# Applying Infinite State Model Checking and Other Analysis Techniques to Tabular Requirements Specifications of Safety-Critical Systems

Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106 USA
bultan@cs.ucsb.edu

Constance Heitmeyer
Naval Research Laboratory (Code 5546)
Washington, DC 20375 USA
heitmeyer@itd.nrl.navy.mil

## Abstract

*Although it is most often applied to finite state models, in recent years, symbolic model checking has been extended to infinite state models using symbolic representations that encode infinite sets. This paper investigates the application of an infinite state symbolic model checker called Action Language Verifier (ALV) to formal requirements specifications of safety-critical systems represented in the SCR (Software Cost Reduction) tabular notation. After reviewing the SCR method and tools, the Action Language for representing state machine models, and the ALV infinite state model checker, the paper presents experimental results of formally analyzing two SCR specifications using ALV. The application of ALV to verify or falsify (by generating counterexample behaviors) the state and transition invariants of SCR specifications and to check Disjointness and Coverage properties is described. The results of formal analysis with ALV are then compared with the results of formal analysis using techniques that have been integrated into the SCR toolset. Based on the experimental results, strengths and weaknesses of infinite state model checking with respect to other formal analysis approaches such as explicit and finite state model checking and theorem proving are discussed.*

## 1 Introduction

Because the cost of fixing software errors increases significantly if the errors are found late in development, detecting software errors as early as possible is crucial. Formal requirements languages such as RSML (Requirements State Machine Language) [37], a Statecharts variant, and SCR (Software Cost Reduction) [28, 25], a tabular language for specifying requirements, have been successfully used in specifying the required behavior of real-world, safety-critical systems such as air traffic control systems and nuclear power plants. Because formal languages such as RSML and SCR produce precise, unambiguous specifications of the required system behavior, they can expose errors before the errors creep into the software implementation. The explicit formal semantics of these languages makes it possible to use automated techniques to detect errors in requirements specifications. By using such techniques, software practitioners can correct the requirements specifications early in development when the cost of correcting errors is low.

[*] This is an extended version of a paper published in the Proceedings of the Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2006).

[†] The major part of this effort was performed while Tevfik 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 grants CCF-0341365 and CCF-0614002.

[‡] Constance Heitmeyer's research is supported by the Office of Naval Research.

In the last two decades, significant progress has been made in automated techniques for verifying finite state systems, especially in techniques based on model checking [20]. In hardware design, model checking has been successfully transferred from academic research into industrial use. Given a state machine model and a property (often expressed in temporal logic), a model checker exhaustively searches the model's state space to determine if the model satisfies the given property. Since early model checking research focused on the analysis of finite state models, model checking is known mainly as a finite state verification technique.

As an exception, verification techniques that focus on real-time systems have been successful in analyzing specifications with infinite state spaces (see, for example, [34]). These techniques, which are based on the timed automata model [1], show that verification of timed automata can be achieved by using a finite representation based on clock regions. Verification of hybrid systems has also been studied extensively (see, for example, [30]) to analyze behaviors of discrete control systems that interact with continuously changing, external environments. Research in this area typically assumes a continuous domain, thus resulting in infinite state specifications.

In recent years, model checking using symbolic representations that can encode infinite sets has been extended to analyze infinite state models outside the domain of real-time or hybrid systems [2, 23, 35, 36]. These 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) [19, 44] to the formal analysis of requirements of safety-critical systems specified in the SCR (Software Cost Reduction) tabular notation. It then compares the results of applying infinite state model checking using ALV with the results of applying the formal analysis tools and techniques of the SCR toolset [25], identifying both the strengths and weaknesses of the two approaches and discussing potential synergies among the various analysis techniques.

The remainder of the paper is organized as follows. After reviewing the SCR model and tools, Section 2 presents the two SCR specifications used in our experiments. Section 3 gives an overview of the Action Language, and Section 4 discusses the verification techniques used in ALV. Section 5 describes the application of ALV and the SCR tools to the verification of the two SCR specifications presented in Section 2. Section 6 discusses how ALV can be used for consistency checking and how its approach and performance compare with those of the SCR tools for consistency checking. Section 7 compares the infinite state model checking techniques used in ALV with the verification techniques used in the SCR toolset, and Section 8 discusses the utility of ALV in the SCR toolset. Finally, section 9 presents some conclusions.

## 2 SCR Language and Toolset

The objective of an SCR specification is to specify the required externally visible behavior of a software system as a relation on environmental quantities. In an SCR specification [25, 28], *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 clearly and concisely, the SCR language provides two types of auxiliary variables, *terms* and *mode classes*. A *term* is an auxiliary variable which makes the specification more understandable by assigning some logical expression to the variable or more concise by assigning a logical expression occurring more than once in the specification to the variable. A *mode class* is a special case of a term whose values are modes. If a monitored variable changes, the effects of the change may differ if the system is in one mode rather than another mode. Also of significance in SCR specifications are conditions and events, where a *condition* is a predicate defined on a system state, and a basic *event*, represented as `@T(c)`, indicates that condition `c` changes from false to true. The event `@F(c)` is defined by `@T(`$\neg$`c)`, and indicates that condition `c` changes from true to false. 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 $\neg c \wedge c' \wedge d$.

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

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

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

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

In SCR specifications, the monitored variables are *independent variables*, and the mode classes, terms and controlled variables are *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, and 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 a 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 $M$ and a set of conditions $c_{i,j}$. The value of variable $r$ described by the table can be defined as a formula $F_r$,

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

where $n$ is the number of modes in $M$, $p$ is the number of conditions for each mode in the table, and the value $v_j$ 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.

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 can be defined as a formula $F_{r'}$,

$$F_{r'} \equiv \left[ \bigvee_{i=1}^{n} \bigvee_{j=1}^{p} (M = m_i \wedge e_{i,j} \wedge r' = v_j) \right] \bigvee \left[ \neg (\bigvee_{i=1}^{n} \bigvee_{j=1}^{p} M = m_i \wedge e_{i,j}) \wedge r' = r \right], \tag{2}$$

where $n$, $p$, and $v_j$ are defined above. The right-hand expression in the disjunction in $(2)$ describes the *no-change* case; it states that when an event occurs that is not explicitly specified in the table, the value of variable $r$ does not change.

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 $F_{M'}$,

$$F_{M'} \equiv \left[ \bigvee_{i=1}^{n} \bigvee_{j=1}^{k_i} (M = m_i \wedge e_{i,j} \wedge M' = m_{i,j}) \right] \bigvee \left[ \neg (\bigvee_{i=1}^{n} \bigvee_{j=1}^{k_i} M = m_i \wedge e_{i,j}) \wedge M' = M \right], \qquad (3)$$

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. As in event tables, if an event occurs that is not specified in the table, the semantics of a mode transition table are that the mode does not change.

Two relations introduced in Parnas' Four Variable Model [40], NAT and REQ, define constraints on the monitored and controlled variables. NAT specifies the natural constraints on the system imposed by physical laws and the system environment. In SCR, the NAT constraints are called assumptions. REQ specifies the required system behavior as further constraints on the relation between monitored and controlled variables. REQ is written using the SCR tables and it specifies the required system behavior as constraints on the dependent variables. In this paper, we focus on verification of the REQ specifications written using SCR tables. Given a set of dependent variables, REQ is defined as the conjunction of the semantics described by each table.

An SCR specification defines the required system behavior as 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. The transition relation of an SCR specification is based on the One Input Assumption, which states that in any state transition, exactly one monitored variable changes its value. For further details of the SCR semantics, see [28].

## 2.1 Two SCR Specifications

This section provides excerpts from SCR specifications of two safety-critical software systems—a Safety Injection System (SIS) [21], which controls safety injection in a nuclear power plant, and a Cruise Control System (CCS) [25], which manages cruise control in an automobile. The specifications of both the SIS and the CCS describe how each system is required to respond to changes in the monitored variables. Both specifications use modes to capture the history of changes in the monitored variables and one or more modes to make the specification more understandable and more concise.

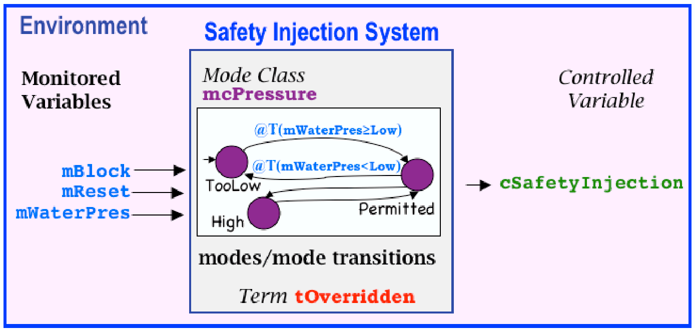

**Figure 1. SCR requirements specification of the Safety Injection System.**

**The Safety Injection System (SIS).** The SIS specification describes the requirements of the control software for a nuclear reactor's cooling system [21]. SIS monitors the water pressure of the cooling system. When water pressure drops below a certain constant value `Low`, the system starts safety injection (if it is not overridden). In the SCR specification of SIS, the monitored variables—`mBlock`, `mReset`,

**Table 3. Types and initial values of variables in the SIS specification.**

| Variable Name | Class | Type | Init. Value |
|---|---|---|---|
| mWaterPres | Monitored | yPresRange | 14 |
| mBlock | Monitored | ySwitch | Off |
| mReset | Monitored | ySwitch | On |
| tOverridden | Term | Boolean | false |
| mcPressure | Mode Class | Enumerated | TooLow |
| cSafetyInjection | Controlled | ySwitch | On |

and mWaterPres—denote the states of the block and reset switches and the water pressure reading; the mode class mcPressure indicates one of three system modes, TooLow, Permitted, and High; the Boolean term tOverridden indicates whether safety injection is overridden; and the controlled variable cSafetyInjection indicates whether safety injection is turned on. To illustrate the SCR method for specifying the required behavior of a software system, Figure 1 illustrates the relationship between the SIS inputs, represented as monitored variables; the SIS modes; and the single SIS output, represented as a controlled variable.It also gives examples of two conditioned events, those that label the transitions between TooLow and Permitted. The figure shows that when the system is in TooLow and the water pressure reading changes from Low to greater than or equal to Low, the SIS mode changes to Permitted; similarly, when SIS is in the mode Permitted and the water pressure reading changes from a value greater than or equal to Low to less than Low, the SIS mode changes to TooLow. In the SIS specification, a NAT assumption is that mWaterPres has a value between 0 and 2000, i.e., $0 \leq \texttt{mWaterPres} \leq 2000$, and that mWaterPres can change by at most one from one state to the next, i.e., $|\texttt{mWaterPres}' - \texttt{mWaterPres}| \leq 1$. An alternate NAT assumption could allow mWaterPres to change by more than one in each step, e.g., $|\texttt{mWaterPres}' - \texttt{mWaterPres}| \leq 10$.

Table 3 indicates the class, type, and initial value of each variable in the SCR specification of SIS, stating for example that the type of two monitored variables, mBlock and mReset, and the controlled variable cThrottle is ySwitch, a user-defined enumerated type with values in {On, Off}. Table 3 also indicates that, in the initial state, water pressure has the value 14, tOverridden is false, the switches mReset and cSafetyInjection are on, and the switch mBlock is off. In the SCR specification of SIS, the user-defined types ySwitch and PresRange are defined in a type dictionary (not shown).

Figure 2 contains the mode transition, event, and condition tables defining the transition relation of the SIS specification. The top table, the mode transition table, defines the transitions for the mode class mcPressure, and the middle table, the event table, the transitions for the term variable tOverridden. The first row of the mode transition table states that if the system is in mode TooLow when water pressure changes from less than Low to a value exceeding or equal to Low, the system mode changes to Permitted. The entry “Never” in the event table means that if the system is in the mode High, no event can cause tOverridden to change to true. The middle entry in the second row of the event table indicates that if the system is in either TooLow or Permitted and the user turns the Block switch on when the Reset switch is off, then the value of tOverridden in the new state is true. In the middle table, the constants Low and Permit have the values 900 and 1000. The bottom table, a condition table defining the controlled variable cSafetyInjection, describes a state invariant defining the required relation between the values of cSafetyInjection, tOverridden, and mcPressure. Because the water pressure reading can vary from 0 to 2000 and because water pressure affects the value of cSafetyInjection (by determining the current mode), analyzing the SIS specification mechanically can be challenging.

| **Current Mode** mcPressure | **Event** | **New Mode** mcPressure |
|---|---|---|
| TooLow | @T(mWaterPres ≥ Low) | Permitted |
| Permitted | @T(mWaterPres ≥ Permit) | High |
| Permitted | @T(mWaterPres < Low) | TooLow |
| High | @T(mWaterPres < Permit) | Permitted |

| **Mode** mcPressure | **Events** | |
|---|---|---|
| High | Never | @F(mcPressure = High) |
| TooLow,<br>Permitted | @T(mBlock=On)<br>WHEN mReset=Off | @T(mcPressure = High) OR<br>@T(mReset=On) |
| tOverridden′ | True | False |

| **Mode** mcPressure | **Conditions** | |
|---|---|---|
| High, Permitted | True | False |
| TooLow | tOverridden | NOT tOverridden |
| cSafetyInjection | Off | On |

**Figure 2. Three tables in the SCR specification of the SIS.**

**The Cruise Control System (CCS).** The specification of CCS describes the required behavior of a cruise control system for an automobile. CCS controls the automobile's throttle and speed (via the throttle position) based on the state of the brake, engine, ignition switch, actual and desired car speeds, and cruise control lever. The SCR specification of CCS [25] uses a mode class mcCruise to indicate the current mode of CCS—either Off, Inactive, Cruise, or Override. The monitored variables of CCS—mIgnOn, mSpeed, mBrake, mLever, and mEngRunning—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. The lever position is either off, const, release, or resume. Another monitored variable represents time. The controlled variable cThrottle represents the state of the throttle, and the term tDesiredSpeed represents the desired speed.

Figure 3 illustrates the relationship between the CCS monitored variables, the CCS modes, and the single CCS controlled variable. It also shows the two events that trigger transitions between Off and Inactive. For example, in response to the driver turning the ignition on (represented as "@T(mIgnOn)") when the system is in mode Off, the system enters the mode Inactive, and when the driver turns the ignition off (represented as "@F(mIgnOn)") when the system is not already in mode Off, the system re-enters the mode Off. The value of the term tDesiredSpeed represents the desired car speed. The value of the controlled variable cThrottle depends on the mode the system is in, the actual car speed compared to the desired automobile speed, and the length of time the cruise control system has been in the const position. The

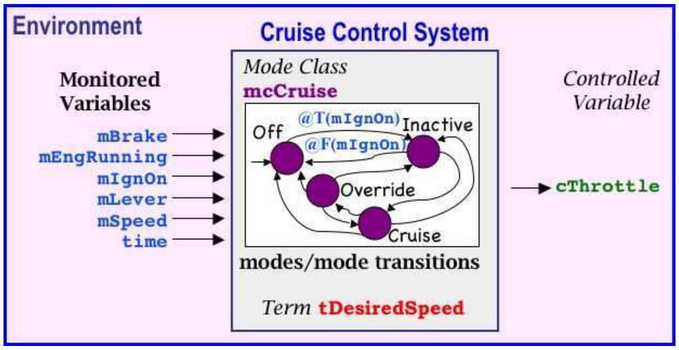

**Figure 3. SCR requirements specification of the Cruise Control System.**

**Table 4. Types and initial values of variables in the CCS specification.**

| Variable Name | Class | Type | Init. Value |
|---|---|---|---|
| mBrake | Monitored | Boolean | false |
| mEngRunning | Monitored | Boolean | false |
| mIgnOn | Monitored | Boolean | false |
| mLever | Monitored | yLever | release |
| mSpeed | Monitored | ySpeed | 0.0 |
| time | Monitored | integer | 0 |
| tDesiredSpeed | Term | ySpeed | 0.0 |
| tDURLeverEQconst | Term | integer | 0 |
| mcCruise | Mode Class | Enumerated | Off |
| cThrottle | Controlled | yThrottle | Off |

**Table 5. Excerpts from the Mode Transition Table for mcCruise**

| Current Mode<br>mcCruise | Event | New Mode<br>mcCruise |
|---|---|---|
| Off | @T(mIgnOn) | Inactive |
| Inactive | @F(mIgnOn) | Off |
| Inactive | @T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake | Cruise |
| ... | ... | ... |
| Override | @T(mLever=resume) WHEN mIgnOn AND mEngRunning AND NOT mBrake OR ... | Cruise |

amount of time that the cruise control system has been in the const position is represented by a term called tDURLeverEQconst. Table 4 gives the class, type, and initial value of each variable in the SCR specification of CCS. The user-defined types in Table 4 are ySpeed, an integer in $[0, 180]$, and yLever and yThrottle, enumerated types defined by yLever in $\{$const, release, off, resume$\}$ and yThrottle in $\{$accel, maintain, decel, off$\}$.

Table 5 contains excerpts from the mode transition table for the CCS. The first row specifies the monitored event ("@T(mIgnOn)") that causes the system to make a transition from Off to Inactive. The third row states that when the system is in the mode Inactive and the driver puts the cruise control lever in the position const when the ignition is on, the engine is running, and the brake is off, the mode changes to Cruise. 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. For the complete SCR specification of the CCS, see [25].

### 2.2 SCR Toolset

The SCR toolset integrates the formal analysis tools and techniques presented below. Each may be used alone, or in combination with others in the toolset, to analyze an SCR specification [25]. The toolset includes an editor, a consistency checker, a simulator, and several tools and techniques useful in verifying critical application properties, such as safety properties. Among the latter are Spin, an explicit state model checker; two theorem provers, Salsa and TAME; a set of abstraction techniques; and an invariant generator.

- **Specification Editor.** The editor supports the creation of SCR specifications, which consist of a set of tables and a set of dictionaries. The dictionaries specify the values of constants, user-defined types, the names, initial values, and types of variables, and sets of assertions and assumptions.

- **Consistency Checking.** The SCR consistency checker checks for syntax and type errors, circular definitions, and for violations of Disjointness and Coverage [28]. The checks other than those for Disjointness and Coverage are analogous to standard compiler checks. Checking Disjointness detects nondeterminism in the SCR tables. 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 [10], the language of the model checker Spin [31]. (Translators from SCR also exist for symbolic model checkers, such as SMV.) Using Spin, a user 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 [12] 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 [39], offers templates for specifying automata models and customized strategies which implement high-level proof steps for proving automaton properties [3]. 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 [10, 11, 27], 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 SCR toolset can apply slicing automatically. The second and third techniques perform *data abstraction* by replacing variables with large domains (such as integers) with enumerated type 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 [32, 33]. Such invariants are useful as auxiliary lemmas in proving properties of SCR specifications with TAME and Salsa. The SCR invariant generator generates invariants automatically. The user may choose which algorithms to apply and may also choose which tables (condition, event, or mode transition tables) to analyze.

## 3 Action Language

Action Language is a specification language for reactive software systems [14, 19]. An Action Language specification consists of integer, Boolean and enumerated variables, parameterized integer constants, and a set of modules and actions which are composed using synchronous and asynchronous composition operators. The state space of a module is defined by its variables and a restrict expression, its initial states by an initial expression, and its transition relation by a module expression. Action Language expressions can contain linear arithmetic operators, Boolean logic connectives, and universal and existential quantification over integers. 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.

A state of an Action Language specification assigns a value to each variable in the specification. The state space of the module is defined as all states that satisfy the *restrict expression* of that module. For example, if a variable `r` is declared as an integer in Action Language, by default its domain is the infinite set of all integers. 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 following restrict expression, on the other hand, `restrict: r < t` restricts the state space to states in which the value of variable `r` is less than the value of the variable `t`; however, in this case, variables `r` and `t` can take an infinite number of values.

The initial expression of a module defines the set of initial states of that module. For example, given two variables `r` and `t`, all of the following are valid initial expressions: `initial: r = 1 and t = 2`; `initial: r >= 1 and r <= 10 and t = r`; `initial: r >= 1 and r < t`. According to the first initial expression, in any initial state, `r` equals 1 and `t` equals 2. According to the second initial expression, in any initial state, `r` and `t` must have the same value, and they can take any value between 1 and 10, inclusive. Finally, the third initial expression states that, in any initial state, `r` must have a value greater than or equal to 1 but less than the value of `t`. In this case there is no upper bound for `r` or `t` in the initial states. However, in any initial state (or any other state), the variables cannot have a value disallowed by the restrict expression.

In Action Language, the top level module is always called the `main` module. A module expression (which starts with the name of the module) defines the transition relation of the module. A module expression can either be written as a logical expression on primed and unprimed variables, or as a composition expression. In module expressions, primed variables (called *next-state variables*) denote the next-state values and unprimed variables (called *current-state variables*) denote the current-state values. A module expression can be written using current and next-state variables and logical connectives `and`, `or`, and `not`.

A composition expression, on the other hand, is written in terms of actions and submodules of a module using asynchronous and synchronous composition operators [14]. The composition operations are not discussed here, because they are not needed in translating SCR specifications to Action Language. Since Action Language Verifier (ALV) constructs a symbolic representation from the input specification, specification of the input system with or without the composition operators is not likely to affect the verification performance, as long as specifications are semantically equivalent.

**SCR to Action Language Translation.** Given the SCR semantics in Section 2, translating SCR specifications to Action Language is straightforward. In Action Language, the module expression for the `main` module defines the transition relation of the system. To translate an SCR specification to Action Language, we generate a module expression which is a conjunction in which each conjunct corresponds to one of the tables in the SCR specification. The formula for each table is in the disjunctive form following the structure of the formulas in (1)–(3) that define the semantics of the SCR tables.

The semantics of SCR tables can be expressed in Action Language using logical expressions on current and next-state variables. Consider the the semantics of the mode transition tables in (3). The formula on the right hand side can be expressed in Action Language directly using the current and next state variables, and the logical connectives. Because the semantics of event tables have the same structure, event tables can be translated to Action Language in the same manner.

The semantics of condition tables are defined in (1) (see Section 2). These semantics define state invariants rather than transition invariants. A condition table can be translated into Action Language by making sure that the invariant defined by the condition table holds 1) in the initial states of the system and 2) in each possible next-state. This can be achieved by 1) adding the formula on the right hand side of (1) to the initial expression of the Action Language translation as a conjunct, and 2) rewriting the formula on the right hand side of (1) by replacing all current state variables with next state variables, and adding the resulting formula as a conjunct to the module expression of the Action Language translation.

Appendix A contains the translation of the SCR specification of SIS (shown in Figure 2) to Action Language. The invariants of the SCR specification are listed at the end of the Action Language specification. The Action Language specification in Appendix A starts with the declaration of the variables (lines 2–5). To restrict the values of the variable `mWaterPres` to those defined in the SCR specification, the Action Language specification contains a restrict expression (line 6). If this restrict expression is removed, ALV would interpret the variable `mWaterPres` as an integer variable which can take on any integer value; i.e., in the Action Language specification, integer variables are true integers without any implicit maximum or minimum value. Lines 8–9 in the Action Language specification correspond to the initial condition of the SCR specification.

The module expression for the `main module` contains four conjuncts: C1: lines 11–21; C2: lines 24–46; C3: lines 49–65; and C4: lines 68–71. The conjunct C1 (lines 11-21) represents the One Input Assumption of the SCR language. For the SIS specification, this means that in any transition only one of the variables `mBlock`, `mReset`, or `mWaterPres` can change value, and the conjunct C1 contains one disjunct for each one of these three possibilities. The conjunct C2 (lines 24–46) is the translation of the mode transition table for `mcPressure` shown in Figure 2. Each row of the mode transition table is represented by a disjunct in C2: lines 25–26 represent the first row, lines 28–29 represent the second row, lines 31–32 represent the third row, and lines 34–35 represent the fourth row. Finally, the last disjunct (i.e., the lines 37–45) represents the case when none of the events in the rows of the mode transition table hold. In that case, the mode class does not change value (line 45). The conjunct C3 (lines 49-64) is the translation of the event table for the term variable `tOverridden`. The translation is similar to the translation of the mode transition table discussed above. The last conjunct, conjunct C4 (lines 68–71), is the translation of the condition table for the controlled variable `cSafetyInjection`. Note that this conjunct only refers to next-state variables. This is because condition tables correspond to state invariants rather than transition invariants. In the transition relation, we need to ensure that the invariant defined by the condition table holds in each next-state. If we ensure that it also holds in the initial state, then it is guaranteed to hold for every reachable state of the system. The last part of the Action Language specification (lines 73–81) states four SIS invariants. invariant properties, ALV also supports verification of liveness properties. In fact, ALV supports all CTL temporal operators. As another example, Appendix B contains the translation of the SCR specification of CCS to Action Language.

## 4 Action Language Verifier

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 counter-example behaviors) CTL properties of Action Language specifications [19, 44]. Summarized below are the symbolic representations, fixpoint computations and automated abstraction and counter-example generation techniques used in ALV.

### 4.1 Symbolic Representations

**Composite Symbolic Representation.** ALV uses the Composite Symbolic Library [46, 45] 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.) Composite Symbolic Library integrates multiple symbolic representations: BDDs for Boolean and enumerated variables, polyhedral or automata representations for integer variables, and BDDs for bounded integer variables. Composite Symbolic Library uses an object oriented design based on an abstract interface that is inherited by every symbolic representation that is integrated to the library. This abstract interface requires the implementation of the intersection, union, complement, backward image, and forward image operations, and the subsumption, emptiness and equivalence tests for each symbolic representation. A disjunctive, composite representation, which uses the same interface, handles operations on multiple symbolic representations [15, 16]. A composite representation consists of a finite set of disjuncts where each disjunct is a conjunction that consists of one conjunct for each symbolic representation. The intersection, union and complement, operations and the subsumption, emptiness and equivalence tests are implemented on the composite representation by reducing them to operations and tests on the basic symbolic representations. The forward and backward image computations distribute over the disjunctions in the composite representation. Due to the partitioning of the variables based on the symbolic representations, the image computations also distribute over the conjunctions in the composite representations.

Composite Symbolic Library implements several heuristics for efficient manipulation of this composite representation [45]: 1) efficient operations on some symbolic representations (for example equivalence test in BDDs) are used to avoid more expensive operations on other symbolic representations (for example, equivalence tests on polyhedra), 2) the disjunctive composite representation is exploited by interleaving the computation of image computations and the subsumption checks during the fixpoint computations, and 3) the size of a composite representation is reduced by iteratively merging matching constraints and removing redundant ones. Composite Symbolic Library also contains an extended automata representation for both integer, Boolean and enumerated variables. When this alternative representation is used, use of the disjunctive composite representation is unnecessary because all variables are mapped to the same extended automata representation [7].

The object-oriented design based on the abstract symbolic interface discussed above provides the following advantages: 1) the Composite Symbolic Library and ALV can be easily extended with new symbolic representations, 2) verification procedures in ALV interact with different symbolic representation libraries using a single interface, and 3) verification procedures are polymorphic, i.e., the verifier decides which symbolic representations to use at run-time. This enables the users to choose different symbolic representations without recompiling the tool (the encoding to be used is given as a command line argument to ALV).

**Polyhedral vs. Automata Representation.** The current version of the Composite Symbolic Library uses two different symbolic representations for integer variables: 1) In the *polyhedral representation*, the values of integer variables are represented in a disjunctive form, where each disjunct corresponds to a convex polyhedron and each polyhedron corresponds to a conjunction of linear arithmetic constraints [17, 18, 24, 30, 22]. This approach is extended to full Presburger arithmetic by including divisibility constraints (represented as equality constraints with an existentially quantified variable) [38, 17, 18]. 2) In the *automata representation*, a Presburger arithmetic formula on $v$ integer variables is represented by a $v$-track automaton that accepts a string if it corresponds to a $v$-dimensional integer vector (in binary representation) that satisfies the corresponding formula [13, 41, 42, 4, 7]. Both symbolic representations are integrated into the Composite Symbolic Library by implementing the operations required by the abstract symbolic interface. The polyhedral representation is implemented by writing a wrapper around the Omega Library [38]. The automata representation is implemented using the automata package of the MONA tool [29] and based on the algorithms discussed in [4, 7, 6, 8].

**BDD Representation for Bounded Integers.** Algorithms for constructing efficient BDDs for linear arithmetic formulas on bounded integer variables are also implemented in ALV [5, 9]. The size of the BDD for a linear arithmetic formula is linear in the number of variables and the number of bits used to encode each variable, but can be exponential in the number of *and* and *or* operators. This bounded representation can be used in three scenarios: 1) all the integer variables in a specification can be bounded, 2) infinite state representations discussed above may exhaust the available resources during verification, or 3) infinite state fixpoint computations may not converge. Note that, for cases 2 and 3, verification using the bounded representation does not guarantee that the property holds in the unbounded case, i.e., the bounded representation is used to find counter-examples.

## 4.2 Fixpoint Computations

ALV, a symbolic model checker for CTL, uses the least and greatest fixpoint characterizations of CTL operators to compute the truth set of a given temporal property. It iteratively computes the fixpoints starting from the fixpoint for the innermost temporal operator in the given CTL formula. At the end, it checks if all the initial states are included in the truth set of the given CTL formula. ALV supports both the $\{\mathrm{EX}, \mathrm{EG}, \mathrm{EU}\}$ basis and the $\{\mathrm{EX}, \mathrm{EU}, \mathrm{AU}\}$ basis for CTL. ALV uses various heuristics to improve the performance of the fixpoint computations [19, 43, 44], some of which are discussed below.

**Marking Heuristic.** Since composite representation is disjunctive, during least fixpoint computations result of an iteration can include disjuncts from the previous iteration. A naive fixpoint computation approach would end up recomputing images of such disjuncts coming from the previous iteration. To reduce this problem, the disjuncts coming from earlier iterations are marked, and only the images of the disjuncts that are unmarked are computed [43]. These markings are preserved during all operations, and they are also useful during subsumption check and simplification. When the result of the current iteration is compared to the previous one, only the unmarked disjuncts are checked to determine if they are subsumed by the previous iteration. During the simplification of the composite representation (which reduces the number of disjuncts), two disjuncts are targeted for merging only if one of them is unmarked.

**Dependency Analysis.** Using the disjunctive composite representation, the transition relation of a software specification can be expressed as the disjunction of a set of atomic actions. Because the image computation distributes over disjunctions, during fixpoint computations, the image of each action can be computed separately. ALV uses a dependency analysis to determine if the output of an action can enable another action [43]. These dependencies are computed statically before the fixpoint computations start. Then, during the fixpoint computations, the results of the image computations are tagged with the labels of the actions that produce them in order to prevent redundant image computations. For example, if the output of action $a_1$ never enables action $a_2$, the output of action $a_1$ is ignored during the image computation for action $a_2$.

**Conservative Approximations, Forward Reachability and Accelerations.** For infinite state systems specified in Action Language, model checking is undecidable. Hence, ALV uses conservative approximation techniques during verification. Three outcomes are possible when one uses ALV to verify a system: 1) ALV verifies the property which means that the property is provably correct, 2) ALV generates a counter-example which means that the property is provably incorrect, and 3) ALV is unable to verify or falsify the property. The goal of the heuristics used in ALV are to minimize the third outcome.

The undecidability of the model checking problem for ALV implies that the fixpoint computations are not guaranteed to converge. ALV uses several conservative approximation heuristics to achieve convergence [18, 16, 19]: 1) truncated fixpoint computations to compute lower bounds for least fixpoints and upper bounds for greatest fixpoints, 2) widening heuristics both for polyhedra [18] and automata representations [8] to compute upper bounds for least fixpoints (and their duals to compute lower bounds for greatest fixpoints), 3) approximate reachability analysis using a forward fixpoint computation and widening heuristics, and 4) accelerations based on loop-closures which extract disjuncts from the transition relation that preserve the Boolean and enumerated variables but modify the integer variables, and then compute approximations of the transitive closures of the integer part.

**Table 6. Desired Properties of CCS**

| | |
|---|---|
| A1 | $\mathtt{mBrake} \Rightarrow \mathtt{cThrottle} = \mathtt{off}$ |
| A2 | $\mathtt{cThrottle} = \mathtt{accel} \Rightarrow \mathtt{tDesiredSpeed} > \mathtt{mSpeed}$ |
| A3 | $\mathtt{mcCruiseControl} \neq \mathtt{Off}\ \wedge\ \mathtt{mEngRunning} \Rightarrow \mathtt{IgnOn}$ |
| A4 | $\mathtt{cThrottle} \neq \mathtt{off}\ \wedge\ \mathtt{mEngRunning} \Rightarrow \mathtt{IgnOn}$ |
| A5 | $\mathtt{mSpeed}' = \mathtt{mSpeed} \Rightarrow \mathtt{cThrottle}' \neq \mathtt{accel}$ |
| A6 | $\neg\mathtt{mEngRunning}\ \wedge\ \mathtt{mEngRunning}' \Rightarrow \mathtt{cThrottle}' = \mathtt{off}$ |
| A7 | $\mathtt{mcCruiseControl} = \mathtt{Off} \Leftrightarrow \neg\mathtt{mIgnOn}$ |
| A8 | $\mathtt{cThrottle} = \mathtt{accel} \Rightarrow (\mathtt{tDesiredSpeed} > \mathtt{mSpeed} \vee \mathtt{tDURLeverEQconst} > 500)$ |
| A9 | $\mathtt{tDesiredSpeed}' \neq \mathtt{tDesiredSpeed} \Rightarrow \mathtt{mcCruiseControl}' = \mathtt{Cruise}$<br>$\wedge\ (\mathtt{mcCruiseControl} = \mathtt{Cruise}\ \vee\ \mathtt{mcCruiseControl} = \mathtt{Inactive})$ |
| A10 | $\mathtt{tDesiredSpeed} = \mathtt{mSpeed} \Rightarrow (\mathtt{mcCruiseControl} \neq \mathtt{Override} \vee \mathtt{mLever} \neq \mathtt{release}\ \vee\ \mathtt{cThrottle} = \mathtt{off})$ |
| A11 | $\mathtt{mSpeed}' = \mathtt{mSpeed}\ \wedge\ \mathtt{tDesiredSpeed} = \mathtt{mSpeed} \Rightarrow \mathtt{tDesiredSpeed}' = \mathtt{tDesiredSpeed}$ |

### 4.3 Counterexample Generation for Invariants

To generate a counterexample for an invariant, ALV negates the property and generates a witness for the negated property. Note that the negation of an invariant property corresponds to a reachability query and can be witnessed by a single execution path. ALV first computes the truth set of the negated temporal property using the backward fixpoint computations. To generate counterexamples, ALV stores the results of all fixpoint iterations during the backward-fixpoint computations. When the truth set of the negated temporal property is computed, the intersection of the result with the initial system states is computed. If the intersection is not empty, ALV picks an initial state that is in the intersection. This state becomes the initial state of the counterexample path. Then, ALV reverses the order of the iterations of the fixpoints that have been stored, and generates a counterexample path by traversing the fixpoint iterations starting from the initial state. Note that, in some cases the counterexample path may be infinite, and there may not be a finite counterexample path. In such cases, ALV generates a prefix of the counterexample path. ALV is guaranteed to generate the shortest counterexample path for an invariant.

## 5 Analyzing Properties of the SCR Specifications

The properties listed in Table 6 and Table 7 are critical invariants expressing the expected behaviors of the CCS and SIS specifications, respectively. We obtained these properties from earlier studies on verification of CCS and SIS specifications [25, 11]. The properties were selected because they are representative of the kinds of properties of interest in embedded systems, the class of software systems for which the SCR method was designed. All of these properties are safety properties. In our experience, liveness properties are rarely of interest in the requirements specifications of embedded systems.

Table 6 lists eleven properties for evaluating the CCS specification. Because properties A1–A4, A7, A8, and A10 are defined on a single state, each is a state invariant. In contrast, properties A5, A6, A9 and A11 are transition invariants since each is defined in terms of both the current state and the next state. Table 7 lists four properties for evaluating the SIS specification, each a state invariant. Section 5.1 describes how ALV was used to analyze the CCS and SIS specifications for these properties and the resources ALV needed to perform the analysis. For purposes of comparison, Section 5.2 describes the analysis of these same properties with tools in the SCR toolset along with the resources required. Experiments were conducted on Linux machines with 2.8 GHertz Pentium 4 processors and 2 GBytes of main memory.

**Table 7. Desired Properties of SIS**

| | |
|---|---|
| S1 | $\texttt{tOverridden} \Rightarrow \texttt{mcPressure} \neq \texttt{High} \wedge \texttt{mReset} = \texttt{Off}$ |
| S2 | $\texttt{mReset} = \texttt{On} \wedge \texttt{mcPressure} \neq \texttt{High} \Rightarrow \neg\texttt{tOverridden} \wedge \texttt{cSafetyInjection} = \texttt{On}$ |
| S3 | $\texttt{mReset} = \texttt{On} \wedge \texttt{mcPressure} \neq \texttt{High} \Rightarrow \neg\texttt{tOverridden}$ |
| S4 | $\texttt{mReset} = \texttt{On} \wedge \texttt{mcPressure} = \texttt{TooLow} \Rightarrow \texttt{cSafetyInjection} = \texttt{On}$ |

**Table 8. Results of verifying the CCS properties with ALV.**

| 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 |

### 5.1 Analyzing Properties with ALV

**Analyzing the CCS properties with ALV.** The representation of the CCS in ALV consists of 13 Boolean and 4 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).

Because properties A1–A4, A7, A8, and A10, in Table 6 are desired state invariants, each can be expressed using the CTL temporal operator AG. In contrast, each of the properties A5, A6, A9 and A11 is a transition invariant. Transition invariants 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 (see Appendix B):

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

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 8 lists the time ALV required to verify the nine true properties in Table 6. 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. The fact that the memory consumption is identical for verification of each property shows that the memory consumption is dominated by the size of the transition relation, not the size of the fixpoint iterations during verification. Since the transition relation is the same for each property, the memory consumption does not change. Note that the transition relation can be constructed once, and then the conjunction of all nine properties can be verified as one verification task. Using this approach, ALV verified all properties in 1.65 sec. (which is less than the sum, 4.05 sec., of the times shown in Table 8) 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. Fixpoints for all properties listed in Table 8 converged directly with no approximations.

**Table 9. Results of verifying the SIS properties with ALV.**

| 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 |
| S4 (w) | 1.15 sec | 2.5 MB | 12 |

For all results listed in Table 8, 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 the BDD representation. The transition relation of the CCS using this representation consists 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 8 are A2 and A5. ALV shows that property A5 is false by generating a counterexample. Fixpoint computations for generating the counterexample, which has length 6, required 11 iterations and took 42.91 sec. and used 2.6 MB of memory. Counterexample generation was done using the automata representation for all the variables. The automata representation for 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 [7]. 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 9. Counterexample generation for property A2 using the automata representation for all the variables required 32.86 sec. and 3.0 MB of memory.

**Analyzing the SIS properties with ALV.** 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 9 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.

The third row of Table 9 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. In order to reduce the number of fixpoint iterations we used ALV's widening heuristic. Using the widening heuristic the fixpoint computations converge in 12 iterations to an upper approximation of the fixpoint in 1.15 sec. using 2.5 MB memory. Moreover, the approximation computed by the widening heuristic is strong enough to prove the property.

**Table 10. State invariants automatically generated from the SIS specification**

| Name | Invariant | Time |
|---|---|---|
| J1 | $\texttt{cSafetyInjection} = \texttt{On} \Leftrightarrow \texttt{NOT}\ \texttt{tOverrridden}\ \wedge\ \texttt{mcPressure} = \texttt{TooLow}$ | 0.17 sec |
| J2 | $\texttt{cSafetyInjection} = \texttt{Off} \Rightarrow \texttt{mcPressure} = \texttt{Permitted} \vee$ $\texttt{mcPressure} = \texttt{Permitted}\ \vee\ (\texttt{tOverrridden}\ \wedge\ \texttt{mcPressure} = \texttt{TooLow})$ | |
| J3 | $\texttt{tOverrridden} \Rightarrow \texttt{mcPressure} \neq \texttt{High}\ \texttt{AND}\ \texttt{mReset} = \texttt{Off}$ | 1.46 sec |

**Table 11. State invariants automatically generated from the CCS specification**

| Name | Invariant | Time |
|---|---|---|
| I1 | $\texttt{mcCruise} = \texttt{Off} \Rightarrow \texttt{NOT}\ \texttt{mIgnOn}$ | 2.20 sec |
| I2 | $\texttt{mcCruise} = \texttt{Inactive} \Rightarrow \texttt{mIgnOn}$ | |
| I3 | $\texttt{mcCruise} = \texttt{Cruise} \Rightarrow \texttt{mIgnOn}\ \texttt{AND}\ \texttt{mEngRunning}$ $\wedge\ \texttt{NOT}\ \texttt{mBrake}\ \texttt{AND}\ \texttt{mLever} \neq \texttt{off}$ | |
| I4 | $\texttt{mcCruise} = \texttt{Override} \Rightarrow \texttt{mIgnOn}\ \texttt{AND}\ \texttt{mEngRunning}$ | |
| I5 | $\texttt{cThrottle} = \texttt{off} \Leftrightarrow \texttt{mcCruise} \neq \texttt{Cruise}$ | 0.21 sec |
| I6 | $\texttt{cThrottle} = \texttt{accel} \Leftrightarrow \texttt{mcCruise} = \texttt{Cruise}\ \texttt{AND}$ $(\texttt{tDesiredSpeed} - \texttt{kTolerance} > \texttt{mSpeed}$ $\texttt{OR}\ \texttt{tDURLeverEQconst} > \texttt{kStartIncr})$ | |

In analyzing S2, ALV generates a counterexample to show that S2 does not hold. The fixpoint computations for generating the counterexample required 1,104 iterations, and the counterexample has length 886. Using the automata representation, ALV generates the counterexample in 25.71 seconds using 72.3 MB of memory. To reduce the resources needed to analyze S2 and the length of the counterexample for S2, we constructed a parameterized CCS specification from the original one as follows. In the original specification of SIS, the three modes of the Pressure mode class are defined based on two constants, $\texttt{Low} = 900$ and $\texttt{Permit} = 1000$. Thus, any verification of this specification holds for these concrete values only. In our parameterized SIS specification, the constants Low and Permit are declared as parameterized constants, where $0 \leq \texttt{Low} < \texttt{Permit}$; i.e., Low and Permit are constants with unknown values but Low has any nonnegative value less than Permit. A counterexample for 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.

### 5.2 Verification with the SCR Tools

Prior to analyzing the SIS and CCS properties, we first executed the SCR invariant generator to automatically construct state invariants from the SIS and CCS specifications. Such invariants are useful as lemmas in verifying properties using a theorem prover. After listing several examples of the generated state invariants, this section presents the results of formal analysis of the two specifications with TAME, Salsa, and Spin.

**Automatically Generated Invariants.** Table 10 lists three state invariants, J1-J3, generated automatically by the SCR invariant generator from the SIS specification. The invariant generator constructed invariants J1 and J2 from the condition table defining cSafetyInjection and invariant J3 from the event table defining tOverridden. The time required by the invariant generator to construct these invariants is shown in the right-most column of Table 10.

Table 11 lists six state invariants automatically generated from the CCS specification. The invariant generator constructed invariants I1-I4 from the mode transition table defining mcCruise and invariants, I5 and I6 from the condition table defining cThrottle. (It also generated two additional invariants for cThrottle not shown in the table.) The time required by the invariant generator to construct these invariants is shown in the right-most column of Table 11.

**Table 12. TAME overhead for SIS and CCS**

| Description | SIS Overhead | CCS Overhead |
|---|---|---|
| Restoring library theories | 0.51 sec | 0.48 sec |
| Parsing theories | 0.30 sec | 0.62 sec |
| Type checking theories | 3.27 sec | 5.53 sec |
| Total Overhead | 4.08 sec | 6.63 sec |

**Table 13. Results of analyzing the SIS properties with TAME**

| Property | Time | Lemmas Needed |
|---|---|---|
| S1 | 2.70 sec | none |
| S3 | 2.14 sec | none |
| S4 | 4.00 sec | J3 |

**Table 14. Results of analyzing the CCS properties with TAME**

| Property | Time | Lemmas Needed |
|---|---|---|
| A1 | 7.42 sec | I3 |
| A3 | 4.29 sec | I3, I4, I5 |
| A4 | 7.94 sec | I3 |
| A6 | 1.91 sec | I3 |
| A7 | 5.14 sec | none |
| A8 | 5.20 sec | none |
| A9 | 4.82 sec | none |
| A10 | 6.06 sec | none |
| A11 | 2.03 sec | none |

**Verification with TAME.** TAME, a front-end to PVS, provides templates for specifying state machine models and specialized strategies which include high-level proof steps for proving invariant properties. Initially designed for analyzing timed automata, TAME has been adapted to SCR by an automatic SCR-to-TAME translator. In analyzing an SCR specification with TAME, some overhead is incurred. Table 12 describes the overhead TAME incurred prior to analyzing the SIS and CCS specifications.

Table 13 shows the time required by TAME to verify S1, S3, and S4 and indicates the invariant lemmas, if any, needed to complete the proofs. Given the TAME specification of SIS and the four properties listed in Table 7, TAME is able to verify properties S1 and S3 automatically. Once the automatically generated invariants in Table 10 are provided, TAME is also able to prove property S4. Examining the proof script provided by TAME shows that invariant J3 was used as an auxiliary invariant lemma in the proof of S4. With and without the automatically generated invariants for SIS, TAME was unable to verify property S2, a property that ALV has shown to be false. For S2, TAME produces five dead-ends, three of which cannot be proved using the generated invariants.*

Table 14 shows the time required by TAME to verify A1, A3, A4, and A6-A11 and indicates the invariant lemmas, if any, needed to complete the proofs.Given the SCR specification of CCS and the properties listed in Table 6, TAME proved A7-A11 automatically without invariants [25]. Completing the proofs of the remaining properties—A1, A3, A4, and A6—required the use of one or more of the automatically generated invariants as auxiliary lemmas [25]. Although almost all branches in these proofs required only a single invariant, namely, I3, completing the proof of property A3 required three invariants, I2, I3, and I4, thus demonstrating that combinations of invariants can be useful. As demonstrated by ALV, the remaining two assertions, A2 and A5, are false. For A5, TAME produces 26 dead ends. For A2, TAME produces two dead ends. None of these dead ends can be proven using the generated invariants.

*The number of dead ends TAME produces varies depending on the order of simplification. TAME varies the order of simplification in a search for best average performance. Thus, the dead end data reported here differ from those reported in [25].

**Table 15. Results of analyzing the SIS properties with Salsa.**

| Property | Time (no invars.) | Verified? | Time (with invars.) | Verified? |
|---|---|---|---|---|
| S1 | 0.10 sec | yes | - | - |
| S2 | 0.13 sec | no | 0.14 sec | no |
| S3 | 0.10 sec | yes | - | - |
| S4 | 0.17 sec | no | 0.12 sec | yes |

**Table 16. Results of analyzing the CCS properties with Salsa**

| Property | Time (no invars.) | Verified? | Time (invars.) | Verified? |
|---|---|---|---|---|
| A1 | 0.19 sec | no | 0.10 sec | yes |
| A2 | 0.38 sec | no | 0.56 sec | no |
| A3 | 0.15 sec | no | 0.21 sec | yes |
| A4 | 0.25 sec | no | 0.33 sec | yes |
| A5 | 0.35 sec | no | 0.42 sec | no |
| A6 | 0.27 sec | no | 0.35 sec | yes |
| A7 | 0.14 sec | yes | - | - |
| A8 | 0.35 sec | yes | - | - |
| A9 | 0.22 sec | yes | - | - |
| A10 | 0.23 sec | yes | - | - |
| A11 | 0.23 sec | yes | - | - |

**Verification with Salsa.** To analyze application properties, Salsa carries out an induction proof, treating all automatically generated invariants as axioms. Salsa also applies slicing to remove all variables irrelevant to a property's validity from the specification. When a proof fails, Salsa returns a state pair. Because the transition represented by the state pair may be unreachable, the user must show either that the property is false by finding a scenario which violates the property or that the property is true by applying additional invariants to complete the proof. The SCR toolset automatically translates SCR specifications into SAL (SCR Abstract Language), the language of Salsa. Verification with Salsa was completely automated without any manual abstraction of the specification. Using Salsa produced the same results as TAME but with faster execution time.

In analyzing the SIS specification for the properties in Table 7, Salsa was able to verify S1 and S3 without invariants and S4 by including the automatically generated invariants, listed in Table 10, as assumptions in the SIS specification. Salsa was unable to verify S2, even when invariants were added. As analysis with ALV has shown, S2 is false. However, in the case of Salsa, the user has the responsibility of finding a counterexample using the state pair returned by Salsa. Table 15 shows the time required by Salsa to analyze the SIS properties. For S2 and S4, the analysis times without invariants and with invariants are shown. Note that without invariants, Salsa could not verify S4; when the invariants were added to the SIS specification, Salsa was able to verify S4.

In analyzing the CCS specification for the properties in Table 6, Salsa was able, as expected, to verify nine assertions—A1, A3, A4, and A6-A11—and produced a state pair for each of the false properties, A2 and A5. Properties A7-A11 were proven directly without invariants. As illustrated by the analysis results using TAME, invariants are needed to verify four CCS properties  A1, A3, A4, and A6. By including all of the automatically generated invariants in its analysis, Salsa established the validity of all four of these properties. Table 16 shows the times required by Salsa to verify the nine true CCS properties and to suggest that properties A2 and A5 are false.

**Table 17. Results of analyzing the SIS properties with Spin.**

| Property | Time | Memory | Valid? |
|---|---|---|---|
| S1 | 0.22 sec | 12.26 MB | yes |
| S2 (BFS) | 0.12 sec | 7.24 MB | no |
| S3 | 0.22 sec | 12.26 MB | yes |
| S4 | 0.22 sec | 12.26 MB | yes |

**Verification with Spin.** The SCR toolset automatically translates an SCR specification into Promela, the language of Spin, representing the property to be proved as a Promela `assert` statement. One deficiency of Spin is that it cannot handle assumptions, so each assumption in an SCR specification must be inserted by hand into the Promela code. This is achieved by restricting the inputs (i.e., the monitored variable changes) in the Promela model. For example, the assumption that time is nondecreasing is encoded by requiring every new value of time to be no less than the old value.

We ran Spin[†] with 20 MB of memory[‡] and its default parameters to check the four properties in Table 7. Spin verified three properties, S1, S3 and S4, and detected a violation of a fourth property, S2. Table 17 shows the time and memory Spin required to analyze each of the four SIS properties.properties, Spin examined 204,032 states. For the invalid property, it examined 42,993 states and returned a counterexample of length 29,256. Rerunning Spin with its breadth-first search option returned a shorter counterexample of length 5316. This counterexample corresponds to an SCR counterexample of length 886—i.e., 886 changes are required for the monitored variable `WaterPres` to change, incremented by one unit each time, from its initial value 14 to the threshold value 900. Since the SIS specification is deterministic, only changes of the monitored variables are needed in the SCR counterexample. (This SCR counterexample is constructed automatically by the SCR toolset. The sequence of monitored event changes can then be executed using the SCR simulator to validate the counterexample. This is especially valuable when Spin is applied to an abstraction of the original SCR specification.)

The results of applying Spin to the eleven CCS properties were previously reported in [25]. Due to the large numbers and wide numerical ranges in the CCS specification, applying Spin without reducing the state space of the specification leads to memory overruns due to state explosion. Combatting this state explosion required careful thought and significant manual effort to produce abstractions of the CCS specification with much smaller state spaces. Running Spin on these abstractions ultimately led to the verification of eight properties, partial verification of a ninth property, and the detection of violations in the remaining two properties. For details, see [25]. Because the previous effort did not record the time and amount of memory Spin needed to perform its analyses, we have re-applied Spin to four CCS properties (the ones requiring the least amount of manual effort to construct abstractions). When run on the original CCS specification, Spin was unable to verify (or falsify) any of these properties without running out of memory. To make model checking with Spin feasible, we constructed abstractions of the original specification, automatically in the case of properties A3 and A7 and manually in the case of A2 and A5.

In the case of A3 and A7, we first applied slicing, which the SCR toolset performs automatically, to eliminate all variables irrelevant to the validity of the property from the CCS specification. In each case, the resulting abstract specification, a sound abstraction of the original, has a much smaller state space than the original. When executed on this abstraction, Spin was able to quickly verify both A3 and A7. To check properties A2 and A5, we manually constructed an abstract specification that is sound for refutation, but not for verification. To do so, we changed the units of time from ms to tenths of seconds and reduced the maximum car speed from 180 to 10 mph. Running SPIN on the CCS abstraction resulted in counterexamples

[†] Spin 4.2.7 released June 23, 2006.

[‡] Spin needs large amounts of memory to analyze the SIS specification because the water pressure variable has a large number of possible values and because the variable value can only change by one at each step.

**Table 18. Results of analyzing four CCS properties with Spin.**

| Property | Time | Memory | Valid? |
|---|---|---|---|
| A2 (BFS) | 1.53 sec | 59.851 MB | no |
| A3 | 0.00 sec | 2.622 MB | yes |
| A5 (BFS) | 0.16 sec | 8.958 MB | no |
| A7 | 0.28 sec | 2.622 MB | yes |

for both A2 and A5. Applying Spin with its default parameters to A2 detected an error but required 9,999 MB of memory and 0.71 sec of elapsed time and the checking of 553,645 states. Re-running Spin using the breadth-first search option constructed a shorter counterexample but required somewhat more memory and time, 1.53 sec and 59.851 MB and a search of 766,971 states.[§] The counterexample to A2 that Spin constructed had length 54, which translates into eight SCR steps (i.e., eight changes in monitored variables). Applying Spin with its default parameters to A5 also required 9,999 MB of memory but only 0.08 sec of elapsed time and the checking of 45,068 states. As expected, rerunning Spin using the breadth-first search option constructed a shorter counterexample but required about twice the memory and time, 0.16 sec and 8.958 MB of memory and a search of 89,490 states. The counterexample to A5 that Spin constructed had length 35, which translates into six SCR steps. Table 18 shows the time and memory Spin required to analyze each of the four CCS properties.

## 6 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 [28]. 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:

$$\forall i, 1 \leq i \leq n, \ \forall j, k, 1 \leq j, k \leq p : j \neq k \Rightarrow c_{i,j} \wedge c_{i,k} \equiv \mathbf{false}, \tag{4}$$

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 (4) 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. This can be done by checking that the disjunction of a set of conditions evaluates to $\mathbf{true}$. Based on the semantics of the SCR tables given in Section 2, the Coverage check for a condition table analyzes the following formula:

$$\forall i, 1 \leq i \leq n : \bigvee_{j=1}^{p} c_{i,j} \equiv \mathbf{true}. \tag{5}$$

### 6.1 Consistency Checking with ALV

Because ALV is a CTL model checker, using ALV to check Disjointness and Coverage requires the reformulation of these properties as CTL formulas.

[§] As stated in Section 5.1, the reason Spin needs so much memory is because the time variable in CCS, which starts at zero and increases by at most one at each step, must reach a value that exceeds a large constant. This requires Spin to store hundreds of states.

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

$$\mathrm{EX}(r = v_r \wedge \bigwedge_{\hat{r} \in R} (\hat{r} = v_{\hat{r}})) \Rightarrow \mathrm{AX}(\bigwedge_{\hat{r} \in R} (\hat{r} = v_{\hat{r}}) \Rightarrow r = v_r). \tag{6}$$

In (6), $R$ 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 (6) has been proven to be equivalent to the original SCR formulation in (4).

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 can 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:

$$F_r \equiv \left[ \bigvee_{i=1}^{n} \bigvee_{j=1}^{p} (M' = m_i \wedge c'_{i,j} \wedge r' = v_{i,j} \wedge b'_r = \mathbf{true}) \right] \bigvee \left[ \neg(\bigvee_{i=1}^{n} \bigvee_{j=1}^{p} (M' = m_i \wedge c'_{i,j})) \wedge b'_r = \mathbf{false} \right] \tag{7}$$

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 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 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.

### 6.2 Consistency Checking with the SCR Toolset

In addition to checking for Disjointness and Coverage, the SCR consistency checker also checks for syntax and type errors, circular definitions, undefined variables, etc. In using the consistency checker, the user may perform checks individually on selected tables or may select "All Checks" to perform all of the checks, including Disjointness and Coverage checks, on the entire SCR specification. The SCR consistency checker uses the formulations in (4) and (5) to check the SCR tables for Disjointness and Coverage.

The second and third columns of Table 19 show the time the SCR consistency checker required to perform various checks of the SIS and CCS specifications. In Table 19, "CC" refers to the SCR consistency checker; "Type+" includes syntax and type checking, checking for circular dependencies and other minor checks; "Coverage" and "Disjointness" refer to coverage and disjointness checking when "Type+" has already been run; "All-checks" performs all checks at one time; and "Coverage+" and "Disjointness+" refer to coverage and disjointness checking when "Type+" has not already been run. (Prior to checking a specification for Disjointness and Coverage, the consistency checker ensures that the specification is syntax and type correct. If these checks have not been run, the checker automatically executes them.) A limitation of the consistency checker is that it sometimes cannot complete a Coverage or Disjointness check. In checking the CCS specification for disjointness, for example, the checker could not decide whether the table defining `cThrottle` satisfies disjointness. The problem is the combination of numbers and arithmetic in the definition of `cThrottle`. Because the consistency checker was not designed to analyze predicates containing

**Table 19. Results of consistency checking with the SCR consistency checker and with Salsa.**

| Check | SIS (CC) | CCS (CC) | SIS (Salsa) | CCS (Salsa) |
|---|---|---|---|---|
| Type+ | 0.35 sec | 0.40 sec | - | - |
| Coverage | 0.26 sec | 0.27 sec | 0.10 sec | 0.10 sec |
| Disjointness | 0.19 sec | 0.39 sec | 0.15 sec | 0.26 sec |
| All-Checks | 1.16 sec | 0.82 sec | - | - |
| Coverage+ | 0.48 sec | 0.58 sec | - | - |
| Disjointness+ | 0.55 sec | 0.75 sec | - | - |

**Table 20. Comparing Verification Results for SIS**

| Tool | Verif. Succ.? | Invars. Req.? | Abstraction Method | Average Time | Shortest Time | Longest Time | Memory Req. Range |
|---|---|---|---|---|---|---|---|
| ALV | all | no | none | 0.40 sec | 0.03 sec | 1.15 sec | 1.2 MB–2.5 MB |
| TAME | all | yes (1) | none | 2.95 sec | 2.14 sec | 4.00 sec | - |
| Salsa | all | yes (1) | slicing | 0.12 sec | 0.10 sec | 0.17 sec | - |
| Spin | all | no | none | 0.22 sec | 0.22 sec | 0.22 sec | 2.6 MB |

complex numerical constraints, it cannot reason about predicates used in the definition of `cThrottle`, e.g., “`tDesiredSpeed + kTolerance > mSpeed`” and “`DesiredSpeed + kTolerance < mSpeed`.” To overcome this limitation of the consistency checker, Salsa was developed.

Salsa can either analyze an application property as described in Section 5.2, or it can analyze a specification for Coverage and Disjointness. The two right-most columns of Table 19 indicate the time required by Salsa to perform the Coverage and Disjointness checks for the SIS and CCS specifications. Because Salsa assumes that the specification to be analyzed is syntax and type correct, these performance times do not include the time needed to perform checks such as “Type+”.

## 7 Comparing ALV Results with Results Produced by the SCR Tools

The combined set of SIS and CCS properties contains twelve properties that are valid and three properties that are invalid. Section 7.1 compares the results of analyzing the twelve valid properties with ALV, TAME, Salsa, and Spin; Section 7.2 compares the results of the analyses using these same tools which detected property violations. Section 7.3 compares the results of using ALV, the SCR Consistency Checker, and Salsa to perform coverage and disjointness checks. The remainder of this section discusses the differences among the verification techniques used by these tools.

### 7.1 Comparing the Results of Verification

Tables 20 and 21, which compare the results of analyzing the twelve valid properties with the four tools, indicate whether verification succeeded; whether the verification required invariants and, if so, the number of properties whose proofs required invariants; the abstraction method that was applied (if any); the average, shortest, and longest analysis time; and the required memory. These tables show that ALV, Salsa, and TAME were each able to verify all twelve properties. Due to the large state space of the CCS specification, Spin's success in verifying the valid CCS properties was limited. For both specifications, Salsa, ALV, and Spin (at least for the five properties that it verified) had similar execution times, whereas TAME had significantly longer execution times. To verify five of the properties, Salsa and TAME required invariants. Because the invariants were automatically generated by the SCR invariant generator, we do not regard this as a serious limitation. (Note that the verification times reported in Tables 20 and 21 for Salsa and TAME do not include the invariant generation time.) In the Salsa and TAME analyses, slicing was used to reduce the CCS

**Table 21. Comparing Verification Results for CCS**

| Tool | Verif. Succ.? | Invars. Req.? | Abstraction Method | Average Time | Shortest Time | Longest Time | Memory Req. |
|---|---|---|---|---|---|---|---|
| ALV | all | no | none | 0.45 sec | 0.32 sec | 0.76 sec | 23.2 MB |
| TAME | all | yes (4) | slicing | 4.98 sec | 1.91 sec | 7.94 sec | - |
| Salsa | all | yes (4) | slicing | 0.22 sec | 0.14 sec | 0.35 sec | - |
| Spin | limited | no | slicing | 0.33 sec | 0.28 sec | 0.37 sec | 2.6 MB |

**Table 22. Comparing Results for the Invalid SIS Property**

| Tool | Error Detected? | Abstr. Method | CE Generated? | Shortest CE? | Analysis Time | Memory Required |
|---|---|---|---|---|---|---|
| ALV | yes | none | yes | yes | 25.71 sec | 72.3 MB |
| TAME | inconcl. | none | - | - | - | - |
| Salsa | inconcl. | none | - | - | - | - |
| Spin | yes | none | yes | yes | 0.12 sec | 7.2 MB |

**Table 23. Comparing CCS Results for the Invalid Properties**

| Tool | Error Detected? | Abstr. Method | CE Generated? | Shortest CE? | Average Time | Shortest Time | Longest Time | Memory Req. Range |
|---|---|---|---|---|---|---|---|---|
| ALV | yes | manual (1) | yes | yes | 37.89 sec | 32.86 sec | 42.91 sec | 2.6–3.0 MB |
| TAME | inconcl. | slicing | - | - | - | - | - | - |
| Salsa | inconcl. | slicing | - | - | - | - | - | - |
| Spin | yes | manual (2) | yes | yes | 0.85 sec | 0.16 sec | 1.53 sec | 9.0–59.9 MB |

state space. Because the SCR toolset performs slicing automatically, we do not regard the use of slicing as a significant limitation. In the ALV analysis of S4 widening technique was used to achieve faster convergence.

Although Spin verified the three true SIS properties without abstraction, the large numbers in the CCS specification limited Spin's effectiveness in analyzing the nine true CCS properties. As noted in Section 5.2, verifying two of these properties with Spin is straightforward if slicing is applied. However, verifying the remaining seven properties with Spin requires significant manual abstraction which was not repeated in this effort. Hence, the times shown in Table 21 for Spin are those obtained in analyzing only two of the nine properties.

### 7.2 Comparing the Results of Detecting Property Violations

Tables 22 and 23, which compare the results of analyzing the three invalid properties with the four tools, indicate either that an error was detected or that the results of analysis were inconclusive; the abstraction method that was applied (if any); whether a counterexample (CE) was generated and, if so, whether it was the shortest counterexample; the time required for the analysis; and the amount of required memory. The tables show that both ALV and Spin detected violations of all three properties—S2, A2, and A5—and that Spin's analysis times were significantly better than ALV's. For the one invalid SIS property, Spin required less memory than ALV, whereas for the two invalid CCS properties, Spin required more memory than ALV. In the case of A2 and A5, the numbers and numerical ranges in the CCS specification needed to be reduced prior to analysis with Spin. In the case of A2, the size of one constant needed to be reduced prior to analysis with ALV. The results of analyzing invalid properties with TAME and Salsa were inconclusive because, in each case, either additional invariant lemmas were needed to complete the proof, or the property was false. For each tool, the user is required to inspect the tool's results of the analysis to determine the correct alternative; if it is determined that the property is false, the user, not the tool, must construct a counterexample.

**Table 24. Performance Times for Consistency Checking**

| Tool | SIS Coverage | SIS Disjointness | CCS Coverage | CCS Disjointness |
|---|---|---|---|---|
| ALV | 0.06 sec | 2.07 sec | 0.02 sec | 8.07 sec |
| SCR CC | 0.26 sec | 0.19 sec | 0.27 sec | 0.39 sec |
| Salsa | 0.10 sec | 0.15 sec | 0.10 sec | 0.26 sec |

## 7.3 Comparing the Results of Consistency Checking

Table 24 presents the time required by ALV, the SCR consistency checker, and Salsa to perform coverage and disjointness checks of the SIS and CCS specifications. In Section 6, we showed that all of these checks were successful, except the disjointness check by the SCR Consistency Checker of the table defining `cThrottle`. Although ALV and Salsa completed all of the required checks, Salsa's performance times were somewhat better than ALV's for disjointness checking. The three tools had comparable times for coverage checking.

## 7.4 Symbolic Representations

**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 s and 6.6 MB for the polyhedral representation and 50.78 s and 2.7 MB for the automata representation. In verifying the Disjointness property for SIS, the polyhedral representation required 2.07 s and 257.8 MB, while the automata representation required 1.13 s 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.

**Abstraction vs. Symbolic Representation.** In [11, 27], 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 ALV's performance. For example, the SCR dependency analyzer determines that property A3 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 above, using ALV, we were able to verify the SIS specification without restricting the integer variables or unspecified constants to finite domains and without using any abstractions.

### 7.5 Model Checking Techniques

**Explicit State vs. Symbolic Model Checking.** Spin has been used to check properties of many SCR specifications (see, e.g., [27]) other than the SIS and CCS. The default for explicit state model checkers such as Spin is to use efficient depth-first search algorithms to find property violations. The first detected violation causes Spin to halt its search and to construct a counterexample. If Spin exhausts the state space without detecting a violation, it reports that the property is verified. As shown by our experiments, for systems with large state spaces, Spin is more efficient in finding errors than in verifying correctness. Further, when a violation is detected, Spin's breath-first search option is capable of finding the shortest counterexample.

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 violation it finds and then constructs a counterexample.

If a specification with a large state space has no errors (or even if it has errors), Spin can run out of memory before it searches the complete state space. One cause of state space explosion is variables with large domains, such as integers. Since Spin performs state exploration starting from the initial states, the size of the *reachable* state space is important. A specification with integer variables can easily be 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 may be difficult to verify with Spin. Such cases sometimes occur in SCR specifications since SCR is a requirements specification language, and the input variables in requirements specifications may have a range of initial values rather than a single initial value.

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 [10], 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 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., $0 \leq \texttt{WaterPres} < 900$. ALV's performance for this generalized specification was identical to that reported in Section 5. 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 analyze 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.

The SIS specification verified in Section 5 with ALV and the other three tools has a finite state space in which `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. By declaring that `WaterPres` can have any nonnegative value, we converted the SIS specification to an infinite state specification. ALV verified properties S1 and S3 on this infinite state specification in exactly the same time as for the finite case reported in Section 5. 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 problem, the approximate fixpoint computations 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 S4, the widening heuristic was applied as discussed in Section 5.1 which allowed the fixpoint computations to converge and the property was verified in 0.77 sec. using 1.8 MB memory for the infinite case.

Next, we consider the parameterized SIS specification described in Section 5.1 in which the constants `Low` and `Permit` are declared as parameterized constants, where $0 \leq$ `Low` $<$ `Permit`. ALV verified S1 and S3 on the parameterized SIS specification using the same amount of time and memory as reported in Section 5.1 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). With widening, ALV verified S4 in 2.07 sec. using 3.8 MB of memory; the fixpoint computation required 15 iterations. As described in Section 5.1, a shortest counterexample for S2 was generated using the truncated fixpoint computations in 3.08 sec. with 2.7 MB of memory.

### 7.6 Invariant Generation, Theorem Proving, and Decision Procedures

**Invariant Generation.** The techniques for invariant generation presented in [32, 33] exploit the structure of SCR tables. These techniques construct invariants defined on Boolean and enumerated variables from event and mode transition tables.[¶] 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 [32, 33] 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 overspecified), and thus discovery of more general properties is

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

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, 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.

An advantage of a theorem prover such as TAME is that examining the proof script produced by TAME when it verifies a property indicates the facts from the SCR specification that were needed in proving the property. The only feedback that the user receives from a model checker is that the proof is valid. At times, understanding why a proof succeeded can be of value. Similarly, examining the proof script of a failed TAME proof provides often provides information that the user can used to correct the specification (in the case that the specification is incorrect) or to correct the formulation of the analyzed property (in the case in which the property has been incorrectly formulated. Moreover, like ALV, when a proof fails, TAME provides a characterization of *all* states that violate the property; it does not simply return a single counterexample.

**Decision Procedures.** Salsa implements a decision procedure for linear arithmetic constraints based on an automata representation similar to the one used in ALV. However, unlike ALV, Salsa does not compute fixpoints. It converts verification queries to a single satisfiability query on linear arithmetic formulas. This approach leads to more efficient verification performance for inductive invariants. However, for invariants that cannot be verified inductively, Salsa requires strengthening invariants computed by an invariant generator for verification to succeed.

## 8 Utility of ALV in the SCR Toolset

The experimental results described in Sections 5–7 suggest several ways in which ALV could prove useful in verifying SCR specifications. Below we discuss some of these and also indicate 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 [25] 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 7. 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 expressiveness of the SCR language.

**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 7, the SCR abstraction techniques could be applied first and then ALV's approximation techniques could be applied 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 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 7), it is unlikely that users will understand such invariants.

**Using ALV for Consistency Checking.** While Section 6 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 (4) and (5) 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.

## 9 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. Two candidates which are safety-critical are the weapons control system in [27] and the NASA systems in [26].

**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 (4) and (6). The second author is especially grateful to Carolyn Gasarch for extending the SCR toolset to provide timing information about the various analyses performed by the toolset and to Myla Archer for producing the TAME statistics.

## References

[1] R. Alur and D. Dill. Automata for modeling real-time systems. *Theoretical Computer Science (TCS)*, 126:183–236, 1994.

[2] Action Language Verifier (ALV). Available at `http://www.cs.ucsb.edu/~bultan/composite/`.

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

[4] C. Bartzis and T. Bultan. Automata-based representations for arithmetic constraints in automated verification. In J. Champarnaud and D. Maurel, editors, *Proceedings of the Seventh International Conference on Implementation and Application of Automata (CIAA 2002)*, volume 2608 of *Lecture Notes in Computer Science*, pages 282–288. Springer-Verlag, July 2002.

[5] C. Bartzis and T. Bultan. Construction of efficient BDDs for bounded arithmetic constraints. In H. Garavel and J. Hatcliff, editors, *Proceedings of the Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003)*, volume 2619 of *Lecture Notes in Computer Science*, pages 394–408. Springer-Verlag, April 2003.

[6] C. Bartzis and T. Bultan. Efficient image computation in infinite state model checking. In J. Warren A. Hunt and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV 2003)*, volume 2725 of *Lecture Notes in Computer Science*, pages 249–261. Springer-Verlag, July 2003.

[7] 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.

[8] C. Bartzis and T. Bultan. Widening arithmetic automata. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–333. Springer-Verlag, July 2004.

[9] C. Bartzis and T. Bultan. Efficient bdds for bounded arithmetic constraints. *International Journal on Software Tools for Technology Transfer (STTT)*, to appear.

[10] 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.

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

[12] 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.

[13] A. Boudet and H. Comon. Diophantine equations, presburger arithmetic and finite automata. In H. Kirchner, editor, *Proceedings of the 21st International Colloquium on Trees in Algebra and Programming - CAAP'96*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, April 1996.

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

[15] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 113–123, March 1998.

[16] T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(1):3–50, January 2000.

[17] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[18] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):747–789, July 1999.

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

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

[21] 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.

[22] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Journal of Software Tools and Technology Transfer*, 3(3):250–270, 2001.

[23] Fast Acceleration of Symbolic Transition systems (FAST). Available at `http://www.lsv.ens-cachan.fr/fast/`.

[24] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[25] 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.

[26] C. Heitmeyer and R. Jeffords. Applying a formal requirements method to three NASA systems: Lessons learned. In *Proc. 2007 IEEE Aerospace Conf.*, 2007.

[27] 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.

[28] 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.

[29] J. G. Henriksen, J. Jensen, M. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Proc. TACAS 1995*, 1995.

[30] T. A. Henzinger, P. Ho, and H. Wong-Toi. Hytech: a model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

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

[32] 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.

[33] 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.

[34] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal of Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997.

[35] The Liège Automata-based Symbolic Handler (LASH). Available at `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[36] LEarning to VERify properties (LEVER). Available at `http://www.cs.uiuc.edu/homes/vardhan/lever.html`.

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

[38] The Omega project. Available at `http://www.cs.umd.edu/projects/omega/`.

[39] 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.

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

[41] P. Wolper and B. Boigelot. An automata-theoretic approach to Presburger arithmetic constraints. In *Proceedings of the Static Analysis Symposium*, September 1995.

[42] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 1–19. Springer, April 2000.

[43] T. Yavuz-Kahveci. *Specification and Automated Verification of Concurrent Software Systems*. PhD thesis, University of California, Santa Barbara, 2004.

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

[45] 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.

[46] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In T. Margaria and W. Yi, editors, *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *Lecture Notes in Computer Science*, pages 335–344. Springer-Verlag, April 2001.

## A Appendix

```
module main()
  enumerated mBlock, mReset, cSafetyInjection { Off, On };
  integer mWaterPres;
  enumerated mcPressure { TooLow, Permitted, High };
  boolean tOverridden;
  restrict: 0<=mWaterPres and mWaterPres<=2000;
  initial: mBlock=Off and mReset=On and mWaterPres=14 and mcPressure=TooLow
           and !tOverridden and cSafetyInjection=On;
 main:
    // C1: One input assumption and changes in the monitored variables
    (
      ((mBlock=Off and mBlock'=On or mBlock=On and mBlock'=Off)
        and mReset'=mReset and mWaterPres'=mWaterPres)
      or
      ((mReset=Off and mReset'=On or mReset=On and mReset'=Off)
        and mBlock'=mBlock and mWaterPres'=mWaterPres)
      or
      ((mWaterPres<2000 and mWaterPres'=mWaterPres+1
         or mWaterPres>0 and mWaterPres'=mWaterPres-1)
        and mReset'=mReset and mBlock'=mBlock)
    )
    and
    // C2: Transition relation for mcPressure
    (
      (mcPressure=TooLow and !(mWaterPres>=900) and mWaterPres’>=900
        and mcPressure’=Permitted)
      or
      (mcPressure=Permitted and !(mWaterPres>=1000) and mWaterPres’>=1000
        and mcPressure’=High)
      or
      (mcPressure=Permitted and !(mWaterPres<900) and mWaterPres’<900
        and mcPressure’=TooLow)
      or
      (mcPressure=High and !(mWaterPres<1000) and mWaterPres’<1000
        and mcPressure’=Permitted)
      or
      (!((mcPressure=TooLow and !(mWaterPres>=900) and mWaterPres’>=900)
          or
          (mcPressure=Permitted and !(mWaterPres>=1000) and mWaterPres’>=1000)
          or
          (mcPressure=Permitted and !(mWaterPres<900) and mWaterPres’<900)
          or
          (mcPressure=High and !(mWaterPres<1000) and mWaterPres’<1000)
        )
        and mcPressure’=mcPressure)
    )
    and
    // C3: Transition relation for tOverridden
    (
      (!(mcPressure=High) and !(mBlock=On) and mBlock’=On and mReset=Off
        and tOverridden’)
      or
      (((!(mcPressure=High) and mcPressure’=High)
        or (mcPressure=High and !(mcPressure’=High))
        or (!(mReset=On) and mReset’=On and !(mcPressure=High)))
        and !tOverridden’)
      or
      (!((!(mcPressure=High) and !(mBlock=On) and mBlock’=On and mReset=Off)
        or
        ((!(mcPressure=High) and mcPressure’=High)
        or (mcPressure=High and !(mcPressure’=High))
        or (!(mReset=On) and mReset’=On and !(mcPressure=High)))
        )
        and tOverridden’=tOverridden)
    )
    and
    // C4: Transition relation for cSafetyInjection
    (
      (mcPressure’=TooLow and !tOverridden’ and cSafetyInjection’=On)
      or (!(mcPressure’=TooLow and !tOverridden’) and  cSafetyInjection’=Off)
    )
  ;
// *** S1
 spec: AG(tOverridden => (!(mcPressure=High) and mReset=Off))
// *** S2
 spec: AG((mReset=On and mcPressure!=High) =>
                         !tOverridden and cSafetyInjection=On)
// *** S3
 spec: AG((mReset=On and !(mcPressure=High)) => !tOverridden)
```

```
// *** S4
 spec: AG((mReset=On and mcPressure=TooLow) => cSafetyInjection=On)
endmodule
```

## B Appendix

```
module main()
  boolean mBrake, mEngRunning, mIgnOn;
  enumerated mLever { const, release, off, resume };
  integer mSpeed;
  integer time;
  enumerated cThrottle { accel, maintain, decel, off };
  enumerated mcCruise { Off, Inactive, Cruise, Override };
  integer tDesiredSpeed;
  integer tDURLeverEQconst;
  boolean PropertyA5, PropertyA6, PropertyA9, PropertyA11;
  initial: mBrake=false and mEngRunning=false and mIgnOn=false and
           mLever=release and mSpeed=0 and time=0 and cThrottle=off
           and mcCruise=Off and tDesiredSpeed=0 and tDURLeverEQconst=0
           and PropertyA5=true and PropertyA6=true and PropertyA9=true
           and PropertyA11=true;
  restrict: time>=0 and mSpeed>=0 and mSpeed<=180 and
            tDesiredSpeed>=0 and tDesiredSpeed<=180 and tDURLeverEQconst>=0;
  main:
    // One input assumption and changes in the monitored variables
    (
      (mBrake’=!mBrake and
        mEngRunning’=mEngRunning and
        mIgnOn’=mIgnOn and
        mLever’=mLever and
        mSpeed’=mSpeed and
        time’=time)
      or
      (mBrake’=mBrake and
        mEngRunning’=!mEngRunning and
        mIgnOn’=mIgnOn and
        mLever’=mLever and
        mSpeed’=mSpeed and
        time’=time)
      or
      (mBrake’=mBrake and
        mEngRunning’=mEngRunning and
        mIgnOn’=!mIgnOn and
        mLever’=mLever and
        mSpeed’=mSpeed and
        time’=time)
      or
      (mBrake’=mBrake and
        mEngRunning’=mEngRunning and
        mIgnOn’=mIgnOn and
        (mLever’=release and !(mLever=release)
        or !(mLever’=release) and mLever=release ) and
        mSpeed’=mSpeed and
        time’=time)
      or
      (mBrake’=mBrake and
        mEngRunning’=mEngRunning and
        mIgnOn’=mIgnOn and
        mLever’=mLever and
        (-20<=mSpeed’-mSpeed and mSpeed’-mSpeed<=6 and !(mSpeed’-mSpeed=0)) and
        time’=time)
      or
      (mBrake’=mBrake and
        mEngRunning’=mEngRunning and
        mIgnOn’=mIgnOn and
        mLever’=mLever and
        mSpeed’=mSpeed and
        time’=time+1)
      or
      (mBrake’=mBrake and
        mEngRunning’=mEngRunning and
        mIgnOn’=mIgnOn and
        mLever’=mLever and
        mSpeed’=mSpeed and
        time’=time)
    )
    and
    // transition relation for cThrottle
    (
      mcCruise’=Cruise and
      (
        ((tDesiredSpeed’-2>mSpeed’ or
          tDURLeverEQconst’>500) and cThrottle’=accel)
        or
        (tDesiredSpeed’-2<=mSpeed’ and
          tDesiredSpeed’+2>=mSpeed’ and
          tDURLeverEQconst’<=500 and cThrottle’=maintain)
        or
        (tDesiredSpeed’+2<mSpeed’ and
          tDURLeverEQconst’<=500 and cThrottle’=decel)
        or
        (!((tDesiredSpeed’-2>mSpeed’ or
              tDURLeverEQconst’>500)
            or
            (tDesiredSpeed’-2<=mSpeed’ and
              tDesiredSpeed’+2>=mSpeed’ and
              tDURLeverEQconst’<=500)
            or
            (tDesiredSpeed’+2<mSpeed’ and
              tDURLeverEQconst’<=500)
          )
          and cThrottle’=off
        )
      )
      or
      !(mcCruise’=Cruise) and cThrottle’=off
    )
    and
    // transition relation for mcCruise
    (
      mcCruise=Off and
      (
        (!mIgnOn and mIgnOn’ and mcCruise’=Inactive)
        or
        (!(!mIgnOn and mIgnOn’) and mcCruise’=mcCruise)
      )
      or mcCruise=Inactive and
```

```
(
  (mIgnOn and !mIgnOn’ and mcCruise’=Off)
  or
  (!(mLever=const) and mLever’=const and
    mIgnOn and mEngRunning and !mBrake and mcCruise’=Cruise)
  or
  (!(
      (mIgnOn and !mIgnOn’)
      or
      (!(mLever=const) and mLever’=const and
        mIgnOn and mEngRunning and !mBrake)
    )
    and mcCruise’=mcCruise
  )
)
or mcCruise=Cruise and
(
  (mIgnOn and !mIgnOn’ and mcCruise’=Off)
  or
  (mEngRunning and !mEngRunning’ and mcCruise’=Inactive)
  or
  (!mBrake and mBrake’ and mcCruise’=Override)
  or
  (!(mLever=off) and mLever’=off and mcCruise’=Override)
  or
  (!(
      (mIgnOn and !mIgnOn’)
      or
      (mEngRunning and !mEngRunning’)
      or
      (!mBrake and mBrake’)
      or
      (!(mLever=off) and mLever’=off)
    )
    and mcCruise’=mcCruise
  )
)
or mcCruise=Override and
(
  (mIgnOn and !mIgnOn’ and mcCruise’=Off)
  or
  (mEngRunning and !mEngRunning’ and mcCruise’=Inactive)
  or
  (!(mLever=resume) and mLever’=resume and
      mIgnOn and mEngRunning and !mBrake
      and mcCruise’=Cruise)
  or
  (!(mLever=const) and mLever’=const and
      mIgnOn and mEngRunning and !mBrake
      and mcCruise’=Cruise)
  or
  (!(
      (mIgnOn and !mIgnOn’)
      or
      (mEngRunning and !mEngRunning’)
      or
      (!(mLever=resume) and mLever’=resume and
        mIgnOn and mEngRunning and !mBrake)
      or
```

```
          (!(mLever=const) and mLever’=const and
            mIgnOn and mEngRunning and !mBrake)
        )
        and mcCruise’=mcCruise
      )
    )
  )
  and
  // transition relation for tDesiredSpeed
  (
    mcCruise=Cruise and
    (
      (tDURLeverEQconst>500
        and !(tDURLeverEQconst’>500) and tDesiredSpeed’=mSpeed’)
      or
      (!(tDURLeverEQconst>500
         and !(tDURLeverEQconst’>500)) and tDesiredSpeed’=tDesiredSpeed)
    )
    or mcCruise=Inactive and
    (
      (!(mLever=const) and mLever’=const and
         mIgnOn and mEngRunning and !mBrake and tDesiredSpeed’=mSpeed’)
      or
      (!(!(mLever=const) and mLever’=const and
         mIgnOn and mEngRunning and !mBrake) and tDesiredSpeed’=tDesiredSpeed)
    )
    or !(mcCruise=Cruise or mcCruise=Inactive) and tDesiredSpeed’=tDesiredSpeed
  )
  and
  // transition relation for tDURLeverEQconst
  (
    (mLever=const and !(mLever’=const) and tDURLeverEQconst’=0)
    or
    (!(time’=time) and mLever=const and mLever’=const and
      tDURLeverEQconst’=tDURLeverEQconst+(time’-time))
    or
    (!((mLever=const and !(mLever’=const))
        or (!(time’=time) and mLever=const and mLever’=const))
      and tDURLeverEQconst’=tDURLeverEQconst)
  )
  and
  // transition relation for PropertyA5
  (PropertyA5’<=>(mSpeed’=mSpeed => !(cThrottle’=accel)))
  and
  // transition relation for PropertyA6
  (PropertyA6’<=>(!mEngRunning and mEngRunning’ => cThrottle’=off))
  and
  // transition relation for PropertyA9
  (PropertyA9’<=>
        ((!(tDesiredSpeed’=tDesiredSpeed)) =>
          (mcCruise’=Cruise and (mcCruise=Cruise or mcCruise=Inactive))
        )
  )
  and
  // transition relation for PropertyA11
  (PropertyA11’<=>
        ((mSpeed’=mSpeed and tDesiredSpeed=mSpeed) => tDesiredSpeed’=tDesiredSpeed)
  )
;
```

```
//*** A1
  spec: invariant(mBrake => cThrottle=off)
//*** A2
 spec: invariant(cThrottle=accel => tDesiredSpeed>mSpeed)
//*** A3
  spec: invariant(!(mcCruise=Off) and mEngRunning => mIgnOn)
//*** A4
  spec: invariant(!(cThrottle=off) and mEngRunning => mIgnOn)
//*** A5
  spec: invariant(PropertyA5)
//*** A6
  spec: invariant(PropertyA6)
//*** A7
  spec: invariant(mcCruise=Off <=> !mIgnOn)
//*** A8
  spec: invariant(
          cThrottle=accel =>
          (tDesiredSpeed>mSpeed or tDURLeverEQconst>500)
        )
//*** A9
  spec: invariant(PropertyA9)
//*** A10
  spec: invariant(tDesiredSpeed=mSpeed =>
           (!(mcCruise=Override) or !(mLever=release) or cThrottle=off))
//*** A11
 spec: invariant(PropertyA11)
endmodule
```