Comput Syst Sci & Eng (2005) 1: 19-35
© 2005 CRL Publishing Ltd

International Journal of
Computer Systems
Science & Engineering

NRL Release Number 04-1226-3131

Tools for constructing requirements specifications: The SCR toolset at the age of ten

Constance Heitmeyer, Myla Archer, Ramesh Bharadwaj and Ralph Jeffords

Center for High Assurance Computer Systems, Naval Research Laboratory, Washington, DC 20375, USA
Email: {heitmeyer, archer, ramesh, jeffords}@itd.nrl.navy.mil

While human effort is critical to creating requirements specifications and human inspection can detect many specification errors, software tools find errors inspections miss and also find certain classes of errors more cheaply. This paper describes a set of tools for constructing and analyzing requirements specifications in the SCR (Software Cost Reduction) tabular notation. The tools include a specification editor, a consistency checker, a simulator, and tools for verifying application properties – including a model checker, a verifier, a property checker based on decision procedures, and an invariant generator. The paper also describes the practical systems to which the tools are being applied as well as some new tools recently added to the toolset, e.g. a tool that constructs a sound and complete abstraction from a property and a specification. To illustrate the tools, the paper describes their use in developing a requirements specification for an automobile cruise control system.

Keywords: requirements, specifications, tools, SCR, formal methods, verification, consistency checking, formal models

1. INTRODUCTION

The need for unambiguous, easy-to-understand notations for specifying and analyzing the requirements of systems is widely recognized. Tables have been demonstrated to offer a precise, relatively compact notation for specifying system requirements in a wide range of applications, including avionics systems, systems for controlling nuclear power plants, and telephone networks (see, for example, [32, 47, 13, 53, 34]). Developers have also found tabular notations easier to write and to understand than alternative notations, such as Z and Petri nets. In addition, tables can be assigned a precise mathematical semantics and thus can be analyzed either manually or mechanically to expose defects in requirements specifications. Finally, tabular notations have been demonstrated to scale to practical systems.

In 1978, the requirements document for the flight program of the A-7 aircraft [32, 33] introduced a special tabular notation for writing specifications. Part of the SCR (Software Cost Reduction) requirements method, this notation was designed to document the requirements of real-time, embedded systems concisely and unambiguously. During the 1980s and 1990s, SCR tables were used by several organizations in industry and government, e.g. Grumman [47], Bell Laboratories [34], Ontario Hydro [53], the Naval Research Laboratory [31], and Lockheed [13], to document the requirements of many practical systems, including a submarine communications system [31], the shutdown system for the Darlington nuclear power plant [53], and the flight program for Lockheed's C-130J aircraft [13]. The Lockheed specification contains over 1000 tables and the corresponding flight program over 250K lines of Ada [59] – solid evidence that the tabular notation scales.

Analysis of these tables for errors was largely manual. A serious problem with manual inspections is their high cost – the inspection of tables in the certification of the Darlington shutdown system, for example, cost millions of dollars. Moreover, manual inspections often miss certain classes of specification errors software tools detect. In a study conducted in 1996, a mechanized analysis of the A-7 requirements specification, which had previously undergone manual inspections by two independent review teams, exposed 17 missing cases and 57 instances of unwanted nondeterminism within a few minutes [26]. These flaws were detected even though the tabular format of the specification was designed to make such flaws obvious. In a later study in 1998, software tools exposed 28 errors, many of them serious, in a requirements specification of a commercial flight guidance system [49]. The detection of so many errors was surprising given that the specification, in the words of the project leader [48], "represented our best effort at producing a correct specification manually."

While human effort is critical to creating the specifications and human inspection can detect many errors, effective and wider usage of tabular notations in industrial settings requires powerful, robust tool support. Not only can software tools find errors inspections miss, they can also do so more cheaply. To explore what form tabular tools should take, we have developed a suite of software tools for constructing and analyzing requirements specifications in the SCR tabular notation. The tools include a *specification editor* for creating the tabular specification, a *simulator* for validating that the specification satisfies the customer's intent, a *dependency graph browser* for understanding the relationship between different parts of the specification, and a *consistency checker* to analyze the specification for properties such as syntax and type correctness, determinism, case coverage, and lack of circularity. The toolset also contains a *model checker*, a *verifier*, a *property checker*, and an *invariant generator,* all of which are useful in analyzing specifications for critical application properties, such as safety and security properties.

To make the tools accessible to software developers, we have followed two major guidelines in designing the tools. First, the effort and expertise needed to apply the tools have been minimized. Using the tools requires neither theorem proving skills nor advanced mathematical training. Nor do tool users need to understand special logics, such as temporal or higher order logics. Second, the tools are designed to be "pushbutton", i.e. they analyze the specifications automatically and provide useful feedback when an error is detected; for many of the tools, this feedback is expressed in the original tabular notation.

In addition to ease of use, the SCR tools have two more attributes which distinguish them from other tools developed by researchers. First, unlike most research tools, the SCR tools are designed to work together. This is especially true of the SCR *basic tools,* i.e. the specification editor, consistency checker, dependency graph browser, and simulator, which we developed from scratch. The basis for tool integration is a shared formal semantics [26, 30], i.e. the state machine model that underlies the SCR tabular notation, and a shared interpretation of the tabular notation. Second, unlike the vast majority of tools developed by researchers, some of the SCR tools, namely, the basic tools, are already being used by industry to develop real-world software.

This paper has three goals. Since the introduction of the SCR tools in a 1995 conference paper [23], many papers have been published describing new SCR tools and tool extensions [26, 28, 37, 15, 29, 6], the use of externally developed tools to analyze SCR specifications [7, 25, 1], and the application of the SCR tools to practical systems, e.g. [25, 39, 27]. The first goal of this paper is to present in a single document an overview of the current toolset and to illustrate via a simple cruise control example how one can use the SCR tools to create a software requirements specification, to detect specification errors, to validate a specification (e.g. using simulation), and to prove (or refute) that a specification satisfies one or more application properties.

A second goal is to describe recent enhancements to the toolset. These include (1) the integration of a property checker called Salsa [6] to do consistency checking, to verify properties, and to return candidate counterexamples when a property cannot be proven true (see Sections 3.3 and 3.5.3), (2) the implementation of a tool that uses the algorithms described in [37] and [38] to construct state invariants automatically (see Section 3.4.2), (3) the integration of the TAME interface [2] to PVS [57] to verify properties and to provide user feedback when a proof fails (see Section 3.5.2), and (4) the design and implementation of a tool that automatically constructs a sound and complete abstraction from a property and an SCR specification (see Section 3.5.1). A third goal is to describe the current use of the SCR tools in software practice. In recent years, the tools have been used routinely by a large U.S. company to construct and analyze requirements specifications, and several new applications of the tools to practical systems are currently in progress.

While the toolset enhancements described in the paper are straightforward extensions of results published in previous papers, tool support for the development of reliable, correct software and software systems is of growing importance. For researchers and tool builders to be able to exploit our results and to ease the transition of technology based on SCR into industrial practice, what is needed is a clear, accessible description of the SCR tools, the concepts on which they are based, and how they can be used together to construct a consistent, complete, and correct requirements specification. This paper provides such a description and thereby a link between academic research and industrial practice.

The paper is organized as follows. To provide background, Section 2 reviews the formal model that underlies the tools and the different tables used in SCR specifications. Section 3, the body of the paper, introduces a process for developing a requirements specification, describes how that process can be applied using the SCR tools, and illustrates the tools using the cruise control example. This section describes how the tools can be used to create a tabular requirements specification, to detect specification errors, to validate a specification (e.g. using simulation), and to prove properties about the specification. Section 4 describes how the tools have been used to specify and to analyze requirements for practical systems. Section 5 discusses tool support for rapid prototyping, source code and test case generation, the trade-offs between different analysis techniques, and the benefits of a suite of tools. Section 6 describes related tools and toolsets. Finally, Section 7 presents some conclusions.

2. BACKGROUND

The A-7 requirements document introduced many features of the SCR requirements method – the tabular notation, the underlying state machine model, and special constructs for specifying requirements, such as conditions, events, mode classes, and terms. Since the publication of the A-7 document, researchers, including Faulk [14, 12] and Parnas [55], have extended the original SCR method and strengthened its formal foundation. Based on this foundation, we have formulated the SCR requirements model, which is briefly reviewed in this section. For a complete exposition of the model, see [30, 26]. This section also describes the different classes of SCR tables and introduces a simple automobile cruise control system, used as a running example in the remainder of the paper.

2.1 The SCR formal model

An SCR requirements specification describes both the system environment, which is nondeterministic, and the required system behavior, which is usually deterministic [26, 30]. 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) [55] 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 the required constraints that the system must maintain between the monitored and controlled variables. To specify REQ concisely, the SCR model defines 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.

Our 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 [30, 26]. 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 $s' \in S$; 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 single system state; and an *event* is a predicate defined on two system states which implies that the two states are different.

When the value of a state variable (or more generally a condition) changes, we say that an event "occurs". Given conditions c and d, 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 c evaluated in the next state. (In this paper, an unprimed variable refers to the variable value in the current state, whereas a primed variable refers to the variable value in the next state.) Often, the `WHEN d` is missing; in such cases, `@T(c)` $\overset{\text{def}}{=}$ `@T(c) WHEN` *true*. The notation "`@F(c)`" is defined by `@F(c)` $\overset{\text{def}}{=}$ `@T(¬c)`. Informally, "`@T(c)`" means that c becomes true and "`@F(c)`" means that c becomes false. The notation "`@C(x)`" denotes the event "variable x has changed value".

2.2 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; 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 (in particular, the Disjointness and Coverage Properties which are described in Section 3.1) guarantee that each table describes a total function [26].

2.3 Example: The Cruise Control System

To demonstrate the SCR tables and tools, we consider an SCR specification of a simple version of a real automobile cruise control system [40]. 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 shows how SCR state variables can be used to specify the CCS requirements. The monitored 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 brake and cruise control lever. The distinguished monitored variable `time` indicates time passage. The controlled 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 of the required relation between the monitored and controlled variables concise.

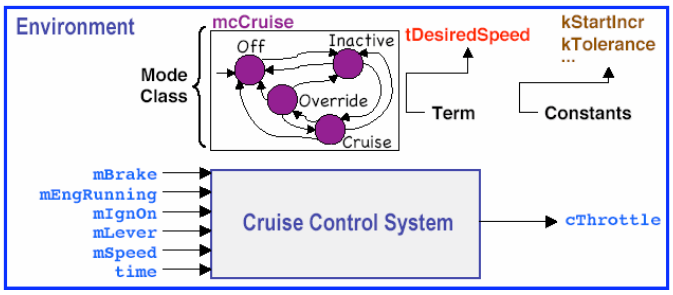


Figure 1 Specifying the Cruise Control System in SCR

3. THE SCR TOOLS

Section 3.1 introduces a systematic process for developing a requirements specification and then describes how the SCR tools can be used to support each step of this process. Sections 3.2–3.5 illustrate the tools by showing how they can be used to create, debug, validate, and verify a tabular SCR specification of the Cruise Control System introduced in Section 2.3.

3.1 A process for constructing requirements specifications

We envision the following four-step process for constructing a requirements specification. An idealization of a real-world process [54], the process describes a logical sequence in which the SCR tools can be used to create, debug, validate, and verify a tabular requirements specification. First, a specification is developed that represents the required system behavior in a notation, such as a tabular notation, with a well-defined semantics. Second, the specification is analyzed to detect and correct well-formedness errors, e.g. syntax and type errors, missing cases, unwanted nondeterminism, and circular definitions. Third, the specification is validated to ensure that it captures the customers' intent. Fourth, the specification is analyzed first to detect violations of, and then to verify, application properties.

Eight tools in the SCR toolset support this four-step process. To begin, the user invokes the Specification Editor to create the two types of tables contained in an SCR requirements specification: *function tables*, which define the values of the dependent variables, and tables called *dictionaries*, which contain other information in the specification, such as variable declarations, environmental assumptions, and type definitions.

To detect and correct well-formedness errors, the user may invoke the Consistency Checker [26], the property checker Salsa, and the Dependency Graph Browser [28]. The Consistency Checker detects many well-formedness errors automatically. To perform the most computationally complex checks, the checks for the *Disjointness Property* (no nondeterminism) and the *Coverage Property* (no missing cases), the Consistency Checker uses an extension of the semantic tableaux algorithm [58]. Because this technique cannot analyze some formulae containing arithmetic expressions, the property checker Salsa was developed. Salsa applies a tightly integrated set of *decision procedures*, algorithms that establish the truth or falsity of logic formulae, to check specifications for Disjointness and Coverage. Once consistency checking exposes a well-formedness error, the user, to correct the error, often needs to understand the relationship between different parts of the specification. To do this, the user may invoke the Dependency Graph Browser, which displays a graph showing the dependencies among the variables in the SCR specification.

To validate the specification, the user may invoke the Simulator [23, 28, 29] or the Invariant Generator [37, 38]. To expose inconsistencies between the intended system behavior and the behavior captured in the specification, the user may run scenarios, where a *scenario* is a sequence of monitored events, through the Simulator. Alternatively, the user may generate *state invariants* – properties true of every reachable state – from the specification using the Invariant Generator and then ask customers or others familiar with the system under development whether the generated invariants accurately describe the intended behavior. When inconsistencies between the desired behavior and the invariants are found, the specification can be modified to remove the inconsistencies.

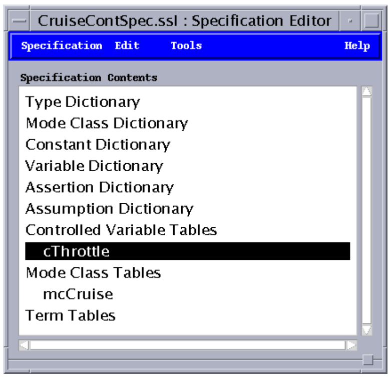
To analyze an SCR specification for application properties, the user can apply any one of three tools – a model checker, the verifier TAME [2], or the property checker Salsa. Such analysis may expose a property violation: either the property or the specification is incorrect. Once such errors are corrected, the user may be able to verify selected properties. The user can invoke a model checker, such as the explicit state model checker SPIN [35], to analyze a finite state model of the SCR specification for application properties. Alternatively, the user may apply the verifier TAME, a specialized interface to the general-purpose theorem prover PVS, to prove properties automatically. As a third alternative, the user may invoke Salsa, which, in addition to analyzing the specification for Disjointness and Coverage, can also check specifications for application properties. In applying either TAME or Salsa to verify application properties, the state invariants generated by the Invariant Generator may be used as auxiliary lemmas.

3.2 Constructing the specification

To construct a function table, the user invokes the Specification Editor (see Figure 2) and pushes the Edit button to create a table, selects the table class (e.g. event, condition, or mode transition), and then enters the appropriate information into the table. Tables 1–3, all constructed with the Specification Editor, are function tables defining the values of the three dependent variables in the CCS specification: `mcCruise`, `tDesiredSpeed` and `cThrottle`.

Table 1 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 2 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`

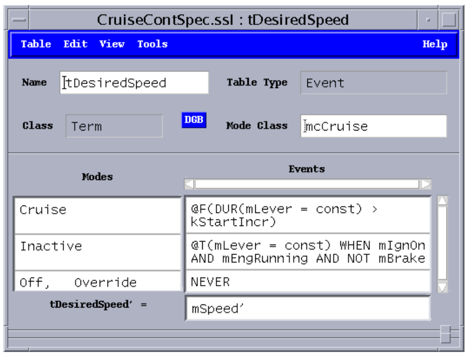


Specification Contents

- Type Dictionary
- Mode Class Dictionary
- Constant Dictionary
- Variable Dictionary
- Assertion Dictionary
- Assumption Dictionary
- Controlled Variable Tables
  - cThrottle
- Mode Class Tables
  - mcCruise
- Term Tables

Figure 2 The Spec Editor lists the contents of the CCS specification

Table 2 Event table defining the term `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′

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 `mSpeed`, the automobile's current speed. The first row contains an operator called `DUR` (or `DURATION`), useful for specifying timeouts [61]. The condition appearing in this row, "`DUR(mLever=const) > kStartIncr`", is true iff the length of time that the lever has been in `const` exceeds the constant `kStartIncr`. The event "`@F(DUR(mLever=const) > kStartIncr)`" occurs when the lever is changed to some position other than `const` after being 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 `tDesired-Speed` when the system is in either `Off` or `Override`.

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

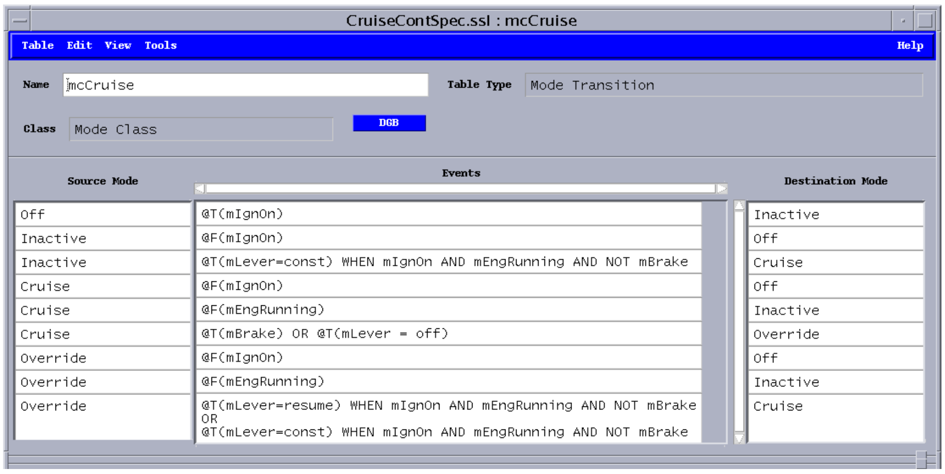
As shown in Figure 2, the editor lists the dictionaries and function tables under the label "Specification Contents". By double clicking on a dictionary type, the user can display, and add entries to, the dictionary of that type. By double clicking on a function table type, the user can list all variables of that type. In Figure 2, the user has clicked on the entry, "Controlled Variable Tables," and in response the editor lists the single controlled variable in the CCS specification, `cThrottle`. Double clicking on "`cThrottle`" displays Table 3.

Each SCR specification contains six dictionaries. Figures 3–6 show the *constant dictionary*, the *type dictionary*, the *mode class dictionary* and the *variable dictionary* for the CCS. The constant dictionary in Figure 3 defines the values and types of five constants, including the two constants, `kStartIncr` and `kTolerance`, that appear in Tables 2 and 3. The type dictionary in Figure 4 defines three user-defined types: two enumerated types `yLever` and `yThrottle` and a real-valued type `ySpeed` with values between 0.0 and some constant `kMaxSpeed`. The mode class dictionary in Figure 5 lists the four modes in the single mode class `mcCruise` and identifies the initial mode as `Off`. The variable dictionary in Figure 6 defines the types, initial values, and accuracy requirements of the state variables other than `mcCruise`: the six monitored variables, the term, and the single controlled variable.* (While listed in Figure 6, accuracy requirements are not used at present in the automated analyses.)

Figures 7–8 show the other two dictionaries in SCR specifications: the *environmental assumption dictionary*, which

*Not shown in Figure 6 (but appearing in Figures 9, 12, 13 and 15), is "`_DUR_mLever_EQ_const`", an extra term that the toolset creates to denote `DUR(mLever=const)`, the length of time the cruise control lever has been in the `const` position.

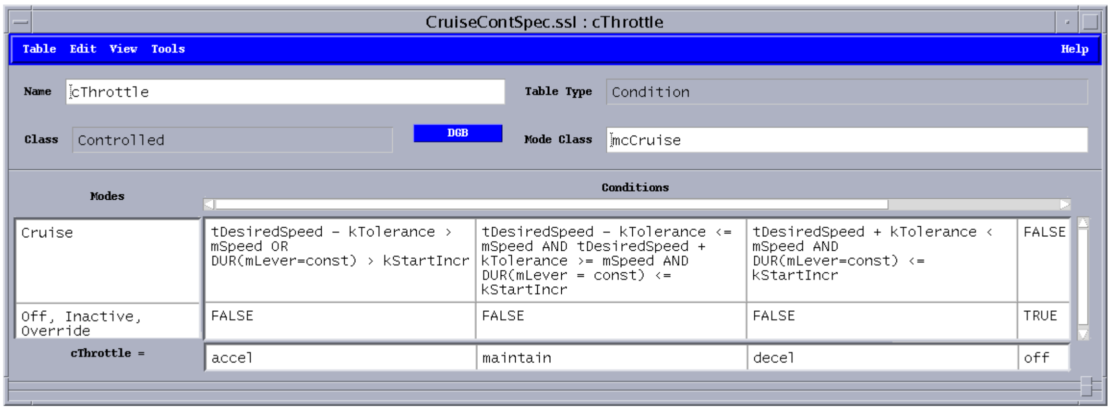
Table 1 Mode transition table defining the mode class `mcCruise`



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

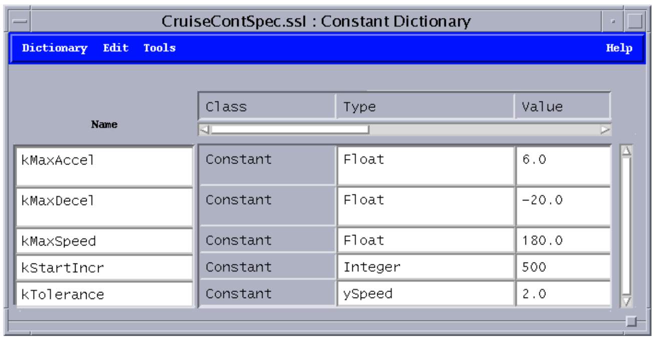
Source Mode	Events	Destination Mode
Off	@T(mIgnOn)	Inactive
Inactive	@F(mIgnOn)	Off
Inactive	@T(mLever=const) WHEN mIgnOn AND mEngRunning AND NOT mBrake	Cruise
Cruise	@F(mIgnOn)	Off
Cruise	@F(mEngRunning)	Inactive
Cruise	@T(mBrake) OR @T(mLever = off)	Override
Override	@F(mIgnOn)	Off
Override	@F(mEngRunning)	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 3 Condition table defining the controlled variable `cThrottle`



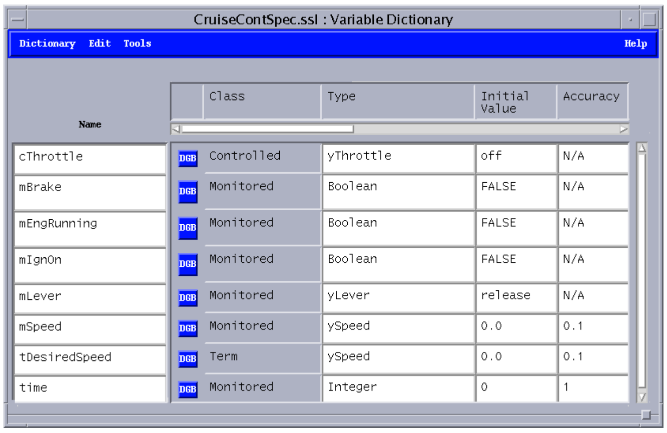
Name	cThrottle	Table Type	Condition
Class	Controlled	Mode Class	mcCruise

Modes	Conditions			
Cruise	tDesiredSpeed - kTolerance > mSpeed OR DUR(mLever=const) > kStartIncr	tDesiredSpeed - kTolerance <= mSpeed AND tDesiredSpeed + 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



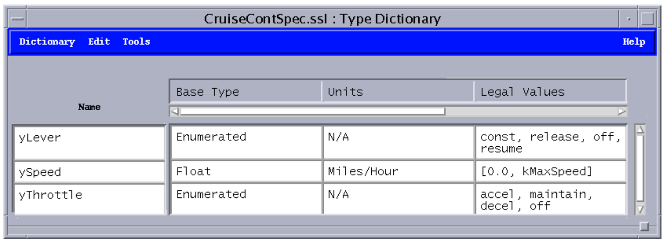
Name	Class	Type	Value
kMaxAccel	Constant	Float	6.0
kMaxDecel	Constant	Float	-20.0
kMaxSpeed	Constant	Float	180.0
kStartIncr	Constant	Integer	500
kTolerance	Constant	ySpeed	2.0

Figure 3 The Constant Dictionary for the CCS



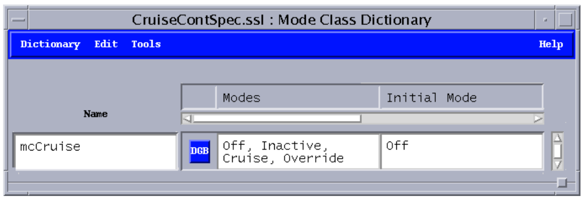
Name		Class	Type	Initial Value	Accuracy
cThrottle	DGB	Controlled	yThrottle	off	N/A
mBrake	DGB	Monitored	Boolean	FALSE	N/A
mEngRunning	DGB	Monitored	Boolean	FALSE	N/A
mIgnOn	DGB	Monitored	Boolean	FALSE	N/A
mLever	DGB	Monitored	yLever	release	N/A
mSpeed	DGB	Monitored	ySpeed	0.0	0.1
tDesiredSpeed	DGB	Term	ySpeed	0.0	0.1
time	DGB	Monitored	Integer	0	1

Figure 6 The Variable Dictionary for the CCS



Name	Base Type	Units	Legal Values
yLever	Enumerated	N/A	const, release, off, resume
ySpeed	Float	Miles/Hour	[0.0, kMaxSpeed]
yThrottle	Enumerated	N/A	accel, maintain, decel, off

Figure 4 The Type Dictionary for the CCS

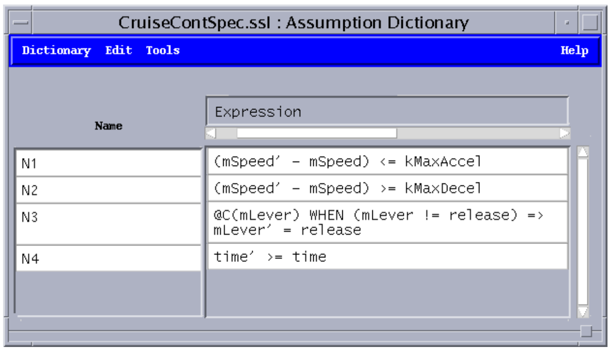


Name		Modes	Initial Mode
mcCruise	DGB	Off, Inactive, Cruise, Override	Off

Figure 5 The Mode Class Dictionary for the CCS

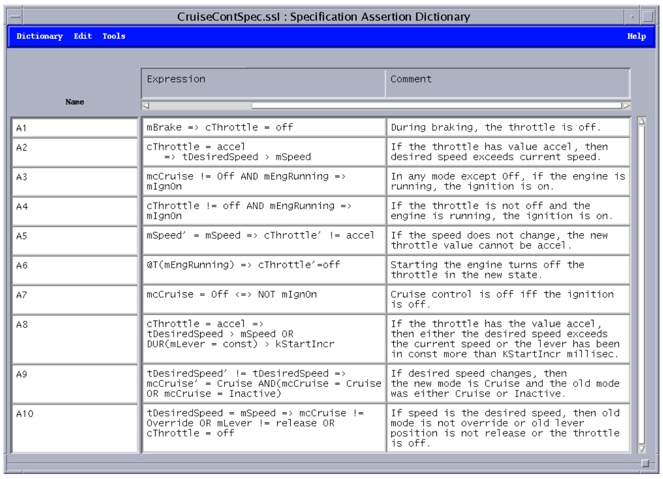
defines constraints imposed on the monitored and controlled variables by the system environment, and the *assertion dictionary*, which lists application properties that the specification is expected to satisfy. Checking whether the specification satisfies the assertions helps uncover both defects in the specifications and incorrectly formulated assertions. Both the assumptions and assertions are expressed as predicates on one or two states. This paper refers to predicates on a single state as *one-state properties*, predicates on two states as *two-state properties*, one-state properties that are true in every reachable state as *state invariants*, and two-state properties true in every reachable transition as *transition invariants*.

The environmental assumption dictionary (see Figure 7) lists four assumptions, all two-state properties, that constrain the behavior of the CCS. Assumptions N1 and N2 limit the rate at which the automobile can accelerate or decelerate from one state to the next using the constants `kMaxAccel` and `kMaxDecel`. Assumption N3 states that, starting from any position other than `release`, the driver can only move the lever to `release`. Assumption N4 states that the monitored variable `time` must be nondecreasing. The assertion dictionary (see Figure 8) lists ten assertions, three two-state properties (A5, A6, and A9) and seven one-state properties. Both



Name	Expression
N1	(mSpeed' - mSpeed) <= kMaxAccel
N2	(mSpeed' - mSpeed) >= kMaxDecel
N3	@C(mLever) WHEN (mLever != release) => mLever' = release
N4	time' >= time

Figure 7 The Environmental Assumption Dictionary for the CCS


CruiseContSpec.ssl : Specification Assertion Dictionary
Dictionary Edit Tools Help

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 new throttle value cannot be accel.
A6	@T(mEngRunning) => cThrottle′=off	Starting the engine turns off the throttle in the new state.
A7	mcCruise = Off <=> NOT mIgnOn	Cruise control is off iff the ignition is off.
A8	cThrottle = accel => tDesiredSpeed > mSpeed OR DUR(mLever = const) > kStartIncr	If the throttle has the value accel, then either the desired speed exceeds the current speed or the lever has been in const more than KStartIncr millisec.
A9	tDesiredSpeed′ != tDesiredSpeed => mcCruise′ = Cruise AND(mcCruise = Cruise OR mcCruise = Inactive)	If desired speed changes, then the new mode is Cruise and the old mode was either Cruise or Inactive.
A10	tDesiredSpeed = mSpeed => mcCruise != Override OR mLever != release OR cThrottle = off	If speed is the desired speed, then old mode is not override or old lever position is not release or the throttle is off.

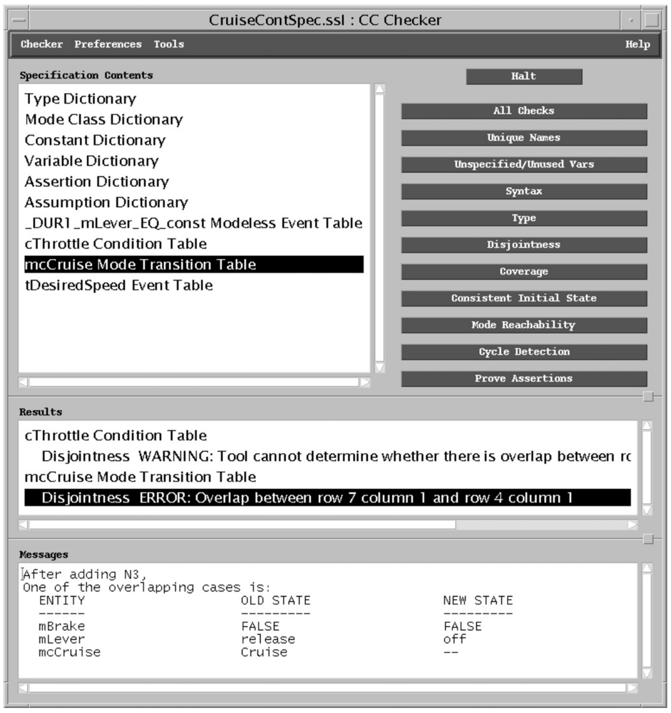
Figure 8 The Assertion Dictionary for the CCS

the assumptions and assertions are expressed in the same notation as the tables with the addition of "=>" for *implies* and "<=>" for *iff*. Hence, assertion A1 states that "if `mBrake` is true, then `cThrottle` is off", and assertion A7 states that "`mcCruise` is off iff `mIgnOn` is false."

3.3 Checking well-formedness

3.3.1 The Consistency Checker

After the tabular specification has been created, the consistency checker (CC) [26] can be invoked to check the specification for well-formedness (see Figure 9). Among the errors the CC detects are syntax and type errors, duplicate names, unspecified and unused variables, missing or inconsistent initial values, circular definitions, and violations of the Disjointness and Coverage Properties.



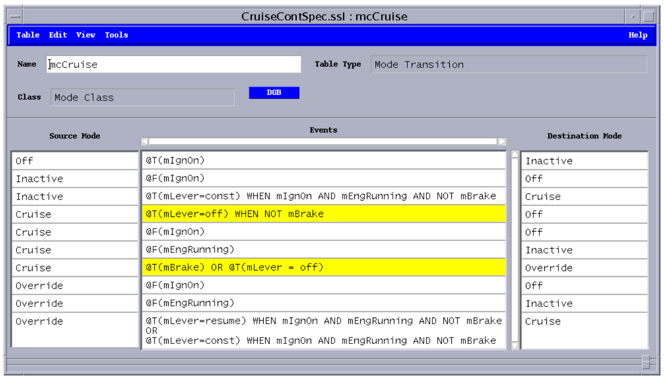
Type Dictionary
Mode Class Dictionary
Constant Dictionary
Variable Dictionary
Assertion Dictionary
Assumption Dictionary
_DUR1_mLever_EQ_const Modeless Event Table
cThrottle Condition Table
mcCruise Mode Transition Table
tDesiredSpeed Event Table

```
cThrottle Condition Table
  Disjointness  WARNING: Tool cannot determine whether there is overlap between rc
mcCruise Mode Transition Table
  Disjointness  ERROR: Overlap between row 7 column 1 and row 4 column 1
```

```
After adding N3,
One of the overlapping cases is:
  ENTITY                 OLD STATE               NEW STATE
  ------                 ---------               ---------
  mBrake                 FALSE                   FALSE
  mLever                 release                 off
  mcCruise               Cruise                  --
```

Figure 9 Applying the Consistency Checker to the CCS specification

Table 4 Mode transition table for `mcCruise` containing a disjointness error


CruiseContSpec.ssl : mcCruise
Table Edit View Tools Help
Name mcCruise
Table Type Mode Transition
Class Mode Class
DGB

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	@T(mLever=off) WHEN NOT mBrake	Off
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

To check a table for Disjointness or Coverage, the CC checks a logical formula for validity. For example, to check two conditions c_1 and c_2 in a row of a condition table for Disjointness, the CC checks the validity of the formula $\neg[c_1 \wedge c_2]$. To check conditions c_1, c_2, ..., c_n in a row of a condition table for Coverage, the tool checks the validity of the formula $[c_1 \vee c_2 \vee \ldots \vee c_n]$. To apply these checks, the CC adds extra information when necessary, such as the definitions of variables that appear in the formula and environmental assumptions (including the One Input Assumption). When the CC detects either a Coverage or a Disjointness error, it provides detailed feedback to facilitate correction of the error – it identifies the location of the error in the table and displays a counterexample in the form of a predicate on a state or state pair. In the case of a Disjointness error, the counterexample identifies a situation in which a table defines the value of a variable ambiguously. In the case of a Coverage error, the counterexample identifies a situation in which a table fails to define the required behavior.

To demonstrate how the CC handles a Disjointness error, we have modified Table 1 to include between rows 3 and 4 a new row stating, "If in `Cruise` the driver moves `mLever` to `off` when `mBrake` is `FALSE`, then the new mode is `Off`" (see Table 4) and then invoked the CC on the modified specification by pushing the CC's All Checks button. The Results Box in the middle of Figure 9 reports a Disjointness error. Double clicking on the line "`Disjointness ERROR...`" displays the table containing the error with the pair of entries that overlap highlighted (see Table 4) and a specific case of overlap (i.e. a counterexample) in the CC's Messages Box (see the bottom of Figure 9). This message states that any pair of adjacent states satisfying $\texttt{mLever} = \texttt{release} \wedge \texttt{mLever}' = \texttt{off} \wedge \neg\texttt{mBrake} \wedge \neg\texttt{mBrake}' \wedge \texttt{mcCruise} = \texttt{Cruise}$ satisfies both highlighted entries.

To check the two entries for Disjointness, the CC analyzed the formula, $\neg[(\texttt{@T(mLever = off)} \wedge \neg\texttt{mBrake}) \wedge (\texttt{@T(mBrake)} \vee \texttt{@T(mLever = off)})]$, which can be rewritten as $\neg[(\texttt{@T(mLever = off)} \wedge \neg\texttt{mBrake} \wedge \texttt{@T(mBrake)}) \vee (\texttt{@T(mLever = off)} \wedge \neg\texttt{mBrake} \wedge \texttt{@T(mLever = off)})]$. The first disjunct inside the square brackets simplifies to *false* due to the One Input Assumption. The second disjunct can be simplified to $\texttt{@T(mLever = off)} \wedge \neg\texttt{mBrake}$, which does not simplify to *false*. Hence, the original formula is not valid. This simplification of the second disjunct forms the basis for the counterexample displayed in the Messages Box.

While deleting the fourth row from Table 4 and invoking the CC removes the Disjointness error from the table defining `mcCruise`, the CC has identified another possible Disjointness problem in the table defining `cThrottle` – the message "`Disjointness WARNING...`" in Figure 9 indicates that the CC cannot decide whether the Disjointness property has been violated. The problem is the combination of numbers and arithmetic in the definition of `cThrottle`. Because CC was not designed to analyze predicates involving numerical constraints, it cannot reason about the relationship between predicates such as "`tDesiredSpeed – kTolerance > mSpeed`" and "`tDesiredSpeed + kTolerance < mSpeed`". To address this and other limitations of CC, the property checker was developed.

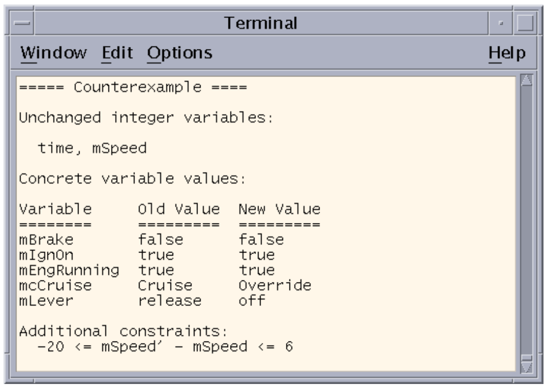
3.3.2 The Property Checker

The property checker Salsa [6] analyzes state machine descriptions in a language called SAL (SCR Abstract Language). The toolset automatically translates an SCR specification into SAL. Like CC, Salsa can analyze a specification for Disjointness and Coverage and thus checks formulae in the forms given in Section 3.3.1. Also like CC, Salsa may add information to the formula, such as environmental assumptions and definitions of variables that appear in the formula. After an initial term-rewriting phase, Salsa applies a set of decision procedures, using BDDs (Binary Decision Diagrams) to analyze propositional formulae and formulae containing enumerated type variables and a constraint solver to reason about linear integer arithmetic (Presburger arithmetic) formulae. Salsa's decision procedures for propositional logic and enumerated types are based on standard BDD algorithms [9]. Its decision procedure for integer linear arithmetic uses the automata-theoretic algorithm of Boudet and Comon [8], extended to handle negative numbers as proposed by Wolper [62]. Salsa integrates these decision procedures with the BDD algorithms.

Salsa decides that a property is valid if it is able to refute the negation of the formula being analyzed. In the initial step, a BDD is constructed which uses fresh boolean variables to represent each integer constraint and each enumerated type expression in a given formula. Searching for a feasible path in the BDD from root to *true* yields a set of associated integer constraints. If each such set is infeasible (i.e. has no solutions), the property holds. If not, an assignment of variable values that satisfies the constraints is returned as a possible counterexample.

Due to its more powerful decision procedures, Salsa can establish the Disjointness and Coverage Properties in certain situations where CC cannot. For example, in analyzing the CCS specification, Salsa was able to establish the Disjointness Property for Table 3, the condition table which defines `cThrottle`, whereas CC could not (see Figure 9). In addition, for propositional reasoning, the BDD-based algorithm of Salsa has proven more efficient than the modified semantic tableaux algorithm of CC. (However, because most consistency checking, even of specifications for practical systems, evaluates tables defining relatively simple functions, the tableaux-based CC continues to be useful.) To illustrate its utility for checking Disjointness, we applied Salsa to the modified mode transition table in Table 4. Salsa generated the same counter-example (see Figure 10) as CC with some added information.*

*Some of the added information in Figure 10 (e.g. that listed under Additional Constraints) overlaps information in the Assumption Dictionary, while other information (e.g. that `time` and `mSpeed` are unchanged) can be inferred from the One Input Assumption.



```
===== Counterexample ====

Unchanged integer variables:

  time, mSpeed

Concrete variable values:

Variable     Old Value  New Value
========     =========  =========
mBrake       false      false
mIgnOn       true       true
mEngRunning  true       true
mcCruise     Cruise     Override
mLever       release    off

Additional constraints:
  -20 <= mSpeed' - mSpeed <= 6
```

Figure 10 Counterexample returned by Salsa for violation of Disjointness

3.3.3 The Dependency Graph Browser

A limitation of tabular notations is that understanding how different tables are related can be difficult, especially for large specifications. To address this problem, the SCR toolset contains a Dependency Graph Browser (DGB), which constructs a graph showing the dependencies among the variables defined by the tables. Figure 11 contains a graph showing the variable dependencies in the CCS specification. In the graph, the monitored variables appear on the left, the controlled variables on the right, and the mode classes and terms in the middle. This graph shows, for example, that the value of `mcCruise` depends on its previous value (denoted by the backward arrow inside the node for `mcCruise`) and the values of four monitored variables and that the value of `cThrottle` depends (directly) on the values of `mcCruise`, `tDesiredSpeed`, and three monitored variables.

The dependency graph can be used to navigate the specification. For example, in Figure 11, double clicking on the node for `cThrottle` displays Table 3, while shift double clicking displays the variable dictionary in Figure 6 with the entry for `cThrottle` highlighted. In addition, the user can select a subgraph closed under the dependency relation (e.g. in Figure 11, the nodes for `mcCruise` and the four monitored variables upon which `mcCruise` depends), and save the SCR specification associated with that subgraph in a file. This

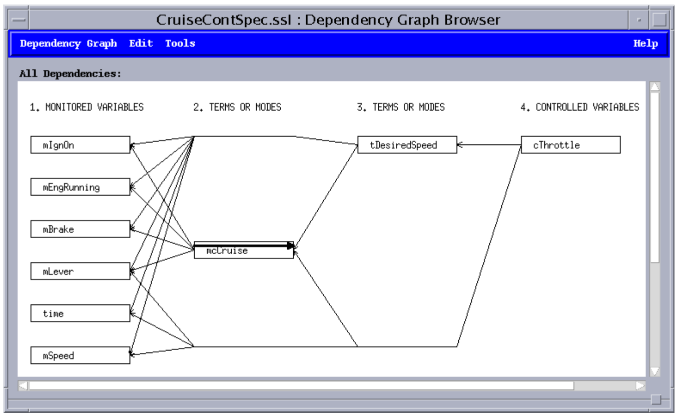


Figure 11 The dependency graph for the CCS

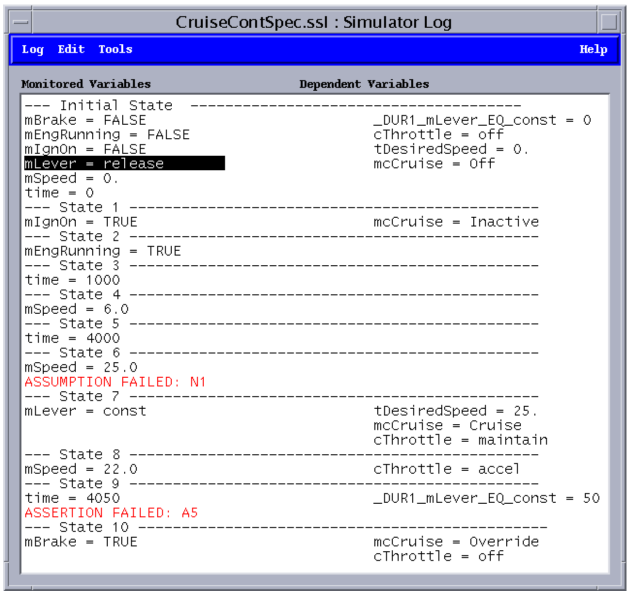
smaller SCR specification satisfies the SCR formal model and can therefore be analyzed separately from the larger specification using the SCR tools. This capability to extract a smaller SCR specification and analyze it separately is especially useful for understanding and analyzing the large specifications associated with most practical systems, for example, the weapons control system described in [25]. Such specifications are so large that sometimes their dependency graphs cannot be displayed in full on the user's display. Thus, it is often useful to decompose them into smaller, more manageable specifications using the DGB.

3.4 Validating the specification

3.4.1 The Simulator

To initiate simulation, the user loads a scenario – a sequence of monitored variable values which implicitly define a sequence of monitored events – into the Simulator. To execute the scenario, the user either uses the Step button to move through the scenario one step at a time or presses the Run button to execute the entire sequence with a single command (see the Simulator Display in Figure 12). To compute each new state from a monitored event and the current state, the Simulator applies the transform function *T* of our requirements model. As each new state is computed, the Simulator Display is updated to reflect the new state. During simulation, the user may display a Log (see Figure 13) of the state history. The Log displays the initial state in full. For each subsequent state, it lists the monitored event that caused the transition along with each dependent variable whose value has changed. During simulation, the Simulator may, at user request, check both the assumptions and the assertions. When an assumption or assertion is violated, the Simulator notifies the user of the violation.

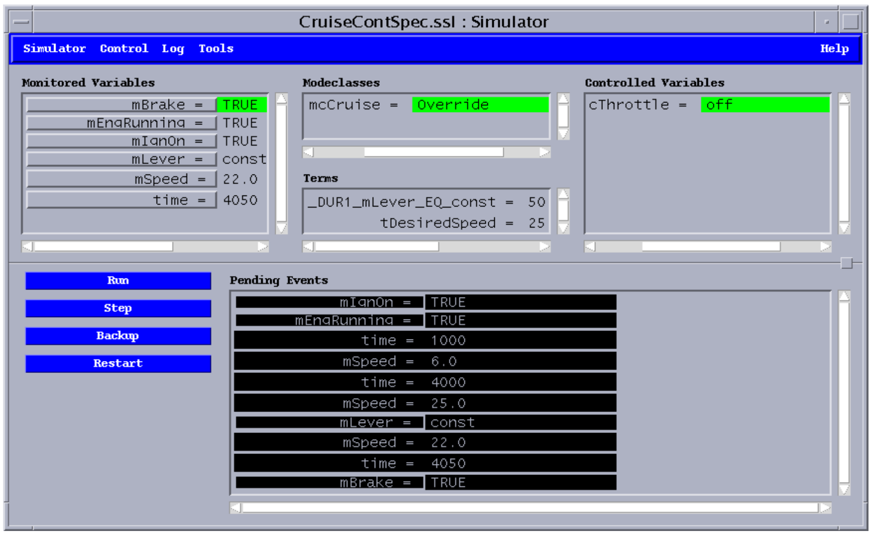
To illustrate simulation, we loaded a ten-step scenario into the Simulator (see the scenario under Pending Events in Figure 12). As one steps through this scenario, new values of the variables appear in the top half of the Simulator Display. For example, the Simulator Display in Figure 12 shows and highlights the values of the three variables that change at step 10, namely, `mBrake`, `mcCruise` and `cThrottle`.



```
--- Initial State  ----------------------------------------
mBrake = FALSE                                 _DUR1_mLever_EQ_const = 0
mEngRunning = FALSE                            cThrottle = off
mIgnOn = FALSE                                 tDesiredSpeed = 0.
mLever = release                               mcCruise = Off
mSpeed = 0.
time = 0
--- State 1 ----------------------------------------------
mIgnOn = TRUE                                  mcCruise = Inactive
--- State 2 ----------------------------------------------
mEngRunning = TRUE
--- State 3 ----------------------------------------------
time = 1000
--- State 4 ----------------------------------------------
mSpeed = 6.0
--- State 5 ----------------------------------------------
time = 4000
--- State 6 ----------------------------------------------
mSpeed = 25.0
ASSUMPTION FAILED: N1
--- State 7 ----------------------------------------------
mLever = const                                 tDesiredSpeed = 25.
                                               mcCruise = Cruise
                                               cThrottle = maintain
--- State 8 ----------------------------------------------
mSpeed = 22.0                                  cThrottle = accel
--- State 9 ----------------------------------------------
time = 4050                                    _DUR1_mLever_EQ_const = 50
ASSERTION FAILED: A5
--- State 10 ----------------------------------------------
mBrake = TRUE                                  mcCruise = Override
                                               cThrottle = off
```

Figure 13 Simulator Log showing violations of an assumption and an assertion

In addition to describing the initial state and the history of state changes, the Log in Figure 13 reports two problems: 1) at step 6, assumption N1 is violated (because the automobile speed increased by more than `kMaxAccel` = 6), and 2) at step 9, assertion A5 is violated. Double clicking on the name of the violated assumption or assertion displays the appropriate dictionary with the violated property highlighted. For example, clicking on "`ASSERTION FAILED: A5`" (see Figure 13) displays the Assertion Dictionary shown in Figure 8 with the entry for A5 highlighted. Double clicking on "`cThrottle = off`", a dependent variable that changed during step 10 (see Figure 13), displays Table 5 with the rule that caused `cThrottle` to change highlighted. This information is useful in helping the user understand and correct the error. Changing this scenario so that it does not violate assumption N1 (e.g. change Step 6 to "`mSpeed = 12`") and running the modified scenario through the Simulator also violates A5, thus demonstrating that A5 is not an invariant of the CCS specification.

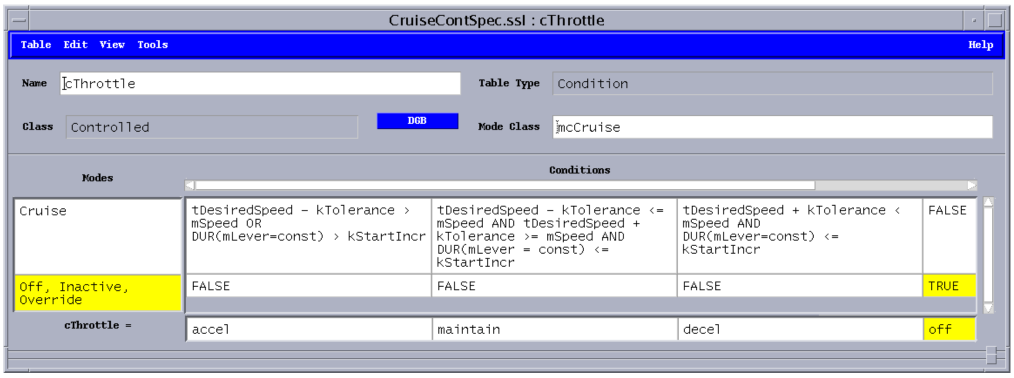


```
Monitored Variables          Modeclasses                          Controlled Variables
      mBrake =  TRUE         mcCruise =  Override                 cThrottle =  off
 mEngRunning =  TRUE
      mIgnOn =  TRUE         Terms
      mLever =  const        _DUR1_mLever_EQ_const =  50
      mSpeed =  22.0                 tDesiredSpeed =  25
        time =  4050

Pending Events
      mIgnOn =  TRUE
 mEngRunning =  TRUE
        time =  1000
      mSpeed =  6.0
        time =  4000
      mSpeed =  25.0
      mLever =  const
      mSpeed =  22.0
        time =  4050
      mBrake =  TRUE
```

Figure 12 Simulator Display showing a scenario containing ten events

Table 5 Table defining cThrottle with rule highlighted


CruiseContSpec.ssl : cThrottle

Name	cThrottle	**Table Type**	Condition	
Class	Controlled	DGB	**Mode Class**	mcCruise

Modes	**Conditions**			
Cruise	tDesiredSpeed - kTolerance > mSpeed OR DUR(mLever=const) > kStartIncr	tDesiredSpeed - kTolerance <= mSpeed AND tDesiredSpeed + 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 6 Automatically generated state invariants for the CCS

	Description	Generated Invariant
I1	In Off mode, the ignition is off	mcCruise = Off ⇒ NOT mIgnOn
I2	In Inactive mode, the ignition is on	mcCruise = Inactive ⇒ mIgnOn
I3	In Cruise mode, the ignition is on, the engine is running, the brake is off, and the lever if not off	mcCruise = Cruise ⇒ mIgnOn AND mEngRunning AND NOT mBrake AND mLever != off
I4	In Override mode, the ignition is on and the engine is running	mcCruise = Override ⇒ mIgnOn AND mEngRunning
I5	The throttle is off iff the system is not in Cruise mode	cThrottle = off ⇔ mcCruise != Cruise
I6	The throttle is accel iff the system is in Cruise mode and either desired speed minus tolerance exceeds actual speed or the lever has been in const more than kStartIncr	cThrottle = accel ⇔ mcCruise = Cruise AND (tDesiredSpeed - kTolerance > mSpeed OR DUR(mLever = const) > kStartIncr)
I7	If the lever has been in const for some time then the lever is const	DUR(mLever = const) > 0 ⇒ mLever = const

3.4.2 The Invariant Generator

The SCR invariant generator is based on two algorithms that construct state invariants from the tables defining the dependent variables [37, 38]. Suppose that dependent variable r has values in a finite set $\{v_1, v_2, ..., v_n\}$. If r is defined by a mode transition table or event table, then for each v_i, the algorithm generates an invariant of the form

$$r = v_i \Rightarrow C_i$$

where C_i is a predicate over variables upon which r depends.

Invariant generation from a mode transition or event table is based on the following intuitive result from [3]: 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 (or $r = v_i$ initially), and 2) the event @F(C_i) unconditionally causes r to have a value other than v_i. Since stronger invariants may be computed using knowledge of previously computed invariants, the full algorithms repeat the computations of the set of invariants until a fixpoint is reached.

In SCR, a condition table for a variable r defines a total function if it satisfies the Disjointness and Coverage Properties. This one-state function can be represented syntactically as an expression E over the variables on which r depends. This expression captures the content of the condition table. If this table passes the Consistent Initial State check, i.e. r and E have the same value in any initial state (see Figure 9), then $r = E$ is a state invariant. From this general result for r defined by a condition table, we may generate a state invariant for each value v_i of r in the form

$$r = v_i \Leftrightarrow C_i$$

where each C_i is the simplification of the expression $E = v_i$. In many cases, such as the condition table defining `cThrottle` in Table 3, the expressions C_i in such invariants are easily determined directly by inspection of the table.

The implementation of the SCR invariant generator has been extended to apply the algorithms described in [37, 38] to all formulae containing Boolean variables and enumerated type variables that have been derived from the tables that comprise an SCR requirements specification. Table 6 lists seven invariants, I1-I7. The tool generated I1-I4 automatically from the mode transition table defining `mcCruise` and invariants I5 and I6 from the condition table defining `cThrottle` (along with similar invariants for `cThrottle` with the values `maintain` or `decel`). Manual application of the algorithm generated invariant I7 from the event table defining `_DUR_mLever_EQ_const`, a table constructed automatically by the toolset. Our tool could not generate this invariant automatically because the table contains a numerical expression.

3.5 Analyzing application properties

3.5.1 The Model Checker

A model checker, such as SPIN [35], can analyze a finite state specification for application properties, e.g. the properties listed in Figure 8. To use model checking either for verification or to detect property violations, the *state explosion problem* must be addressed, i.e. the size of the state space to be analyzed must be reduced.

Most users of model checkers perform such reductions routinely, but usually in ad hoc ways: the correspondence between the "abstract model" and the original specification is based on informal, intuitive arguments. To make model checking more accessible to software developers and to reduce the need for ad hoc abstraction, we have developed methods for automatically constructing sound and sometimes complete abstract models from specifications based on the property to be analyzed [7, 25]. In a *sound* model, each property that holds in the model corresponds to a property that holds in the original specification. In a *complete* (or *refutation-sound*) model, each property violation in the model corresponds to a property violation in the original specification. One abstraction method called *slicing*, which produces sound and complete abstract models, removes all irrelevant variables from the specification. Another abstraction method, called *variable abstraction*, replaces a variable having a large (often infinite) type set with another variable having a smaller, discrete type set. Done with appropriate attention to the values to which the variable is compared in the specification and the property to be verified, variable abstraction produces sound, and sometimes complete, abstract models [25].

Our approach to model checking consists of three steps. First, based on the property, an abstract model is constructed. Next, the abstract model is automatically translated from SCR to the language of the model checker, and the model checker is invoked to analyze the property. Finally, if the model checker detects a property violation, the counterexample produced by the model checker is translated into a corresponding counterexample in the original specification.* This last step is crucial because the user's understanding of the system will be in terms of the original specification rather than the abstract model.

Although addressing the state explosion problem required significant effort, model checking was effective in verifying seven of the ten CCS properties, in partially verifying an eighth, and in detecting violations of the remaining two properties. Because three numeric variables, `mSpeed`, `tDesiredSpeed` and `time`, have infinitely many values, reducing the state space of the CCS specification was essential. To model check A3 and A7, we used slicing to eliminate all irrelevant variables from the specification. For example, to automatically construct the abstract model for A3 using slicing, we invoked the DGB, displayed the Assertion List, and selected A3; after the DGB identified the subgraph of variables needed to analyze A3, we saved the CCS specification associated with this subgraph to a file (see Figure 14). The resulting abstract model, which is sound relative to A3, contains five variables, `mcCruise` and the four monitored variables on which `mcCruise` depends. It has a much smaller state space than the original specification since it omits the three numeric variables `mSpeed`, `tDesiredSpeed` and `time`. Next, the toolset translated the abstract model from SCR into Promela, the language of SPIN, and executed SPIN on the Promela code. A similar procedure was used to model check A7. SPIN verified that the abstract model, and hence the CCS specification, satisfies both A3 and A7.

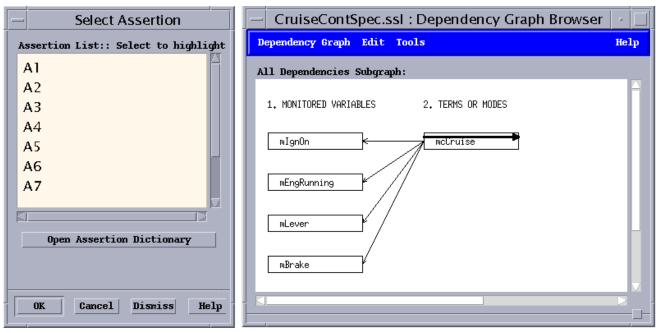


Figure 14 Constructing an abstraction using slicing

*If the abstract model is not complete, the counterexample constructed by the model checker may not correspond to a valid counterexample in the original specification.

Because the six assertions A1, A2, A4, A5, A6, and A8 all refer to `cThrottle`, which depends directly or indirectly on all other variables, slicing cannot remove any irrelevant variables. To reduce the state space of the specification prior to analyzing these six properties, we constructed an abstract model describing a proper subset of CCS behaviors that is sound for refutation, though not for verification. To do so, we changed the type `ySpeed` from real to integer and the units of time from milliseconds to tenths of seconds and reduced the maximum automobile speed from 180 to 10 mph. Running SPIN on this abstract model generated counterexamples for properties A2 and A5, which were validated using the Simulator

Because model checking the abstract model used to refute properties A2 and A5 did not detect violations of properties A1, A4, A6, and A8, we tried next to verify these properties. We succeeded in doing so by constructing an abstract model, in an ad hoc fashion, that is sound for verification. The approach used to construct the abstract model is a variant of predicate abstraction. The model eliminates the real valued variables `tDesiredSpeed`, `mSpeed` and `time`, and introduces two new variables, `tSpeedDelta` and `tLeverConst`, to capture the information in the expressions `tDesiredSpeed - mSpeed` and `DUR(mLever = const) > kStartIncr`. Paying attention to the important threshold values of `tDesiredSpeed - mSpeed` allows `tSpeedDelta` to be assigned an enumerated type with five values rather than type real. To achieve soundness, a third new "driver" variable is introduced to ensure that all simultaneous changes in value of the new variables evaluated in the context of the original specification are covered in the abstract model.

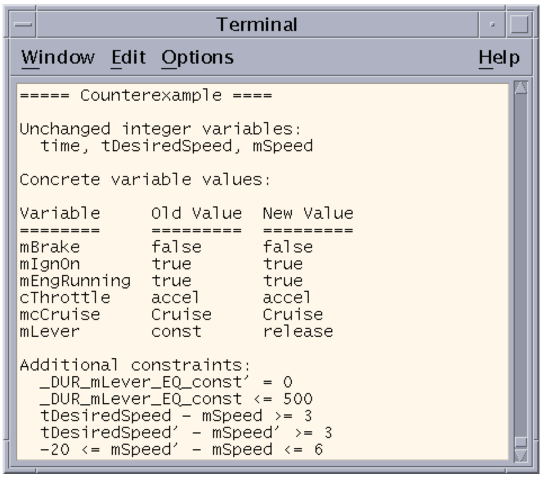
Because properties A9 and A10 both refer to `tDesiredSpeed`, which depends directly or indirectly on all the other variables except `cThrottle`, slicing on these properties eliminates only `cThrottle`. Since `cThrottle` has only four possible values, the reduction in size of the state space is small. Again, ad hoc abstraction is needed to verify these properties using SPIN. (These two properties have been verified using both the verifier and the property checker; see Sections 3.5.2 and 3.5.3.) Because property A10 can be reformulated in terms of `tSpeedDelta`, it is possible to verify it with SPIN using the same abstraction used to verify A1, A4, A6, and A8. However, verifying property A9 with SPIN is not as simple. First, because the validity of A9 depends on the environmental assumptions, which are not included in the Promela translation of the CCS specification, verifying A9 would require the assumptions to be included as a hypothesis on A9. Second, state explosion needs to be addressed. We discovered that eliminating the variables `cThrottle` and `time` and replacing `DUR(mLever = const) > kStartIncr` with the new variable `tLeverConst` allowed SPIN to terminate, provided the constant `kMaxSpeed` was reduced sufficiently. However, formally establishing that this abstract model is sufficient to verify property A9 would require us to prove a theorem to that effect. Since theorem proving is able to establish the property directly, we did not put forth the extra effort required to verify A9 using SPIN.

3.5.2 The Verifier

The verifier TAME, an interface to the theorem prover PVS, simplifies the use of PVS to specify and prove properties of addressing the environmental assumption and state explosion problems as described automata models by supplying templates for specifying automata (i.e. state machine) models and more than 20 specialized strategies that provide high-level proof steps for proving invariant properties. Recently, support for proving refinement properties has been added. Initially developed for timed automata [46], TAME has also been adapted to the state machine model that underlies SCR [1]. TAME's strategies [2] are supported by a standard set of theories together with both auxiliary theories and auxiliary strategies associated with a particular template instantiation. TAME is integrated into the SCR toolset by an automatic SCR-to-TAME translator which uses information from an SCR specification to fill in TAME's SCR automaton template and which generates the associated auxiliary theories and strategies.

TAME provides two strategies, **SCR_INV_PROOF** and **ANALYZE**, especially designed for SCR. The strategy **SCR_INV_PROOF**, which proves many invariants automatically, determines whether a property is a one-state property or a two-state property, and attempts accordingly either an induction proof or a direct proof using the transition relation. When the proof does not complete, **SCR_INV_PROOF** explores whether any combination of the automatically generated state invariants (see Section 3.4.2) can complete the proof. This is done by applying, in each unproved subgoal, one or more of the generated invariants, and backtracking if the subgoal is not proved. For each proved subgoal, TAME determines exactly which auxiliary invariants are needed. Through an optional numerical argument to **SCR_INV_PROOF**, the user may limit the size of the combinations of generated invariants to be tried.

When a proof fails, **SCR_INV_PROOF** generates as unproved subgoals one or more *dead ends*. From these dead ends, **ANALYZE** helps the user determine whether the invariant is false or whether additional invariants can be used to complete the proof. Associated with each dead end is a set of *problem transitions* that either do not preserve the invariant (for state invariants) or violate the invariant (for transition invariants). **ANALYZE** causes PVS to display details of the problem transitions, including the monitored event, known values of variables in the prestate or poststate, and variables whose values are unchanged. A prototype translator has been built which represents this information as a predicate on state pairs. TAME's feedback from failed proofs is more extensive than the feedback from the property checker (see Section 3.5.3). That is, every possible



```
===== Counterexample ====

Unchanged integer variables:
  time, tDesiredSpeed, mSpeed

Concrete variable values:

Variable     Old Value  New Value
========     =========  =========
mBrake       false      false
mIgnOn       true       true
mEngRunning  true       true
cThrottle    accel      accel
mcCruise     Cruise     Cruise
mLever       const      release

Additional constraints:
  _DUR_mLever_EQ_const' = 0
  _DUR_mLever_EQ_const <= 500
  tDesiredSpeed - mSpeed >= 3
  tDesiredSpeed' - mSpeed' >= 3
  -20 <= mSpeed' - mSpeed <= 6
```

Figure 15 State pair predicate returned by Salsa for A5

problem transition is represented among the proof dead ends. While an individual problem transition may help determine whether *some* state history leads to a violation, the full set of problem transitions contains all transitions corresponding to violations and may also suggest how to modify an assertion to make it true. In fact, the assertion A10 was arrived at based on the information in the full set of counterexamples to the candidate property `tDesiredSpeed=mSpeed => cThrottle=off`.

While, in SCR specifications, abstractions such as slicing can reduce both the number of induction cases and the complexity of reasoning about state transitions, the CCS specification was small enough that TAME could be used without abstraction. The strategy **SCR_INV_PROOF** automatically verified the eight true assertions, namely, A1, A3, A4, and A6-A10 (see Figure 8). It proved A7-A10 directly without invariants. Completing the proofs of the remaining true properties – A1, A3, A4, and A6 – required the use of generated invariants (see Table 6) as auxiliary lemmas. Although almost all branches in these proofs required only a single invariant, namely, I3, completing the proof of property A3 required three invariants, I2, I3, and I4, thus demonstrating that combinations of the invariants can be useful. As shown above, the remaining two assertions, A2 and A5, are false. For A5, TAME produces 23 dead ends. The first of these subsumes the state pair detected by Salsa (see Figure 15 in Section 3.5.3). For A2, TAME produces two dead ends which cannot be proved using the generated invariants.

3.5.3 The Property Checker

To analyze application properties such as those listed in Figure 8, Salsa carries out an induction proof, treating all of the automatically generated invariants (see Table 6) as axioms. Salsa also applies slicing to remove all variables irrelevant to a property's validity from the specification. When a proof fails, Salsa returns a state pair. Because the transition represented by the state pair may be unreachable, the user must show either that the property is false by finding a scenario which violates the property or that the property is true by applying additional invariants to complete the proof.

Salsa was applied to all of the assertions listed in Figure 8. Applying Salsa was more automated than applying SPIN since Salsa required no manual abstraction. Like TAME, Salsa was able to verify eight assertions – A1, A3, A4, and A6-A10. In each case, its proof times were faster than those of both SPIN and TAME. Salsa produced a single state pair for each of the false properties, A2 and A5. The state pair returned by Salsa for A5 is shown in Figure 15. This state pair can be used in conjunction with the Simulator to derive a valid counterexample for A5.

The utility of automatically generated invariants was demonstrated both in verifying properties with Salsa and in producing a state pair that can be associated with valid counterexamples. As shown in Section 3.5.2, invariants are needed to verify four properties – A1, A3, A4, and A6. By including all of the automatically generated invariants in its analysis, Salsa established the validity of all of these properties. However, in the initial analysis of property A2 with Salsa, invariant I7 (see Table 6) was omitted. The state pair predicate returned by Salsa as one of the possible problem transitions for A2 required that, in the old state, the lever is not in `const`, but the length of time that the lever has been in `const` exceeds `kStartIncr` – a contradiction. Thus a counterexample that ends in this state pair is infeasible. Including invariant I7, which states that `DUR(mLever=const) > 0` ⇒ `mLever=const`, eliminates this problem transition and produces a state pair that can be used in conjunction with the Simulator to derive a valid counterexample for A2.

4. APPLYING THE TOOLS TO PRACTICAL SYSTEMS

Currently, more than 200 academic, industrial, and government organizations in the U.S., Canada, and a number of other countries are experimenting with the SCR tools. The utility of the SCR tools has also been evaluated in four case studies involving real-world systems. In one study, NASA researchers used the CC to detect several missing assumptions and instances of ambiguity in the requirements specification of the International Space Station [11]. In a second study, engineers at Rockwell Aviation used the SCR tools to detect 28 errors, many of them serious, in the requirements specification of a flight guidance system [49]. In the Rockwell effort, one-third of these errors were found by entering the tabular requirements specification into the SCR toolset, another third by applying the CC, and the remaining third by running scenarios through the Simulator. These results suggest that different tools find different classes of errors.

In a third study, our group at NRL used the SCR tools to expose several errors, including a safety violation, in a moderately large contractor-produced specification of a U.S. weapons control system [25]. This contractor specification, which contains over 250 variables, and six safety properties, was translated semiautomatically into the SCR tabular notation. By applying slicing and variable abstraction to the SCR specification and then invoking SPIN on an abstract model that was sound but not complete, a possible safety violation was detected. Translating the counterexample returned by SPIN, which was in terms of the abstract model, into a corresponding counterexample in the original specification and running the counterexample through the Simulator validated that the detected violation was an actual violation.

In a fourth study, our group used the SCR tools to specify the requirements of a cryptographic device (CD), to verify that the CD specification satisfies seven security properties, and to demonstrate that the specification violates an eighth property [39]. Although model checking with SPIN was useful in performing sanity checks, it was not useful in either verifying or refuting these security properties. Instead, both TAME and Salsa, in combination with several invariants generated by the invariant generation algorithm, verified the seven safety properties. They also refuted a property (although experimentation with the Simulator was needed to validate that a counterexample satisfying the state pair predicates returned by Salsa and TAME was feasible). To be useful in practice, the benefits of using a method should be sufficient to warrant the cost in human effort of applying the method. In the latter two projects, the potential cost-effectiveness of the SCR tools was demonstrated: in each case, specifying and analyzing a moderately complex system required less than five person-weeks of effort.

The SCR tools are also being used in software practice. The largest industrial users of the SCR technology are three sites of Lockheed Martin where the tools have been used since 1999 to construct and analyze requirements for flight control and flight guidance systems, fire control systems, and many other avionics applications. In addition, our group at NRL is using the tools to construct a formal specification of, and to verify, the security kernel of a second member of the CD family of cryptographic devices. The formal specification and proofs of the kernel properties and the results of our tool-based analysis will comprise a significant component of the evidence that the National Security Agency considers in evaluating CD II for certification. Finally, as part of their program in Software Engineering Research Infusion, NASA recently identified the SCR technology as one of a group of technologies [50] that are “sufficiently mature and promising that they can be adopted by NASA development teams.” Recently, NRL conducted a course to introduce NASA software analysts to the SCR method and tools and to demonstrate the use of SCR to check the current requirements document for a safety-critical software component of the International Space Station.

5. MORE ABOUT THE SCR TOOLS

5.1 A graphical front-end for the simulator

With a Graphical User Interface Builder, a graphical front-end can be quickly built and attached to the Simulator, thereby producing a rapid prototype for demonstrating and validating the required behavior of the system under development [24, 29]. A specifier or a developer can run scenarios through this prototype and thus demonstrate the behavior represented by the SCR requirements specification. More important, without understanding the SCR tables or the generic interface to the Simulator, a customer or a future system user can execute the prototype to ensure that the SCR specification captures the intended behavior. Raising questions about the specified behavior, identifying missing or misspecified behavior, and changing the specification to remove any detected errors can lead to significant improvements in the specification.

5.2 Constructing and testing a system implementation

While specifying, verifying, and validating the system requirements is important, especially in the development of safety-critical systems, ultimately a system must be constructed that satisfies the requirements. A specification that has been verified and validated using the SCR tools provides a solid basis both for developing executable code and for generating "test cases" useful in evaluating a system implementation. Although automatically generating code may be infeasible for some purposes (e.g. code that implements complex algorithms or provides an interface to a physical device), such an approach is feasible for code that implements a program's control logic. In the latter cases, the code can be automatically generated from an SCR requirements specification. Recently, we have developed a grammar and a set of semantic rules to describe SCR specifications and used the APTS transformational system [51] to automatically generate C source code from an SCR specification for CD [39]. See [41] for details.

To convince customers that the implementation is acceptable and to detect errors, the system implementation must be tested. An enormous problem, however, is that software testing is extremely costly and time-consuming. It is estimated that current testing methods consume between 40% and 70% of the software development effort [4]. We have developed an automated technique [15] and built a prototype tool in Java that constructs a suite of test cases from an SCR requirements specification. (A *test case* is a sequence of system inputs in which each input is coupled with the required system outputs.) To ensure that the test cases "cover" the set of all possible system behaviors, our technique organizes all possible system executions into equivalence classes and builds one or more test cases for each class. These test cases can be used to automatically evaluate the implementation. By reducing the human effort needed to build and to run the test cases, this approach can reduce both the enormous cost and the significant time and human effort associated with current software testing methods.

5.3 Model checking versus theorem proving

Section 3.5.1 describes the use of model checking both to verify and to refute properties. Due to the state explosion problem, model checking is most often used not for verification but to detect errors. In some cases, the use of model checking to detect property violations may not succeed. For example, in analyzing the CD specification, theorem proving (rather than model checking) detected an invalid property – the model checker ran out of memory prior to detecting any violations [39].

To combat state explosion, users may construct an abstract model of the specification, but as Section 3.5.1 suggests, constructing an "acceptable" model often requires significant effort. To detect property violations, an abstract model that is complete is desirable. To verify a property, an abstract model that is sound is desirable. The penalty for using an abstract model which fails to be sound for refutation is not too serious – if one finds that a violation in the abstract model does not correspond to a violation in the specification (for example, using simulation), one can either try to fix the abstract model so that it is complete or find other violations in the abstract model and check whether any of these correspond to violations in the original specification.

In contrast, verification based on an unsound model is dangerous, since a property that holds in an abstract model may not correspond to a property that holds in the original specification. To obtain sound models, we need techniques that automatically construct sound abstractions (see, e.g., [17, 7, 25]) or that demonstrate (automatically, if possible) the soundness of an abstract model relative to a selected property.

An advantage of theorem proving (e.g. using either TAME or Salsa) over model checking is that it avoids the state explosion problem, using only the facts needed to reason about a property at a high level of abstraction. In theorem proving, the construction of an abstract model is usually unnecessary. One serious problem with theorem proving occurs, however, when a proof fails – the user does not know whether the dead end occurs because auxiliary lemmas are needed to complete the proof or because the property is false. Techniques are needed which use the information returned by TAME and Salsa to determine whether a counterexample is feasible and, if so, to construct such a counterexample.

5.4 Advantages of a tool suite

As Section 3 shows, the analyses performed by some tools can be used in later analyses by other tools. For example, the TAME analysis uses variable dependency information and assumes that the specification does not contain circular definitions or violations of the Disjointness Property – each of which is guaranteed by the Consistency Checker. Further, both TAME and Salsa use the invariants generated by the Invariant Generator in verification and in identifying candidate counterexamples. Moreover, the Simulator is an important adjunct to the other analyses: it can be used both to demonstrate property violations to users and to validate candidate counterexamples. Finally, several tools may be used to perform sanity checks; for example, verifying the eight true properties in Figure 8 using TAME, Salsa, and SPIN increases confidence that these properties do indeed hold.

6. RELATED WORK

Our research on tabular notations and tools for analyzing tables complements research on tabular expressions at McMaster University. While the McMaster group has developed a general theory of tables [36] and tools for manipulating a general form of tables [56], we focus on a few simple table types useful in specifying system requirements. In our research, tables and their semantics are just one part of our overall approach to specifying and analyzing state-based requirements.

Like SCR, the Requirements State Machine Language (RSML) [43] is designed to specify system requirements. Like the SCR model, the RSML model describes deterministic, state-based behavior. RSML's hierarchical graphical notation is borrowed from Statecharts [20]. To make RSML specifications more readable, the guards on state transitions are presented in special tables, called AND-OR tables. In RSML, whose step semantics is similar to that of Statecharts, a step is triggered by an external event and may include a potentially infinite number of *microsteps*. Several tools have been applied to RSML specifications: a BDD-based tool which checks AND-OR tables for nondeterminism and missing cases [22], the Stanford Validity Checker which also checks AND-OR tables for nondeterminism [52], and symbolic model checking, which was used to check various properties, including safety properties [10]. Due to its simpler step semantics, consistency checking in SCR is significantly easier than consistency checking in RSML.

To simplify the graphical representation of hierarchical state machines and to eliminate the RSML step semantics, which was difficult to understand and error-prone [42], two new languages based on RSML, SpecTRM [44] and RSML^{-e} [60, 21], have been developed. Both languages have a step semantics similar to SCR's. A toolset NIMBUS, which supports the prototyping of systems in RSML^{-e}, includes capabilities for consistency checking, simulation, model checking, theorem proving, test case generation, and code generation [21].

In the Input/Output Automaton (IOA) model [45], a formal model for representing asynchronous concurrent systems, each automaton is defined by a set of states and a set of parameterized actions. At each step, the automaton chooses some action (enabled according to its "precondition") and performs updates to variables based on the "effect" of that action. Recently, a complete textual language and toolset have been developed to support the IOA model [16]. Currently, the toolset supports parsing and semantic analysis of IOA models, simulation, theorem proving, and code generation. The tool TAME, described in Section 3.5.2, provides automated support for specification and verification of both the IOA model and the timed automata model.

Originally designed as a reactive programming language for critical "kernel" software, LUSTRE [19, 18] is a synchronous data flow language which has also been used as a hardware description language. In basic LUSTRE, programs are represented in a textual language. In contrast, SCADE, a commercial toolset for LUSTRE, uses a notation based on operator network diagrams, a generalization of circuit diagrams. LUSTRE programs in textual form are a set of equations, each of which defines a new value for some variable via some function of other variables. Such equations are essentially a generalization of the definition of SCR dependent variables via table functions. The prototype tools developed for LUSTRE perform theorem proving, model checking, code generation, simulation, and test case generation. The toolset SCADE provides a graphical editor for creating operator network diagrams, simulation, and code generation.

7. CONCLUSIONS

This paper has described how tables may be used to organize and to represent a state-based requirements specification, the assumptions that constrain the specification, and the properties that the specification is expected to satisfy. It has also described how tools may be used to create and display these tables and to show how they are related, to find errors in tables and provide feedback useful for correcting the errors, and to validate and verify properties of tabular specifica-tions.

Our current research is building on both the SCR formal model and the SCR method and tools. Among the research topics we are investigating are the following:

- the design of tabular representations and tool support of more complex data structures (e.g. arrays and records),
- an XML-based intermediate representation for SCR that will provide a standard interface for integrating tools (both front-ends and back-ends) into the toolset, and
- extension of the invariant generation algorithms to handle logical expressions containing numbers and numerical operations.

A more long-term goal is to investigate two important but very difficult problems in requirements. The first is to develop libraries, templates, guidebooks, and other technology that will help practitioners construct a precise, unambiguous requirements specification. In teaching courses about the SCR method, we have found that, for practitioners, constructing a well-formed, precise, unambiguous requirements specification is a significant challenge. A second goal is to develop technology that helps practitioners understand the required system behavior. Needed is support for scenarios, libraries, techniques that synthesize requirements specifications from scenarios, and other techniques that allow a user to understand system and software requirements and translate them into a rigorous requirements specification.

A well-designed suite of tools can allow software practitioners without advanced mathematical training and theorem proving skills to perform relatively complex analyses of tabular specifications. In our view, SCR offers such a suite of tools and thus provides the basis for a practical method for software developers to specify and analyze tabular representations of system requirements.

ACKNOWLEDGMENTS

This research was supported by the Office of Naval Research. The research on code generation described in Section 5 is that of our colleague Beth Leonard. We also gratefully acknowledge Bruce Labaw's very significant contributions to the design and implementation of the SCR tools and Beth's useful comments on this paper. Jim Kirby developed the original specification of the CCS in a tabular format.

REFERENCES

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

2 **Myla Archer.** TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence,* 29(1-4):131–189, February 2001.

3 **Joanne M. Atlee.** Automated Analysis of Software Requirements. PhD thesis, Dept. of Computer Science, Univ. of Maryland, College Park, MD, 1992.

4 **B. Beizer.** *Software Testing Techniques*. Van Nostrand Reinhold, 1983.

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

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

7 **Ramesh Bharadwaj and Constance Heitmeyer.** Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1), January 1999.

8 **A. Boudet and H. Comon.** Diophantine equations, Presburger arithmetic and finite automata. In: *Trees and Algebra in Programming–CAAP*, LNCS 1059, 1996.

9 **R. E. Bryant.** Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

10 **William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese.** Model checking large software specifications. *IEEE Trans. Softw. Eng.*, 24(7), July 1998.

11 **Steve Easterbrook and John Callahan.** Formal methods for verification and validation of partial specifications: A case study. *J. Syst. Soft.*, 1997.

12 **S. R. Faulk, J. Brackett, P. Ward, and J. Kirby, Jr.** The CoRE method for real-time requirements. *IEEE Software*, 9(5):22–33, September 1992.

13 **S. R. Faulk, L. Finneran, J. Kirby, Jr., S. Shah, and J. Sutton.** Experience applying the CoRE method to the Lockheed C-130J. *Proc. 9th Annual Conf. on Computer Assurance (COMPASS '94),* Gaithersburg, MD, June 1994.

14 **Stuart R. Faulk.** State Determination in Hard-Embedded Systems. PhD thesis, Univ. of NC, Chapel Hill, NC, 1989.

15 **A. Gargantini and C. Heitmeyer.** Using model checking to generate tests from requirements specifications. Proc. ACM *Softw. Eng. Conf. and 7th ACM SIGSOFT Symp. Foundations of Softw. Eng. (ESEC/FSE99)*, LNCS 1687:146-162, 1999.

16 **Stephen J. Garland and Nancy Lynch.** Using I/O automata for developing distributed systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge Univ. Press, 2000.

17 **S. Graf.** Characterization of a sequentially consistent memory and verification of a cache memory by abstraction. *Proc. Computer Aided Verification (CAV'94)*, 1994.

18 **Nicolas Halbwachs.** *Synchronous Programming of Reactive Systems*. Kluwer Academic, Boston, MA, 1993.

19 **Nicolas Halbwachs, Fabienne Lagnier, and Christophe Ratel.** Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. Softw. Eng.*, 18(9):785–793, September 1992.

20 **David Harel.** Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

21 **M. P. E. Heimdahl, M. W. Whalen, and J. M. Thompson.** Nimbus: A tool for specification centered development. *Proc. 11th IEEE Int. Req. Eng. Conf. (RE'03),* September 2003.

22 **Mats P. E. Heimdahl and Nancy Leveson.** Completeness and consistency in hierarchical state-based requirements. *IEEE Trans. Softw. Eng.*, 22(6):363–377, June 1996.

23 **C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw.** SCR*: A toolset for specifying and analyzing requirements. *Proc. 10th Annual Conf. on Computer Assurance (COMPASS'95),* pages 109–122, Gaithersburg, MD, June 1995.

24 **C. Heitmeyer, J. Kirby, Jr., and B. Labaw.** The SCR method for formally specifying, verifying, and validating requirements: Tool support. *Proc. 19th Int. Conf. Softw. Eng. (ICSE'97),* Boston, MA, 1997.

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

26 **C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw.** Automated consistency checking of requirements specifications. *ACM Trans. Softw. Eng. Meth.*, 5(3):231–261, April–June 1996.

27 **Constance Heitmeyer.** Developing high assurance systems: On the role of software tools. *Proc. 22nd Int. Conf. on Computer Safety, Reliability and Security (SAFECOMP 2003)*, Edinburgh, Scotland, September 2003 (invited).

28 **Constance Heitmeyer, James Kirby, Jr., and Bruce Labaw.** Tools for formal specification, verification, and validation of requirements. *Proc. 12th Annual Conf. on Computer Assurance (COMPASS'97),* Gaithersburg, MD, June 1997.

29 **Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, and Ramesh Bharadwaj.** SCR: A toolset for specifying and analyzing software requirements. *Lecture Notes in Computer Science* 1427:526-531, 1998.

30 **Constance L. Heitmeyer and Ralph D. Jeffords.** The SCR tabular notation: A formal foundation. Technical report, Naval Research Lab., Wash., DC, 2005. To appear.

31 **Constance L. Heitmeyer and John McLean.** Abstract requirements specifications: A new approach and its application. *IEEE Transactions on Softw. Eng.*, SE-9(5):580–589, September 1983.

32 **Kathryn Heninger, David L. Parnas, John E. Shore, and John W. Kallander.** Software requirements for the A-7E aircraft. Technical Report 3876, Naval Research Lab., Wash., DC, 1978.

33 **Kathryn L. Heninger.** Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Eng.,* SE-6(1):2–13, January 1980.

34 **S. D. Hester, D. L. Parnas, and D. F. Utter.** Using documentation as a software design medium. *Bell System Tech. J.*, 60(8):1941–1977, October 1981.

35 **G. J. Holzmann.** The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997.

36 **R. Janicki.** Towards a formal semantics of Parnas tables. *Proc. 17th Int. Conf. Softw. Eng. (ICSE '95),* pages 231–240, Seattle, WA, April 1995. ACM.

37 **Ralph Jeffords and Constance Heitmeyer.** Automatic generation of state invariants from requirements specifications. *Proc. 6th ACM SIGSOFT Symp. Foundations Softw. Eng.,* November 1998.

38 **Ralph Jeffords and Constance Heitmeyer.** An algorithm for strengthening state invariants generated from requirements specifications. *Proc. 5th IEEE Int. Symp. Req. Eng. (RE'01),* Toronto, Canada, August 2001.

39 **J. Kirby, Jr., M. Archer, and C. Heitmeyer.** SCR: A practical approach to building a high assurance COMSEC system. *Proc. 15th Annual Computer Security Applications Conf. (ACSAC '99).* IEEE Computer Society Press, December 1999.

40 **James 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.

41 **E. I. Leonard and C. L. Heitmeyer.** Program synthesis from formal requirements specifications using APTS. *Higher-Order and Symbolic Computation*, 16(1-2):63–92, 2003.

42 **N. G. Leveson, M. P. E. Heimdahl, and J. D. Reese.** Designing specification languages for process control systems: Lessons learned and steps to the future. *Proc. Joint 7th Eur. Softw. Eng. Conf. and 7th ACM SIGSOFT Symp. Foundations Softw. Eng. (ESEC/FSE'99)*, pages 127–145, Toulouse, FR, September 1999. Springer. LNCS 1687.

43 **N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese.** Requirements specification for process-control systems. *IEEE Trans. Softw. Eng.* 20(9):684–707, September 1994.

44 **N. G. Leveson, J. D. Reese, and M. P. E. Heimdahl.** SpecTRM: A CAD system for digital automation. *Proc. 17th Digital Avionics Systems Conf.*, Bellevue, WA, November 1998. IEEE/AIAA/SAE.

45 **N. Lynch.** *Distributed Algorithms*. Morgan Kaufman, San Mateo, CA, 1996.

46 **N. Lynch and F. Vaandrager.** Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

47 **S. Meyers and S. White.** Software requirements methodology and tool study for A6-E technology transfer. Technical report, Grumman Aerospace Corp., Bethpage, NY, July 1983.

48 **S. P. Miller,** March 1997. Personal communication.

49 **Steve Miller.** Specifying the mode logic of a flight guidance system in CoRE and SCR. *Proc. 2nd ACM Workshop on Formal Methods in Softw. Practice (FMSP'98),* 1998.

50 **NASA Software Engineering Research Infusion Program**. http://ic.arc.nasa.gov/researchinfusion.

51 **R. Paige.** Viewing a program transformation system at work. *Proc. Joint 6th Int. Conf. on Programming Language Implementation and Logic Programming (PLICLP) and 4th Int. Conf. on Algebraic and Logic Programming (ALP)*. Springer-Verlag, September 1994. LNCS 844.

52 **D. Y. W. Park, J. U. Skakkebæk, M. P. E. Heimdahl, B. J. Czerny, and D. L. Dill.** Checking properties of safety critical specifications using efficient decision procedures. *Proc. 2nd Workshop on Formal Methods in Softw. Practice (FMSP'98),* Clearwater Beach, FL, March 1998. ACM SIGSOFT.

53 **D. L. Parnas, G. J. K. Asmis, and Jan Madey.** Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2), April–June 1991.

54 **David L. Parnas and Paul C. Clements.** A rational design process: How and why to fake it. *IEEE Trans. Softw. Eng.*, SE-12(2):251–257, February 1986.

55 **David L. Parnas and Jan Madey.** Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, October 1995.

56 **David Lorge Parnas and Dennis K. Peters.** An easily extensible toolset for tabular mathematical expressions. *Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99),* pages 345–359, Amsterdam, March 1999. Springer LNCS 1579.

57 **N. Shankar, S. Owre, J. M. Rushby and D. W. J. Stringer-Calvert,** *PVS Prover Guide*, Version 2.4. Computer Science Lab, SRI International, Menlo Park, CA, November 2001.

58 **R. M. Smullyan.** *First-Order Logic*. Springer-Verlag, 1968. Republished by Dover Publications Inc., 1993.

59 **J. Sutton,** September 1997. Personal communication.

60 **J. M. Thompson, M. P. E. Heimdahl, and D. M. Erickson.** Structuring formal control systems specifications for reuse: Surviving hardware changes. *Proc. 5th NASA Langley Formal Methods Workshop (LFM2000),* Williamsburg, VA, June 2000.

61 **A. John 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.

62 **P. Wolper and B. Boigelot.** Verifying systems with infinite but regular state spaces. *10th Int. Conf. Computer-Aided Verification (CAV'98),* LNCS 1427, 1998.