NRL Release Number 01-1221.1-0512

An Algorithm for Strengthening State Invariants Generated from Requirements Specifications*

To be presented at RE'01, Toronto, August 27–31, 2001

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

Abstract

In earlier work, we developed a fixpoint algorithm for automatically generating state invariants, properties that hold in each reachable state of a state machine model, from state-based requirements specifications. Such invariants are useful both in validating requirements specifications and as auxiliary lemmas in proofs that a requirements specification satisfies other invariant properties. This paper describes a new related algorithm that strengthens state invariants generated by our initial algorithm and demonstrates the new algorithm on a simplified version of an automobile cruise control system. The paper concludes by describing how the two algorithms were used to generate state invariants from a requirements specification of a cryptographic device and how the invariants in conjunction with a theorem prover were used to prove formally that the device satisfies a set of critical security properties.

1. Introduction

Automatic generation of *state invariants*, properties that hold in every reachable state of a state machine model, can be valuable in software development. Not only can such invariants be presented to system users for validation, in addition, they can be used as auxiliary lemmas in proving other invariant properties. While most algorithms for constructing state invariants operate on programs, we recently described an algorithm, called *KEEP*, for automatically generating state invariants from state-based requirements specifications [18]. Generating invariants from requirements specifications rather than programs has two major advantages: 1) because requirements specifications, unlike programs, are at a high level of abstraction, generation of and analysis using such invariants is easier, and 2) using invariants to detect errors during the requirements phase is considerably more cost-effective than using invariants later in software development.

This paper describes a new algorithm called *GROUP* for strengthening state invariants produced by the *KEEP* algorithm. It also provides evidence of the utility of automatically generated invariants in developing practical systems by describing how invariants generated by *KEEP* and *GROUP* were used to prove critical properties of a secure system.

1.1. A Simple Example

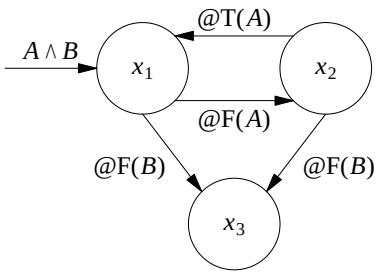

Figure 1. State diagram of simple example.

To illustrate how the *KEEP* and *GROUP* algorithms may be used to generate invariants, we consider a simple state machine that defines the value of a variable X with values from $\{x_1, x_2, x_3\}$ comprising the "states" of the machine. Figure 1 contains a finite state diagram describing the behavior of this simple state machine. The state machine determines the current value of X from changes in two Boolean variables A and B.[1] At each transition of this state machine, the value of either A or B changes but not both. We use the notation "@T(A)" to indicate that A changes from false to true and "@F(A)" to indicate that A changes from true to false. Initially, it is assumed that $A \wedge B$ holds. We say that the state machine "enters state x_i" if x_i is an

* This work was supported by the Office of Naval Research.

[1] This differs from the classical finite state machine: labels on arrows represent changes in state variables rather than inputs.

initial state or if there is a state transition satisfying $X \neq x_i$ and $X' = x_i$. (We use an unprimed variable to indicate the value before a transition and a primed variable to indicate the value after a transition.) Similarly, we say that the machine is "in state x_i" if $X = x_i$ and that the machine "exits state x_i" if there is a state transition satisfying $X = x_i$ and $X' \neq x_i$.

Theorem 1 describes a special case of *KEEP*, introduced by Atlee (Theorem 3.1 from [3]), which defines a rule for testing whether a literal ℓ, e.g., A, B, $\neg A$, or $\neg B$, is invariant in some state x_i:

Theorem 1 *A literal ℓ is invariant in state x_i if (a) ℓ is always true when x_i is entered, and (b) event $@\mathrm{F}(\ell)$ causes an exit from x_i.*

Applying the test in Theorem 1 determines invariants for values x_1 and x_2 in Figure 1 as follows. To find an invariant with respect to A when the state machine is in state x_1, we note that the only way to enter x_1 is either 1) to start out initially in x_1, in which case A holds, or 2) to enter x_1 from x_2 when the event $@\mathrm{T}(A)$ occurs, in which case A also holds. Thus A always holds upon entry to state x_1. Further, if in state x_1, the event $@\mathrm{F}(A)$ occurs, the state machine exits x_1 and enters x_2. Hence, by Theorem 1, A is an invariant for state x_1. A similar analysis for state x_2 determines that $\neg A$ is an invariant for x_2. The *KEEP* algorithm would also give these same results.

Checking state x_1 with respect to variable B, we note that the event $@\mathrm{F}(B)$ causes exit from x_1 and that B holds in the initial state. However, whether B holds if the system enters x_1 from x_2 is unknown. Hence, we cannot determine whether B is an invariant for x_1 using Theorem 1. A similar situation occurs for state x_2. The *KEEP* algorithm exhibits the same limitations.

To overcome these limitations, our new *GROUP* algorithm treats the two states x_1 and x_2 together as a set $\{x_1, x_2\}$. For example, upon entry into the set $\{x_1, x_2\}$, B is true (due to the initial state assumption). Further, if the system is already in $\{x_1, x_2\}$, the event $@\mathrm{F}(B)$ causes exit from $\{x_1, x_2\}$ and entry into x_3. Hence, B is an invariant for both x_1 and x_2. This illustrates the central idea of our new algorithm *GROUP*, which is to apply Theorem 1 to a "group" of states treated as a superstate.

Given G, a subset of states, our new theorem describes the test that *GROUP* applies to decide whether a literal ℓ is an invariant for each state in G:

Theorem 2 *A literal ℓ is invariant in state x for each x in G if (*) ℓ is always true upon entry to G (either in G initially or via a transition from some state not in G), and (**) event $@\mathrm{F}(\ell)$ causes an exit from G.*

1.2. Organization of the Paper

This paper introduces our new algorithm *GROUP*, a companion to *KEEP*, for generating state invariants from requirements specifications in the SCR (Software Cost Reduction) tabular notation. To provide background, Section 2 reviews the formal model, the special notation, and tools associated with SCR. To demonstrate how the *GROUP* algorithm is used in conjunction with the *KEEP* algorithm, Section 3 shows how special state invariants called "mode invariants" can be derived from a mode transition table (a type of table appearing in SCR specifications) by applying *KEEP* and how these mode invariants can be strengthened by applying *GROUP*. Section 4 formalizes Theorem 2 and the corresponding *GROUP* algorithm. It also describes a more complete algorithm that applies to more general systems, such as nondeterministic systems. This more general result should be applicable to other state machine models. To demonstrate the practical utility of automatically generated invariants, Section 5 describes how invariants constructed with *KEEP* and *GROUP* were used as auxiliary lemmas in proving three critical security properties of a requirements specification for a cryptographic device. Finally, Sections 6 and 7 discuss related work and present some conclusions.

2. Background: SCR Method

The SCR requirements method is designed to detect and correct errors during the requirements phase of software development. Originally formulated to document the requirements of the flight program for the U.S. Navy's A-7 aircraft [17], the SCR method has been used by many organizations in industry (e.g., Bell Laboratories, Grumman, Ontario Hydro, and Lockheed) to specify the requirements of practical systems.

2.1. The SCR Model and Tools

An SCR requirements specification describes a nondeterministic environment and the required system behavior (usually deterministic) [15]. *Monitored* (also called *input*) *variables* and *controlled* (also called *output*) *variables*, represent the environmental quantities that the system monitors and controls. The environment nondeterministically generates a sequence of input events, where each *input event* changes some monitored variable. Each input event may cause the system to change one or more of the controlled variables.

In SCR, NAT and REQ, two relations of the Four Variable Model [23], describe the required system behavior. NAT describes physical constraints on the environment; REQ describes the relation between monitored and controlled variables that the system must en-

force. To specify REQ concisely, SCR specifications use two types of auxiliary variables: *mode classes*, whose values are modes, and *terms*. Both mode classes and terms may be used to capture historical information.

More formally, an SCR system Σ is represented as a state machine $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and the transform T maps each input event and old state to a new state [15]. A simplifying assumption, called the *One Input Assumption*, states that exactly one input event occurs at each state transition. The transform T is the composition of smaller functions, called *table functions*, derived from the tables in an SCR requirements specification. Each table defines a term, a mode class, or a controlled variable.

The SCR requirements model includes a set $RF = \{r_1, r_2, \ldots, r_n\}$ containing the names of all state variables in a given specification and a function TY mapping each variable to its type, i.e., set of legal values. In the model, a *state* is a function mapping each variable r to some value in $TY(r)$, a *condition* is a predicate defined on the system state, and an *event* is a predicate defined on two successive system states that denotes some change between those states. The notation "@T(c) WHEN d" denotes a *conditioned event*, defined as

$$@\mathrm{T}(c) \text{ WHEN } d \stackrel{\text{def}}{=} \neg c \wedge c' \wedge d. \qquad (1)$$

Informally, "@T(c) WHEN d" means that c is *false* in the old state and changes to *true* in the new state, while d is true in the old state but unrestricted in the new state. In this paper, both $\neg c$ and $\overline{c}$ denote the negation of condition c.

Introduced in 1995, the SCR toolset [14, 15, 16] is an integrated suite of tools supporting the SCR method. The tools include a *specification editor* for creating the specification, a *simulator* for validating that the specification satisfies the customer's intent [14], and a *consistency checker* [15] to analyze the specification for properties such as syntax and type correctness, determinism, case coverage, and absence of circularity. The toolset also contains a *model checker*, a *verifier* called TAME [2], a *property checker* called Salsa [6], and an automatic *invariant generator* [18], which implements the *KEEP* algorithm.

The utility of the SCR tools has been demonstrated in several projects involving real-world systems. In one project, NASA researchers used the SCR consistency checker to detect missing assumptions and ambiguity in the requirements specification of the International Space Station [11]. In a second project, engineers at Rockwell Aviation used the SCR tools to detect 28 errors, many of them serious, in the requirements specification of a flight guidance system [22]. Recently, NRL used the SCR tools to uncover numerous errors, including a safety violation, in a sizable contractor-produced requirements specification of a weapons control panel for a safety-critical U.S. military system [13].

2.2. Modes and Mode Invariants

Three kinds of tables found in most SCR specifications are mode transition tables, condition tables, and event tables. Although this paper focuses on the generation of invariants from mode tables, extending the *GROUP* algorithm to event tables is straightforward (just as extending *KEEP* to event tables is straightforward [18]).

Figure 2 contains a mode transition table, part of an SCR specification for an Automobile Cruise Control System [16]. The table defines the values of a mode class M. In isolation, a mode class, its initial states, its inputs, and its transitions—which we call a *mode machine*—may be viewed as a very simple system Σ with a single dependent variable, a mode class M. In this machine, the mode is the "state" referred to in the simple example in Section 1. A mode transition table represents the transitions of a mode machine in a tabular format. The inputs of the mode machine are the variables appearing in the predicates that define the transitions. We informally say that the condition q is a *mode invariant* for mode m if $M = m \Rightarrow q$ is a state invariant.

In the Cruise Control system, the set of state variables RF is defined by RF = {`IgnOn`, `Lever`, `EngRunning`, `Brake`, M}, where `IgnOn`, `Lever`, `EngRunning`, and `Brake` are monitored variables and M is a mode class with values in the set {`Off`, `Inactive`, `Cruise`, `Override`}. The variables `IgnOn`, `EngRunning`, and `Brake` are Boolean; the variable `Lever` has the enumerated type {`off, const, resume, release`}. In the initial states of Cruise Control, both `IgnOn` and `EngRunning` are *false* and M = `Off`.

Figure 2 defines the transform T for this simple system. T maps the old state and an event, a change in the value of one of the monitored variables, to a new state. For example, the third row of Figure 2 states that if the system is in mode `Inactive` and the event @T(`Lever = const`) occurs when the engine is running but the brake is not applied, then the new mode is `Cruise`. If, in a given state, none of the events defining transitions from the current mode occurs (yet some input event has occurred), then there is no change in mode. For example, if the system is in mode `Inactive` and the brake is on when @T(`Lever = const`) occurs, the system remains in `Inactive`. This is because neither event that triggers exit from `Inactive` can occur: the

Old Mode M	Event	New Mode M
1 Off	@T(IgnOn)	Inactive
2 Inactive	@F(IgnOn)	Off
3 Inactive	@T(Lever = const) WHEN EngRunning AND NOT Brake	Cruise
4 Cruise	@F(IgnOn)	Off
5 Cruise	@F(EngRunning)	Inactive
6 Cruise	@T(Brake) OR @T(Lever = off)	Override
7 Override	@F(IgnOn)	Off
8 Override	@F(EngRunning)	Inactive
9 Override	@T(Lever = resume) WHEN NOT Brake OR @T(Lever = const) WHEN NOT Brake	Cruise

Initially: $M = \texttt{Off} \wedge \neg\texttt{IgnOn} \wedge \neg\texttt{EngRunning}$

Figure 2. Mode Transition Table for Cruise Control.

brake is on means that @T(Lever = const) WHEN EngRunning AND NOT Brake does not occur, while the One Input Assumption prevents the occurrence of @F(IgnOn).

3. State Invariants for Cruise Control

Section 3.1 briefly reviews the *KEEP* algorithm introduced in [18] by applying *KEEP* to the mode transition table in Figure 2. Then, Section 3.2 describes the new *GROUP* algorithm by showing how the state invariants generated by *KEEP* can be strengthened by applying *GROUP*.

3.1. Applying the *KEEP* Algorithm

Our technique automatically generates mode invariants from propositional formulas extracted from a mode machine and constraints on the input variables associated with that mode machine. To compute the mode invariants for the mode class M, we first identify the input variables appearing in the events of the mode transition table and in any constraints on the mode machine (such as the One Input Assumption). We then choose a set of atomic conditions that provides a sufficient basis for a Boolean encoding of all events in the table and these constraints. For example, in the Cruise Control specification, we choose the atomic conditions $I \equiv$ IgnOn, $E \equiv$ EngRunning, $B \equiv$ Brake, $O \equiv$ Lever=off, $C \equiv$ Lever=const, $R \equiv$ Lever=resume, and $L \equiv$ Lever=release.[2] Below, the term *literal* refers to either an atomic condition or its negation. The *KEEP* algorithm consists of the following three steps, repeated until a fixpoint is reached:

1. For each mode m, compute the *mode entry condition* $N(m)$, the disjunction of the conditions which may be true upon entry into mode m.

2. For each mode m, compute the *exit set* $X(m)$, where $X(m)$ is the set of literals, each of whose falsification causes exit from m.

3. For each mode m, compute the *mode invariant* $P(m)$ by removing from each disjunct in $N(m)$ all literals that are not members of $X(m)$, while "keeping" those literals which are members of $X(m)$. More precisely, replace each literal that is not in $X(m)$ by *true*.

Let $N_i(m)$, $X_i(m)$, and $P_i(m)$ represent the values of the mode entry condition, the exit set, and the invariant for mode m at the end of the ith pass of the algorithm. During each pass, a number of additional facts may be used to strengthen the invariant: environmental constraints, such as the One Input Assumption; constraints on enumerated type variables; and invariants computed on previous passes. A constraint on an enumerated type (needed due to our Boolean encoding) simply states that an enumerated type variable has exactly one value. For example, in the Cruise Control System, $C \Leftrightarrow \overline{O} \wedge \overline{R} \wedge \overline{L}$.

Applying *KEEP* to the mode transition table in Figure 2 produces the mode entry conditions, the exit sets, and the invariants shown in Figure 3 on the first pass. Because applying *KEEP* on the second pass does not change these results, the algorithm reaches a fixpoint on the second pass. Thus, two passes of *KEEP* generate the following state invariants:

- $M = \texttt{Off} \Rightarrow \neg\texttt{IgnOn}$
- $M = \texttt{Cruise} \Rightarrow \neg\texttt{Brake} \wedge \texttt{Lever} \neq \texttt{off}$
- $M = \texttt{Override} \Rightarrow \mathit{true}$
- $M = \texttt{Inactive} \Rightarrow \mathit{true}$

[2] Our Boolean encoding assigns one atomic condition for each of the four values of Lever even though Figure 2 mentions only three of these values.

Mode m	$N_1(m)$	$X_1(m)$	$P_1(m)$
`Off`	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I} \wedge \overline{E}$	$\{\overline{I}\}$	$\overline{I}$
`Cruise`	$C \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee C \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee R \wedge \overline{B} \wedge \overline{O} \wedge \overline{C} \wedge \overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$\overline{B} \wedge \overline{O}$
`Override`	$B \wedge \overline{O} \vee O \wedge \overline{B} \wedge \overline{C} \wedge \overline{R} \wedge \overline{L}$	$\{I, E\}$	$true$
`Inactive`	$I \vee \overline{E} \wedge \overline{O} \wedge \overline{B} \vee \overline{E}$	$\{I\}$	$true$

Figure 3. Mode Invariant Generation for Cruise Control

To illustrate the *GROUP* algorithm, we slightly modified the table in [18] to produce the table in Figure 2. Figure 2 omits `IgnOn` from the WHEN clause for transition 3 and `IgnOn` and `EngRunning` from the WHEN clauses for transition 9. Due to these modifications to the table, the *KEEP* algorithm generates weaker invariants than the algorithm generated for the original table in [18]. However, as Section 3.2 will show, the *GROUP* algorithm generates the remaining invariants.

3.2. Applying the GROUP Algorithm

Suppose $TY(M)$ is the set of modes of a mode class M, L is a set of literals, and $X(m)$ is the exit set for each m in $TY(M)$. For each candidate ℓ in L, the *GROUP* algorithm consists of the following four steps:

1. Let G be the largest subset of $TY(M)$ such that, for all m in G, ℓ belongs to $X(m)$.

2. If for some m in G, (*) of Theorem 2 fails to hold (i.e., there is some m in G whose mode entry condition—either initially or from some mode outside of G—does not imply ℓ), then remove m from G.

3. If for some m in G, (**) of Theorem 2 fails to hold because @F(ℓ) causes transition to some other mode m' in G, then remove m' from G.

4. Repeat steps 2 and 3 until no more modes may be eliminated from G.

Step 1 is natural since it provides the largest potential G. Step 2 is also straightforward. However, note that step 3 removes m' (not m) when (**) fails. While removal of m would be sound, it produces weaker invariants. This shows that the translation of even a simple intuitive criterion (such as the criterion in Theorem 2) to an appropriate algorithm requires careful deliberation. As with *KEEP*, environmental constraints and constraints on enumerated types may always be used to strengthen invariants. Also similar to *KEEP*, more than one pass may be required because invariants computed during one pass may either strengthen the mode entry conditions $N(m)$ or increase the exit sets $X(m)$ for the next pass.

To illustrate the *GROUP* algorithm, we apply it to the Cruise Control example. Step 1 of the algorithm limits our choice of ℓ to members of $X_1(m)$ in Figure 3. Applying the *GROUP* algorithm to this example produces nontrivial results for two cases: the literals I and E. Consider the case in which $\ell = I$. At step 1, $G = \{$`Cruise`, `Override`, `Inactive`$\}$, since I belongs to the exit set $X_1(m)$ of each mode m in G (see Figure 3). At step 2, (*) holds, since the only possible entry into G is from `Off` into `Inactive`, and Figure 4 shows that I holds upon entry into `Inactive`.[3] At step 3, (**) holds, since each occurrence of @F(I) from a mode in G causes an exit to `Off`, a mode outside G. Hence, the *GROUP* algorithm determines that the initial G satisfies Theorem 2 after a single pass.

Mode m	Mode entry from outside G	New invariant
`Cruise`	$false$	I
`Override`	$false$	I
`Inactive`	I	I

Figure 4. *GROUP* for Cruise Control ($\ell = I$)

In the case of $\ell = E$, at step 1, $G = \{$`Cruise`, `Override`$\}$. Step 2 computes the mode entry condition from outside G, which is limited to transition 3 of Figure 2. This transition from `Inactive` to `Cruise` is triggered by the event @T(C) WHEN $E \wedge \overline{B}$, which means that C holds in the new state and both E and $\overline{B}$ hold in the old state. Because *GROUP* computed I as an invariant of `Inactive`, I also holds in the old state. Because of the One Input Assumption, the three old state values are preserved in the new state. Finally, the enumerated type constraint means that $\overline{O} \wedge \overline{R} \wedge \overline{L}$ also holds in the new state (see Figure 5). (Although not shown in this example, previously computed invariants and enumerated constraints—besides the One Input Assumption—can be critical for ensuring that the mode entry condition implies ℓ.) As in the first case, *GROUP* determines that the initial G satisfies Theorem 2 after a single pass.

In summary, Figure 4 indicates that $m \in \{$`Cruise`, `Override`, `Inactive`$\} \Rightarrow$ `IgnOn`, and Figure 5 shows that $m \in \{$`Cruise`, `Override`$\} \Rightarrow$ `EngRunning`.

[3] The entries labeled *false* in Figure 4 indicate that the system can never enter `Cruise` or `Override` from outside G.

Mode m	Mode entry from outside G	New invariant
`Cruise`	$C \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L}$	E
`Override`	$false$	E

Figure 5. *GROUP* **for Cruise Control ($\ell = E$)**

Combining the invariants generated by the *GROUP* algorithm with those generated by the *KEEP* algorithm produces strengthened invariants as follows:

- $M =$ `Off` $\Rightarrow \neg$`IgnOn`
- $M =$ `Cruise` $\Rightarrow$ `IgnOn` $\wedge$ `EngRunning` $\wedge$ $\neg$`Brake` $\wedge$ `Lever` $\neq$ `off`
- $M =$ `Override` $\Rightarrow$ `IgnOn` $\wedge$ `EngRunning`
- $M =$ `Inactive` $\Rightarrow$ `IgnOn`

While the invariant for mode `Off` is unchanged, each of the remaining three mode invariants are strengthened.

Thus combining *KEEP* and *GROUP* derives the same invariants for the modified table in Figure 2 as the *KEEP* algorithm alone derived for the original table in Cruise Control [18]. It is now easy to show that the two tables are equivalent: the inclusion of I in the WHEN clause for transition 3 in the original table is redundant—adding it back to the modified system will cause no change in behavior since we know that I is invariant for mode `Inactive`; a similar argument holds for the other modifications to the table. With this observation, one may prefer the modified table in Figure 2 in certain contexts (e.g., in doing analysis of the specification) since it has less redundant information encoded in the WHEN clauses.

4. A Formal Treatment of *GROUP*

This section formally defines and extends concepts described informally above. Section 4.1 defines a mode machine as an abstract state machine, Section 4.2 formalizes Theorem 2, and Section 4.3 gives a more general, but less intuitive, formalization that also applies to nondeterministic systems and to mode tables with self-transitions. The weaker formalization in Section 4.2 appears so far to be sufficient in practice. While much of the notation in this section is borrowed from [13] and [18], some definitions have been simplified. The proofs of the two theorems presented in this section appear in [19]. The correctness of our results has been checked using the PVS prover [10].

4.1. Mode Machines as Abstract State Machines

We represent a system as a state machine $\Sigma = (S, \Theta, \rho)$, where S is the set of states, Θ is the initial state predicate, and ρ is the next-state relation on pairs of states. To define the state machine Σ corresponding to an SCR machine (S, S_0, E^m, T), we define 1) the initial-state predicate Θ on a state $s \in S$ such that $\Theta(s)$ is true iff $s \in S_0$ and 2) the next-state predicate ρ on pairs of states $s, s' \in S$ such that $\rho(s, s')$ is true iff there exists an event $e \in E^m$, enabled in s, such that $T(e, s) = s'$. Thus the predicate ρ is simply a concise and abstract way of expressing the transform T without reference to events.

A full SCR specification modeled as a state machine $\Sigma = (S, \Theta, \rho)$ has, for each mode class, an abstraction $\Sigma_A = (S_A, \Theta_A, \rho_A)$ that is a mode machine. We define abstraction so that a mode invariant q_A computed for a mode machine Σ_A corresponds to a mode invariant q in the overall state machine Σ. See [18] for details.

Suppose M is a mode class, $TY(M)$ the set of possible values (i.e., modes) of M, and E_A the set of events in the mode transition table for M. As in [18], four constructs define the mode machine for mode class M: the relation Υ_A describing the mode transitions, the initial state predicate Θ_A, and two predicates C_1 and C_2 on the monitored variables of Σ_A, which capture environmental constraints:

- Υ_A is a relation on $TY(M) \times E_A \times TY(M)$. In SCR specifications, this relation is represented by the Boolean encoded form of the mode transition table for M. We assume that Υ_A has no self-transitions, i.e., transitions of the form (m, e, m).
- Θ_A is the condition over Σ_A which describes the initial states. Additionally, we define the initial states associated with each $m \in TY(m)$ as $\theta_A(m)$, where $\theta_A(m) = \{s \mid \Theta_A(s) \wedge s(M) = m\}$.
- C_1 is a conjunction of encoded constraints on monitored variables *in a single state*. Among these constraints are the axioms needed to encode finite types as mutually exclusive Booleans. Other constraints are derived from NAT.
- C_2 is a conjunction of encoded constraints on monitored variables *in two consecutive states*. One constraint C_2 for the Cruise Control system is the One Input Assumption.

4.2. Formalization of Theorem 2 and *GROUP*

To formalize Theorem 2 and the *GROUP* algorithm, we borrow two functions from [18]: *NEW*, which is used to define the mode entry conditions, and *EX*, which is used to define the exit sets. Also required in the formalization are a variant of *EX* called EX^+, the exit set $X(m)$, and a variant of the mode entry condition $N(m)$ denoted $N^+(\hat{m}, m)$.

The function *NEW*, defined formally in [18], computes the strongest condition known to be true upon entry into the new state. Applying *NEW* to a two-state predicate in Disjunctive Form, i.e., a disjunction of non-*false* terms, simply replaces each old state literal with the literal *true* and suppresses the primes on the remaining new state literals. For example, the following shows how the event in transition 3 in Figure 2 can be rewritten first by applying (1), next by applying the One Input Assumption, and third by applying *NEW*:

$$\begin{aligned} NEW(@\mathrm{T}(C) \text{ WHEN } E\wedge\overline{B}) &= \\ NEW(\overline{C}\wedge C'\wedge E\wedge\overline{B}) &= \\ NEW(\overline{C}\wedge C'\wedge E\wedge\overline{B}\wedge E'\wedge\overline{B}') &= C\wedge E\wedge\overline{B}. \end{aligned}$$

The *GROUP* algorithm requires two versions of the function *EX*. The version in [18] defines a two-state predicate which describes the events causing exit from m as a disjunction. Formally, EX is defined by

$$EX(m) \stackrel{\text{def}}{=} \left(\bigvee_{e,m':m'\neq m \& \Upsilon_A(m,e,m')} e \right).$$

Applying this definition to lines 2 and 3 of Figure 2 means that

$$EX(\texttt{Inactive}) = @\mathrm{F}(I) \ \vee \ @\mathrm{T}(C) \text{ WHEN } (E\wedge\overline{B}).$$

The second version, EX^+, a slight modification to describe an exit from G, is a two-state predicate which is the same except for the qualifier $m' \notin G$. This predicate is defined by

$$EX^+(m,G) \stackrel{\text{def}}{=} \left(\bigvee_{e,m':m'\neq m \& m'\notin G \& \Upsilon_A(m,e,m')} e \right).$$

For example,

$$EX^+(\texttt{Cruise}, \{\texttt{Cruise}, \texttt{Override}\}) = @\mathrm{F}(I) \vee @\mathrm{F}(E).$$

Also needed are the exit sets $X(m)$ and the mode entry conditions $N^+(\hat{m}, m)$. In both definitions, P is the vector of invariants (computed prior to the application of *GROUP*) known to hold in the old state. Each exit set $X(m)$ is the set of literals whose falsification in the context of known invariants and environmental conditions causes exit from m:[4]

$$X(m) \stackrel{\text{def}}{=} \{\ell \mid @\mathrm{F}(\ell) \wedge P(m) \wedge C_2 \Rightarrow EX(m)\}.$$

[4]This definition is slightly less general than the definition in [18]. However, in practice, use of this definition for both *KEEP* and *GROUP* appears to cause no loss of precision in the computed invariants.

Each $N^+(\hat{m}, m)$ defines the mode entry condition into mode m from mode $\hat{m}$. $N^+(\hat{m}, m)$, a special case of the mode entry condition $N(m)$ (defined formally in [18]), is defined by

$$N^+(\hat{m}, m) \stackrel{\text{def}}{=} \left(\bigvee_{e:\Upsilon_A(\hat{m},e,m)} NEW(P(\hat{m}) \wedge C_2 \wedge e) \right) \wedge C_1.$$

To show that this definition correctly captures our intuitive notion of what is known upon mode entry, a more formal computation of the mode entry condition $N^+(\texttt{Cruise, Override})$ for the Cruise Control follows:

$$\begin{aligned} N^+(\texttt{Cruise, Override}) &= NEW[(P(\texttt{Cruise}) \wedge C_2 \\ &\quad \wedge (@\mathrm{T}(B) \vee @\mathrm{T}(O))] \wedge C_1 \\ &= NEW[\overline{B}\wedge\overline{O} \wedge C_2 \wedge (\overline{B}\wedge B' \vee \overline{O}\wedge O')] \wedge C_1 \\ &= NEW[\overline{B}\wedge B'\wedge\overline{O}\wedge\overline{O}' \ \vee \ \overline{O}\wedge O'\wedge\overline{B}\wedge\overline{B}'] \wedge C_1 \\ &= [B\wedge\overline{O} \vee O\wedge\overline{B}] \wedge C_1 \\ &= B\wedge\overline{O} \vee O\wedge\overline{B}\wedge\overline{C}\wedge\overline{R}\wedge\overline{L} \end{aligned}$$

In the above computation, C_2 represents the One Input Assumption and C_1 the enumerated type constraint $O \Leftrightarrow \overline{C}\wedge\overline{R}\wedge\overline{L}$.

Next, we present Theorem 3, a formalization of Theorem 2:

Theorem 3 *$M = m \Rightarrow \ell$ is a state invariant for each m in G of a deterministic mode machine Σ_A if for each $m \in G$: (*) $\theta_A(m) \wedge C_1 \Rightarrow \ell$; $N^+(\hat{m}, m) \Rightarrow \ell$ for each $\hat{m} \notin G$, and (**) $@\mathrm{F}(\ell) \wedge P(m) \wedge C_2 \Rightarrow EX^+(m, G)$.*

The *GROUP* algorithm is derived from Theorem 3:

To test if some literal ℓ is an invariant:

(a) Initially choose G to be all modes m for which $@\mathrm{F}(\ell)$ causes exit from m, i.e., ℓ belongs to $X(m)$.

(b) If for some mode m in G (*) fails, then remove m from G.

(c) If for some mode m in G there is some $m'(\neq m)$ in G and event e such that $\Upsilon_A(m, e, m')$ and the formula $@\mathrm{F}(\ell) \wedge P(m) \wedge C_2 \wedge e$ is satisfiable (i.e., the formula holds for some state pair (s, s')), then remove m' from G.

(d) Repeat steps (b) and (c) until no more modes may be removed.

There is a subtlety to the test in step (c): it detects the special case of the failure of (**) when m' is reachable from m via the occurrence of e. If (**) fails but m' is not reachable via the occurrence of e (i.e.,

$@\mathrm{F}(\ell) \wedge P(m) \wedge C_2 \wedge e$ is unsatisfiable) then we may ignore this case as it has no effect. In such situations, the event e should never have been included in the definitions of $EX(m)$ or $EX^+(m, G)$. Making this restriction in general, however, would complicate matters for the *KEEP* algorithm, where the distinction is irrelevant. Such subtleties are easily overlooked in informal proofs of algorithm correctness but are essential in our formal proofs using PVS.

4.3. A More General Formalization

The more general formalization of Theorem 2 applies to nondeterministic systems, allows self-transitions in the mode table, and handles the more general definition of $X(m)$ in [18]. We give an informal exposition of this more general result. For formal details and the transcription of this Theorem to a more general version of the *GROUP* algorithm, see [19]:

Theorem 4 *(More General Formalization): For a given literal ℓ and subset of modes G, if for each $m \in G$: (#) Assuming ℓ is an invariant for all of $G \Leftrightarrow \{m\}$ means ℓ is true upon entry to m and (##) $@\mathrm{F}(\ell)$ always enables exit from m,[5] then we conclude that ℓ is a mode invariant for each $m \in G$.*

5. Applying Invariants in Practice

5.1. A Cryptographic Device CD

COMSEC (Communications Security) devices, devices which manage encrypted communications, are vital to the correct operation of U.S. military systems. CD is a COMSEC device designed to provide cryptographic processing for a U.S. Navy radio receiver. CD, based on a technology 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.

To provide a high degree of assurance in the correctness of CD's specification, we applied the SCR tools [20]. Our results suggest that applying SCR 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 SCR notation and to establish that the SCR specification satisfies seven critical security properties. The SCR specification of CD is moderately complex, consisting of 39 variables (17 input variables, three auxiliary variables, and 19 output variables). Figure 6 lists the seven security properties that we verified with the SCR tools.

[5] We say "enables exit from m" since a nondeterministic system with explicit self-transitions among the modes may allow a self-transition to be taken even if exit is another possibility.

No.	Description	Property
1	If CD is tampered with, then key 1 in keybank 1 is zeroized	@T(mTamper) ⇒ cKeyBank1Key1′ = 0
2	When the zeroize switch is activated, key 1 in keybank 1 is zeroized	@T(mZeroizeSwitch = on) ⇒ 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 ≠ 0 ⇒ cAlgStoreSegment1 ≠ 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 ⇒ smOperation′ = sAlarm OR smOperation′ = sOff
5	If backup power is overvoltage then the CD is in Initialization, Standby, Alarm, or Off mode	mBackupPower = overvoltage ⇒ 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 ⇒ 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 ⇒ smOperation = sStandby OR smOperation = sAlarm OR smOperation = sOff OR smOperation′ = sInitialization

Figure 6. Security properties CD must satisfy.

5.2. Proving Security Properties Using Invariants

Our experience is that state invariants automatically generated using *KEEP* and *GROUP* are often sufficient to establish interesting safety properties. These generated invariants played an extremely useful role when applied in conjunction with TAME, a user interface to PVS that can prove many invariants automatically. In the case of CD, the automatically generated state invariants produced by *KEEP* and *GROUP* were sufficient to complete the proofs of all valid security properties that were investigated.

TAME was able to prove four of the CD security properties (3, 5, 6, and 7 in Figure 6) automatically. To prove the remaining three properties, TAME required the user to apply the five auxiliary invariants[6] listed in Figure 7. The *KEEP* algorithm generated the unbracketed parts of the invariants, with the remaining bracketed parts ([ ]) generated by *GROUP*. *KEEP* generated the fifth invariant from the event table for `cKeyBank1Key1` with "`cKeyBank1Key1` $= 0$" treated as one mode and "`cKeyBank1Key1` $\neq 0$" treated as a second mode. The *KEEP* invariants are not strong enough

[6] The more general form of *KEEP* generated additional invariants not used here, e.g., the mode invariant for `sStandBy`: `mBackupPower` $\notin$ {`unavailable`, `undervoltage`} $\wedge$ ($\neg$`mTamper` $\wedge$ `mZeroizeSwitch` $\neq$ `on` $\wedge$ `mHealthyFull` $\vee$ `mPrimaryPower` $\neq$ `unavailable`). This invariant, much stronger than the invariant generated by Theorem 1, demonstrates the power of *KEEP*.

Mode	Invariant for mode
`sInitialize`	[mPrimaryPower ≠ unavailable]
`sConfiguration`	mBackupPower ≠ overvoltage [∧ mPrimaryPower ≠ unavailable]
`sIdle`	mBackupPower ≠ overvoltage [∧ mPrimaryPower ≠ unavailable]
`sTrafficProcessing`	mBackupPower ≠ overvoltage [∧ mPrimaryPower ≠ unavailable]
"cKeyBank1Key1 ≠ 0"	smOperation ≠ sOff

Figure 7. Invariants generated for CD

to prove the fourth security property—the invariants generated by *GROUP* are also required. All of these results were also verified with the SCR property checker Salsa [6].

Our invariant generation tool implements part of the *KEEP* algorithm. For example, our tool constructed the unbracketed results in the first four lines of Figure 7. However, the fifth invariant, which was constructed from an event table, and the invariant mentioned in footnote 6, were generated by hand. The *GROUP* algorithm has not yet been implemented.

6. Related Work

Our *KEEP* and *GROUP* algorithms for generating invariants from SCR specifications extend work by Atlee and Gannon [3, 4], who used mode invariants to analyze SCR specifications with the MCB model checker. However, their automated technique only addressed a special case of our *KEEP* algorithm and did not cover the *GROUP* technique. Their work provided the inspiration for our research on mode invariant generation.

Mode invariants are similar to local invariants of programs, where the program location is analogous to the mode. Bensalem and his colleagues have refined techniques for generating local invariants [5]. However, their generation process is different from ours. For each process they determine "generalized reaffirmed invariants without (with) cycles" which are analogous to our mode entry condition computation (*GROUP* computation). The invariants from the processes are combined into overall system invariants. In contrast, we consider a single process (a mode machine) with effects of other processes expressed by the constraints (C_1 and C_2).

Recently, researchers at SRI have developed a theoretical framework for invariant generation based upon under- and over-approximation of inductive[7] invariants [25]. An *under-approximation* is a formula that is too strong to be an invariant, while an *over-approximation* is a formula that is an invariant, but is weaker than the best invariant. In this framework, each computation of mode entry conditions at step 1 of a pass of the *KEEP* algorithm is an under-approximation of the mode invariants. The remaining two steps of *KEEP* provide an inductive over-approximation. The first step of the *GROUP* algorithm is also an under-approximation. The remaining steps of *GROUP* comprise another inductive over-approximation, which provides invariants that often strengthen the results computed by *KEEP*. We have also shown (see [19]) how these approximations relate to abstract interpretation with widening and narrowing [8].

Other static techniques which analyze a state machine specification include the techniques of Halbwachs [9] and of Bjørner et al. [7]. In Halbwachs' technique, a system is represented as a system of linear inequalities, whose solution (a convex, closed polyhedron) is determined by successive approximations. While our techniques generate only simple invariants, Bjørner et al. have investigated the generation of general safety properties, using past temporal operators over the evolution of the system.

There are also dynamic techniques for obtaining invariants. Ernst et al. [12] obtain "potential invariants" for programs by monitoring likely expressions over many executions (although these would then have to be proved sound).

7. Conclusions

We have developed a new algorithm, *GROUP*, which improves upon invariants generated by a previous algorithm, *KEEP*. Because our algorithms produce intuitive, readable invariants (as compared to the more complete, detailed invariants that would be generated by a full reachability analysis of the mode machine Σ_A), our invariants can be presented to system users for validation.

Our algorithms are designed for SCR systems, but we have strived for generality that should make them applicable to other ways of modeling systems. The SCR concepts of the One Input Assumption and "strong causality" (i.e., transitions "*must* occur" when an appropriate event occurs, as opposed to a weaker concept of "*may* occur") support systems with stronger invariants than would occur without such assumptions. Nevertheless both the *KEEP* and the generalized version of *GROUP* (Theorem 4) do not require these assumptions. Nor do they require determinism. In the future we will investigate the applicability of both algorithms to other state-based models such as TLA (Temporal Logic of Actions) [21], Reactive Modules [1], and RSML^{-e} [24].

[7] A formula is inductive if it satisfies (i) $\Theta_A \Rightarrow q$, and (ii) $q \wedge \rho_A \Rightarrow q'$.

In practice, we have found that both the *GROUP* and *KEEP* algorithms are needed for generating auxiliary lemmas useful in proving major system properties of requirements specifications. Our experience applying these two methods to a cryptographic device showed that these generated invariants were sufficient for proving the desired security properties. Although there is no guarantee that this will always happen, our experience suggests that applying invariant generation is a useful first step in verifying a set of properties, particularly since, once both algorithms are completely implemented in the SCR toolset, invariant generation will be fully automatic.

Acknowledgments. Myla Archer provided very helpful comments on earlier drafts of this paper and supplied Figure 1. We also thank an anonymous referee for the improvement of step 3 in the *GROUP* algorithm.

References

[1] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, July 1999.

[2] M. Archer, C. Heitmeyer, and E. Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *Proc. FMSP'00*, pp. 25–36, Aug. 2000.

[3] J. M. Atlee. *Automated Analysis of Software Requirements.* PhD thesis, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1992.

[4] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Trans. Softw. Eng.*, 19(1):24–40, Jan. 1993.

[5] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.

[6] R. Bharadwaj and S. Sims. Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. TACAS'2000*, pp. 378–394, Berlin, Mar. 2000.

[7] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Comput. Sci.*, 173(1):49–87, Feb. 1997.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'77*, pp. 238–252, Los Angeles, CA, Jan. 1977.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL'78*, pp. 84–97, Tucson, AZ, Jan. 1978.

[10] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. Technical report, Computer Science Lab, SRI Int'l, Menlo Park, CA, Apr. 1995.

[11] S. Easterbrook and J. Callahan. Formal methods for verification and validation of partial specifications: A case study. *J. Syst. Softw.*, 40(3):199–210, 1998.

[12] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, Feb. 2001.

[13] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. Softw. Eng.*, 24(11):927–948, Nov. 1998.

[14] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. CAV'98*, pp. 526–531, Vancouver, Canada, June 1998.

[15] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Method.*, 5(3):231–261, July 1996.

[16] C. L. Heitmeyer, J. Kirby, Jr., and B. G. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proc. COMPASS'97*, pp. 35–47, Gaithersburg, MD, June 1997. IEEE.

[17] 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.

[18] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. 6th Int'l Symp. on Foundations of Softw. Eng. (FSE-6)*, pp. 56–69, Orlando, FL, Nov. 1998. ACM.

[19] R. D. Jeffords and C. L. Heitmeyer. Efficient automatic generation of state invariants from executable requirements specifications. Technical report, Naval Research Lab., Washington DC, 2001. (Draft).

[20] J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proc. of the 15th Annual Computer Security Applications Conf. (ACSAC '99)*, pp. 109–118, Dec. 1999.

[21] L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

[22] S. Miller. Specifying the mode logic of a flight guidance system in CoRE and SCR. In *Proc. 2nd ACM Workshop on Formal Methods in Software Practice (FMSP'98)*, pp. 44–53, 1998.

[23] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Sci. Comput. Programming*, 25(1):41–61, Oct. 1995.

[24] J. Thompson, M. P. E. Heimdahl, and D. Erickson. Structuring formal control systems specifications for reuse: Surviving hardware changes. In *Proc. 5th NASA Langley Formal Methods Workshop*, pp. 117–128, Williamsburg, VA, June 2000.

[25] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In T. Margaria and W. Yi, editors, *TACAS 2001*, volume 2031 of *LNCS*, pp. 113–127, Genova, Italy, Apr. 2001.