# Salsa: Combining Constraint Solvers with BDDs for Automatic Invariant Checking

Ramesh Bharadwaj[1] and Steve Sims[2]

[1] Center for High Assurance Computer Systems, Naval Research Laboratory
Washington, DC 20375-5320
ramesh@itd.nrl.navy.mil
[2] Reactive Systems, Inc.
sims@reactive-systems.com
www.reactive-systems.com

**Abstract.** *Salsa* is an invariant checker for specifications in SAL (the **S**CR **A**bstract **L**anguage). To establish a formula as an invariant without any user guidance, Salsa carries out an induction proof that utilizes tightly integrated decision procedures, currently a combination of BDD algorithms and a constraint solver for integer linear arithmetic, for discharging the verification conditions. The user interface of Salsa is designed to mimic the interfaces of model checkers; i.e., given a formula and a system description, Salsa either establishes the formula as an invariant of the system (but returns no proof) or provides a *counterexample*. In either case, the algorithm will terminate. Unlike model checkers, Salsa returns a state pair as a counterexample and not an execution sequence. Also, due to the incompleteness of induction, users must *validate* the counterexamples. The use of induction enables Salsa to combat the state explosion problem that plagues model checkers – it can handle specifications whose state spaces are too large for model checkers to analyze. Also, unlike general purpose theorem provers, Salsa concentrates on a single task and gains efficiency by employing a set of optimized heuristics.

## 1 Introduction

Model checking[17] has emerged as an effective technique for the *automated* analysis of descriptions of hardware and protocols. To analyze software system descriptions, however, a direct application of model checking to a problem (i.e., without a prior reduction of its state space size by the application of abstraction) rarely succeeds [9]. For such systems, theorem proving affords an interesting alternative. Conventional theorem proving systems, however, are often too general or too expensive to use in a practical setting because they require considerable user sophistication, human effort, and system resources. Additionally, the counterexample provided by a model checker when a check fails serves practitioners as a valuable debugging aid. However, in contrast, conventional theorem provers provide little or no diagnostic information (or worse, may not terminate) when a theorem is *not* true.

S. Graf and M. Schwartzbach (Eds.): TACAS/ETAPS 2000, LNCS 1785, pp. 378–395, 2000.

*Salsa* is an invariant checker for system descriptions written in a language based on the tabular notation of SCR [24] called SAL (the **S**CR **A**bstract **L**anguage). Given a logical formula and a system description in SAL, Salsa uses induction to determine whether the formula is true in all states (or transitions) the system may reach. Unlike concurrent algorithms or protocol descriptions, on which model checkers are very effective, practical SAL models usually do not contain interleaving concurrency and are more easily amenable to induction proofs. If a proof fails, Salsa provides a *counterexample*. Unlike model checkers, however, the returned counterexample is a state or a state pair and not an execution sequence. Also, due to the incompleteness of induction, users must *validate* a returned counterexample. Salsa has the attributes of both a model checker and a theorem prover: It is automatic and provides counterexamples just like a model checker. Like a theorem prover, it uses decision procedures, can handle infinite state systems, and can use auxiliary lemmas to complete an analysis.

The design of Salsa was motivated by the need within the SCR Toolset [23] for more automation during consistency checking [24] and invariant checking [9,22]. Salsa achieves complete automation of proofs by its reliance on *decision procedures*, i.e., algorithms that establish the logical truth or falsity of formulae of *decidable* sub-theories, such as the fragment of arithmetic involving only integer linear constraints called Presburger arithmetic. Salsa's invariant checker consists of a tightly integrated set of decision procedures, each optimized to work within a particular domain. Currently, Salsa implements decision procedures for propositional logic, the theory of unordered enumerations, and integer linear arithmetic.

Although they are capable of checking more general properties (such as liveness), in practice model checkers are most often used to check invariant properties. The advantage of using Salsa over a standard model checker for this task is that Salsa can handle large (even infinite state) specifications that current day model checkers cannot analyze. This is due to the use of induction and the symbolic encoding of expressions involving integers as linear constraints. The primary disadvantage of Salsa (and proof by induction in general) is its incompleteness – a failed check does not necessarily imply that a formula is not an invariant because the returned state pair may not be reachable.

After some experimentation, we arrived at the following practical method for checking state and transition invariants using Salsa (see Figure 1): Initially apply Salsa. If Salsa returns *yes* then the property is an invariant of the system, and we are done. If Salsa returns *no*, then we examine the counterexample to determine whether the states corresponding to the counterexample are reachable in the system. If so, the property is false and we are done. However, if one concludes after this analysis that the counterexample states are unreachable, then one looks for *stronger invariants* to prove the property. Salsa currently includes a facility that allows users to include such auxiliary lemmas during invariant checking. There are promising algorithms for automatically deducing such invariants [5,6,11,26], although Salsa currently does not implement them.

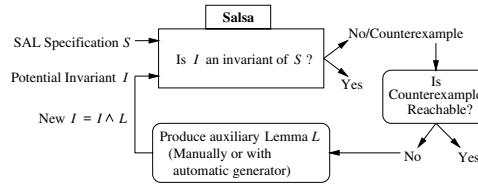



**Fig. 1.** Process for applying Salsa

*Related Work.* The use of SMV [28] and SPIN [25] on software specifications for consistency and invariant checking has been well documented [2,9,16,22]. SCR* [23] is a toolset that includes a *consistency checker* which uses a method based on semantic tableaux extended to handle simple constraints over the integers and reals. This tool has proved very useful in a number of practical case studies; however, the tool is unable to complete the checks on certain examples involving numbers. Systems that largely automate induction proofs by employing decision procedures include the Stanford Temporal Prover (STeP) [11]. Other tools that are built upon the interactive theorem prover PVS [30] include TAME (Timed Automata Modeling Environment) [3] and the tools of Graf et al. [21,32]. These tools are implemented as a set of special-purpose PVS strategies. The tool InVeSt includes sophisticated algorithms for invariant generation and heuristics for invariant strengthening [5,6]. Also, if invariance cannot be established on a finite abstraction, an execution sequence is provided as a diagnostic. Validity checkers such as Mona [18], Mosel [31], and the Stanford Validity Checker (SVC) [4] are another class of systems that employ decision procedures for proving logical formulae. Although these tools do not directly check invariants, they may be used to discharge the verification conditions generated during an induction proof in a tool such as Salsa.

The idea of combining decision procedures for program verification dates back to the work of Shostak [33] and Nelson and Oppen [29]. The decision procedures of Salsa for both propositional logic and enumerated types are based on standard BDD algorithms. The integer constraint solver employs an automata-theoretic algorithm presented in [12], with extensions to handle negative numbers using ideas from [34]. Salsa's technique of combining BDD algorithms with constraint solvers was largely inspired by the approaches of [14] and [15] where, by incorporating constraint solvers into BDD-based fixpoint computation algorithms, verification of infinite state systems becomes a possibility. However, since the underlying algorithms of these systems are variants of the model checking algorithm for computing a fixpoint, we speculate that Salsa, due to its use of induction, can handle larger specifications than these systems. Also, the constraint solver of [14] is incomplete for integer linear arithmetic, whereas the one used by Salsa is complete. The system of [15], which uses an off-the-shelf backtracking solver that can be very inefficient in practice, can handle a class of non-linear constraints in addition to linear constraints.

The rest of this paper is organized as follows. In the following section we introduce the state machines that serve as the underlying models for SAL specifications and define the invariant checking problem. Section 3 describes the core algorithms of Salsa, and Section 4 presents the algorithms and heuristics of the unsatisfiability checker which is used by Salsa to discharge the verification conditions. Section 5 provides some preliminary experimental results of applying Salsa to several practical specifications of moderate size. Finally, Section 6 discusses ongoing work and future research.

## 2 Background

### 2.1 Model for System Behavior

The SCR Abstract Language (SAL), a specification language based on the SCR Formal Model [24], was designed to serve as an abstract interface to analysis tools such as theorem provers, model checkers, and consistency checkers. An example SCR specification in SAL is presented in Appendix A. Unlike concurrent algorithms or protocol descriptions, practical SAL specifications usually do not involve interleaving concurrency and are therefore more easily amenable to induction proofs.

A SAL specification may be translated into a state machine that models a system's behavior. We now introduce the state machine model for systems and the supporting machinery used in the paper. We define formulae in a simple constraint logic *(SCL)* by the following grammar:

$$
\begin{array}{rll}
\Phi := C \mid X_b \mid \neg X_b \mid \Phi \vee \Phi \mid \Phi \wedge \Phi & & \text{(simple formulae)} \\
C := C_i \mid C_e & & \text{(constraints)} \\
C_e := X_e = Val_e \mid X_e \neq Val_e \mid X_e = Y_e \mid X_e \neq Y_e & & \text{(enum. constraints)} \\
C_i := SUM \leq Val_i \mid SUM = Val_i \mid SUM \neq Val_i & & \text{(integer constraints)} \\
SUM := Val_i \times X_i \mid SUM + SUM & &
\end{array}
$$

where $X_b$, $X_e/Y_e$, and $X_i$ range over boolean, enumerated, and integer variables respectively. Similarly $Val_b, Val_e$, and $Val_i$ respectively range over constants of the three types. We let $Vars(\Phi)$ denote the free variables in $\Phi$. Set $Vars(\Phi)$ is partitioned by the three variable types: $Vars(\Phi) = Vars_b(\Phi) \cup Vars_e(\Phi) \cup Vars_i(\Phi)$. Note that SCL formulae will be interpreted in the context of either 1) a single state $s$ that maps variable names to values or 2) a pair of states $(s, s')$, where $s'$ is a successor of $s$. We adopt the convention that primed formulae and variable names (those ending in $'$) are evaluated in the "new state" whereas unprimed names are evaluated in the "old state." Formulae containing primed variables are called *two-state predicates* and those without primed variables are called *one-state predicates*.

**Definition 1.** A state machine $\Sigma$ is a quadruple $\langle \mathcal{V}, \mathcal{S}, \theta, \rho \rangle$ where

- $\mathcal{V}$ is a set of variable names. This set is partitioned into *monitored variables* which denote environmental quantities the system observes; *controlled variables* which denote quantities in the environment that the system controls;

and *internal variables* which are updated by the system but not visible to the environment.
- $\mathcal{S}$ is the set of system states such that each state $s \in \mathcal{S}$ maps each $x \in \mathcal{V}$ to a value in its set of legal values. We write $x(s)$ to mean the value of variable $x$ in state $s$, $\Phi_1(s)$ to mean the value of one-state predicate $\Phi_1$ evaluated in $s$, and $\Phi_2(s, s')$ to mean the value of two-state predicate $\Phi_2$ evaluated with values from $s$ replacing unprimed variables and values from $s'$ replacing primed variables.
- $\theta$ is a one-state SCL predicate defining the set of initial states.
- $\rho$ is a two-state SCL predicate defining the transitions (execution steps) of $\Sigma$. A state $s$ may evolve to a state $s'$ if $\rho(s, s')$ is true.
  The transition relation $\rho$ additionally includes environmental assumptions as well as assumptions introduced by users. For details, see [10].

### 2.2 The Invariant Checking Problem

**Definition 2.** Given a state machine $\Sigma = \langle \mathcal{V}, \mathcal{S}, \theta, \rho \rangle$, a state $s \in \mathcal{S}$ is *reachable* (denoted $Reachable_\Sigma(s)$) if and only if $\theta(s)$ or $\exists s_2 \in \mathcal{S} : Reachable_\Sigma(s_2) \wedge \rho(s_2, s)$

**Definition 3.** Given a state machine $\Sigma = \langle \mathcal{V}, \mathcal{S}, \theta, \rho \rangle$, a one-state SCL predicate $\Phi_1$ is a *state invariant* of $\Sigma$ if and only if

$$\forall s \in \mathcal{S} : Reachable_\Sigma(s) \Rightarrow \Phi_1(s)$$

A two-state SCL predicate $\Phi_2$ is a *transition invariant* of $\Sigma$ if and only if

$$\forall s, s' \in \mathcal{S} : (Reachable_\Sigma(s) \wedge \rho(s, s')) \Rightarrow \Phi_2(s, s')$$

*The invariant checking problem* : Given a state machine $\Sigma$ and a one(two)-state predicate $\Phi$, determine whether $\Phi$ is a state(transition) invariant.

## 3 The Invariant Checker

**Theorem 1.** Let $\Sigma = \langle \mathcal{V}, \mathcal{S}, \theta, \rho \rangle$, then $\Phi_1$ is a state invariant of $\Sigma$ if the following hold: 1) $\forall s \in \mathcal{S} : \theta(s) \Rightarrow \Phi_1(s)$ and 2) $\forall s, s' \in \mathcal{S} : \Phi_1(s) \wedge \rho(s, s') \Rightarrow \Phi_1(s')$

**Proof:** By induction on the number of steps of $\Sigma$ to reach a state.

**Theorem 2.** Let $\Sigma = \langle \mathcal{V}, \mathcal{S}, \theta, \rho \rangle$, then $\Phi_2$ is a transition invariant of $\Sigma$ if the following holds:

$\forall s, s' \in \mathcal{S} : \rho(s, s') \Rightarrow \Phi_2(s, s')$

**Proof:** Follows directly from Definition 3.

### 3.1 The Invariant Checking Algorithms

Using Theorems 1 and 2 we check invariants of $\Sigma = \langle \mathcal{V}, \mathcal{S}, \theta, \rho \rangle$ as follows:

**State Invariants.** To determine if $\Phi_1$ is a state invariant of $\Sigma$:

0. if $\neg\Phi_1$ is unsatisfiable then return *yes*.
1. if $\theta \wedge \neg\Phi_1$ is *not* unsatisfiable then return *no* and the satisfying state as counterexample.
2. if $\Phi_1 \wedge \rho \wedge \neg\Phi'_1$ is unsatisfiable then return *yes*.
   else return *no* and the satisfying state pair as counterexample.

**Transition Invariants.** To determine if $\Phi_2$ is a transition invariant of $\Sigma$:

0. if $\neg\Phi_2$ is unsatisfiable then return *yes*.
1. if $\rho \wedge \neg\Phi_2$ is unsatisfiable then return *yes*.
   else return *no* and the satisfying state pair as counterexample.

These algorithms are sound but not complete – whenever Salsa returns *yes* the given formula is an invariant; however, a *no* answer with a counterexample (a state or a state pair) does not necessarily mean that the formula is not an invariant. Consequently, the user must validate that the counterexample is reachable[1]. Either there is a problem or additional theorems are used to "push through" the invariant. Of course, all added theorems should be proved as invariants by the user (either with Salsa or by some other means).

### 3.2 Optimizations

A naive application of the above algorithms to invariant checking will always fail, even for specifications of a moderate size. We perform several optimizations in Salsa to make invariant checking feasible. One important technique used extensively is to *cache* results as they are computed. In addition to the caching provided by BDD algorithms, we cache the results of calls to the integer constraint solver, the BDD encodings of components of the transition relation, etc.

To partition an unsatisfiability check into simpler sub-problems, we use a technique called *disjunctive partitioning* which corresponds to a case split in a standard proof. This approach takes advantage of the fact that a disjunction is unsatisfiable only if each of its disjuncts is unsatisfiable. The disjunctive form of the transition relation in SAL specifications has proven to be an effective basis for disjunctive partitioning.

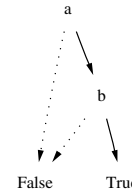
The application of *abstraction* [9,22] is also very beneficial. We restrict ourselves to applying abstractions that are both sound and complete, by which we mean the following. Given a property $\Phi$ and a state machine $\Sigma$, an abstraction $\Sigma_A$ is a sound and complete abstraction of $\Sigma$ relative to $\Phi$ when $\Phi$ is an invariant of $\Sigma_A$ if and only if $\Phi$ is an invariant of $\Sigma$. Currently, we apply what is termed "*Abstraction Method 1*" [8,9] that uses the set of variable names occurring in the predicate $\Phi$ and dataflow analysis to eliminate unneeded variables.

[1] The single state counterexample returned by step 1 of the algorithm for State Invariants (for a failed check of unsatisfiability of $\theta \wedge \neg\Phi_1$) is always a true counterexample.

# 4 The Unsatisfiability Checker

## 4.1 Overview

To discharge the verification conditions that arise during invariant checking, Salsa uses a routine that decides the unsatisfiability of SCL formulae. Both the problem of propositional unsatisfiability and the decision problem for integer linear arithmetic are NP-complete [20], and known algorithms for the latter problem have super-exponential worst case behavior [19]. The unsatisfiability checker uses a combination of binary decision diagrams and an integer constraint solver as a decision procedure for SCL formulae. Using the formula $x \leq 4 \wedge x = 7$ as an example we outline the algorithm (for specifics, see [10]). The initial step transforms a formula into one containing only logical connectives and boolean variables. This is done by assigning a fresh boolean variable to each integer constraint in the original formula. Fresh boolean variables are also introduced to encode expressions involving variables of enumerated type in the obvious way [10,28]. For the example, substituting $a$ for $x \leq 4$ and $b$ for $x = 7$ yields the formula $a \wedge b$. Next, a BDD for this formula (which encodes the propositional structure of the original formula) is constructed:




The next step brings in the information contained in the integer constraints. This is done by searching for paths from the root to "True", each path yielding a set of integer constraints. For the example, the only path from the root to "True" sets both $a$ and $b$ to true, which yields the set $\{x \leq 4,\ x = 7\}$. The final step is to determine whether each such set is infeasible (i.e., has no solution) using an integer constraint solver. If a set is feasible, this information is returned to the user as a counterexample. For the example, the (single) set of constraints is infeasible and the formula is unsatisfiable. We now describe the integer constraint solver in detail.

## 4.2 The Integer Constraint Solver

As an initial step, a set of integer constraints is partitioned into independent subsets. For example, the set of constraints $\{x < 4, x > 7, y < 10\}$ may be partitioned into $\{x < 4, x > 7\}$ and $\{y < 10\}$.

**Definition 4.** Constraints $c_1$ and $c_2$ are *independent* if $Vars(c_1) \cap Vars(c_2) = \emptyset$. The partition of a set of constraints $CS = \{c_1, ..., c_n\}$ into independent subsets (denoted $\Pi(CS)$) is defined as $\Pi(CS) = \{CS_1, ..., CS_m\}$ such that:

1. $\Pi(CS)$ partitions $CS$.
2. Constraints in different partitions are independent.
3. For each partition containing more than one constraint, every constraint in the partition depends on some other constraint in the partition.

To compute $\Pi(CS)$ Salsa uses a union-find algorithm that starts with each constraint in its own partition and iteratively merges partitions when they contain dependent constraints.

After partitioning a set of constraints into independent subsets, an integer constraint solver determines the feasibility of each independent subset. For a set of constraints, we may conclude that the whole set is infeasible if any independent subset is infeasible.

Salsa's constraint solver is a decision procedure that determines whether a set of integer constraints is infeasible, i.e., given $\{c_1, c_2, ..., c_n\}$ the solver checks whether $c_1 \wedge c_2 \wedge ... \wedge c_n$ is unsatisfiable. Note that the $c_i$ are terms from the integer constraint fragment of SCL (defined in Section 2.1). Among several methods available for solving linear integer constraints, one possible approach is the use of automata theoretic methods. The idea, which dates back to Büchi in the early sixties [13], is to associate with each constraint an automaton accepting the solutions of the constraint. The feasibility of a set of constraints may then be computed by constructing a composite automaton (from the constraint automata for each $c_i$, $1 \leq i \leq n$) using the standard construction for automata intersection. Salsa's solver employs the algorithm of Boudet and Comon [12], extended to handle negative number based on ideas of Wolper [34]. We give an overview of the algorithm, for details see the above references.

Let us first examine how a constraint automaton may encode constraints over the natural numbers, and then extend this idea to automata for integer constraints. Let $c$ be a constraint, let $Vars(c) = \{x_1, x_2, ..., x_n\}$, and let $c[y_1/x_1, y_2/x_2, ..., y_n/x_n]$ denote the result of substituting $y_i$ for each $x_i$ in $c$. We then define the *constraint automaton* for $c$, denoted CAut($c$), such that the language of CAut($c$) is $\{(y_1, ..., y_n) \in Int^n \mid c[y_1/x_1, ..., y_n/x_n] \text{ is true}\}$. Each number $y_i$ is encoded in base two, so each $y_i$ is a string in $\{0,1\}^*$. The constraint automaton will recognize solutions to a constraint by simultaneously reading one bit for each of its free variables, i.e., the edges of the automaton will be labeled by elements of $\{0,1\}^n$. For example, the satisfying assignments of "$x_1 + x_2 = 4$" are $\{(0,4), (1,3), (2,2), (3,1), (4,0)\}$, so CAut($x_1 + x_2 = 4$) encodes this as shown in Figure 2.

We now explain how to construct a constraint automaton for a constraint $c$ of the form $a_1x_1 + a_2x_2 + \ldots + a_nx_n = b$, where $a_1, a_2, \ldots\ a_n$, $b$ are integer constants and $x_1, x_2, \ldots\ x_n$ are variables over the natural numbers. The resulting automaton will be of the form CAut($c$) $= \langle S, E, St, Acc \rangle$ where $S \subseteq Integers$ is the set of states and $E \subseteq S \times \{0,1\}^n \times S$ is the set of edges, $St \subseteq S$ is the set of start states, and $Acc \subseteq S$ is the set of accepting states. During construction we let $S_{new}$ represent the set of states still to be processed. The construction proceeds backwards from the accepting state as follows.

| $(x_1, x_2)$ | Accepted string |
|---|---|
| (0,4) | 0 0 0<br>1 0 0 |
| (1,3) | 0 1<br>1 1 |
| (2,2) | 1 0<br>1 0 |
| (3,1) | 1 1<br>0 1 |
| (4,0) | 1 0 0<br>0 0 0 |

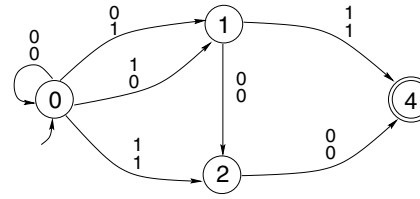



**Fig. 2.** The constraint automaton encoding $x_1 + x_2 = 4$

1. Initialize both $S_{new}$ and $Acc$ to contain only $b$ (the right hand side of the constraint) and initialize $S = E = St = \emptyset$.
2. Remove a state $s$ from $S_{new}$ for processing.
   (a) Add $s$ to $S$. If $s = 0$ then also add $s$ to $St$.
   (b) For each $\beta \in \{0,1\}^n$ where $\beta = \langle b_1, b_2, ..., b_n \rangle$
   let $\delta = s - (a_1 b_1 + a_2 b_2 + \ldots + a_n b_n)$ in
   if $\delta$ is even then
   - add edge $(\delta \text{ div } 2) \xrightarrow{\beta} s$ to $E$
   - if $(\delta \text{ div } 2) \notin (S \cup S_{new})$ then add $(\delta \text{ div } 2)$ to $S_{new}$
3. if $S_{new} = \emptyset$ then return $\langle S, E, St, Acc \rangle$ else goto 2.

Some simple modifications to the above algorithm extend it to handle negative numbers. For integer constraint $c$, the states of CAut($c$) range over integers and we add a special state $\mathcal{I}$ that will encode the start state, thus $S \subseteq Int \cup \mathcal{I}$. Instead of the standard binary encoding employed for natural numbers the two's complement representation is used for integers. The above algorithm must also be modified to handle the sign bit of the two's complement notation via a special encoding for the start state ($\mathcal{I}$) and extra edges from $\mathcal{I}$. We do this by removing "if $s = 0$ then also add $s$ to $St$" from 2(a) and adding the following to 2(b) above.

if $s = (-a_1 b_1 - a_2 b_2 - \ldots - a_n b_n)$
then add $\mathcal{I}$ to $S$ and $St$ and add edge $\mathcal{I} \xrightarrow{\beta} s$ to $E$.

The basic algorithm may also be changed to build constraint automata for constraints involving "$\neq$" and "$\leq$". For "$\neq$" the construction is exactly the same except that $Acc = S - b$, i.e., the accepting state becomes non-accepting and all others become accepting. For details of the slightly more complicated modifications for "$\leq$" see [10].

The constraint automaton for a set of constraints $CS = \{c_1, c_2, ..., c_n\}$, denoted CAut($CS$), is defined as CAut($CS$) $= \bigcap_{i=1}^{n}$ CAut($c_i$). The automaton CAut($CS$) is constructed *on the fly*, thereby avoiding the need to build each CAut($c_i$). Let $S_i$ denote the states of CAut($c_i$), then the states of CAut($CS$) are $S_{CS} \subseteq S_1 \times S_2 \times \ldots \times S_n$. An unsatisfiability check of $CS$ then proceeds backwards from the accepting state and terminates with false when the initial state is reached or terminates with true if the automaton construction completes without reaching the start state.

## 5 Empirical Results

### 5.1 Motivation

Salsa was designed expressly for the problems of consistency checking and invariant checking SCR requirements specifications. More specifically, the consistency checker of the SCR Toolset [23] was unable to carry out certain checks, such as checks for unwanted nondeterminism called disjointness checks, especially on specifications containing expressions with numbers. We have also been using SPIN and SMV, and more recently TAME [3], to verify user formulated properties of SCR specifications. We compare Salsa with TAME/PVS to gain an insight into how well the Salsa approach performs in relation to that of a state-of-the-art theorem prover.

We compare Salsa with model checkers for the following reason. During the course of our experiments with SCR specifications we have discovered that for model checking to succeed on these specifications requires the application of abstraction, which currently requires user-direction but is automatable [9,22]. Further, SPIN and SMV are unable to provide a definitive answer for invariant checks on a number of examples, especially when they contain a large number of expressions with numbers [27]. Also, since several researchers are currently investigating the use of SPIN and SMV for invariant checking software specifications, it is our intention to demonstrate that Salsa affords a viable, perhaps more automated and cheaper, alternative to model checking. Whereas mechanical theorem provers are regarded as being difficult to use and therefore restricted to sophisticated users, model checking too is often misrepresented as fully automatic or "push button". Our intention is to demonstrate an approach to invariant checking that avoids both the ad hoc abstraction used in model checking and the sophistication required to apply mechanical theorem proving.

The specifications we use in our experiments were developed using the SCR Toolset. Since Salsa seems to work well on *all* of this limited set of examples, readers may express skepticism about the generality of our results – they may feel that there *must* be benchmarks for which the roles would be reversed. By using induction, abstract encodings for linear constraints, and application-specific heuristics, our experience is that the Salsa approach can in general be more efficient than fixpoint computation over a finite domain, i.e., model checking. However, Salsa has the disadvantage of not working in *all* cases, due to the associated problem of incompleteness.

*Test Cases.* These include a simplified specification of the control software for a nuclear power plant [24] (**safety-injection**), versions of the bomb-release component of the flight-control software of an attack aircraft [1] (**bomb-release-1** and **bomb-release-2**), a simplified mode control panel for the Boeing 737 autopilot [7] (**autopilot**), a control system for home heating (**home-heating**), an automobile cruise control system (**cruise-control**), a navy application [27] (**navy**), the mode logic for the Operational Flight Program of an attack aircraft [1] (**a7-modes**), and a weapons control panel [22] (**wcp**).

### 5.2 Disjointness Checking

To evaluate the performance of Salsa, we checked the above specifications for disjointness errors (unwanted nondeterminism) and compared the results with the consistency checker of the SCR Toolset. The results of our experiments are shown in the table of Figure 3. **No auxiliary lemmas were used for any of the checks.** The column labeled "number of verification conditions" indicates how many invariant checks are required to establish disjointness for the corresponding entire specification. The number of BDD variables is an indicator of a specification's size, and the number of integer constraints correlates loosely with the degree to which integers are used in the specification. In these tables, symbol "$\infty_t$" means that the corresponding system either ran out of memory or failed to terminate (over a weekend). The column labeled "number of failed VCs" shows the number of verification conditions that were not provable. Note: for the specification **a7-modes** Salsa reports more failed VCs than the SCR toolset because certain cases of overlap in table entries are misdiagnosed as disjointness errors when they should probably be warnings. For specification **cruise-control** Salsa establishes disjointness in three cases for which the SCR Toolset cannot. The tests were conducted on a PC running Linux with a 450 MHz Pentium II processor and 256 MBytes RAM.

| | *Number of* | | | *Time (in seconds) to Check Disjointness* | | *Number of Failed VCs* | |
|---|---|---|---|---|---|---|---|
| *Specification* | *Verification Conditions* | *BDD Variables* | *Constraints* | *SCR Toolset* | *Salsa* | *SCR Toolset* | *Salsa* |
| *Specifications containing mostly booleans and enumerated types* | | | | | | | |
| safety-injection | 13 | 16 | 3 | 0.5 | 0.2 | 0 | 0 |
| bomb-release-1 | 12 | 34 | 9 | 0.4 | 0.2 | 0 | 0 |
| a7-modes | 6171 | 158 | 3 | 145.9 | 68.9 | 110 | 152 |
| *Specifications containing mostly numerical variables* | | | | | | | |
| home-heating | 98 | 112 | 55 | $\infty_t$ | 4.8 | n.a. | 0 |
| cruise-control | 123 | 114 | 75 | 21.0 | 3.6 | 6 | 3 |
| navy | 397 | 147 | 102 | 390.1 | 198.2 | 0 | 0 |
| bomb-release-2 | 339 | 319 | 230 | $\infty_t$ | 246.0 | n.a. | 11 |

**Fig. 3.** Results of Disjointness Checks

Figure 3 shows that for specifications containing mostly variables of boolean and enumerated type, both the SCR Toolset and Salsa can complete the analysis but Salsa is somewhat faster. For specifications containing mostly numerical variables, there were two specifications in which Salsa could perform the analysis but the SCR Toolset could not.

### 5.3 Checking Application Properties

To evaluate Salsa's performance on properties formulated by users, we compared the run times with the theorem prover TAME/PVS and the two popular model checkers SPIN [25] and SMV [28]. (We used SPIN Version 2.9.7 of April 18, 1997, SMV r2.4 of December 16, 1994, and PVS version 2.1 for our experiments.) The results are shown in Figure 4. Note that the PVS proof times do not include time for type checking, which can be substantial. We ran the experiments on a SPARC Ultra-2 running Solaris with a 296 MHz UltraSparc II processor and 262 MBytes RAM. All Salsa proofs were completely automatic, but for `property 304` of **wcp**, which had to be split into two verification conditions for Salsa to complete; the time indicated with an asterisk is the sum of the running times of the two sub-proofs. **All auxiliary lemmas were automatically generated** by the algorithm of [26] and proved as invariants by Salsa. Both SPIN and SMV ran out of memory (or ran indefinitely) when run on all examples other than **safety-injection**. This is probably because they contain a large number of numerical variables. Dashes ("-") in the SMV column indicate that we did not run SMV on these examples.

| *Specification* | *Number of Properties* | *Time (in seconds)* | | | | *Properties Proved?* | *Auxiliary Lemmas Used?* |
|---|---|---|---|---|---|---|---|
| | | *Salsa* | *SPIN* | *SMV* | *TAME/PVS* | | |
| safety-injection | 4 | 0.8 | 36.0 | 155.0 | 68 | Yes | Yes |
| bomb-release-1 | 2 | 1.3 | $\infty_t$ | $\infty_t$ | 30 | Yes | No |
| autopilot | 2 | 1.5 | $\infty_t$ | $\infty_t$ | 82 | Yes | No |
| navy | 7 | 396.0 | $\infty_t$ | - | 874 | Yes | Yes |
| wcp | property 303 | 295.4 | $\infty_t$ | - | $\infty_t$ | No | No |
| | property 304 | 923.3* | $\infty_t$ | - | 19 | No | No |
| | property 305 | 2.4 | $\infty_t$ | - | 8 | No | No |

**Fig. 4.** Results of Invariant Checks

## 6 Conclusions

In this paper, we show that the Salsa approach affords a useful alternative to model checking, especially for the analysis of descriptions of software. Mechanical theorem provers such as PVS are regarded as being too general and too expensive to use, requiring sophistication on the part of their users. Salsa provides the advantages of both mechanical theorem proving and model checking – it is automatic, easy to use, and provides counterexamples along the lines of model checkers. The counterexamples, however, are over two adjacent states and not entire execution sequences. The main advantage of our approach is that we are able to handle much larger specifications, even infinite state specifications, that current day model checkers cannot handle (without a prior application of abstraction).

The major disadvantage of the Salsa approach over conventional model checking is its incompleteness – a proof failure does not imply that the theorem does not hold. However, this is generally true of model checking too, because an initial application of model checking to a practical problem rarely succeeds – users of model checkers routinely apply abstractions (mostly manually and sometimes in ad-hoc ways) for model checking to proceed [9]. These abstractions are usually sound, but are often incomplete – consequently, if one model checks an incomplete abstraction of a problem, the entire process is incomplete. Model checking, however, remains very useful for *refuting* properties, i.e., as a debugging aid. As with Salsa, the resulting counterexample must be validated against the full specification.

We plan to extend Salsa to include decision procedures for the rationals, the congruence closure algorithm to reason about uninterpreted function symbols, and special-purpose theories such as for arrays and lists. We would also like to reason about quantifiers. We have designed Salsa to be general, i.e., to check a variety of state machine models for invariant properties. We plan on trying out the tool on state machine models other than SCR.

## Acknowledgements

This project is funded by the Office of Naval Research. The work of Steve Sims was carried out at NRL under a contract from ONR. We thank Susanne Graf, Connie Heitmeyer, Ralph Jeffords, and the anonymous referees for their comments on previous drafts of this paper. Connie's very useful comments, her constructive criticism, and numerous suggestions for improvement greatly helped the presentation. Ralph was the first user of Salsa! We thank Myla Archer for data on the PVS proofs and Ralph for the mode invariants.

## References

1. T. A. Alspaugh et al. Software requirements for the A-7E aircraft. Technical Report NRL-9194, Naval Research Laboratory, Wash., DC, 1992. 387
2. R. J. Anderson, P. Beame, et al. Model checking large software specifications. In *Proc. Fourth ACM FSE*, October 1996. 380
3. M. Archer, C. Heitmeyer, and S. Sims. TAME: A PVS interface to simplify proofs for automata models. In *Proc. User Interfaces for Theorem Provers*, Eindhoven, Netherlands, July 1998. Eindhoven University CS Technical Report. 380, 387
4. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In *Formal Methods In Computer-Aided Design*, volume 1166 of *LNCS*, pages 187–201, November 1996. 380
5. S. Bensalem and Y. Lakhnech. Automatic Generation of Invariants. *Formal Methods in Systems Design*, July 1998. 379, 380
6. S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In *Conference on Computer Aided Verification CAV'96*, LNCS 1102, July 1996. 379, 380

7. R. Bharadwaj and C. Heitmeyer. Applying the SCR requirements method to a simple autopilot. In *Proc. Fourth NASA Langley Formal Methods Workshop (LFM97)*, NASA Langley Research Center, September 1997. 387
8. R. Bharadwaj and C. Heitmeyer. Verifying SCR requirements specifications using state exploration. In *1st ACM Workshop on Autom. Analysis of Software*, 1997. 383
9. R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Journal of Automated Software Eng.*, January 1999. 378, 379, 380, 383, 387, 390, 393
10. R. Bharadwaj and S. Sims. Salsa: Combining decision procedures for fully-automatic verification. Technical report, Naval Research Laboratory, To appear. 382, 384, 386
11. N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 1999. 379, 380
12. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Trees and Algebra in Programming – CAAP*, LNCS 1059, 1996. 380, 385
13. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congress Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1960. 385
14. T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. Technical Report UMIACS-TR-97-62, University of Maryland, College Park, MD, August 1997. 380
15. W. Chan, R. Anderson, P. Beame, and D. Notkin. Combining constraint solving with symbolic model checking for a class of systems with non-linear constraints. In *Computer Aided Verification*, LNCS, pages 316–327, 1997. 380
16. William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model checking large software specifications. *IEEE Trans. on Softw. Eng.*, 24(7), July 1998. 380
17. E. M. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Trans. on Prog. Lang. and Systems*, 8(2):244–263, April 1986. 378
18. J. Elgaard, N. Klarlund, and A. Moller. Mona 1.x: new techniques for ws1s and ws2s. In *Computer Aided Verification, CAV '98*, LNCS 1427, 1998. 380
19. Fischer and Rabin. Super-exponential complexity of Presburger arithmetic. In *Complexity of Computation: Proceedings of a Symposium in Applied Mathematics of the American Mathematical Society and the Society for Industrial and Applied Mathematics*, 1974. 384
20. M.R. Garey and D.S. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W.H. Freeman and Company, 1979. 384
21. S. Graf and H. Saidi. Verifying invariants using theorem proving. In *Conference on Computer Aided Verification CAV'96*, LNCS 1102, Springer Verlag, 1996. 380
22. 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*, 24:927–947, November 1998. 379, 380, 383, 387
23. C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, June 1998. 379, 380, 387

24. 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. 379, 381, 387
25. G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Softw. Eng.*, 23(5):279–295, May 1997. 380, 389
26. R. Jeffords and C. Heitmeyer. Automatic generation of state invariants from requirements specifications. In *Proc. Sixth ACM SIGSOFT Symp. on Foundations of Software Engineering*, November 1998. 379, 389
27. J. Kirby, Jr., M. Archer, and C. Heitmeyer. SCR: A practical approach to building a high assurance COMSEC system. In *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC '99)*, December 1999. 387
28. K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993. 380, 384, 389
29. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979. 380
30. 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, February 1995. 380
31. P.Kelb, T.Margaria, M.Mendler, and C.Gsottberger. Mosel: A flexible toolset for monadic second–order logic. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *(LNCS)*, pages 183–202, Heidelberg, Germany, March 1997. Springer–Verlag. 380
32. Y. Lakhnech S. Bensalem and S. Owre. InVeSt: A tool for the verification of invariants. In *Proc. Computer-Aided Verification, 10th Annual Conf. (CAV'98)*, Vancouver, Canada, June 1998. 380
33. Robert E. Shostak. Deciding combinations of theories. *JACM*, 31(1):1–12, January 1984. 380
34. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Computer Aided Verification – $10^{th}$ Int'l. Conference*, LNCS 1427, 1998. 380, 385

## A SAL Specification of Safety Injection System

A module is the unit of specification in SAL and comprises variable declarations, assumptions and guarantees, and definitions. The `assumptions` section typically includes assumptions about the environment and previously proved invariants (lemmas). The required invariants of a module are specified in the `guarantees` section. The `definitions` section specifies updates to internal and controlled variables. A *one-state* definition, of the form `var` $x = rhs1$ (where $rhs1$ is a one-state SAL expression), defines the value of variable `x` in terms of the values of other variables *in the same state*. A two-state variable definition, of the form `var` $x$ `initially` $init := rhs2$ (where $rhs2$ is a two-state SAL expression), requires the initial value of `x` to equal expression $init$; the value of `x` in the "new" state of each state transition is defined in terms of the values of variables in the "new" state *as well as* the "old" state. Expression `@T(x) WHEN y` is syntactic sugar for $\neg x \wedge x' \wedge y$ and `@F(x)` denotes `@T(NOT x)`. A *conditional expression* consists of a sequence of branches "`[]` guard $\rightarrow$ expression", where the guards

are boolean expressions, bracketed by the keywords "`if`" and "`fi`". In a given state, the value of a guarded expression is equivalent to the expression on the right hand side of the arrow whose associated guard is true. If more than one guard is true, the expression is nondeterministic. A conditional event expression (which is bracketed by the keywords "`ev`" and "`ve`") requires each guard to denote an *event*, where an event is a two-state expression that is true in a pair of states only if they differ in the value of at least one state variable.

We specify in SAL a simplified version of a control system for safety injection [9]. The system monitors water pressure and injects coolant into the reactor core when the pressure falls below a threshold. The system operator may override safety injection by pressing a "Block" button and may reset the system by pressing a "Reset" button. To specify the requirements of the control system, we use variables `WaterPres`, `Block`, and `Reset` to denote the monitored quantities and variable `SafetyInjection` to denote the controlled quantity. The specification includes a mode class `Pressure`, an abstract model of `WaterPres`, which has three modes: `TooLow`, `Permitted`, and `High`. It also includes a term `Overridden` and several conditions and events.

```
module sis
functions
  Low = 900; Permit = 1000;
monitored variables
  Block, Reset : {On, Off};
  WaterPres : int in [0,2000];
controlled variables
  SafetyInjection : {On, Off};
internal variables
  Overridden : bool;
  Pressure   : {TooLow, Permitted, High};

assumptions
  /* Mode invariant generated by the algorithm of Jeffords [26] */
  LemmaZ = (Overridden => Reset = Off and not (Pressure = High));
guarantees
  /* The following properties are true */
  Property1 = (Reset = On and  Pressure != High) =>  not Overridden;
  Property2 = (Reset = On and Pressure = TooLow) => SafetyInjection = On;
  /* The following properties are false */
  Property3 =(Block = Off and Pressure = TooLow) => SafetyInjection = On;
  Property4 =(@T(Pressure=TooLow) when Block=Off) => SafetyInjection'=On;

definitions
  var Overridden initially false :=
    ev
      [] @T(Pressure = High) -> false
      [] @T(Block = On) when (Reset = Off and Pressure != High) -> true
      [] @T(Pressure != High) or
         @T(Reset = On) when (Pressure != High) -> false
    ve
```

```
  var Pressure initially TooLow :=
    case Pressure
      [] TooLow ->    ev [] @T(WaterPres >= Low) -> Permitted   ve
      [] Permitted -> ev [] @T(WaterPres < Low) -> TooLow
                         [] @T(WaterPres >= Permit) -> High
                      ve
      [] High ->      ev [] @T(WaterPres < Permit) -> Permitted ve
    esac

  var SafetyInjection =
    case Pressure
      [] High, Permitted -> if [] true  -> Off  [] false -> On fi
      [] TooLow -> if [] Overridden -> Off [] not Overridden -> On fi
    esac
end module
```

**Fig. 5.** SAL specification of Safety Injection System