem prover PVS, and various other analysis tools. Our experience provides evidence that use of the SCR* toolset to develop high-quality requirements specifications of moderately complex COMSEC systems is both practical and low-cost.

1 Introduction

COMSEC (Communications Security) devices, devices which manage encrypted communications, are vital to the correct operation of U.S. military systems. CD, the COMSEC device of interest in this paper, is designed to provide cryptographic processing for a U.S. Navy radio receiver. In addition to generating keystreams compatible with another cryptographic device and supporting multiple channels and multiple cryptographic algorithms, CD downloads associated algorithms and keys into working storage, assigns them to designated communication channels, maintains the association between an algorithm and its keys, and clears algorithms and keys from memory. CD, based on a technology called PEIP (Programmable, Embeddable INFOSEC Product) for implementing COMSEC devices in software as well as hardware, presents a new challenge in the development of COMSEC devices. While a solid base of experience exists for implementing trustworthy COMSEC devices in hardware, implementing COMSEC devices in software is rare.

During the last decade, numerous formal methods, many with automated support, have been proposed for developing high assurance software systems. Because studies (e.g., [6]) show that errors, such as security property violations, that are introduced early in system development are both the most common and the most expensive to fix, the goal of many formal methods is to discover and eliminate flaws during the early stages of system development. While mechanically supported formal methods hold great promise for identifying errors early, the exceptional user expertise and effort usually required to apply them present a major barrier to their use in the development of practical systems.

The SCR (Software Cost Reduction) method [15, 11] is a formal method which offers a user-friendly tabular notation for specifying system requirements, and a set of tools called SCR* for detecting, often automatically, flaws in the requirements specification. Although originally designed to specify the requirements of safety-critical control systems, SCR can also be used to specify the required behavior of other systems, such as COMSEC systems. To make SCR* useful to practitioners, the tools are designed to be as automatic as possible and to complement and support one another. Included among the tools in SCR* are an automated *consistency checker*, a *simulator*, and various verification tools.

To provide a high degree of assurance in the correctness of CD's specification, we have applied the SCR method, including the SCR* tools [12, 13, 11]. Our results suggest that applying the SCR method in the development of COMSEC devices of moderate size and complexity is practical, effective, and low-cost. In approximately one person-month, we were able to represent a significant subset of a prose requirements document for CD in the the SCR notation and to establish that the SCR specification satisfies a set of security properties. The product of this effort is a high-quality requirements specification in whose correctness we have a high degree of confidence. This requirements specification can guide both the development of the source code and the development of test sets for eval-

---

uating the conformance of the source code with the system requirements.

The paper is organized as follows. It first introduces the SCR method and the SCR* toolset in Section 2, and then describes in Section 3 how the tools were applied to CD. Section 4 discusses the results of applying SCR* to the CD specification. Finally, section 5 discusses related work, and Section 6 presents our conclusions.

2 The SCR Method and Tools

The SCR method is a formal method designed to specify and analyze the requirements of safety-critical control systems. Since its introduction in 1978, the SCR requirements method has been applied successfully to a wide range of critical systems, including avionics systems, space systems, telephone networks, and control systems for nuclear power plants. See, e.g., [15, 23, 8, 7, 22, 19].

An SCR requirements specification describes both the system environment, which is nondeterministic, and the required system behavior, which is usually deterministic [12, 14]. Quantities in the environment that the system monitors and controls are represented by *monitored* and *controlled* variables. SCR specifications also use two types of auxiliary variables: *mode classes* (whose values are called *modes*) and *terms*, both of which often capture historical information. In the SCR model, the system environment nondeterministically produces a sequence of input events, where an *input event* is a change in some monitored quantity. The system is represented as a state machine (i.e., automaton) whose current state is determined by the values of the state variables, where a state variable is either a monitored or controlled variable, a mode class, or a term. Executions of the system begin in some initial state, after which the system responds to each input event in turn by changing state and by producing zero or more output events, where an *output event* is a change in a controlled quantity. The system behavior is assumed to be *synchronous*: the system completely processes one input event before the next input event is processed.

An SCR specification defines the transitions of a system using of a set of tables. Each table describes the value of a given state variable in the new state. Each *dependent variable*, i.e., each controlled variable, term, and mode class, has a corresponding table. Two constructs used in the tables are conditions and events. A *condition* is a predicate on system states. An *event* occurs when the value of any variable changes. The notation "@T(c) WHEN d" denotes a *conditioned event* defined as

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

where the unprimed conditions c and d are evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. Informally, this denotes the event "predicate c becomes true in the new state when predicate d holds in the old state". The table for a mode class is a *mode transition table*, which maps a source mode and an event to a destination mode. The table for any term or controlled variable is either an *event table*, which maps conditioned events to values of the variable in the next state, or a *condition table*, which maps conditions on the next state to values of the variable in the next state.

In addition to tables, an SCR specification contains dictionaries of *types*, *variable declarations*, *constant declarations*, *environmental assumptions*, and *specification assertions*. The specification assertion dictionary records required system properties, e.g., security properties. Our experience with practical systems is that most system properties can be represented as either state invariants or transition invariants, where a *state invariant* is a property that holds in every reachable state and a *transition invariant* is a property that holds in every reachable prestate/poststate pair (i.e., reachable transition).

The SCR* toolset [12, 13, 11] is a set of software tools developed by NRL to provide mechanized support for the SCR method. The tools include a *specification editor* for creating and modifying both an operational requirements specification (i.e., a state-machine representation of the required behavior) and a set of properties, such as safety and security properties; a *dependency graph browser* to display the dependencies among the variables in the specification; an automated *consistency checker* to expose missing cases, unwanted nondeterminism, and other application-independent errors [12]; a *simulator* to allow users to validate the specification; an interface to the *model checker* Spin [16] to detect violations of critical application properties; and an *invariant generator* [18] that computes state invariants from an SCR specification. To provide formal underpinnings for the tools and for the analysis techniques the tools implement, a formal model defines the semantics of SCR requirements specifications [14, 12].

Several additional tools have been recently integrated with SCR* by automatically translating the internal representation of an SCR specification into the input languages of the tools. These tools include *TAME* (Timed Automata Modeling Environment) [1, 2], an interface to the theorem prover PVS [25] for proving properties of automata models, a *validity checker* [4] which uses an integrated set of decision procedures to automatically check whether a given

property is a state or transition invariant of an SCR specification, and a *test set generator* [9] that automatically generates test sets from an SCR specification.

3 Applying SCR* to CD

This section describes the translation of a subset of the prose specification provided by the CD developers into an SCR specification and the results of applying the SCR* tools to the SCR specification. The tools and analysis techniques that were applied include the consistency checker, simulator, invariant generator, Spin, TAME, and the validity checker. This section also describes our plan to use the SCR* testing tool to automatically construct test sets from the SCR specification of CD.

3.1 From Prose to SCR Requirements

To develop the SCR specification, we studied the CD Systems Requirement Document (SRD) provided by the CD project manager, focusing on the constraints it imposed on the required system behavior and representing those constraints using SCR constructs. The CD SRD, a traditional 2167A-style document, was sufficiently precise and complete about key and algorithm management, modes of operation, and security requirements relating to power, tampering, and zeroizing for us to capture the required behavior in the SCR specification of CD. We obtained security properties by examining the SCR specification and surmising the goals of the required behavior and by interpreting descriptions of functions in the CD SRD as security requirements. The CD project manager has reviewed the set of security properties that we formulated and confirmed that, except for one, they are reasonable security properties of CD. The exception, according to the project manager, was a property whose hypothesis (backup power is over voltage), would never be satisfied.

Our SCR specification describes the part of CD's behavior (as described in the SRD) that is consistent with the SCR model of black-box 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. In addition, our specification describes some memory management behavior that goes beyond SCR's usual modeling of black-box requirements. Usually, in SCR, memory is considered to be internal to the black box, and thus invisible from the outside, but we treat it as externally visible by defining controlled variables that represent the memory locations in which the CD software can store algorithms and keys. This memory management behavior models the rules in the CD SRD for loading algorithms and keys, associating them with channels,

and clearing them from memory. There is (intentionally) not enough information in the CD SRD to specify the rules for cryptographic synchronization and generating keystreams. As a result, our SCR specification omits some required behavior that would be relevant and useful to reason about.

The CD SRD assumes that an unlimited number of algorithms and keys can be distributed among an unspecified number of storage locations and an unspecified number of channels. In the SCR specification, we assume that there are two key banks, each with two key storage locations; at most 1,000 different algorithms and 1,000 different keys; and two channels. The SCR CD specification has one more mode than described in the CD SRD: we add an Off mode so that the system is always in exactly one mode.
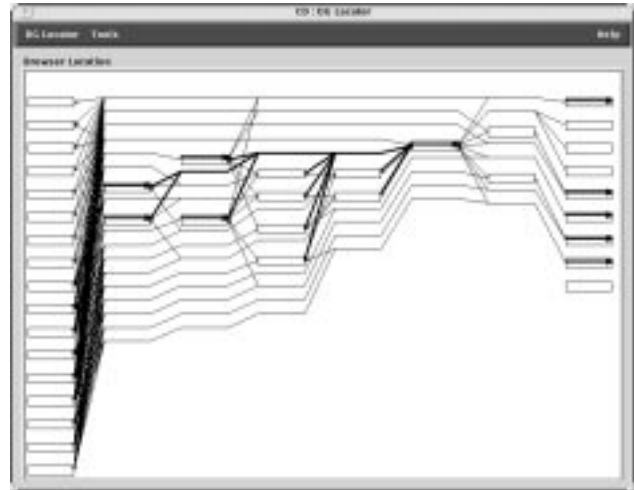

Figure 1. Full dependency graph for SCR CD.

Our SCR specification contains 39 variables—17 monitored variables, one mode class, two terms, and 19 controlled variables. Figure 1 shows the variable dependency graph for the complete SCR specification of CD. Variables are represented as boxes, and an arrow from one variable to a second variable indicates that the value of the first variable in the new state depends on the value of the second variable in either the current state or the new state. The heavy lines are backarrows; the number of backarrows reflects the complexity of the dependencies among the variables, which is also reflected in the complexities of the tables. Although this graph has cycles, the SCR* consistency checker was used to assure that there were no circular dependencies among the "new-state" variables (see Section 3.2).

Most of the effort spent in building the SCR specification of CD took place as a background activity over a nine-month period. The initial build of the specifica-

tion required approximately one person-week. About one additional person-week was needed to refine and complete the specification, with frequent use of the consistency checker (see Section 3.2).

3.2   Applying the Consistency Checker

The consistency checker uses static analysis techniques to expose syntax and type errors, variable name discrepancies, unwanted nondeterminism (called *disjointness errors*), missing cases (called *coverage errors*), and circular definitions (i.e., cycles in the dependencies among new-state variables). The checks are fully automatic and thus 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 (e.g., disjointness and coverage errors), the checker, in addition to describing the error, will highlight where in the specification the error occurs, and display a transition or state that demonstrates the error.

The consistency checker may be used at any stage in the development of a specification. All checks, except those for missing cases and nondeterminism, execute in a few seconds and are typically invoked many times during an editing session. In developing the CD specification, we frequently used the less expensive consistency checks as "sanity" checks. Since applying the more expensive checks for missing cases and nondeterminism to the entire CD specification usually requires between five and nine minutes, we invoked these checks less frequently.

Error Message from Tool	SCR* Highlights	Diagnosis
"smOperation Mode Transition Table: Cycle Detection ERROR: Cycle #1: Table smOperation uses mode class smOperation in the Name field; Function is smOperation Mode Transition Table"	Name of mode class in mode transition table	Events in table for smOperation introduce a cycle in the new state variable dependencies

Figure 2. Consistency checker feedback.

Figure 2 gives an example of an error message generated by the consistency checker during our development of the SCR specification of CD. The first column gives the error message displayed by the tool. The second and third columns describe the part of the specification that is highlighted at user request and our diagnosis of the error. In this example, the error is a circular definition, i.e., a cycle among the new state dependencies. This cycle occurred on our first attempt to describe CD's mode transitions in cases where the prose requirements described entry into a mode as ultimately resulting in exit from that mode to some other mode.

3.3   Simulating the CD Specification

In contrast to other tools in SCR*, which are for verification, the simulator is a tool for *validation*. The purpose of verification is to prove that the specification satisfies selected system properties, such as state and transition invariants; the purpose of validation is to confirm that the specification captures the operational system behavior intended by the customer. The simulator permits application experts to validate the behavior defined by the specification before the system is built. They can do so by running scenarios through the simulator rather than by reading the detailed SCR specification.

A scenario is a sequence of input events, each of which assigns a new value to one of the monitored variables. For each input event in the sequence, the simulator updates the values of the dependent variables before processing the next input event. In addition to presenting the current state of the execution, the simulator can present a history of the execution and report when a scenario violates specified properties.

The simulator's standard generic interface presents the current state of an execution in terms of the current values of the state variables, i.e., the monitored variables, mode classes, terms, and controlled variables. A disadvantage of the generic interface is that it presents an abstract description of the system state that application experts find unnatural. To overcome this problem, the simulator supports the rapid construction of graphical front-ends customized for particular applications. Each application-specific front-end contains graphical representations of switches, indicator lights, dials, and other entities in the human-computer interface that, in contrast to the generic interface, clearly and directly communicate information about the system behavior to the user.

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 the CD. Thus, evaluation of the CD specification through this front-end to the simulator allowed a very effective use of a very scarce commodity, the project manager's time.

3.4   Automatic Invariant Generation

The SCR* invariant generator is based on an algorithm for constructing state invariants from the functions defining the dependent variables. Consider a dependent variable v, defined by a mode transition table or an event table, which takes values in a finite set $\{a_1, a_2, \ldots, a_n\}$. The algorithm examines the conditions that can cause the value of variable v to change and generates for each a_i an invariant of the form

$$(v = a_i) \Rightarrow C_i,$$

4

where C_i is a predicate defined in terms of variables on which v depends. When v can take values in a very large (even infinite) set, the hypotheses $v = a_i$ are replaced by predicates defining a finite partition on the range of v; for example, when v has a numeric value, each predicate will define an interval. The appropriate intervals can often be computed automatically from the specification by identifying the values with which v is compared.

The automatic invariant generator currently provided in SCR* partially implements the algorithm for generating invariants from a mode transition table. The full algorithm, which we currently execute by hand, includes methods for generating invariants from event tables and condition tables and a strengthened method for mode transition tables; it will ultimately be implemented in SCR*. Figure 3 lists the nontrivial invariants that were generated automatically from the mode transition table for the CD mode class smOperation.

No.	Description	Generated Invariant
1	In Idle mode, the system is healthy and backup power is not overvoltage	smOperation = sIdle $\Rightarrow$ mHealthyBackground AND mBackupPower $\neq$ overvoltage
2	In Standby mode, backup power is neither undervoltage nor unavailable	smOperation = sStandby $\Rightarrow$ mBackupPower $\neq$ undervoltage AND mBackupPower $\neq$ unavailable
3	In Traffic Processing mode, the system is healthy and backup power is not overvoltage	smOperation = sTrafficProcessing $\Rightarrow$ mHealthyBackground AND mBackupPower $\neq$ overvoltage
4	In Configuration mode, the system is healthy and backup power is not overvoltage	smOperation = sConfiguration $\Rightarrow$ mHealthyBackground AND mBackupPower $\neq$ overvoltage

Figure 3. Nontrivial invariants generated automatically for the mode class smOperation.

Although the invariants generated from the specification are not the strongest possible invariants, they are often sufficient to establish interesting safety properties [18]. While applying the full invariant generation algorithm to CD did not provide results sufficient by themselves to establish the security properties we wished to verify, the generated invariants did play an extremely useful role: they provided every auxiliary lemma we needed to complete the proofs of all valid security properties that we investigated. Although there is no guarantee that this will always happen, that it did happen for CD suggests that applying invariant generation is a useful first step in verifying a set of properties, particularly since, once the full algorithm is implemented in SCR*, invariant generation will be fully automatic.

Three of the seven properties that we analyzed could not be proven automatically with either TAME or the SCR* validity checker (see Sections 3.6 and 3.7). To prove these properties, a total of five auxiliary invari-

ants were needed. Of these five invariants, which are listed in Figure 4, invariants 1A, 2A, and 3A can be derived from invariants generated by the implemented algorithm. For example, invariant 1A is implied by invariant 4 in Figure 3, one of the invariants generated automatically from the mode transition table for smOperation. Invariant 4A follows immediately from additional invariants which were generated by hand using the strengthened algorithm for mode transition tables. The contrapositive of invariant 5A is generated by applying the algorithm by hand to the event table defining the integer-valued variable cKeyBank1Key1.

No.	Description	Auxiliary Invariant
1A	In Configuration mode, backup power is not overvoltage	smOperation = sConfiguration $\Rightarrow$ mBackupPower $\neq$ overvoltage
2A	In Idle mode, backup power is not overvoltage	smOperation = sIdle $\Rightarrow$ mBackupPower $\neq$ overvoltage
3A	In Traffic Processing mode, backup power is not overvoltage	smOperation = sTrafficProcessing $\Rightarrow$ mBackupPower $\neq$ overvoltage
4A	If primary power is unavailable, then CD is in Standby, Alarm, or Off mode	mPrimaryPower = unavailable $\Rightarrow$ (smOperation = sStandby or smOperation = sAlarm or smOperation = sOff)
5A	If CD is in Off mode, then key 1 in keybank 1 is 0	smOperation = sOff $\Rightarrow$ cKeyBank1Key1 = 0

Figure 4. Auxiliary invariants needed for CD.

3.5 Model Checking Properties

When, as in SCR, a software specification describes a finite-state automaton, one can model check its properties. Model checking performs an exhaustive search of the state space of the automaton. If the number of state variables is large, and particularly if—as is common in software specifications—the individual variables take values in a large (even infinite) set, the state space can become so large that direct exhaustive search of the entire space is difficult or impossible. This problem, referred to as the *state explosion* problem, can often be alleviated by abstraction.

For SCR*, we have developed automatable abstraction methods that reduce the state space either by eliminating variables irrelevant to a property (*variable restriction*) or by reducing the range of variable values (*variable abstraction*) [5, 11]. When, as often happens, even abstraction does not allow the state space to be searched exhaustively, a partial search of the state space can often find states that *violate* a specified property. In addition to finding property violations, most model checkers produce counterexamples in the form of scenarios (i.e., execution sequences) that lead to the bad state. Below, we refer to counterexample scenarios simply as *counterexamples*.

Since model checking is largely automatic, using a model checker to check the validity of a property before trying to establish the property with a theorem prover

is often a useful screening strategy. If Spin finds a violation, it produces a counterexample, thus saving the effort needed to generate a counterexample from a dead-end in a proof. In checking security properties for CD, we followed this strategy.

No.	Description	Property
1	If CD is tampered with, then key 1 in keybank 1 is zeroized	@T(mTamper) $\Rightarrow$ cKeyBank1Key1$'$ = 0
2	When the zeroize switch is activated, key 1 in keybank 1 is zeroized	@T(mZeroizeSwitch = on) $\Rightarrow$ cKeyBank1Key1$'$ = 0
3	No key can be stored in location 1 of keybank 1 before an algorithm has been loaded into the first location of algorithm storage segment 1	cKeyBank1Key1 $\neq$ 0 $\Rightarrow$ cAlgStoreSegment1 $\neq$ 0
4	If backup power has an undervoltage when primary power is unavailable, the CD enters either Alarm mode or Off mode	@T(mBackupPower = undervoltage) WHEN mPrimaryPower = unavailable $\Rightarrow$ smOperation$'$ = sAlarm OR smOperation$'$ = sOff
5	If backup power is overvoltage then the CD is in Initialization, Standby, Alarm, or Off mode	mBackupPower = overvoltage $\Rightarrow$ smOperation = sInitialization OR smOperation = sStandby OR smOperation = sAlarm OR smOperation = sOff
6	If primary power has an overvoltage then either the CD is in Initialization, Standby, Alarm, or Off mode, or the CD enters Initialization mode	@T(mPrimaryPower) = overvoltage $\Rightarrow$ smOperation = sStandby OR smOperation = sAlarm OR smOperation = sOff OR smOperation$'$ = sInitialization
7	If primary power has an undervoltage then either the CD is in Initialization, Standby, Alarm, or Off mode, or the CD enters Initialization mode	@T(mPrimaryPower) = undervoltage $\Rightarrow$ smOperation = sStandby OR smOperation = sAlarm OR smOperation = sOff OR smOperation$'$ = sInitialization

Figure 5. Sample true properties for SCR CD.

Figure 5 lists seven security properties that the SCR specification of CD satisfies. Before we tried to prove any CD security property with TAME (see Section 3.6), we first used the Spin model checker to search for violations of the property. For each property, we used SCR* to automatically extract an abstraction from the CD specification and the property, using the variable restriction method described in [5, 11] to remove all variables irrelevant to the validity of the property. Then, by hand, we applied the variable abstraction method described in [11]. By limiting the range of values that certain variables can assume, this method usually produces a smaller abstraction. In our CD study, the abstractions for different properties varied very little. A typical abstraction contained 28 variables, a reduction of 28% from the 39 variables in the complete SCR specification.

Using Spin, we discovered a few property violations. In each case, closer examination of the property showed that the formulation of the property was incorrect. As one would expect, model checking was unable to find any violations of the properties subsequently verified by theorem proving. Because the model checker ran out of memory before the analysis was complete, we were unable to search the complete state space of any of the abstract specifications and therefore to *verify* any of the properties listed in Figure 5. The importance

of the theorem proving phase was demonstrated when we were able to use theorem proving both to prove that certain properties were invariants and to establish that one property for which Spin was unable to find a violation is not an invariant (see below).

3.6   Checking Properties with TAME

The tool TAME provides an interface to PVS for proving properties of automata models. TAME's goal is to reduce the human effort required in using PVS to specify these automata models and to prove state invariant properties for the models. TAME was originally designed to specify and reason about Lynch-Vaandrager (LV) timed automata [21] but has been adapted to I/O automata [20] and the automata model underlying SCR (see [2]). TAME provides more than twenty specialized strategies that implement proof steps mimicking 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.

We have integrated TAME into SCR* by developing an automatic SCR-to-TAME translator and special TAME strategies for the automatic analysis of properties of SCR automata [2]. For many SCR automata—in particular, those not involving timing constraints or other complexities such as tolerances for controlled quantities—a single TAME strategy can automatically prove many state invariants.

As stated in Section 2, most invariant properties of interest for an SCR automaton are either state invariants (one-state properties) or transition invariants (two-state properties). State invariants are typically proved by induction, with a base case for the initial states and an action case for each kind of input event. Although induction can be used in proving transition invariants, it 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 transition relation of the SCR automaton.

In TAME, the strategy SCR_INDUCT_PROOF performs the standard parts of an induction proof for a state invariant, and SCR_DIRECT_PROOF does the same for a transition invariant. A universal invariant proof strategy identifies the invariant as either a one-state or two-state property and then applies either SCR_INDUCT_PROOF or SCR_DIRECT_PROOF as appropriate.

Properties 1, 2, and 3 in Figure 5 took a few days to prove because the initial TAME representation of

CD combined with the initial versions of the strategies SCR_INDUCT_PROOF and SCR_DIRECT_PROOF led to unmanageably large data structures in the PVS prover. These problems led us to improve both our translation scheme and our proof strategies. After these improvements were made, we were able to prove properties 5, 6, and 7 in Figure 5 in less than an hour. The proof of property 4 took longer—about 2 days—because we needed to discover and to prove two layers of auxiliary invariants. This time would have been greatly reduced if the full invariant generation algorithm (see Section 3.4) had been automated.

When TAME's universal invariant strategy fails to complete the proof of an invariant, two possibilities exist: either the invariant is false, or additional invariants are needed in the proof. Associated with every proof "dead-end" is a problem transition. For one-state properties, this is the transition of the action case in the induction proof in which the dead-end appears. For two-state properties, this is the transition from the given state via some enabled automaton action to the successor state; the strategy SCR_DIRECT_PROOF produces only dead-ends in which the action is known, and hence for deterministic SCR specifications, the successor state (in terms of the given state) is known. TAME provides an analysis strategy to display the details of any problem transition. Once these details are understood, the user can determine whether the transition is reachable—in which case, the property is false—or whether it is unreachable, either because it would violate some transition invariant, or because one or the other of the states in the transition violates some state invariant.

Applying abstraction to a specification is less important in theorem proving than in model checking. Since a theorem prover can reason about abstract values, reducing the range of a variable using variable abstraction results in little or no improvement in the number of cases the theorem prover must consider. However, variable restriction can reduce both the number of cases and the complexity of reasoning about state transitions. Therefore, prior to analyzing a property with TAME, we applied variable restriction to the specification. Because the resulting abstractions for the individual properties were very similar, we used the same abstraction for all.

Applying the TAME strategies to the seven properties in Figure 5 resulted in the automatic proof of four of the properties. For two of the remaining properties, we proposed an auxiliary invariant, which was proved automatically and then applied to complete the proof. Property 1 in Figure 5 is an example of a property requiring a single auxiliary invariant, invari-

ant 5A in Figure 4, in its proof. Examination of the event table for the variable cKeyBank1Key1, in Figure 6 shows why the auxiliary invariant is needed:[1] when CD is in mode sOff, the event @T(mTamper) does not change the value of cKeyBank1Key1. Invariant 5A, which states that cKeyBank1Key1 is 0 in mode sOff, clearly covers this case.

Figure 6. Event table for cKeyBank1Key1

In the case of the third remaining property, we suggested an auxiliary invariant that completed the proof. However, applying the automatic proof strategy to the auxiliary invariant resulted in several dead-ends, and we therefore proposed three additional auxiliary invariants. These three new invariants were then proved automatically using the universal invariant strategy. As noted in Section 3.4, each of the needed auxiliary invariants was subsumed or implied by invariants that either were, or could be, generated automatically. Thus, once the invariant generator is extended and communication between the invariant generator and TAME is possible, the class of invariants that can be proved automatically using TAME will be extended.

Figure 7 shows a proposed eighth property that does not hold in the CD specification. Although Spin

Description	Property
If CD is in Alarm mode, then key 1 in keybank 1 is 0	$smOperation = sAlarm$ $\Rightarrow cKeyBank1Key1 = 0$

Figure 7. A false property for SCR CD.

---

[1] Some of the conditions and events have been abbreviated using ellipsis.

was unable to produce a counterexample for this property, TAME's analysis of the property found 14 problem transitions. Some intelligent exploration using the SCR* simulator produced a scenario that leads to one of these transitions, thus demonstrating that the property does not hold in the SCR specification. Detailed examination of the feedback from TAME shows that no obvious invariants forbid the other problem transitions, so it is likely that they also correspond to counterexamples.

3.7 Applying the Validity Checker

The SCR* validity checker VC [4] 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 that uses BDDs (binary decision diagrams) to evaluate propositional formulae and a constraint solver to reduce simple integer arithmetic formulae (Presburger formulae). The variable ordering used in the BDDs is particularly efficient for SCR specifications. VC can also perform an induction proof of a property by first applying a preprocessor to generate the appropriate base and induction cases and then applying the direct method to the generated cases. An automatic translation of SCR specifications into input for VC has been built.

VC has been applied to many of the same examples to which TAME has been applied, including the CD properties (after abstraction, as with TAME). The run time required by VC to analyze the CD properties was about half the time required by TAME. For the false property in Figure 7, VC produced a single problem transition, a special case of one of the 14 problem transitions reported by TAME. This is the same problem transition for which we used the simulator to find a counterexample. Thus, VC, like TAME, can be used in demonstrating that a property is invalid.

Unlike TAME, VC cannot be used to prove properties interactively. Therefore, the CD properties whose proofs required auxiliary invariants were checked after first including all necessary auxiliary invariants as assumptions, rather than by interactively invoking an analog of TAME's strategy for applying an invariant lemma. Mechanically checking the validity of complex properties (such as properties involving nonlinear numerical constraints or numerical constraints over real numbers, or properties whose proofs require types of higher-order reasoning other than induction over reachable states) requires a general-purpose theorem prover, such as PVS through TAME. However, VC can provide an efficient first screening for invariance for any property of an automaton that involves only propositional logic, simple integer constraints, and universal quantification over states or state pairs.

3.8 Generating Test Sets

Applying the formal techniques described above produces very high-quality requirements specifications. Although such 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 errors introduced by the implementation and to convince customers that the system performance is acceptable, the software needs to be tested. An enormous problem, however, is that software testing, especially of secure 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 [3].

The high-quality specification produced by the SCR method can play a valuable role in software testing. We have developed an automated technique [9] that constructs a suite of *test sets* from an SCR requirements specification. Each test set is a sequence of system inputs in which each input is coupled with the required system outputs. To ensure that the test sets "cover" the set of all possible system behaviors, our technique organizes all possible system executions (i.e., traces) into equivalence classes and builds one or more test sets for each class. These test sets can then be used to automatically evaluate the implemented software. By reducing the human effort needed to build and to run the test sets, such an approach can reduce both the enormous cost and the significant time and human effort associated with current testing methods.

With our technique, a model checker's ability to produce counterexamples is used to construct the test sets. The requirements specification is used both to generate a valid sequence of inputs and as an *oracle* that determines the outputs the system is required to generate from a given system input. To obtain a valid sequence of inputs, the input sequence is constrained to satisfy the environmental assumptions in the SCR requirements specification.

We have built a prototype tool in Java that automatically translates an SCR specification into the language of either of two model checkers, executes the model checker to build the test sets, analyzes its outputs, and finally produces a file containing the generated test sets. Our prototype tool has been applied to a number of specifications, including a sizable component of a contractor-specified weapons system [11]. Given the tool's early success in constructing test sets efficiently, we expect that applying the tool to the CD specification should be equally successful. The CD project manager has expressed interest in using test sets generated by our tool to test the CD software.

4 Discussion

The Complexity of CD. Our SCR specification of CD reflects the application's significant complexity and moderate size. As noted in Section 3.1, the SCR specification has 39 variables, and the relationships among these variables is 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 17 other input variables. As a result, the mode transition table is large, involving 55 events to define 25 mode transitions, and 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.

Time and effort required. Despite the complexity of CD, the total time taken in this study to develop and analyze the SCR CD specification was only one person-month.[2] Formalizing the specification of CD in SCR, including the use of the analysis tools to perform sanity checks, took only two person-weeks, and even the most complex consistency checks ran in minutes. The graphical front-end for simulation of CD was constructed in one day. Improvements to formulation of the properties based on feedback from the model checker took only a few days. TAME and the validity checker underwent significant improvement during our analysis of the SCR specification of CD, and as a result, analyzing a property with these tools now takes at most a few minutes, and sometimes only a few seconds. The most labor-intensive part of the analysis of a property is analyzing proof dead-ends to determine their cause and their resolution. As noted above, we plan to fully implement the invariant generation algorithm. This extension of SCR* should reduce significantly the problem of discovering useful auxiliary invariants.

The practicality of SCR. For our SCR specification of CD, we analyzed eight security properties. For each property, we were able to definitively answer the question, "Does the operational specification satisfy this property?" When the answer was "No," we provided a counterexample illustrating the failure. Although the full set of security properties for CD (to which we do not have access) numbers in the hundreds, our success with the properties we considered and the relatively short time required support the proposition that SCR and the analysis techniques supported in SCR* provide a practical, low-cost approach to providing high assurance. Typical concerns expressed by practitioners

---

[2]Additional time was needed to make improvements in some of the SCR* techniques. These improvements were suggested by our experience with CD.

regarding the practicality of formal methods are addressed in more detail in our discussion in [17] of the lessons we learned from our application of the SCR* tools to CD.

5 Related Work

RSML (Requirements State Machine Language) [10] is another requirements method in which, as in SCR, a system is specified as a state machine. 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). Like SCR specifications, RSML specifications include a set of tables and may be checked for consistency and for (a version of) completeness. SCR and RSML also have important differences. First, RSML has a Statecharts-style interface through which it explicitly supports specification features, such as hierarchical states and local variables, not explicitly supported in SCR (although similar effects can be obtained with SCR). Further, the AND/OR tables in RSML specify details of *transitions*, while SCR tables specify how *dependent state variables* are updated. Because a state machine has many more transitions than state variables, an RSML specification of a system contains many more tables than an SCR specification of the same system. Finally, automated support for the analysis of RSML specification properties beyond consistency and completeness is not yet extensive.

Reference [24] describes an earlier application of SCR to the development of another COMSEC device, the External COMSEC Adaptor (ECA). The development, from modeling the device through implementing and verifying its design, was done using the high-level SCR method, but not the SCR* toolset. The operational requirements were specified using SCR tables, and the critical requirements model—the desired properties—was specified using the CSP (Communicating Sequential Processes) language. In this effort, both formal and informal transitions between stages were used, with some automated support for the formal transitions from another mechanized theorem prover.

6 Conclusion

SCR offers a practical, low-cost approach to building a high assurance COMSEC device. Before implementation and design, the SCR* toolset can be used to build and analyze, often automatically, a mathematically precise requirements specification. Operational personnel can use the SCR* simulator to validate the behavior of the specified system. Further, model checking often can be used to identify security properties which the operational specification violates, and theorem proving can be used to verify the correct-

ness of security properties and suggest possible property violations. When analyses have established sufficient confidence in the requirements specification, the system can be built to satisfy that specification. A planned extension to SCR*, a tool to generate Java code from specifications, will help with this phase of development. Once the source code is available, test sets automatically generated from the operational requirements specification by the test set generator can be used to test the system implementation.

Acknowledgements

References

[1] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proc. 1996 IEEE Real-Time Technology and Applications Symp. (RTAS'96)*, pages 192–203. IEEE Computer Society Press, 1996.

[2] 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 (UITP '98)*, Eindhoven, Netherlands, July 1998.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

[4] R. Bharadwaj and S. Sims. Salsa: Combining decision procedures for fully automatic verification. Draft.

[5] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.

[6] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[7] S. Easterbrook and J. Callahan. Formal methods for verification and validation of partial specifications: A case study. *Journal of Systems and Software*, 1997.

[8] S. R. Faulk, J. Brackett, P. Ward, and J. Kirby. The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

[9] 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, September 1999.

[10] 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.

[11] 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):927–948, November 1998.

[12] 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.

[13] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. 12th Annual Conf. on Computer Assurance (COMPASS '97)*, Gaithersburg, MD, June 1997.

[14] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Tools for analyzing SCR-style requirements specifications: A formal foundation. 1999. Draft.

[15] 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.

[16] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[17] J. Kirby, M. Archer, and C. Heitmeyer. Applying formal methods to an information security device: An experience report. In *Proc. 4th IEEE International Symposium on High Assurance Systems Engineering (HASE '99)*. IEEE Computer Society Press, November 1999.

[18] 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.

[19] R. R. Lutz and H.-Y. Shaw. Applying the SCR* requirements toolset to DS-1 fault protection. Technical Report JPL-D15198, Jet Propulsion Laboratory, Pasadena, CA, December 1997.

[20] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[21] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[22] 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.

[23] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April–June 1991.

[24] C. N. Payne, A. P. Moore, and D. M. Mihelcic. An experience modeling critical requirements. In *Proc. COMPASS '94*, pages 245–256, Gaithersburg, MD, June 1994. IEEE Press.

[25] 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.