Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications

Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj

Abstract—Exposing inconsistencies can uncover many defects in software specifications. One approach to exposing inconsistencies analyzes two redundant specifications, one operational and the other property-based, and reports discrepancies. This paper describes a "practical" formal method, based on this approach and the SCR (Software Cost Reduction) tabular notation, that can expose inconsistencies in software requirements specifications. Because users of the method do not need advanced mathematical training or theorem proving skills, most software developers should be able to apply the method without extraordinary effort. This paper also describes an application of the method which exposed a safety violation in the contractor-produced software requirements specification of a sizable, safety-critical control system. Because the enormous state space of specifications of practical software usually renders direct analysis impractical, a common approach is to apply abstraction to the specification. To reduce the state space of the control system specification, two "pushbutton" abstraction methods were applied, one which automatically removes irrelevant variables and a second which replaces the large, possibly infinite, type sets of certain variables with smaller type sets. Analyzing the reduced specification with the model checker Spin uncovered a possible safety violation. Simulation demonstrated that the safety violation was not spurious but an actual defect in the original specification.

Index Terms—Requirements, specification, abstraction, model checking, formal methods, verification, safety analysis, simulation, consistency checking, SCR.

1 Introduction

GIVEN the high frequency of defects in software requirements specifications [64], the huge cost of correcting the defects late in development [12], [23], and the serious accidents they may cause [52], techniques for the early detection and removal of defects from software requirements specifications are crucial. Exposing inconsistencies can uncover many classes of defects in requirements specifications. One technique called *consistency checking* finds inconsistencies between different parts of a specification or between a specification and a formal model describing the information required in the specification [32]. Examples of such inconsistencies include type errors, missing cases, and circular definitions. Applying a second technique called *simulation* can detect a different class of inconsistencies—inconsistencies between a user's notion of the required system behavior and the system behavior captured by the requirements specification [34], [35].

A formal requirements method called SCR (Software Cost Reduction) is designed to expose such inconsistencies in requirements specifications. Since its introduction in 1978, [38], [39], [1], the SCR requirements method has been applied successfully to a broad range of critical systems, including avionics systems [57], [24], [58], space systems [21], telephone networks [40], and control systems for nuclear power plants [60]. To support the SCR method, we have developed a set of software tools for specifying and analyzing software requirements [32], [34], [35], [36]. In addition to a *specification editor* for creating and modifying a requirements specification and a *dependency graph browser* to display the dependencies among the variables in the specification, the toolset contains tools that expose inconsistencies. These include an automated *consistency checker*, which detects type errors, missing cases, circular definitions, and other types of application-independent errors, and a *simulator*, which allows users to symbolically execute the specification to ensure that it captures their intent.

In addition to consistency checking and simulation, one may apply a third technique to expose inconsistencies. This technique compares two different specifications of the required behavior, one operational and the other property-based, and uses a formal analysis tool, such as a model checker, to detect discrepancies between them. The *operational* (or *model-based*) specification describes how the system operates, while the *property-based* specification describes the required system properties [59]. An operational specification may represent the system as a state machine, whereas a property-based specification usually expresses properties as formulas in some logic. To detect inconsistencies between an operational SCR specification and a property-based specification, we recently integrated the explicit state model checker Spin [41], [42] into the SCR toolset [10], [9], [11].

• C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj are with the Center for High Assurance Computer Systems, Code 5546, U.S. Naval Research Laboratory, 4555 Overlook Ave., SW, Washington, DC 20375. E-mail: {heitmeyer, kirby, labaw, archer, ramesh}@itd.nrl.navy.mil.

Manuscript received 15 Sept. 1997; revised 13 Mar. 1998.
Recommended for acceptance by C. Ghezzi and N. Nuseibeh.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 107209.

1.1 Dual-Language Approach

The use of two redundant specifications to represent the same required behavior is called the *dual language approach* [59]. Among the operational/property-based language pairs that have been proposed are Modechart and Real-Time Logic [45], Timed Transition Models and Real Time Temporal Logic [59], and Petri nets coupled with the TRIO logic [54]. In the SCR requirements method, the operational specification is expressed in tables and the system properties as formulas in first-order logic. Examples of the dual language approach in SCR specifications include the A-7 requirements document [38], [1], which, in addition to the tabular operational specification, contains properties of the system modes, and Kirby's cruise control specification [47], which contains both a tabular specification of the required system operations and a list of required system properties.

The dual language approach is useful because each specification style has advantages: operational specifications are less likely to omit required behavior and are often executable, whereas property-based specifications are concise, abstract, and minimize implementation bias. Another advantage of the dual language approach is that detecting inconsistencies between two different specifications of the same behavior is an effective technique for debugging both the statements of the required properties and the operational specification. For example, when Dill and his colleagues used model checking to analyze a hardware design for several properties of interest, they detected errors both in the design *and* in the stated properties [20].

1.2 A *Practical* Formal Method

This paper shows how a "practical" formal method [33], based on SCR and the dual language approach, can expose inconsistencies in specifications of safety-critical software. Although the SCR method has a formal basis, use of the method does not require advanced mathematical training or theorem proving skills, and hence most software developers should be able to apply the method without extraordinary effort. The method is supported by a suite of tools, carefully integrated to work together. The advantage of a suite of tools is that properties shown to hold using one tool may simplify the analysis performed with a second tool. For example, proving with our consistency checker that a specification is deterministic can simplify later analysis with a model checker [11]. Further, as shown below, some tools (e.g., a model checker and a simulator) can be used in concert to detect inconsistencies.

A practical method should be supported by analysis techniques that can be invoked with the mere push of a button. One example of a "pushbutton" technique is automated consistency checking [32]. Another possible pushbutton technique is model checking. A significant problem in model checking software specifications arises, however, because software specifications routinely contain a wide range of variable types (including types with infinite ranges, such as real numbers) and little regularity or symmetry. As a result, their state spaces are often enormous—in many cases, infinite. Hence, before practical software specifications can be analyzed with a model checker, the *state explosion* problem must be addressed—i.e., the size of the state space to be analyzed must be reduced.

A promising way to reduce state explosion in model checking, proposed by Clarke and his colleagues in 1994, is to apply abstraction [15]. Currently, two different approaches to constructing abstractions are used in model checking. The first approach (and, according to some, the most common [43]) is to develop the abstraction in ad hoc ways—the correspondence between the abstract model and the original specification is based on informal, intuitive arguments. A second approach is mathematically sound but requires user ingenuity to construct the abstraction (see, e.g., [15], [27], [28]). 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., [44]).

Our use of abstraction in model checking differs from both of these approaches. In contrast to the first approach, ours is guaranteed to be mathematically sound. In contrast to the second, our abstraction methods are practical and efficient: they automatically construct a sound and, under certain restrictions, complete abstraction without requiring mathematical sophistication on the part of the user. Because our abstractions are sound and often complete, they may be safely applied repeatedly, thus compounding the magnitude of the state space reductions. This can lead to huge state space reductions.

1.3 Applying the Method

To demonstrate our method, this paper describes how its use uncovered a violation of a critical system property in an operational specification of a United States military system. Both the property and the operational specification were taken from a draft software requirements specification (SRS) prepared by a military contractor. The SRS describes the required behavior of a moderately large program supporting a Weapons Control Panel (WCP). Program officials estimate that the WCP program will contain on the order of 15K lines of source code. Incorrect behavior of the WCP can lead to serious accidents, such as the premature or unintended release of a weapon, serious injury to an operator, or major damage to a weapon. To prevent behavior that could result in such accidents, the SRS contains precise prose descriptions of six properties, including the critical property mentioned above, that the WCP must satisfy to operate safely.

To apply the SCR method to the WCP SRS, we first translated the SRS into the SCR tabular notation. Then, we applied the consistency checker and the dependency graph browser, which automatically exposed numerous errors in the original contractor SRS, most of which we were able to correct. Next, we translated the six safety properties into logical formulas. Our initial analysis of the WCP specification suggested that checking the validity of these properties would be nontrivial, partly because the validity of each property depends on numerous variable definitions distributed throughout the specification. This fact coupled with the significant complexity of some parts of the WCP specification suggested further that using inspection to analyze the properties was impractical. Hence, we applied automation—in particular, the Spin model checker—which quickly detected an inconsistency between the operational SCR specification and one of the properties. Executing the scenario returned by Spin in the SCR simulator demonstrated a safety violation that could damage the weapons system.

Applying the SCR tools to the WCP specification was clearly valuable: as noted, the SCR toolset exposed a number of errors, including a serious safety violation, in the original contractor SRS. Applying the SCR tools was also cost-effective. Translating the SRS to the SCR notation, detecting and correcting errors with the automated consistency checker, and detecting and validating the safety violation required slightly more than one person-week, an extremely small effort given the large size and significant complexity of the SRS and the fact that the military contractor developed the SRS *with no knowledge* of the SCR method and tools.

1.4 Contributions of the Paper

In [9], [11], we propose two abstraction methods useful in model checking requirements specifications, demonstrate the methods on two small examples, and describe how SCR specifications can be translated into the languages of Spin and the symbolic model checker SMV [56]. In [31], we describe our experiences in applying formal methods to the WCP specification, e.g., the positive reaction to the customized WCP simulator that we developed and the barriers that hindered the transfer of our technology to the government contractor. This paper gives the details of our approach to abstraction and shows how our automatable abstraction methods make model checking software specifications practical. We illustrate our abstraction methods by describing our analysis of the SRS for the WCP. This paper makes the following contributions:

- We demonstrate our approach to "pushbutton" model checking by showing how automatable abstraction methods can be supported in SCR. Our approach to abstraction has four important aspects. First, the focus is on properties commonly found in requirements specifications—state and transition invariants. Second, the abstraction is constructed based on a single property, rather than a collection of properties. Third, two automatable abstraction techniques are applied: *variable restriction*, which eliminates variables irrelevant to the property of interest from the specification, and the *variable abstraction* technique discussed below. Fourth, our approach demonstrates how candidate counterexamples in the original state machine can be computed from counterexamples found in the abstract machine.
- We introduce a technique for automating an important special case of the *variable abstraction* method proposed by Clarke et al. [15]. This technique replaces a "detailed" variable in the original specification (that is, a variable with a large, sometimes infinite, range of values) with a more abstract variable. Our technique analyzes all occurrences of particular detailed variables in a concrete SCR specification to determine whether abstraction of the variables is possible. If so, the technique derives the appropriate type abstractions and transforms the concrete specification into an abstract version in which these variables are replaced by abstract variables.
- We introduce a theoretical framework within which we can demonstrate the correctness of a set of abstractions constructed automatically and in standard ways from SCR requirements specifications. While our theory of abstraction may be viewed as a specialization of the very general theory developed by Loiseaux et al. [51], our emphasis on particular automatable, computationally inexpensive abstraction methods and on invariant properties rather than properties of execution sequences allows us to prove general correctness properties of our abstractions within our simpler framework. This emphasis also distinguishes our approach to abstraction from the approaches of Loiseaux, Clarke, Kurshan [50], and others.
- To demonstrate the utility of the SCR tools, we describe their application to the WCP SRS. Applying the SCR tools to this large, contractor-produced specification demonstrates how a relatively complete set of automated techniques can expose inconsistencies in software specifications. While the use of tools such as simulators, model checkers, equivalence checkers, and code synthesizers is becoming standard in hardware design [48], suites of tools customized to expose inconsistencies in software specifications are rare.

1.5 Summary of the Paper

Section 2 reviews the formal underpinnings of the SCR method for requirements specification and briefly describes recent applications of the SCR tools to practical systems. Section 3 describes the contractor-developed SRS of the WCP, the translation of the SRS into the SCR tabular notation, and the application of the consistency checker and the dependency graph browser to the WCP specification. Section 4 summarizes our theory of abstraction, reviews the two abstraction methods described in [11], and introduces the variable abstraction technique mentioned above. Section 5 shows how this technique and another abstraction method were used to dramatically reduce the state space of the state machine model that underlies the original SRS for the WCP. (Because the WCP specification includes numerous real-valued variables, the state space of the WCP specification is infinite.) Section 5 also describes our use of Spin to detect a safety violation in the reduced model and the use of the SCR simulator to demonstrate the violation in the complete SCR specification. Section 6 discusses several issues, such as the role of redundancy in requirements specifications, criteria useful in selecting an analysis technique to detect inconsistencies, and how our abstraction methods can be used with methods other than SCR. Section 7 describes related work. Finally, Section 8 presents some conclusions.

2 The SCR Requirements Method: Background

The tabular-based SCR method was formulated in 1978 to specify the requirements of the Operational Flight Program (OFP) of the United States Navy's A-7 aircraft [38], [39]. During the '80s and the early '90s, many organizations in industry, including Bell Laboratories [40], Grumman [57], Ontario Hydro [60], and Lockheed [25], applied the SCR requirements method to practical systems. The largest application of SCR to date occurred in 1993-1994 when engineers at Lockheed used a version of SCR called CoRE [24] to specify the complete requirements of the OFP of Lockheed's C-130J aircraft [25], a program containing approximately 230K lines of Ada code [63]. The sizable C-130J requirements specification, which contains more than 1,000 tables, provides strong evidence that the SCR method scales.

Each of the above applications of SCR had, at most, weak tool support. To provide powerful, robust tool support customized for the SCR method, we have developed the SCR toolset. To provide formal underpinnings for the the method, we have developed a formal model which defines the semantics of SCR requirements specifications [37], [32]. Below, we review the SCR requirements model and then briefly describe three pilot projects in which other groups used the SCR tools to analyze requirements specifications of safety-critical systems. For more details about the SCR model, see [32], [9], [11].

2.1 SCR Requirements Model

An SCR requirements specification describes the required relation between the system environment, which is nondeterministic, and the required system behavior, which is usually deterministic [32], [37]. The system environment contains *monitored quantities*, quantities that the system monitors, and *controlled quantities*, quantities that the system controls. The SCR model represents these environmental quantities as *monitored* and *controlled variables*. The environment nondeterministically produces a sequence of input events, where an *input event* is a change in some monitored quantity. The system, represented in the formal model as a state machine, begins execution in some initial state and then responds to each input event in turn by changing state and by possibly producing one or more output events, where an *output event* is a change in a controlled quantity. In SCR, the system behavior is assumed to be synchronous (similar to Esterel's Synchrony Hypothesis [7]): 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 Four Variable Model [61]. NAT describes the constraints imposed by physical laws and the system environment. REQ describes the required relation between the monitored and the controlled variables that the system must enforce. To specify REQ concisely, SCR specifications use two types of auxiliary variables, *mode classes*, whose values are called *modes*, and *terms*. In many cases, mode classes and terms capture historical information.

In our requirements model, a system Σ 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. In the initial version of the SCR formal model, the transform T is deterministic, i.e., a function that maps an input event and the current state to a (unique) new state. Each transition from one state to the next state is called a *state transition* or, alternately, a *step*. To compute the next state, the transform T composes smaller functions, called *table functions*, which are derived from the condition tables, event tables, and other tables that comprise an SCR requirements specification using the semantics in [32], [37]. These tables describe the values of the *dependent variables*—the controlled variables, the mode classes, and the terms. Our formal model requires the information in each table to satisfy certain properties. These properties guarantee that each table describes a total function.

The SCR requirements model includes a set $RF = \{r_1, r_2, \ldots, r_n\}$ containing the names of all state variables (i.e., the monitored and dependent variables) in a given specification and a function TY which maps each variable to its set of legal values. In the model, a *state* s is a function that maps each variable in RF to its value; i.e., for all $r \in RF$, $s(r) \in TY(r)$. A *condition* is a predicate defined on a system state, whereas 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 (or a condition) changes from one state to the next, we say that an event "occurs." The notation "$\texttt{@T}(c)$," which denotes an event, is defined by

$$\texttt{@T}(c) \overset{\Delta}{=} \neg c \wedge c'$$

where the unprimed condition c is evaluated in the *old* (or *current* state) and the primed condition c' is evaluated in the *new* (or *next* state). (In this paper, an unprimed variable refers to the variable's value in the current state, whereas a primed variable refers to the variable's value in the next state. Analogous notation is also used to distinguish conditions in the old state from conditions in the new state.) The notation "$\texttt{@F}(c)$" is defined by $\texttt{@F}(c) = \texttt{@T}(\neg c)$. Informally, "$\texttt{@T}(c)$" means that condition c becomes true, and "$\texttt{@F}(c)$" means that c becomes false.

To compute the new state, the transform T uses the values of variables in both the old state and the new state. Generally, the value of a variable in the new state may "directly" depend on the values of variables in the old state or the new state. To define the *direct dependency* relation, we define a dependency set $\mathcal{D}$, a set of ordered pairs (x, y), where $x, y \in RF$ and $(x, y) \in \mathcal{D}$ iff the value of y in either the old state or new state is an argument of the function defining x [11].

To avoid circular definitions, the direct dependencies of given variables on other variables in the same state are required to define a partial order. Because they have no dependencies on other variables (they represent inputs from 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 in the partial order. The assumptions that the tables define total functions 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).

2.2 Practical Applications of the SCR Tools

Recently, the practical utility of the SCR tools for detecting errors in software specifications has been evaluated in three pilot projects. In one project, researchers at NASA's IV&V facility used the SCR consistency checker to detect several ambiguities and missing assumptions in the prose requirements specification of software for the International Space Station [21], [22]. 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 [58]. In a third project, researchers at the Jet Propulsion Laboratory (JPL) used the SCR consistency checker and the simulator to analyze specifications of two components of NASA's Deep Space-1 spacecraft for errors [53]. 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.

3 Specifying the SRS for WCP in SCR

3.1 The WCP and the Contractor SRS

Operators of a United States military system use the Weapons Control Panel (WCP) to monitor the status and set up the launch of one or more weapons. The WCP is linked to numerous subsystems and I/O devices that support the launch operation. In particular, it interacts with the LCS Launch Control System, (LCS) which determines when one or more weapons are to be launched and oversees launch preparation. Operators use the panel to open and close valves and doors and to monitor the doors, valves, and other system devices for faults. The panel consists of lights, numeric displays, and switches. The lights display nonnumeric sensor information (e.g., a door is open, a subsystem has failed) and commands from the LCS (e.g., Make Launcher Ready). Numeric displays present numeric information read from sensors, such as air pressure. Switches instruct the WCP software to energize and de-energize solenoids, the actuators that open and close doors and valves.

The contractor-developed SRS of the required behavior of WCP is a semiformal document composed of formal descriptions, diagrams, and prose. It describes a total of 258 variables (108 input variables, 90 output variables, and 60 internal variables), listing in tables the names, types, and source or destination (device or subsystem) of all system inputs and outputs. In contrast to most other SRSs that we have reviewed, the SRS for the WCP is highly precise and relatively complete—it contains a precise description of most of the information needed in a "build-to" specification. In contrast to most requirements specifications for practical systems, which consist largely of prose, the SRS uses formal descriptions, in particular, a set of assignment statements,[1] to specify the values of most WCP outputs and internal variables. The remaining variables are described more informally. In particular, the Boolean outputs and internal variables that are not defined by assignment statements and all numeric outputs are described in prose. Also described in prose are four modes of operation (initialization, monitor, operate, and test) and how the values of various outputs and internal variables differ depending upon the mode (e.g., relays are disabled in monitor mode). With only a few exceptions, the assignment statements do not refer to modes.

3.2 Translation of the SRS to SCR

The existence in the SRS of input, output, and internal variables and of assignment statements to define the output and internal variables facilitated the translation of the SRS into the SCR notation. The WCP inputs correspond to SCR monitored variables, the WCP outputs to SCR controlled variables, the WCP internal variables to terms and modes, and the WCP assignment statements to SCR tables. Because modes were not used systematically in the SRS (as noted above, most assignment statements ignore the modes), we decided to follow the lead of the SRS and represent the WCP modes as Boolean terms rather than as SCR modes. Thus, both internal variables and modes in the SRS are represented as terms in the SCR specification, and SCR modes are not used.

To obtain an online specification of the SRS, we electronically scanned the variable tables and the assignment statements from the SRS into a computer file and used optical character recognition to convert the scanned images to text. To obtain an SCR specification, we edited the text in minor ways (e.g., removed embedded blanks from the variable names) and then translated the results into SCR tables, inserting traceability links back to the SRS.

To evaluate the SCR version of the SRS, we first invoked the consistency checker, which automatically exposed some type errors, a few missing cases, and numerous inconsistencies in the definitions and uses of variables. Nearly all of these errors were traced to errors in the original specification. Due to the extensive formality of the original SRS, the accuracy of electronic scanning and optical character recognition, and our limited manual effort, translating the original contractor SRS into SCR had apparently introduced few new errors. We corrected the type errors, some variable name discrepancies, and other minor problems in the SCR specification. We also defined "reasonable" initial values. (Although the SRS generally omitted initial values, some initial values were found in another contractor specification.) Because we are not domain experts, we were unable to fix every error detected by consistency checking (e.g., some unused and undefined variables and a few missing cases). Finally, to maintain close compatibility between the contractor SRS and the SCR specification, we avoided making major changes in translating the original SRS into SCR. Compatibility between the two specifications could be important later in the project in convincing United States government personnel and the government contractor that problems exposed by our methods were actual problems in the SRS, *not* problems introduced by our translation into SCR.

Translating the SRS into SCR and applying our analysis tools reduced the total number of variables from 258 to 233. Of the 233 variables, 74 were monitored variables (35 Booleans, nine reals, two integers, and 28 enumerated types), 72 were terms (67 Booleans, three reals, and two integers), and 87 were controlled variables (72 Booleans, four reals, and 11 enumerated types). The reduction in the number of input and output variables was due partly to the elimination of duplicate variable names. Moreover, in many cases, the original SRS used *n* Boolean variables to represent an input with *n* possible values. To clarify the WCP specification and improve its readability, we modeled each of these inputs as a single enumerated type variable with *n* values, and in the process eliminated numerous Boolean variables. The increase in the number of internal variables (represented in SCR as terms) resulted because our earlier informal analysis had overlooked several internal variables in the prose portion of the contractor SRS.

1. The SRS calls these *logic equations*. Calling them *assignment statements* is more accurate.

Another tool in the toolset, the dependency graph browser (DGB), displays a graph whose nodes represent all variables in an SCR specification and whose edges indicate the variable dependencies [35]. Fig. 1 shows the dependency graph for the complete SCR specification of WCP after the above improvements. (Fig. 1 omits the variable names because they are unreadable.) To determine the dependency graph, the SCR toolset uses the condition tables and event tables to identify the variables on which each term and controlled variable depends in both the old and new states. The leftmost nodes represent the 74 monitored variables, none of which depends upon other variables. The rightmost nodes represent the 87 controlled variables, which can depend on any other variables. In the middle of the graph are nodes representing the 72 terms, each of which can depend on monitored variables and any preceding terms in the partial order.

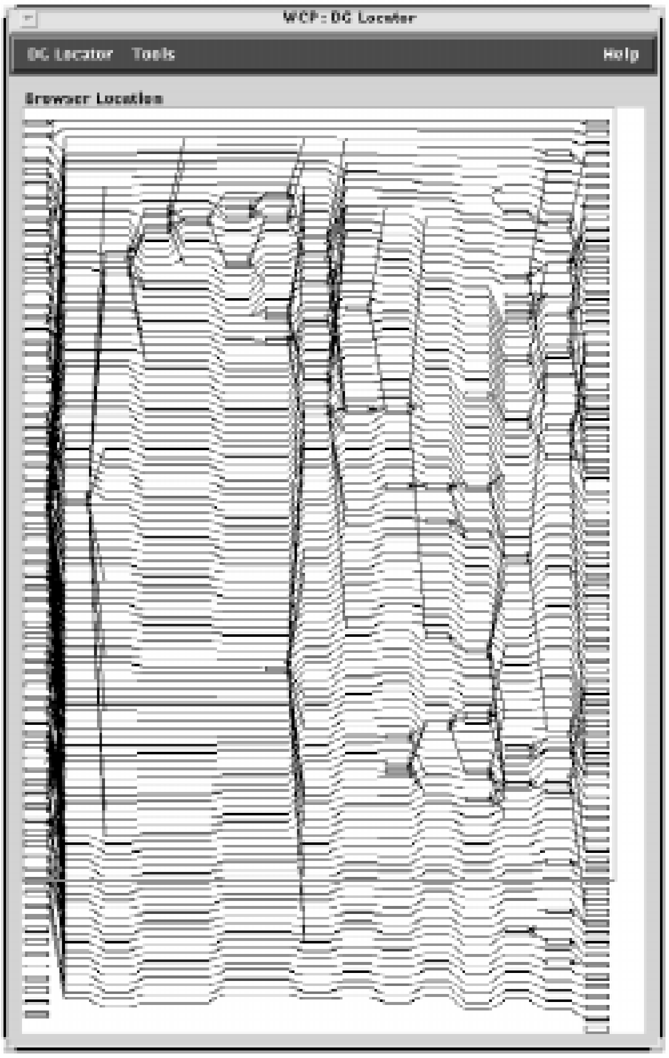


Fig. 1. Dependency graph for SCR Specification of WCP.

In addition to displaying the relationship among variables, the dependency graph also exposes unused variables, undefined variables, and circular dependencies. (The consistency checker also exposes such errors.) For example, nine unused monitored variables appear in the bottom left-hand corner and one undefined output variable in the bottom right-hand corner of Fig. 1. These 10 unconnected variables are among the problems we were unable to correct without more domain knowledge. In addition to displaying the dependency graph, the DGB facilitates user navigation of the specification. By clicking on the node that corresponds to a dependent variable, the user can display the table describing that variable. The DGB can also be used to select and extract parts of the specification for further analysis. Clearly, the enormous size of the dependency graph for the WCP limits its usefulness. Using the DGB to extract and analyze parts of the WCP specification was, therefore, extremely valuable.

3.3 Representing the WCP Functions as Tables

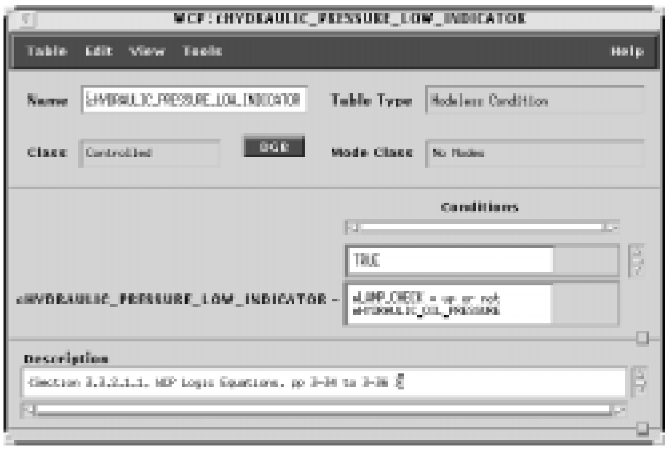
In SCR, each controlled variable and term is described by either a condition table or an event table. A condition table describes a variable as a function of other variables in the same state; an event table describes a variable as a function of variables in two consecutive states. The value of a variable defined by a condition table may or may not depend on history (i.e., previous system behavior): the value of the variable depends on history if the variable is defined in terms of another variable whose value is history-dependent. In contrast, the value of a variable defined by an event table is always history-dependent.

Most assignment statements in the SRS represent the value of an output or internal variable as a function of other variables in the same state. Such assignments are naturally represented in SCR with condition tables. For example, the condition table in Table 1 represents the following assignment statement from the SRS:

cHYDRAULIC_PRESSURE_LOW_INDICATOR :=
  mLAMP_CHECK=up or not **mHYDRAULIC_OIL_PRESSURE**

This table, which defines an output as a simple function of two inputs, states that the controlled variable **cHYDRAULIC_PRESSURE_LOW_INDICATOR** is *true* if **mLAMP_CHECK** is **up** or **mHYDRAULIC_OIL_PRESSURE** is *false*, and *false* otherwise. (In the tables and expressions in this section, the initial letter of each variable name indicates the class of the variable: 'm' indicates a monitored variable, 'c' a controlled variable, and 't' a term.)

TABLE 1
CONDITION TABLE DEFINING
cHYDRAULIC_PRESSURE_LOW_INDICATOR



Name	cHYDRAULIC_PRESSURE_LOW_INDICATOR
Table Type	Modeless Condition
Class	Controlled
Mode Class	No Modes

	Conditions
cHYDRAULIC_PRESSURE_LOW_INDICATOR =	TRUE
	mLAMP_CHECK = up or not mHYDRAULIC_OIL_PRESSURE

Description
Section 3.3.2.1.1. WCP Logic Equations, pp 3-34 to 3-36

The remaining assignment statements in the SRS define 20 Boolean variables called *latches*. In the SRS, most assignment statements for a latch x are of the form

$$x := (y \vee x) \wedge z, \qquad (1)$$

where x is a Boolean variable and y and z are conditions (i.e., predicates defined on the system state). Informally, the latch x becomes *true* when the expression $(y \wedge z)$ becomes *true*, *false* when z becomes *false*, and otherwise does not change. Because the values of latches are history-dependent, they are represented in SCR as event tables. The event table used to represent these latches has the form

	$@T(y \wedge z)$	$@F(z)$
$x' :=$	*true*	*false*

Based on the semantics of event tables presented in [32] and the definitions of the event operators "`@T`" and "`@F`" presented in Section 2.1, the latch x' is defined by[2]

$$x' = \begin{cases} \mathit{true} & \text{if } y' \wedge z' \wedge \neg(y \wedge z) \\ \mathit{false} & \text{if } \neg z' \wedge z \\ x & \text{otherwise.} \end{cases}$$

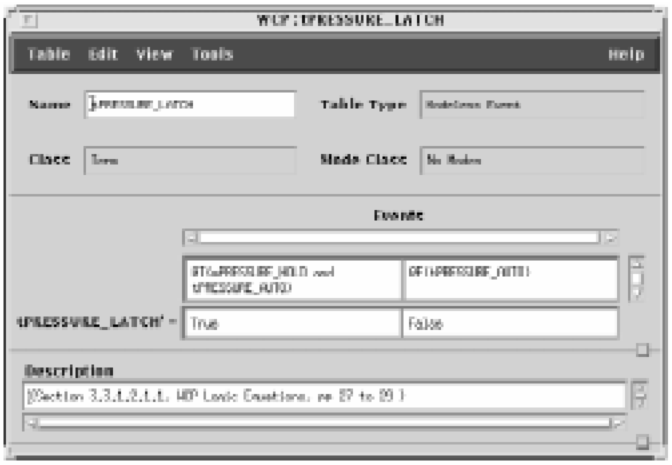
The other assignment statements for latches are variations of the form shown in (1) and can be similarly translated into event tables. Two safety engineers familiar with the WCP have confirmed our interpretation of latches.

In the SRS, a specific example of a latch is the internal variable **`tPRESSURE_LATCH`**, which is defined by the assignment statement

`tPRESSURE_LATCH` := **`tPRESSURE_AUTO`**
and (**`mPRESSURE_HOLD`** or **`tPRESSURE_LATCH`**).

Table 2, which represents this assignment statement as an event table, states that **`tPRESSURE_LATCH`** is true in the new state if either **`mPRESSURE_HOLD`** or **`tPRESSURE_AUTO`** is false in the old state and both **`mPRESSURE_HOLD`** and **`tPRESSURE_AUTO`** are true in the new state, **`tPRESSURE_LATCH`** is false in the new state if **`tPRESSURE_AUTO`** becomes false in the new state, and **`tPRESSURE_LATCH`** is unchanged otherwise.

TABLE 2
EVENT TABLE DEFINING **`tPRESSURE_LATCH`**



Name	tPRESSURE_LATCH	Table Type	Modeless Event
Class	Term	Mode Class	No Modes

Events

	@T(mPRESSURE_HOLD and tPRESSURE_AUTO)	@F(tPRESSURE_AUTO)
tPRESSURE_LATCH' =	True	False

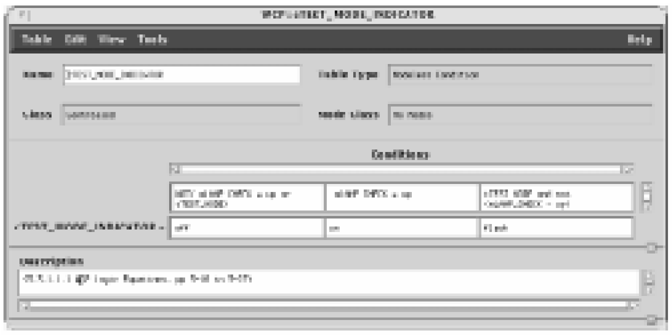
Description
[Section 3.3.1.2.1.1. WCP Logic Equations, pp 27 to 29]

2. In any initial state $s_0 \in S_0$, x, y, and z must satisfy the predicate $(x \wedge y \wedge z) \vee (\neg x \wedge \neg z) \vee (\neg y \wedge z)$.

Like the controlled variable defined by Table 1, many controlled variables in the SRS (41 out of 87) are simple functions of the monitored variables which do not depend on history and hence are defined by condition tables. Most of these 41 variables represent lights on the operator control panel. An additional 12 controlled variables are functions of terms and monitored variables and also do not depend on history. These controlled variables, as well as the terms on which they depend, are defined by condition tables. The values of the remaining 34 controlled variables are all defined in terms of previous system behavior. Hence, these variables are either defined by event tables or are functions of one or more terms defined by event tables. These 34 controlled variables, the source of most of the complexity in the WCP SRS, represent the states of valves, doors, shutters, and other hardware devices involved in preparing the launch of a weapon.

An SCR specification is designed to describe the required behavior without implementation detail. Whether to remove implementation detail in the original SRS from the SCR specification was an issue. In a few cases, we deviated slightly from the SRS and removed the detail, substituting an equivalent representation with the same externally visible behavior. To illustrate these cases, we consider the variable **`cTEST_MODE_INDICATOR`**, which represents a light on the operator control panel. The SRS represents the light as a Boolean that depends on a flasher circuit. We decided that the required behavior was clearer if the SCR specification omitted the flasher circuit and therefore represented the light as an enumerated type with three values—**`off`**, **`on`**, and **`flash`**—rather than two—**`off`** and **`on`**. Table 3, the condition table which defines **`cTEST_MODE_INDICATOR`**, states that the light is **`on`** when the switch **`mLAMP_CHECK`** is up, **`off`** when the **`mLAMP_CHECK`** switch is down and the system is not in **`tTEST_MODE`**, and **`flash`** when the **`mLAMP_CHECK`** switch is down and the system is in **`tTEST_MODE`**.

TABLE 3
CONDITION TABLE DEFINING **`cTEST_MODE_INDICATOR`**



Name	cTEST_MODE_INDICATOR	Table Type	[illegible]
Class	[illegible]	Mode Class	No Modes

Conditions

	NOT(mLAMP_CHECK = up or tTEST_MODE)	mLAMP_CHECK = up	tTEST_MODE and not (mLAMP_CHECK = up)
cTEST_MODE_INDICATOR =	off	on	flash

Description
[illegible]

3.4 Representing the WCP Inputs in SCR

In SCR, the behavior of a monitored variable is represented as a simple state machine with a set of possible states (defined by the "type-of" function TY), a next-state relation, and an initial state set. For example, the monitored variable **`mLAMP_CHECK`**, which indicates the position of a switch, has the set of possible values {**`up`**, **`down`**}, the next-state relation {(**`down`**, **`up`**), (**`up`**, **`down`**)}, and the initial value **`down`**. Similarly, the monitored variable **`mBANK_SWITCH_MODE`**, which indicates the position of a dial, has the set of possible values {**`off`**, **`monitor`**, **`operate`**}, the next-state relation {(**`off`**, **`monitor`**), (**`monitor`**, **`operate`**), (**`operate`**, **`monitor`**),

(`monitor`, `off`)}, and the initial value `off`. Note that the user cannot rotate the dial directly from `off` to `operate` or vice versa.

Like many other researchers (e.g., [59], [55], [42]), we use an interleaving model: If two monitored quantities change simultaneously, the SCR model represents the changes as two input events that are processed in sequence in either order. In SCR, interleaving is captured by the One Input Assumption—at each state transition, exactly one monitored variable changes. In the WCP specification, therefore, we assume that, at each transition, either `mLAMP_CHECK` or `mBANK_SWITCH_MODE` or one of the 72 remaining monitored variables changes. If `mLAMP_CHECK` changes, it can only change in one way—from `down` to `up` or vice versa. In contrast, if `mBANK_SWITCH_MODE` changes, it may change in more than one way; for example, starting in the position `monitor`, it may change to `off` or to `operate`.

3.5 Computing the WCP Transitions in SCR

Analyzing and simulating SCR specifications requires that the semantics of state transitions be defined. As noted above, the transform T computes the new state from an enabled input event and the current state. The current state and the next-state relations of the monitored variables determine which input events are enabled. Once one of the enabled input events is selected, the new state can be computed from the selected input event, the current state, and the functions (derived from the SCR tables) that define the values of the dependent variables. To represent these functions and the next-state relations of the monitored variables, reference [9] introduced "conditional assignments," which are similar to the enumerated assignments of UNITY [14]. Below, we give conditional assignments for the dependent variables described by Tables 2 and 3 and for the next-state relation associated with the monitored variable `mBANK_SWITCH_MODE`. We also describe how we use conditional assignments to compute the next state.

Each variable r_i in an SCR specification is associated with a conditional assignment of the form:

```
if
   □ $g_{i,1} \rightarrow r_i := v_{i,1}$
   □ $g_{i,2} \rightarrow r_i := v_{i,2}$
⋮
   □ $g_{i,n_i} \rightarrow r_i := v_{i,n_i}$
fi
```

Here, the $g_{i,j}$ are Boolean expressions (guards) and the $v_{i,j}$ are expressions that are type compatible with variable r_i. The $g_{i,j}$ and the $v_{i,j}$ may refer to both old and new values of r variables, provided that the references do not lead to circular definitions [37], [32].

The conditional assignment for the term `tPRESSURE_LATCH`, which can be derived from the event table in Table 2 using the semantics in [32], [37] is given by

```
if
   □ @T(mPRESSURE_HOLD AND tPRESSURE_AUTO)
       -> tPRESSURE_LATCH := true
   □ @F(tPRESSURE_AUTO)
       -> tPRESSURE_LATCH := false
fi
```

Similarly, the conditional assignment for the controlled variable `cTEST_MODE_INDICATOR` is derived from the condition table in Table 3 using the semantics in [32], [37] and is given by

```
if
   □ NOT(mLAMP_CHECK=up OR tTEST_MODE)
       -> cTEST_MODE_INDICATOR := off
   □ mLAMP_CHECK=up
       -> cTEST_MODE_INDICATOR := on
   □ tTEST_MODE AND NOT(mLAMP_CHECK=up)
       -> cTEST_MODE_INDICATOR := flash
fi
```

As stated in Section 2.1, the information in the SCR tables must satisfy the properties described in [32]. For example, in a condition table, two properties are required of the conditions c_i in each row: 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 in each row must be false.

These and other properties required of condition tables and event tables guarantee that, at each transition, either one guard or no guard of the conditional assignment for a dependent variable will be true. The properties given above for condition tables guarantee that in the conditional assignment derived from a condition table, exactly one guard will be true at each transition. Similarly, the property given above for event tables and the assumption that events which never change the value of the variable defined by the event table are omitted imply that in the conditional assignment derived from an event table, either one guard or no guard will be 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.

The conditional assignment for the monitored variable `mBANK_SWITCH_MODE` is given by

```
if
   □  mBANK_SWITCH_MODE=off
        -> mBANK_SWITCH_MODE := monitor
   □  mBANK_SWITCH_MODE=monitor
        -> mBANK_SWITCH_MODE := operate
   □  mBANK_SWITCH_MODE=monitor
        -> mBANK_SWITCH_MODE := off
   □  mBANK_SWITCH_MODE=operate
        -> mBANK_SWITCH_MODE := monitor
fi
```

This conditional assignment states, for example, that starting in the current state in the `off` position, `mBANK_SWITCH_MODE` is enabled to change to `monitor` in the next state, whereas starting in the current state in the `monitor` position, `mBANK_SWITCH_MODE` is enabled to change to either `off` *or* to `operate` in the next state. Conditional assignments for `mLAMP_CHECK` and other monitored variables in the WCP specification are expressed similarly. (In the case of real-valued variables and other variables with very large type sets, we use an alternate, more compact representation, based on a predicate rather than a conditional assignment. See [11] for an example.) For each monitored variable, more than one guard of the corresponding conditional assignment may be true at a given transition, and each assignment associated with a true guard is enabled.

Because the WCP SRS does not constrain the manner in which most monitored variables change at each transition, most WCP monitored variables are enabled to change to any other value in their type sets. The disadvantage of this approach is that "impossible" transitions may occur. For example, often a monitored variable with numerical values cannot assume a value below some minimum in one state and above some maximum in the next state. However, to minimize the differences between the original SRS and the SCR specification, we allowed such behavior. This means that some property violations detected by model checking may be spurious because the changes in monitored variables that produce them are impossible.

Given a current state and the conditional assignments for *all* input variables, the set of input events that are enabled in the current state can be determined by evaluating each guard. Each guard that evaluates to true, along with the associated assignment, determines an input event that is enabled to occur in the next state. Because only a single input event can occur at each transition, one of the enabled input events is selected nondeterministically by the environment. As noted above, the selected input event and the current state determine a unique new state. The values of the monitored variables in the new state are determined solely by the input event; the values of the other variables in the new state (the dependent variables) can be computed from the conditional assignments for these variables. The partial order of the variables (described in Section 2.1) determines the sequence in which the conditional assignments are evaluated to compute the new state.

3.6 Representing the Safety Properties in SCR

To illustrate the six safety properties in the SRS, we consider two of these properties. Property 1 states,

> "*Opening the Launcher Vent Valve shall be prevented unless the weapon-to-launcher differential pressure is within safe limits.*"

More specifically, the vent valve shall open only if at least one of two transducers reports a value in a "safe region" in the new state. Property 2 states,

> "*When the launcher receives the LCS Pressure Hold signal, the Pressurize Valve shall shut, and the Pressure Vent Blocking Valve shall open.*"

Formal statements of these two properties are given by

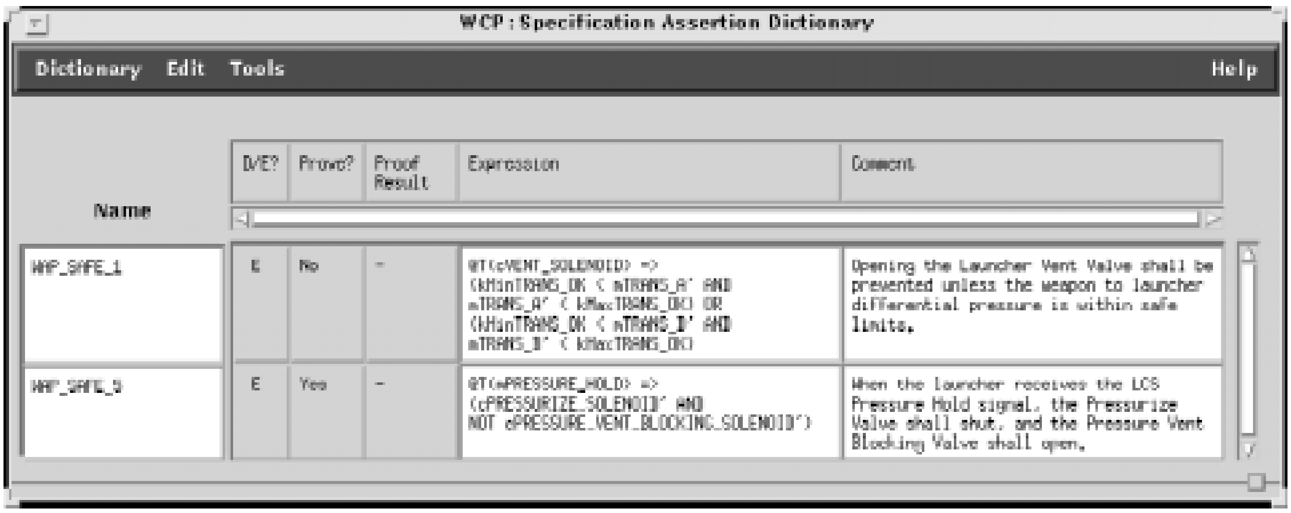
1) $@T(\texttt{cVENT\_SOLENOID}) \Rightarrow \texttt{kMinTRANS} < \texttt{mTRANS\_A}' \wedge \texttt{mTRANS\_A}' < \texttt{kMaxTRANS} \vee \texttt{kMinTRANS} < \texttt{mTRANS\_B}' \wedge \texttt{mTRANS\_B}' < \texttt{kMaxTRANS}$

and

2) $@T(\texttt{mPRESSURE\_HOLD}) \Rightarrow \texttt{cPRESSURIZE\_SOLENOID}' \wedge \mathrm{NOT}(\texttt{cPRESSURE\_VENT\_BLOCKING\_SOLENOID}')$.

In Property 1, **mTRANS_A** and **mTRANS_B** represent the two transducers, and **kMinTRANS** and **kMaxTRANS** represent the constants 7.7 and 15.3. Translating the prose versions of the safety properties into logical formulas required consultation with safety engineers familiar with WCP. The property-based specification of WCP, which consists of these six safety properties, appears in an Assertion Dictionary, one of several dictionaries in an SCR specification. Fig. 2 shows how the two above properties appear in the SCR specification of WCP.

Like the other four properties in the contractor SRS, Properties 1 and 2 describe a required relation between the monitored and controlled variables, that is, a required constraint on the externally observable WCP behavior. Because each of the six controlled variables that appear in the six properties is a function of many terms and monitored variables, evaluating the validity of a property requires the analysis of execution sequences involving a large number of terms and environmental variables. For example, checking the validity of Property 1 requires the analysis of execution sequences involving 55 variables—20 monitored variables, 34 terms, and one controlled variable. Checking the validity of the other five properties requires the analysis of execution sequences containing 18, 40, 41, 53, and 56 variables, respectively. Because the number of variables involved in analyzing each property is so large and because the values of many of the relevant variables depend on history, the use of inspection to check the WCP specification (even the SCR version) for property violations would be error-prone and extremely time-consuming. Clearly, automated analysis, if feasible, is a more cost-effective approach than inspection for detecting property violations in the WCP specification.



Name	D/E?	Prove?	Proof Result	Expression	Comment
WAP_SAFE_1	E	No	-	@T(cVENT_SOLENOID) => (kMinTRANS_OK < mTRANS_A' AND mTRANS_A' < kMaxTRANS_OK) OR (kMinTRANS_OK < mTRANS_B' AND mTRANS_B' < kMaxTRANS_OK)	Opening the Launcher Vent Valve shall be prevented unless the weapon to launcher differential pressure is within safe limits.
WAP_SAFE_5	E	Yes	-	@T(mPRESSURE_HOLD) => (cPRESSURIZE_SOLENOID' AND NOT cPRESSURE_VENT_BLOCKING_SOLENOID')	When the launcher receives the LCS Pressure Hold signal, the Pressurize Valve shall shut, and the Pressure Vent Blocking Valve shall open.

Fig. 2. Assertion dictionary showing two WCP properties.

4 BUILDING ABSTRACTIONS OF SCR SPECIFICATIONS

The properties analyzed with model checking may be represented as logical formulas. In analyzing SCR specifications, we focus on *state invariants* (properties of reachable states) and *transition invariants* (properties of adjacent pairs of reachable states) because they are common in specifications of practical systems that we have studied, e.g., the A-7 OFP [1] and the WCP. All six properties in the WCP SRS, including the two properties listed above, are transition invariants. (Also interesting to note is that all properties of the TCAS II requirements specification [30] analyzed by Anderson et al. [2], except one designed to expose circular definitions,[3] are state invariants.)

Numerous techniques have been proposed to combat state explosion in model checking (see, e.g., [26]). One promising yet relatively unexplored approach uses abstraction. Below, we describe three methods for deriving abstractions from SCR requirements specifications based on the formula to be analyzed. The first two methods were originally proposed in [10] and are further developed in [9], [11]. All three methods are practical: None requires significant ingenuity on the user's part, and each can be used to derive a smaller, more abstract specification automatically or semiautomatically. Further, each method systematizes techniques that current users of model checkers apply routinely but often in ad hoc ways. The first method always yields a sound and complete abstraction. (The soundness and completeness of abstractions are defined formally in Definition 3 below.) The remaining two methods always yield a sound abstraction and, under certain simple restrictions that frequently hold in practice, also produce complete abstractions.

This section first presents a formal framework for abstraction, taken from [3], which establishes some notation and defines soundness and completeness precisely. The section concludes with a detailed description of the three abstraction methods.

4.1 Formal Framework for Abstraction

Presented below is the formal framework within which we deduce and describe properties of our abstraction methods. First, we define a state machine, state and transition invariants, soundness and completeness, and an abstraction map. Then, Theorem 1 establishes sufficient conditions for abstractions to be sound or complete.

DEFINITION 1. (State Machine). *A* state machine *is a triple* $\Sigma = (S, \Theta, \rho)$, *where:* 1) *the set S of* states *of* Σ *is nonempty,* 2) *the* initial state predicate $\Theta : [S \to Boolean]$ *of* Σ *is true for at least one element of S, and* 3) *the predicate* $\rho : [S \times S \to Boolean]$ *describes the* transition relation *of* Σ. *When* $\Sigma = (S, \Theta, \rho)$ *is a state machine, the states that satisfy* Θ *are called the* initial states *of* Σ *and the pairs of states* (s, s') *that satisfy* ρ *are called the* transitions *of* Σ. *A state of* Σ *is* reachable *if it can be reached by a finite sequence of transitions from an initial state. A transition of* Σ *is* reachable *if it is a transition from a reachable state.*

3. In contrast to Anderson et al. [2], who use model checking to check for circular definitions, the SCR method uses consistency checking [30], a form of static analysis. The use of static analysis to detect circularities is possible because SCR's step semantics is much simpler than the step semantics of RSML [30], the Statecharts variant used to specify TCAS II.

The properties of Σ of interest to us are either *one-state* or *two-state* properties; these are defined, respectively, by predicates on single states or on pairs of states. A one-state (respectively, two-state) property is a *state invariant* (respectively, *transition invariant*) if it is true for all reachable states (respectively, true for all reachable transitions). In SCR applications, the states are represented as mappings from *state variables* to values, and predicates are given by formulas over the state variables of one or two states.

DEFINITION 2 (Machine Abstraction and Sound Predicate Abstraction). *Let* $\Sigma = (S, \Theta, \rho)$ and $\Sigma_A = (S_A, \Theta_A, \rho_A)$ *be state machines.*

1) Σ_A *is a* (machine) abstraction *of* Σ *if there is a map* $\alpha : S \to S_A$, $s \overset{\alpha}{\mapsto} s_A$, *called the* abstraction map, *such that* a) *for all s in S*: $\Theta(s)$ *implies* $\Theta_A(s_A)$ *and* b) *for all s, s′ in S*: $\rho(s, s')$ *implies* $\rho_A(s_A, s'_A)$. *If* Σ_A *is an abstraction of* Σ *under* α, *the map* $\alpha : S \to S_A$ *is a* machine homomorphism *from* Σ *to* Σ_A.
2) *If q and* q_A *are one-state* (*respectively*, *two-state*) *predicates for* Σ *and* Σ_A *and* $\alpha : S \to S_A$, $s \overset{\alpha}{\mapsto} s_A$, *is a map, then* q_A *is a* sound predicate abstraction *of q under* α *if for all s in S*, $q_A(s_A)$ *implies* $q(s)$ (*respectively, for all s,* $\tilde{s}$ *in S*, $q_A(s_A, \tilde{s}_A)$ *implies* $q(s, \tilde{s})$).

DEFINITION 3 (Sound/Complete Abstractions). (Σ_A, q_A) *is a* sound abstraction *for* (Σ, q) *if whenever* q_A *is an invariant for* Σ_A, *q is an invariant for* Σ; (Σ_A, q_A) *is a* complete abstraction *for* (Σ, q) *if it is a sound abstraction and it is also a* refutation-sound *abstraction, i.e., whenever q is an invariant for* Σ, q_A *is an invariant for* Σ_A.

Clearly, any abstraction map α from S to S_A defines an equivalence relation $\equiv_\alpha$ on the states of Σ in which two states are equivalent if they have the same image under α. We say that a predicate q on the elements of a set S, on which there is an equivalence relation $\equiv$, *respects* $\equiv$ (and vice versa) if and only if the truth of q depends only on the equivalence class(es) of its argument(s).

THEOREM 1 (Soundness and Completeness of Abstractions). *Suppose* $\Sigma_A = (S_A, \Theta_A, \rho_A)$ *is an abstraction of* $\Sigma = (S, \Theta, \rho)$ *with abstraction map* $\alpha : S \to S_A$, *i.e.,* α *is a machine homomorphism from* Σ *to* Σ_A.

1) *If* q_A *is a sound predicate abstraction of q under* α, *then* (Σ_A, q_A) *is a sound abstraction for* (Σ, q).
2) *Suppose that every state* $\hat{s} \in S_A$ *reachable in* Σ_A *is the image under* α *of some state* $s \in S$ *that is reachable in* Σ (i.e., $\hat{s} = \alpha(s)$). *Then, if the predicate q is a one-state predicate for* Σ *that respects* $\equiv_\alpha$ *and the predicate* q_A *on* S_A *is defined by* $q_A(\hat{s}) \overset{\Delta}{=} \exists s \in S : \alpha(s) = \hat{s} \wedge q(s)$, *then q is a state invariant of* Σ *iff* q_A *is a state invariant of* Σ_A. *That is,* (Σ_A, q_A) *is a complete abstraction of* (Σ, q).
3) *Suppose that every reachable transition* $(\hat{s}, \hat{s}')$ *of* Σ_A *is the image under* α *of a reachable transition* (s, s') *of* Σ (i.e., $\alpha(s) = \hat{s}$ *and* $\alpha(s') = \hat{s}'$). *Then, if the predicate q is a two-state predicate for* Σ *that respects* $\equiv_\alpha$ *and the*

predicate q_A on $S_A \times S_A$ is defined by $q_A(\hat{s}_1, \hat{s}_2) \stackrel{\Delta}{=} \exists s_1, s_2 \in S : \alpha(s_1) = \hat{s}_1 \wedge \alpha(s_2) = \hat{s}_2 \wedge q(s_1, s_2)$, then q is a transition invariant of Σ iff q_A is a transition invariant of Σ_A. That is, (Σ_A, q_A) is a complete abstraction of (Σ, q).

Based on Definition 3 and Theorem 1, we have established the soundness, and sufficient conditions for completeness, of abstractions of SCR specifications obtained by three specific methods. These methods include the two methods originally proposed in [10] and further developed in [9], [11] and a new third method introduced below. The complete formal exposition of the above framework, including the proof of Theorem 1 and a detailed discussion of the connection between our theoretical results and the three abstraction methods, is presented in [3].

To define the state machine $\Sigma = (S, \Theta, \rho)$ corresponding to an SCR machine represented as a 4-tuple (S, S_0, E^m, T), we define: 1) the initial-state predicate Θ on a state $s \in S$ such that $\Theta(s)$ is true iff $s \in S_0$ and 2) the next-state predicate ρ on pairs of states $s, s' \in S$ such that $\rho(s, s')$ is true iff there exists an event $e \in E^m$ enabled in s such that $T(e, s) = s'$. Thus, the predicate ρ is simply a concise and abstract way of expressing the transform T without reference to events.

4.2 Three Abstraction Methods

The first two abstraction methods use *variable restriction*: they eliminate certain variables and their associated tables (or monitored variable definitions) from the SCR specification. The third abstraction method, developed in this study to transform an infinite state space into a finite state space, uses *variable abstraction*, which replaces a detailed variable with a more abstract variable. Thus, the third abstraction method applies the method suggested by Clarke et al. [15] to SCR specifications. Our abstraction methods are complementary. The first method reduces the state space of the model by removing variables irrelevant to the analysis from the specification, whereas the two other methods reduce the state space by either removing detailed variables or by replacing detailed variables with more abstract variables. Usually, we apply the first abstraction method first and then apply the second or third abstraction methods to the result. Both the second and third abstraction methods may be applied many times, each time to remove one or more detailed variables (in the case of the second method) or to replace detailed variables (in the case of the third method).

4.3 Method 1—Remove Irrelevant Variables

This simple abstraction method, analogous to a technique called "program slicing" which removes irrelevant variables in analyzing programs [65], uses the set of variable names which occur in the formula being analyzed to remove unneeded variables and their definitions (tables in the case of dependent variables) from the analysis. To apply this method to an SCR specification Σ with state variable set RF, we construct the set $O \subseteq RF$ of variables occurring in the formula q. Then, we let set $O^* \subseteq RF$ be the reflexive and transitive closure of O under the direct dependency relation $\mathcal{D}$.

The SCR specification of the abstract machine Σ_A with the set of variables $RF_A = O^*$ is obtained by deleting all associated tables (and, in the case of monitored variables, all associated definitions) for variables in the set $RF - RF_A$. 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. The abstraction map α maps every state s of Σ to the unique state s_A of Σ_A such that $s_A(r) = s(r)$ for all $r \in RF_A$. This abstraction method is always sound and complete. In large specifications, applying this method can significantly reduce the size of the state space to be model checked.

4.4 Method 2—Remove Detailed Monitored Variable

Suppose that $r \in RF$ is a monitored variable which does not appear in the formula q, that $\hat{r} \in RF$ 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 Σ_A, the dependence of $\hat{r}$ on r is eliminated by treating $\hat{r}$ as a monitored variable. The initial state set, the set of possible states, and the next-state relation for the new monitored variable $\hat{r}$—that is, the state machine for $\hat{r}$—can be computed from $\hat{r}$'s initial state set, the table defining $\hat{r}$, and the state machine for r from Σ. 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 directly depends on $r_1, r_2, \ldots, r_m$; $\hat{r}$ depends directly only on $r_1, r_2, \ldots, r_m$; and none of the variables $r_1, r_2, \ldots, r_m$ appear in q. As for Method 1, the abstract property q_A for Method 2 is syntactically identical to property q. The abstraction map α for Method 2 is defined exactly as in Method 1.

Abstractions obtained by this method are always sound. A sufficient condition for completeness, which frequently holds in practice, is that every two states equivalent under $\equiv_\alpha$ must be connected by a finite chain of transitions. For example, this condition holds when the value of $\hat{r}$ is determined by whether r belongs to some particular interval, and the next-state relation of r permits r to change from any value in any particular interval to any other value in the same interval, either in a single step or in incremental steps inside the interval. This sufficient condition guarantees that every reachable abstract state or transition is the image of a reachable concrete state or transition, so that Theorem 1 applies. For a detailed example which illustrates Method 2, see [9], [11].

4.5 Method 3—Replace Detailed Variable with Abstract Variable

Suppose $r \in RF$ is a detailed variable (r may or may not appear in the formula q). Based on q and the details in the SCR specification that determine the next-state relation, one can partition $TY(r)$, the type set of r, into equivalence classes $A_1, A_2, \ldots, A_N$ such that any dependence of q's value and the values of the variables in $RF-\{r\}$ on the value of r is only a dependence on the equivalence class A_i of the value of r. Then, an abstract variable $\hat{r}$ may be constructed whose type set $TY(\hat{r})$ is in a one-to-one correspondence with these equivalence classes. This correspondence determines a function $f : TY(r) \rightarrow TY(\hat{r})$ that maps every element of $TY(r)$ to the element of $TY(\hat{r})$ corresponding to its equivalence class. The function f provides a way to assign a value to the variable $\hat{r}$ in any state s, i.e., $s(\hat{r}) = f(s(r))$. The abstract variable $\hat{r}$ can then be treated as a new state variable, the detailed variable r can be eliminated, and $\hat{r}$ can be used in place of r.

The state variable set RF_A of Σ_A is $(RF - \{r\}) \cup \{\hat{r}\}$. The function f clearly induces an equivalence relation on the states of Σ; the equivalence classes correspond to the states of Σ_A. The initial states and transitions in Σ_A are defined to be those induced by the initial states and transitions in Σ, thus guaranteeing that Σ_A is an abstraction of Σ. The abstraction map α for Method 3 maps every state s of Σ to the state of Σ_A corresponding to its equivalence class induced by f, i.e., to the unique abstract state s_A of Σ_A such that $s_A(\tilde{r}) = s(\tilde{r})$ for all $\tilde{r} \in RF - \{r\}$ and $s_A(\hat{r}) = f(s(r))$. The equivalence relation $\equiv_\alpha$ is the same as that induced by f. The formula q_A is obtained by deriving a formula equivalent to q that uses $\hat{r}$ in place of r. The guards and the assignments in the SCR specification for Σ_A are obtained similarly.

Given that, by construction, q is constant on every equivalence class of $\equiv_\alpha$ and α is a machine homomorphism, it follows that this abstraction method is sound [3]. As with Method 2, a sufficient condition for completeness is that any state s in an equivalence class 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 often satisfied for reasons analogous to those given above for Method 2.

4.5.1 Example

To illustrate Method 3, we show how equivalence classes are determined for **tSELECTED_TRANS**, a real-valued internal variable in the SRS for WCP. One Boolean variable that depends on **tSELECTED_TRANS** is the internal variable **tPRESSURIZING_LATCH**. In the dependency graph shown in Fig. 3, nodes representing the real-valued variable **tSELECTED_TRANS** and the Boolean variable **tPRESSURIZING_LATCH** appear in the bottom right-hand corner. The dependency on **tSELECTED_TRANS** is completely described by the following fragment from the conditional assignment for **tPRESSURIZING_LATCH**:

```
if
    □   … tSELECTED_TRANS ≥ 14.8 …
        → tPRESSURIZING_LATCH := …
    □   … tSELECTED_TRANS ≤ 9.2 …
        → tPRESSURIZING_LATCH := …
    ⋮
    □   …
        → tPRESSURIZING_LATCH := …
fi
```

Suppose the detailed variable **tSELECTED_TRANS** has the type set TY(**tSELECTED_TRANS**) $= [l, u] \subset \mathcal{R}$. ($\mathcal{R}$ denotes the real numbers.) Then, we can use the guard fragments, "**tSELECTED_TRANS** ≥ 14.8" and "**tSELECTED_TRANS** ≤ 9.2," to compute a set of fixed subintervals of $[l, u]$. The first guard partitions $[l, u]$ into the subintervals $[l, 14.8)$ and $[14.8, u]$. The second guard further partitions the first subinterval $[l, 14.8)$ into $[l, 9.2]$ and $(9.2, 14.8)$. Thus, the set of relevant subintervals for the new abstract variable is $\{K_0, K_1, K_2\}$,

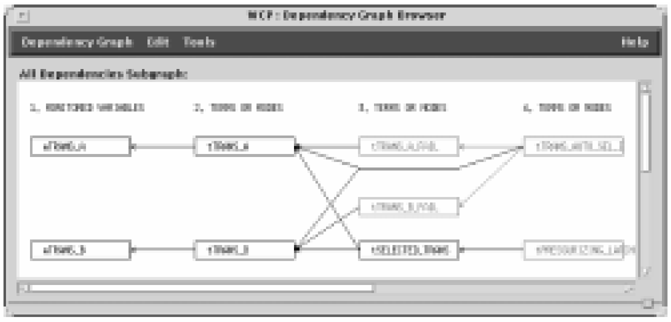


Fig. 3. Real-valued variables and variables that depend on them.

where $K_0 = [l, 9.2]$, $K_1 = (9.2, 14.8)$, and $K_2 = [14.8, u]$. Then, rather than reasoning about a value of **tSELECTED_TRANS** in $[l, u]$, an infinite set of real numbers, we can reason about an abstract variable $\widehat{\texttt{tSELECTED\_TRANS}}$, whose value represents an arbitrary value from one of three subintervals that form a partition of $[l, u]$.

This method replaces the value in $[l, u]$ of the original variable **tSELECTED_TRANS** with the value in $\{K_0, K_1, K_2\}$ of the abstract variable $\widehat{\texttt{tSELECTED\_TRANS}}$. We can define a function f that maps the value of the variable **tSELECTED_TRANS** to its equivalence class K_i :

$$f(s(r)) = \begin{cases} K_0 & \text{if} \quad s(\texttt{tSELECTED\_TRANS}) \in K_0, \\ K_1 & \text{if} \quad s(\texttt{tSELECTED\_TRANS}) \in K_1, \text{ and} \\ K_2 & \text{if} \quad s(\texttt{tSELECTED\_TRANS}) \in K_2. \end{cases}$$

More generally, f determines a partition on the set of states S such that two states $s, \tilde{s} \in S$ are equivalent if f maps the value of **tSELECTED_TRANS** in both s and $\tilde{s}$ to the same K_i and if s and $\tilde{s}$ agree on all other variables.

4.5.2 Uses of the Abstracted Variable

An important issue is how concrete variables subject to abstraction by Method 3 can be used in guards and assignment statements in the concrete specification and how these guards and assignment statements can be transformed into guards and assignment statements in the abstract specification. Variables in the concrete specification to which we apply Method 3 may be used only in linear expressions and inequalities involving constants and no other variables.

In an assignment statement, a concrete variable y may be used in two possible ways:

(1) in a copy operation (e.g., $x := y$)
(2) in a comparison with a constant (e.g., $x := y \geq 14.8$)

In (1), the concrete variable x must also be subject to abstraction, and the type of its abstract version $\hat{x}$ must be compatible with the type of the abstract version $\hat{y}$ of y; that is, $TY(x) = TY(y)$[4] and the partition of $TY(y)$ that yields $TY(\hat{y})$ must be a refinement of the partition of $TY(x)$ that yields

4. The condition $TY(x) = TY(y)$, which holds in all cases to which we apply Method 3 in the WCP, can be relaxed: $TY(x)$ and $TY(y)$ must simply have a common supertype. Of course, this generalization imposes more complex conditions on the relationship among partitions of $TY(x)$ and $TY(y)$ and the guards on the copy operations.

$TY(\hat{x})$. Given that the elements of $TY(\hat{y})$ and $TY(\hat{x})$ are subsets of $TY(y) = TY(x)$ and given the preceding conditions, there is a natural mapping *abs*: $TY(\hat{y}) \rightarrow TY(\hat{x})$ from each element in $TY(\hat{y})$ to the (possibly coarser) element of $TY(\hat{x})$ that contains it. Then, the assignments in the abstract specification corresponding to the examples in (1) and (2) become

$(\hat{1})\ \hat{x} := abs(\hat{y})$

$(\hat{2})\ x := \hat{y} \geq \widehat{14.8}$

In $(\hat{2})$, $\widehat{14.8}$ is the element of $TY(\hat{y})$ that contains the element 14.8 of $TY(y)$. Note that, in this example, we retain the comparison "≥" in $(\hat{2})$. This is valid provided $TY(y)$ is an ordered set, such as the real numbers or the integers, and $TY(\hat{y})$ is a partition of $TY(y)$ into intervals with the ordering inherited from that of $TY(y)$. All type abstractions using Method 3 in the WCP example satisfy these conditions.

Comparisons in assignments, such as the example in (2), can always be transformed into comparisons in guards. For example,

```
if
    □  $true \rightarrow x := y \geq 14.8$
fi
```

becomes

```
if
    □  $y \geq 14.8 \rightarrow x := true$
    □  $NOT(y \geq 14.8) \rightarrow x := false$
fi
```

If this transformation is not performed, then all comparisons in assignments of form (2) must be taken into account (along with the comparisons of a variable with a constant appearing in guards) in determining how to partition the type of that variable to obtain the appropriate type abstraction. Assignments of form (1) also have an effect on the computation of type abstractions: in particular, the type of variable being copied inherits all subdivisions in the partition of the type of the variable being copied to. This inheritance must thus be considered in forming the partition into subintervals for the type of the variable that is copied. Examples of both forms (1) and (2) occur in the assignments in the WCP specification and are discussed in Section 5.2.

In guards, concrete variables subject to abstraction can appear only in comparisons with a constant. The effect of comparisons in guards on the computation of type abstractions has already been discussed in Section 4.5.1. Guards involving abstract variables are transformed in the abstract specification in analogy with the transformation $(2) \rightarrow (\hat{2})$.

5 Model Checking Using Abstraction

This section shows how we make model checking feasible by using two of the abstraction methods, Methods 1 and 3, to construct a reduced WCP specification sufficient for model checking Property 1 (defined in Section 3.6), which we refer to below as the property *q*. First, Method 1 is applied automatically to remove variables irrelevant to the validity of *q*. Then, Method 3 is applied five times in succession to eliminate the five real-valued variables in the reduced specification produced by Method 1. Although, in applying Method 3, we manually replaced each real-valued variable with an abstract variable, the procedure we describe below for applying Method 3 is clearly automatable. The section concludes by describing how model checking the reduced specification with Spin exposed a violation of *q*, how the counterexample produced by Spin was translated into a counterexample in the full specification, and how running the counterexample through the SCR simulator validated the property violation.

5.1 Applying Method 1

To analyze the SCR specification of the WCP for *q*, we first apply Method 1 to form the set $O \subseteq$ RF of variables occurring in formula *q* in either the old state or the new state, namely,

$$O = \{\texttt{cVENT\_SOLENOID}, \texttt{mTRANS\_A}, \texttt{mTRANS\_B}\},$$

and the set O^*, the reflexive and transitive closure of O under the direct dependency relation of the SCR specification for WCP. We refer to the state machine defined by the SCR specification as $\Sigma = (S, \Theta, \rho)$.

The user can invoke the DGB to automatically compute O^*. To do so, the user first displays the complete dependency graph of the WCP (see Fig. 1), identifies the set O of variables in the formula *q* by clicking on the corresponding entry for *q* in the Assertion Dictionary, and then requests the DGB to select the variables on which the variables in O depend. The DGB responds by displaying the graph shown in Fig. 4, which is the subgraph of the graph in Fig. 1 that is relevant to the property *q*. The remaining 55 nodes in this subgraph represent the members of O^*. Applying this abstraction method dramatically reduces the state space by removing more than 76 percent of the variables in the original SCR specification. Using the DGB, the user can save the 55 variables, along with their declarations and tables, in a new reduced SCR specification. This specification defines an abstract SCR machine $\Sigma_A = (S_A, \Theta_A, \rho_A)$ that one can use to reason about the property *q*.

5.2 Applying Method 3

A logical next step is to replace real-valued variables in the SCR specification of Σ_A with finite-valued variables using Method 3. To do so, we first identify the set of real-valued variables in the reduced specification. This set contains **mTRANS_A** and **mTRANS_B**, which represent the two transducers, and three other real-valued terms, **tTRANS_A**, **tTRANS_B**, and **tSELECTED_TRANS**. Then, we identify all variables of finite type that directly depend in either the current state or the next state on any of the real-valued variables: these finite-valued variables are **tTRANS_A_FAIL**, **tTRANS_B_FAIL**, **tPRESSURIZING_LATCH**, and **tTRANS_AUTO_SEL_B**. The graph in Fig. 3 shows all nine variables and their dependencies. Boxes drawn with solid lines distinguish the five real-valued variables from the four finite-valued variables.

Next, we apply Method 3 three times in succession to replace the three real-valued variables **tSELECTED_TRANS**, **tTRANS_B**, and **mTRANS_B** (represented by the three leftmost nodes in the bottom row of Fig. 3). The approach is to elimi-

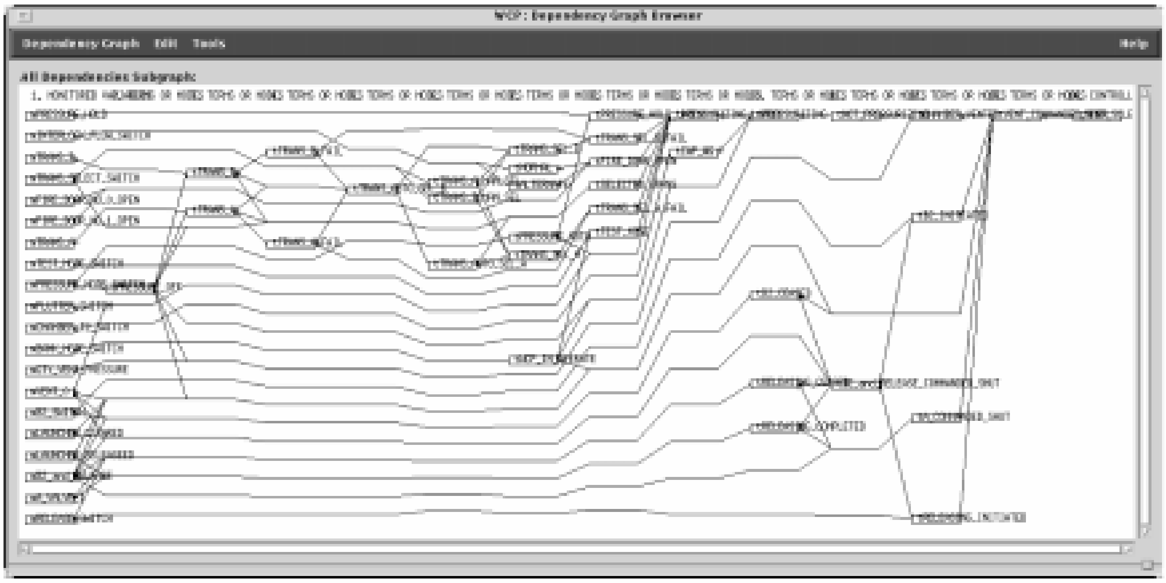


Fig. 4. Dependency graph for reduced SCR specification.

nate the real-valued variables from right to left, replacing each in turn with a finite-valued variable. The remaining two real-valued variables, **tTRANS_A** and **mTRANS_A**, may be replaced by applying Method 3 in an analogous manner.

Section 4.5.1 describes how two guard fragments that refer to the real-valued variable **tSELECTED_TRANS** were used to obtain the abstract variable $\widehat{\mathtt{tSELECTED\_TRANS}}$ with type set $TY(\widehat{\mathtt{tSELECTED\_TRANS}}) = \{K_0, K_1, K_2\}$. Clearly, $\widehat{\mathtt{tSELECTED\_TRANS}}$ is a finite-valued variable that can be substituted for the real-valued variable **tSELECTED_TRANS** in reasoning about the property *q*.

To replace the real-valued variable **tTRANS_B** with a finite-valued variable, we consider three finite-valued variables (see Fig. 3) which depend on **tTRANS_B**: **tTRANS_B_FAIL**, $\widehat{\mathtt{tSELECTED\_TRANS}}$, and **tTRANS_AUTO_SEL_B**. Just as the dependency of the term **tPRESSURIZING_LATCH** on **tSELECTED_TRANS** was used to compute the type of $\widehat{\mathtt{tSELECTED\_TRANS}}$, the dependency of the first variable **tTRANS_B_FAIL** on the real-valued variable **tTRANS_B** is used to compute an abstract replacement, $\widehat{\mathtt{tTRANS\_B}}$, for **tTRANS_B**. The type set of the second variable, $\widehat{\mathtt{tSELECTED\_TRANS}}$, is used to refine the type set of the abstract variable $\widehat{\mathtt{tTRANS\_B}}$. The third variable, the Boolean variable **tTRANS_AUTO_SEL_B**, must be handled differently because its value depends on the *difference* between **tTRANS_A** and **tTRANS_B**. Section 5.3 describes how we handle this special case which involves arithmetic. To replace the real-valued variable **mTRANS_B** with a finite-valued variable, we use the newly constructed finite-valued variable $\widehat{\mathtt{tTRANS\_B}}$ and the intervals defined by the property *q* to construct the abstract replacement $\widehat{\mathtt{mTRANS\_B}}$.

The first variable on which **tTRANS_B** depends, **tTRANS_B_FAIL**, refers to **tTRANS_B** in two assignment statements, each involving a comparison with a constant. The two comparisons which refer to **tTRANS_B** are "**tTRANS_B** > 21.0" and "**tTRANS_B** < 1.8." These comparisons determine a partition of the interval $[l, u]$ into the subintervals $[l, 1.8)$, $[1.8, 21.0]$, and $(21.0, u]$.

The second variable **tSELECTED_TRANS** depends on **tTRANS_B** via a copy operation. Thus, we must also apply the subdivisions in its abstract replacement, $\widehat{\mathtt{tSELECTED\_TRANS}}$, to the partition of **tTRANS_B**. Hence the abstract replacement variable $\widehat{\mathtt{tTRANS\_B}}$ has five relevant subintervals: $[l, 1.8)$, $[1.8, 9.2]$, $(9.2, 14.8)$, $[14.8, 21.0]$, and $(21.0, u]$.

Finally, the real-valued variable **tTRANS_B** is defined in terms of the real-valued monitored variable **mTRANS_B** (see Fig. 3) using a copy operation. Constructing the abstract replacement variable $\widehat{\mathtt{mTRANS\_B}}$ for **mTRANS_B** requires two steps. To form the type set of $\widehat{\mathtt{mTRANS\_B}}$, we first apply the five subintervals in the type set of the abstract variable $\widehat{\mathtt{tTRANS\_B}}$. Next, we apply the subdivisions defined by the two constants that appear in the property *q*, **kMinTRANS** = 7.7 and **kMaxTRANS** = 15.3. Including these subdivisions produces the following seven subintervals for the abstract replacement variable $\widehat{\mathtt{mTRANS\_B}}$: $I_0 = [l, 1.8)$, $I_1 = [1.8, 7.7]$, $I_2 = (7.7, 9.2]$, $I_3 = (9.2, 14.8)$, $I_4 = [14.8, 15.3)$, $I_5 = [15.3, 21.0]$, and $I_6 = (21.0, u]$.

5.3 A Complication: Arithmetic

Replacement of a real-valued variable with an enumerated type produces a sound and, under certain mild restrictions, complete abstraction as long as only simple comparisons are made. Such comparisons are of the form $x \circ k$, where x is a variable, $\circ$ is a relational operator, and k is a constant. The situation becomes more complex when arithmetic is necessary. Because the Boolean variable **tTRANS_AUTO_SEL_B** depends on the *difference* between **tTRANS_A** and **tTRANS_B**, our abstract model does not have the desired property that all variables in RF – {**tTRANS_B**} which depend on **tTRANS_B** depend only on $\widehat{\texttt{tTRANS\_B}}$. To handle this problem, we observe that it is sound to simply allow the value of **tTRANS_AUTO_SEL_B** to be computed nondeterministically as either *true* or *false* in the abstraction. This can be refined by restricting the nondeterministic choice of the value of **tTRANS_AUTO_SEL_B** to its possible values when **tTRANS_A** $\in$ $\widehat{\texttt{tTRANS\_A}}$ and **tTRANS_B** $\in$ $\widehat{\texttt{tTRANS\_B}}$, and further refined by also taking into account any conditions depending upon other variables. Because **tTRANS_AUTO_SEL_B** is Boolean, introduction of some nondeterminism into its value does not significantly increase the size of the state space. Abstractions using this technique will always be sound but may not be complete. In such situations, it is necessary to validate any counterexample using our simulator.

The appropriate point in the abstraction process at which to apply this "nondeterministic variable" technique is after applying Method 1, since Method 1 can eliminate some of the variables whose definitions involve arithmetic (which happened in the WCP example). One question concerns the likelihood that, in such situations, a counterexample to an abstract property in an abstract specification will correspond to a counterexample in the concrete specification. In the WCP example, inspection of the relevant tables suggests that a change in **tTRANS_AUTO_SEL_B** affects the truth of the property q in only a few cases. Therefore, in the WCP example, it is highly likely that, if an abstract counterexample is found, a corresponding concrete counterexample can be found. In fact, we found abstract counterexamples for the WCP example using every version of the nondeterministic variable technique described above, and in addition, found an abstract counterexample for an unsound abstraction in which the table for **tTRANS_AUTO_SEL_B** was translated into a qualitatively similar deterministic version in the abstraction. In all cases, the abstract counterexamples proved to correspond to concrete ones.

5.4 Constructing the Abstract Property q_A

Because the original property q refers to the real-valued variables **mTRANS_A** and **mTRANS_B**, it is necessary to construct an abstract version of the property q that refers to the abstract variables in the reduced specification rather than the real-valued variables **mTRANS_A** and **mTRANS_B** in the original specification. Doing so is straightforward: The abstract property, which we call q_A, can be represented as

$$@\mathrm{T}(\texttt{cVENT\_SOLENOID}) \Rightarrow (I_1 < a' \wedge a' < I_5) \vee (I_1 < b' \wedge b' < I_5),$$

where $a = \widehat{\texttt{mTRANS\_A}}$ and $b = \widehat{\texttt{mTRANS\_B}}$ and, in the notation of $(\hat{2})$ in Section 4.5.2, $I_1 = [1.8, 7.7] = \widehat{7.7} = \widehat{kMinTrans}$ and $I_5 = [15.3, 21.0] = \widehat{15.3} = \widehat{kMaxTrans}$.

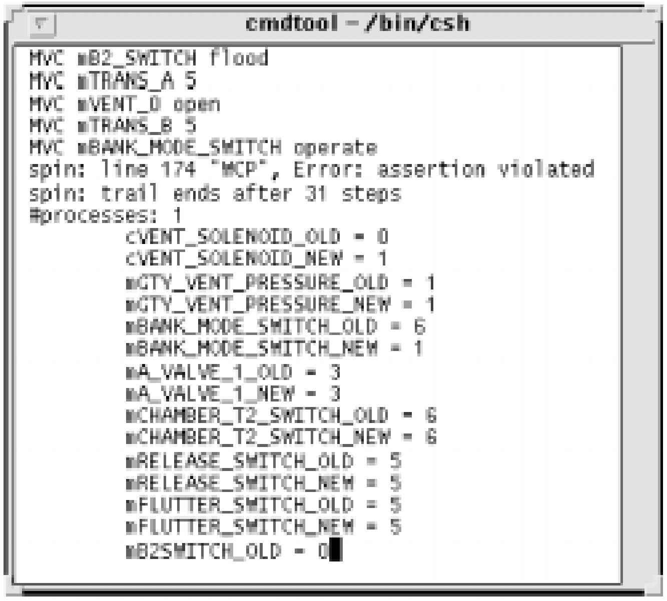
5.5 Model Checking with Spin

By applying Methods 1 and 3 as described above, we obtain a sound abstraction of the SCR specification of WCP for reasoning about property q. To analyze the reduced SCR specification, we invoked the explicit state model checker Spin from within our toolset, which automatically translated the reduced specification into Promela, the language of Spin. For complete details and fully worked-out examples of how SCR specifications can be translated into either Promela, the language of Spin, or into the language of the SMV model checker, see [9], [11]. Because our tools support a very limited property language (e.g., no ordered enumerated types, no set membership), we represented the ordered enumerated types as subranges of integers. Hence, the representation of q_A that we actually checked was

$$@\mathrm{T}(\texttt{cVENT\_SOLENOID}) \Rightarrow (1 < a' \wedge a' < 5) \vee (1 < b' \wedge b' < 5),$$

where a' and b' are integers in {0, 1, …, 6} rather than elements of an enumerated type.

Running Spin exposed a pair of reachable states that violate property q and produced a counterexample, i.e., an execution sequence that starts in a valid initial state and, through a sequence of transitions permitted by the next-state relation of the abstract machine, ends in the pair of states that violate the property. Fig. 5 shows both the execution sequence consisting of five input events that lead to the bad pair of states (the top five lines) and the first parts of the final states in the sequence (the final 15 lines). Spin required less than 30 seconds to find the violation.



```
MVC mB2_SWITCH flood
MVC mTRANS_A 5
MVC mVENT_0 open
MVC mTRANS_B 5
MVC mBANK_MODE_SWITCH operate
spin: line 174 "WCP", Error: assertion violated
spin: trail ends after 31 steps
#processes: 1
        cVENT_SOLENOID_OLD = 0
        cVENT_SOLENOID_NEW = 1
        mGTY_VENT_PRESSURE_OLD = 1
        mGTY_VENT_PRESSURE_NEW = 1
        mBANK_MODE_SWITCH_OLD = 6
        mBANK_MODE_SWITCH_NEW = 1
        mA_VALVE_1_OLD = 3
        mA_VALVE_1_NEW = 3
        mCHAMBER_T2_SWITCH_OLD = 6
        mCHAMBER_T2_SWITCH_NEW = 6
        mRELEASE_SWITCH_OLD = 5
        mRELEASE_SWITCH_NEW = 5
        mFLUTTER_SWITCH_OLD = 5
        mFLUTTER_SWITCH_NEW = 5
        mB2SWITCH_OLD = 0
```

Fig. 5. *Results of running Spin.*

5.6 Using the Simulator for Validation

Using our methods, a counterexample obtained from model checking, i.e., a sequence of input events, will be in terms of an abstract state machine. We have a standard method for translating each input event in the abstract machine to a sequence of one or more input events in the concrete machine. In the case of an input event involving an abstract monitored variable constructed with Method 3, we simply choose a member of the corresponding equivalence class to obtain a corresponding monitored variable in the concrete execution sequence. In the case of an input variable involving an abstract monitored variable constructed with Method 2, we may need to translate the abstract input event into a sequence of two or more input events in the concrete machine.

If the derived sequence of input events is executable in the concrete model, as it is in the WCP example, executing the input sequence will produce a sequence of concrete states. This sequence of concrete states can be mapped to a sequence of abstract states that typically corresponds to the abstract counterexample, perhaps with some stuttering added. (By *stuttering*, we mean a sequence of two or more concrete states that correspond to one abstract state.)

In the general case, the sequence of concrete events that we derive may not be directly executable, because some of the translated input events might not be enabled according to the next-state relation for the corresponding input variable. A typical example occurs when some numerical input quantity takes too large a leap in one of the derived events. However, in such cases, this large leap can usually be achieved in several legal small steps, and inserting the extra small steps gives us a good candidate for a concrete counterexample. When our sufficient conditions for completeness are satisfied, we can automatically construct a sequence of small concrete steps that can be mapped to a sequence of abstract steps with stuttering. Since by construction (using the notation of Section 4) $q(s) \Rightarrow q_A(s_A)$ for every concrete state s, we can be confident that our derived sequence of small steps is indeed a concrete counterexample.

To evaluate the validity of q in the original SCR specification, we manually translated the sequence of input events obtained from Spin (see Fig. 5), which is in terms of the abstract state machine, to a corresponding sequence of input events in the original state machine. In the WCP example, the translation is trivial. We simply replace each value of an abstract variable with a real number in the corresponding interval. Steps 2 and 4 are the two steps in the scenario where this translation is required. In each case, we selected the value 18.0, which lies in the interval I_5 = [15.3, 21.0]. (The “5” in the abstract scenario refers to interval I_5.) Each of the remaining three steps involve a monitored variable that is the same in both the abstract and the concrete machines. Hence, no translation of these steps is necessary.

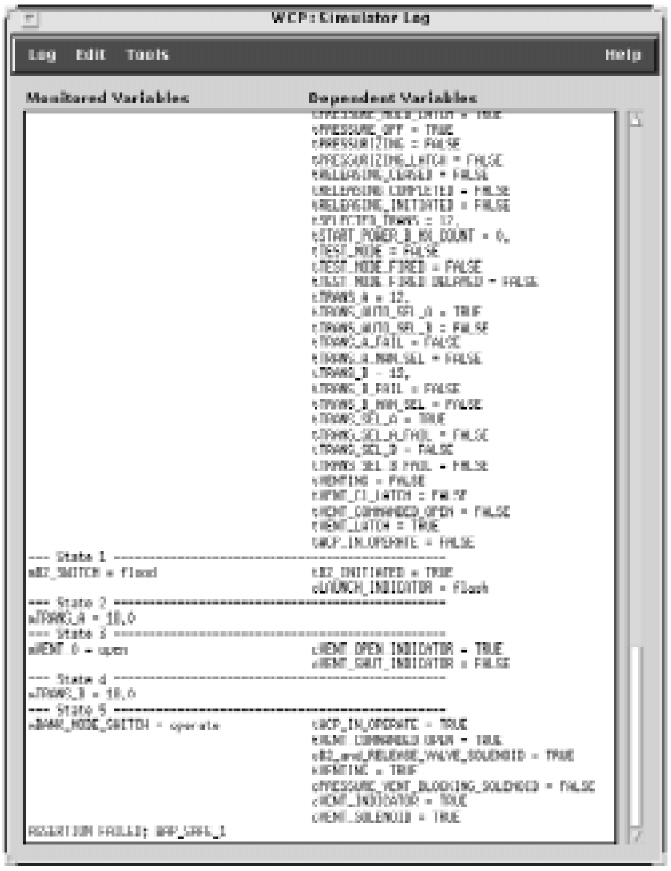
Next, we ran this concrete scenario through our simulator. To check for the property of interest q, we added the property to the Assertion Dictionary. Running the scenario through our simulator leads to an assertion failure (see the bottom of Fig. 6). Clicking on the line reporting the assertion failure highlights the first property in the Assertion Dictionary shown in Fig. 2, thus validating that the property violation detected by model checking corresponds to a property violation in the original specification. Safety engineers familiar with the WCP have confirmed that this is a true safety violation.

When used in conjunction with model checking, simulation may be used either to *demonstrate* a property violation or to *evaluate* a candidate violation. In the case of abstractions that are complete, simulation may be used to *demonstrate* a counterexample in the original specification that corresponds to a counterexample produced by a model checker in analyzing the reduced specification. In the case of abstractions for which completeness is not guaranteed, simulation is useful for *validation*, i.e., for testing whether an error detected by model checking is an actual error in the original specification. In many cases, such as the case of WCP, the reduced specification is not guaranteed to be a complete abstraction. Hence, some method is needed to evaluate the counterexample in the original specification to ensure that the property violation is not spurious. Moreover, when any part of the abstraction process and scenario translation is done by hand, errors can be introduced, either in developing the reduced model from the original specification or in translating a counterexample produced by model checking into a scenario in the original specification. Clearly, when manual processes are used, simulation is effective for ensuring that a suspected violation is a true violation in the original specification.

5.7 More About the Safety Violation

Applying Spin to the reduced specification described above exposed numerous violations of property q. The probability of some of these violations occurring in practice is low; for example, some occur only when the vent valve sensor fails (i.e., in step 3 in Fig. 5, the sensor reports the vent valve is open when it is closed) *and* both transducers fail (i.e., both **mTRANS_A** and **mTRANS_B** have values in interval I_0). Other scenarios which produce unsafe behavior are more likely. For example, both transducers can report a pressure reading in a hazardous region (either I_1 or I_5) at the same time that the vent valve sensor fails. In all of these cases, the WCP specification allows the system to open the valve—i.e., to set the value of **cVENT_SOLENOID** to *true*—even though doing so is very dangerous.

The scenario in Fig. 6 contains two seemingly unrealistic transitions, i.e., each transducer reports a pressure reading of 12.0 psi in one state and a reading of 18.0 psi later on, a difference of 6 psi. Note that this scenario is a correct abstraction of the possible behavior of a continuous system. A change in pressure of 6 psi could take place over minutes and in a series of steps that do not affect the abstract state. It is also worth noting that we selected some values somewhat arbitrarily (i.e., 12.0 as the initial value, 18.0 as the representative concrete value of the abstract value I_5). We could easily pick more realistic concrete values for scenarios, e.g., smaller changes in the transducer values from one step to the next. What is important is that a wide range of concrete values (realistic or unrealistic) violate the safety assertion.



```
Monitored Variables                    Dependent Variables
                                       cPRESSURE_HOLD_LATCH = TRUE
                                       cPRESSURE_OFF = TRUE
                                       cPRESSURIZING = FALSE
                                       cPRESSURIZING_LATCH = FALSE
                                       cRELEASING_CLOSED = FALSE
                                       cRELEASING_COMPLETED = FALSE
                                       cRELEASING_INITIATED = FALSE
                                       cSELECTED_TRANS = 12,
                                       cSTART_POWER_B_HX_COUNT = 0,
                                       cTEST_MODE = FALSE
                                       cTEST_MODE_FIRED = FALSE
                                       cTEST_MODE_FIRED_DELAYED = FALSE
                                       cTRANS_A = 12,
                                       cTRANS_AUTO_SEL_A = TRUE
                                       cTRANS_AUTO_SEL_B = FALSE
                                       cTRANS_A_FAIL = FALSE
                                       cTRANS_A_MAN_SEL = FALSE
                                       cTRANS_B = 15,
                                       cTRANS_B_FAIL = FALSE
                                       cTRANS_B_MAN_SEL = FALSE
                                       cTRANS_SEL_A = TRUE
                                       cTRANS_SEL_A_FAIL = FALSE
                                       cTRANS_SEL_B = FALSE
                                       cTRANS_SEL_B_FAIL = FALSE
                                       cVENTING = FALSE
                                       cVENT_CI_LATCH = FALSE
                                       cVENT_COMMANDED_OPEN = FALSE
                                       cVENT_LATCH = TRUE
                                       cWCP_IN_OPERATE = FALSE
--- State 1 ------------------------------------------------------
mBZ_SWITCH = flood                     cBZ_INITIATED = TRUE
                                       cLAUNCH_INDICATOR = flash
=== State 2 ======================================================
mTRANS_A = 10,0
=== State 3 ======================================================
mVENT_0 = open                         cVENT_OPEN_INDICATOR = TRUE
                                       cVENT_SHUT_INDICATOR = FALSE
--- State 4 ------------------------------------------------------
mTRANS_B = 10,0
--- State 5 ------------------------------------------------------
mBAAR_MODE_SWITCH = operate            cWCP_IN_OPERATE = TRUE
                                       cVENT_COMMANDED_OPEN = TRUE
                                       cBZ_and_RELEASE_VALVE_SOLENOID = TRUE
                                       cVENTING = TRUE
                                       cPRESSURE_VENT_BLOCKING_SOLENOID = FALSE
                                       cVENT_INDICATOR = TRUE
                                       cVENT_SOLENOID = TRUE
ASSERTION FAILED: WAP_SAFE_1
```

Fig. 6. Simulator log showing an assertion failure.

6 DISCUSSION

This section discusses five issues that arose in our study—the role of redundancy in specifying requirements, the criteria for selecting an analysis method, contexts in which automatic methods can support the use of abstraction in model checking, how automatic abstraction methods combined with other methods can reduce the state explosion problem, and how our approach to abstraction can be extended to methods other than SCR.

6.1 Role of Redundancy in Requirements Specification

In its original form, the A-7 requirements specification [38] was designed to minimize redundancy. For example, the required values of each variable under all possible conditions were defined in one unique place in the specification. By minimizing redundancy, the designers hoped to facilitate learning about a given aspect of the specification, to facilitate changing the specification, and, more generally, to limit opportunities for inconsistency in the specification. As shown in this paper, some carefully designed redundancy in requirement specifications can be valuable. Above, we showed that analyzing two redundant specifications of the required behavior, one operational and the other property-based, can expose inconsistencies, which in turn can expose errors. Like us, Atlee and her colleagues have used model checking to detect inconsistencies in SCR specifications. Using this approach, they have exposed errors in the A-7 requirements specification [62], in Kirby's specification of automobile cruise control [6], and in other specifications. In each case, they analyzed two redundant specifications of the required behavior—one property-based, the other operational.

6.2 Selecting an Analysis Method

Among the available techniques for analyzing the consistency of a safety property and an operational specification are human inspection, model checking, and mechanical theorem proving. Software developers choose a particular technique based on its relative effectiveness and its relative costs. The cost of using a technique depends on two factors: the amount of human effort needed to apply the technique and any specialized expertise that is needed (e.g., theorem proving skills).

For detecting certain classes of errors, tools are far more cost-effective than human inspection. For example, the SCR consistency checker needed only a few minutes to automatically detect numerous errors (17 missing cases and 57 instances of nondeterminism) in a revised version of the A-7 requirements document [32]. Systematic inspection of the document by two independent review teams had overlooked these errors. The existence of so many overlooked errors was quite surprising given that the tabular notation used in the document was designed to make such errors obvious—most of the information needed to detect missing cases and nondeterminism was located in a single table. Miller reports similar results in the development of a specification for a flight guidance system: the SCR tools uncovered many errors overlooked by inspection [58]. Inspection is even less likely to detect violations of safety properties, since analyzing safety properties usually involves dozens of variable definitions, distributed throughout the specification. Moreover, checking for safety violations is much more complex than checking for missing cases or nondeterminism, since reachability, rather than simply function definitions, must be analyzed.

Applying model checking prior to theorem proving is an effective way to use the two technologies. Because model checking does not require the mathematical sophistication and human effort that theorem proving currently entails, model checking is relatively inexpensive. Moreover, where model checkers are good for detecting property violations, the treatment of invalid properties by theorem provers is often problematic. Hence, even if one's primary goal is to use theorem proving, a good strategy is to weed out simple errors with model checking *before* one invests the time and effort usually required for theorem proving [29].

Theorem proving is usually most effective after model checking. Although model checking may be used to *verify* properties of practical software specifications (i.e., to demonstrate that every possible scenario satisfies the properties of interest), the enormous size of most of these specifications makes model checking more effective for *detecting errors*. When model checking fails to reveal an error in such specifications or produces many spurious counterexamples because the abstraction used was not complete, the user may successfully use theorem proving to establish the property. We have, in fact, done this for a small SCR specification, using the proof techniques described in [4]. Moreover, when model checking fails to expose a violation of a property, theorem proving, strengthened with appropriate auxiliary techniques, may do so. As in model checking, abstraction is useful in theorem proving. However, in theorem proving, one can reason about generic values. As a result, abstractions that are not complete can usually be avoided. Thus, analysis by theorem proving is usually exact.

6.3 Role of Automation in Supporting Abstraction

Above, we show how standard abstraction methods can be used to reduce the state space of the model analyzed by a model checker. We are developing automated methods that support the use of our standard abstraction methods in five different contexts:

- Based on the property of interest, identify variables to be eliminated and variables whose type sets can be more abstract
- Once the relevant variables and the nature of the needed abstraction have been determined, construct the specification of the abstract state machine
- Translate the original property q to an abstract property q_A
- In the case of Methods 2 and 3, determine whether sufficient conditions for completeness are satisfied
- Translate the counterexample obtained by model checking (if any) into a corresponding scenario in the original state machine model

6.4 Combining Automatic Abstraction Methods with Other Methods

After automatic abstraction methods, such as those described above, are used to produce a reduced state machine model, other abstraction methods, such as those described by Clarke et al. [15] and Graf et al. [27], [28], which use mathematical reasoning, perhaps supported by automated theorem proving and the automated generation of invariants [46], may further reduce the state space. Moreover, once the specification of the state machine model has been reduced, various methods for efficient analysis of different representations, such as BDDs, or efficient decision procedures, such as techniques for deciding Presburger arithmetic formulae and the congruence closure algorithm for handling uninterpreted function symbols, may be applied to the reduced model [8]. Eventually, a combination of these methods may allow us to verify large software specifications, a task that currently is infeasible for many practical software specifications.

6.5 Extending Automated Abstraction Beyond SCR

Two major features of SCR specifications facilitate the support of automated abstraction. Largely the result of good engineering practice, such features should not be difficult to support in notations and tools associated with other software development methods. In particular, software specifications represented in other notations that have these features should easily support automated abstraction methods, such as those described above.

One extremely important feature of high-quality specifications is well-formedness; the specifications should be free of type errors, circular definitions, missing cases, and unwanted nondeterminism. Tools, such as our consistency checker, should be available that automatically detect and report such problems to the user so they may be corrected.

A second important feature of high-quality specifications lies in their structure [34], [35]: All information about each variable should be localized. For example, in an SCR specification, all information about each variable is in exactly two places: a table that describes the function defining the variable's value (or, in the case of a monitored variable, a dictionary entry that defines the associated relation) and a dictionary entry that provides the variable's static information (type, initial value, etc.). This facilitates the removal or replacement of the variable when the abstract machine is constructed. For example, removing a dependent variable from an SCR specification is easy—the tool simply deletes the table defining the variable and the variable's dictionary entry. In addition, the dependency graph shows the impact of removing a variable.

7 Related Work

Below, we describe other work in which model checking has been applied to requirements specifications. We also compare our use of abstraction in model checking with other approaches. While our objective is to develop mathematically sound abstraction methods that can be applied automatically to requirements specifications, the major objective of other work on abstraction has been to formulate a theory of abstraction. The most complete treatment is the very general theory of abstraction relations formulated by Loiseaux et al. [51] and extended with some modifications by Dams et al. [19]. Our approach is a special case of the approach in [51] in which the abstraction relation is a map.

7.1 Model Checking Requirements Specifications

An early application of model checking to SCR requirements specifications was reported in 1993 by Atlee and Gannon, who used the model checker MCB [16] to analyze properties of individual mode transition tables taken from SCR specifications [6]. 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 [6], [5], [62]. While the techniques of Atlee et al. are designed to analyze properties of mode transition tables with Boolean input variables, the approach we describe in [9], [11], appropriately extended, can be used to analyze properties of a complete SCR specification: The properties analyzed can contain any variable in the specification, and variables can range over varied domains, such as integer subranges, enumerated values, and infinite subranges of the real numbers.

In [13], a revision of the 1996 paper by Anderson et al. [2], Chan et al. describe the use of SMV to analyze a component of the Traffic Alert and Collision Avoidance System (TCAS II) requirements specification expressed in the Requirements State Machine Language (RSML) notation [30]. 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 [13, p. 511]. Another 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 (or real) 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

Work on abstraction, both in the context of model checking and on the related topic of error-preserving abstractions [66], ultimately derives from the seminal work in 1977 of the Cousots on abstract interpretation [18]. Our work on abstraction in model checking is most closely related to later work on abstraction, largely theoretical, by Clarke et al. [15], Loiseaux et al. [51], Graf and Loiseaux [27], [28], Dams et al. [19], and Kurshan [50]. We note that our first two abstraction methods are related to methods proposed by Kurshan as early as 1987 [49], [17]. Like ours, Kurshan's methods, which he calls *localization reductions* [48], remove parts of the specification irrelevant to the property of interest. Below, we describe five significant aspects of our approach to abstraction that distinguish our approach from other approaches.

First, we focus on invariant 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 μ-calculus, or automata that accept languages with infinite words [50].

Second, the abstractions we apply use variable restriction, which eliminates certain variables, and variable abstraction, which abstracts the data types of certain variables, in specific limited ways that can be automated. Both can be viewed as special cases of the data abstractions introduced by Clarke et al., since variable restriction 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 [51], [28], and [27] all of the abstraction relations are in fact maps.) Although our abstractions are a proper subset of those considered by Clarke et al. [15], we can obtain fairly complex abstractions by performing a sequence of our simple abstractions.

Third, besides restricting attention to simple state and transition invariants, we construct the abstraction based on a single property. By contrast, in [15], [51], [19], and [50], 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, our concept of a "complete" abstraction, though analogous to the "exact" abstractions of Clarke [15] and Kurshan [50] and the "strong preservation" of properties by abstractions described by Loiseaux et al. [51], is less restrictive. Unlike other authors [51], [50], [15], we have established sufficient conditions for completeness that do not require the abstraction mapping to determine a bisimulation. We often can establish completeness for our abstractions using automatic techniques.

Fourth, because we also focus on certain specific abstraction methods, we are able to automate the choice and construction of abstractions, including abstraction of the property. In the work of others, the user must typically propose the abstraction, or at least the abstraction relation, and provide appropriate interpretations of primitive predicates. At least one author, Graf [27], like us, 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 usually automatic and not computationally expensive. As a result, our methods do not require the modification of a BDD description of the automaton as in [15] nor the processing of the state transition graph as in [50]. Rather, building the abstraction and establishing correctness are done at the SCR specification level.

8 Conclusions

This paper showed how we applied our abstraction methods, our model checking approach, and our simulator to expose a safety violation in a contractor-produced specification of a safety-critical system, and how the three abstraction methods described in the paper make the analysis of requirements specifications practical. An important aspect of our approach is that the contractor-produced specification was not developed with the SCR method in mind. That the model underlying the contractor SRS matches the model that underlies the SCR method suggests that the SCR method is relevant to practical software development.

Method 1 as well as the construction of the abstract SCR machine for Method 1 have been implemented and integrated into the SCR toolset. Prototype implementations of special cases of Methods 2 and 3 have also been developed. We are extending our work in several ways:

- We are designing algorithms for Methods 2 and 3 that automatically extract the abstraction Σ_A and the property q_A from the original SCR specification and a given property q. We also are investigating the extent to which we can automatically check that the conditions for completeness are satisfied.
- We are also developing software that will automatically translate any counterexample produced by model checking the abstraction Σ_A into a corresponding scenario in the original specification.

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. Our long-term goal is to combine the power of theorem proving technology with the ease of use of model checking technology. A major problem with current theorem proving technology is that applying the technology requires mathematical sophistication and theorem proving skills. The major problem with model checking is state explosion. Clearly, theorem proving has the potential to dramatically reduce the number of states that a model checker analyzes. Automatic theorem proving methods that can be applied for this purpose are therefore worth developing.

We are also exploring other automated techniques useful for analyzing large, complex requirements specifications. One is automatic invariant generation [46]. Another goal of our current research is to integrate one or more decision procedures into the toolset—we are especially interested in decision procedures that can evaluate logical expressions containing numbers and arithmetic. Currently, the analysis of such specifications is problematic, e.g., for the consistency checker. Finally, we are investigating how to support time and feedback in the SCR model, how to improve support for hierarchy and modularization in SCR specifications, and how to automatically generate correct, efficient source code from SCR specifications. Yet to be explored is how SCR specifications containing time and feedback can be analyzed efficiently using model checking and other analysis techniques.

To date, our requirements model has provided a solid foundation for a suite of analysis tools which can detect errors automatically and which clearly explain the cause of those errors, thereby facilitating error correction. Such an approach should lead to the production of high-quality requirements specifications, which should in turn lead to software that is 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

We are extremely grateful to C. Colket, K. Andree, B. Capoccia, P. Roach, J. Rainone, and D. Sjoblom, who made us aware of the WCP specification, answered our many questions, and provided further information about the WCP. Without their help, the analyses described in the paper could not have been performed. We also appreciate the insights on our abstraction methods of Neil Immerman of the University of Massachusetts, whose detailed comments helped us improve our formal framework. We also gratefully acknowledge the constructive comments of Ralph Jeffords and Steve Sims at NRL and comments from the anonymous referees on drafts of this paper. Finally, we thank Todd Grimm for his substantial contributions to the construction of the SCR toolset; Carolyn Gasarch and Cheryl Sarteschi for implementing the first abstraction method and integrating it into the toolset; and Steve Sims for building prototype implementations of the two other abstraction methods. This research was supported by the Office of Naval Research and the Space and Naval Warfare Command.

REFERENCES

[1] T.A. Alspaugh, S.R. Faulk, K. Heninger Britton, R.A. Parker, D.L. Parnas, and J.E. Shore. “Software Requirements for the A-7E Aircraft,” Technical Report NRL-9194, Naval Research Laboratory, Washington, D.C., 1992.

[2] R.J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J.D. Reese, “Model Checking Large Software Specifications,” *Proc. Fourth ACM SIGSOFT Symp. Foundations of Software Eng.*, Oct. 1996.

[3] M. Archer and C. Heitmeyer, “The Use of Model Checking and Abstraction in Analyzing Requirements Specifications: A Formal Foundation,” Technical Report, Naval Research Laboratory, Washington, D.C., 1998. Draft.

[4] M. Archer, C. Heitmeyer, and S. Sims, “TAME: A PVS Interface to Simplify Proofs for Automata Models,” *Proc. User Interfaces for Theorem Provers*, Eindhoven, Netherlands, Eindhoven Univ. technical report, Eindhoven Univ. of Technology, July 1998.

[5] J.M. Atlee and M.A. Buckley, “A Logic-Model Semantics for SCR Specifications,” *Proc. Int’l Symp. Software Testing and Analysis*, Jan. 1996.

[6] J.M. Atlee and J. Gannon, “State-Based Model Checking of Event-Driven System Requirements,” *IEEE Trans. Software Eng.*, vol. 19, no. 1, pp. 24–40, Jan. 1993.

[7] G. Berry and G. Gonthier, “The Esterel Synchronous Programming Language: Design, Semantics, Implementation,” *Science of Computer Programming*, vol. 19, 1992.

[8] R. Bharadwaj, “A Generalized Validity Checker,” technical report, Naval Research Laboratory, Washington, D.C., 1996.

[9] R. Bharadwaj and C. Heitmeyer, “Model Checking Complete Requirements Specifications Using Abstraction,” Technical Report NRL-7999, Naval Research Laboratory, Washington, D.C., Nov. 1997.

[10] R. Bharadwaj and C. Heitmeyer, “Verifying SCR Requirements Specifications Using State Exploration,” *Proc. First ACM SIGPLAN Workshop Automatic Analysis of Software*, 1997.

[11] R. Bharadwaj and C. Heitmeyer, “Model Checking Complete Requirements Specifications Using Abstraction,” *Automated Software Eng. J.*, vol. 6, no. 1, Jan. 1999.

[12] B.W. Boehm, *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice Hall, 1981.

[13] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese, “Model Checking Large Software Specifications,” *IEEE Trans. Software Eng.*, vol. 24, no. 7, July 1998.

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

[15] E. Clarke, O. Grumberg, and D. Long, “Model Checking and Abstraction,” *Proc., Principles of Programming Languages* (*POPL*), 1994.

[16] E.M. Clarke, E. Emerson, and A. Sistla, “Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications,” *ACM Trans. Program Language and Systems*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[17] E.M. Clarke and R.P. Kurshan, “Computer-Aided Verification,” *IEEE Spectrum*, pp. 61–67, June 1996.

[18] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” *Proc. Symp. Principles of Programming Languages*, 1977.

[19] D. Dams, R. Gerth, and O. Grumberg, “Abstract Interpretation of Reactive Systems,” *ACM Trans. Program Language and Systems*, pp. 111–149, 1997.

[20] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang, "Protocol Verification as a Hardware Design Aid," *Proc. IEEE Int'l Conf. Computer Design*: *VLSI in Computers and Processors*, pp. 522–525, 1992.

[21] S. Easterbrook and J. Callahan, "Formal Methods for Verification and Validation of Partial Specifications: A Case Study," *J. Systems and Software*, 1997.

[22] S. Easterbrook, R. Lutz, R. Covington, Y. Ampo, and D. Hamilton, "Experiences Using Lightweight Formal Methods for Requirements Modeling," *IEEE Trans. Software Eng.*, vol. 24, no. 1, Jan. 1998.

[23] R. Fairley, *Software Eng. Concepts*. New York: McGraw-Hill, 1985.

[24] S.R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr., "The CoRE Method for Real-Time Requirements," *IEEE Software*, vol. 9, no. 5, pp. 22–33, Sept. 1992.

[25] S.R. Faulk, L. Finneran, J. Kirby, Jr., S. Shah, and J. Sutton, "Experience Applying the CoRE Method to the Lockheed C-130J," *Proc. Ninth Ann. Conf. Computer Assurance* (*COMPASS'94*), pp. 3–8, Gaithersburg, Md., June 1994.

[26] P. Godefroid, "Using Partial Orders to Improve Automatic Verification Methods," *Proc. Second Int'l Workshop Computer-Aided Verification*, pp. 176–185, 1990.

[27] S. Graf, "Characterization of a Sequentially Consistent Memory and Verification of a Cache Memory by Abstraction," *Proc. Computer Aided Verification*, 1994.

[28] S. Graf and C. Loiseaux, "A Tool for Symbolic Program Verification and Abstraction," *Proc. Computer Aided Verification*, pp. 71–84, 1993.

[29] K. Havelund and N. Shankar, "Experiments in Theorem Proving and Model Checking for Protocol Verification," *Proc. Formal Methods Europe* (*FME'96*), pp. 662–681, Lecture Notes in Computer Science 1051, Springer-Verlag, Mar. 1996.

[30] M.P.E. Heimdahl and N. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements," *IEEE Trans. Software Eng.*, vol. 22, no. 6, pp. 363–377, June 1996.

[31] C. Heitmeyer, J. Kirby, Jr., and B. Labaw, "Applying the SCR Requirements Method to a Weapons Control Panel: An Experience Report," *Proc. Second ACM Workshop Formal Methods in Software Practice* (*FMSP'98*), 1998.

[32] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology*, vol. 5, no. 3, pp. 231–261, July 1996.

[33] C. Heitmeyer, "On the Need for Practical Formal Methods," *Proc. Formal Techniques in Real-Time and Fault-Tolerant Systems* (*FTRTFT'98*), Lyngby, Denmark, Sept. 1998.

[34] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," *Proc. 10th Ann. Conf. Computer Assurance* (*COMPASS'95*), pp. 109–122, Gaithersburg, Md., June 1995.

[35] C. Heitmeyer, J. Kirby, Jr., and B. Labaw, "Tools for Formal Specification, Verification, and Validation of Requirements," *Proc. 12th Ann. Conf. Computer Assurance* (*COMPASS'97*), Gaithersburg, Md., June 1997.

[36] C. Heitmeyer, J. Kirby, Jr., B. Labaw, and R. Bharadwaj, "SCR*: A Toolset for Specifying and Analyzing Software Requirements," *Proc. Computer-Aided Verification, 10th Ann. Conf. (CAV'98)*, Vancouver, Canada, 1998.

[37] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Tools for Analyzing SCR-Style Requirements Specifications: A Formal Foundation," technical report, Naval Research Laboratory, Washington, D.C., 1998. Draft.

[38] K. Heninger, D.L. Parnas, J.E. Shore, and J.W. Kallander, "Software Requirements for the A-7E Aircraft," Technical Report 3876, Naval Research Laboratory, Washington, D.C., 1978.

[39] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Software Eng.*, vol. 6, no. 1, pp. 2–13, Jan. 1980.

[40] S.D. Hester, D.L. Parnas, and D.F. Utter, "Using Documentation as a Software Design Medium," *Bell System Technical J.*, vol. 60, no. 8, pp. 1941–1977, Oct. 1981.

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

[42] G.J. Holzmann, "The Model Checker SPIN," *IEEE Trans. Software Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[43] D. Jackson, "Requirements and Model Checking," minitutorial, *Third Int'l IEEE Symp. Requirements Eng.*, Jan. 1997.

[44] D. Jackson, S. Jha, and C.A. Damon, "Faster Checking of Software Specifications Using Isomorphs," *Proc., Principles of Programming Languages* (*POPL*), 1994.

[45] F. Jahanian and A.K. Mok, "Modechart: A Specification Language for Real-Time Systems," *IEEE Trans. Software Eng.*, vol. 20, no. 10, pp. 879–889, Oct. 1994.

[46] R. Jeffords and C. Heitmeyer, "Automatic Generation of State Invariants from Requirements Specifications," *Proc. Sixth ACM SIGSOFT Symp. Foundations of Software Eng.*, Nov. 1998.

[47] J. Kirby, Jr., "Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System," Technical Report TR-87-07, Wang Inst. of Graduate Studies, 1987.

[48] R.P. Kurshan, "Formal Verification in a Commercial Setting. *Proc. Design Automation Conf.*, June 1997.

[49] R.P. Kurshan, "Reducibility in Analysis of Coordination," P. Varaiya and A.B. Kurzhanski, eds., *Discrete Event Systems: Models and Applications*, pp. 19–39. New York: Springer-Verlag, 1987.

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

[51] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem, "Property Preserving Abstractions for the Verification of Concurrent Systems," *Formal Methods in System Design*, vol. 6, pp. 1–35, 1995.

[52] R.R. Lutz, "Targeting Safety-Related Errors During Software Requirements Analysis," *Proc. First ACM SIGSOFT Symp. Foundations of Software Eng.*, Los Angeles, Dec. 1993.

[53] R.R. Lutz and H.-Y. Shaw, "Applying the SCR* Requirements Toolset to DS-1 Fault Protection," Technical Report JPL-D15198, Jet Propulsion Laboratory, Pasadena, Calif., Dec. 1997.

[54] D. Mandrioli, A. Morzenti, M. Pezze, P. SanPietro, and S. Silva, "A Petri Net and Logic Approach to the Specification and Analysis of Real-Time Systems," C. Heitmeyer and Dino Mandrioli, eds., *Formal Methods for Real-Time Computing, Trends in Software.* Chichester, England: John Wiley & Sons Ltd, 1996.

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

[56] K.L. McMillan, *Symbolic Model Checkin*g. Englewood Cliffs, N.J.: Kluwer Academic, 1993.

[57] S. Meyer and S. White, "Software Requirements Methodology and Tool Study for A6-E Technology Transfer," technical report, Grumman Aerospace Corp., Bethpage, New York, July 1983.

[58] S. Miller, "Specifying the Mode Logic of a Flight Guidance Ssystem in CoRE and SCR," *Proc. Second ACM Workshop Formal Methods in Software Practice* (*FMSP'98*), 1998.

[59] J. Ostroff, "A Visual Toolset for the Design of Real-Time Discrete-Event Systems," *IEEE Trans. Control Systems Technology*, vol. 5, no. 3, pp. 320–337, May 1997.

[60] D.L. Parnas, G.J.K. Asmis, and J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety*, vol. 32, no. 2, pp. 189–198, Apr.-June 1991.

[61] D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, Oct. 1995.

[62] T. Sreemani and J.M. Atlee, "Feasibility of Model Checking Software Requirements," *Proc. 11th Ann. Conf. Computer Assurance* (*COMPASS'96*), Gaithersburg, Md., June 1996.

[63] J. Sutton, personal communication, Sept.1997.

[64] U.S. General Accounting Office, "Mission Critical Systems: Defense Attempting to Address Major Software Challenges," Technical Report GAO/IMTEC-93-13, U.S. General Accounting Office, Washington, D.C., Dec. 1992.

[65] M. Weiser, "Program Slicing," *IEEE Trans. Software Eng.*, vol. 10, no. 4, pp. 352–357, July 1984.

[66] M. Young, "How to Leave out Details: Error Preserving Abstractions of State-Space Models," *Proc. ACM Conf. Software Testing, Analysis and Verification*, 1988.

Constance Heitmeyer holds MA degrees in mathematics and history from the University of Michigan. She heads the Software Engineering Section of NRL's Center for High Assurance Computer Systems. Before assuming her current position at NRL, she was a visiting scientist at the NATO Undersea Research Center in La Spezia, Italy. She served in 1996 as program co-chair for the 11th Annual COMPASS Conference and in 1997 as general chair for the Third IEEE Symposium on Requirements Engineering; serves as an associate editor for the *Journal of Real-Time Computing Systems*; and co-edited a book in 1996 entitled *Formal Methods for Real-Time Computing*. Her research interests are in formal methods, requirements, and real-time computing.

James Kirby, Jr. holds a master of software engineering from the Wang Institute of Graduate Studies. He is a member of the Software Engineering Section of NRL's Center for High Assurance Computer Systems. Prior to coming to NRL, Kirby was on the technical staff of the Software Productivity Consortium and on the faculty technical staff at the Wang Institute. While at the Software Productivity Consortium, he was a developer of CoRE, the Consortium Requirements Engineering Method. His research interests include software requirements and design methods and software process.

Bruce Labaw holds an MS degree in computer science from the University of Maryland. He is a member of the Software Engineering Section of NRL's Center for High Assurance Computer Systems. He conducts and supervises research in software engineering of hard real-time computer systems and supports the transition of this technology to the Navy. He previously worked on the NRL Software Cost Reduction project, which documented the requirements of the Operational Flight Program of the Navy's A-7 aircraft. His research interests are in formal methods, real-time computing, and software requirements.

Myla Archer holds an AM degree in mathematics from Harvard University and a PhD degree in computer science from the University of Illinois at Urbana-Champaign. She is a computer scientist in the Software Engineering Section of NRL's Center for High Assurance Computer Systems. Prior to coming to NRL, she taught mathematics at Wheaton College in Norton, Massachusetts, and served on the Computer Science faculty of the University of California, Davis. She was general chair of the 1991 Tutorial and Workshop on the HOL Theorem Proving System and Its Applications, and doctoral consortium chair of the Third IEEE Symposium on Requirements Engineering in 1997. Her research interests include formal methods, verification, requirements, and real-time computing.

Ramesh Bharadwaj holds the BE, ME, and PhD degrees in electrical engineering. He is a computer engineer in the Software Engineering Section at the Naval Research Laboratory in Washington, D.C. He has hands-on experience in systems development, having worked in industry as a hardware engineer, senior software engineer, and senior systems programmer. He has also spent two years managing software development projects. He has held research and development positions at the Philips Research Laboratories in Eindhoven; the Tata Institute of Fundamental Research in Bombay; Stanford University; and AT&T Bell Laboratories in Murray Hill, New Jersey. His research interests include tools and methods for software engineering, model checking, and decision procedures.