Verifying SCR Requirements Specifications Using State Exploration*

In Proc. *First ACM SIGPLAN Workshop on Automatic Analysis of Software, Jan 1997*

Ramesh Bharadwaj and Constance Heitmeyer
Center for High Assurance Computer Systems (Code 5546)
Naval Research Laboratory
Washington, DC 20375
{ramesh,heitmeyer}@itd.nrl.navy.mil

Abstract

Researchers at the Naval Research Laboratory (NRL) have been developing a formal method, known as the SCR (Software Cost Reduction) method, to specify the requirements of software systems using tables. NRL has developed a formal state machine model defining the SCR semantics and support tools for analysis and validation. Recently, a verification capability was added to the SCR toolset. Users can now invoke the Spin model checker within the toolset to establish properties of a specification. This paper describes the results of our initial experiments to verify properties of SCR requirements specifications using Spin. After reviewing the SCR requirements method and introducing our formal requirements model, we describe how SCR specifications can be translated into an imperative programming notation. We also describe how we limit state explosion by verifying abstractions of the original requirements specification. These abstractions are derived using the formula to be verified and special attributes of SCR specifications. The paper concludes with the results of our experiments with Spin and a discussion of ongoing and future work.

1 Introduction

For a number of years, researchers at the Naval Research Laboratory (NRL) have been developing a formal method based on *tables* to specify the requirements of software systems [AF+92, Hen80]. The method, known as the Software Cost Reduction (SCR) method, was originally formulated to document the requirements of the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft. Since SCR's introduction more than a decade ago, many industrial organizations, including Lockheed, Grumman, and Ontario Hydro, have used the SCR method to specify requirements.

NRL has developed both a formal state machine model to define the SCR semantics [HJL96b, HJL96a, HLK95] and a set of software tools to support analysis and validation of SCR requirements specifications [H+95, HJL96a]. The toolset supports creation and editing of an SCR requirements specification, consistency and completeness checking, and simulation. Recently, we added a verification capability to the toolset. After using the tools to develop and refine a formal requirements specification, a specifier can now invoke the Spin model checker [Hol91] within the toolset to verify properties of the specification.

For several years, formal verification of programs has been an active area of research [Flo67, CM88, MP91a, MP91b]. Recently, model checking has emerged as a practical method for verifying finite-state system descriptions, in particular, descriptions of hardware and communications protocols [CES86, Hol91, Kur89, McM93]. An early effort to apply model checking to requirements specifications was reported in 1993 by Atlee and Gannon, who used the model checker MCB [CES86] to analyze properties of individual mode transition tables taken from SCR specifications [AG93]. More recently, Sreemani and Atlee [SA96] used the symbolic model checker SMV [McM93] to verify that the mode transition tables in the original A-7 requirements document satisfied assertions about combinations of modes. The latter experiment demonstrates that model checking can be used to analyze moderately large requirements specifications.

A major goal of our work is to generalize some aspects of the earlier work on verifying requirements specifications. While the work by Atlee et al. verified properties of mode transition tables with boolean input variables, the approach we describe may be used

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

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1997		**00-00-1997 to 00-00-1997**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Verifying SCR Requirements Specifications Using State Exploration	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**		**14**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

to verify properties of a complete SCR specification. Further, we allow variables to range over arbitrary (finite) domains such as integer subranges and enumerated values.

Another goal of our work is to describe formally the relationship between a requirements specification and the abstract models we verify. Our approach starts with a requirements specification, a black box description of all acceptable system implementations [HM83]. Because existing verification methods do not scale well for such descriptions (they are too detailed), we verify properties of more abstract models. Others have also taken this approach. For example, Atlee et al. analyze abstract models, and recently Butler has used PVS [ORS92] to analyze abstractions of SCR-like system models [But96]. However, the correspondence between these abstract models and the requirements specification is informal – the correctness of the abstraction is based on arguments that appeal to intuition. In contrast, our approach is based on principles that allow us to *derive* the abstract models, either automatically or under user guidance, from the requirements specification within a formal framework.

This paper describes the results of our initial experiments to verify properties of SCR requirements specifications using the Spin model checker. After reviewing the SCR approach to requirements specifications and introducing our formal requirements model, the paper describes how SCR specifications can be translated into an imperative programming notation. The paper also describes how we limit state explosion by verifying an abstract model of the original requirements specification. Such abstractions can be derived using the formula to be verified and the special attributes of SCR specifications. The paper concludes by presenting the results of our experiments with Spin and a discussion of ongoing and future work.

2 Background

2.1 SCR Requirements Specifications

In SCR, the required system behavior is described by REQ, the required relation between *monitored variables*, environmental quantities that the system monitors, and *controlled variables*, environmental quantities that the system controls [PM95]. To specify this relation concisely, the SCR approach uses four constructs – modes, terms, conditions, and events. A *mode class* is a variable whose values are *system modes* (or simply *modes*), while a *term* is any function of monitored variables, modes, or other terms. A *condition* is a predicate defined on one or more system entities (an *entity* is a monitored or controlled variable, mode class, or term). An *event* occurs when the value of any system entity changes. The notation "`@T(c) WHEN d`" denotes a *conditioned event*, defined as

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

where the unprimed condition c is evaluated in the "old" state, and the primed condition c' is evaluated in the "new" state. The environment may change a monitored quantity, causing an *input event*. In response, the system changes controlled quantities and updates terms and mode classes.

Old Mode	Event	New Mode
TooLow	@T(WaterPres $\geq$ Low)	Permitted
Permitted	@T(WaterPres $\geq$ Permit)	High
Permitted	@T(WaterPres $<$ Low)	TooLow
High	@T(WaterPres $<$ Permit)	Permitted

Table 1: Mode Transition Table for `Pressure`.

To introduce the SCR constructs, we consider a simplified version of a control system for safety injection [CP93]. The system monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold. The system operator may block this process by pressing a "Block" switch. The system is reset by a "Reset" switch. To specify the requirements of the control system, we use the monitored variables `WaterPres`, `Block`, and `Reset` to denote monitored quantities, and a controlled variable `SafetyInjection` to denote the controlled quantity. The specification includes a mode class `Pressure`, a term `Overridden`, and several conditions and events.

The mode class `Pressure`, an abstract model of `WaterPres`, has three modes: `TooLow`, `Permitted`, and `High`. At any given time, the system must be in one and only one of these modes. A drop in water pressure below a constant `Low` causes the system to enter mode `TooLow`; an increase in pressure above a larger constant `Permit` causes the system to enter mode `High`. Table 1 is a mode transition table which specifies the mode class `Pressure`.

The term `Overridden` is *true* if safety injection is blocked, and *false* otherwise. Table 2 is an event table which specifies the behavior of `Overridden`. The expression "`@T(Inmode)`" in a row of an event table denotes the event "system enters the corresponding mode". For instance, the entry in the first row of Table 2 specifies the event "the system enters mode `High`".

Table 3 is a condition table that specifies the controlled quantity `SafetyInjection`. The table

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

Table 2: Event Table for `Overridden`.

states that "If `Pressure` is `High` or `Permitted` *or* if `Pressure` is `TooLow` and `Overridden` is *true*, then `SafetyInjection` is `Off`; otherwise, it is `On`".

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

Table 3: Condition Table for `Safety Injection`.

2.2 Semantics of SCR Specifications

The following is an informal introduction to the SCR formal model. For a formal treatment, see [HJL96b].

The model, which describes the system being specified as a finite state machine, includes a set $RF = \{r_1, r_2, \ldots, r_n\}$ of entities and a special function TY which maps each entity to its legal values. In the model, a state s is represented as a function that maps each entity in RF to its value in s.

Our formal model describes a state machine Σ as a 4-tuple, $\Sigma = (S, s_0, E^m, T)$, where S is a set of states, $s_0 \in S$ is the initial state,[1] E^m is the set of input events, and T is the transform describing the allowed state transitions. In the initial version of our formal model, the transform T is deterministic. That is, T is a function that maps an input event and the current state to a new state. The system begins in state s_0. When an input event signals a change in a monitored quantity, the transform T specifies the value of each controlled variable, term, and mode class in the new state. The transform T is the composition of smaller functions, called *table functions*. These table functions are derived from the SCR tables that define the controlled variables, terms, and mode classes. Our formal model requires that each condition, event, and mode transition table satisfies certain properties. These properties guarantee that each table function is a total function.

[1] To simplify the examples in this paper, we assume a unique initial state. The formal model described in [HJL96b] allows the initial state to have more than a single value.

To compute an entity's value in the new state, the transform T may use the values of entities in both the old state and the new state. To describe the entities on which a given entity "directly depends" in the new state, we define *dependency relations* D_{new}, D_{old}, and D on RF$\times$RF. For entities r_i and r_j, the pair $(r_i, r_j) \in D_{new}$ iff r_j' is a parameter of the function defining r_i'; the pair $(r_i, r_j) \in D_{old}$ iff r_j is a parameter of the function defining r_i'; and $D = D_{new} \cup D_{old}$. To avoid circular definitions, we require D_{new}^+, the transitive closure of the D_{new} relation, to define a partial order.

The assumptions that the table functions are total and that the entities in RF are partially ordered guarantee that the transform T is a *function* (at most one new system state is defined) and *complete* (for each enabled input event, at least one new system state is completely defined).

To illustrate the formal model, we reconsider the control system for safety injection described above. In this system, the set of entity names RF contains the three monitored variables `Block`, `Reset`, and `WaterPres`, the mode class `Pressure`, the term `Overridden`, and the controlled variable `SafetyInjection`. The type definitions are

$$\text{TY}(\texttt{Block}) = \{\texttt{On}, \texttt{Off}\}$$
$$\text{TY}(\texttt{Reset}) = \{\texttt{On}, \texttt{Off}\}$$
$$\text{TY}(\texttt{SafetyInjection}) = \{\texttt{On}, \texttt{Off}\}$$
$$\text{TY}(\texttt{Pressure}) = \{\texttt{TooLow}, \texttt{Permitted}, \texttt{High}\}$$
$$\text{TY}(\texttt{Overridden}) = \{\mathit{true}, \mathit{false}\}$$

The new state dependency relation D_{new} for safety injection system is

$$\{(\texttt{SafetyInjection}, \texttt{Pressure}),$$
$$(\texttt{SafetyInjection}, \texttt{Overridden}),$$
$$(\texttt{Pressure}, \texttt{WaterPres}), (\texttt{Overridden}, \texttt{Pressure}),$$
$$(\texttt{Overridden}, \texttt{Block}), (\texttt{Overridden}, \texttt{Reset})\}.$$

By applying the definitions in [HJL96b] to the above tables, we derive the following table functions for the mode class `Pressure`, the term `Overridden`, and the controlled variable `SafetyInjection`:

$\texttt{Pressure}' =$

$$\begin{cases} \texttt{TooLow} & \textit{if } \texttt{Pressure} = \texttt{Permitted} \wedge \\ & \quad \texttt{WaterPres}' < \texttt{Low} \wedge \texttt{WaterPres} \not< \texttt{Low} \\ \texttt{High} & \textit{if } \texttt{Pressure} = \texttt{Permitted} \wedge \\ & \quad \texttt{WaterPres}' \geq \texttt{Permit} \wedge \texttt{WaterPres} \not\geq \texttt{Permit} \\ \texttt{Permitted} & \textit{if } (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{WaterPres}' \geq \texttt{Low} \wedge \\ & \quad \texttt{WaterPres} \not\geq \texttt{Low}) \vee (\texttt{Pressure} = \texttt{High} \wedge \\ & \quad \texttt{WaterPres}' < \texttt{Permit} \wedge \texttt{WaterPres} \not< \texttt{Permit}) \\ \texttt{Pressure} & \textit{otherwise.} \end{cases}$$

$\texttt{Overridden}' =$

$$\begin{cases} \textit{true} & \textit{if } (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Block}' = \texttt{On} \wedge \texttt{Block} = \texttt{Off} \wedge \texttt{Reset} = \texttt{Off}) \vee \\ & \quad (\texttt{Pressure} = \texttt{Permitted} \wedge \texttt{Block}' = \texttt{On} \wedge \texttt{Block} = \texttt{Off} \wedge \texttt{Reset} = \texttt{Off}) \\ \textit{false} & \textit{if } (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Reset}' = \texttt{On} \wedge \texttt{Reset} = \texttt{Off}) \vee \\ & \quad (\texttt{Pressure} = \texttt{Permitted} \wedge \texttt{Reset}' = \texttt{On} \wedge \texttt{Reset} = \texttt{Off}) \vee \\ & \quad (\texttt{Pressure}' = \texttt{High} \wedge \texttt{Pressure} \neq \texttt{High}) \vee \\ & \quad (\texttt{Pressure}' = \texttt{TooLow} \wedge \texttt{Pressure} \neq \texttt{TooLow}) \vee \\ & \quad (\texttt{Pressure}' = \texttt{Permitted} \wedge \texttt{Pressure} \neq \texttt{Permitted}) \\ \texttt{Overridden} & \textit{otherwise.} \end{cases}$$

$\texttt{SafetyInjection} =$

$$\begin{cases} \texttt{Off} & \textit{if } \texttt{Pressure} = \texttt{High} \vee \texttt{Pressure} = \texttt{Permitted} \vee (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \textit{true}) \\ \texttt{On} & \textit{if } \texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \textit{false} \end{cases}$$

The system behavior described by our model has a nondeterministic part and a deterministic part. While the transform T is deterministic, the input events, which are produced by the environment, are nondeterministic. The monitored variables involved in the input events may each be represented as simple finite state machines with an initial state, a set of possible states, and a next-state relation. For example, the monitored variables `Block` and `Reset` in the example both have `Off` as their initial state, the set {`Off`, `On`} as their possible states, and the set {(`Off`, `On`), (`On`, `Off`)} as the next-state relation. One possible model for the monitored variable `WaterPres` has an initial state of 14, possible states of $\{0, 1, 2, \ldots, 2000\}$, and a next-state relation which only allows `WaterPres` to change by 10 units from one state to the next, i.e.,

$$\{\ (x, x')\ |\ 1 \leq |x' - x| \leq 10,\ 0 \leq x \leq 2000,\ 0 \leq x' \leq 2000\}.$$

An important assumption of our model, the One Input Assumption, states that only one monitored variable changes at each state transition. Using the above models of the monitored variables, we can show that when the sample system is in its initial state, all of the following input events are enabled: @T(`Block=On`), @T(`Reset=On`), and @T(`WaterPres=`x), where $4 \leq x \leq 24$ and $x \neq 14$. The One Input Assumption allows only one of these input events to occur at the next state transition.

3 Verification

This section describes our use of model checking, based on state exploration methods, to perform verification. By verification, we mean the process of establishing logical properties of an SCR specification. We specify the properties as logical formulae. In this paper, we focus on a class of properties known as *state invariants*, which assert the truth of a predicate formula in all reachable states of a system. In the following, we assume that a given SCR specification satisfies application-independent properties – that is, the specification is type correct, the table functions derived from the specification are total functions, etc. Such properties can be established using our toolset. (For details of how these checks are carried out, see [HJL96a].)

To demonstrate the properties we would like to establish, we consider the following properties for the safety injection specification:

1. $\texttt{Reset} = \texttt{On} \wedge \texttt{Pressure} \neq \texttt{High} \Rightarrow \neg\texttt{Overridden}$
2. $\texttt{Reset} = \texttt{On} \wedge \texttt{Pressure} = \texttt{TooLow} \Rightarrow \texttt{SafetyInjection} = \texttt{On}$
3. $\texttt{Block} = \texttt{Off} \wedge \texttt{Pressure} = \texttt{TooLow} \Rightarrow \texttt{SafetyInjection} = \texttt{On}$

3.1 Establishing Invariants

To establish a formula q as an invariant of a state machine, we need to show that q holds in every reachable state of the machine. We do this by starting from the initial state and repeatedly computing the next states until a fixpoint is reached. To compute the possible new states given a current state, we need representations of the transform T and of the input events that trigger the state transitions. Recall that the system transform T (which specifies the new values of system entities) is deterministic, whereas the state machines for monitored variables are nondeterministic. The nondeterminism has two aspects: (a) more than one monitored variable may change (i.e., is enabled) and (b) a given monitored variable may change in more than one way.

To compute the next states, we associate with each entity r_i in RF, a conditional assignment of the form:

$$\begin{array}{l} \textbf{if} \\ \quad \Box \quad g_{i,1} \rightarrow r_i := v_{i,1} \\ \quad \Box \quad g_{i,2} \rightarrow r_i := v_{i,2} \\ \vdots \\ \quad \Box \quad g_{i,n_i} \rightarrow r_i := v_{i,n_i} \\ \textbf{fi} \end{array}$$

Here, $g_{i,1}, g_{i,2}, \ldots, g_{i,n_i}$ are boolean expressions (guards) and $v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}$ are expressions that are type compatible with entity r_i. We define the semantics of a conditional assignment along the lines of the enumerated assignment of UNITY [CM88] – one

assignment whose associated guard is "true" is executed at each transition. If more than one guard is "true", then any one of the associated assignments is nondeterministically chosen. If no guard is "true", the entity value is left unchanged. To represent the functions defined by SCR tables, we allow the expressions $g_{i,1}, g_{i,2}, \ldots, g_{i,n_i}$ and $v_{i,1}, v_{i,2}, \ldots, v_{i,n_i}$ to refer to both "old" and "new" values of entities, provided that the "new" references are not circular.

For the control system example, the conditional assignment for the term `Overridden` is given below. Conditional assignments for entities `Pressure` and `SafetyInjection` can be expressed in a similar fashion.

```
if
□ (Pressure=TooLow) AND @T(Block=On) AND
   (Reset=Off) -> Overridden := true
□ (Pressure=Permitted) AND @T(Block=On) AND
   (Reset=Off) -> Overridden := true
□ @T(Pressure=High)
   -> Overridden := false
□ @T((Pressure=TooLow) OR (Pressure=Permitted))
   -> Overridden := false
□ (Pressure=TooLow) AND @T(Reset=On)
   -> Overridden := false
□ (Pressure=Permitted) AND @T(Reset=On)
   -> Overridden := false
fi
```

The following is the conditional assignment for the monitored variable `Block`. Conditional assignments for the monitored variables `WaterPres` and `Reset` can be expressed in a similar fashion.

```
if
□   (Block=Off)   -> Block := On
□   (Block=On)    -> Block := Off
fi
```

Given a current state s and the conditional assignments for all monitored variables, we can determine the set of input events that are enabled in s by evaluating each guard. Each guard that evaluates to "true" along with the associated assignment determines an input event that is enabled in s. Because the One Input Assumption only allows a single input event to occur at each transition, one of the enabled input events e is selected nondeterministically.

The selected input event e and the current state s determine the new state s'. Because the transform T is a function, s' is unique. The values of the monitored variables in the new state s' are determined solely by the input event e. The values of the other entities in the new state s' (the mode classes, the terms, and the controlled variables) can be computed from the conditional assignments for these entities. The partial order determines the sequence in which the conditional assignments are evaluated. Because a total function defines the value of each mode class, term, and controlled variable, only one guard of each conditional assignment will evaluate to "true" and only one assignment can be executed per entity.

4 State Space Reduction

Model checking may be ineffective in practice because of *state explosion*. By their very nature, the number of reachable states of practical systems is usually very large in relation to their logical representation. Therefore, most fixpoint computations fail to terminate (by running out of memory) for realistic specifications.

Several techniques are proposed in the literature for limiting state explosion. The technique we use to contain state explosion is *abstraction* – instead of model checking a full SCR specification, we model check a smaller, more abstract model. To derive the abstraction, we exploit the structure of the formula and the structure inherent in all SCR specifications. Below, we state two correctness preserving reductions that derive a more abstract model from an SCR specification. We state two general theorems and a corollary which describe the conditions under which an invariant of the more abstract model is also an invariant of the full SCR specification.

In the theorems and corollary below, we use two predicates to characterize the initial state s_0 and the transform T of a state machine $\Sigma = (S, s_0, E^m, T)$. To characterize the initial state s_0, we define a predicate Θ such that $\Theta(s)$ is true iff $s = s_0$. To characterize the allowed state transitions, we define a predicate ρ on pairs of states such that $\rho(s, s')$ is true iff there exists an enabled event $e \in E^m$ such that $T(e, s) = s'$.

Given two state machines and a mapping between their states, the first theorem describes conditions which guarantee that a predicate formula which holds in a state of one machine also holds in the corresponding state of the second machine. The second theorem states conditions which guarantee that an invariant of one machine is also an invariant for a second machine. Finally, the corollary combines the two theorems to give general conditions which guarantee that an invariant of one machine is also an invariant of a second machine.

Theorem 4.1 *Consider two finite state machines, Σ and Σ_A, with entity sets RF and RF_A, type functions TY and TY_A, and state sets S and S_A between which there is a mapping $s \mapsto s_A$. Suppose (a) $RF_A \subseteq RF$, (b) for all $r \in RF_A : TY_A(r) = TY(r)$, (c) $s \mapsto s_A$ is a mapping such that for all $r \in RF_A$: $s_A(r) = s(r)$, and (d) q is any predicate formula defined over RF_A. Then, if q holds for state s_A, then q also holds for state s, that is, $q(s_A) \Rightarrow q(s)$.*

Theorem 4.2 *Consider two finite state machines, Σ and Σ_A, with state sets S and S_A between which there is a mapping $s \mapsto s_A$, initial state predicates Θ and Θ_A, and next state predicates ρ and ρ_A. Suppose (a) $s \mapsto s_A$ is a mapping such that for all $r \in RF_A$: $s_A(r) = s(r)$, (b) for all s in S: $\Theta(s) \Rightarrow \Theta_A(s_A)$, (c) for all s, s' in S: $\rho(s, s') \Rightarrow \rho_A(s_A, s'_A)$, and (d) q is any predicate formula over RF_A such that for all $s_A \in S_A$: $q(s_A) \Rightarrow q(s)$. Then, if q is an invariant of Σ_A, then q is also an invariant of Σ.*

Corollary 4.3 *If the hypotheses of Theorems 4.1 and 4.2 hold, then any predicate formula q over RF_A that is an invariant of Σ_A is also an invariant of Σ.*

Presented below are two principles we can use to derive a more abstract SCR specification from a full SCR specification. By applying these principles, we eliminate certain entities and their associated tables from the full SCR specification. Instead of model checking the full specification of the state machine Σ, we model check an abstract SCR specification of state machine Σ_A. Because the transformations we apply to derive the abstract machine Σ_A satisfy the conditions in the corollary, any invariant we prove for the abstract machine Σ_A is also an invariant of the original state machine Σ.

4.1 Reduction Principle 1: Eliminate Irrelevant Entities

This principle uses the set of entity names which occur in the formula being verified to eliminate unneeded entities and the tables that define them from the analysis. To apply this principle, we identify the set $\mathcal{O} \subseteq RF$ of entities occurring in formula q. Then, we let set $\mathcal{O}^*$ be the reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D of an SCR specification for state machine Σ. It is sound to infer the invariance of q for Σ if q is an invariant of the abstract machine Σ_A with $RF_A = \mathcal{O}^*$ and if the system transform of Σ_A, T_A, is obtained from T by deleting all associated tables or state machines for entities in set $RF - RF_A$.

This reduction principle is also complete – if q is an invariant of a finite state machine Σ, and Σ_A is an abstract machine derived from Σ by the application of this principle, then q is an invariant of Σ_A. In other words, we will always be able to establish an invariant of Σ by model checking Σ_A. Therefore, we always apply this reduction automatically before every verification.

For example, suppose we are establishing the invariance of formula (1) for the safety injection system. We identify the set of entities $\mathcal{O}$ occurring in the formula as

$$\mathcal{O} = \{\texttt{Pressure}, \texttt{Overridden}, \texttt{Reset}\}.$$

The reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D for safety injection is $\mathcal{O}^*$, which is defined by

$$\mathcal{O}^* = \{\texttt{Pressure}, \texttt{Overridden}, \texttt{Reset}, \texttt{Block}, \texttt{WaterPres}\}.$$

Applying the reduction principle, we eliminate controlled variable `SafetyInjection`, together with its table, for model checking property (1).

Applying this principle reduces the size of the state space in two ways. Recall that each state s of state machine Σ is a function that maps each entity in RF to its value in s. When model checking Σ, we store the values of all entities for each state s in Σ's state space. For the corresponding abstract machine Σ_A, we are only required to store values of entities in $\mathcal{O}^* \subseteq$ RF for each state s_A in Σ_A's state space, which reduces the memory requirement for storing each state. In addition, states of Σ that map entities in $\mathcal{O}^*$ to identical values will be represented as a single state, which reduces the number of states in the state space.

4.2 Reduction Principle 2: Monitored Variable Abstraction

This principle uses more abstract representations of monitored variables for model checking. To accomplish this, we identify a set of entities $\mathcal{O} \subset RF$ of the state machine Σ described by an SCR specification. These entities may or may not occur in formula q. Let set $\mathcal{O}^*$ be the reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D for Σ. We require (a) for each entity $r_i \in RF - \mathcal{O}^*$, and $r_j \in \mathcal{O}^* - \mathcal{O}$: $(r_i, r_j) \notin D$, and (b) entities in $\mathcal{O}^* - \mathcal{O}$ do not occur in formula q. It is sufficient to verify the invariance of q for the abstract state machine Σ_A with $RF_A = RF - (\mathcal{O}^* - \mathcal{O})$ and with the system transform T_A, which is obtained from T by deleting all associated tables or state machines of entities in $\mathcal{O}^*$. By doing this, we associate with each entity $e \in \mathcal{O}$, the "most

general" state machine[2].

This principle identifies an abstract machine whose "monitored quantities" are entities in $\mathcal{O}$. The values of these entities in a new state may depend on the values of other entities in the original specification, which we eliminate in the abstracted version. We therefore allow entities in $\mathcal{O}$ to be modified by the (new) "environment" of the abstract machine, which causes new "input events". It is easy to see that a property that holds under weaker assumptions about the abstract machine's "environment" also holds for the original specification under stronger assumptions about the (real) environment of the system.

The root cause of state explosion when model checking safety injection is monitored variable `WaterPres`. We therefore wish to eliminate this monitored variable. We observe that `WaterPres` only appears in the table for mode class `Pressure`. Therefore, we let $\mathcal{O} = \{\texttt{Pressure}\}$. The reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D for safety injection is $\mathcal{O}^* = \{\texttt{Pressure}, \texttt{WaterPres}\}$. Since `WaterPres` does not occur in any formula (1) to (3), nor in the tables for entities `Overridden` and `SafetyInjection`, we may delete `WaterPres`, and the table for entity `Pressure` when model checking formulae (1) to (3). By deleting the table for entity `Pressure`, we associate with `Pressure` a state machine with initial state `TooLow`, the set {`TooLow`, `Permitted`, `High`} as the possible states, and whose next state relation is the largest binary relation on this set.

Unlike Reduction Principle 1, this principle is not complete. This is because there are "extra" transitions for state machine Σ_A which generate states that are excluded from the mapping $s \mapsto s_A$. That is, the map $s \mapsto s_A$ is no longer onto.

5 Using the Spin Verifier

Spin [Hol91] is a model checker which uses state exploration for verifying properties. Systems are described in a language called *Promela*, and properties are expressed in linear-time temporal logic [MP91b]. Spin has been used to verify communication protocols and asynchronous hardware.

Promela, the language of Spin, is a notation loosely based on Dijkstra's "guarded commands" [Dij76]. Supported data types in *Promela* include `bool` (booleans), `byte` (short unsigned integers), and `int` (signed integers). Control statements include the assignment statement, statement `skip` (which does nothing), sequential composition of statements, the conditional statement, and the iterative statement. The language also has an `assert` statement.

Translating an SCR specification to *Promela* proceeds as follows. Because *Promela* does not allow expressions to refer to both "old" and "new" values of variables, we assign two *Promela* variables to each entity in the SCR specification. We call these the "new" and "old" variables. Further, expressions containing the event notation `@T(c)` must be translated into equivalent forms involving the "old" and the "new" variables. We translate the conditional assignment for each table into a *Promela* conditional statement, which computes the value of the "new" variable for each step. The conditional statements are executed sequentially, in a predetermined order that is consistent with the partial order induced by the new state dependency relation of the SCR specification. After all conditional assignments for table functions are executed, and new values assigned to all "new" variables, all the "old" variables are assigned their corresponding "new" values.

Further, we perform an optimization based on the fact that the system transform of an SCR specification is a function. This ensures that all conditional statements for entities other than the monitored variables are *deterministic*. Therefore, once we have selected an input event, we may compute the new state in a single step. In *Promela* we specify this by enclosing all the statements which correspond to the computation of the mode variables, the terms, and the controlled variables in a `d_step` (deterministic step) construct. This ensures that all intermediate states (i.e., the states generated after each assignment to the "new" and "old" variables) are not entered in the hash table which stores the reachable states.

To generate *Promela* code corresponding to input events, we generate a (non-deterministic) conditional statement for each monitored variable, which assigns any value (in the variable's domain) to the "new" *Promela* variable. We "build in" the One Input Assumption by embedding all assignments to monitored variables in a single (nondeterministic) conditional statement.

Assertion checking an invariant is performed by checking for its truth in the initial state, and in each generated "new" state, by embedding it in a *Promela* `assert` statement.

Appendix A presents the *Promela* code generated by the SCR* toolset (edited to enhance readability) for the safety injection example.

[2] A machine with states $TY(e)$, whose transition relation is characterized by the predicate "true".

Specification	Property	Reduction			States	Time	Memory
		d_step	RP1	RP2			
SIS	(2)				1.7 Million	56s	52 MBytes
SIS	(2)	$\surd$			459,084	13s	17 MBytes
SIS	(2)			$\surd$	632	0s	1.2 MBytes
SIS	(1)		$\surd$	$\surd$	450	0s	1.2 MBytes
SIS	(1)	$\surd$	$\surd$	$\surd$	160	0s	1.2 MBytes
Autopilot	$\mathcal{P}$				∞	–	–
Autopilot	$\mathcal{P}$	$\surd$	$\surd$	$\surd$	109,826	3s	3.6 MBytes

Table 4: Results of Verifying SCR Specifications Using Spin.

5.1 A Note on Partial Order Reduction

In contrast to conventional partial order reduction methods for combating state explosion [Val90, God90, HP94], which compute and keep track of information about redundant interleavings *during* state exploration in order to avoid the exploration of redundant interleavings, it is sufficient in our approach to evaluate the system transform using *only one* predetermined interleaving that is consistent with the partial order induced by the new state dependency relation. This is a property common to all SCR specifications, which follows from the SCR formal model. Therefore, enabling Spin's partial order reduction algorithm will almost never reduce the space requirement, and may *increase* the time requirement for verification (due to additional overhead).

5.2 Limitations of using Spin

Declaring "old" and "new" *Promela* variables for each entity of an SCR specification can potentially increase the state space by an exponential factor. Another problem we encountered was in translating counterexamples generated by Spin to the format of our toolset. This is particularly hard when we use abstractions during model checking. Generating understandable counterexamples is an important requirement for our toolset. For these reasons, we concluded that one way to seamlessly integrate model checking into the toolset may be to implement a special purpose model checker for SCR.

6 Results

In this section, we present the results of some of our verification experiments. To evaluate the translation method and reduction principles outlined in this paper, we have applied them to several "toy" examples, and to a more "industrial-strength" SCR specification.

For the safety injection specification (SIS), we were able to establish properties (1) and (2). We also showed that property (3) is *not* an invariant of the specification. One of the major problems in using model checking to evaluate abstract models is that counterexamples, which are generated in terms of the abstractions, are usually hard to interpret. We had no difficulty in interpreting counterexamples generated for abstractions of SCR specifications, because they are couched in terms of entities in the original specification. We view this as an important advantage of our reduction principles.

We recently used the SCR method to produce a black box requirements specification of a simplified mode control panel for the Boeing 737 autopilot [BH96]. The verification method presented in this paper proved to be valuable in detecting and correcting bugs in the autopilot specification. The specification is fairly large, consisting of more than a dozen tables. More important, the specification is infinite-state, since it contains three real-valued monitored variables. Hence, it would have been impossible to model check the specification had we not applied our reduction principles.

We were initially unable to model check the autopilot specification even after we applied the reduction principles, raising questions in our minds about their scalability. However, by model checking, we were able to find a violation of the property, *"The altitude engage mode will be* armed *only when the flight-path angle select mode is engaged"* (property $\mathcal{P}$ in Table 4). The generated counterexample was a sequence of roughly half a million states. Interpreting the counterexample did not turn out to be too difficult, since we were able to understand the counterexample merely by examining the input event that led to the bad state. Interestingly, the corrected specification had far fewer states and model checked without difficulty.

Table 4 presents some of our results. In Table 4, reduction d_step refers to the optimization where we

enclose all the statements corresponding to the computation of the dependent variables (the mode classes, the terms, and the controlled variables) in a *Promela* `d_step` (deterministic step) statement. Reductions RP1 and RP2 refer to the two reduction principles. The symbol '∞' in the table means that Spin ran out of memory before it could complete its evaluation of the given property.

We ran these experiments on a lightly loaded 75 MHz dual-processor SPARCstation-20 with 128 MBytes of RAM. Our tool built the conditional assignments automatically from the SCR requirements specifications. Moreover, the partial order derived by the tool determined the sequence in which the conditional assignments were executed. The reduction principles were applied manually. The abstract models produced were then analyzed automatically by the toolset using Spin.

7 Conclusions and Future Work

State exploration based model checking can be used to analyze moderately large requirements specifications. However, because most requirements specifications are necessarily too detailed, model checking is only feasible for abstractions of the specifications. In previous approaches, the correspondence between an abstraction and the original requirements specification has been informal – the correctness of the abstraction was based on intuitive arguments. In this paper, we describe a formal framework and two reduction principles for *deriving* an abstraction from an SCR requirements specification and the formula to be verified. The theorems and corollary in Section 4 guarantee that an invariant that holds for the abstraction also holds for the full SCR specification. We have applied our reduction principles to several "toy" examples and to a more "industrial strength" requirements specification. Our initial experiments with the state enumeration tool Spin strongly suggest that our principles can be used to combat state explosion in verifying specifications of practical systems.

In its current form, our principle for abstracting monitored variables is incomplete, since we model check under assumptions that are weaker than the assumptions about the environment in the original specification. We would like to explore techniques such as homomorphic reduction [Kur89], simulation mappings [Lyn94], and abstract interpretation [AD90] to either verify stronger abstractions (provided by users) or derive abstractions from the requirements specification.

In this paper, we perform a static optimization based on the partial order defined by the dependency relation of an SCR specification. In contrast to conventional partial order reduction methods, whose benefits are diminished due to additional space and time overhead during verification, our approach has the advantage of not introducing any overhead during state exploration.

We plan to develop tools that apply our reduction techniques automatically. Another goal is tighter integration of counterexample generation and interpretation with existing tools (such as the simulator) in our toolset. We also plan to use other methods for verifying SCR requirements specifications, including symbolic model checking using binary decision diagrams and mechanical theorem proving.

Acknowledgements

We thank Myla Archer and Ralph Jeffords for many helpful discussions on verifying SCR specifications and for their comments on this paper. We also thank Todd Grimm and Bruce Labaw for implementing our verification techniques into the toolset.

References

[AD90] R. Alur and D. Dill. "Automata for modeling real-time systems". In *Proc. 17th ICALP*, LNCS 736, 1990.

[AF+92] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. *Software Requirements for the A-7E Aircraft.* Technical Report NRL-9194, NRL, Wash. DC, 1992.

[AG93] J. M. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, pp 22–40, January 1993.

[BH96] R. Bharadwaj and C. L. Heitmeyer. "Applying the SCR Requirements Specification Method to Practical Systems: A Case Study". *Proceedings, Twenty-First Annual Software Engineering Workshop*, Greenbelt, MD, December 1996.

[But96] Ricky W. Butler. *An Introduction to Requirements Capture Using PVS: Specification of a Simple Autopilot.* NASA Technical Memorandum 110255. NASA Langley Research Center, Hampton VA 23681.

[CES86] E. M. Clarke, E. A. Emerson and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems* 8(2):244–263, 1986.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley, 1988.

[CP93] P.-J. Courtois and D. L. Parnas. "Documentation for safety critical software". In *Proc. 15th Int'l Conf. on Software Engg.,* Baltimore, 1993.

[Dij76] E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976

[Flo67] R. W. Floyd. "Assigning meanings to programs". *Proc. Amer. Math. Soc. Symposia in Applied Mathematics*, Volume 19, pp.19–31.

[God90] P. Godefroid. "Using partial orders to improve automatic verification methods". In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, pages 176–185. LNCS 513, 1990.

[Hen80] K. L. Heninger. "Specifying software requirements for complex systems: New techniques and their applications". *IEEE Transactions on Software Engineering* SE-6(1), Jan 1980.

[H+95] Constance Heitmeyer, et al. "SCR*: A toolset for specifying and analyzing requirements". In *Proc.,* 10th *Annual Conference on Computer Assurance*, Gaithersburg MD, June 1995.

[HJL96a] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Trans. on Software Engg. and Methodology*, 5(3)231–261, July 1996.

[HJL96b] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. *Tools for Analyzing SCR-style Requirements Specifications: A Formal Foundation.* Technical Report NRL-7499, NRL, Wash. DC, 1996. In preparation.

[HLK95] Constance Heitmeyer, Bruce Labaw, and Daniel Kiskis. "Consistency checking of SCR-style requirements specifications". In *Proc., 1995 Int'l Symposium on Requirements Engg.*, York, England, March 1995.

[HM83] C. L. Heitmeyer and J. McLean. "Abstract requirements specifications: A new approach and its application". *IEEE Transactions on Software Engineering*, SE-9(5), Sep 1983.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, 1991.

[HP94] G. J. Holzmann and D. Peled. "An improvement in formal verification". In *Proc. FORTE94*, October 1994.

[Kur89] R. P. Kurshan. *Analysis of Discrete Event Coordination.* Lecture Notes in Computer Science 430, pp. 414–453.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[MP91a] Z. Manna and A. Pnueli. "Completing the Temporal Picture". Theoretical Computer Science, 83(1):97–130, 1991.

[MP91b] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer Verlag, 1991.

[Lyn94] N. Lynch. " Simulation techniques for proving properties of real-time systems". In *REX Workshop '93*, LNCS 803 pp. 375–424, 1994.

[ORS92] Sam Owre, John Rushby, and Natarajan Shankar. "PVS: A prototype verification system". In 11th *International Conference on Automated Deduction*, LNCS-607, 1992.

[PM95] D. L. Parnas and J. Madey. "Functional documents for computer systems". *Science of Computer Programming,* 25(1), pp 41–62, Oct 1995.

[SA96] T. Sreemani and J. M. Atlee. "Feasibility of Model Checking Software Requirements". In *Proc.,* 11th *Annual Conference on Computer Assurance*, Gaithersburg MD, June 1996.

[Val90] A. Valmari. "A stubborn attack on state explosion". In *Proceedings of the 2nd International Workshop on Computer-Aided Verification*, pages 156–165. LNCS 513, 1990.

A *Promela* code for safety injection

```
/* This file contains the PROMELA/spin version of an SCRTool specification. */
/* It is created by SCRTool and automatically fed to Xspin. */
/* However, this file was left in the file sis.spin */
/* for you to use, look at, etc. */



/*****************************/
/*     numeric constants     */
/*****************************/
bool TRUE = 1;
bool FALSE = 0;
#define TooLow 0
#define Permitted 1
#define High 2
#define On 0
#define Off 1
#define Low 900
#define Permit 1000


/*********************************/
/*     variable declarations     */
/*********************************/
byte Block = Off;
byte BlockP = Off;
bool Overridden = FALSE;
bool OverriddenP = FALSE;
byte Reset = On;
byte ResetP = On;
byte SafetyInjection = On;
byte SafetyInjectionP = On;
int WaterPres = 14;
int WaterPresP = 14;
byte Pressure = TooLow;
byte PressureP = TooLow;
```

```
/***********************/
/*    init function    */
/***********************/

init {

  /*****************************/
  /*    main processing loop    */
  /*****************************/
  do
  ::

    /*********************************/
    /*     specification asserts     */
    /*********************************/
    /* (Reset = On AND Pressure = TooLow) => SafetyInjection = On */
    assert((!((Reset == On) && (Pressure == TooLow))) || (SafetyInjection == On));

    /*************************************************************************/
    /*     simulation of monitored variable changes; do one each pass     */
    /*************************************************************************/
    if
    ::if
      /* randomly select any value except the current one */
      :: (Block != On) -> BlockP = On ;
      :: (Block != Off) -> BlockP = Off ;
      fi
    ::if
      /* randomly select any value except the current one */
      :: (Reset != On) -> ResetP = On ;
      :: (Reset != Off) -> ResetP = Off ;
      fi
    ::if
      /* randomly jump to any value within the legal range of the variable */
      :: ((WaterPres + 1) <= 2000) -> WaterPresP = WaterPres + 1 ;
      :: ((WaterPres - 1) >= 0) -> WaterPresP = WaterPres - 1 ;
      :: ((WaterPres + 2) <= 2000) -> WaterPresP = WaterPres + 2 ;
      :: ((WaterPres - 2) >= 0) -> WaterPresP = WaterPres - 2 ;
      :: ((WaterPres + 3) <= 2000) -> WaterPresP = WaterPres + 3 ;
      :: ((WaterPres - 3) >= 0) -> WaterPresP = WaterPres - 3 ;
      :: ((WaterPres + 4) <= 2000) -> WaterPresP = WaterPres + 4 ;
      :: ((WaterPres - 4) >= 0) -> WaterPresP = WaterPres - 4 ;
      :: ((WaterPres + 5) <= 2000) -> WaterPresP = WaterPres + 5 ;
      :: ((WaterPres - 5) >= 0) -> WaterPresP = WaterPres - 5 ;
      :: ((WaterPres + 6) <= 2000) -> WaterPresP = WaterPres + 6 ;
      :: ((WaterPres - 6) >= 0) -> WaterPresP = WaterPres - 6 ;
      :: ((WaterPres + 7) <= 2000) -> WaterPresP = WaterPres + 7 ;
      :: ((WaterPres - 7) >= 0) -> WaterPresP = WaterPres - 7 ;
      :: ((WaterPres + 8) <= 2000) -> WaterPresP = WaterPres + 8 ;
      :: ((WaterPres - 8) >= 0) -> WaterPresP = WaterPres - 8 ;
      :: ((WaterPres + 9) <= 2000) -> WaterPresP = WaterPres + 9 ;
      :: ((WaterPres - 9) >= 0) -> WaterPresP = WaterPres - 9 ;
```

```
  :: ((WaterPres + 10) <= 2000) -> WaterPresP = WaterPres + 10 ;
  :: ((WaterPres - 10) >= 0) -> WaterPresP = WaterPres - 10 ;
  fi
fi;

/*********************************************************/
/*     executions of the functions in dependency order     */
/*********************************************************/

/* the PROMELA version of the Pressure function */
d_step{
if
/* modes: TooLow */
/* event: @T(WaterPres >= Low) */
:: ((((!(WaterPres > Low)) && ((Pressure == TooLow) &&
   (!(WaterPres == Low)))) && (WaterPresP > Low))
   || ((((!(WaterPres == Low)) && ((Pressure == TooLow) &&
   (!(WaterPres > Low)))) && (WaterPresP == Low))
       -> PressureP = Permitted;
/* modes: Permitted */
/* event: @T(WaterPres < Low) */
:: ((((!(WaterPres < Low)) && (Pressure == Permitted)) && (WaterPresP < Low))
       -> PressureP = TooLow;
/* modes: Permitted */
/* event: @T(WaterPres >= Permit) */
:: ((((!(WaterPres > Permit)) && ((Pressure == Permitted) &&
   (!(WaterPres == Permit)))) && (WaterPresP > Permit))
   || ((((!(WaterPres == Permit)) && ((Pressure == Permitted) &&
      (!(WaterPres > Permit)))) && (WaterPresP == Permit))
       -> PressureP = High;
/* modes: High */
/* event: @T(WaterPres < Permit) */
:: ((((!(WaterPres < Permit)) && (Pressure == High)) && (WaterPresP < Permit))
       -> PressureP = Permitted;
:: else skip;
fi;




/* the PROMELA version of the Overridden function */
if
/* modes: TooLow, Permitted */
/* event: @T(Block = On) WHEN Reset = Off */
:: ((((!(Block == On)) && (((Pressure == TooLow) ||
    (Pressure == Permitted)) && (Reset == Off))) && (BlockP == On))
       -> OverriddenP = TRUE;
/* modes: High */
/* event: @T(Inmode) */
:: ((!(Pressure == High)) && (PressureP == High)) -> OverriddenP = FALSE;
/* modes: TooLow, Permitted */
/* event: @T(Inmode) OR @T(Reset = On) */
:: ((!((Pressure == TooLow) || (Pressure == Permitted))) &&
```

```
        ((PressureP == TooLow) || (PressureP == Permitted)))
       || ((((!(Reset == On)) && ((Pressure == TooLow) ||
           (Pressure == Permitted))) && (ResetP == On)) -> OverriddenP = FALSE;
    :: else skip;
    fi;




    /* the PROMELA version of the SafetyInjection function */
    if
    /* modes:     High, Permitted */
    /* condition: TRUE */
    :: ((PressureP == High) || (PressureP == Permitted)) -> SafetyInjectionP = Off;
    /* modes:     TooLow */
    /* condition: Overridden */
    :: ((PressureP == TooLow) && OverriddenP) -> SafetyInjectionP = Off;
    /* modes:     TooLow */
    /* condition: Not Overridden */
    :: ((PressureP == TooLow) && (!OverriddenP)) -> SafetyInjectionP = On;
    fi;




    /*************************************************************************/
    /*     update each variable and mode class for this state change     */
    /*************************************************************************/
    Block = BlockP; Overridden = OverriddenP;
    Reset = ResetP; SafetyInjection = SafetyInjectionP;
    WaterPres = WaterPresP; Pressure = PressureP;
    }

  od /* end of main processing loop */

}
```