# Analyzing Tabular Requirements Specifications Using Infinite State Model Checking

Tevfik Bultan[1]
Univ. of California, Santa Barbara

Constance Heitmeyer[2]
Naval Research Laboratory

## Abstract

*This paper investigates the application of infinite state model checking to the formal analysis of requirements specifications in the SCR (Software Cost Reduction) tabular notation using Action Language Verifier (ALV). After reviewing the SCR method and tools and the Action Language, experimental results are presented of formally analyzing two SCR specifications using ALV. The application of ALV to verify or falsify (by generating counterexamples) the state and transition invariants of SCR specifications and to check Disjointness and Coverage properties is described. ALV is compared with the verification techniques that have been integrated into the SCR toolset.*

## 1. Introduction

Formal requirements languages such as RSML (Requirements State Machine Language) [17], a Statecharts variant, and SCR (Software Cost Reduction) [13], a tabular language for specifying requirements, have been used to specify the required behavior of real-world, safety-critical systems such as air traffic control systems and nuclear power plants. These formal languages produce precise, unambiguous specifications of the required system behavior that expose errors before they creep into the implementation. Further, these languages have a formal semantics which makes the use of automated formal analysis techniques possible. Using such techniques, defects in the requirements specifications can be detected and corrected early in the development process when correcting errors is cheap.

In the last two decades, significant progress has been made in automated verification techniques for finite state systems, especially in techniques based on model checking [8]. In hardware design, these techniques have been successfully transferred from academic research into industrial use. In recent years, model checking has been extended to analyze infinite state models, and infinite state model checkers have reached a maturity level comparable to that of finite state model checkers a decade ago. This paper investigates the application of an infinite state model checker called the Action Language Verifier (ALV) [7, 20] to the formal analysis of software requirements specified in the SCR tabular notation. The results of applying infinite state model checking using ALV to SCR requirements specifications is compared to the results of analyzing the SCR specifications with the formal analysis tools and techniques of the SCR toolset [10]. Both the strengths and weaknesses of the two approaches are identified, and potential synergies among these analysis techniques are discussed.

[1]This work was done while T. Bultan was visiting the Naval Research Laboratory on sabbatical leave from the University of California, Santa Barbara. His research is supported in part by NSF grant CCR-0341365.

[2]C. Heitmeyer's research is supported by the Office of Naval Research.

## 2. SCR Language and Toolset

In an SCR specification [10, 13], *monitored* and *controlled variables* represent the quantities in the system environment that the system monitors and controls. The required system behavior is specified as relations the system must maintain between the monitored and controlled variables. To specify these relations concisely, the SCR language provides two types of auxiliary variables—*mode classes*, whose values are *modes*, and *terms*—as well as conditions and events. A *condition* is a predicate defined on a system state. A basic *event*, denoted `@T(c)`, indicates that condition `c` changes from false to true. The event `@F(c)` is defined by `@T(¬c)`. If `c`'s value in the current state is denoted $c$ and its value in the next state as $c'$, then the semantics of `@T(c)` is defined by $\neg c \wedge c'$ and the semantics of `@F(c)` by $c \wedge \neg c'$. A *conditioned event*, denoted `@T(c) WHEN d`, adds a qualifying condition $d$ to an event and has the semantics $\neg c \wedge c' \wedge d$.

The mode classes, terms and controlled variables are called *dependent variables*. SCR specifications define the values of dependent variables using three types of tables: *condition*, *event* and *mode transition tables*. Each term and controlled variable is defined by either a condition or an event table. Typically, a condition table defines the value of a variable in terms of a mode class and a set of conditions, whereas an event table defines the value of a variable in terms of a mode class and a set of conditioned events. A mode transition table associates each source mode and a conditioned event with a destination mode. If the given event occurs in the source mode, then in the next state the system makes a transition to the destination mode.

Table 1 gives the format of a condition table defining the value of a dependent variable $r$ in terms of a mode class

**Table 1. Format of a Moded Condition Table**

| Mode $M$ | Condition | | | |
|---|---|---|---|---|
| $m_1$ | $c_{1,1}$ | $c_{1,2}$ | ... | $c_{1,p}$ |
| ... | ... | ... | ... | ... |
| $m_n$ | $c_{n,1}$ | $c_{n,2}$ | ... | $c_{n,p}$ |
| $r$ | $v_1$ | $v_2$ | ... | $v_p$ |

**Table 2. Format of a Mode Transition Table**

| Current Mode $M$ | Event | New Mode $M'$ |
|---|---|---|
| $m_1$ | $e_{1,1}$ | $m_{1,1}$ |
| | ... | ... |
| | $e_{1,k_1}$ | $m_{1,k_1}$ |
| ... | ... | ... |
| $m_n$ | $e_{n,1}$ | $m_{n,1}$ |
| | ... | ... |
| | $e_{n,k_n}$ | $m_{n,k_n}$ |

$M$ and a set of conditions $c_{i,j}$. The value of variable $r$ described by the table can be defined as a formula $F_r$,

$$F_r \equiv \bigvee_{i=1}^{n} \bigvee_{j=1}^{p} (M = m_i \wedge c_{i,j} \wedge r = v_{i,j}), \quad (1)$$

where $n$ is the number of modes in $M$, $p$ is the number of conditions for each mode in the table, and the value $v_{i,j}$ is a type-correct value of $r$. Note that a condition table defines the value of a variable with respect to a single state. In contrast, the value of a mode class or a variable described by an event table is defined on two consecutive states, i.e., on state transitions.

Table 2 shows the format of a mode transition table describing the transitions of a mode class $M$. The semantics of the table can be defined as a formula $F_M$,

$$\begin{aligned} F_M \quad \equiv \quad & \bigvee_{i=1}^{n} \bigvee_{j=1}^{k_i} (M = m_i \wedge e_{i,j} \wedge M' = m_{i,j}) \vee \\ & \neg(\bigvee_{i=1}^{n} \bigvee_{j=1}^{k_i} M = m_i \wedge e_{i,j}) \wedge M' = M, \end{aligned} \quad (2)$$

where $m_i$ denotes the $i$th source mode, $e_{i,j}$ the $j$th conditioned event for the $i$th source mode, and $m_{i,j}$ the associated destination mode.

The format of a moded event table defining a variable $r$ is identical to the format in Table 1 with each condition $c_{i,j}$ replaced by a conditioned event $e_{i,j}$ and $r$ replaced by $r'$. The semantics of a moded event table defining variable $r$ can be represented by a formula similar to (2), where $j$ varies from 1 to $p$, $F_r$ replaces $F_M$, $r' = v_{i,j}$ replaces $M' = m_{i,j}$, and $r' = r$ replaces $M' = M$.

Two relations, NAT and REQ, define the relationship between the current and next state values of all monitored and dependent variables. NAT specifies the natural constraints on monitored and controlled variables, such as constraints imposed by physical laws and the system environment. REQ uses the SCR tables to specify the required system behavior as constraints on the dependent variables. Given a set of dependent variables $D$, REQ is defined as the conjunction of the semantics described by each table; i.e., REQ $\equiv \bigwedge_{r \in D} F_r$. In SCR, the required system behavior is defined by a state machine $\Sigma = (S, \theta, \rho)$, where $S$ is the set of states (each state is a function mapping a state variable name to a type-correct value), $\theta$ is a predicate on $S$ which defines the set of initial states, and $\rho \subseteq S \times S$ is the transition relation which defines the allowable state transitions.

**Table 3. Mode Transition Table for `Cruise`**

| Current Mode `CruiseCtl` | Event | New Mode `CruiseCtl` |
|---|---|---|
| `Off` | `@T(IgnOn)` | `Inactive` |
| `Inactive` | `@F(IgnOn)` | `Off` |
| `Inactive` | `@T(Lever=const)` WHEN `IgnOn` AND `EngRunning` AND NOT `Brake` | `Cruise` |
| ... | ... | ... |
| `Override` | `@T(Lever=resume)` WHEN `IgnOn` AND `EngRunning` AND NOT `Brake` OR ... | `Cruise` |

## 2.1. Two SCR Specifications

**The Cruise Control System (CCS).** The specification of CCS describes the required behavior of a cruise control system for a BMW. CCS controls the car's throttle and speed based on the state of the brake, engine, ignition switch, and cruise control lever. The SCR specification of CCS [11] uses a mode class `CruiseControl` to indicate the different modes of CCS—`Off`, `Inactive`, `Cruise`, and `Override`. The monitored variables of CCS—`IgnOn`, `Speed`, `Brake`, `Lever`, and `EngRunning`—indicate the state of the car's ignition, the speedometer reading, the positions of the brake and cruise control switch, and the state of the engine. Another monitored variable represents time. The controlled variable `Throttle` represents the state of the throttle, and the term `DesiredSpd` represents the desired speed. Table 3 contains an excerpt from the mode transition table for the CCS. The requirement that the throttle value depends on 1) the difference between the car speed and some constant value and 2) whether the CCS has been in a given mode for more than 500 ms can make analysis of the CCS specification difficult.

**The Safety Injection System (SIS).** The SIS specification describes the requirements of the control software for a nuclear reactor's cooling system [9]. SIS monitors the water pressure of the cooling system. When water pressure drops below a certain constant value, the system starts safety injection (if it is not overridden). The monitored variables of SIS—`Block, Reset,` and `WaterPres`—denote the states of the block and reset switches and the water pressure reading. The mode class `Pressure` consists of three system modes: `TooLow`, `Permitted`, and `High`. The Boolean term `Overridden` indicates whether safety injection is overridden, and the controlled variable `SafetyInjection` indicates whether safety injection is turned on. Table 4, a condition table, defines the value of the controlled variable

**Table 4. Condition Table for** `Safety Injection`

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

`SafetyInjection`. Because water pressure can vary from 0 to 2,000 and because the three modes represent the three intervals of water pressure values that affect the value of `SafetyInjection`, analyzing the SIS specification mechanically can be challenging.

### 2.2. SCR Toolset

The SCR toolset integrates the formal analysis techniques and tools presented below. Each tool may be used alone, or in combination with other tools in the toolset, to analyze an SCR specification [10].

**Consistency Checking.** The SCR consistency checker checks for syntax and type errors, circular definitions, and other simple errors, and for violations of Disjointness and Coverage [13]. The checks other than those for Disjointness and Coverage are analogous to standard compiler checks. Checking Disjointness detects nondeterminism in the SCR tables, whereas checking Coverage exposes missing cases.

**Simulator.** The SCR simulator symbolically executes an SCR specification to allow users to validate that the specification captures the intended system behavior. The simulator is also useful for demonstrating and validating property violations detected by a model checker.

**Model Checking with Spin.** The SCR toolset includes a translator from SCR to Promela [3], the language of an explicit state model checker Spin [14]. (Translators from SCR also exist for symbolic model checkers, such as SMV.) Using the translator and Spin, one can check SCR specifications for both *state invariants*, one-state properties that hold in every reachable state, and *transition invariants*, two-state properties that hold in every reachable transition.

**Property Checking with Salsa.** The SCR property checker Salsa [5] may be used to check SCR specifications for Disjointness and Coverage and for satisfaction of state and transition invariants. Salsa can check the validity of formulas on Boolean, enumerated and integer variables restricted to Presburger arithmetic. It uses BDDs for analyzing formulas on Boolean and enumerated variables and an automata representation for analyzing Presburger arithmetic formulas.

**Theorem Proving with TAME.** TAME (Timed Automata Modeling Environment), a specialized interface to PVS [19], offers templates for specifying automata models and customized strategies which implement high-level proof steps for proving automaton properties [1]. Initially developed for Timed Input/Output Automata, TAME has been adapted to SCR by an automatic SCR-to-TAME translator and by adding SCR-specific strategies that prove many properties automatically and exhibit "problem transitions" for undischarged proof goals.

**Abstraction.** In [3, 4, 12], three abstraction techniques are described which reduce the state space of an SCR requirements specification. The first technique called *slicing* removes variables irrelevant to the validity of the property under analysis. The second and third techniques perform *data abstraction* by replacing variables with large domains (such as integers) with enumerated variables, where each value of an enumerated abstract variable represents a range of values for the corresponding concrete variable.

**Invariant Generation.** Algorithms for generating state invariants from SCR specifications are described in [15, 16]. Such invariants are useful as auxiliary lemmas in proving properties of SCR specifications with TAME and Salsa.

## 3. Action Language Verifier

Action Language is a specification language for reactive software systems. The Action Language Verifier (ALV) consists of 1) a compiler that converts Action Language specifications into symbolic representations, and 2) an infinite-state symbolic model checker which verifies or falsifies (by generating counterexamples) CTL properties of Action Language specifications [7, 20].

An Action Language specification contains integer, Boolean and enumerated variables, parameterized integer constants, and a set of modules and actions which are composed using synchronous and asynchronous composition operators [6, 7]. Semantically, each Action Language module corresponds to a transition system $T = (I, S, R)$ where $S$ is the set of states, $I \subseteq S$ is the set of initial states and $R \subseteq S \times S$ is the transition relation. The variable declarations of a module define the module's state set. A *restrict expression* can be used to limit the state space to states satisfying the restrict expression. A module's *initial expression* defines the module's set of initial states. Each *action expression* corresponds to a single execution step and a *module expression* defines the transition relation of the module in terms of its actions and submodules using composition operators.

ALV is a symbolic CTL model checker and uses the least and greatest fixpoint characterizations of CTL operators to compute the truth set of a given CTL property. It iteratively computes the fixpoints starting from the innermost temporal operator in the input CTL formula [18]. Since Action Language allows specifications with unbounded integer variables, fixpoint computations are not guaranteed to converge. To achieve convergence, ALV uses conservative approximation techniques. ALV also implements a forward fixpoint computation starting from the initial states to compute an over-approximation of the reachable state space of a specification.

**Table 5. Desired Properties of CCS**

| | |
|---|---|
| A1 | $\texttt{Brake} \Rightarrow \texttt{Throttle} = \texttt{off}$ |
| A2 | $\texttt{Throttle} = \texttt{accel} \Rightarrow \texttt{DesiredSpd} > \texttt{Speed}$ |
| A3 | $\texttt{CruiseControl} \neq \texttt{Off} \wedge \texttt{EngRunning} \Rightarrow \texttt{IgnOn}$ |
| A4 | $\texttt{Throttle} \neq \texttt{off} \wedge \texttt{EngRunning} \Rightarrow \texttt{IgnOn}$ |
| A5 | $\texttt{Speed}' = \texttt{Speed} \Rightarrow \texttt{Throttle}' \neq \texttt{accel}$ |
| A6 | $\neg\texttt{EngRunning} \wedge \texttt{EngRunning}' \Rightarrow \texttt{Throttle}' = \texttt{off}$ |
| A7 | $\texttt{CruiseControl} = \texttt{Off} \Leftrightarrow \neg\texttt{IgnOn}$ |
| A8 | $\texttt{Throttle} = \texttt{accel} \Rightarrow (\texttt{DesiredSpd} > \texttt{Speed} \vee \texttt{DURLeverEQconst} > 500)$ |
| A9 | $\texttt{DesiredSpd}' \neq \texttt{DesiredSpd} \Rightarrow \texttt{CruiseControl}' = \texttt{Cruise} \wedge (\texttt{CruiseControl} = \texttt{Cruise} \vee \texttt{CruiseControl} = \texttt{Inactive})$ |
| A10 | $\texttt{DesiredSpd} = \texttt{Speed} \Rightarrow (\texttt{CruiseControl} \neq \texttt{Override} \vee \texttt{Lever} \neq \texttt{release} \vee \texttt{Throttle} = \texttt{off})$ |
| A11 | $\texttt{Speed}' = \texttt{Speed} \wedge \texttt{DesiredSpd} = \texttt{Speed} \Rightarrow \texttt{DesiredSpd}' = \texttt{DesiredSpd}$ |

ALV uses the Composite Symbolic Library [21] as its symbolic manipulation engine for Boolean logic and Presburger arithmetic formulas. (Presburger arithmetic consists of linear arithmetic expressions, Boolean connectives and universal and existential quantification.) The Composite Symbolic Library integrates multiple symbolic representations using an abstract interface and a disjunctive, composite representation that handles operations on multiple symbolic representations. Originally developed using a polyhedral representation for Presburger arithmetic constraints [21], ALV was later extended by adding an automata representation for Presburger constraints [2]. ALV also uses BDDs to encode Boolean and enumerated variables. Polyhedral and automata representations can be combined with BDDs using ALV's disjunctive composite representation.

## 4. SCR Verification with ALV

To perform analysis with ALV, the SCR specifications of the CCS and the SIS were translated into Action Language. Both Action Language translations consist of a single module. The monitored and controlled variables, mode classes, and terms in the SCR specification were all declared as variables in the Action Language specification. The restrict expressions in the Action Language specifications were used to restrict the values of variables based on the SCR type declarations. For example, if a variable `r` is declared as an integer in Action Language, by default its domain is the set of all integers, an infinite set. Such a variable can be restricted to a finite domain from 1 to 100 by using the following expression: `restrict: r >= 1 and r <= 100`.

The initial expressions in the Action Language specifications of CCS and SIS are used to declare the initial values of state variables. For both specifications, a single module expression defines the system's transition relation which is constructed based on the SCR semantics given in Section 2.

**Verification of CCS.** The internal representation of the CCS in ALV consists of 13 Boolean and four integer variables. (In the representation of the transition relation, the number of variables is doubled since each variable is represented with one current state variable and one next state variable.) Six of the Boolean variables are automatically generated to represent three enumerated variables in the specification. Four Boolean variables are used to represent the desired transition invariants (as explained below).

**Table 6. Verification Results for CCS**

| Property | Time | Iterations |
|---|---|---|
| A1 | 0.76 sec | 2 |
| A3 | 0.50 sec | 2 |
| A4 | 0.78 sec | 2 |
| A6 | 0.37 sec | 2 |
| A7 | 0.36 sec | 1 |
| A8 | 0.32 sec | 1 |
| A9 | 0.32 sec | 1 |
| A10 | 0.32 sec | 1 |
| A11 | 0.32 sec | 1 |

Table 5 lists the desired invariants of CCS. Since properties A1–A4, A7, A8, and A10, are defined on a single state, each is a desired state invariant and can therefore be expressed using the CTL temporal operator AG. In contrast, each of the properties A5, A6, A9 and A11 is a transition invariant since each is defined in terms of both the current state and the next state. Such properties can be specified in Action Language by declaring a Boolean variable which is true if and only if the property is true. For example, to verify the property A6, an auxiliary Boolean variable $b_{A6}$ is declared and initialized to **true**, and the following constraint defining the value of the variable $b_{A6}$ in the next state is conjoined with the module expression for CCS:

$$(\neg\texttt{EngRunning} \wedge \texttt{EngRunning}' \Rightarrow \texttt{Throttle}' = \texttt{off}) \Leftrightarrow b'_{A6}$$

Then, property A6 is expressed as AG$(b_{A6})$, i.e., property A6 holds if and only if $b_{A6}$ is true in every reachable state.

Table 6 lists the time ALV required to verify the nine true properties in Table 5. ALV was run on a Linux machine with a 2.8 GHertz Pentium 4 processor and 2 GBytes of main memory. For each property, the table lists the time ALV required to verify the property, including the time needed to construct the transition relation (approximately 0.31 sec.). In verifying each property, ALV required 23.2 MB of memory. Note that the transition relation can be constructed once, and then all nine properties can be verified one after another; using this approach, ALV verified all properties in 1.65 sec. (which is less than the sum of the times shown in Table 6) using 23.2 MB memory. The last column shows the number of iterations for each fixpoint computation. The properties requiring only a single iteration are inductive properties, i.e., the fixpoint converges in the first iteration. Fixpoints for all properties listed in Table 6 converged directly with no approximations.

For all results listed in Table 6, we used ALV with the disjunctive composite representation, where the symbolic representation for the integer variables is a polyhedral representation, and the symbolic representation for the Boolean and enumerated variables is in terms of BDDs. The transition relation of the CCS using this representation consists

**Table 7. Desired Properties of SIS**

| | |
|---|---|
| S1 | Overridden ⇒ Pressure ≠ High ∧ Reset = Off |
| S2 | Reset = On ∧ Pressure ≠ High ⇒ ¬Overridden ∧ SafetyInjection = On |
| S3 | Reset = On ∧ Pressure ≠ High ⇒ ¬Overridden |
| S4 | Reset = On ∧ Pressure = TooLow ⇒ SafetyInjection = On |

**Table 8. Verification Results for SIS**

| Property | Time | Memory | Iterations |
|---|---|---|---|
| S1 | 0.03 sec | 1.2 MB | 1 |
| S3 | 0.03 sec | 1.2 MB | 1 |
| S4 | 12.82 sec | 11.9 MB | 1102 |

of 226 disjuncts which contain 5,693 BDD nodes and 326 polyhedra with 3,498 equality or inequality constraints.

Two properties not listed in Table 6 are A2 and A5. ALV shows that property A5 is false by generating a counterexample. Fixpoint computations for generating the counterexample, which has length seven, required 11 iterations. Counterexample generation for property A5 using the automata representation for all the variables required 42.91 sec. and used 2.6 MB of memory. In the automata representation, the definition of the transition relation of the CCS contains 2,004 states. In the automata representation used in ALV, the transitions of the automata are represented using BDDs [2]. The transitions of the automaton representing the CCS transition relation contain 22,744 BDD nodes.

For property A2, the fixpoint computations of ALV did not converge. When the value of one constant in the specification was changed from 500 to 5, ALV was able to generate a counterexample for the property. This failure to converge arose because generating a counterexample for property A2 requires the time variable to reach a value that exceeds the constant. Since the time variable starts at zero and increases by at most one in each execution step, the counterexample for this property contains hundreds of states. When a smaller value of the constant is used, a shorter counterexample can be generated. For the smaller constant value, fixpoint computations for the counterexample generation require 14 iterations, and the generated counterexample has length ten. Counterexample generation for property A2 using the automata representation for all the variables required 32.86 sec. and 3.0 MB of memory.

**Verification of SIS.** ALV uses seven Boolean variables and one integer variable to represent the states of SIS and 14 Boolean and two integer variables to represent its transition relation. Table 7 lists the desired properties of the SIS. Because each property is a desired state invariant, each can be expressed using the CTL temporal operator AG.

Table 8 lists the time, memory, and number of fixpoint computations required to verify each true property of the SIS. The automata representation was used to verify S4 and the composite representation with polyhedra and BDDs to verify properties S1 and S3. The SIS transition relation using the composite representation consists of 21 disjuncts which contain 282 BDD nodes and 51 polyhedra with 148 equality or inequality constraints. Using the automata representation, the same transition system is represented with an automaton containing 95 states and 393 BDD nodes. Table 8 shows that property S4 required 1,102 fixpoint iterations. This means that ALV had to enumerate the complete state space to verify this property. The fact that ALV is able to compute 1,102 fixpoint iterations in 12.82 sec. shows that the size of the symbolic representation does not increase drastically during the image computations.

ALV generates a counterexample to demonstrate that property S2 does not hold. The fixpoint computations for the counterexample generation required 1,104 iterations, and the counterexample has length 887. Using the automata representation, ALV generates the counterexample in 25.71 sec. using 72.3 MB of memory.

## 5. Consistency Checking

In addition to checking for syntax, type, and other simple errors, the SCR consistency checker also checks the specification for the Disjointness and Coverage properties [13].

The Disjointness check ensures that each function defined by an SCR table is well-formed; for example, for each mode and condition, a condition table cannot assign more than one value to a variable. This check, which corresponds to checking that for each mode in an SCR table every pair of conditions for that mode are pairwise disjoint, is based on the semantics of SCR tables given in Section 2. The Disjointness check for condition tables corresponds to checking the following formula:

$$\forall i, 1 \leq i \leq n, \forall j, k, 1 \leq j, k \leq p : j \neq k \Rightarrow c_{i,j} \wedge c_{i,k} \equiv \mathbf{false}, \quad (3)$$

where $n$ is the number of modes and $p$ the number of conditions, and $c_{i,j}$ and $c_{i,k}$ are the $j$th and $k$th conditions for the $i$th mode. The Disjointness check for event tables is defined similarly with $c_{i,j}$ and $c_{i,k}$ in (3) replaced by $e_{i,j}$ and $e_{i,k}$.

The Coverage check analyzes each condition table to ensure that the function defined by the table is a total function. The check is performed by checking that the disjunction of a set of conditions evaluates to **true**. Based on the semantics of the SCR tables given in Section 2, the Coverage check for a condition table analyzes the following formula:

$$\forall i, 1 \leq i \leq n : \bigvee_{j=1}^{p} c_{i,j} \equiv \mathbf{true}. \quad (4)$$

The SCR consistency checker uses the formulations in (3) and (4) to check the SCR tables for Disjointness and Coverage. Because ALV is a CTL model checker, using ALV to check Disjointness and Coverage requires the reformulation of these properties as CTL formulas.

**Disjointness Checking with ALV.** The CTL formulation of the Disjointness property states that, for any current system

state, the value of any dependent variable $r$ in the next state is uniquely determined by the value of the monitored variables in the next state. Formally, the Disjointness property holds for any dependent variable $r \in D$ if and only if all system states satisfy the following CTL property:

$$\mathrm{EX}(r = v_r \wedge \bigwedge_{\hat{r} \in R} (\hat{r} = v_{\hat{r}})) \Rightarrow \mathrm{AX}(\bigwedge_{\hat{r} \in R} (\hat{r} = v_{\hat{r}}) \Rightarrow r = v_r). \quad (5)$$

In (5), $R$ is the set of monitored variables, and $v_r$ and $v_{\hat{r}}$ are type-correct values of variables $r$ and $\hat{r}$. The formulation of Disjointness in (5) has been proven to be equivalent to the original SCR formulation in (3).

The Disjointness property states that the above CTL property must hold for every system state, reachable or not. To achieve this, the initial condition in the Action Language specification was replaced by **true** (which represents all states), and the above CTL property was then checked. Using ALV, the verification of the Disjointness property for all dependent variables required 8.07 sec. and 6.6 MB for CCS and 2.07 sec. and 257.8 MB for SIS.

**Coverage Checking with ALV.** The Coverage property may be represented in CTL using an auxiliary Boolean variable for each variable defined by a condition table. Given a variable $r$ and the auxiliary boolean variable $b_r$, the formula in (1) defining the value of $r$ is modified as follows:

$$F_r \equiv \bigvee_{i=1}^{n} \bigvee_{j=1}^{p} (M' = m_i \wedge c'_{i,j} \wedge r' = v_{i,j} \wedge b'_r = \mathbf{true}) \vee \neg(\bigvee_{i=1}^{n} \bigvee_{j=1}^{p} (M' = m_i \wedge c'_{i,j})) \wedge b'_r = \mathbf{false} \quad (6)$$

The Coverage property for variable $r$ holds if and only if $b'_r$ is never set to false, i.e., if and only if the CTL property $\mathrm{AX}(b_r)$ holds for every system state. The property is evaluated by replacing the initial condition in the Action Language specification with **true** and then checking the CTL property $\mathrm{AX}(b_r)$. Using ALV, verification of the Coverage property for all variables described by condition tables required 0.06 sec. and 4.0 MB for CCS and 0.02 sec. and 1.2 MB for SIS.

## 6. Comparing ALV with Other Analysis Tools

This section compares the results of verifying SCR specifications using ALV with the results of verifying the specifications with other analysis tools.

**Multiple Symbolic Representations.** Unlike most symbolic model checking tools, ALV supports multiple symbolic representations (polyhedra and automata representations for Presburger arithmetic formulas combined with BDD representation). The object-oriented design of the Composite Symbolic Library and ALV (based on an abstract interface for symbolic representations) enables polymorphic verification—to improve the efficiency of verification users can choose different symbolic representations at runtime using command line arguments without recompiling the tool. Experience with ALV shows that the relative efficiency of polyhedral and automata representations can change for different specifications. For example, verifying the Disjointness property for CCS required 8.07 sec. and 6.6 MB for the polyhedral representation and 50.78 sec. and 2.7 MB for the automata representation. In verifying the Disjointness property for SIS, the polyhedral representation required 2.07 sec. and 257.8 MB, while the automata representation required 1.13 sec. and 2.2 MB. For CCS, the verification time for the automata representation and for SIS, the memory usage for the polyhedral representation are unacceptably high, and hence the availability of an alternative representation which required fewer resources was useful.

**Explicit State vs. Symbolic Model Checking.** Spin, an explicit state model checker [14], has been used to verify many properties of SCR specifications [3, 4, 12]. Explicit state model checkers like Spin usually use efficient depth-first search algorithms to find counterexamples. The first counterexample that Spin finds causes it to halt its search and report the counterexample. If Spin exhausts the state space without finding a counterexample, it reports that the property is verified. Hence, Spin is more efficient in finding errors than in verifying correctness.

Symbolic model checkers like ALV work differently from Spin since they typically compute fixpoints corresponding to the truth set of the negation of the input temporal property. If the intersection of the truth set of the negated property and the initial state set is empty, then the property holds; otherwise, a counterexample is constructed starting from a state in the intersection. During its fixpoint computations, ALV computes a characterization of *all* counterexample behaviors, not just one counterexample as in explicit state model checking. Given this difference, Spin is usually more efficient than ALV for detecting property violations, since it reports the first counterexample it finds.

If a specification with a large state space has no errors (or even if it has errors), Spin can run out of memory without exhausting the state space. One cause of state space explosion is variables with large domains, such as integers. Since Spin performs a depth-first search starting from the initial states, the size of the *reachable* state space is important. A specification with integer variables can be easily verified with Spin if those integer variables have only a few values or stay constant. In contrast, specifications with integer variables with no fixed initial value are difficult to verify with Spin. Such cases are very likely to occur in SCR specifications since SCR is a requirements specification language, and often the input variables in requirements specifications do not have a fixed initial value but a range of values.

For symbolic model checkers such as ALV, the size of the reachable state space is not usually a problem because the size of a symbolic representation is not proportional to

the number of states. This is why symbolic representations such as BDDs succeed. Unlike Spin, ALV can verify specifications with a very large (even infinite) state space without running out of memory as long as the size of the symbolic representations stays small during the fixpoint computations. Hence, for specifications with a large number of initial states, e.g., specifications containing some integer variables with no fixed initial value, ALV is usually more efficient than Spin. For example, in the SIS specification verified with Spin in [3], the initial value of monitored variable `WaterPres` is restricted to the single value 14. Thus any verification result obtained only holds if the initial value of `WaterPres` is 14—clearly a very restrictive initial state for SIS. With ALV, the same specification with a more generalized initial condition was verified—the initial value of `WaterPres` is any value consistent with `TooLow`, i.e., $0 \leq$ `WaterPres` $< 900$. ALV's performance for this generalized specification was identical to that reported in Section 4. Thus, increasing the number of initial states from one to 900 states did not change ALV's performance.

**Finite vs. Infinite State Model Checking.** Another important difference between Spin and ALV is that Spin is a finite state model checker, whereas ALV is an infinite state model checker. Thus, ALV can verify infinite state specifications whereas Spin cannot. For example, ALV can verify specifications in which some integer variables have arbitrarily large values. To verify such specifications with Spin, one must either restrict the values of the infinite state variables to finite domains or use abstraction techniques.

ALV can also verify parameterized specifications. For example, ALV can verify specifications with unspecified integer constants. When such a specification is verified, the results hold for all possible values of the parameterized constant. ALV can also verify specifications with a parameterized number of finite state components, i.e., specifications containing an arbitrary number of instantiations of a finite state component. Such parameterized systems cannot be verified using Spin. To verify such systems using a finite state model checker, one must restrict the unspecified constants to a finite set of values and the parameterized components to a finite set of components.

Since infinite state model checking is an undecidable problem in general, the fixpoint computations used in ALV are not guaranteed to converge. To avoid running forever without terminating, ALV uses conservative approximation techniques such as truncated fixpoint computations and widening, which compute lower or upper approximations of the exact fixpoint sequence [7, 20]. The approximations are conservative in the following sense: ALV does not produce false positives or false negatives but may report the result of verification to be inconclusive. Thus, if approximation techniques are used, the output of ALV is one of the following: 1) the property is verified, 2) the property is false and a counterexample is constructed, or 3) the analysis is inconclusive. Clearly, options 1 and 2 are always true negatives and true positives. The goal of ALV's approximation techniques is to minimize the third result as much as possible (although sometimes this is unavoidable due to the undecidability of the verification problem).

The SIS specification verified in Section 4 with ALV is the finite state specification verified with Spin in [3]. In the finite state specification, the monitored variable `WaterPres` has a finite domain: $0 \leq$ `WaterPres` $\leq 2000$. Thus, any verification result for this specification is only guaranteed to hold if `WaterPres` lies in this range. We converted SIS to an infinite state specification by declaring that `WaterPres` can have any nonnegative value. ALV verified properties S1 and S3 on this infinite state specification in exactly the same time as for the finite case reported in Section 4. Hence, the sizes of the symbolic representations used by ALV do not increase for these properties when the finite domain of `WaterPres` is replaced with an infinite domain. However, for the infinite state specification, the counterexample generation for property S2 and the verification of S4 did not converge. To overcome this, the approximate fixpoint computations mentioned above were applied. To construct a counterexample for property S2, the truncated fixpoint computations were applied, and a counterexample was generated in about the same amount of time as in the finite case. For property S4, the widening heuristic was applied. Interestingly, applying the widening heuristic allowed ALV to verify property S4 much faster than in the finite case—in 0.77 sec. using 1.8 MB memory vs. 12.82 sec. and 11.9 MB in the finite case. This is because the widening heuristic causes the fixpoint iterations to converge (to an upper approximation of the fixpoint) much faster than the exact fixpoint computation (11 iterations vs. 1,102 iterations). Moreover, the approximation computed by the widening heuristic is strong enough to prove the property. This demonstrates that approximate fixpoint computations are not only useful for verifying infinite state systems but can also improve the efficiency of symbolic verification of finite state systems.

Next, we consider the constants in the SIS specification. In the finite state specification of SIS in [3], the three modes of the `Pressure` mode class are defined based on two constants, `Low` $= 900$ and `Permit` $= 1000$. This means that any verification of this specification holds only for these concrete values. We constructed a parameterized SIS specification in which the constants `Low` and `Permit` are declared as parameterized constants, where $0 \leq$ `Low` $<$ `Permit`; i.e., `Low` and `Permit` are declared as constants with unknown values but `Low` is assumed to have a nonnegative value less than `Permit`. The result of verifying the parameterized specification holds for any value of the parameterized constants. ALV verified the properties S1 and

S3 on the parameterized SIS specification using the same amount of time and memory as reported in Section 4 for the finite case. In the parameterized case, counterexample generation for property S2 and verification of S4 did not converge as in the infinite case (note that the parameterized constants have an infinite domain). Using widening, ALV verified property S4 in 2.07 sec. using 3.8 MB of memory; the fixpoint computation required 15 iterations. A counterexample for property S2 was generated using the truncated fixpoint computations in 3.08 sec. with 2.7 MB of memory. In the parameterized case, the shortest counterexample has only two states (as opposed to 887 states in the finite state case); hence, counterexample construction requires much less time. The long counterexample path in the finite specification is due to the concrete values assigned to the constants. When these constants are parameterized, shorter counterexamples are generated.

**Abstraction vs. Symbolic Representation.** In [4, 12], two general abstraction techniques are used to reduce the state space of SCR requirements specifications. One technique—slicing—removes variables irrelevant to the validity of the given property using dependency analysis. Another technique—data abstraction—replaces variables with large domains, such as integers, with enumerated variables, where each value of an enumerated abstract variable represents a range of values. The first technique, slicing, can also help ALV in verifying SCR specifications. Because the sizes of the symbolic representations used in ALV increase with the number of variables in the specification, reducing the number of variables reduces the size of the symbolic representations, thus improving the performance of ALV. For example, the SCR dependency analyzer determines that property A3 in Table 5 only depends on the monitored variables `IgnOn`, `EngRunning`, `Brake` and `Lever`, and automatically constructs an abstract SCR specification containing only these monitored variables and the state variables that depend on them. ALV verified property A3 after this reduction in 0.02 sec. using 0.8 MB of memory. Without this reduction, verifying the property required 0.5 sec. and 23.2 MB.

A second technique, data abstraction, replaces variables with large domains with abstract enumerated variables. If a specification contains an unbounded integer variable or an unspecified, arbitrarily large integer constant, some abstraction is necessary before a finite state model checker such as Spin can be applied. As discussed earlier, using ALV, we were able to verify the SIS specification analyzed in [3] without restricting the integer variables or unspecified constants to finite domains and without using any abstractions.

**Invariant Generation.** The techniques for invariant generation presented in [15, 16] exploit the structure of SCR tables. These techniques construct invariants defined on Boolean and enumerated variables from event and mode transition tables.[1] They compute a fixpoint using the function defined by an SCR table, the definitions of monitored variables and other variables on which the function depends, the initial state definition, and previously computed invariants. Although the techniques do not compute the strongest possible invariants (since they do not perform a reachability analysis), the invariants generated with these algorithms have proven highly useful in analyzing the properties of practical systems.

ALV implements forward fixpoint computations starting from the initial states to compute the reachable state space of a specification. The exact characterization of the reachable states corresponds to the strongest invariant of the specification. Usually, this invariant is difficult to compute, and exact fixpoint computations do not converge. ALV's conservative approximation techniques can be used to compute approximations to the reachable state space, i.e., an over-approximation of the strongest invariant. This approximation is still a system invariant, since every system state is included in the over-approximation. Thus, ALV can be used as an alternative invariant generator for SCR specifications.

The forward fixpoint computation used in ALV is sensitive to the characterization of the initial states, whereas the invariant generation techniques in [15, 16] work only on the transition relation and therefore are not sensitive to the characterization of the initial states. Dependence on the characterization of the initial states can be both an advantage and a disadvantage. It can be an advantage because stronger invariants may be produced. It can be a disadvantage if the initial states are arbitrarily restricted to a subset of all possible initial states (e.g., if the specification is over-specified), and thus discovery of more general properties is prevented. This may also adversely affect the approximation techniques. We observed this in the SIS specification where the approximation techniques used in ALV produce a better result when the input condition is less restrictive. For example, when ALV computes the over approximation of the reachable state space for the SIS specification it over-approximates the value of the `WaterPres` variable in `Permitted` and `High` modes. This is because the initial state of the SIS specification has a single initial value for `WaterPres`. When a more general initial state is specified, ALV computes the exact reachable state space for the SIS specification (i.e., generates the strongest invariant).

**Theorem Proving vs. Model Checking.** In automation level and expressiveness, infinite state model checking lies between finite state model checking and theorem proving. While the input languages of model checkers are less expressive than the languages of many theorem provers,

[1] By definition, condition tables define state invariants, and hence generating state invariants from condition tables is straightforward.

model checkers are normally more automated than theorem provers. However, as explained above, fixpoint computations in an infinite state model checker may not converge, which is analogous to a first-order theorem prover that exhaustively searches for a proof without terminating.

Unlike theorem provers, an infinite state model checker, such as ALV, has the ability to generate counterexamples. As explained above, while the verification and refutation results generated by ALV are never spurious, the result of ALV's analysis can be inconclusive. This is analogous to cases in which a theorem prover cannot prove a property using automated techniques; to make progress, it requires user assistance. In ALV, user input is restricted to choosing among different verification heuristics and modifying default parameters of these heuristics whereas typically theorem provers support more interactive verification.

## 7. Utility of ALV in the SCR Toolset

The experimental results described in Sections 4– 6 suggest several ways in which ALV could prove useful in verifying SCR specifications. The next section discusses some of these and also indicates some types of analysis, e.g., consistency checking, where the utility of ALV is less obvious.

**Availability of Different Representations.** ALV's support for more than a single symbolic representation could prove quite useful in the SCR toolset. As noted above, if ALV performs badly for one symbolic representation, the user can switch to another symbolic representation and achieve (perhaps) improved performance. However, the ability to experiment with different representations is less attractive to software practitioners than to researchers. Practitioners usually prefer more specialized tools like Salsa which are designed to perform verification without user intervention and which do not require the user to select from a set of alternatives. One solution to this problem is for ALV to automatically compute two different representations and then to analyze both in parallel. If one analysis completes before another, the results of the faster analysis are returned to the user. A similar approach can be used for choosing among different types of fixpoint computations. A more powerful solution, a topic for future research, would rely on heuristics to select the best representation or fixpoint computation based on the given specification.

**Parameterization.** The experiments with ALV identified some shortcomings in the expressiveness of the language supported by the SCR toolset. One problem is the inability to specify systems with more than a single initial state. A limitation of the current toolset implementation, this is not a limitation of the SCR requirements model [10] nor of Salsa and TAME. Moreover, this limitation can be overcome by using an abstract variable to define the initial state as described in Section 6. A second problem is that the current toolset does not support a parameterized number of finite state components. Integrating ALV into the toolset or extending the language supported by the toolset to handle parameterized specifications would significantly enhance the utility of the SCR language and tools.

**Comparing ALV with Salsa and TAME.** As with ALV, the result of verifying a specification with either Salsa or TAME may be inconclusive. In the case of both Salsa and TAME, an inconclusive result means that 1) the property is true but one or more auxiliary lemmas are needed to prove the property or 2) the property is false. Applying Salsa or TAME to a specification is equivalent to performing one inductive step (i.e., one iteration) in an ALV analysis. By conjoining automatically generated invariants or other proved invariants with the SCR specification, both Salsa and TAME can often automatically verify true properties, such as properties A1, A3, A4, and A6, cases in which ALV needed more than one inductive step to complete the verification. Hence, when auxiliary properties are conjoined with the SCR specification, both Salsa and TAME are able to verify properties that ALV verifies on the original specification only. The advantage of ALV occurs when a property is false; in such cases, ALV can construct a counterexample, whereas Salsa and TAME cannot. However, when a proof fails, TAME generates as unproved subgoals one or more *dead ends*, each associated with a set of *problem transitions*. In the case of a false property, the full set of problem transitions contains all transitions corresponding to property violations. Thus, in the case of a false property, TAME, like ALV, can compute a characterization of *all* property violations.

**Abstraction, Invariants, and Approximation.** One promising approach would be to use both automated abstraction and approximation techniques to reduce the size of the state space. As suggested in Section 6, the SCR abstraction techniques could be applied first and then ALV's approximation techniques could be used to obtain further reductions. However, while approximation heuristics can often be used to verify large, complex specifications, a major problem with approximations, such as widening, is that they sometimes "over-approximate," i.e., eliminate information that is needed to prove the property. Unlike ALV's approximation techniques, the abstractions in SCR are constructed based on the property to be analyzed, and therefore provide more precision.

Another promising approach is to combine invariants, such as those generated by the SCR algorithms, with approximation techniques. One important feature of SCR's automatically generated invariants is that they were designed to be easy for practitioners to understand. Hence, they are not only useful as auxiliary lemmas in verification, they are also helpful in user validation of the specifications. Although the forward fixpoint computation implemented in ALV can be used to automatically construct invariants (as

mentioned in Section 6), it is unlikely that such invariants will be understandable by the users.

**Using ALV for Consistency Checking.** While Section 5 demonstrates the feasibility of analyzing SCR specifications for Disjointness and Coverage using ALV, the benefits of doing so may be limited. Using ALV for consistency checking may be beneficial only if the symbolic representations provided by ALV provide an efficient encoding for the SCR tables. Disjointness and Coverage properties check the functions defined by the SCR tables for well-formedness and totality. Checking the SCR table entries using the formulas in (3) and (4) is both sufficient and much simpler then using a CTL model checker such as ALV. Moreover, the SCR Consistency Checker provides useful diagnostic information when a check fails—both a user-friendly counterexample illustrating an instance of the error *and* a display of the appropriate table with the erroneous entries highlighted.

## 8. Conclusions

Our results demonstrate that infinite state model checking techniques are useful in verifying properties of SCR specifications. They also show that infinite state model checking may also be used for consistency checking and invariant generation, but that the current SCR techniques have the advantage that they produce diagnostic information when the consistency checks fail as well as invariants that are easy for software developers and domain experts to understand. Although the two specifications analyzed with ALV demonstrated the potential utility of infinite state model checking for analyzing practical systems specified in SCR, they are quite modest in size and complexity. Hence, in future work, we plan to further evaluate the utility of infinite state model checkers, such as ALV, for evaluating properties of practical systems specified in SCR. One candidate is the safety-critical weapons control system in [12].

**Acknowledgments.** The authors are grateful to Myla Archer, Ralph Jeffords, and the anonymous referees for their insightful comments on earlier drafts and to Ralph Jeffords for proving the equivalence of formulas (3) and (5).

## References

[1] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9:201–232, 2002.

[2] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science (IJFCS)*, 14(4):605–624, August 2003.

[3] R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *Proceedings of First ACM SIGPLAN Workshop on Automatic Analysis of Software*, January 1997.

[4] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68, January 1999.

[5] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In S. Graf and M. Schwartzbach, editors, *Proc, 6th Internat. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 378–394. Springer, April 2000.

[6] T. Bultan. Action language: A specification language for model checking reactive systems. In *Proc. ICSE 2000*, pages 335–344, June 2000.

[7] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proc. of ASE 2001*, pages 382–386, November 2001.

[8] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[9] P. J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc., 15th Internat. Conf. on Software Engineering*, pages 315–323, May 1993.

[10] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for contructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Software and Systems Engineering*, 20(1):19–35, January 2005.

[11] C. Heitmeyer, J. Kirby, and B. Labaw. Tools for formal specification, verification, and validation of requirements. In *Proceedings of 12th Annual Conference on Computer Assurance (COMPASS '97)*, June 1997.

[12] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

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

[14] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[15] R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc., 6th ACM SIGSOFT Internat. Symp. on Foundations of Software Engineering (FSE '98)*, November 1998.

[16] R. Jeffords and C. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proc., 5th IEEE International Symposium on Requirements Engineering (RE '01)*, August 2001.

[17] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specifications of process-control systems. *IEEE Trans. on Software Engineering*, 20(9), September 1994.

[18] K. L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, Massachusetts, 1993.

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

[20] T. Yavuz-Kahveci, C. Bartzis, and T. Bultan. Action language verifier, extended. In *Proc., 17th Internat. Conf. on Computer Aided Verification (CAV 2005)*, 2005.

[21] T. Yavuz-Kahveci and T. Bultan. A symbolic manipulator for automated verification of reactive systems with heterogeneous data types. *STTT*, 5(1):15–33, November 2003.