Automatic Generation of State Invariants
from Requirements Specifications *


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.
**Report Documentation Page**

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

<table>
<thead>
<tr>
<th>1. REPORT DATE</th>
<th>2. REPORT TYPE</th>
<th>3. DATES COVERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>1998</td>
<td></td>
<td>00-00-1998 to 00-00-1998</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>4. TITLE AND SUBTITLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automatic Generation of State Invariants from Requirements Specifications</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5a. CONTRACT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5b. GRANT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5c. PROGRAM ELEMENT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5d. PROJECT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5e. TASK NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5f. WORK UNIT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>6. AUTHOR(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naval Research Laboratory, Code 5546, 4555 Overlook Avenue, SW, Washington, DC, 20375</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8. PERFORMING ORGANIZATION REPORT NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>10. SPONSOR/MONITOR’S ACRONYM(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11. SPONSOR/MONITOR’S REPORT NUMBER(S)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>12. DISTRIBUTION/AVAILABILITY STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approved for public release; distribution unlimited</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>13. SUPPLEMENTARY NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14. ABSTRACT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>15. SUBJECT TERMS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16. SECURITY CLASSIFICATION OF:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. REPORT</td>
</tr>
<tr>
<td>b. ABSTRACT</td>
</tr>
<tr>
<td>c. THIS PAGE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>17. LIMITATION OF ABSTRACT</th>
<th>18. NUMBER OF PAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>19a. NAME OF RESPONSIBLE PERSON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Standard Form 298 (Rev. 8-98)  
Prescribed by ANSI Std Z39-18
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 statements of the required 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 non-deterministic 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 $\Sigma$ is represented as a state machine $\Sigma = (S, S_0, E^n, T)$, where $S$ is the set of states, $S_0 \subseteq S$ is the initial state set, $E^n$ 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 $r$ 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 $\mathbb{F}(T(c) \text{ WHEN } d' \text{ denotes a conditioned event, defined as} \mathbb{F}(c)\text{ WHEN } d \equiv \neg c \land c' \land d,$

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, $\mathbb{F}(c)\text{ WHEN } d'$ 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 $\mathbb{F}(c)$ is defined by $\mathbb{F}(c) = \mathbb{F}(\neg c).$ 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 $\neg$ 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 $\Sigma$ 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 $RF = \{\text{IgnOn, Lever, EngRunning, Brake, M}\}$, where IgnOn, Lever, EngRunning, and Brake are monitored variables and $M$ is a mode class with values in the set $\{\text{Off, Inactive, Cruise, Override}\}$. The variables IgnOn, EngRunning, and Brake are boolean; the variable Lever has the enumerated type $\{\text{off, const, resume, release}\}$. In the initial states of Cruise Control, both IgnOn and EngRunning are false and $M = \text{Off}$.

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 $\text{F(IgnOn)}$ 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 $\text{F(IgnOn)}, \text{F(EngRunning)}, \text{F(Brake)}$, or $\text{T(Lever = off)}$ 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

- $M = \text{Off} \Rightarrow \neg\text{IgnOn}$
- $M = \text{Cruise} \Rightarrow \text{IgnOn} \land \text{EngRunning} \land \neg\text{Brake} \land \neg\text{Lever} \neq \text{off}$
- $M = \text{Override} \Rightarrow \text{IgnOn} \land \text{EngRunning}$
- $M = \text{Inactive} \Rightarrow \text{IgnOn}$

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 $I \equiv \text{IgnOn}, E \equiv \text{EngRunning}, B \equiv \text{Brake}, O \equiv \text{Lever=off}, C \equiv \text{Lever=const}, R \equiv \text{Lever=resume}, \text{and } L \equiv \text{Lever=release}$. 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 $N(m)$

\footnote{All four values of Lever must be considered, even though the table mentions only three of them.}
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 $i$th 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, $C \otimes \overline{\text{Lever}}$.  

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 $\neg\text{IgnOn}$, denoted as $\overline{\text{ IgnOn}}$. Next, because $M = \text{Off}$ holds in the initial state, we can also include part of the initial state predicate (namely, $\neg\text{IgnOn} \land \neg\text{EngRunning}$, denoted as $\overline{\text{I\&E}}$) in the mode entry condition. Thus, the mode entry condition is $N_1(\text{Off}) = \overline{T} \lor \overline{T} \lor \overline{T} \lor \overline{\text{I\&E}} = \overline{T}$. 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 $X_1(\text{Off})$. The only condition whose falsification causes unconditional exit from Off is $\neg\text{IgnOn}$. Hence, $X_1(\text{Off}) = \{T\}$. In the third step, we restrict the mode entry condition to the members of the unconditional exit set to obtain $P_1(\text{Off}) = \overline{T}$, and hence the mode invariant $M = \text{Off} \Rightarrow \neg\text{IgnOn}$.
<table>
<thead>
<tr>
<th>i</th>
<th>Mode m</th>
<th>( N_i(m) )</th>
<th>( X_i(m) )</th>
<th>( P_i(m) )</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Off</td>
<td>( { } )</td>
<td>( { } )</td>
<td>( )</td>
<td>ISP gives 4th DJ</td>
</tr>
<tr>
<td>Inactive</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Override</td>
<td>( B \lor O )</td>
<td>( { I, E } )</td>
<td>( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cruise</td>
<td>( CAI\langle E\rangle B\bar{O}C \lor RA\langle E\rangle B\bar{O}CAI )</td>
<td>( { I, E, B, O } )</td>
<td>( IAE\bar{B}O )</td>
<td>Apply OIA, CET</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Off</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td>Fixpoint reached?</td>
</tr>
<tr>
<td>Inactive</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Override</td>
<td>( BA\langle E\rangle \bar{O} \lor OA\langle E\rangle B\bar{C}R\bar{A}L )</td>
<td>( { I, E } )</td>
<td>( I\bar{E} )</td>
<td>Apply ( P_1(Cruise) ), OIA to 2nd DJ</td>
<td></td>
</tr>
<tr>
<td>Cruise</td>
<td>( CAI\langle B\rangle \bar{O}CAI \lor RA\langle E\rangle B\bar{O}CAI )</td>
<td>( { I, E, B, O } )</td>
<td>( IAE\bar{B}O )</td>
<td>Fixpoint reached?</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Off</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td>Fixpoint already reached?</td>
</tr>
<tr>
<td>Inactive</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Override</td>
<td>( BA\langle E\rangle \bar{O} \lor OA\langle E\rangle B\bar{C}R\bar{A}L )</td>
<td>( { I, E } )</td>
<td>( I\bar{E} )</td>
<td>Fixpoint reached?</td>
<td></td>
</tr>
<tr>
<td>Cruise</td>
<td>( CAI\langle B\rangle \bar{O}CAI \lor RA\langle E\rangle B\bar{O}CAI )</td>
<td>( { I, E, B, O } )</td>
<td>( IAE\bar{B}O )</td>
<td>Fixpoint already reached?</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Off</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td>Fixpoint reached!</td>
</tr>
<tr>
<td>Inactive</td>
<td>( { } )</td>
<td>( )</td>
<td>( )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Override</td>
<td>( BA\langle E\rangle \bar{O} \lor OA\langle E\rangle B\bar{C}R\bar{A}L )</td>
<td>( { I, E } )</td>
<td>( I\bar{E} )</td>
<td>Fixpoint reached!</td>
<td></td>
</tr>
<tr>
<td>Cruise</td>
<td>( CAI\langle B\rangle \bar{O}CAI \lor RA\langle E\rangle B\bar{O}CAI )</td>
<td>( { I, E, B, O } )</td>
<td>( IAE\bar{B}O )</td>
<td>Fixpoint reached!</td>
<td></td>
</tr>
</tbody>
</table>

**Key**

ISP: Initial State Predicate

OIA: One Input Assumption

CET: Constraint from Enumerated Type

\( N_i(m) \): Mode Entry Condition for Mode \( m \) at \( i \)th pass

\( X_i(m) \): Unconditional Exit Set for Mode \( m \) at \( i \)th pass

\( P_i(m) \): Invariant computed for Mode \( m \) at \( i \)th pass

DJ: Disjunct of \( N_i(m) \)

<table>
<thead>
<tr>
<th></th>
<th>ISP</th>
<th>OIA</th>
<th>CET</th>
<th>ISP</th>
<th>OIA</th>
<th>CET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IgnOn</td>
<td>EngRunning</td>
<td>Lever = off</td>
<td>EngRunning</td>
<td>Lever = const</td>
<td>Lever = release</td>
</tr>
</tbody>
</table>

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 \( CAI\langle E\rangle B\bar{O} \) holds. Thus, the mode entry condition is

\[ N_1(Cruise) = CAI\langle E\rangle B\bar{O} \lor [RAI\langle E\rangle B\bar{O} \lor CAI\langle E\rangle B] \]

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 \( C \Rightarrow O\bar{R} \), can be used to strengthen the mode entry condition \( N_1(Cruise) \), i.e.,

\[ N_1(Cruise) = CAI\langle E\rangle B\bar{O} \lor [RAI\langle E\rangle B\bar{O} \lor CAI\langle E\rangle B] \]

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

\[ M = Cruise \Rightarrow IgnOn \land EngRunning \land \neg Brake \land Lever \neq off. \quad (1) \]

In generating an invariant for the mode Override, the first pass of the algorithm uses row 6 of Table 1 to produce \( N_1(Override) = B \lor O \) and rows 7, 8, and 9 to produce \( X_1(Override) = \{ I, E \} \). 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.,

\[ N_2(Override) = BA\langle E\rangle \bar{O} \lor OA\langle E\rangle B\bar{C}R\bar{A}L \]

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

\[ M = Override \Rightarrow IgnOn \land EngRunning. \quad (2) \]

To generate an invariant for the mode Inactive, rows 1, 5, and 8 of Table 1 are used to compute
\[ N_1(\text{Inactive}) = I \lor \overline{E} \lor \overline{F} \text{ and rows 2 and 3 to compute } X_1(\text{Inactive}) = \{I\}. \] In step 3, we note that \( \overline{E} \), which appears as the second disjunct in \( N_1(\text{Inactive}) \), does not appear in the unconditional exit set \( \{I\} \). Hence, we replace \( \overline{E} \) with \( \text{true} \) in \( N_1(\text{Inactive}) \), thus producing the trivial invariant \( P_1(\text{Inactive}) = \text{true} \).

The second pass uses the One Input Assumption and the invariant \( (1) \) computed during the first pass for \text{Cruise} to strengthen the mode entry condition, that is, \( N_2(\text{Inactive}) = I \lor (\overline{E} \lor \overline{O} \lor \overline{F}) \lor \overline{E}. \) (The third disjunct \( \overline{E} \), the mode entry condition when the current mode is \text{Override}, cannot be strengthened because the invariant computed for \text{Override} during pass 1 is \text{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 \text{Override} during the second pass, can be used to rewrite the mode entry condition as \( N_3(\text{Inactive}) = I \lor (\overline{E} \lor \overline{O} \lor \overline{F}) \lor \overline{E}. \) Restricting the mode entry condition to the single member \( I \) of the unconditional exit set produces \( P_3(\text{Inactive}) = I \), which is equivalent to the mode invariant \( M = \text{Inactive} \Rightarrow \text{IgnOn} \).

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: \text{Off}, \text{Cruise}, \text{Override}, and \text{Inactive}. During pass 1, the mode invariant computed earlier for \text{Cruise} can be used to strengthen the mode entry condition for \text{Override} and the mode invariants for \text{Cruise} and \text{Override} to strengthen the mode entry condition for \text{Inactive}. This leads to strengthened mode invariants at the first pass, rather than later passes, with the fixpoint reached during the second pass.

### Table 3: Event Table for Overridden in Standard Format.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>False</td>
</tr>
<tr>
<td>Low, Permitted</td>
<td>@T[Pressure = High] WHEN Reset = off</td>
</tr>
<tr>
<td>Overridden</td>
<td>True</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Old Value</th>
<th>Event</th>
<th>New Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>@T[Block = on] WHEN Reset = on AND Pressure \neq High</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>@T[Pressure = High] OR @T[Pressure = High]</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

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 whose monitored variables are \text{Block}, \text{Reset}, and \text{Pressure} and whose single controlled variable is \text{Overridden}. In the initial states of the system, \text{Pressure} = \text{Low} \land \text{Overridden} = \text{false} \land \text{SafetyInjection} = \text{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 \( n^2 \) \( \Rightarrow \) possible transitions (excluding self-transitions). In the case of Table 3, the variable \text{Overridden} has only two values, so only two transitions are needed, the transition from TRUE to FALSE 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 \text{Overridden},

\[ \text{false} \land \text{Reset} = \text{on} \land \text{Pressure} \neq \text{High} \Rightarrow \text{FALSE} \]

\[ \text{true} \land \text{Reset} = \text{on} \land \text{Pressure} \neq \text{High} \Rightarrow \text{TRUE} \]

2To avoid confusion with the truth values \text{false} and \text{true}, we denote the values of \text{Overridden} as \text{FALSE} and \text{TRUE}.

3To avoid confusion with the truth values \text{false} and \text{true}, we denote the values of \text{Overridden} as \text{FALSE} and \text{TRUE}.
three atomic conditions are defined: $R \equiv \text{Reset} = \text{On}$, $B \equiv \text{Block} = \text{On}$, and $H \equiv \text{Pressure} = \text{High}$. (The negations of $B$ and $R$ have the obvious meaning; e.g., $\neg B \equiv \text{Block} = \text{Off}$.) Applying the algorithm when \textbf{Overridden} has the value FALSE computes the uninteresting $P_1(\text{FALSE}) = \text{true}$.

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

\[
\text{Overridden} = \text{TRUE} \Rightarrow \text{Pressure} \neq \text{High}.
\]

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 $\text{Overridden} = \text{TRUE} \Rightarrow \neg H$ is an invariant, we deduce from Table 4 that the event $\ominus \text{Reset} = \text{On}$ causes unconditional exit from TRUE. Hence, on the second pass, the unconditional exit set $X_2(\text{TRUE})$ is $\{H, \neg H, R\}$, which produces the strengthened invariant $P_2(\text{TRUE}) = \neg H \land \neg R$, i.e.,

\[
\text{Overridden} = \text{TRUE} \Rightarrow \text{Pressure} \neq \text{High} \land \text{Reset} = \text{Off}. \quad (3)
\]

<table>
<thead>
<tr>
<th>Mode</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>High, Permitted</td>
<td>True</td>
</tr>
<tr>
<td>TooLow</td>
<td>Overridden</td>
</tr>
<tr>
<td>Safety Injection</td>
<td>Off</td>
</tr>
</tbody>
</table>

Table 5: Condition Table for \textbf{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,

\[
\text{SafetyInjection} = \text{On} \Leftrightarrow \text{Pressure} = \text{TooLow} \land \neg \text{Overridden}, \quad (4)
\]

and its equivalent form,

\[
\text{SafetyInjection} = \text{Off} \Leftrightarrow \text{Pressure} \neq \text{TooLow} \lor \text{Overridden}.
\]

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

\[
\text{Property X: } \text{Reset} = \text{On} \land \text{Pressure} \neq \text{High} \Rightarrow \neg \text{Overridden}
\]

\[
\text{Property Y: } \text{Reset} = \text{On} \land \text{Pressure} = \text{TooLow} \Rightarrow \text{SafetyInjection} = \text{On}
\]

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.\footnote{In later work, Sreemani and Atlee [33] use a semantics for SCR equivalent to ours, adopting the One Input Assumption and our WHEN semantics.} 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, $\Theta$ is the initial state predicate, and $\rho$ is the next-state relation on pairs of states. To define the state machine $\Sigma$ corresponding to an SCR machine represented as $(S, S_0, E^{\geq}, T)$, we define (1) the initial-state predicate $\Theta$ on a state $s \in S$ such that $\Theta(s)$ is true iff $s \in S_0$ and (2) the next-state predicate $\rho$ on

<table>
<thead>
<tr>
<th>Mode</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>High, Permitted</td>
<td>True</td>
</tr>
<tr>
<td>TooLow</td>
<td>Overridden</td>
</tr>
<tr>
<td>Safety Injection</td>
<td>Off</td>
</tr>
</tbody>
</table>
pairs of states $s,s' \in S$ such that $\rho(s,s')$ is true if there exists an event $e \in E^m$, enabled in $s$, such that $T(e,s) = s'$. Thus the predicate $\rho$ 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, $\Sigma_a$ is an abstraction of $\Sigma$ if there is a map $a : S \rightarrow S_a$, $s \mapsto s_a$, 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 $\Sigma_a$. The original specification, which includes the mode machine as a component, describes the state machine $\Sigma$.

We guarantee that a mode invariant $q_A$ computed for a mode machine $\Sigma_a$ has a corresponding mode invariant $Z(q_A)$ in the overall state machine $\Sigma$ 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 $a$ be an abstraction map from $S$ to $S_a$. If condition $q_A$ is an invariant for $\Sigma_a$, then $Z(q_A)$ is an invariant for $\Sigma$ 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 $a$, we can often apply the three abstraction methods for SCR systems described in [6, 15] to the specification of the state machine $\Sigma$ to obtain the specification of the mode machine $\Sigma_a$. 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 $\Upsilon_A$ describing the mode transitions, the initial state predicate $\Theta_A$, 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 $\Upsilon_A$, $\Theta_A$, $C_1$, and $C_2$ are represented as follows:

- $\Upsilon_A$ 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 $\Upsilon_A$ omits self-transitions, i.e., transitions of the form $(m,e,m)$.

- $\Theta_A$ is the condition over $\Sigma_A$ 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 $\Theta_A$ with all appearances of the variable $M$ replaced with $m$. For example, in the Cruise Control System, $\Theta_A \overset{\text{def}}{=} M = \text{Off} \land \neg T \land \neg E$; therefore, $\Theta_A(\text{Off}) = \neg T \land \neg E$, and $\Theta_A(m) = \text{false}$ 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

$$\begin{align*}
&(C \Rightarrow \exists \neg R) \land \neg (O \Rightarrow \exists \neg R) \land \\
&(R \Rightarrow \exists \neg R) \land \neg (L \Rightarrow \exists \neg R).
\end{align*}$$

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., $\text{EngRunning} \Rightarrow \text{IgnOn}$ [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 $\text{Lever} \neq \text{Release}$ and $\text{Lever}$ changes, the only possible transition is $\text{Lever} = \text{Release}$.

We have shown that with some restrictions (easily met in practice) that the state machine $\Sigma_A$ 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-true terms, each of which is either true or the conjunction of one or more
literals \( \ell \) or \( \ell' \). (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 \( \text{true} \) 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):

\[
NEW((\neg T(B) \text{ WHEN } \overline{T_B}) \land \neg (B' \neq B) \Rightarrow R = R \land H' = H)) = \neg NEW([B \land \neg T_B \land H = H]) = B \land \neg T_B \land H.
\]

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 \( EX(\text{Inactive}) \) should be defined as

\[
EX(\text{Inactive}) = \neg F(I) \lor \neg T(C) \text{ WHEN } (I \land E \land \overline{B}).
\]

Formally, \( EX \) is defined by

\[
EX(m) = \left( \bigvee_{e : m : m' \neq m \land \neg T_A(m, e, m')} e \right).
\]

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 \( \ell \) that are not in \( U \) replaced by \( \text{true} \). For example, consider \( U = X_2(\text{Override}) \) and \( c = X_2(\text{Override}) \) from Table 2:

\[
KEEP([I, E], B \land \neg T_B \lor O \land \neg T_B \lor \neg T_B L) = I \land E \lor I \land E = I \land E.
\]

For the formal definition of \( KEEP \), see the Appendix.

The mode entry condition \( N_1(m) \) for a given mode \( m \) at the \( i \)th 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, \( N_1(m) \) is defined by

\[
N_1(m) = \theta_A(m) \land C_1 \lor \left( \bigvee_{\overline{m}, e : \neg T_A(\overline{m}, e, m)} NEW(P_{i-1}(\overline{m}) \land C_2 \land e) \right) \land C_1.
\]

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 \( N_2(\text{Override}) \) for the Cruise Control follows:

\[
N_2(\text{Override}) = NEW((P_1(\text{Cruise}) \land C_2) \land \neg (\neg T(B) \lor \neg T(O)) \land C_1) = NEW([I \land E \land \neg T_B \land \neg T_O \land \neg \neg T_O') \land C_1 = NEW([\neg T_B \land \neg T_O \land \neg T_O'] \land C_1 = [B \land \neg T_B \land \neg T_O \land \neg T_O] \land C_1 = B \land \neg T_B \land \neg T_O \land \neg T_O.
\]

The unconditional exit set \( X_i(m) \) for a given mode \( m \) at the \( i \)th pass is computed using the expression \( \neg F(\ell) \) 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

\[
X_i(m) = \{ \ell : \neg F(\ell) \land P_{i-1}(m) \land C_2 \Rightarrow EX(m) \}.
\]

To explain this definition, we first consider the simpler \( \neg F(\ell) \Rightarrow EX(m) \). This states that, for each \( \ell \in X_i(m) \), \( \neg F(\ell) \) 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., \( P_{i-1}(m) \land P_{i-1}(m') \land C_2 \). 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 \( \text{Override} = \text{TRUE} \) in the Safety Injection system. What follows is the proof that \( \text{Reset} = \text{off} \) is in the unconditional exit set computed during the second pass, i.e., \( \overline{F} \in X_2(\text{TRUE}) \):

\[
\overline{F} \in X_2(\text{TRUE}) \Rightarrow \neg F(\overline{R}) \land P_1(\text{TRUE}) \land P_1(\text{TRUE})' \land C_2 \land \text{EX(\text{TRUE})} \Rightarrow \neg T(R) \land \overline{T} \land \overline{C} \land \text{C}_2 \Rightarrow \neg T(R) \land \overline{T} \lor \neg \overline{H} \lor \neg F(H) \land \text{C}_2.
\]

Given the mode entry condition \( N_1(m) \) and the unconditional exit set \( X_2(m) \) at the \( i \)th pass, we can now compute the invariant \( P_1(m) \) at the \( i \)th pass using the \( KEEP \) operator and the constraints \( C_1 \). Formally, \( P_1(m) = KEEP(X_2(m), N_1(m)) \land C_1 \).

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 \( \text{KEEP} \) 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 \( \text{KEEP} \) 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 \( \Sigma_A \) (see the Appendix for the proof):

**Theorem 2** \( M = m \Rightarrow P_i(m) \) is a mode invariant for \( \Sigma_A \) 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 \( \ell \) is a mode invariant (Theorem 3.1 from [3]):

**Corollary 1** \( M = m \Rightarrow \ell \) is a mode invariant of mode \( m \) of \( \Sigma_A \) if (a) \( \ell \) is always true when mode \( m \) is entered, and (b) \( \text{event } \odot F(\ell) \) 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 \( \Sigma \).

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 \( \text{KEEP} \) 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].
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 $\Sigma_4$ 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


Appendix: Proof of Algorithm Correctness

As background we need the formal semantics of the next state relation for $\Sigma_A$. From $\Upsilon_A$ we first define the explicit next modes for two given states as

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

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

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

where $(r, \hat{s}_1(r)) \in U$ means that either $(r, \text{true}) \in U \land \hat{s}_1(r)$ or $(r, \text{false}) \in U \land \neg \hat{s}_1(r)$. 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 $\text{false}$ (which is equivalent to $\hat{s}(r) = \text{true} \lor \hat{s}(r) = \text{false}$ in the final result). The semantic definition of $\text{NEW}$ is similar:

$$\text{NEW}(q) = \{ \hat{s}' \mid \exists \hat{s}_1 : q(\hat{s}, \hat{s}') \}$$

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

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

**Proof:**

The proofs of these four properties are simple applications of the semantic definitions of $\text{NEW}$ and $\text{KEEP}$. For example, (1) requires that we prove $c(\hat{s}, \hat{s}') \Rightarrow \exists \hat{s}_1 : c(\hat{s}_1, \hat{s}')$. 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) $P_i(m) = \text{KEEP}(X_i(m), N_i(m)) \land C_i \leq C_{i-1} = P_{i-1}(m)$. Using this result it is easy to show that $X_i(m) \subseteq X_{i+1}(m)$ and $N_i(m) \leq N_i+1(m)$.

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

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

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

**Proof:**

By contradiction. Assume that $\neg \text{KEEP}(X_i,c)(\hat{s}')$. Then there must be some term $t$ of $\text{KEEP}(X_i,c)$ and some literal $\ell$ of $t$ such that $\ell(\hat{s}) = \neg \ell(\hat{s}')$. We must have $\ell \not\in E_i(m)$. By the definition of $X_i$, $\ell \not\in E_i(m)$ and $P_{i-1}(m)(\hat{s})$ and $\text{KEEP}(X_i,c)(\hat{s})$. Then the there must be $c$ and $c'$ with $\Upsilon_A(c, c', c')$ and $c(\hat{s}, \hat{s}')$. 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 $\Sigma_A$, $P_i(\hat{s}(M))(\hat{s})$. 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) $\forall \hat{s}, \hat{s}' : P_i(\hat{s}(M))(\hat{s}) \land P_i(\hat{s}'(M))(\hat{s}') \Rightarrow P_i(\hat{s}'(M))(\hat{s}')$.

**Lemma 4** If $M = m \Rightarrow P_i(m)$ 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) $\theta_A(m) \Rightarrow \text{KEEP}(X_i(m), N_i(m))$. Finally using the axiom $C_i$ 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 \( \rho_A \):

\[ m' \in \beta(\hat{s}, \hat{s}') : \text{In this case there exists } c \text{ with } Y_A(m, c, m') \text{ and } c(\hat{s}, \hat{s}'). \text{ Lemma 1 part (1) gives } \text{NEW}(P_{i-1}(m) \land C_2 \land c)(\hat{s}') \text{ so } N_i(m')(\hat{s}'), \text{ and by Lemma 1 part (4) } \text{KEEP}(X_i(m'), N_i(m'))(\hat{s}'). \]

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

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

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

\[ \text{Proof: } \]

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