A Strategy for Efficiently Verifying Requirements Specifications Using Composition and Invariants[*]

Ralph D. Jeffords
Naval Research Laboratory (Code 5546)
Washington DC 20375 USA
jeffords@itd.nrl.navy.mil

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

ABSTRACT

This paper describes a compositional proof strategy for verifying properties of requirements specifications. The proof strategy, which may be applied using either a model checker or a theorem prover, uses known state invariants to prove state and transition invariants. Two proof rules are presented: a standard incremental proof rule analogous to Manna and Pnueli's incremental proof rule and a compositional proof rule. The advantage of applying the compositional rule is that it decomposes a large verification problem into smaller problems which often can be solved more efficiently than the larger problem. The steps needed to implement the compositional rule are described, and the results of applying the proof strategy to two examples, a simple cruise control system and a real-world Navy system, are presented. In the Navy example, compositional verification using either theorem proving or model checking was three times faster than verification based on the standard incremental (noncompositional) rule. In addition to the two above rules for proving invariants, a new compositional proof rule is presented for circular assume-guarantee proofs of invariants. While in principle the strategy and rules described for proving invariants may be applied to any state-based specification with parallel composition of components, the specifications in the paper are expressed in the SCR (Software Cost Reduction) tabular notation, the auxiliary invariants used in the proofs are automatically generated invariants, and the verification is supported by the SCR tools.

Categories and Subject Descriptors

D.2.1 [**Requirements/Specifications**]: languages, methodologies, tools; D.2.4 [**Software/Program Verification**]: formal methods, model checking

General Terms

Documentation, Security, Verification

Keywords

requirements specification, formal methods, compositional verification, invariants, software tools, model checking, theorem proving

1. INTRODUCTION

A challenging problem in applying formal techniques in software development is how to demonstrate that the large requirements specifications associated with most practical systems satisfy critical properties. On the one hand, the users of model checking technology [12] must overcome the "state explosion problem," i.e., how to exhaustively search, either explicitly or implicitly, the large state spaces of practical system specifications. On the other hand, while theorem provers, such as PVS [36], can handle large, even infinite state spaces, they lack automation for formulating and proving the inductive invariants often necessary to complete a proof. Typically, applying a theorem prover requires user ingenuity to formulate the needed invariants, theorem proving skills, and detailed knowledge of a particular prover.

This paper makes two contributions to verifying system and software requirements specifications. The first contribution is a compositional proof method for verifying invariant properties of requirements specifications. This method decomposes a large verification problem into smaller problems which often can be solved more efficiently than the larger problem. Based on a mathematically sound foundation, the method may be applied to state machine models with parallel composition, e.g., LUSTRE [14], $RSML^{-e}$ [10], Reactive Modules [3], and SCR (Software Cost Reduction) [15]. In addition, the method may in large part be automated. Inspired by a general compositional proof rule of McMillan [30], the second contribution is a compositional proof rule for circular assume-guarantee proofs of two special classes of invariants. These two classes of invariants—state and transition invariants—are defined in Section 2.2.

This paper shows how our compositional proof strategy may be applied using the SCR method and tools [19, 18, 17] in conjunction with either a model checker or a theorem prover. In the examples in the paper, the required system behavior is specified in the SCR tabular notation and the system properties in a restricted form of first-order logic. The examples use state invariants automatically generated from an SCR specification. The invariants were constructed by applying the algorithms described in [21, 22]. The verification tools used in our experiments include a theorem prover called Salsa [8], which applies a decision procedure combining a BDD algorithm and a linear integer constraint solver, the symbolic model checker SMV [28], and the explicit state model checker SPIN [20]. Salsa has been customized to prove properties of SCR requirements specifications. To use SMV (and SPIN), SCR specifications were translated into the SMV language (and Promela), using the methods described in [7].

To provide a basis for illustrating our proof strategy, Section 2 introduces a simple cruise control system, an SCR specification of the system, a set of state invariants automatically generated from the SCR specification, and a set of properties we wish to prove about the specification. Section 3 describes two classes of proof rules that use invariants in verification, two versions of the standard incremental proof rule introduced by Manna and Pnueli in 1995 [27]

---

[*]This work was supported by the Office of Naval Research.

This paper is authored by employees of the United States Government and is in the public domain.
ESEC/FSE'03, September 1–5, 2003, Helsinki, Finland.
ACM 1-58113-743-5/03/0009 .

and two compositional proof rules whose goal is to improve the efficiency of verification. It also illustrates the application of these proof rules to the cruise control example. Section 4 presents a more general circular assume-guarantee rule and shows that the compositional rule is a special case, while Section 5 describes the results of using Salsa, SMV, and SPIN to apply the proof rules to the verificiation of a practical Navy system. Finally, Section 6 discusses related work, and Section 7 presents some conclusions and future plans.

2. BACKGROUND: A SIMPLE EXAMPLE

2.1 Specifying Cruise Control in SCR

To illustrate our proof strategy, we consider a simplified automobile cruise control system derived from [23]. This Cruise Control System (CCS) monitors several quantities in its environment, e.g., the position of the cruise control lever and the automobile's speed, and uses this information to control a throttle. If the ignition is on, the engine running, and the brake off, the driver enters cruise control mode by moving the lever to the `const` position. In cruise control mode, the automobile's speed determines whether the throttle accelerates or decelerates the automobile or maintains the current speed. The driver overrides cruise control by engaging the brake, resumes cruise control by moving the lever to `resume`, and exits cruise control by moving the lever to `off`.

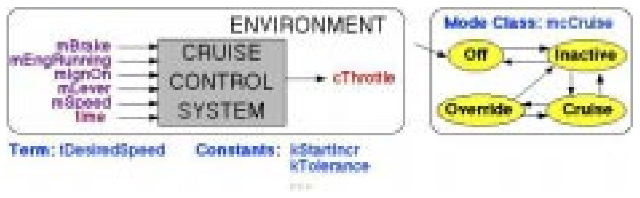


Figure 1: Specifying Cruise Control in SCR

Figure 1 shows how SCR state variables can be used to specify the CCS requirements. The monitored (or input) variables, `mIgnOn`, `mEngRunning`, `mSpeed`, `mBrake`, and `mLever`, represent the state of the automobile's ignition and engine, the automobile's speed, and the positions of the cruise control lever and the brake. The distinguished monitored variable `time` indicates time passage. The controlled (or output) variable `cThrottle` represents the state of the throttle. The CCS specification contains two auxiliary variables, a mode class `mcCruise` and a term `tDesiredSpeed`, which capture state history and make the specification more concise. We refer to the auxiliary and controlled variables in an SCR specification as the *dependent variables*. In SCR, the value of each dependent variable is defined by a function in a tabular format. See Appendix A for an overview of the SCR requirements model, a review of the SCR tabular notation and table types, and three examples of SCR tables. Each of the three tables describes a function. These functions define the values of the three dependent variables in the SCR specification of CCS, namely, `tDesiredSpeed`, `mcCruise`, and `cThrottle`.

For technical reasons, DUR, the time duration operator of SCR, was abstracted from the CCS specification before verification was applied. (See Appendix A for an example of the DUR operator.) An important point is that our compositional strategy applies even when earlier abstractions have already been performed on the specification. For example, *slicing* of the SCR specification [17, 7] (which removes variables that do not affect the validity of the candidate invariant) is automatically performed by Salsa before verification is applied.

2.2 Invariants of the CCS Specification

We define a system Σ as a state machine $\Sigma = (S, \Theta, \rho)$, where S is the set of states, $\Theta : S \rightarrow boolean$ is the initial state predicate, and $\rho : S \times S \rightarrow boolean$ is the next-state predicate. Associated with Σ is a set $RF = \{r_1, r_2, \ldots, r_n\}$ of state variables (i.e., monitored and dependent variables) and a function TY which maps each state variable to its set of legal values. A *state* s in S is a function that maps each variable in RF to its value; i.e., for all $r \in RF, s(r) \in TY(r)$. In SCR specifications, two classes of properties are of interest, one-state properties, predicates defined on a single state, and two-state properties, predicates defined on state pairs. Given a system Σ, we define a *state invariant* as a one-state property that holds in every reachable state of Σ and a *transition invariant* as a two-state property that holds in adjacent pairs of reachable states in Σ.

We have designed two algorithms [21, 22] for constructing state invariants from the tables defining the dependent variables in an SCR specification. Suppose dependent variable r has values in a finite set $\{v_1, v_2, \ldots, v_n\}$. If r's value is defined by a mode transition table or an event table, two types of tables in SCR specifications, then for each v_i the algorithms generate invariants of the form

$$(r = v_i) \;\Rightarrow\; C_i, \tag{1}$$

where C_i is a predicate over the variables in Σ on which r depends. Invariant generation from SCR tables is based on the following intuitive result: In an SCR specification, $(r = v_i) \Rightarrow C_i$ is an invariant if 1) C_i is always true when r's value changes to v_i, and 2) an event falsifying C_i unconditionally causes r to have a value other than v_i. Since stronger invariants may be computed with knowledge of previously computed invariants, the full algorithms repeat the computations of the invariants until a fixpoint is reached.

The current implementation of the SCR invariant generator applies our algorithms to both mode transition tables and event tables. Figure 2 lists four state invariants, I1-I4, of the SCR specification of CCS that our invariant generator constructed from the mode transition table for the mode class `mcCruise` (see Table 2 in Appendix A). State invariants constructed from a mode transition table are called *mode invariants*.

2.3 Properties of the CCS

The Assertion Dictionary in Figure 3 lists eight properties we want to prove about the CCS specification: A6, a two-state property, and seven one-state properties. The assertions are expressed in the same notation as that used in the SCR tables with the addition of "=>" for *implies* and "<=>" for *iff*. In Assertion A6, the expression "@T(`mEngRunning`)" denotes an event indicating a change in the value of the boolean `mEngRunning` from false in the old state to true in the new state. Also in A6, the expression `cThrottle`$'$ represents the value of `cThrottle` in the new state.

3. PROVING INVARIANTS: TWO RULES

This section describes four proof rules: two standard incremental proof rules and two compositional proof rules. These rules use known state invariants to prove new state and transition invariants. The section also enumerates the steps of our compositional proof method and illustrates the application of the proof rules by using them to verify properties of the CCS.

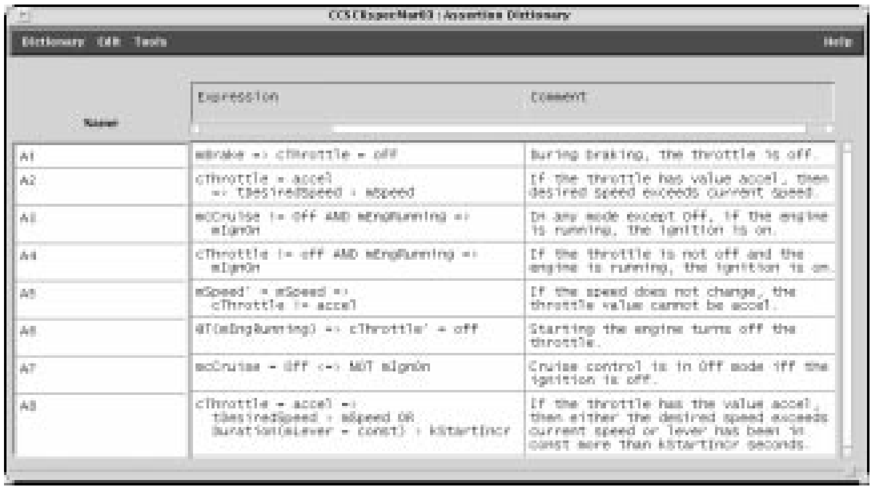
3.1 A Standard Proof Strategy

In many cases, properties such as those listed in Figure 3 cannot be proven directly from the specification. To complete the proof of such properties, one or more auxiliary invariants are usually

Name	Generated Invariant	Description
I1	`mcCruise = Off` $\Rightarrow$ `NOT mIgnOn`	In Off mode, the ignition is off.
I2	`mcCruise = Inactive` $\Rightarrow$ `mIgnOn`	In Inactive mode, the ignition is on.
I3	`mcCruise = Cruise` $\Rightarrow$ `mIgnOn AND mEngRunning AND NOT mBrake AND mLever` $\neq$ `off`	In Cruise mode, the ignition is on, the engine is running, the brake is off, and the lever is not off.
I4	`mcCruise = Override` $\Rightarrow$ `mIgnOn AND mEngRunning`	In Override mode, the ignition is on and the engine is running

Figure 2: Automatically generated invariants for CCS

.



Name	Expression	Comment
A1	mBrake => cThrottle = off	During braking, the throttle is off.
A2	cThrottle = accel => tDesiredSpeed > mSpeed	If the throttle has value accel, then desired speed exceeds current speed.
A3	mcCruise != off AND mEngRunning => mIgnOn	In any mode except Off, if the engine is running, the ignition is on.
A4	cThrottle != off AND mEngRunning => mIgnOn	If the throttle is not off and the engine is running, the ignition is on.
A5	mSpeed' = mSpeed => cThrottle != accel	If the speed does not change, the throttle value cannot be accel.
A6	@T(mEngRunning) => cThrottle' = off	Starting the engine turns off the throttle.
A7	mcCruise = Off <=> NOT mIgnOn	Cruise control is in Off mode iff the ignition is off.
A8	cThrottle = accel => tDesiredSpeed > mSpeed OR Duration(mLever = const) > kStartIncr	If the throttle has the value accel, then either the desired speed exceeds current speed or lever has been in const more than kStartIncr seconds.

Figure 3: CCS assertions.

needed. A variant of a well-known strategy for using an invariant p to prove that property q is invariant is a proof rule, which we call the Basic Incremental Proof (BIP) Rule. Given a system $\Sigma = (S, \Theta, \rho)$, the set $\mathrm{Inv}(\Sigma)$ of state and transition invariants of Σ, and two one-state properties, p and q, defined over the variables of Σ, the Basic Incremental Proof Rule is defined by:

$$\frac{p \in \mathrm{Inv}(\Sigma),\ (p \wedge \Theta) \Rightarrow q,\ (q \wedge p \wedge p' \wedge \rho) \Rightarrow q'}{q \in \mathrm{Inv}(\Sigma)}$$

In the right-most premise of the BIP rule, an unprimed property, e.g., q, denotes the property in the old state, and a primed property, e.g., q', denotes the property in the new state. The BIP proof rule above is slightly better than the incremental rule that Manna and Pnueli derive from their proof rule SV-PSV [27] because one can prove more with this rule than with their rule. A similar rule for proving that a two-state property $\hat{q}$ is invariant, the Modified Incremental Proof (MIP) Rule, is defined by:

$$\frac{p \in \mathrm{Inv}(\Sigma),\ (p \wedge p' \wedge \rho) \Rightarrow \hat{q}}{\hat{q} \in \mathrm{Inv}(\Sigma)}$$

Suppose p is a state invariant of Σ. To prove q is invariant, we can use a verifier to determine whether the two right-most premises of the BIP Rule hold. To prove two-state properties are invariant, we proceed in a similar manner but determine if the right-most premise of the MIP Rule holds.

If we define p as the conjunction of the mode invariants I1–I4 and q as any one of the five one-state properties in Figure 3, namely A1, A3, A4, A7, and A8, application of BIP is sufficient to prove that each of these five properties is a state invariant. (Properties A2 and A5 have been excluded from the one-state properties in Figure 3 to be proven, since we have shown elsewhere using model checking that these properties are false [16].) Similarly, if $\hat{q}$ is defined as the two-state property A6, then MIP proves that A6 is a transition invariant. In general, the automatically generated invariants may not be sufficient to complete the verification. However, the use of automatically generated invariants means that the user is normally required to supply only some, rather than all, auxiliary invariants needed to complete a proof. The STeP prover [26] supports a similar proof strategy, using automatically generated invariants to supplement those that the user must develop by hand.

3.2 A More Efficient Compositional Strategy

The major limitation of the above proof strategy is that the proof relies on the entire system specification. Another strategy which may be more efficient decomposes the system Σ into two smaller systems Σ_1 and Σ_2 and performs verification on the smaller systems rather than the larger one. By smaller, we mean that the number of initial state definitions and function definitions is smaller in each of Σ_1 and Σ_2 than in Σ. (Recall that the function definitions describe the values of the dependent variables.) Like the standard strategy, this strategy depends on auxiliary invariants to complete the proof. Given a system $\Sigma = (S, \Theta, \rho)$, suppose p is a state invariant of Σ, q is a one-state property we wish to prove invariant, $\Sigma_1 = (S, \Theta_1, \rho_1)$ and $\Sigma_2 = (S, \Theta_2, \rho_2)$ are two "compatible" systems derived from Σ, and $\Sigma_1 || \Sigma_2$ is the parallel composition of Σ_1 and Σ_2, where *parallel composition* is defined as conjunction, i.e., $\Sigma_1 || \Sigma_2 = (S, \Theta_1 \wedge \Theta_2, \rho_1 \wedge \rho_2)$. We consider two systems to be *compatible* if they have the same set of variables and the same type set associated with each variable. For compatible systems, the following proof rule, called the Basic Compositional Proof (BCP) Rule, is sound:

$$\frac{p \in \mathrm{Inv}(\Sigma_1), (p \wedge \Theta_2) \Rightarrow q, (q \wedge p \wedge p' \wedge \rho_2) \Rightarrow q'}{q \in \mathrm{Inv}(\Sigma_1 || \Sigma_2)} \quad (2)$$

A similar sound compositional rule for proving a two-state property $\hat{q}$ is invariant is the Modified Compositional Proof (MCP) Rule:

$$\frac{p \in \mathrm{Inv}(\Sigma_1), (p \wedge p' \wedge \rho_2) \Rightarrow \hat{q}}{\hat{q} \in \mathrm{Inv}(\Sigma_1 || \Sigma_2)}$$

3.3 A Compositional Verification Method

Described below is a compositional method for constructing two compatible systems Σ_1 and Σ_2 from Σ and for applying the compositional proof rule. This method consists of six steps:

1. Given the specification of a system Σ, construct a set of state invariants for Σ using algorithms such as [21, 22]. Alternatively, *prove* a set of state invariants of the form shown in (1).
2. Given Σ and state invariant p of Σ derived from dependent variable r, partition the set D of dependent variables in Σ into two sets: $D_1 = \{r\}$ and $D_2 = D \setminus D_1$.
3. Construct Σ_1 from D_1 and Σ as follows:
   (a) Delete from Θ the initial state definitions of variables in D_2, and assign the result to Θ_1.
   (b) Delete from ρ the functions defining the values of variables in D_2, and assign the result to ρ_1.
4. Similarly, construct Σ_2 from D_2 and Σ as follows:
   (a) Delete from Θ the initial state definition of variable r in D_1, and assign the result to Θ_2.
   (b) Delete from ρ the function defining the values of variable r in D_1, and assign the result to ρ_2.
5. Verify that p is an invariant of Σ_1.
6. Use a verifier to check whether the two right-most premises of the BCP rule hold, i.e., that $(p \wedge \Theta_2)$ implies q and that $(q \wedge p \wedge p' \wedge \rho_2)$ implies q'. If so, q is a state invariant of Σ.

Based on the definition of parallel composition and the construction of Σ_1 and Σ_2 described above, it follows that $\Sigma_1 || \Sigma_2 = \Sigma$. Note that although steps 3 and 4 are of the form used in SCR specifications, they could be generalized and then customized for other state machine models. Note also that, in step 5, verifying that p is an invariant of Σ_1 may be problematic in general but will often hold for the automatically generated invariants that our algorithms generate. See Section 4.1 for further details.

An intuitive way to understand how the compositional rules work is that each rule replaces a dependent variable in the specification with an abstraction, the set of state invariants derived from that variable. For example, prior to using the BCP rule to prove properties of the CCS specification, one could replace the definition of the values of the mode class `mcCruise` (a function defined by the mode transition table in Table 2 of Appendix A) with the mode invariants in Figure 2. Analysis of this more abstract system should be more efficient, and is often sufficient, for verifying properties of the original system.

To apply this technique to the CCS specification, recall that invariants I1-I4 were all derived from the mode transition table in Table 2 of Appendix A. This table defines the mode class `mcCruise` as a function of the values of the monitored variables. Based on this information, we form the set D of dependent variables, $D = \{$`mcCruise`, `tDesiredSpeed`, `cThrottle`$\}$, and partition D into two sets, D_1 and D_2, where $D_1 = \{$`mcCruise`$\}$ contains the variable from which the invariants I1–I4 were constructed, and $D_2 = \{$`tDesiredSpeed`, `cThrottle`$\}$ contains the remaining dependent variables. We define system Σ_1 as equal to Σ but with the initial values and function definitions (given by the SCR tables) of `tDesiredSpeed` and `cThrottle` deleted. Similarly, we define Σ_2 as equal to Σ with the initial state definition and the function defining the value of `mcCruise` deleted. Deleting the initial state definition and the function definition of a dependent variable r means that r may nondeterministically take on any type-correct value.

To evaluate the five one-state properties of interest in Figure 3, namely A1, A3, A4, A7, and A8, we define q as one of the five properties and p as the conjunction of the mode invariants I1–I4 in Figure 2. Then, using a verifier, we must prove for each q that the two right-most premises of the BCP proof rule hold. If the proof succeeds, the compositional proof rule tells us that property q is a state invariant of $\Sigma_1 || \Sigma_2 = \Sigma$. Using Salsa, we applied both the standard strategy and the compositional strategy to verify each of the five properties. The standard proof strategy using the BIP Rule required a total of 0.55 seconds to verify all five properties, whereas the compositional strategy using the BCP Rule required a total of 0.47 seconds. Although the compositional proof strategy is more efficient, the speed-up is minor since the CCS example is too small to do justice to this approach.

In the case of the two-state property A6, using Salsa to apply the compositional strategy MCP fails to prove this known transition invariant. This example illustrates an inherent weakness of the compositional proof rules MCP and BCP: in some cases, a property that could be proved incrementally using a noncompositional proof rule, such as the MIP or BIP Rule, with an invariant p cannot be proved using a compositional strategy with the same invariant p. In the case of A6, applying the compositional proof rule MCP replaced a detailed part of the specification (the function defining the values of the mode class `mcCruise`) with an invariant (the conjunction of the invariants in Figure 2), but this produces an "over-approximation" of the original specification—i.e., the more abstract specification allows additional behaviors that invalidate the invariant property to be proved. In such cases, the remedy is often to strengthen the invariant p or to use another dependent variable (in the case of CCS, a dependent variable other than `mcCruise`) as the basis for constructing the subsystems Σ_1 and Σ_2.

4. MORE GENERAL PROOF STRATEGIES

This section describes some basic reachability properties that hold when applying parallel composition to systems and then shows, for the special case of SCR systems, that our automated method for constructing state invariants usually produces invariants satisfying the left-most premise of the BCP and MCP rules. Next, the section introduces a circular assume-guarantee rule which applies in cases where the desired properties p and q require that the proof of p for subsystem Σ_1 depends on q and that the proof of q for subsystem Σ_2 depends on p. A Compositional Model Checking Proof (CMP) Rule is presented as a special case of this assume-guarantee rule. The BCP rule is then easily derived from the CMP Rule.

4.1 Reachability and Invariants

The proof rules in Section 3 and the more general rules introduced in this section hold for any two subsystems (not necessarily SCR systems), $\Sigma_i = (S, \Theta_i, \rho_i)$, $i = 1, 2$, in which composition is defined as conjunction, i.e., $\Sigma_1 || \Sigma_2 = (S, \Theta_1 \wedge \Theta_2, \rho_1 \wedge \rho_2)$. Thus these proof rules apply to systems specified by LUSTRE, Reactive Modules, and other state-based languages similar to SCR.

From the above definition of the composition of two subsystems Σ_1 and Σ_2, it follows that the reachable states of $\Sigma_1||\Sigma_2$ are those that are reachable in both Σ_1 and Σ_2. Consequently, invariants of each of the subsystems are also invariants of the composition, i.e., $\mathrm{Inv}(\Sigma_i) \subseteq \mathrm{Inv}(\Sigma_1||\Sigma_2)$ for $i = 1, 2$.

For the special case of SCR systems, it is straightforward to show that the decomposition of a system as described in Section 3.3 satisfies these properties. Hence, the decomposition satisfies $\Sigma_1||\Sigma_2 = \Sigma$. Moreover, for SCR specifications, it is often easy to show that the left-most premise of the BCP and MCP Rules holds, i.e., that p is an invariant of Σ_1. This is the case, for example, when, given a system Σ, a dependent variable r, p is generated as an invariant from the definition of r, and r only depends on the monitored variables (as for the CCS) or additionally depends on non-monitored variables whose constraining definitions and assumptions are not used during invariant generation (as for the CD system, see Section 5).

4.2 Assume-Guarantee Rules

Our assume-guarantee rule is inspired by McMillan's rule for circular compositional reasoning [30]. McMillan's rule is used for proving the invariance of $\phi_1 \wedge \phi_2$, where ϕ_1 and ϕ_2 are any LTL (Linear Time Temporal Logic) properties. Given a system $\Sigma = (S, \Theta, \rho)$, suppose $\Sigma_1 = (S, \Theta_1, \rho_1)$ and $\Sigma_2 = (S, \Theta_2, \rho_2)$ are two compatible systems derived from Σ, e.g., they are constructed using the method in Section 3.3, and $\Sigma_1||\Sigma_2$ is the parallel composition of Σ_1 and Σ_2. For the special case where ϕ_1 and ϕ_2 are one-state properties, let $\phi_1 = p$ and $\phi_2 = q$. Then, McMillan's rule can be expressed as follows:

$$\frac{\Sigma_1 \models p \triangleright q,\ \Sigma_2 \models q \triangleright p}{p \wedge q \in \mathrm{Inv}(\Sigma_1||\Sigma_2)} \quad (3)$$

In (3), $\triangleright$ is an LTL operator, and the expression $p \triangleright q$ means that, for all steps $n \geq 0$, assuming p holds from 0 up through $n-1$ implies that q holds from 0 up through n, where n represents the number of steps so far in a system history. Intuitively, this means that "p fails before q".

However, our proof strategies only rely on state invariants[1] so "assuming" a formula up through step $n-1$ can soundly be replaced by the conjunction of the assumed one-state property, say x, with both Θ and ρ in $\Sigma = (S, \Theta, \rho)$. This produces a new system $\Sigma' = (S, x \wedge \Theta, x \wedge \rho)$. Applying this replacement to both premises of the McMillan rule (and including the additional premise that $p \wedge q$ hold initially in $\Sigma_1||\Sigma_2$) produces the following preliminary assume-guarantee rule:

$$\frac{\begin{array}{c}\Theta_1 \wedge \Theta_2 \Rightarrow p \wedge q,\\ p \in \mathrm{Inv}(S, q \wedge \Theta_1, q \wedge \rho_1), q \in \mathrm{Inv}(S, p \wedge \Theta_2, p \wedge \rho_2)\end{array}}{p \wedge q \in \mathrm{Inv}(\Sigma_1||\Sigma_2)} \quad (4)$$

The proof rule in (4) can be improved in two ways:

- We include the auxiliary state invariants, $a \in \mathrm{Inv}(\Sigma_1)$ and $b \in \mathrm{Inv}(\Sigma_2)$. By including these auxiliary invariants, we can establish the relative completeness[2] of the proof rule, If the proof rule is relatively complete, then failure of the proof rule means that the property $p \wedge q$ is *not* an invariant of $\Sigma_1||\Sigma_2$. This generalization does not make the verification task easier—it just ensures that as necessary one can theoretically obtain completeness by developing stronger and stronger auxiliary invariants to aid in the proof.
- We assume invariant p holds in *both* the old and new states, whereas q is assumed in the old state only. Thus, the circularity only needs to be broken in one place rather than in two. McMillan's rule for proving general invariant properties can also be improved in this manner.

[1]Transition invariants can be handled similarly, but the rules for transition invariants are not formally developed in this paper.

[2]Relative completeness means completeness under the assumption that all theorems of number theory, etc., hold in the system.

The two above improvements result in a Better Assume-Guarantee (BAG) Rule:

$$\frac{\begin{array}{cr}a \in \mathrm{Inv}(\Sigma_1), b \in \mathrm{Inv}(\Sigma_2), \Theta_1 \wedge \Theta_2 \Rightarrow p \wedge q, & \\ p \in \mathrm{Inv}(S, a \wedge b \wedge q \wedge \Theta_1, a \wedge b \wedge a' \wedge b' \wedge q \wedge \rho_1) & (5)\\ q \in \mathrm{Inv}(S, a \wedge b \wedge p \wedge \Theta_2, a \wedge b \wedge a' \wedge b' \wedge p \wedge p' \wedge \rho_2) & (6)\end{array}}{p \wedge q \in \mathrm{Inv}(\Sigma_1||\Sigma_2)}$$

Note that, in (6), p is assumed in *both* the old and new states, whereas in (5), q is assumed in the old state only.

Reducing the BAG rule to the special case where $b = \mathtt{TRUE}$ and $p = \mathtt{TRUE}$ and then renaming a to p, we obtain the Compositional Model Checking Proof (CMP) Rule:

$$\frac{p \in \mathrm{Inv}(\Sigma_1), q \in \mathrm{Inv}(S, p \wedge \Theta_2, p \wedge p' \wedge \rho_2)}{q \in \mathrm{Inv}(\Sigma_1||\Sigma_2)}$$

Finally, replacing the second premise of the CMP Rule using the Manna/Pnueli Basic Rule [27]

$$\frac{\Theta \Rightarrow x, x \wedge \rho \Rightarrow x'}{x \in \mathrm{Inv}(S, \Theta, \rho)} \quad (7)$$

produces the BCP Rule (see (2) in Section 4):

$$\frac{p \in \mathrm{Inv}(\Sigma_1), (p \wedge \Theta_2) \Rightarrow q, (q \wedge p \wedge p' \wedge \rho_2) \Rightarrow q'}{q \in \mathrm{Inv}(\Sigma_1||\Sigma_2)}$$

The more general form of the second premise of the CMP Rule expressed as an invariant is directly suitable for model checking.[3] However, the CMP Rule is not directly amenable to theorem proving first-order formulae, so we transform CMP using Manna and Pnueli's Basic Rule to obtain first-order formulae in the resulting BCP Rule.

We have also developed a more extensive proof rule from which these and many other rules (including rules for transition invariants) may be derived. See Appendix B for details on the soundness and relative completeness of these proof rules.

5. VERIFYING A PRACTICAL SYSTEM

The CD (Cryptographic Device) is a practical system designed to provide cryptographic processing for a U.S. Navy radio receiver. To evaluate the correctness of the CD prose specification, we developed an SCR specification of CD and formulated several security properties that the specification must satisfy [24]. The SCR specification of CD is moderately complex, consisting of 39 variables (17 monitored variables, one mode class, two terms, and 19 controlled variables). Figure 4 lists three security properties, two one-state properties and one two-state property, that should hold in the SCR specification of CD. Figure 5 lists three mode invariants our invariant generator constructed from the SCR specification of CD.

As in the case of the CCS, the mode class in the CD specification is used to partition the set D of dependent variables into two

[3]Actually any technique that establishes state invariants could be used, but we only consider model checking in this paper.

Name	Property	Description
B1	$\texttt{cKeyBank1Key1} \neq 0 \vee \texttt{cKeyBank1Key2} \neq 0 \Rightarrow \texttt{cAlgStoreSegment1} \neq 0$ AND $\texttt{cKeyBank2Key1} \neq 0 \vee \texttt{cKeyBank2Key2} \neq 0 \Rightarrow \texttt{cAlgStoreSegment2} \neq 0$	For $i = 1, 2,\ j = 1, 2$, no key can be stored in location i of keybank j before an algorithm has been loaded into the first location of alg. storage segment i
B2	$@T(\texttt{mBackupPower} = \texttt{undervoltage})$ WHEN $\texttt{mPrimaryPower} = \texttt{unavailable} \Rightarrow \texttt{mcOperation}' = \texttt{Alarm}$ OR $\texttt{mcOperation}' = \texttt{Off}$	If backup power has an undervoltage when primary power is unavailable CD enters either Alarm or Off mode
B3	$\texttt{mBackupPower} = \texttt{overvoltage} \Rightarrow \texttt{mcOperation} = \texttt{Initialization}$ OR $\texttt{mcOperation} = \texttt{Standby}$ OR $\texttt{mcOperation} = \texttt{Alarm}$ OR $\texttt{mcOperation} = \texttt{Off}$	If backup power is overvoltage then CD is in Initialization, Standby, Alarm, or Off mode.

Figure 4: Required CD security properties

.

Name	Generated Invariant	Description
J1	$\texttt{mcOperation} = \texttt{Off} \Rightarrow \texttt{mBackupPower} \neq \texttt{available}$	In Off mode, backup power is unavailable.
J2	$\texttt{mcOperation} = \texttt{Standby} \Rightarrow \texttt{mBackupPower} \neq \texttt{unavailable}$ AND $\texttt{mBackupPower} \neq \texttt{undervoltage}$ AND (NOT mTamper AND $\texttt{mZeroizeSw} \neq \texttt{on}$ AND mHealthyFull OR $\texttt{mPrimaryPower} \neq \texttt{available}$)	In Standby mode, backup power is available and not under voltage, the device has not been tampered with, the Zeroize switch is off, Healthyfull is true, or primary power is unavailable.
J3	$\texttt{mcOperation} = \texttt{Config}$ OR $\texttt{mcOperation} = \texttt{Idle}$ OR $\texttt{mcOperation} = \texttt{TrafficProc} \Rightarrow$ mHealthyBackground AND $\texttt{mBackupPower} \neq \texttt{overvoltage}$ AND $\texttt{mPrimaryPower} = \texttt{available}$	In Config., Idle or TrafficProc mode, Healthy Background is true, backup power is not over voltage, and primary power is available.

Figure 5: Automatically generated invariants for CD

.

sets, D_1 and D_2, where $D_1 = \{\texttt{mcOperation}\}$ contains the mode class mcOperation and $D_2 = D \setminus D_1$ contains the remaining dependent variables. Then, the systems Σ_1 and Σ_2 are constructed as described in steps 2 and 3 in Section 3.2. We define p as the conjunction of the three state invariants in Figure 5 and q as one of the three properties in Figure 4.

5.1 Theorem Proving: Salsa

Using Salsa, we applied both the standard proof strategy (using the BIP and MIP Rules) and the compositional strategy (using the BCP and MCP Rules) to verify that the CD specification satisfies each of the three properties in Figure 4 [34]. To verify the three properties, the standard strategy using the BIP Rule for B1 and B3 and the MIP Rule for the two-state property B2 required a total of 73.8 seconds, whereas our compositional strategy using BCP for B1 and B3 and MCP for B2 required a total of 24.4 seconds. Thus, the compositional proof strategy was approximately three times faster than the non-compositional strategy.

5.2 Symbolic Model Checking: SMV

We also applied the CMP Rule using SMV [28] and compared the result to SMV model checking without composition. (Unlike theorem proving, model checking without composition does not require the auxiliary invariant p.) Due to the inefficient encoding of integers in SMV, it was necessary to reduce the size of one system parameter from 1000 to 20. (Without this change, SMV failed to finish when left to run overnight.) It is easy to show that this data abstraction is sound with respect to the properties in Figure 4. Moreover, verification using SMV was highly sensitive not only to the variable ordering used in constructing the BDDs but also to the order in which the SCR function definitions appear in the SMV specification. The time required to model check the three CD properties without composition using different BDD orderings and different definition orderings varied from 432.6 seconds to 23.8 seconds. Using these corresponding orderings for model checking with the CMP Rule required 0.38 seconds to 0.36 seconds, so even for the fastest ordering without composition, the compositional approach was significantly faster [34]. We expect similar speed-ups in verifications of other SCR specifications of practical systems using either model checking or theorem proving.

5.3 Explicit State Model Checking: SPIN

For comparison, we also applied compositional verification using SPIN, an explicit state model checker. Unlike SMV, which represents the states of the system symbolically, SPIN explicitly generates the actual states of the system. Intuitively, one can expect compositional verification using BCP to be slower with SPIN because the number of concrete states increases when concrete definitions are replaced by more abstract sets of invariants in the system Σ_2. Indeed, compositional verification of the Cruise Control System using Spin, while giving sound results, generated more reachable states and was thus slower than symbolic model checking. Hence, our compositional proof strategy is not appropriate for explicit state model checking and therefore we did not try to verify the CD properties using a compositional approach with SPIN [34]. On the other hand, efficiency in verification was achieved with both SMV and Salsa. This is because the performance of these two verification tools is affected not by the number of concrete states but rather by the size of the formulas encoding the state transition system. Since the invariants generated for a variable have a simpler logical expression than the corresponding tabular definition, it is to be expected that verification using these invariants will usually be more efficient.

6. RELATED WORK

In model checking LUSTRE programs, whose specifications are similar to specifications of SCR systems, Halbwachs et al. [14] introduced the Compositional Model Checking Proof Rule. Some of the earliest work on the more general case of "circular" assume-guarantee reasoning appeared in [9] in the context of safety properties of networks of processes. There have been essentially two approaches to "breaking the circularity" when a proof of φ for Σ_1 depends on ψ, and the proof of ψ for Σ_2 depends on φ. For safety properties it suffices to show that both subsystems have non-blocking behavior, that is, assuming ψ within Σ_1 allows that subsystem to proceed to some next state, and similarly assuming φ within Σ_2 allows that subsystem to proceed. Abadi and Lamport for the case of interleaved processes [1] and Alur and Henzinger for the case of the synchronous Reactive Modules [3, 2] ensure this behavior by placing restrictions on the types of properties that may be proved. Another approach, which also works for liveness properties, is to define proof rules that explicitly break the circularity. Such approaches include the imposition of a well-founded order on the properties and auxiliaries [33, 30] or rules involving the LTL temporal logic operator $\triangleright$ [30, 31]. The importance of relative completeness, which we have incorporated in our rules with auxiliaries, is described in [31].

In a related project at the Naval Research Laboratory, compositional verification is also being addressed in the Secure Operations Language (SOL) and its toolset. SOL is a prototype language for the specification and analysis of Multi-Agent Systems [6].

7. CONCLUSIONS AND FUTURE WORK

Our preliminary results show that the compositional approach may be easily integrated with the current SCR toolset and method to make verification of state and transition invariants more efficient. The CMU SMV model checker and the Salsa theorem prover may each be applied unchanged to prove properties via both the Basic Compositional Proof Rule in (2) and the more general assume-guarantee rule (i.e., the BAG Rule). Preliminary experiments have shown that this is also true for the TAME interface to the PVS theorem prover [4], provided that TAME performs an extra simplification that has not yet been needed in other applications. The best way to incorporate this extra simplification into the TAME invariant proof strategy remains to be determined but so far significantly better average proof times for state invariants have been achieved using TAME with the BCP Rule. We also plan to apply our compositional strategy using newer symbolic model checkers, such as NuSMV [11] and Cadence SMV [29].

It would be a simple task to implement the decompositions and coordination of subproofs for Salsa as a text processor of Salsa input specifications, although it would be more convenient to incorporate compositional theorem proving within the tool to avoid the recompilation of the textual form multiple times for the various subsystems. In fact, Cadence SMV includes a scripting language for coordinating the various analyses required in performing assume-guarantee proofs [29].

There still remains the problem of how best to decompose the system when doing compositional verification. In the two systems examined in this paper it was natural to do the simple decomposition $D_1 = \{m\}$ and $D_2 = D \setminus D_1$, where m is an SCR dependent variable and p the automatically generated invariants for that variable. In general, when there are multiple sets of invariants describing abstractions of the various variables, how should one decompose the system? We would like to automate this process as much as possible in keeping with our philosophy that verification capabilities of a toolset should be directly accessible to non-experts.

We have developed an assume-guarantee rule that places no direct restrictions on the properties but may require the user to derive necessary auxiliary properties. Although the need for this circular rule has not been encountered so far for SCR specifications (and indeed at the requirements phase, circularity seems to be less of a problem than during other phases of system development), we anticipate the need for circular reasoning in future requirements specifications. It would be interesting to compare our proof strategy to that of Reactive Modules to understand which is more effective in practice.

Our compositional verification strategy applies even after other abstractions have been performed (most notably those abstractions which are both sound and complete). For example, Salsa automatically employs "slicing" before verification. Additional abstractions were also performed in preparing the Cruise Control specification and in making SMV verification of the CD system feasible. However, proofs may fail in the compositional approach due to over-approximation, so the difficulty in strengthening the invariant may outweigh the advantages of a faster verification.

The approach to verification provided by Salsa and SMV (and supplemented by automated generation of invariants as well as our new approach incorporating compositionality) contrasts with the "predicate abstraction with refinement" approach [13, 35]. In predicate abstraction, one constructs abstract versions of subsystems, often with the goal of reducing the model's state space to a tractable size so that model checking is practical. Lakhnech et al. [25] show that, from a theoretical point of view, each approach seems equally difficult, i.e., it is just as difficult to find strong enough auxiliary invariants to prove a property as it is to design an appropriate predicate abstraction. Also, iterations of strengthening invariants after failure of the compositional approach due to "over-approximation" is analogous to failure in the predicate abstraction approach where you must repeat the process using a refinement of the current predicate abstraction (e.g., based upon the counterexample returned from the failed model checking step). We plan to further investigate how these methods could complement each other or be combined in verifying requirements specifications expressed in SCR.

Acknowledgments

We give special thanks to our NRL colleague, Myla Archer, for many useful discussions and for her insightful comments on the paper. We are also especially grateful to John Ramish who conducted numerous experiments to evaluate our compositional proof strategies with Salsa, SMV, SPIN, and TAME. We also thank our NRL colleague Ramesh Bharadwaj, our summer interns, John Ramish, Dejan Desovski, and Muralidhar Talupur, and the anonymous reviewers for constructive comments. Finally, we thank Ramesh Bharadwaj and Steve Sims, who designed and implemented Salsa.

8. REFERENCES

[1] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. and Sys.*, 17(3):507–535, May 1995.

[2] R. Alur and T. A. Henzinger. Computer-aided verification. draft book manuscript, Sept. 1999.

[3] R. Alur and T. A. Henzinger. Reactive modules. *Formal Methods in Sys. Design*, 15:7–48, July 1999.

[4] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4), February 2001.

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

[6] R. Bharadwaj. A framework for the formal analysis of multi-agent systems. In *Proc. Formal Aspects of Multi-Agent Systems (FAMAS'03)*, Warsaw, Poland, Apr. 2003. ETAPS 2003.

[7] R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering Journal*, 6(1), Jan. 1999.

[8] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for automatic invariant checking. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000)*, Berlin, Mar. 2000.

[9] K. M. Chandy and J. Misra. Proofs of networks of processes. *IEEE Trans. Softw. Engin.*, SE-7(4):417–426, Apr. 1981.

[10] Y. Choi and M. P. E. Heimdahl. Model checking $RSML^{-e}$ requirements. In *Proc. 7th IEEE Int'l Symp. on High Assurance Sys. Engin. (HASE'02)*, Tokyo, Japan, Oct. 2002.

[11] A. Cimatti et al. NuSMV 2: An open source tool for symbolic model checking. In *Proc. Computer-Aided Verification (CAV 2002)*, Copenhagen, Denmark, July 2002.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT press, 1999.

[13] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254, pages 72–83. Springer Verlag, 1997.

[14] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Softw. Engin.*, 18(9):785–793, Sept. 1992.

[15] C. Heitmeyer. Software Cost Reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., New York, NY, second edition, 2002.

[16] C. Heitmeyer, M. Archer, R. Bharadwaj, R. Jeffords, and J. Kirby, Jr. Tools for tabular specification and analysis of requirements. Technical report, Naval Research Laboratory, Washington, DC, 2003. To appear.

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

[18] C. Heitmeyer, J. Kirby, Jr., 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, 1998.

[19] 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, April–June 1996.

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

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

[22] R. D. Jeffords and C. L. Heitmeyer. An algorithm for strengthening state invariants generated from requirements specifications. In *Proc. of the Fifth IEEE International Symposium on Requirements Engineering*, Aug. 2001.

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

[24] 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)*. IEEE Computer Society Press, Dec. 1999.

[25] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In T. Margaria and W. Yi, editors, *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS'01)*, number 2031 in LNCS, pages 98–112. Springer-Verlag, Apr. 2001.

[26] Z. Manna et al. STeP: the Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Stanford Univ., Stanford, CA, June 1994.

[27] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, NY, 1995.

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

[29] K. L. McMillan. *Getting Started with SMV: User's Manual*. Cadence Berkeley Laboratories, Berkeley, CA, Mar. 1999.

[30] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Comput. Prog.*, 37:279–309, 2000.

[31] K. S. Namjoshi and R. J. Trefler. On the completeness of compositional reasoning. In E. A. Emerson and A. P. Sistla, editors, *Proc. Computer-Aided Verification (CAV'00)*, Chicago, IL, July 2000. LNCS 1855, Springer-Verlag.

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

[33] A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. R. Apt, editor, *Proc. NATO Adv. Study Inst. on Logics and Models of Concurrent Systems*, La Colle-sur-Loup, France, Oct. 1984. Springer-Verlag.

[34] J. Ramish. Empirical studies of compositional abstraction. Technical report, Naval Research Laboratory, Washington, DC, 2003. Draft.

[35] H. Saïdi and N. Shankar. Abstract and model check while you prove. In *Proc. Computer-Aided Verification (CAV99)*, Trento,Italy, July 1999.

[36] N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab, SRI Intern., Menlo Park, CA, 1993.

[37] A. J. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application for monitoring systems. Technical Report TR 90-276, Queen's Univ., Kingston, ON, Canada, 1990.

APPENDICES

A. SCR OVERVIEW

This appendix briefly reviews the state machine model that underlies the SCR requirements model and gives examples of the tables used to define the values of the dependent variables in SCR specifications. Example tables are presented defining the values of the three dependent variables in the CCS specification introduced in Section 2.

A.1 SCR Requirements Model

An SCR requirements specification describes both the system environment, which is nondeterministic, and the required system behavior, which is usually deterministic [19]. The environment contains quantities that the system monitors, represented as *monitored variables*, and quantities that the system controls, represented as *controlled variables*. The environment nondeterministically produces a sequence of monitored events, where a *monitored event* signals a change in the value of some monitored variable. The system, represented in the model as a state machine, begins execution in some initial state and then responds to each monitored event in turn by changing state. In SCR as in Esterel [5], the system behavior is assumed to be *synchronous*: the system completely processes one set of inputs before processing the next set. In SCR (in contrast to Esterel which allows more than one input to change per transition), the *One Input Assumption* allows at most one monitored variable to change from one state to the next.

Our state machine model, a special case of Parnas' Four Variable Model (FVM), [32] uses two relations of the FVM, NAT and REQ, to define the required system behavior. NAT, which describes the natural constraints on the system behavior, such as constraints imposed by physical laws and the system environment, defines the possible values of the monitored and controlled variables. REQ defines additional constraints on the system as relations the system must maintain between the monitored and controlled variables. To specify REQ concisely, our SCR model contains two types of auxiliary variables: *mode classes*, whose values are called *modes*, and terms. Each mode is an equivalence class of system states useful in specifying the required system behavior. A *term* is a state variable whose value depends on monitored variables, mode classes, or other terms.

The SCR model represents a system as a state machine $\Sigma = (S, S_0, E^m, T)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of monitored events, and T is the transform describing the allowed state transitions [19]. In our model, the transform T is a function that maps a monitored event $e \in E^m$ and the current state $s \in S$ to the next state $s' \in S$. Further, a *state* is a function that maps each *state variable*, i.e., each monitored or controlled variable, mode class, or term, to a type-correct value; a *condition* is a predicate defined on a system state, and an *event* is a predicate requiring that two system states differ in the value of at least one state variable.

When the value of a state variable (or a condition) changes, we say that an event "occurs". The notation "`@T(c) WHEN d`" denotes a *conditioned event*, which is defined by

$$\texttt{@T(c) WHEN d} \overset{\text{def}}{=} \neg c \wedge c' \wedge d,$$

where the unprimed conditions c and d are evaluated in the current state and the primed condition c' is evaluated in the next state.

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

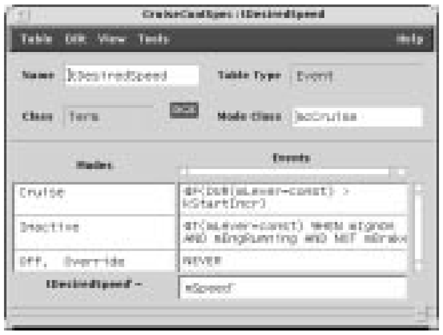
A.2 The SCR Tables

The transform T is the composition of smaller functions called *table functions*, which are derived from the condition tables, event tables, and mode transition tables in SCR requirements specifications. These tables define the values of the *dependent variables*—the controlled variables, mode classes, and terms. For T to be well-defined, no circular dependencies are allowed in the definitions of the dependent variables. The variables are partially ordered based on the dependencies among the next state values.

Each table defining a term or controlled variable is either a condition or an event table. A *condition table* associates a mode and a condition in the next state with a variable value in the next state, whereas an *event table* associates a mode and a conditioned event with a variable value in the next state. Each table defining a mode class is a *mode transition table*, which associates a source mode and an event with a destination mode. Our formal model requires the information in each table to satisfy certain properties. These properties guarantee that each table describes a total function [19].

To illustrate the SCR tabular notation, three example tables are presented. These tables define the values of the three dependent variables in the Cruise Control System specification—`mcCruise`, `tDesiredSpeed`, and `cThrottle`.

Table 1 is an event table defining the term `tDesiredSpeed` as a function of the current mode and the monitored variables. The second row states that if the system is in `Inactive` and the driver changes the lever to `const` with the ignition on, the engine running, and the brake off, the new value of `tDesiredSpeed` equals the `mSpeed`, the speed of the automobile. The first row contains the `DUR` operator, introduced in [37] to describe time-dependent behavior. In the first row, the expression "$\texttt{DUR}(\texttt{mLever} = \texttt{const}) > \texttt{kStartIncr}$" is true iff the length of time that the lever has been in the `const` position exceeds the constant `kStartIncr`. The event described by "$\texttt{@F}(\texttt{DUR}(\texttt{mLever} = \texttt{const}) > \texttt{kStartIncr})$" occurs when the lever is changed to some position other than `const` after having been in `const` for more than `kStartIncr` milliseconds. The first row states that when the system is in `Cruise` and this conditioned event occurs, the new value of `tDesiredSpeed` is the actual speed. The presence of `NEVER` in the third row indicates that no event can change the value of `tDesiredSpeed` when the system is in either `Off` or `Override`.


CruiseCntlSpec : tDesiredSpeed

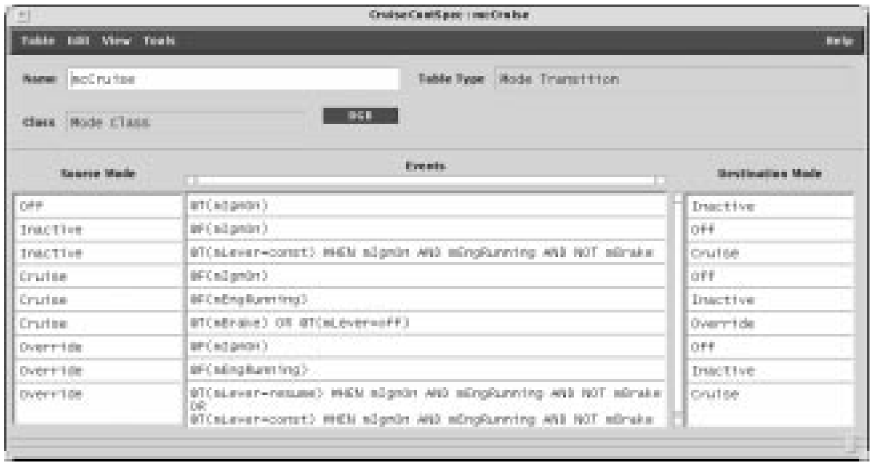
Name: tDesiredSpeed   Table Type: Event
Class: Term   Mode Class: mcCruise

Modes	Events
Cruise	@F(DUR(mLever=const) > kStartIncr)
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake
Off, Override	NEVER
tDesiredSpeed' =	mSpeed'

Table 1: Event table defining the term `tDesiredSpeed`

Table 2 is a mode transition table defining the new value of the mode class `mcCruise` as a function of the current mode and the monitored variables. For example, the first row of the table states that if the current mode is `Off` and the driver turns the ignition on, the new mode is `Inactive`, while the third row states that if the system is in `Inactive` and the driver puts the lever in `const` with the ignition on, the engine running, and the brake off, the system enters `Cruise` mode.

Table 3 is a condition table defining the value of controlled variable `cThrottle` as a function of the modes, the monitored vari-

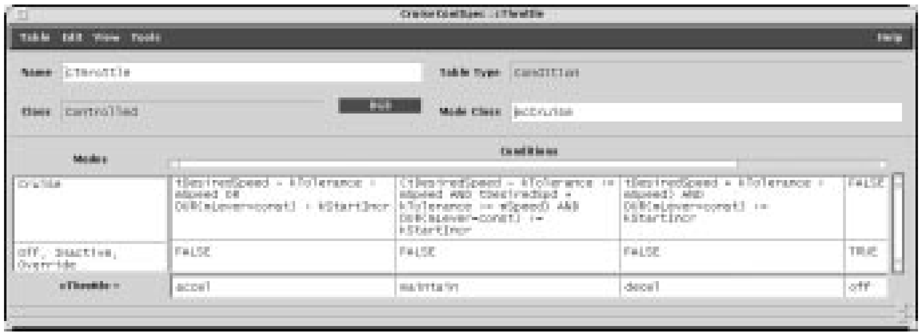


CruiseContSpec::mcCruise

Name: mcCruise
Table Type: Mode Transition
Class: Mode Class

Source Mode	Events	Destination Mode
Off	@T(mIgnOn)	Inactive
Inactive	@F(mIgnOn)	Off
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Cruise	@F(mIgnOn)	Off
Cruise	@F(mEngRunning)	Inactive
Cruise	@T(mBrake) OR @T(mLever=off)	Override
Override	@F(mIgnOn)	Off
Override	@F(mEngRunning)	Inactive
Override	@T(mLever=resume) WHEN mIgnOn AND mEngRunning AND NOT mBrake OR @T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise

Table 2: Mode transition table defining the mode class `mcCruise`



CruiseContSpec::cThrottle

Name: cThrottle
Table Type: Condition
Class: Controlled
Mode Class: mcCruise

Modes	Conditions			
Cruise	tDesiredSpeed - kTolerance > mSpeed OR DUR(mLever=const) > kStartIncr	(tDesiredSpeed - kTolerance <= mSpeed AND tDesiredSpd + kTolerance >= mSpeed) AND DUR(mLever=const) <= kStartIncr	tDesiredSpeed + kTolerance < mSpeed) AND DUR(mLever=const) <= kStartIncr	FALSE
Off, Inactive, Override	FALSE	FALSE	FALSE	TRUE
cThrottle =	accel	maintain	decel	off

Table 3: Condition table defining the controlled variable `cThrottle`

ables, and the term `tDesiredSpeed`. The first row states that in `Cruise` mode the system should accelerate the automobile if the desired speed minus some constant tolerance `kTolerance` exceeds the actual speed or if the time the lever is in `const` exceeds `kStartIncr`, and gives similar conditions for when the system should decelerate the automobile or maintain the current speed. The second row states that in modes other than `Cruise`, the throttle is `off`.

B. SOUNDNESS AND COMPLETENESS

If we apply the Manna/Pnueli Basic Rule to both (5) and (6) of the Better Assume-Guarantee Rule (BAG) presented in Section 4, we obtain the Theorem Proving Assume-Guarantee (TAG) Rule:

$$\begin{array}{ll} a \in \mathrm{Inv}(\Sigma_1), b \in \mathrm{Inv}(\Sigma_2), \Theta_1 \wedge \Theta_2 \Rightarrow p \wedge q, & \\ p \wedge (b \wedge a' \wedge b' \wedge q \wedge \rho_1) \Rightarrow p' & (8) \\ \underline{q \wedge (a \wedge b \wedge a' \wedge b' \wedge p \wedge p' \wedge \rho_2) \Rightarrow q'} & (9) \\ (p \wedge q) \in \mathrm{Inv}(\Sigma_1 || \Sigma_2) & \end{array}$$

Because (8) is stronger than (5) and (9) is stronger than (6), the soundness of both rules (as well as the two rules where just one of (5) and (6) is replaced via the Manna/Pnueli Basic Rule) follows from the soundness of the BAG Rule. As a special case, the soundness of the BCP Rule follows from the soundness of the CMP Rule. Similarly, the completeness of both rules (as well as the two rules, where just one of (5) and (6) is replaced via the Manna/Pnueli Basic Rule) follows from the completeness of the TAG Rule.[4]

To establish the soundness of the BAG rule, we require a lemma stating that any reachable state s of $\Sigma_1 || \Sigma_2$ is also a reachable state of both system $(S, a \wedge b \wedge q \wedge \Theta_1, a \wedge b \wedge a' \wedge b' \wedge q \wedge \rho_1)$ and system $(S, a \wedge b \wedge p \wedge \Theta_2, a \wedge b \wedge a' \wedge b' \wedge p \wedge p' \wedge \rho_2)$. The proof of this lemma is a straightforward induction on the number of steps from an initial state to reach s. (We omit the proof.)

The relative completeness of the TAG Rule is established by choosing $a = \mathcal{R}_1$ and $b = \mathcal{R}_2$, where $\mathcal{R}_i$ is the characteristic predicate of the set of reachable states in Σ_i, $i = 1, 2$. In proving completeness, we are given the conclusion of the TCP Rule: $p \wedge q \in \mathrm{Inv}(\Sigma_1 || \Sigma_2)$. This may be expressed equivalently as $\mathcal{R}_1 \wedge \mathcal{R}_2 \Rightarrow p \wedge q$. We must prove that $\Theta_1 \wedge \Theta_2 \Rightarrow p \wedge q$ holds and that (8) and (9) hold. Because the initial states in each system are a subset of the reachable states, i.e., $\Theta_i \Rightarrow \mathcal{R}_i$, it is always true that $\Theta_1 \wedge \Theta_2 \Rightarrow p \wedge q$. After the substitutions indicated above, (8) and (9) become the respective (10) and (11):

$$p \wedge (\mathcal{R}_1 \wedge \mathcal{R}_2 \wedge \mathcal{R}_1' \wedge \mathcal{R}_2' \wedge q \wedge \rho_1) \Rightarrow p' \qquad (10)$$

$$q \wedge (\mathcal{R}_1 \wedge \mathcal{R}_2 \wedge \mathcal{R}_1' \wedge \mathcal{R}_2' \wedge p \wedge p' \wedge \rho_2) \Rightarrow q' \qquad (11)$$

Thus, both (10) and (11) follow immediately from the primed version of our given fact, i.e., $\mathcal{R}_1' \wedge \mathcal{R}_2' \Rightarrow p' \wedge q'$.

[4]The BCP and CMP Rules are not complete but can easily be made complete by using additional auxiliary invariants.