Automated Software Engineering, 6, 37–68 (1999)


# Model Checking Complete Requirements Specifications Using Abstraction

RAMESH BHARADWAJ ramesh@itd.nrl.navy.mil
CONSTANCE L. HEITMEYER heitmeyer@itd.nrl.navy.mil
*Center for High Assurance Computer Systems (Code 5546), Naval Research Laboratory, Washington, DC 20375*

**Abstract.** Although model checking has proven remarkably effective in detecting errors in hardware designs, its success in the analysis of software specifications has been limited. Model checking algorithms for hardware verification commonly use Binary Decision Diagrams (BDDs) to represent predicates involving the many Boolean variables commonly found in hardware descriptions. Unfortunately, BDD representations may be less effective for analyzing software specifications, which usually contain not only Booleans but variables spanning a wide range of data types. Further, software specifications typically have huge, sometimes infinite, state spaces that cannot be model checked directly using conventional symbolic methods. One promising but largely unexplored approach to model checking software specifications is to apply mathematically sound abstraction methods. Such methods extract a reduced model from the specification, thus making model checking feasible. Currently, users of model checkers routinely analyze reduced models but often generate the models in ad hoc ways. As a result, the reduced models may be incorrect.

This paper, an expanded version of (Bharadwaj and Heitmeyer, 1997), describes how one can model check a complete requirements specification expressed in the SCR (Software Cost Reduction) tabular notation. Unlike previous approaches which applied model checking to mode transition tables with Boolean variables, we use model checking to analyze properties of a complete SCR specification with variables ranging over many data types. The paper also describes two sound and, under certain conditions, complete methods for producing abstractions from requirements specifications. These abstractions are derived from the specification and the property to be analyzed. Finally, the paper describes how SCR requirements specifications can be translated into the languages of Spin, an explicit state model checker, and SMV, a symbolic model checker, and presents the results of model checking two sample SCR specifications using our abstraction methods and the two model checkers.

**Keywords:** SCR, requirements specification, verification, abstraction, model checking

## 1. Introduction

During the last decade, model checking has proven remarkably effective for detecting errors in hardware designs and protocols. Much of this success can be traced to the use of Binary Decision Diagrams (BDDs) (Bryant, 1986, Bryant, 1992), an efficient technique for symbolically representing Boolean formulae. Unfortunately, model checking has had only limited success in analyzing software specifications, largely because software specifications routinely contain not only Booleans but variables spanning a wide range of data types, including integers, reals, and enumerated types. Moreover, specifications of practical software often have enormous (in many cases infinite) state spaces, which makes the direct model checking of these specifications infeasible.

Before practical software specifications can be analyzed efficiently using a model checker, the *state explosion problem* must be addressed, i.e., the size of the state space to be analyzed must be reduced. Current users of model checkers generate such reductions routinely, but usually in ad hoc ways (Jackson, 1997): the correspondence between the reduced models and the original specifications is based on informal, intuitive arguments. One

consequence of this informal process is that the models analyzed by model checkers are often incorrect. For example, when Dill and his colleagues analyzed the errors detected by the model checker Murphi, they found that valid design errors were very rare, whereas human errors in translating the original design to the model analyzed by Murphi were frequent (Dill et al, 1992). Hence, a serious problem is that reduced models generated informally and by hand may not be true abstractions of the original design.

In contrast, our approach derives the abstract models systematically from the requirements specification and the formula to be analyzed. Users of our methods need not design the abstractions; instead, the abstractions can be derived automatically. With our approach, analyzing a specification for errors consists of three steps. First, our abstraction methods are used to produce an abstract model. Next, a model checker is executed to analyze the abstract model for the property of interest. In the third step (required when the model checker detects a violation of the property), the counterexample produced by the model checker is translated to a corresponding counterexample in the original specification. This last step is crucial because the user's understanding of the system will usually be in terms of the original specification rather than the abstract model.

Our abstraction methods are designed for specifications expressed in a tabular notation called SCR (Software Cost Reduction). For a number of years, researchers at the Naval Research Laboratory (NRL) have been developing formal methods based on the SCR notation to specify the requirements of computer systems (Heninger et al, 1978, Alspaugh et al, 1992). The SCR method, originally formulated to document the requirements of the Operational Flight Program (OFP) for the U.S. Navy's A-7 aircraft, was introduced two decades ago. Since then, many industrial organizations, including Bell Laboratories (Hester, Parnas and Utter, 1981) Grumman (Meyers and White, 1983), and Ontario Hydro (Parnas, Asmis and Madey, 1991), have used the SCR method to specify the software requirements of critical systems. Recently, a version of the SCR method called CoRE (Faulk et al, 1992) was used to document the complete requirements of Lockheed's C-130J Operational Flight Program (OFP) (Faulk et al, 1994). The OFP contains more than 230K lines of Ada code (Sutton, 1997), thus demonstrating the scalability of SCR.

We have developed a formal state machine model to define the SCR semantics (Heitmeyer, Jeffords and Labaw, 1999, Heitmeyer, Jeffords, and Labaw, 1996) and a set of formal techniques and software tools to analyze requirements specifications in the SCR notation (Heitmeyer et al, 1995, Heitmeyer, Jeffords and Labaw, 1996, Heitmeyer, Kirby and Labaw, 1997, Heitmeyer et al, 1998). The tools include a *specification editor* for creating and modifying a requirements specification, a *consistency checker* which checks the specification for well-formedness (e.g., syntax and type correctness, no missing cases, no circular definitions, and no unwanted nondeterminism), and a *simulator* for symbolically executing the specification to ensure that it captures the specifier's intent. Recently, we added a model checking capability to the toolset. Once an SCR requirements specification is developed and refined using our tools, the user can invoke the explicit state model checker Spin (Holzmann, 1997) within the toolset to analyze the specification for application properties. To make model checking feasible, the user can apply our abstraction methods to the specification prior to invoking Spin.

Recently, the practical utility of the SCR methods and tools for detecting errors in software specifications has been demonstrated in four substantial pilot projects. In one project,

researchers at NASA's IV&V Facility used the SCR consistency checker to detect several errors in the prose requirements specification of software for the International Space Station (Easterbrook and Callahan, 1997). In another project, engineers at Rockwell-Collins used the specification editor, the consistency checker, and the simulator to detect 24 errors, many of them serious, in the requirements specification of an example flight guidance system (Miller, 1998). In a third project, researchers at JPL (Jet Propulsion Laboratory) used the SCR tools to analyze specifications of two components of NASA's Deep Space-1 spacecraft (Lutz and Shaw, 1997); these components are designed to reduce the likelihood that a single fault can lead to total or partial loss of a spacecraft's functions or mission-critical data. In the fourth project, our group at NRL used the SCR tools along with the newly integrated model checker to expose several errors, including a safety violation, in a contractor-produced specification of a Weapons Control Panel of a U.S. military system (Heitmeyer, Kirby and Labaw, 1998, Heitmeyer et al, 1998).

An early application of model checking to SCR requirements specifications was reported in 1993 by Atlee and Gannon, who used the model checker MCB (Clarke, Emerson and Sistla, 1986) to analyze properties of individual mode transition tables taken from SCR specifications (Atlee and Gannon, 1993). More recently, Sreemani and Atlee (Sreemani and Atlee, 1996) used the symbolic model checker SMV (McMillan, 1993) to determine whether 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 analyze software requirements specifications of moderate size.

A major goal of our work is to generalize and extend some aspects of the earlier techniques for model checking SCR requirements specifications. While the techniques of Atlee et al. are designed to analyze properties of mode transition tables with Boolean input variables, the approach we describe can be used to analyze properties of a complete SCR specification: The properties to be analyzed can contain *any* variable in the specification, and we allow variables to range over varied domains, such as integer subranges, enumerated values, and infinite subranges of the real numbers.

This paper makes the following contributions to the model checking of software specifications:

- In Section 3, a method is described for model checking complete SCR specifications rather than individual mode transition tables. The variables in the specification can have varied types—e.g., Boolean, enumerated, integer, or real.

- In Section 4, two methods are presented for deriving abstractions from SCR specifications. These abstractions are derived automatically from the property to be analyzed and the SCR specification.

- In Section 5, methods are presented for translating an SCR specification into both *Promela*, the language of Spin, and the language of SMV.

Finally, Section 6 summarizes the results of our experiments with Spin and SMV, Section 7 discusses related work, and Section 8 describes our ongoing and future work.

## 2. Background

### *2.1. SCR Requirements Model*

An SCR requirements specification describes both the system environnment, which is nondeterministic, and the required system behavior, which is usually deterministic (Heitmeyer, Jeffords and Labaw, 1996, Heitmeyer, Jeffords and Labaw, 1999). The system environment contains *monitored quantities*, which are environmental quantities that the system monitors, and *controlled quantities*, environmental quantities that the system controls. The SCR model represents the environmental quantities as *monitored* and *controlled variables*. The environment nondeterministically produces a sequence of input events, where an *input event* signals a change in some monitored quantity. The system, which is represented in our model as a state machine, begins execution in some initial state and responds to each input event in turn by changing state and by possibly producing one or more system outputs, where a *system output* is a change in a controlled quantity. In SCR, we assume that the system behavior is synchronous (similar to Esterel's Synchrony Hypothesis (Berry, 1992)): the system completely processes one input event before processing the next input event.

In SCR, the required system behavior is described by NAT and REQ, two relations of the Parnas-Madey Four Variable Model (Parnas and Madey, 1995). NAT describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment. REQ describes the required relation between the monitored and the controlled quantities that the system must maintain. To specify REQ concisely, SCR specifications use two types of auxiliary variables, *mode classes*, whose values are called *system modes* (or simply *modes*), and *terms*. In many cases, mode classes and terms capture historical information.

In our requirements model, a system $\Sigma$ is represented as a 4-tuple, $\Sigma = (S, S_0, E^m, T)$, where $S$ is the set of states, $S_0 \subseteq S$ is the initial state set, $E^m$ is the set of input events, and $T$ is the transform describing the allowed state transitions (Heitmeyer, Jeffords and Labaw, 1999). In the initial version of our formal model, the transform $T$ is deterministic, i.e., a function that maps an input event and the current state to a new state. The transform $T$ is the composition of smaller functions, called *table functions*, which are derived from the tables in an SCR requirements specification. Our formal model requires the information in each table to satisfy certain properties. These properties guarantee that each table describes a total function.

A *variable* is any monitored or controlled variable, mode class, or term. The SCR requirements model includes a set $RF = \{r_1, r_2, \ldots, r_n\}$ containing the names of all variables in a given specification and a function TY which maps each variable to the set of its legal values. In the model, a *state* $s$ is a function that maps each variable in $RF$ to its value, a *condition* is a predicate defined on a system state, and an *event* is a predicate defined on two system states that differ in the value of at least one variable. When the value of a variable changes, we say that an event "occurs". 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 conditions $c$ and $d$ are evaluated in the "old" (or "current") state, and the primed condition $c'$ is evaluated in the "new" (or "next") state.

To compute the new state, the transform $T$ uses the values of variables in both the old state and the new state. To describe the variables on which a given variable "directly depends" in the new state, we define *dependency relations* $D_{new}$, $D_{old}$, and $\mathcal{D}$ on RF $\times$ RF. For variables $r_i$ and $r_j$, the pair $(r_i, r_j) \in D_{new}$ if $r'_j$ is a parameter of the function defining $r'_i$; the pair $(r_i, r_j) \in D_{old}$ if $r_j$ is a parameter of the function defining $r'_i$; and $\mathcal{D} = D_{new} \cup D_{old}$.[1] To avoid circular definitions, we require $D_{new}$ to define a partial order. Because they depend only on changes in the environment, the monitored variables are first in the partial order. Because they can depend on any monitored variable, term, or mode class, the controlled variables come last in the partial order. The mode classes and terms come between the monitored and controlled variables. The assumptions that the table functions are total and that the variables in RF are partially ordered guarantee that the transform $T$ is a *function* (at most one new system state is defined) and *well-defined* (for each enabled input event, at least one new system state is completely defined) (Heitmeyer, Jeffords and Labaw, 1996, Heitmeyer, Jeffords and Labaw, 1999).

### *2.2. Types and Dependency Sets*

To illustrate the SCR constructs, we consider a simplified specification of a control system for a nuclear power plant. This safety injection system injects coolant into the reactor core under certain conditions (Courtois and Parnas, 1993). Appendix A.1 contains a prose description of the behavior of this system, three tables taken from an SCR specification of the required system behavior, and the functions that can be derived from the tables using our formal model. In the example system, the set of variable 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

$$\begin{aligned}
&\text{TY(}\texttt{Block}\text{)} = \{\, \texttt{On},\ \texttt{Off}\}\\
&\text{TY(}\texttt{Reset}\text{)} = \{\, \texttt{On},\ \texttt{Off}\}\\
&\text{TY(}\texttt{SafetyInjection}\text{)} = \{\, \texttt{On},\ \texttt{Off}\}\\
&\text{TY(}\texttt{Pressure}\text{)} = \{\, \texttt{TooLow, Permitted, High}\}\\
&\text{TY(}\texttt{Overridden}\text{)} = \{\, \textit{true, false}\}\\
&\text{TY(}\texttt{WaterPres}\text{)} = \{0, 1, 2, \ldots, 2000\}
\end{aligned}$$

The new state dependency relation $D_{new}$ for the example system is

$$\begin{aligned}
\{\,&(\,\texttt{SafetyInjection},\ \texttt{Pressure}),\\
&(\,\texttt{SafetyInjection},\ \texttt{Overridden}),\\
&(\,\texttt{Pressure},\ \texttt{WaterPres}),\,(\,\texttt{Overridden},\ \texttt{Pressure}),\\
&(\,\texttt{Overridden},\ \texttt{Block}),\,(\,\texttt{Overridden},\ \texttt{Reset})\,\}.
\end{aligned}$$

The partial order defined by $D_{new}$ is

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

### *2.3. Models of the Monitored Variables*

As noted above, 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 state machines with an initial state, a set of possible states (defined by the function TY), and a next-state relation. For example, the monitored variables `Block` and `Reset` in the sample system both have {`Off`, `On`} as the set of possible states and the set {(`Off`, `On`), (`On`, `Off`)} as the next-state relation; the initial state of `Block` is `Off` and of `Reset` is `On`. One possible model of the monitored variable `WaterPres` has an initial state of 14, possible values defined by TY(`WaterPres`) $= \{0, 1, 2, \ldots, 2000\}$, and a next-state relation $\tau_{wp}$, which allows `WaterPres` to change by at most 10 units from one state to the next, i.e.,

$$\tau_{wp} = \{(x, x') : 1 \leq |x' - x| \leq 10,\ 0 \leq x \leq 2000,\ 0 \leq x' \leq 2000\}. \quad (1)$$

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=Off`), and @T(`WaterPres=`$x$), where $4 \leq x \leq 24$ and $x \neq 14$. By the One Input Assumption, exactly one of these input events can occur and thereby trigger the next state transition.

## 3. Verifying SCR Specifications

This section describes our use of model checking for verification and error detection. By verification, we mean the process of establishing logical properties of an SCR specification. We specify the properties as logical formulae. This paper focuses on a class of properties known as *invariants*. Each invariant may be either a *state* invariant, a property of every reachable state $s \in S$, or a *transition* invariant, a property of every pair of reachable states $(s, s')$, where $s, s' \in S$ and there exists an input event $e \in E^m$ enabled in $s$ such that $T(e, s) = s'$. We focus on state and transition invariants because they are the properties most commonly found in specifications of practical systems we have studied (e.g., the A-7 OFP (Alspaugh et al, 1992), Kirby's cruise control system (Kirby, 1987), and, most recently, a safety-critical component of a military system (Heitmeyer, Kirby and Labaw, 1998, Heitmeyer et al, 1998)).

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 performed, see (Heitmeyer, Jeffords and Labaw, 1996, Heitmeyer, Kirby and Labaw, 1997).

### *3.1. Sample Properties*

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. $\texttt{Block} = \texttt{Off} \wedge \texttt{Pressure} = \texttt{TooLow} \Rightarrow \texttt{SafetyInjection} = \texttt{On}$
4. $@\mathrm{T}(\texttt{Pressure} = \texttt{TooLow})$ WHEN $\texttt{Block} = \texttt{Off} \Rightarrow \texttt{SafetyInjection}' = \texttt{On}$

The first three properties are state invariants. The fourth property is a transition invariant, which states, "*If* `Pressure` *becomes* `TooLow` *in the new state and* `Block` *is* `Off` *in the old state, then* `SafetyInjection` *is* `On` *in the new state.*"

### *3.2. Fixpoint Computation*

To establish a formula $q$ as a state invariant (transition invariant) of a state machine, we need to show that $q$ holds in every reachable state (every reachable transition) of the machine. We do this by starting from the initial state and repeatedly computing the new states until a fixpoint is reached, i.e., a point at which all the new states have been generated in previous iterations. 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.

*3.2.1. Conditional Assignment* We associate with each variable $r_i$ in RF a conditional assignment of the form:

$$
\begin{array}{ll}
\texttt{if} & \\
 & \quad \square \quad g_{i,1} \rightarrow r_i := v_{i,1} \\
 & \quad \square \quad g_{i,2} \rightarrow r_i := v_{i,2} \\
\vdots & \\
 & \quad \square \quad g_{i,n_i} \rightarrow r_i := v_{i,n_i} \\
\texttt{fi} &
\end{array}
$$

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 variable $r_i$. We define the semantics of a conditional assignment along the lines of the enumerated assignment of UNITY (Chandy and Misra, 1988). To represent the functions and relations in SCR specifications, 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 variables, provided that the "new" references are not circular.

First, we consider the conditional assignments for two *dependent* variables in the control system specification: the controlled variable `SafetyInjection`, which is defined by a condition table, and the term `Overridden`, which is defined by an event table. Appendix A.1 contains the condition table for `SafetyInjection`, Table A.3, and the function derived

from Table A.3. Based on this function, the conditional assignment for SafetyInjection is defined by

```
if
     □ (Pressure=High) OR (Pressure=TooLow)
       -> SafetyInjection := Off
     □ (Pressure=TooLow) AND (Overridden = true)
       -> SafetyInjection := Off
     □ (Pressure=TooLow) AND (Overridden = false)
       -> SafetyInjection := On
fi
```

The event table for the term Overridden appears in Appendix A.1 as Table A.2. The function derived from Table A.2 also appears in Appendix A.1. Based on this function, the conditional assignment for Overridden is defined by

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

The conditional assignment for the mode class Pressure is expressed in a similar fashion.

At each transition, either one guard or no guard of the conditional assignment for a dependent variable will be true. The properties of condition tables and event tables (see Appendix A.1) guarantee this. In the conditional assignment derived from a condition table, exactly one guard will be true at each transition. In the conditional assignment derived from an event table, either one guard or no guard is true at each transition. If one guard is true, then the assignment associated with that guard is selected; if no guard is true, then the variable value is left unchanged.

Next, we consider the conditional assignments for the monitored variables. In the simple control system, the conditional assignment for the monitored variable Block is defined by

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

This conditional assignment states that if Block is Off in the current state, it is enabled to change to On in the new state, whereas if Block is On in the current state, it is enabled

to change to `Off` in the new state. Conditional assignments for the monitored variables `Reset` and `WaterPres` can be expressed in a similar fashion. Note that `Block`, like `Reset`, is enabled to change in only one way at a given transition, whereas `WaterPres` is enabled to change in many ways. Thus, for each monitored variable, one or more guards of the corresponding conditional assignment may be true at a given transition, and each assignment associated with a true guard is enabled.

*3.2.2. Computing the New State* 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. Because the One Input Assumption allows only a single input event to occur at each transition, one of the enabled input events $e$ is selected nondeterministically. Thus, in the control system example, exactly one of the three monitored variables (`Block`, `Reset`, or `WaterPres`) changes at each state transition.

The selected input event $e$ and the current state $s$ determine the new state $s'$. In the new state $s'$, the values of the monitored variables are determined solely by the input event $e$; the values of the dependent variables are computed from the conditional assignments for these variables. The partial order determines the sequence in which the conditional assignments for dependent variables are evaluated. For each dependent variable, either one guard of the corresponding conditional assignment is true and the assignment associated with that guard is executed, or no guard is true and the variable value is left unchanged.

## 4. Two Abstraction Methods

In practical software systems, the number of reachable states is usually very large in relation to their logical representation. Hence, for realistic software specifications, most fixpoint computations (see Section 3.2) may fail to terminate because they run out of memory. Several techniques have been proposed to combat state explosion in model checking. One approach proposed by Clarke and his colleagues in 1994 uses abstraction (Clarke, Grumberg and Long, 1994). Although abstraction could theoretically reduce a huge (and even infinite) state space to a much smaller state space, practical yet sound ways of deriving abstractions from software specifications have not emerged. In fact, this reliance on user ingenuity has led some to conclude that, at least in some contexts, the use of abstraction in model checking is impractical (see, e.g., (Jackson, Jha and Damon, 1994)).

Below, we describe two methods for deriving abstractions from SCR requirements specifications based on the formula to be analyzed. Both methods are practical: Neither requires ingenuity on the user's part, and each derives a smaller, more abstract model automatically. Further, each method systematizes techniques that current users of model checkers routinely apply but in ad hoc ways.

Applying our methods eliminates certain variables and their associated tables from the full SCR specification. Instead of model checking the full SCR specification of the state machine $\Sigma$ for the property $q$, we model check an abstract SCR machine $\Sigma_A$ for a corresponding property $q_A$. (The abstract property $q_A$ is syntactically identical to property $q$, but because it is defined over a projection of the domain over which $q$ is defined, we call it $q_A$.) Given property $q$ and state machine $\Sigma$, we say that $\Sigma_A$ is a *sound* abstraction of $\Sigma$ relative to $q$

if $q_A$ is an invariant of $\Sigma_A$ implies that $q$ is an invariant of $\Sigma$. Given property $q$ and state machine $\Sigma$, we say that $\Sigma_A$ is a *complete* abstraction of $\Sigma$ relative to $q$ if $q$ is an invariant of $\Sigma$ implies that $q_A$ is an invariant of $\Sigma_A$. Both our abstraction methods are sound, and, with minor restrictions on the second method, both methods are also complete. Completeness is an especially desirable property in model checking. Because most practical specifications are much too large to analyze completely, the most useful function of model checking is to detect errors. Detecting an error when the abstraction is complete means that any counterexample detected in the abstraction corresponds to a counterexample in the original specification.

To characterize the allowed state transitions of $\Sigma$ below, we define a next-state predicate $\rho$ on pairs of states such that $\rho(s, s')$ is true iff there exists an enabled event $e \in E^m$ such that $T(e, s) = s'$. The predicate $\rho$ is simply a concise and abstract way of expressing the transform $T$. The corresponding next-state predicate for $\Sigma_A$ is $\rho_A$.

### *4.1. Abstraction Method 1: Remove Irrelevant Variables*

This simple abstraction method, analogous to a technique called "program slicing" which removes irrelevant variables in analyzing programs (Weiser, 1984), uses the set of variable names which occur in the formula being analyzed to eliminate unneeded variables and the tables that define them, and the state machines in the case of monitored variables, from the analysis. To apply this method, we identify the set $\mathcal{O} \subseteq RF$ of variables occurring in formula $q$. Then, we let set $\mathcal{O}^*$ be the reflexive and transitive closure of $\mathcal{O}$ under the dependency relation $\mathcal{D}$ of an SCR specification for state machine $\Sigma$. It is sound to infer the invariance of $q$ for $\Sigma$ if $q_A$ is an invariant of the abstract machine $\Sigma_A$ with $RF_A = \mathcal{O}^*$ and if the system transform of $\Sigma_A$, $\rho_A$, is obtained from $\rho$ by deleting all associated tables (or state machines in the case of monitored variables) for variables in the set $RF - RF_A$. Because the set $\mathcal{O}^*$ contains all the variables, including the monitored variables, upon which the variables in $q$ depend, this abstraction method is also complete. We always apply this abstraction method automatically before every verification.

For example, suppose we are analyzing the invariance of property 1 in Section 3.1 for the safety injection system. We identify the set of variables $\mathcal{O}$ 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 $\mathcal{D}$ for safety injection is $\mathcal{O}^*$, which is defined by

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

Applying this abstraction method eliminates the controlled variable `SafetyInjection`, together with its table, from the specification of the state machine $\Sigma$. The reduced specification describes the abstract machine $\Sigma_A$. Then, given an SCR specification of the state machine $\Sigma$ and the property $q$, model checking the abstract machine $\Sigma_A$ for property $q_A$ is equivalent to model checking the original machine $\Sigma$ for property $q$ (in this case, property 1).

Applying this method can significantly reduce the size of the state space to be model checked. Suppose that variable $r_i$ can be eliminated by this abstraction method and that the

cardinality of $r_i$'s type set is $k_i$. Then, the size of the state space that needs to be analyzed can be reduced by as much as a factor of $k_i$. If $k_i$ is large, the amount of memory required to model check $\Sigma_A$ may be considerably smaller than the amount of memory needed to model check $\Sigma$. Further, in many cases, many variables may be eliminated by this method and hence the savings in memory can be considerable, especially if some of the variables have large type sets.

### *4.2. Abstraction Method 2: Remove Detailed Monitored Variables*

Suppose that $r$ is a monitored variable that does not appear in the formula $q$, that $\hat{r}$ depends directly only on $r$, and that $\hat{r}$ is the only variable that directly depends on $r$. We define the set of variables of the abstract machine as $RF_A = RF - \{r\}$. That is, we simply remove $r$ from the set of variables. In $\Sigma_A$, the dependence of $\hat{r}$ on $r$ is eliminated by treating $\hat{r}$ as a monitored variable. The initial state(s), the set of possible states, and the next-state relation for the new monitored variable can be computed from $\hat{r}$'s initial state, the table in $\Sigma$ defining $\hat{r}$, and the next-state relation of $r$. We can generalize this method to eliminate many input variables $r_1, r_2, \ldots, r_m$ from $RF$. This reduction can be performed if $\hat{r}$ is the only variable that depends on $r_1, r_2, \ldots, r_m$; if $\hat{r}$ depends directly only on $r_1$, $r_2$,...,$r_m$; and if none of the variables $r_1, r_2, \ldots, r_m$ appear in $q$.

To illustrate this abstraction method, we consider the specification of the safety injection system. The root cause of state explosion when model checking this specification is the monitored variable `WaterPres`. We therefore wish to eliminate `WaterPres` from the analysis. Studying the specification of the safety injection system (see Appendix A.1) reveals that `WaterPres` only appears in Table A.1, the table defining the mode class `Pressure`. Since `WaterPres` does not occur in any of the properties 1-4, nor in the tables for the variables `Overridden` and `SafetyInjection`, we may delete `WaterPres` and the table for variable `Pressure` when model checking properties 1-4. In constructing $\Sigma_A$, we define `Pressure` as a monitored variable with initial state `TooLow` and the set $\{$`TooLow`, `Permitted`, `High`$\}$ as the possible states. The next-state relation for `Pressure`, namely,

$$\{(\texttt{TooLow}, \texttt{Permitted}), (\texttt{High}, \texttt{Permitted}), (\texttt{Permitted}, \texttt{High}), (\texttt{Permitted}, \texttt{TooLow})\},$$

can be computed from the table defining `Pressure` and the next-state relation of `WaterPres`.

In this application of Abstraction Method 2, the values of the detailed variable $r$ are organized into equivalence classes by the abstract variable $\hat{r}$. In the safety injection example, we can define a function $h$ that maps the type set of `WaterPres` onto the type set of `Pressure` in such a way that the values of $h($`WaterPres`$)$ and `Pressure` are equal in every state. The mapping $h$, which is from the set $\{0, 1, 2, \ldots, 2000\}$ onto the set $\{$`TooLow`, `Permitted`, `High`$\}$, is defined by

$$h(\texttt{WaterPres}) = \begin{cases} \texttt{TooLow} & \text{if } \texttt{WaterPres} < \texttt{Low} \\ \texttt{Permitted} & \text{if } \texttt{WaterPres} \geq \texttt{Low} \wedge \texttt{WaterPres} < \texttt{Permit} \\ \texttt{High} & \text{if } \texttt{WaterPres} \geq \texttt{Permit} \end{cases}$$

(The constants `Low` and `Permit` are defined in Appendix A.1.) The function $h$ determines a partition on the values of `WaterPres` into three equivalence classes, one corresponding

to each of the three possible values of Pressure. More generally, $h$ determines a partition on the set of states: two states $s$ and $\tilde{s}$ are equivalent if $h$ maps the value of WaterPres in both $s$ and $\tilde{s}$ to the same value of Pressure and if $s$ and $\tilde{s}$ agree on all other variables.

Because $q$ is constant on every equivalence class, it is easy to see that this abstraction method is sound. Given some mild restrictions on the relationship between two states in the same equivalence class, this abstraction method is also complete. A sufficient condition for completeness is that any state $s$ in an equivalence class must be reachable in a finite number of steps from any other state $\tilde{s}$ in the equivalence class. In the classes of systems we model, this condition is usually satisfied. For example, in the safety injection system, if the abstract machine $\Sigma_A$ is in its initial state, then the variable Pressure has the value TooLow. Consider a step in the abstract machine triggered by a change in Pressure from TooLow to Permitted. Starting in the initial state, the original machine $\Sigma$, which can only change WaterPres by at most 10 units from one state to the next, will require many steps (at least 89!) to reach the Permitted range. The definition of WaterPres's next-state relation $\tau_{wp}$ (see (1) in Section 2.2) guarantees that the above condition is satisfied. That is, given any two values, $x$ and $\tilde{x}$, of WaterPres in the same equivalence class, it is possible in a finite number of steps for WaterPres to transition from $x$ to $\tilde{x}$ without leaving the equivalence class, and hence without affecting other state variables. Therefore, a transition from a state $s$ to any equivalent state $\tilde{s}$ is possible in a finite number of steps (that in fact stay inside the equivalence class of $s$ and $\tilde{s}$).

## 5. Model Checking SCR Specifications.

This section describes how SCR requirements specifications can be translated into the languages of two model checkers—the explicit state model checker Spin and the symbolic model checker SMV.

### *5.1. Using the Spin Verifier*

Spin (Holzmann, 1991, Holzmann, 1997), an explicit state model checker, uses state exploration to verify properties. Systems are described in a language called *Promela* (Holzmann, 1991) and properties are expressed either in assert statements or in linear-time temporal logic (LTL) (Manna and Pnueli, 1991). Spin has been used mostly to verify communication protocols and hardware designs. *Promela*, the language of Spin, is a notation loosely based on Dijkstra's "guarded commands" (Dijkstra, 1976). Supported data types in *Promela* include bool (Booleans), byte (short unsigned integers), and int (signed integers). Supported statements in *Promela* include the assignment statement, statement skip (which does nothing), sequential composition of statements, the conditional statement, and the iterative statement.

Translating an SCR specification to *Promela* proceeds as follows. Because *Promela* does not allow expressions containing both "old" and "new" values of variables, we create two *Promela* variables for each variable in the SCR specification and call these the "old" and "new" variables. Expressions containing the event notation @T(c) are translated into equivalent forms involving the "old" and the "new" variables. We translate the conditional assignment for each SCR table into a *Promela* conditional statement, which computes

the value of the "new" variable at each step. The conditional statements are executed sequentially, in a predetermined order 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 "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 variables other than the monitored variables are *deterministic*. Therefore, once we have (nondeterministically) selected an input event, we may compute the new state in a single step. We specify this in *Promela* by enclosing all statements which compute the values of the mode variables, the terms, and the controlled variables in a `d_step` (deterministic step) construct. This ensures that, for each input event, only one state (i.e., the new state) is entered into the hash table which stores the reachable states.

To generate *Promela* code corresponding to input events, we generate a nondeterministic conditional statement for each monitored variable, based on the conditional assignment for that variable (see Section 3.2.1). We build in the One Input Assumption by embedding all such statements in a single (nondeterministic) conditional statement. Appendix A.2 presents the *Promela* code generated by the SCR* toolset (edited to enhance readability) for the safety injection example.

To express a state invariant in Spin, we embed the invariant in a *Promela* `assert` statement. Then, Spin checks the truth of the invariant in the initial state and in each generated "new" state. To denote a transition invariant, one could express the invariant in LTL and invoke the built-in translator of Spin to construct an equivalent `never` automaton. Since an SCR variable's "new" and "old" values are explicitly assigned to two *Promela* variables, we avoid this automaton construction for transition invariants; instead, we check them directly in an `assert` statement. An advantage of our approach is that in Spin checking invariants is computationally more efficient than checking properties expressed as `never` automata.

To combat state explosion, conventional partial order reduction methods avoid the exploration of redundant interleavings by computing and keeping track of information about redundant interleavings *during* state exploration (Valmari, 1990, Godefroid, 1990, Holzmann and Peled, 1994). In our approach, which relies on Spin's `d_step` construct, it is sufficient to evaluate the new state using *only one* predetermined interleaving consistent with the partial order induced by the new state dependency relation. This approach can be applied to all deterministic SCR specifications. In model checking a deterministic SCR specification, enabling Spin's partial order reduction algorithm will never reduce the space requirement and may actually *increase* the required analysis time due to additional overhead.

### *5.2. Using the SMV Model Checker*

SMV (McMillan, 1993) is a tool for verifying properties of system descriptions expressed in a special-purpose language (also called SMV). Properties are expressed in the branching time temporal logic CTL. A system is described in SMV as a set of initial states and a transition (next-state) relation. Users may either use predicate logic or a more restricted description language to specify the transition relation. Although predicate logic provides considerable flexibility, its use can lead to inconsistency—if a formula specifying the transi-

tion relation is a logical contradiction, many properties will be trivially true. Using only the restricted description language of SMV, which has a parallel assignment syntax similar to the notation of SCR, avoids this problem. For specifications in the restricted language, SMV checks for multiple parallel assignments to a variable, circular definitions, and type errors. These checks help ensure that all specifications in the restricted syntax are consistent.

Our translation of SCR specifications into SMV, which uses the restricted language of SMV, is similar to that of Atlee et al. (Atlee and Buckley, 1996) but differs in two important respects. First, as mentioned above, we translate "complete" SCR specifications, that is, specifications which include monitored and controlled variables, mode classes, and terms, any of which may be of type Boolean, an integer subrange, or an enumeration. In contrast, Atlee et al. translate mode transition tables with Boolean input variables into SMV. Second, to check properties on two states, Atlee et al. introduce additional SMV variables for each SCR state variable whose old value is referenced in some transition invariant. In contrast, we encode transition invariants directly into CTL, using the algorithm of Jeffords (Jeffords, 1997). Because we do not duplicate these state variables in the SMV model (as we do in *Promela*), our method can potentially reduce the memory requirement for model checking by an exponential factor.

To translate an SCR specification into SMV, we express the conditional assignment corresponding to each table as an SMV `case` statement. This computes the value of the corresponding variable in the "new" state. Because the system transform $T$ of an SCR specification is deterministic, the guards of all conditional assignments corresponding to system variables (i.e., variables other than monitored variables) are mutually disjoint. For these variables, it is straightforward to translate a conditional assignment into an SMV `case` statement. Note that in SMV a primed occurrence $x'$ of a variable $x$ is denoted by `next(x)`. Thus, the conditional assignment for `Overridden` is translated into the following SMV construct:

```
next(Overridden) :=
 case
    (next(Block) = On & Block = Off &
     Pressure = TooLow & Reset = Off) |
    (next(Block) = On & Block = Off &
     Pressure = Permitted & Reset = Off) : TRUE;
    (next(Reset) = On & Reset = Off &
     Pressure = TooLow) |
    (next(Reset) = On & Reset = Off &
     Pressure = Permitted) |
    (next(Pressure) = High & !(Pressure = High)) |
    ((next(Pressure) = Permitted | next(Pressure) = TooLow) &
    !(Pressure = Permitted | Pressure = TooLow)) : FALSE;
    1: Overridden; -- This means "otherwise Overridden"
 esac;
```

Unfortunately, when more than one guard is true, the semantics of the SMV `case` statement differs from that of the enumerated assignment statement discussed in Section 3.2.1. In SMV, the first assignment whose guard is true is chosen, rather than a nondeterministic

choice of any one of the assignments whose guards are true. Therefore, when the guards of a conditional assignment are not mutually disjoint, a straightforward translation would be incorrect. To solve this problem, we model nondeterministic choice by an explicit assignment of an arbitrary element among those in a set, using the SMV syntax of set assignment to denote this operation. For example, in the safety injection system, applying Abstraction Method 2 eliminates the monitored variable `WaterPres` (see Section 4.2). The next step is to define the next-state relation of the mode class `Pressure` in SMV. This relation may be expressed using SMV's set assignment construct as follows:

```
next(Pressure) :=
 case
    Pressure = Permitted : {TooLow, High};
    Pressure = TooLow    : Permitted;
    Pressure = High      : Permitted;
 esac;
```

We generate a nondeterministic SMV assignment that corresponds to the conditional assignment of each monitored variable and a deterministic assignment for each term, mode class, and controlled variable. Unlike Spin, which executes conditional assignments sequentially, SMV performs all assignments in parallel, i.e., in "one step". Therefore, unlike in the *Promela* model, the assignments in SMV may be ordered arbitrarily.

Unlike the translation to *Promela*, where we build the One Input Assumption and other restrictions imposed by NAT into the assignments, we encode NAT restrictions in SMV as predicates in a "`TRANS`" section. For example, the encoding of the state machine for `WaterPres` has two parts—the assignment statement for `WaterPres` allows *any* value from its domain for the new state (irrespective of WaterPres in the old state), while the predicate in the `TRANS` section restricts this change to be at most 10 units. Appendix A.3 presents the SMV code generated by this method for the safety injection example.

### *5.3. Spin vs SMV*

Discussions about the relative merits of explicit state (also called concrete) model checkers, such as Spin, and "symbolic" model checkers, such as SMV, have sparked considerable controversy. Explicit model checkers compute the set of reachable states by enumeration (i.e., by "running the model"), whereas symbolic model checkers execute the model symbolically by representing the set of reachable states as a logical formula using a BDD. The state spaces of some hardware designs with a certain regularity in their structure have been shown to have very compact BDD representations. For such systems, the space requirement using BDDs has a linear, rather than an exponential, relationship with the number of state variables in the model. However, BDDs may "blow up" (i.e., have an exponential memory requirement) when the models are more irregular, which is often the case in software specifications. Explicit state model checkers sometimes do better than BDDs on descriptions of communication protocols and control systems. This is because the space requirement of explicit state model checking is proportional not to the number of *possible* states but to the number of *reachable* states, which are far fewer for such systems. Not surprisingly, there-

fore, algorithms for explicit state enumeration sometimes require less space than symbolic algorithms when model checking SCR specifications.

We note that it is possible to construct an explicit state model checker with an exponentially smaller memory requirement than the memory required by Spin in our experiments. In using Spin, we were forced to declare two *Promela* variables for each SCR variable with a potential increase in space requirements by an exponential factor. (This limitation of *Promela* has nothing to do with the explicit state enumeration algorithms of Spin.) Because the SMV language allows references to both the "old" and "new" values of a variable, our SMV models avoided this problem. However, in SMV, there is a potential for exponential BDD blowups for specifications containing integer variables (as in our analysis of the original Safety Injection specification, which is presented below) because SMV does not handle integer variables optimally (Chan et al, 1998). As Chan et al. show (Chan et al, 1998), a more optimal encoding for integer variables in SMV is possible and may produce exponential reductions in memory requirements.

Symbolic model checkers for CTL, such as SMV, provide counterexamples for transition invariants in the form of a linear trace (as opposed to a branching trace). Such counterexamples may be difficult to interpret because the property is expressed in CTL, a branching time logic. However, symbolic model checkers have a distinct advantage over state enumeration in one respect: Expressing constraints (such as environmental restrictions on monitored variables) symbolically as logical formulae is more convenient than representing them as state machines. It is natural to allow, and efficient to implement, constraints expressed as logical formulae in symbolic model checkers. SMV allows users to intermix predicate logic with the more restrictive descriptive language. However, this feature should be used with caution because it permits the specification of logical contradictions.

Another advantage of SMV over Spin is that, when a property violation is detected, SMV is guaranteed to produce the shortest possible counterexample. Spin provides an algorithm that finds short counterexamples, but the algorithm does not always find the shortest one. This is because the symbolic model checking algorithms of SMV perform a breadth-first search of the state space in contrast to the depth-first search performed by the algorithms of Spin. Explicit state model checkers that perform a breadth-first search do exist; for example, the explicit state model checker Murphi (Dill et al, 1992) implements a breadth-first search. However, algorithms used to generate counterexamples in symbolic model checking and explicit state enumeration by breadth-first search are considerably more expensive and complex than the corresponding algorithm for explicit state enumeration by depth-first search.

For many practical problems, a complete search of the state space (using either explicit state or symbolic methods) is often infeasible. In such situations, model checking remains useful for *error detection*. In our experiments, we found that explicit state methods were computationally less expensive than symbolic model checking for error detection. The next section provides details.

## 6. Experimental Results

This section presents and discusses some results of our experiments with Spin and SMV. To evaluate the abstraction methods described above, we have applied them to several small examples and to a more realistic SCR specification.

For the safety injection specification (SIS), we were able to verify properties 1 and 2. We were also able to show that properties 3 and 4 are *not* invariants 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 often hard to interpret (see, e.g., (Probst, 1996)). We had little difficulty interpreting counterexamples generated for abstractions of SCR specifications, because they are expressed in terms of variables in the original specification. Also, because our abstraction methods are sound and, under certain conditions, complete, a counterexample for a property will be generated for the abstract machine if and only if the property does not hold for the original machine. We view these as important advantages of our abstractions.

We recently applied our abstraction methods to a simplified subset of the bomb release requirements of a U.S. Navy attack aircraft (Alspaugh et al, 1992). The SCR requirements specification of this system describes conditions under which the aircraft's OFP is required to issue a bomb release pulse. This specification, called `Bombrel`, contains several seeded errors. In addition to uncovering all the seeded errors with other tools in our toolset, we also established by model checking that the original formulation of a presumed state invariant, "*The aircraft should not drop a bomb unless the pilot has pressed release enable*" (property $\mathcal{P}$ in Table 2), does not hold for the corrected SCR specification. In consultation with Kirby, the creator of the specification, we reformulated the property as a transition invariant (property $\mathcal{Q}$ in Table 1) and verified the restated property using both Spin and SMV.

We ran these experiments on a lightly loaded 167 MHz Sparc Ultra-1 with 130 MB of RAM. We used Spin Version 2.9.7 of April 18, 1997, and SMV r2.4 of December 16, 1994, in our experiments. Our tool generated the *Promela* code automatically from the SCR requirements specifications. The first abstraction method was applied automatically, while the second method was applied manually. The abstract models produced were then analyzed automatically by the toolset using Spin. Generation and analysis of the SMV model were carried out manually.

Tables 1 and 2 present some of our results. In the tables, AM1 and AM2 refer to the two abstraction methods and the symbol '$\infty$' means that the corresponding model checker ran out of memory before its evaluation of the given property was complete.

Table 1 shows that Spin does better than SMV (in terms of space and time requirements) in analyzing the complete SIS specification. However, SMV does somewhat better than Spin on the abstraction. As we expected, both Spin and SMV consume much less space and time on the abstraction than on the complete specification. For `Bombrel`, both Spin and SMV ran out of space during model checking. As in the SIS example, both Spin and SMV are able to model check an abstraction of `Bombrel`, and SMV does slightly better than Spin.

Table 2 shows that our abstraction methods dramatically reduce the time and space requirements for counterexample generation. Moreover, the generated counterexamples are significantly shorter, and therefore more easily understood. The use of abstraction did

*Table 1.* Verifying SCR Specifications with Model Checkers.

**Verifying Properties With Spin**

| Specification | Property | AM1 | AM2 | States | Time | Memory |
|---|---|---|---|---|---|---|
| SIS | 1 or 2 | | | 459,084 | 10s | 16 MB |
| SIS | 1 | √ | | 459,084 | 10s | 16 MB |
| SIS | 1 or 2 | | √ | 160 | 0s | 3.1 MB |
| SIS | 1 | √ | √ | 160 | 0s | 3.1 MB |
| Bombrel | $\mathcal{Q}$ | | | $\infty$ | – | – |
| Bombrel | $\mathcal{Q}$ | | √ | 148,354 | 2s | 6.2 MB |

**Verifying Properties With SMV**

| Specification | Property | AM1 | AM2 | BDD Nodes | Time | Memory |
|---|---|---|---|---|---|---|
| SIS | 1 or 2 | | | 44,653 | 309s | 34 MB |
| SIS | 1 | √ | | 44,648 | 308s | 34 MB |
| SIS | 1 or 2 | | √ | 314 | 0s | 0.9 MB |
| SIS | 1 | √ | √ | 251 | 0s | 0.9 MB |
| Bombrel | $\mathcal{Q}$ | | | $\infty$ | – | – |
| Bombrel | $\mathcal{Q}$ | | √ | 1,912 | 0s | 0.9 MB |

*Table 2.* Detecting Errors in SCR Specifications With Model Checking.

**Generating Counterexamples With Spin**

| Specification | Property | AM1 | AM2 | Length | Time | Memory |
|---|---|---|---|---|---|---|
| SIS | 3 | | | 6 | 0.1s | 2.5 MB |
| SIS | 3 | √ | √ | 6 | 0.1s | 3.1 MB |
| Bombrel | $\mathcal{P}$ | | | 1,383 | 315s | 18 MB |
| Bombrel | $\mathcal{P}$ | | √ | 25 | 1.1s | 5 MB |

**Generating Counterexamples With SMV**

| Specification | Property | AM1 | AM2 | Length | Time | Memory |
|---|---|---|---|---|---|---|
| SIS | 3 | | | 4 | 309s | 34 MB |
| SIS | 3 | √ | √ | 4 | 0s | 0.9 MB |
| Bombrel | $\mathcal{P}$ | | | – | 13 Hrs | ? |
| Bombrel | $\mathcal{P}$ | | √ | 7 | 0.3s | 1 MB |

not affect the performance of Spin for the SIS example but did substantially reduce the time and space required by SMV. The results for Property 3 are given in the table. For Property 4, the results are comparable. For the unabstracted `Bombrel` specification, the shortest counterexample produced by Spin had 1,383 states. SMV failed to terminate for this example and therefore did not generate a counterexample. An initial run of Spin on an abstraction of `Bombrel` produced a counterexample with 104 states. The shortest coun-

terexample produced by Spin had 25 states. After examining this counterexample, Kirby manually shortened the counterexample to 9 states. SMV, however, did better by producing a counterexample with only 7 states.

We note that, for the safety injection example, using Abstraction Method 1 to verify Property 1 has no effect on the number of states or the memory requirement. This is not surprising since Abstraction Method 1 only eliminates a single Boolean variable `SafetyInjection`; the size of the state vector (28 bytes = 24 bytes + 4 bytes overhead) remains the same and hence the space for storing each state remains unaffected. Further, because `SafetyInjection` is defined by a condition table, `SafetyInjection` is a function of the new state only. Hence, elimination of the variable `SafetyInjection` does not reduce the number of reachable states.

Although Abstraction Method 1 produced negligible reductions in both the SIS specification and the `Bombrel` specification, this method can produce substantial reductions in larger specifications. Sizable reductions are possible because large specifications are much more likely than small ones to have components that are largely independent of one another. For example, in analyzing the requirements specification of a military system for a critical safety property (Heitmeyer, Kirby and Labaw, 1998, Heitmeyer et al, 1998), we used Abstraction Method 1 to reduce the number of variables in the analysis from over 250 to 55, a reduction of almost 80%.

## 7. Related Work

Our approach to model checking SCR requirements specifications is a generalization and extension of the approach originally formulated and further developed by Atlee and her colleagues (Atlee and Gannon, 1993, Atlee and Buckley, 1996, Sreemani and Atlee, 1996). The relationship between our work and Atlee's work was described earlier. Below, we describe other work in which model checking has been applied to requirements specifications. We also compare our approach to the use of abstraction in model checking with several other approaches. While our objective is to develop mathematically sound abstraction methods that can be applied automatically to requirements specifications, the major focus of other work on abstraction has been theoretical. The most complete treatment is the very general theory of abstraction relations formulated by Loiseaux et al. (Loiseaux et al, 1995) and extended with some modifications by Dams et al. (Dams and Gerth, 1997). Our approach is a special case of (Loiseaux et al, 1995) in which the abstraction relation is a map.

### *7.1. Model Checking Requirements Specifications*

In (Chan et al, 1998), Chan et al. use SMV to analyze a fragment of the TCAS II (Traffic Alert and Collision Avoidance System) requirements specification expressed in the RSML (Requirements State Machine Language) notation (Heimdahl and Leveson, 1996). They define schemas for translating RSML constructs (such as events, input variables, environment assumptions, and the synchrony hypothesis) into suitable SMV constructs, just as we do for SCR. However, unlike our translation of SCR specifications into SMV which is *semantics-preserving*, the semantics of the SMV model generated by their translation may differ from the semantics of the original RSML specification (Chan et al, 1998, p. 511). An-

other important difference between their approach and ours is that their translation involved significant manual effort, such as modifications to SMV and the use of special-purpose macro processors. In contrast, we use both Spin and SMV "out of the box".

Another significant difference between the two approaches lies in the way integer variables and constants are handled. The problem is state explosion—since the encoding in SMV for integer variables (and operations on them) is not optimal, the BDDs blow up, even in specifications containing just one or two integer variables. To solve this problem, Chan et al. directly encode integer variables as BDD bits and implement addition and comparison at the source code level by defining parameterized macros which are preprocessed using *awk* scripts. In contrast, we effectively avoid the problem by applying our correctness preserving abstraction methods to specifications containing integer variables. Because we only model check the abstractions, the state spaces of the abstractions in our examples may be orders of magnitude smaller than the state spaces Chan et al. analyze.

### *7.2. Model Checking and Abstraction*

Our work on abstraction is related to earlier work on abstraction, largely theoretical, by Clarke et al. (Clarke, Grumberg and Long, 1994), Loiseaux et al. (Loiseaux et al, 1995), Graf and Loiseaux (Graf, 1994, Graf and Loiseaux, 1993), Dams et al. (Dams and Gerth, 1997), and Kurshan (Kurshan, 1994). Below, we describe four significant aspects of our approach to abstraction which distinguish it from other approaches.

First, we focus on properties of single states or transition state-pairs rather than properties of execution sequences. As stated above, we have found that the most common properties in software requirements specifications are state and transition invariants. Expressing these properties does not require any of the techniques useful for describing execution sequences, such as temporal logics (e.g., CTL and CTL*), the mu-calculus, or automata that accept languages with infinite words (Kurshan, 1994).

Second, the abstractions we apply use *variable restriction*, which eliminates certain variables. Variable restriction, a special case of the data abstractions introduced by Clarke et al., is equivalent to abstracting the data type of each eliminated variable to a single value. Both our abstractions and those of Clarke et al. are a special case of the more general abstraction relations described by Loiseaux et al. (We note that, in the examples provided in (Loiseaux et al, 1995), (Graf and Loiseaux, 1993), and (Graf, 1994), all of the abstraction relations are in fact maps.) Although our abstractions are a proper subset of those considered by Clarke (Clarke, Grumberg and Long, 1994), we can obtain fairly complex abstractions by performing a sequence of our simple abstractions.

Third, we focus on abstraction with respect to a single simple state invariant or transition invariant. By contrast, in (Clarke, Grumberg and Long, 1994), (Loiseaux et al, 1995), (Dams and Gerth, 1997), and (Kurshan, 1994), the focus is on abstractions that preserve an entire class of properties of execution sequences derived from some set of primitive predicates. Focusing on a single simple property offers some advantages. For one, the size of the abstract model is generally smaller. Further, because we also focus on certain specific abstraction methods, we are able to automate the choice and construction of abstractions. In the work of others, the user must typically propose the abstraction, or at least, the abstraction relation. Like us, at least one author, Graf (Graf, 1994), uses abstractions

tailored to single properties, but user ingenuity is needed to find the abstractions, even when a library of abstractions and heuristics are used to aid in the search.

Finally, building and establishing the correctness (soundness or completeness) of our abstractions is automatic and not computationally expensive. As a result, our methods do not require the processing of the state transition graph as in (Kurshan, 1994) nor the modification of a BDD description of the automaton as in (Clarke, Grumberg and Long, 1994). Rather, establishing correctness and building the abstraction are done at the specification level.

We note that our two abstraction methods are related to methods proposed by Kurshan as early as 1987 (Clarke and Kurshan, 1996). Like ours, Kurshan's methods, which he calls *automatic localization reduction* (Kurshan, 1997), remove parts of the specification irrelevant to the property of interest.

## 8. Conclusions

This paper has presented an approach based on the formal requirements model defined in (Heitmeyer, Jeffords and Labaw, 1996, Heitmeyer, Jeffords and Labaw, 1999) for model checking complete SCR requirements specifications with a variety of variable types for state and transition invariants, the two classes of properties commonly found in specifications of real-world software. The paper also presented two abstraction methods that make the analysis of SCR specifications practical and techniques for translating SCR specifications into the languages of the explicit state model checker, Spin, and the symbolic model checker, SMV. Finally, the paper has presented some experimental results which suggest that symbolic model checking does not always perform better than explicit state model checking in detecting errors in software specifications.

We have successfully applied our abstraction methods to the specification of a safety-critical component of a U.S. military system (Heitmeyer, Kirby and Labaw, 1998, Heitmeyer et al, 1998). In analyzing this specification, we applied Abstraction Method 1 and developed a third abstraction method which replaces the original type set of certain variables with a smaller type set. For the details of this new abstraction method, see (Heitmeyer et al, 1998). Our abstraction methods for SCR work well in practice primarily because SCR specifications, if written and organized in accordance with the SCR method, *already contain many useful abstractions*. Therefore, unlike other approaches where the abstractions that make verification feasible must be "reverse-engineered" from scratch, useful abstractions already exist or are easy to derive (sometimes automatically) for well-written SCR specifications.

As noted above, the translation of SCR specifications into the restricted language of SMV is being automated. The importance of automatic translation cannot be overemphasized. Hand translation of the specifications is highly error-prone; in fact, we made some subtle mistakes that were caught because the results of model checking using Spin, where the translation was automatic, were inconsistent with the results of model checking using a manual translation to SMV.

We are extending our work in model checking SCR specifications in several ways:

- We are developing additional abstraction methods.
- We are designing algorithms to implement our abstraction methods: these algorithms automatically extract the abstraction $\Sigma_A$ and the property $q_A$ from the original SCR

specification and a given property $q$. (In the new abstraction methods that we are developing, $q_A$ may not be syntactically identical to $q$.) We also are investigating the extent to which we can automatically check that the conditions for completeness described in Section 4.2 are satisfied.

- Finally, we are developing software that will automatically translate a counterexample produced by model checking the abstraction $\Sigma_A$ into a corresponding counterexample in the original specification. Currently, we perform this translation manually; whether we can produce "natural" counterexamples in a completely automatic way is an open question.

Our long-term goal is to combine the power of theorem proving technology with the ease of use of model checking technology. The major problem with current theorem proving technology, e.g., PVS (Owre et al, 1995) and ACL2 (Kaufmann and Moore, 1997), is that applying the technology requires mathematical sophistication and theorem proving skills. The major problem with model checking is state explosion. Clearly, theorem proving, in many cases, *automatic* theorem proving, can dramatically reduce the number of states that a model checker analyzes. Our abstraction methods are mathematically sound methods that can dramatically reduce the state space by eliminating information irrelevant to the property of interest and abstracting away unneeded detail. We are also exploring other automated techniques that address the state explosion problem, including the automatic generation of invariants (Jeffords and Heitmeyer, 1998) and the use of powerful decision procedures.

To date, our requirements model has provided a solid foundation for a suite of analysis tools which can detect errors automatically. These tools are designed to make the cause of detected errors understandable, thereby facilitating error correction. Such an approach should lead to the production of high quality requirements specifications, which should in turn produce systems that are more likely to perform as required and less likely to lead to accidents. Such high-quality specifications should also lead to significant reductions in software development costs.

## Acknowledgments

This work was supported by the Office of Naval Research. We gratefully acknowledge Myla Archer, who identified the conditions sufficient for completeness of the second abstraction method and who helped clarify the relationship between our abstraction methods and alternative abstraction methods. Moreover, her detailed review of earlier drafts helped us improve both the presentation and the technical content of this paper. We are also grateful for the constructive comments of Angelo Gargantini, Ralph Jeffords, Steve Sims, and the anonymous referees. Finally, we thank Todd Grimm and Bruce Labaw for automating the first abstraction method and the translation of SCR specifications into *Promela*.

## Appendix

### A.1. Specifying a Simple Control System in SCR

The system, a simplified version of a control system for safety injection (Courtois and Parnas, 1993), monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold. The system operator may override safety injection by turning a "Block" switch to "On" and may reset the system after blockage by setting a "Reset" switch to "On". 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`. (In this example, the constants `Low` and `Permit` have the values 900 and 1000.) Table A.1 is a mode transition table which describes the behavior of the mode class `Pressure`.

| 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 A.1.* Mode Transition Table for `Pressure`.

The term `Overridden` is $true$ if safety injection is blocked, and $false$ otherwise. Table A.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 rightmost entry in the first row of Table A.2 specifies the event "the system enters mode `High`". In SCR specifications, each condition 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 A.2.* Event Table for `Overridden`.

describes constraints on the new system state. Table A.3, a condition table which describes

the controlled quantity SafetyInjection, requires that in every new system state, "If Pressure is High or Permitted *or* if Pressure is TooLow and Overridden is $true$, then SafetyInjection is Off; otherwise, it is On". By applying the definitions in (Heitmeyer,

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

*Table A.3.* Condition Table for Safety Injection.

Jeffords and Labaw, 1996, Heitmeyer, Jeffords
and Labaw, 1999) to Tables A.1–A.3, we obtain the following table functions for the mode class Pressure, the term Overridden, and the controlled variable SafetyInjection:
$\mathtt{Pressure}' =$
$F_4(\mathtt{Pressure}, \mathtt{WaterPres}, \mathtt{WaterPres}'\} =$

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

$\texttt{Overridden}' =$
$F_5(\texttt{Block}, \texttt{Reset}, \texttt{Pressure}, \texttt{Overridden}, \texttt{Block}', \texttt{Reset}', \texttt{Pressure}'\} =$

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

$\texttt{SafetyInjection} =$

$$F_6(\texttt{Pressure}, \texttt{Overridden}) = \begin{cases} \texttt{Off} & \text{if} \quad \texttt{Pressure} = \texttt{High} \vee \texttt{Pressure} = \texttt{Permitted} \vee \\ & \qquad (\texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \textit{true}) \\ \texttt{On} & \text{if} \quad \texttt{Pressure} = \texttt{TooLow} \wedge \texttt{Overridden} = \textit{false} \end{cases}$$

To ensure that the tables in an SCR specification define total functions, the information in each table is required to satisfy certain properties. For example, in a condition table, two properties are required of the conditions $c_i$ presented in each row of the table: the disjunction of the $c_i$ must be true, and the pairwise conjunction of the $c_i$ must be false. In an event table, the pairwise conjunction of the events $e_i$ presented in each row of the table must be false. For a complete description of the properties that tables in SCR specifications must satisfy, see (Heitmeyer, Jeffords and Labaw, 1996).

### A.2. *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
  ::

    /***************************************/
    /*     asserts for state invariants     */
    /***************************************/
    d_step{
    /* (Reset = On AND Pressure =/= High) => NOT Overridden */
    assert((!((Reset == On) && (Pressure != High))) || !Overridden);
    /* ((Reset = On) AND (Pressure = TooLow)) => (SafetyInjection = On) */
    assert((!((Reset == On) && (Pressure == TooLow))) || (SafetyInjection == On));
    /* ((Block = Off) AND (Pressure = TooLow)) => (SafetyInjection = On) */
    assert((!((Block == Off) && (Pressure == TooLow)))||(SafetyInjectionP == 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;

    /****************************************/
    /*   asserts for transition invariants   */
    /****************************************/
    /* (@T(Pressure = TooLow) WHEN (Block = Off)) => (SafetyInjection' = On) */
    assert((!(((PressureP == TooLow) && (!(Pressure == TooLow))) &&
           (Block == Off))) || (SafetyInjectionP == On));

    /*********************************************************************/
    /*     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 */
}
```

## A.3. SMV code for safety injection

```
MODULE main
VAR
  Block : {Off, On};
  Reset : {Off, On};
  WaterPres : 0..2000;
  Pressure : {TooLow, Permitted, High};
  Overridden : boolean;
  SafetyInjection : {Off, On};
DEFINE
  Low := 900;
  Permit := 1000;
ASSIGN
  init(Block) := Off;
  init(Reset) := On;
  init(WaterPres) := 14;
  init(Overridden) := 0;
  init(SafetyInjection) := On;
  init(Pressure) := TooLow;

  next(Block) := {Off, On};
  next(Reset) := {Off, On};
  next(WaterPres) := 0..2000;
  next(Pressure)   :=
                 case
                   Pressure = Permitted &
                    next(WaterPres) < Low & !(WaterPres < Low) : TooLow;
                   Pressure = Permitted &
                    next(WaterPres) >= Permit & !(WaterPres >= Permit) : High;
                   Pressure = TooLow & next(WaterPres) >= Low &
                   !(WaterPres >= Low) | (Pressure = High &
                   next(WaterPres) < Permit & !(WaterPres < Permit)): Permitted;
                   1 : Pressure;
                 esac;
  next(Overridden) :=
                 case
                   Pressure = TooLow & next(Block) = On &
                   Block = Off & Reset = Off |
                   (Pressure = Permitted & next(Block) = On &
                    Block = Off & Reset = Off) : 1;
                   Pressure = TooLow & next(Reset) = On & Reset = Off |
                   (Pressure = Permitted & next(Reset) = On & Reset = Off) |
                   next(Pressure) = High & !(Pressure = High) |
                   (next(Pressure) = TooLow | next(Pressure) = Permitted) &
                   !(Pressure = Permitted | Pressure = TooLow) : 0;
                   1: Overridden;
                 esac;
  next(SafetyInjection) :=
                 case
                   next(Pressure) = High | next(Pressure) = Permitted |
                   (next(Pressure) = TooLow & next(Overridden)) : Off;
                   next(Pressure) = TooLow & !next(Overridden) : On;
                 esac;
```

```
TRANS
  ((next(WaterPres) - WaterPres >= 1 & next(WaterPres) - WaterPres <= 10 |
    WaterPres - next(WaterPres) >= 1 & WaterPres - next(WaterPres) <= 10) &
  next(Block) = Block & next(Reset) = Reset) |
  (next(WaterPres) = WaterPres & !(next(Block) = Block) &
   next(Reset) = Reset) |
  (next(WaterPres) = WaterPres & next(Block) = Block &
   !(next(Reset) = Reset))

SPEC
 AG((Reset = On & !(Pressure = High)) -> !Overridden)
SPEC
 AG((Reset = On & Pressure = TooLow) -> SafetyInjection = On)
SPEC
 AG((Block = Off & Pressure = TooLow) -> SafetyInjection = On)
SPEC
   AG((!(Pressure = TooLow) & Block = Off) ->
      AX(Pressure = TooLow -> SafetyInjection = On))
```

## Notes

1. The dependency set $\mathcal{D}$ does not distinguish variable dependencies in the old state from variable dependencies in the new state. For example, $(r_i, r_j) \in \mathcal{D}$ implies that the parameters of $r'_i$ include $r_j$ or $r'_j$ or both $r_j$ and $r'_j$. Thus, $\mathcal{D}$ differs from the dependency set $D_{r_i}$ defined in (Heitmeyer, Kirby and Labaw, 1997), which distinguishes old state dependencies from new state dependencies.

## References


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

Atlee, J. M. and Buckley, M. A. 1996. A logic-model semantics for SCR specifications. In *Proc. Int'l Symposium on Software Testing and Analysis*.

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

Berry, G. and Gonthier, G. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19.

Bharadwaj, R. and Heitmeyer, C. 1997. Verifying SCR requirements specifications using state exploration. In Rance Cleaveland and Daniel Jackson, editors, *Proc. First ACM SIGPLAN Workshop on the Automated Analysis of Software*, ACM, Paris, France, pages 9–23.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, 8(C-35):677–691.

Bryant, R. E. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318.

Chan, W., Anderson, R. J., Beame, P., Burns, S., Modugno, F., Notkin, D. and Reese, J. D. 1998. Model checking large software specifications. *IEEE Trans. on Softw. Eng.*, 24(7).

Chandy, K. M. and Misra, J. 1988. *Parallel Program Design – A Foundation*. Reading, MA: Addison-Wesley.

Clarke, E., Grumberg, O., and Long, D. 1994. Model checking and abstraction. In *Proc., Principles of Programming Languages (POPL)*, 1994.

Clarke, E.M., Emerson, E. and Sistla, A. 1986. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang. and Systems*, 8(2):244–263.

Clarke, E.M. and Kurshan, R.P. 1996. Computer-aided verification. *IEEE Spectrum*, pages 61–67.

Courtois, P.-J. and Parnas, D.L. 1993. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD.

Dams, D. and Gerth, R. 1997. Abstract interpretation of reactive systems. *ACM Trans. on Prog. Lang. and Systems*, pages 111–149.

Dijkstra, E.W. 1976. *A Discipline of Programming*. Prentice-Hall.

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

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

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

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

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

Graf, S. 1994. Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. In *Proc. Computer Aided Verification*.

Graf, S. and Loiseaux, C. 1993. A tool for symbolic program verification and abstraction. In *Proc. Computer Aided Verification*, pages 71–84.

Heimdahl, M.P.E. and Leveson, N. 1996. Completeness and consistency in hierarchical state-based requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377.

Heitmeyer, C., Kirby, J., Labaw, B., Archer, M. and Bharadwaj, R. 1998. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11).

Heitmeyer, C.L., Jeffords, R.D. and Labaw, B.G. 1996. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261.

Heitmeyer, C., Bull, A., Gasarch, C. and Labaw, B. 1995. SCR*: A toolset for specifying and analyzing requirements. In *Proc. 10th Annual Conf. on Computer Assurance (COMPASS '95)*, pages 109–122, Gaithersburg, MD.

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

Heitmeyer, C., Kirby, J. and Labaw, B. 1998. Applying the SCR requirements method to a weapons control panel: An experience report. In *Proc. 2nd Workshop on Formal Methods in Software Practice (FMSP'98)*.

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

Heitmeyer, C.L., Jeffords, R.D. and Labaw, B.G. 1999. Tools for analyzing SCR-style requirements specifications: A formal foundation. Technical report, Naval Research Lab., Wash., DC. In preparation.

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

Hester, S.D., Parnas, D.L. and Utter, D.F. 1981. Using documentation as a software design medium. *Bell System Tech. J.*, 60(8):1941–1977.

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

Holzmann, G.J. 1997. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295.

Holzmann, G.J. and Peled, D. 1994. An improvement in formal verification. In *Proc. FORTE94*.

Jackson, D. 1997. Model checking and requirements. Minitutorial, *Third IEEE Intern. Symposium on Requirements Engineering (RE '97)*.

Jackson, D., Jha, S., and Damon, C.A. 1994. Faster checking of software specifications using isomorphs. In *Proc., Principles of Programming Languages (POPL)*.

Jeffords, R. 1997. Translating SCR properties into LTL and CTL. Technical Memorandum 5540-293A:rdj.

Jeffords, R. and Heitmeyer, C. 1998. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*.

Kaufmann, M. and Moore, J.S. 1997. An industrial-strength theorem prover based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213.

Kirby, J. 1987. Example NRL/SCR software requirements for an automobile cruise control and monitoring system. Technical Report TR-87-07, Wang Institute of Graduate Studies.

Kurshan, R. 1997. Formal verification in a commercial setting. In *Proc., Design Automation Conference*.

Kurshan, R.P. 1994. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press.

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

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

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

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

Meyers, S. and White, S. 1983. Software requirements methodology and tool study for A6-E technology transfer. Technical report, Grumman Aerospace Corp., Bethpage, NY.

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

Owre, S., Rushby, J., Shankar, N. and von Henke, F. 1995. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125.

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

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

Probst, S.T. 1996. *Chemical Process Safety and Operability Analysis using Symbolic Model Checking*. PhD thesis, Carnegie-Mellon University, Department of Chemical Engineering, Pittsburgh, PA.

Sreemani, T. and Atlee, J.M. 1996. Feasibility of model checking software requirements. In *Proc. 11th Annual Conference on Computer Assurance (COMPASS '96)*, Gaithersburg, MD.

Sutton, J. 1997. Personal communication.

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

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357.