Automatic Generation of State Invariants from Requirements Specifications *

To appear in *Proc. Sixth Int'l Symp. on Foundations of Software Engineering (FSE-6)*, Orlando FL, Nov. 3–5, 1998.

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

Abstract

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 assertions in proving other invariants. This paper describes an algorithm for the automatic generation of state invariants that, in contrast to most other such algorithms, which operate on programs, derives invariants from requirements specifications. Generating invariants from requirements specifications rather than programs has two 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. To illustrate the algorithm, we use it to generate state invariants from requirements specifications of an automobile cruise control system and a simple control system for a nuclear plant. The invariants are derived from specifications expressed in the SCR (Software Cost Reduction) tabular notation.

Keywords—requirements, specification, formal methods, invariants, verification, validation, software tools

1 Introduction

Given the high frequency of defects in software requirements specifications, the high cost of correcting them late in software development, and the serious accidents such defects may cause, techniques for the early detection and removal of defects from software requirements specifications are crucial. One formal method that is designed to detect and correct errors during the requirements phase of software development is the SCR (Software Cost Reduction) method. Originally formulated to document the requirements of the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft [19], 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. The largest application of SCR to date occurred in 1993-94 when engineers at Lockheed used a version of SCR to document the complete requirements of Lockheed's C-130J OFP [12], a program containing more than 230K lines of Ada code.

Introduced in 1995, the SCR toolset [16, 17, 18] is an integrated suite of tools supporting the SCR requirements method. Each tool in the suite detects a special class of errors. For example, the *specification editor* helps the user detect ambiguous requirements; the *consistency checker* automatically detects violations of application-independent properties, such as type errors and missing cases; the *simulator* helps the user detect cases in which the specification fails to satisfy the specifier's intent, and a newly integrated model checker SPIN [21] detects violations of application-specific properties, such as safety properties [6]. Recently, NRL applied the SCR tools to a sizable contractor-produced requirements specification of the Weapons Control Panel (WCP) for a safety-critical U.S. military system [15]. The tools uncovered numerous errors in the contractor specification, including a safety violation. This violation, which could lead to a serious malfunction of the weapons system, was detected by model checking.

To specify the required system behavior, users of the SCR method may adopt a *dual language approach* [29]. In this approach, two different specifications are developed, one operational and the other property-based. The *operational* (or *model-based*) specification describes how the system operates, while the *property-based* spec-

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

ification describes the required system properties. An operational specification may represent the system as a state machine, whereas a property-based specification usually expresses properties as logic formulas. In the SCR requirements method, the operational specification is expressed in tables and the system properties as first-order logic formulas. Examples of the dual language approach in SCR specifications include the A-7 requirements document [19, 20], which, in addition to the tabular operational specification, contains properties of the system modes, and Kirby's cruise control specification [23], which contains both a tabular specification of the required system operations and a list of required system properties.

The dual language approach is useful because each specification style has advantages: operational specifications are less likely to omit required behavior and are often executable, whereas property-based specifications are concise, abstract, and minimize implementation bias. Another advantage of the dual language approach is that detecting inconsistencies between two different specifications of the same behavior is an effective technique for debugging both the statements of the required properties and the operational specification. For example, when Dill and his colleagues used model checking to analyze a hardware design for several properties of interest, they detected errors both in the design *and* in the stated properties [11].

One formal technique useful in conjunction with the dual-language approach is automatic invariant generation. This technique automatically generates state invariants, properties that hold in every reachable state of a state machine model, from the operational specification. Such state invariants can be presented to system users for validation or, alternately, can be used as auxiliary invariants in proving additional properties from the requirements specification, such as the properties included in the property-based specification.

This paper introduces an efficient, automatable algorithm for generating state invariants. In contrast to most other such algorithms, which operate on programs, our algorithm derives invariants from requirements specifications. Generating invariants from requirements specifications has two major advantages: 1) requirements specifications, unlike programs, are at a high level of abstraction, and hence 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.

Our algorithm, which extends the methods described in [3, 4], generates invariants from specifications expressed in the SCR tabular notation. The invariants are computed by combining information from a table taken from an SCR specification and various other facts, such as environmental assumptions. To illustrate the algorithm, we show how special invariants called "mode invariants" can be derived from a mode transition table, a type of table appearing in SCR specifications. Next, we obtain invariants from two other types of tables, both extracted from the same SCR specification. To demonstrate the utility of our approach, we use these invariants as auxiliary invariants in proving two properties of the specification. These properties were previously proved using model checking. Finally, we present a more formal description of a generalized version of our algorithm. This generalized version may be used to extract invariants from other state-based specifications, such as specifications in TLA (Temporal Logic of Actions) [24] and specifications for STeP (Stanford Temporal Prover) [27]. The Appendix presents the proof of the generalized version of the algorithm. We have formally proved the correctness of the generalized algorithm using the PVS prover [10, 30].

2 SCR Requirements Model

An SCR requirements specification describes a nondeterministic environment and the required system behavior (usually deterministic) [17]. *Monitored* (also called *input*) *variables* and *controlled* (also called *output*) *variables*, which represent the respective quantities the system monitors and controls, model the system environment. The environment nondeterministically generates a sequence of input events, where each *input event* is a single change in 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 [32], 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 enforce. 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 [17]. A simplifying assumption, called the *One Input Assumption*, states that 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. (Alternatively, the transform can be expressed in relational form—see Section 6.) 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 which maps each variable to its type, i.e., its set of legal values. In the model, a *state* s is a function that maps each variable r to some value in $TY(r)$. A *condition* is a predicate defined on the system state, whereas 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

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

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, "@T(c) WHEN d" means that c was *false* in the old state and has changed to *true* in the new state, while d was true in the old state but is unrestricted in the new state. The notation "@F(c)" is defined by @F(c) = @T($\neg c$). In reasoning about conditions c and d, we say that c *strengthens* d (also expressed as $c < d$) if $c \Rightarrow d$ is a tautology, but $c \neq d$. In this paper, both $\neg c$ and $\overline{c}$ denote the negation of condition c.

3 Mode Classes and Mode Invariants

The three kinds of tables found in most SCR specifications are mode transition tables, condition tables, and event tables. While the focus in this paper is on generating invariants from mode transition tables, Section 5 describes how invariants can be obtained from condition tables and event tables.

In isolation, a mode class, its inputs, and the associated transitions—which we call a *mode machine*—may be viewed as a very simple system Σ with a single output, a mode class. 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. Table 1 contains a mode transition table, part of an SCR specification for the Automobile Cruise Control System [18]. In this 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`.

Table 1 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 fourth row of Table 1 states that if the system is currently in a state where the mode is `Cruise` and the event @F(`IgnOn`) occurs, then, in the new state, the mode is `Off`. If, in a given state, none of the events defining transitions from the current mode occur (yet some input event has occurred), then there is no change in mode. For example, if the system is in `Cruise` mode in the old state and some input event occurs, but none of @F(`IgnOn`), @F(`EngRunning`), @T(`Brake`), or @T(`Lever` = `off`) occurs, then the system remains in `Cruise` mode in the new state.

A *mode invariant* for mode m, $M = m \Rightarrow P(m)$, is a special case of a state invariant, where $P(m)$ is a proposition over the state variables. For example, four mode invariants of the Cruise Control System that can be derived from Table 1 and other information about the Cruise Control System, such as environmental constraints and assumptions about the initial states, are

- 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`

4 Mode Invariant Generation

Our technique automatically generates mode invariants from propositional formulas derived from a mode machine and constraints on the input variables associated with that mode machine. To compute the mode invariants for a mode class M, we first identify the set of atomic conditions appearing in the events of the mode transition table for M. For example, in the Cruise Control specification, we have $I \equiv$ `IgnOn`, $E \equiv$ `EngRunning`, $B \equiv$ `Brake`, $O \equiv$ `Lever=off`, $C \equiv$ `Lever=const`, $R \equiv$ `Lever=resume`, and $L \equiv$ `Lever=release`.[1] Below, the term *literal* refers to either an atomic condition or its negation. The algorithm consists of the following three steps:

1. For each mode m, compute the *mode entry condition* $N(m)$, the disjunction of the conditions true upon entry into mode m from other modes or upon entry into an initial state when $M = m$.
2. For each mode m, compute the *unconditional exit set* $X(m)$, where $X(m)$ is the set of literals whose falsification cause unconditional exit from m.
3. For each mode m, compute the *mode invariant* $P(m)$ by eliminating from each disjunct in $N(m)$

[1] All four values of `Lever` must be considered, even though the table mentions only three of them.

Old Mode	Event	New Mode
1 Off	@T(IgnOn)	Inactive
2 Inactive	@F(IgnOn)	Off
3 Inactive	@T(Lever = const) WHEN IgnOn AND 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 IgnOn AND EngRunning AND NOT Brake OR @T(Lever = const) WHEN IgnOn AND EngRunning AND NOT Brake	Cruise

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

Table 1: Mode Transition Table for Cruise Control.

all literals that are not members of $X(m)$. More precisely, replace each literal that is not in $X(m)$ by *true*.

In the examples below, only an intuitive special case of step 3 is needed: that is, $M = m \Rightarrow c$ is a mode invariant if c is true in each disjunct of $N(m)$ and c is a conjunction of literals in $X(m)$.

The algorithm repeats these three steps until a fixpoint is reached. Let $N_i(m)$, $X_i(m)$, and $P_i(m)$ represent the values of the mode entry condition, the unconditional exit set, and the invariant for mode m at the end of the ith pass of the algorithm. During each pass of the algorithm, the information in the table as well as a number of additional facts may be used to strengthen the invariant computed at that pass. The additional facts include the initial state predicate (a predicate describing the states $s \in S_0$), environmental constraints, such as the One Input Assumption and 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 if an enumerated type variable has one value, it cannot have other values. For example, in the Cruise Control System, if `Lever` has the value `const`, it cannot have any other value; more precisely, $C \Leftrightarrow \overline{O} \wedge \overline{R} \wedge \overline{L}$.

Table 2 summarizes the results of applying the algorithm to the mode transition table shown in Table 1. Applying the algorithm generates the four invariants listed at the end of the previous section. For each pass i, Table 2 shows the mode entry condition $N_i(m)$, the unconditional exit set $X_i(m)$, and the invariant $P_i(m)$ computed during that pass for each of the four modes m in the mode class M. For each mode m and each pass i, the table identifies the additional facts that were used to strengthen the invariant. Table 2 shows that, for this example, four passes are needed to reach a fixpoint.

Below, we describe how the information in Table 2 and the additional facts described above are used to compute the four mode invariants. Although each step of the algorithm is actually applied to all modes at once, below we simplify our description of the algorithm by treating one mode at a time. Generating the invariant for the mode `Off` uses information from Table 1 as well as the initial state predicate. Generating the invariant for the mode `Cruise` shows how the One Input Assumption and the constraints on an enumerated type variable are used to strengthen the mode entry condition computed from Table 1, which in turn strengthens the computed invariant. In generating the invariant for the mode `Override`, an invariant generated on the first pass for a different mode is used to strengthen the mode entry condition computed in the second pass. Then, the strengthened mode entry condition is used to strengthen the computed invariant. Computing the strongest invariant for the mode `Inactive` requires three passes of the algorithm. In the second and third passes, invariants generated for other modes during the first and second passes are used to strengthen the mode entry condition and subsequently the mode invariant for `Inactive`.

To apply the algorithm to the mode `Off`, we first analyze rows 2, 4, and 7, the three rows of Table 1 that cause the system to enter the `Off` mode. In each case, the condition that holds upon entry into `Off` is $\neg$`IgnOn`, denoted as $\overline{I}$. Next, because $M =$ `Off` holds in the initial state, we can also include part of the initial state predicate (namely, $\neg$`IgnOn` $\wedge$ $\neg$`EngRunning`, denoted as $\overline{I} \wedge \overline{E}$) in the mode entry condition. Thus, the mode entry condition is $N_1(\texttt{Off}) = \overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I} \wedge \overline{E} = \overline{I}$. In the second step, we analyze row 1, the only row of Table 1 that describes an exit from `Off`, to compute the unconditional exit set $X_1(\texttt{Off})$. The only condition whose falsification causes unconditional exit from `Off` is $\neg$`IgnOn`. Hence, $X_1(\texttt{Off}) = \{\overline{I}\}$. In the third step, we restrict the mode entry condition to the members of the unconditional exit set to obtain $P_1(\texttt{Off}) = \overline{I}$, and hence the mode invariant $M = \texttt{Off} \Rightarrow \neg\texttt{IgnOn}$.

i	Mode m	$N_i(m)$	$X_i(m)$	$P_i(m)$	Comments
1	`Off`	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I} \wedge \overline{E}$	$\{\overline{I}\}$	$\overline{I}$	ISP gives 4th DJ
	`Inactive`	$I \vee \overline{E} \vee \overline{E}$	$\{I\}$	$true$	—
	`Override`	$B \vee O$	$\{I, E\}$	$true$	—
	`Cruise`	$C \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee R \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{C} \wedge \overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I \wedge E \wedge \overline{B} \wedge \overline{O}$	Apply OIA, CET
2	`Off`	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I} \wedge \overline{E}$	$\{\overline{I}\}$	$\overline{I}$	Fixpoint reached?
	`Inactive`	$I \vee \overline{E} \wedge I \wedge \overline{O} \wedge \overline{B} \vee \overline{E}$	$\{I\}$	$true$	Apply P_1(`Cruise`), OIA to 2nd DJ
	`Override`	$B \wedge I \wedge E \wedge \overline{O} \vee O \wedge I \wedge E \wedge \overline{B} \wedge \overline{C} \wedge \overline{R} \wedge \overline{L}$	$\{I, E\}$	$I \wedge E$	Apply P_1(`Cruise`), OIA, & CET
	`Cruise`	$C \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee R \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{C} \wedge \overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I \wedge E \wedge \overline{B} \wedge \overline{O}$	Fixpoint reached?
3	`Off`	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I} \wedge \overline{E}$	$\{\overline{I}\}$	$\overline{I}$	Fixpoint already reached?
	`Inactive`	$I \vee \overline{E} \wedge I \wedge \overline{O} \wedge \overline{B} \vee \overline{E} \wedge I$	$\{I\}$	I	Apply P_2(`Override`), OIA to 3rd DJ
	`Override`	$B \wedge I \wedge E \wedge \overline{O} \vee O \wedge I \wedge E \wedge \overline{B} \wedge \overline{C} \wedge \overline{R} \wedge \overline{L}$	$\{I, E\}$	$I \wedge E$	Fixpoint reached?
	`Cruise`	$C \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee R \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{C} \wedge \overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I \wedge E \wedge \overline{B} \wedge \overline{O}$	Fixpoint already reached?
4	`Off`	$\overline{I} \vee \overline{I} \vee \overline{I} \vee \overline{I} \wedge \overline{E}$	$\{\overline{I}\}$	$\overline{I}$	Fixpoint reached!
	`Inactive`	$I \vee \overline{E} \wedge I \wedge \overline{O} \wedge \overline{B} \vee \overline{E} \wedge I$	$\{I\}$	I	Fixpoint reached!
	`Override`	$B \wedge I \wedge E \wedge \overline{O} \vee O \wedge I \wedge E \wedge \overline{B} \wedge \overline{C} \wedge \overline{R} \wedge \overline{L}$	$\{I, E\}$	$I \wedge E$	Fixpoint reached!
	`Cruise`	$C \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee R \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{C} \wedge \overline{L}$	$\{I, E, \overline{B}, \overline{O}\}$	$I \wedge E \wedge \overline{B} \wedge \overline{O}$	Fixpoint reached!

Key

ISP:	Initial State Predicate	I:	`IgnOn`
OIA:	One Input Assumption	E:	`EngRunning`
CET:	Constraint from Enumerated Type	B:	`Brake`
$N_i(m)$:	Mode Entry Condition for Mode m at ith pass	O:	`Lever =off`
$X_i(m)$:	Unconditional Exit Set for Mode m at ith pass	C:	`Lever = const`
$P_i(m)$:	Invariant computed for Mode m at ith pass	R:	`Lever = resume`
DJ:	Disjunct of $N_i(m)$	L:	`Lever = release`

Table 2: Mode Invariant Generation for Cruise Control

Next, we use our algorithm to generate a mode invariant for the mode `Cruise`. First, we use rows 3 and 9, the two rows of Table 1 that cause entry into `Cruise`, to compute the mode entry condition. The One Input Assumption guarantees that the conditions in the WHEN clauses remain true upon entry into `Cruise`; hence, for example, the conditioned event in row 3 and the One Input Assumption imply that, upon entry into mode `Cruise`, the condition $C \wedge I \wedge E \wedge \overline{B}$ holds. Thus, the mode entry condition is

$$N_1(\texttt{Cruise}) = C \wedge I \wedge E \wedge \overline{B} \;\; \vee \;\; [R \wedge I \wedge E \wedge \overline{B} \;\vee\; C \wedge I \wedge E \wedge \overline{B}].$$

Further, because `Lever` is an enumerated type, only one of the atomic conditions, O, C, R, and L, can be true at a given time. Hence, constraints, such as $C \Leftrightarrow \overline{O} \wedge \overline{R} \wedge \overline{L}$, can be used to strengthen the mode entry condition N_1(`Cruise`), i.e.,

$$N_1(\texttt{Cruise}) = C \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{R} \wedge \overline{L} \vee R \wedge I \wedge E \wedge \overline{B} \wedge \overline{O} \wedge \overline{C} \wedge \overline{L}.^{2}$$

In the second step, we use rows 4-6 of Table 1 to compute the unconditional exit set $X_1(\texttt{Cruise}) = \{I, E, \overline{B}, \overline{O}\}$. Finally, eliminating all literals not in the unconditional exit set from the mode entry condition produces $P_1(\texttt{Cruise}) = I \wedge E \wedge \overline{B} \wedge \overline{O}$. This is equivalent to the mode invariant

$$M = \texttt{Cruise} \;\Rightarrow\; \texttt{IgnOn} \wedge \texttt{EngRunning} \wedge \neg\texttt{Brake} \wedge \texttt{Lever} \neq \texttt{off}. \quad (1)$$

In generating an invariant for the mode `Override`, the first pass of the algorithm uses row 6 of Table 1 to produce $N_1(\texttt{Override}) = B \vee O$ and rows 7, 8, and 9 to produce $X_1(\texttt{Override}) = \{I, E\}$. Because no literals in the unconditional exit set appear in the mode entry condition, after the first pass, P_1(`Override`) is trivially $true$. On the second pass, the mode entry condition can be strengthened by recognizing that the only mode from which `Override` can be entered is `Cruise` (see row 6). Applying the invariant in (1) generated for the mode Cruise during the first pass, the One Input Assumption and the constraint on the enumerated type `Lever` strengthens the mode entry condition, i.e.,

$$N_2(\texttt{Override}) = B \wedge I \wedge E \wedge \overline{O} \;\vee\; O \wedge I \wedge E \wedge \overline{B} \wedge \overline{C} \wedge \overline{R} \wedge \overline{L}.$$

Finally, restricting the mode entry condition to the members of $X_1(\texttt{Override}) = \{I, E\}$ produces $P_2(\texttt{Override}) = I \wedge E$, i.e., the invariant

$$M = \texttt{Override} \Rightarrow \texttt{IgnOn} \wedge \texttt{EngRunning}. \quad (2)$$

To generate an invariant for the mode `Inactive`, rows 1, 5, and 8 of Table 1 are used to compute

[2] For readability, the form of this condition has been simplified. When used to strengthen the invariant based on previously computed invariants, the condition must be expressed in a form that distinguishes the source modes.

Mode	Events	
High	False	@F(Pressure = High)
TooLow, Permitted	@T(Block = On) WHEN Reset = Off	@T(Pressure = High) OR @T(Reset = On)
Overridden	True	False

Table 3: Event Table for Overridden in Standard Format.

Old Value	Event	New Value
FALSE	@T(Block = On) WHEN Reset = Off AND Pressure ≠ High	TRUE
TRUE	@T(Reset = On) WHEN Pressure ≠ High OR @T(Pressure = High) OR @F(Pressure = High)	FALSE

Table 4: Event Table for Overridden Rewritten as a Mode Transition Table.

$N_1(\texttt{Inactive}) = I \vee \overline{E} \vee \overline{E}$ and rows 2 and 3 to compute $X_1(\texttt{Inactive}) = \{I\}$. In step 3, we note that $\overline{E}$, which appears as the second disjunct in $N_1(\texttt{Inactive})$, does not appear in the unconditional exit set $\{I\}$. Hence, we replace $\overline{E}$ with *true* in $N_1(\texttt{Inactive})$, thus producing the trivial invariant $P_1(\texttt{Inactive}) = \mathit{true}$. The second pass uses the One Input Assumption and the invariant (1) computed during the first pass for Cruise to strengthen the mode entry condition, that is, $N_2(\texttt{Inactive}) = I \vee [\overline{E} \wedge I \wedge \overline{O} \wedge \overline{B}] \vee \overline{E}$. (The third disjunct $\overline{E}$, the mode entry condition when the current mode is Override, cannot be strengthened because the invariant computed for Override during pass 1 is *true*.) Applying step 3 at the second pass produces no change in the mode invariant. Finally, on the third pass, the One Input Assumption and the invariant (2), computed for Override during the second pass, can be used to rewrite the mode entry condition as $N_3(\texttt{Inactive}) = I \vee [\overline{E} \wedge I \wedge \overline{O} \wedge \overline{B}] \vee [\overline{E} \wedge I]$. Restricting the mode entry condition to the single member I of the unconditional exit set produces $P_3(\texttt{Inactive}) = I$, which is equivalent to the mode invariant $M = \texttt{Inactive} \Rightarrow \texttt{IgnOn}$.

An analysis of Table 2 shows that, for each mode m, the exit set computed at pass 1, $X_1(m)$, predicts the invariant computed at pass 4, $P_4(m)$. As the example in the next section shows, this is generally not the case.

In the example shown in Table 2, reaching a fixpoint requires four passes. The number of needed passes can often be reduced by computing the invariants in a different order and applying an invariant as soon as it is computed rather than waiting until the next pass. To illustrate this approach in the Cruise Control example, we use an alternate ordering: Off, Cruise, Override, and Inactive. During pass 1, the mode invariant computed earlier for Cruise can be used to strengthen the mode entry condition for Override and the mode invariants for Cruise and Override to strengthen the mode entry condition for Inactive. This leads to strengthened mode invariants at the first pass, rather than later passes, with the fixpoint reached during the second pass.

5 Generating Invariants from Other Tables

The previous section described our algorithm for generating mode invariants from mode transition tables. This section shows how this algorithm can be used to generate state invariants from event tables and also presents an example of a state invariant derived from a condition table. Because the invariant is easily derived from the semantics of condition tables, applying our algorithm is unnecessary. We also show by example how these generated invariants may be used to prove additional invariants.

Consider the event table in Table 3, part of a requirements specification for a simple system controlling safety injection in a nuclear plant. (This table, equivalent to a similar table in [17], avoids the "Inmode" notation.) Table 3, which describes when safety injection is overridden, can be viewed as a simple SCR system Σ whose monitored variables are Block, Reset, and Pressure and whose single controlled variable is Overridden. In the initial states of the system, $\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \mathit{false} \wedge \texttt{SafetyInjection} = \texttt{On}$.

Before our algorithm can be applied to an event table, the table must be represented in the format of a mode transition table. To accomplish this, we first treat each mode in the first column of the event table as an additional condition in the WHEN clause of conditioned events in the appropriate row. Then, the table is rewritten to describe the variable transitions—how the variable defined by the table changes from one value to any other possible value. If a variable has n possible values, there are $n^2 \Leftrightarrow n$ possible transitions (excluding self-transitions). In the case of Table 3, the variable Overridden has only two values, so only two transitions are needed, the transition from TRUE to FALSE[3] and vice versa. Rewriting the event table in Table 3 in the form of a mode transition table produces Table 4.

To generate a state invariant involving Overridden,

[3] To avoid confusion with the truth values *false* and *true*, we denote the values of Overridden as FALSE and TRUE.

three atomic conditions are defined: $R \equiv$ `Reset=On`, $B \equiv$ `Block=On`, and $H \equiv$ `Pressure=High`. (The negations of B and R have the obvious meaning; e.g., $\overline{B} \equiv$ `Block=Off`.) Applying the algorithm when `Overridden` has the value `FALSE` computes the uninteresting $P_1(\texttt{FALSE}) = \texttt{true}$.

On the first pass of the algorithm when `Overridden` is `TRUE`, we compute $N_1(\texttt{TRUE}) = B \wedge \overline{R} \wedge \overline{H}$ (assuming the One Input Assumption) and $X_1(\texttt{TRUE}) = \{H, \overline{H}\}$. This yields $P_1(\texttt{TRUE}) = \overline{H}$, i.e., the state invariant

$$\texttt{Overridden} = \texttt{TRUE} \Rightarrow \texttt{Pressure} \neq \texttt{High}.$$

On the second pass, the mode entry condition cannot be strengthened, but the invariant computed on the first pass allows us to revise the unconditional exit set. Since $\texttt{Overridden} = \texttt{TRUE} \Rightarrow \overline{H}$ is an invariant, we deduce from Table 4 that the event @T(`Reset=On`) causes unconditional exit from `TRUE`. Hence, on the second pass, the unconditional exit set $X_2(\texttt{TRUE})$ is $\{H, \overline{H}, \overline{R}\}$, which produces the strengthened invariant $P_2(\texttt{TRUE}) = \overline{H} \wedge \overline{R}$, i.e.,

$$\begin{aligned} \texttt{Overridden} &= \texttt{TRUE} \Rightarrow \\ \texttt{Pressure} &\neq \texttt{High} \wedge \texttt{Reset} = \texttt{Off}. \end{aligned} \quad (3)$$

Mode	Conditions	
High, Permitted	True	False
TooLow	Overridden	NOT Overridden
Safety Injection	Off	On

Table 5: Condition Table for `Safety Injection`.

Table 5 is a condition table, taken from the same specification as Table 3, which specifies when the system turns safety injection on and off. The semantics of condition tables presented in [17] requires the conditions c_i in each row of the table to satisfy two properties: the disjunction of the c_i is true, and the pairwise conjunction of c_i and c_j, $i \neq j$, is false. Using this semantics along with the assumption about initial states, we can easily derive the following state invariants from Table 5,

$$\begin{aligned} \texttt{SafetyInjection} &= \texttt{On} \Leftrightarrow \\ \texttt{Pressure} &= \texttt{TooLow} \wedge \neg\texttt{Overridden}, \end{aligned} \quad (4)$$

and its equivalent form,

$$\begin{aligned} \texttt{SafetyInjection} &= \texttt{Off} \Leftrightarrow \\ \texttt{Pressure} &\neq \texttt{TooLow} \vee \texttt{Overridden}. \end{aligned}$$

In [6], the following two properties of the Safety Injection System are proved using model checking:

Property X: $\texttt{Reset} = \texttt{On} \wedge \texttt{Pressure} \neq \texttt{High} \Rightarrow \neg\texttt{Overridden}$

Property Y: $\texttt{Reset} = \texttt{On} \wedge \texttt{Pressure} = \texttt{TooLow} \Rightarrow \texttt{SafetyInjection} = \texttt{On}$

Property X is easily derived from the invariant in (3), since (3) is stronger. Moreover, Property Y follows directly from (3) and (4). This result suggests that our invariant generation algorithm can, at times, supplement other techniques, such as model checking, in verifying properties of state machine models.

6 Generalizing the Algorithm

This section generalizes our algorithm by describing formally how the algorithm can be applied to general state machine models. The current SCR requirements model is a special case of this general model. The general model allows the transform T to be nondeterministic, that is, a relation rather than a function, and makes very general assumptions about the environment—the One Input Assumption and NAT constraints of the current SCR model are special cases. Further, the events defining transitions are not limited to the special unconditioned event form found in SCR tables.

Our general algorithm for generating state invariants can be applied to other state machine models. For example, we have applied the algorithm to two SCR specifications analyzed by Atlee and Gannon [4], whose SCR semantics omits the One Input Assumption, and corroborated their results.[4] The algorithm also applies to models, such as TLA [24] and STeP [27], whose transitions are expressed as changes in one or more system variables. In other models, such as Statecharts [14] and RSML (Requirements State Machine Language) [25], which include hierarchical states and internal events, the algorithm is also applicable but due to the complex step semantics of these two models, applying the algorithm would be less straightforward. A recent paper by Park et al. [31] discusses the generation of invariants from RSML specifications (see Section 7).

6.1 Mode Machines as Abstract State Machines

We consider 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 represented as (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

[4] In later work, Sreemani and Atlee [33] use a semantics for SCR equivalent to ours, adopting the One Input Assumption and our WHEN semantics.

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.

Consider two state machines, $\Sigma = (S, \Theta, \rho)$ and $\Sigma_A = (S_A, \Theta_A, \rho_A)$. Then, Σ_A is an *abstraction* of Σ if there is a map $\alpha : S \rightarrow S_A$, $s \stackrel{\alpha}{\mapsto} s_A$, called the *abstraction map*, such that (a) for all s in S: $\Theta(s) \Rightarrow \Theta_A(s_A)$ and (b) for all s, s' in S: $\rho(s,s') \Rightarrow \rho_A(s_A, s'_A)$. A mode machine is an example of an abstract state machine Σ_A. The original specification, which includes the mode machine as a component, describes the state machine Σ.

We guarantee that a mode invariant q_A computed for a mode machine Σ_A has a corresponding mode invariant $Z(q_A)$ in the overall state machine Σ if the following theorem is satisfied:

Theorem 1 *Let $\Sigma = (S, \Theta, \rho)$ and $\Sigma_A = (S_A, \Theta_A, \rho_A)$ be two state machines, and let α be an abstraction map from S to S_A. If condition q_A is an invariant for Σ_A, then $Z(q_A)$ is an invariant for Σ where $Z(q_A) = \{\ s \mid q_A(s_A)\ \}$.*

This theorem is a special case of Theorem 1.1, Part 1, in [2]. It is also a special case of Corollary 5.7 in [9], which is generalized in [26].

To obtain the abstraction map α, we can often apply the three abstraction methods for SCR systems described in [6, 15] to the specification of the state machine Σ to obtain the specification of the mode machine Σ_A. Abstraction Method 1 eliminates all variables, except those on which the mode class depends. Abstraction Method 2 removes detailed monitored variables (i.e., variables with large ranges of values), while Abstraction Method 3 replaces detailed variables (perhaps with infinitely many values) with more abstract, finite-valued variables. Encoding the variables as atomic conditions is then required. Normally, we encode a finite type using one atomic condition for each value of the type.

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, where each $e \in E_A$ is represented as a logical formula over the encoded atomic conditions. Then, the mode machine for the mode class M is defined by four constructs: the relation Υ_A describing the mode transitions, the initial state predicate Θ_A, and two constraints C_1 and C_2 on the monitored variables. C_1 and C_2 capture the environmental constraints described in the previous examples. The constructs Υ_A, Θ_A, C_1, and C_2 are represented as follows:

- Υ_A is a relation on $TY(M) \times E_A \times TY(M)$. In SCR specifications, this relation is represented by the encoded form of the mode transition table for M. We assume that Υ_A omits 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 as

$$\theta_A(m) = \Theta_A|_{M:=m},$$

  where $\Theta_A|_{M:=m}$ is Θ_A with all appearances of the variable M replaced with m. For example, in the Cruise Control System, $\Theta_A \stackrel{\text{def}}{=} M = \texttt{Off} \wedge \overline{I} \wedge \overline{E}$; therefore, $\theta_A(\texttt{Off}) = \overline{I} \wedge \overline{E}$, and $\theta_A(m) = \mathit{false}$ otherwise.

- 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 booleans. For example, in the Cruise Control System, C_1 is the axiom

$$\begin{array}{lll} (C \Leftrightarrow \overline{O} \wedge \overline{R} \wedge \overline{L}) & \wedge & (O \Leftrightarrow \overline{C} \wedge \overline{R} \wedge \overline{L}) \quad \wedge \\ (R \Leftrightarrow \overline{O} \wedge \overline{C} \wedge \overline{L}) & \wedge & (L \Leftrightarrow \overline{O} \wedge \overline{C} \wedge \overline{R}). \end{array}$$

  Other constraints are derived from NAT; for example, in the Cruise Control System, a possible physical constraint (not used in our case) is that $E \Rightarrow I$ (i.e., $\texttt{EngRunning} \Rightarrow \texttt{IgnOn}$ [4]).

- C_2 is a conjunction of encoded constraints on monitored variables *in two consecutive states*. One possible constraint C_2 for the Cruise Control system is the One Input Assumption. Other possibilities are physical constraints; one example (not used here) is the encoded version of the constraint: when $\texttt{Lever} \neq \texttt{release}$ and Lever changes, the only possible transition is $\texttt{Lever}' = \texttt{release}$.

We have shown that with some restrictions (easily met in practice) that the state machine Σ_A defined by the above constructs satisfies Theorem 1 [22].

6.2 Details of Mode Invariant Generation

In addition to the four constructs defined above, our algorithm uses three functions—*NEW*, *EX*, and *KEEP*. To compute the the mode entry condition, Step 1 uses *NEW*, which extracts the new state information from a two-state predicate. To compute the unconditional exit set, Step 2 uses *EX*, which describes the events causing exit from a mode. Finally, to compute the mode invariant, Step 3 uses *KEEP*, a projection operator. Below, we describe these three functions and then use them to define the generalized version of our algorithm.

The function *NEW* has a single argument q, a predicate on two states expressed in Disjunctive Form. More precisely, q is the disjunction of non-false terms, each of which is either *true* or the conjunction of one or more

literals ℓ or ℓ'. (Any two-state predicate can be expressed in Disjunctive Form, since any two-state predicate can be expressed in standard disjunctive normal form, a special case.) The function *NEW* computes the strongest condition known to be true in the new state. Applying *NEW* to a two-state predicate simply replaces each old state literal with *true* and suppresses the primes on the remaining new state literals. For example, the following shows the application of *NEW* to the conjunction of the event in the first line of Table 4 and an appropriate part of the One Input Assumption (shown in brackets):

$$\begin{aligned} &NEW((@\mathrm{T}(B) \text{ WHEN } \overline{R}\wedge\overline{H}) \wedge \\ &[B' \neq B \Rightarrow R' = R \ \wedge \ H' = H]) \ \ = \\ &\qquad NEW(\overline{B}\wedge B'\wedge\overline{R}\wedge\overline{H}\wedge\overline{R}'\wedge\overline{H}') \ \ = \ \ B\wedge\overline{R}\wedge\overline{H}. \end{aligned}$$

For the formal definition of *NEW*, see the Appendix.

The function *EX* is a two-state predicate which describes the events causing exit from a mode as a disjunction. For example, in the Cruise Control System, lines 2 and 3 of Table 1 show that $EX(\texttt{Inactive})$ should be defined as

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

Formally, EX is defined by

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

The function *KEEP* has two arguments, a set U of literals and a condition c (i.e., a one-state predicate) expressed in Disjunctive Form. Then, $KEEP(U,c)$ is c with all literals ℓ that are not in U replaced by *true*. For example, consider $U = X_2(\texttt{Override})$ and $c = N_2(\texttt{Override})$ from Table 2:

$$\begin{aligned} &KEEP(\{I,E\}, B\wedge I\wedge E\wedge\overline{O} \ \vee \ O\wedge I\wedge E\wedge\overline{B}\wedge\overline{C}\wedge\overline{R}\wedge\overline{L}) = \\ &\qquad I\wedge E \ \vee \ I\wedge E = I\wedge E \end{aligned}$$

For the formal definition of *KEEP*, see the Appendix.

The mode entry condition $N_i(m)$ for a given mode m at the ith pass is defined in terms of $\theta_A(m)$, the invariants computed on the previous pass, the constraints C_1 and C_2, and the events e causing entry into m. Formally, $N_i(m)$ is defined by

$$\begin{aligned} N_i(m) &= \theta_A(m) \wedge C_1 \\ &\vee \left(\bigvee_{\hat{m},e:\Upsilon_A(\hat{m},e,m)} NEW(P_{i-1}(\hat{m}) \wedge C_2 \wedge e) \right) \wedge C_1. \end{aligned}$$

To demonstrate that this definition correctly captures our intuitive notion of "what is known upon mode entry," a more formal computation of the mode entry condition $N_2(\texttt{Override})$ for the Cruise Control follows:

$$\begin{aligned} N_2(\texttt{Override}) &= NEW[(P_1(\texttt{Cruise}) \wedge C_2 \\ &\quad \wedge(@\mathrm{T}(B) \vee @\mathrm{T}(O))] \wedge C_1 \\ &= NEW[I\wedge E\wedge\overline{B}\wedge\overline{O} \wedge C_2 \wedge (\overline{B}\wedge B' \vee \overline{O}\wedge O')] \wedge C_1 \\ &= NEW[\overline{B}\wedge B'\wedge I\wedge E\wedge\overline{O}\wedge I'\wedge E'\wedge\overline{O}' \\ &\quad \vee \ \overline{O}\wedge O'\wedge I\wedge E\wedge\overline{B}\wedge I'\wedge E'\wedge\overline{B}'] \wedge C_1 \\ &= [B\wedge I\wedge E\wedge\overline{O} \vee O\wedge I\wedge E\wedge\overline{B}] \wedge C_1 \\ &= B\wedge I\wedge E\wedge\overline{O} \vee O\wedge I\wedge E\wedge\overline{B}\wedge\overline{C}\wedge\overline{R}\wedge\overline{L} \end{aligned}$$

The unconditional exit set $X_i(m)$ for a given mode m at the ith pass is computed using the events $@\mathrm{F}(\ell)$ that cause exit from m, the invariant $P_{i-1}(m)$ computed on the previous pass, the constraints C_2, and the function *EX*. Formally, $X_i(m)$ is defined by

$$\begin{aligned} X_i(m) = \{\ell \ \mid \ &@\mathrm{F}(\ell) \wedge P_{i-1}(m) \\ &\wedge P_{i-1}(m)' \wedge C_2 \Rightarrow EX(m)\}. \end{aligned}$$

To explain this definition, we first consider the simpler $@\mathrm{F}(\ell) \Rightarrow EX(m)$. This states that, for each $\ell \in X_i(m)$, $@\mathrm{F}(\ell)$ is either impossible (thus making the implication vacuously true) or its occurrence must cause exit from mode m. However, we can strengthen this simple form by applying additional facts about the system when in mode m, i.e., $P_{i-1}(m)\wedge P_{i-1}(m)'\wedge C_2$. The inclusion of $P_{i-1}(m)'$ is rather subtle since it seems that we don't know that $M = m$ in the new state. However, if $\neg P_{i-1}(m)'$ then we know that $EX(m)$ holds so we don't need to consider that alternative and are left with $P_{i-1}(m)'$.

As an example, consider the computation of the invariant for $\texttt{Overridden} = \mathit{TRUE}$ in the Safety Injection system. What follows is the proof that $\texttt{Reset} = \texttt{off}$ is in the unconditional exit set computed during the second pass, i.e., $\overline{R} \in X_2(\mathit{TRUE})$:

$$\begin{aligned} &\overline{R} \in X_2(\mathit{TRUE}) \\ &\quad \Leftrightarrow \ @\mathrm{F}(\overline{R}) \wedge P_1(\mathit{TRUE}) \wedge P_1(\mathit{TRUE})' \wedge C_2 \\ &\qquad \stackrel{?}{\Rightarrow} EX(\mathit{TRUE}) \\ &\quad \Leftrightarrow \ @\mathrm{T}(R) \wedge \overline{H} \wedge \overline{H}' \wedge C_2 \\ &\qquad \Rightarrow @\mathrm{T}(R) \wedge \overline{H} \ \vee \ @\mathrm{T}(H) \ \vee \ @\mathrm{F}(H) \end{aligned}$$

Given the mode entry condition $N_i(m)$ and the unconditional exit set $X_i(m)$ at the ith pass, we can now compute the invariant $P_i(m)$ at the ith pass using the *KEEP* operator and the constraints C_1. Formally,

$$P_i(m) = KEEP(X_i(m), N_i(m)) \wedge C_1.$$

That *KEEP* computes a mode invariant in this equation is based upon the following intuition: Consider the simplest case when the mode entry condition (in Disjunctive Form) is a single conjunction of literals. Applying

the *KEEP* operator produces $P_i(m)$, a conjunction of literals found in $X_i(m)$. First, we note that $P_i(m)$ must be true upon entry into mode m (the *KEEP* construction ensures that $N_i(m) \Rightarrow P_i(m)$). Then, $P_i(m)$ must be a mode invariant, for if not then there must be some transition that falsifies $P_i(m)$ but leaves $M = m$. This is impossible because falsification of $P_i(m)$ requires at least one of the literals in $P_i(m)$ (i.e., some $\ell \in X_i(m)$) to become false, which means that the system must exit m. In generalizing from the simplest case, we require the disjunction of the above technique over all alternative possibilities.

To complete our description of the algorithm, we define the initial case

$$P_0(m) = C_1.$$

That is, the mode invariant is simply C_1 initially, and we iterate computing $P_i(m)$ for each m until a fixpoint is reached, i.e., when there exists n such that the $P_{n+1}(m)$ for each m computed at step $n+1$ equals the $P_n(m)$ computed at step n. The major result that we have proved is that the algorithm computes mode invariants for Σ_A (see the Appendix for the proof):

Theorem 2 *$M = m \Rightarrow P_i(m)$ is a mode invariant for Σ_A for each m and each pass i. Furthermore, $(M = m \Rightarrow P_i(m)) \leq (M = m \Rightarrow P_{i-1}(m))$, with at least one invariant strengthened on each pass i before the fixpoint is reached.*

As a corollary to the proof of the major result, we have the following simple test that a literal ℓ is a mode invariant (Theorem 3.1 from [3]):

Corollary 1 *$M = m \Rightarrow \ell$ is a mode invariant of mode m of Σ_A if (a) ℓ is always true when mode m is entered, and (b) event @F(ℓ) causes an unconditional exit from mode m.*

Further, if Theorem 1 holds, then the generated mode invariants can be translated into mode invariants for the original state machine Σ.

This algorithm sacrifices completeness, i.e., the ability to generate the *strongest* invariants, for ease of computation. While our algorithm makes it easy to compute an invariant, it does not necessarily produce the strongest invariant, i.e., the invariant that would result from a complete reachability analysis. An additional source of incompleteness is that the environmental assumptions may be too weak; e.g., the precise environmental constraints may not be known, so a conservative approximation is used instead. Incompleteness is further discussed in [22].

7 Related Work

Our technique for generating invariants from SCR specifications extends work by Atlee and Gannon [3, 4], who used hand-computed mode invariants in their analysis of SCR specifications using the MCB model checker [8]. However, their technique was not automatable and only addressed a special case of our general algorithm. They derived mode invariants where the $P_i(m)$ were conjunctions of conditions (rather than general expressions), handled limited event expressions (rather than general events), and handled only special cases of the environmental constraints C_1 and C_2.

Mode invariants are analogous to local invariants, where a *local invariant* $PC = i \Rightarrow I(i)$ is a property that holds when a program is in location i. In particular, mode invariants are related to the subclass of "reaffirmed invariants": local invariants defined on data variables [27]. The Stanford Temporal Prover [27] automatically generates such invariants.

Bensalem and his colleagues have recently refined techniques for generating local invariants [5]. However, their generation process is considerably different from ours. They first generate invariants that hold for each process in isolation. This consists of "generalized reaffirmed invariants without cycles" which are analogous to our computation of mode entry conditions. Next, these invariants are propagated within each isolated process. Finally, the invariants from the isolated processes are combined into overall system invariants. In contrast, we concentrate on a single process (a mode machine) with effects of other processes expressed by the constraints C_1 and C_2. After computing the mode entry conditions, we obtain overall system invariants via the *KEEP* operator. Our iterations propagate these invariants throughout the system. They also consider additional techniques, such as reaffirmed invariants with cycles, outside the scope of our generation process.

In recent years, there has been a resurgence of interest in the automatic generation of invariants in conjunction with advances in automated proof techniques [5, 7, 13, 27, 34]. These methods may be classified as "bottom-up" or "top-down" [7]. Bottom-up methods, which generate local program invariants and our mode invariants, derive the invariants automatically from the state machine specification. Top-down methods start with a candidate invariant and use this to determine an invariant that is no weaker (if the candidate is indeed invariant). The method used by Park et al. [31] to generate invariants to aid in consistency checking of RSML [25] specifications is top-down. While our technique generates only simple invariants, the generation of general safety properties (properties using temporal operators that refer to the evolution of the system over more than one state) has also been investigated [7].

8 Conclusions and Future Work

This paper describes how state invariants can be automatically derived from SCR specifications without generating a representation of the complete state space of the system. The algorithm provides successively better invariants at each pass (not just approximations), so that valid invariants are obtained even if the algorithm is not run to completion.

To illustrate the utility of our approach, we used invariants derived by our algorithm to prove two safety properties of the Safety Injection System. This result suggests that our algorithm can usefully supplement other techniques, such as model checking, in verifying properties of state machine models. We envision a development environment that offers many complementary techniques—to include model checking, invariant generation, and theorem proving. For some problems, using one technique will be more cost-effective than using others; for other problems, two or more techniques may be useful in concert.

We have explored a version of the algorithm which does not require the boolean encoding of events. This algorithm is more complete, and therefore generates stronger invariants, but is considerably less efficient. We are also exploring the extension of our algorithm to more general invariants than state invariants. Moreover, we are investigating the use of our abstraction methods [2, 6, 15], which were originally designed to build the abstract machine Σ_A from a property of interest q, to automatically construct the mode machine needed to generate state invariants from an SCR specification.

We have developed a prototype tool that uses our algorithm to automatically generate state invariants from SCR requirements specifications and applied it to the three mode transition tables in an updated version of the A-7 requirements document [1]. Together the tables contain a total of 700 rows and 46 modes. In less than five minutes, the tool generated over 20 "interesting" invariants, that is, invariants not equal to *true*, that could be presented to system users for validation. These preliminary results demonstrate the algorithm's potential efficiency and practical utility.

Acknowledgments

We gratefully acknowledge our colleagues, Myla Archer, Ramesh Bharadwaj, Steve Sims, and Jim Kirby; Neil Immerman of the University of Massachusetts; and Axel van Lamsweerde and the anonymous referees whose constructive comments helped us improve the paper. We also thank Bruce Labaw and Russ Beall for building the prototype tool that implements our algorithm.

References

[1] Alspaugh, T. A., Faulk, S. R., Britton, K. H., Parker, R. A., Parnas, D. L., and Shore, J. E. Software requirements for the A-7E aircraft. Tech. Rep. NRL-9194, Naval Research Lab., Wash., DC, 1992.

[2] Archer, M., and Heitmeyer, C. The use of model checking and abstraction in analyzing requirements specifications: A formal foundation. Tech. rep., Naval Research Lab., Washington, DC, 1998. (Draft).

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

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

[5] Bensalem, S., Lakhnech, Y., and Saïdi, H. Powerful techniques for the automatic generation of invariants. In *Proc. Computer Aided Verification, 8th Int'l Conf. (CAV'96), Vol. 1102 LNCS* (New Brunswick, NJ, July 1996), R. Alur and T. Henzinger, Eds., Springer-Verlag, pp. 323–335.

[6] Bharadwaj, R., and Heitmeyer, C. Model checking complete requirements specifications using abstraction. *Journal of Automated Software Engineering* (Jan. 1999). To appear.

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

[8] Browne, M. C. *Automatic Verification of Finite State Machines Using Temporal Logic.* PhD thesis, Carnegie Mellon Univ., Pittsburgh, PA, 1989.

[9] Clarke, E. M., Grumberg, O., and Long, D. E. Model checking and abstraction. *Trans. Prog. Lang. and Systems 16*, 5 (Sept. 1994), 1512–1542.

[10] Crow, J., Owre, S., Rushby, J., Shankar, N., and Srivas, M. A tutorial introduction to PVS. Tech. rep., Computer Science Lab, SRI Int'l, Menlo Park, CA, Apr. 1995. (Presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, FL).

[11] Dill, D. L., Drexler, A. J., Hu, A. J., and Yang, C. H. Protocol verification as a hardware design aid. In *Proc. IEEE Int'l Conference on Computer Design: VLSI in Computers and Processors* (1992), pp. 522–525.

[12] Faulk, S. R., Finneran, L., Kirby, Jr., J., Shah, S., and Sutton, J. Experience applying the CoRE method to the Lockheed C-130J. In *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94)* (Gaithersburg, MD, June 1994), pp. 3–8.

[13] Graf, S., and Saïdi, H. Verifying invariants using theorem proving. In *Proc. Computer Aided Verification, 8th Int'l Conf. (CAV'96), Vol. 1102 LNCS* (New Brunswick, NJ, July 1996), R. Alur and T. Henzinger, Eds., Springer-Verlag, pp. 196–207.

[14] Harel, D., and Naamad, A. The STATEMATE semantics of statecharts. *ACM Trans. Softw. Eng. and Methodology 5*, 4 (Oct. 1996), 293–333.

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

[16] Heitmeyer, C., Kirby, J., Labaw, B., and Bharadwaj, R. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)* (Vancouver, Canada, June 1998), pp. 526–531.

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

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

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

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

[21] Holzmann, G. J. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[22] Jeffords, R. D., and Heitmeyer, C. L. Efficient automatic generation of state invariants from executable requirements specifications. Tech. rep., Naval Research Lab., Washington DC, 1998. (Draft).

[23] Kirby, Jr., J. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Tech. Rep. TR-87-07, Wang Inst. of Graduate Studies, Tyngsboro, MA, July 1987.

[24] Lamport, L. The temporal logic of actions. *ACM Trans. Programming Lang. and Systems 16*, 3 (May 1994), 872–923.

[25] Leveson, N. G., Heimdahl, M. P., Hildreth, H., and Reese, J. D. Requirements specification for process-control systems. *IEEE Trans. Softw. Eng. 20*, 9 (Sept. 1994), 684–707.

[26] Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design 6* (1995), 1–35.

[27] Manna, Z., et al. STeP: the Stanford Temporal Prover. Tech. Rep. STAN-CS-TR-94-1518, Stanford Univ., Stanford, CA, June 1994.

[28] Manna, Z., and Pnueli, A. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, New York, NY, 1995.

[29] Ostroff, J. *Temporal Logic For Real-Time Systems.* Research Studies Press LTD., Taunton, Somerset, England, 1989.

[30] Owre, S., Shankar, N., and Rushby, J. User guide for the PVS specification and verification system (Draft). Tech. rep., Computer Science Lab, SRI Int'l, Menlo Park, CA, 1993.

[31] Park, D. Y. W., Skakkebæk, J. U., and Dill, D. L. Static analysis to identify invariants in RSML specifications. In *Proc. 5th Int'l Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)* (Lyngby, Denmark, Sept. 1998), Springer.

[32] Parnas, D. L., and Madey, J. Functional documentation for computer systems. *Science of Computer Programming 25*, 1 (Oct. 1995), 41–61.

[33] Sreemani, T., and Atlee, J. M. Feasibility of model checking software requirements: A case study. In *Proc. 11th Annual Conference on Computer Assurance (COMPASS '96)* (Gaithersburg, MD, June 1996), pp. 77–88.

[34] Su, J. X., Dill, D. L., and Barrett, C. W. Automatic generation of invariants in processor verification. In *Proc. Int'l Conf. on Formal Methods in Computer-Aided Design (FMCAD'96)* (Palo Alto, CA, Nov. 1996).

Appendix: Proof of Algorithm Correctness

As background we need the formal semantics of the next state relation for Σ_A. From Υ_A we first define the explicit next modes for two given states as

$$\beta(\hat{s},\hat{s}') = \{m \mid \exists e : \Upsilon_A(\hat{s}(M), e, m) \wedge e(\hat{s},\hat{s}')\}$$

We require the next state relation to satisfy the following:

$$\rho_A(\hat{s},\hat{s}') \Rightarrow \begin{cases} \hat{s}'(M) \in \beta(\hat{s},\hat{s}') & \text{if } \beta(\hat{s},\hat{s}') \neq \emptyset \\ \hat{s}'(M) = \hat{s}(M) & \text{otherwise} \end{cases}$$

Thus $\rho_A(\hat{s},\hat{s}')$ implies that the mode in the new state is always taken from among one of the alternatives of Υ_A for $\hat{s}(M)$ having an event occurrence for $\hat{s}$ and $\hat{s}'$, otherwise there is no change in the mode value.

The semantic definitions of *KEEP* and *NEW*, equivalent to the respective syntactic forms, are also useful in the proofs. Let each positive literal be represented as $(r, true)$ and each negative literal as $(r, false)$. The semantic definition of *KEEP* is as follows:

$$KEEP(U,c) = \{\hat{s} \mid \exists \hat{s}_1 : c(\hat{s}_1) \\ \wedge \forall r : (r, \hat{s}_1(r)) \in U \Rightarrow \hat{s}(r) = \hat{s}_1(r)\},$$

where $(r, \hat{s}_1(r)) \in U$ means that either $(r, true) \in U \wedge \hat{s}_1(r)$ or $(r, false) \in U \wedge \neg\hat{s}_1(r)$. Intuitively, each $\hat{s}_1$ such that $c(\hat{s}_1)$ holds corresponds directly to one of the disjuncts (minterms) of the standard disjunctive normal form of c. For each such disjunct, if the literal appears in that disjunct and is found in U—i.e., $(r, \hat{s}_1(r)) \in U$—then we "keep" it ($\hat{s}(r) = \hat{s}_1(r)$); otherwise, we replace it by *true* (which is equivalent to $\hat{s}(r) = true \vee \hat{s}(r) = false$ in the final result). The semantic definition of *NEW* is similar:

$$NEW(q) = \{\hat{s}' \mid \exists \hat{s} : q(\hat{s},\hat{s}')\}$$

Several lemmas will be useful in the proof that our algorithm computes mode invariants. First, we prove some properties of the *KEEP* and *NEW* operators:

Lemma 1 *(1) $e(\hat{s},\hat{s}') \Rightarrow NEW(e)(\hat{s}')$, (2) $c \leq d$ implies $KEEP(U,c) \leq KEEP(U,d)$, (3) $U \subseteq V$ implies $KEEP(V,c) \leq KEEP(U,c)$, and (4) $c \leq KEEP(U,c)$.*

Proof:
The proofs of these four properties are simple applications of the semantic definitions of *NEW* and *KEEP*. For example, (1) requires that we prove $e(\hat{s},\hat{s}') \Rightarrow \exists\hat{s}_1 : e(\hat{s}_1,\hat{s}')$. To complete the proof we simply choose $\hat{s}_1 = \hat{s}$. ■

Next, we prove some properties of the invariant generation operators:

Lemma 2 *(1) $P_i(m) \leq P_{i-1}(m)$, (2) $X_i(m) \subseteq X_{i+1}(m)$, and (3) $N_{i+1}(m) \leq N_i(m)$.*

Proof: By induction on the number of passes i.

(i) $P_1(m) = KEEP(X_1(m), N_1(m)) \wedge C_1 \leq C_1 = P_0(m)$. Using this result it is easy to show that $X_1(m) \subseteq X_2(m)$ and $N_2(m) \leq N_1(m)$.

(ii) Assume the claim is true for $i = k$. Parts (2) and (3) of the induction hypothesis and parts (2) and (3) of Lemma 1 imply $P_{k+1} = KEEP(X_{k+1}(m), N_{k+1}(m)) \leq KEEP(X_{k+1}(m), N_k(m)) \leq KEEP(X_k(m), N_k(m)) = P_k(m)$. From $P_{k+1} \leq P_k$, it follows immediately that $X_{k+1}(m) \subseteq X_{k+2}(m)$ and $N_{k+2}(m) \leq N_{k+1}(m)$. ■

The next lemma says basically that if the mode does not change when there is no β transition possible, then $KEEP(X_i, c)$ remains *true*:

Lemma 3 *If we let $m = \hat{s}(M) = \hat{s}'(M)$, $\beta(\hat{s},\hat{s}') = \emptyset$, $C_2(\hat{s},\hat{s}')$, $P_{i-1}(m)(\hat{s})$, $P_{i-1}(m)(\hat{s}')$, and $KEEP(X_i,c)(\hat{s})$, then $KEEP(X_i,c)(\hat{s}')$*

Proof: By contradiction.
Assume that $\neg KEEP(X_i,c)(\hat{s}')$. Then there must be some term t of $KEEP(X_i,c)$ and some literal ℓ of t such that $\ell(\hat{s})$ yet $\neg\ell(\hat{s}')$. We must have $\ell \in X_i(m)$. By the definition of X_i, $@\mathrm{F}(\ell) \wedge P_{i-1}(m) \wedge P_{i-1}(m)' \wedge C_2 \Rightarrow EX(m)$. Thus from the hypotheses and the assumption we have $EX(m)(\hat{s},\hat{s}')$. By the definition of EX we have that there exists e and m' with $\Upsilon_A(m,e,m')$ and $e(\hat{s},\hat{s}')$. Finally this gives $m' \in \beta(\hat{s},\hat{s}')$ in contradiction to a hypothesis. ■

In our setting, for each m the formula $M = m \Rightarrow P_i(m)$ is a mode invariant if and only if for all reachable states $\hat{s}$ of Σ_A, $P_i(\hat{s}(M))(\hat{s})$. It is sufficient for our purposes to prove correctness of the generation algorithm using an analog of the Basic Rule of Manna and Pnueli [28]. We say that a mode invariant is a *basic mode invariant* if it can be proved using this rule:

> **Basic Mode Invariance Rule**: To show $M = m \Rightarrow P_i(m)$ is a mode invariant for all m it is sufficient to show (i) $\forall m : \theta_A(m) \Rightarrow P_i(m)$, and (ii) $\forall\hat{s},\hat{s}' : P_i(\hat{s}(M))(\hat{s}) \wedge \rho_A(\hat{s},\hat{s}') \Rightarrow P_i(\hat{s}'(M))(\hat{s}')$

Lemma 4 *If $M = m \Rightarrow P_{i-1}(m)$ is a basic mode invariant for all m then $M = m \Rightarrow P_i(m)$ is a basic mode invariant for each m.*

Proof: By the Basic Mode Invariance Rule.

(i) $\theta_A(m) \Rightarrow N_i(m)$ via the of the initial states case for the definition of $N_i(m)$. By Lemma 1 part (4) $\theta_A(m) \Rightarrow KEEP(X_i(m), N_i(m))$. Finally using the axiom C_1 and the definition of P_i we have $\theta_A(m) \Rightarrow P_i(m)$.

(ii) Assume that $P_i(m)(\hat{s})$ and $\rho_A(\hat{s},\hat{s}')$ where $m = \hat{s}(M)$ and $m' = \hat{s}'(M)$. By Lemma 2 part (1) $P_{i-1}(m)(\hat{s})$. The given basic mode invariance also means that $P_{i-1}(m')(\hat{s}')$ holds. There are now two cases to consider from the assumption about ρ_A:

$m' \in \beta(\hat{s},\hat{s}')$: In this case there exists e with $\Upsilon_A(m,e,m')$ and $e(\hat{s},\hat{s}')$. Lemma 1 part (1) gives $NEW(P_{i-1}(m) \wedge C_2 \wedge e)(\hat{s}')$ so $N_i(m')(\hat{s}')$, and by Lemma 1 part (4) $KEEP(X_i(m'),N_i(m'))(\hat{s}')$.

$m' = m$ and $\beta(\hat{s},\hat{s}') = \emptyset$: $KEEP(X_i(m'),N_i(m'))(\hat{s}')$ follows from Lemma 3.

Finally, applying axiom C_1 and the definition of P_i, we have $P_i(m')(\hat{s}')$. ■

Theorem 2 *$M = m \Rightarrow P_i(m)$ is a basic mode invariant for Σ_A for each m and each pass i. Furthermore, $(M = m \Rightarrow P_i(m)) \leq (M = m \Rightarrow P_{i-1}(m))$, with at least one invariant strengthened on each pass i before the fixpoint is reached.*

Proof:
The first part is an induction: (i) $M = m \Rightarrow P_0(m)$ is clearly a basic mode invariant. (ii) The induction step follows immediately from Lemma 4. The second part follows from Lemma 2 part (1). ■