SC(R)^3: Towards Usability of Formal Methods

Marsha Chechik
University of Toronto
Department of Computer Science
Toronto, Ontario M5S 3G4 Canada
chechik@cs.toronto.edu

Draft of August 20, 1999

Abstract

Formal methods have long been criticized as too expensive, too hard to apply (i.e., require a level of mathematical sophistication beyond that possessed by the majority of software developers), and not relevant for "real" software systems. In order to make formal methods more attractive, it is essential to improve their *usability*, i.e., create methods and tools that can be applied by software developers and enable the reuse of formal specifications across several development activities.

This paper presents an integrated toolset called SC(R)^3 aimed at increasing the usability of formal methods by amortizing the cost associated with creating precise software requirements. Requirements are specified in an easy-to-use-and-review SCR notation, and then used in checking requirements for correctness and in verifying consistency between annotated code and requirements.

In this paper we discuss motivations behind this work, describe several tools which are part of SC(R)^3, and illustrate their operation on an example of a Cruise Control system.

1 Introduction

Researchers have long been advocating the use of mathematically-precise ("formal") specifications in software projects [71]. These specifications aid in removing errors early in the software lifecycle and can be reused in a variety of ways: as a reference point during system design and during development of test cases, as documentation during maintenance, etc. However, there is a strong resistance against adopting formal methods in practice, especially outside the domain of safety-critical systems. The primary reason for this resistance is the perception of software developers regarding the applicability of formal methods – these methods are considered to be hard (require a level of mathematical sophistication beyond that possessed by many software developers), expensive, and not relevant for "real" software systems [31]. Although case-studies (e.g. [21, 47, 25, 82, 65, 60]) have shown the applicability and the effectiveness of formal methods in the specification and the analysis of industrial-size systems, this perception still remains. Currently, most research in formal methods concentrates on improving modeling languages and tool support to be able to specify and verify larger and more complex problems (e.g. [49, 58]). However, to facilitate a wide-spread use of formal methods, another, complimentary approach is necessary: to improve *usability* of the methods and tools and to demonstrate the cost-effectiveness of applying them to software systems. Such cost-effectiveness, in our opinion, comes from reuse, automation, and utilization of easy to use and review notations. The cost of creating formal documentation should be amortized throughout the software lifecycle, reusing this documentation for checking correctness of programs, generating test cases and test environments, etc.

In recent years, the Naval Research Lab-developed SCR notation emerged as a practical and usable tabular notation [41]. It has been effectively applied to a broad range of critical systems, including control systems for nuclear power plants [69], avionics systems [76, 28, 64, 61], space systems [24, 25], and military systems [42]. The SCR notation was developed as part of the Software Cost Reduction project [45, 46] (A-7). This notation specifies event-driven systems as communicating state machines which move between states as the environment changes. The functional part of the system requirements describes the values of the system's output variables as a function of the system's input (event) and internal state. These requirements can be formally specified by providing structured decision tables. For each output variable, there is a table which specifies how to compute the variable's value based on its previous value and input events (changes in the monitored environment quantities). A group at the Naval Research Lab developed an industrial-quality tool called `SCRTool` which allows specifying and reasoning about complex systems using SCR [41, 42, 44]. `SCRTool` performs checks to ensure that the tables are complete and consistent and symbolically executes requirements in order to ensure that the specification captures the intended behavior [43]. Since tabular notation allows for high-yield mechanical analysis, tools were developed to check SCR tables for correctness with respect to global properties using model-checking [77, 10] and theorem-proving [68, 4]. Some of these techniques are incorporated into `SCRTool`, making it an effective, scalable tool for reasoning about event-driven systems.

Representing logical formulae using tables is a powerful way to visualize information and to make specifications easy to write and review, as advocated by many researchers and supported by several tools. For example, David Parnas and his colleagues are working on methods and tools to document programs using tables [72, 70, 74, 1], and a group at Odyssey Research Associates are developing `Tablewise` — a tool to reason about decision tables [50]. Tables have also been heavily used in RSML [60, 66].

The goal of our project is to facilitate the reuse of tabular requirements throughout the software development lifecycle. We decided to reuse `SCRTool` and built an auxiliary tool, $SC(R)^3$ which stands for SCR Requirements Reuse. $SC(R)^3$ invokes `SCRTool` for specifying and executing SCR requirements, and compliments `SCRTool`'s verification capabilities by analyzing SCR tables using a symbolic model-checker `SMV` [63] and checking consistency between SCR requirements and annotated C code. In building the tool, we made the design decisions as recommended by the Working Group on Formal Methods for High Assurance Systems [39]:

Provide more automation. Our tool is completely automatic and uses model-checking and static analysis as its verification technologies.

Provide good feedback when errors are detected. When the analysis tool detects a problem, the error report must clearly identify the problem in the specification. Errors in specifications are reported using the native, SCR format. Errors in the code are reported using an easy-to-understand format that includes a program line number.

Provide tool integration and tool interoperability. Different tools are good for different tasks and must also be usable in concert. There should be no need to use more than a single notation. `SCRTool` has been designed to output specifications in ASCII-readable format called SSL. The grammar for this format is given in the User Guide [57]. This makes `SCRTool` easy to integrate with other tools, and we used it in the design of SC(R)^3.

Integrate methods and tools into the software development process. The resulting toolset is applicable to several requirements analysis activities and to code validation, thus allowing for a single development methodology. We envision the following as a possible methodology for using the future version of SC(R)^3:

1. A (human) requirements designer provides an SCR tabular specification of the system's required behavior and a set of global properties that should be satisfied by the system. Using a variety of verification and validation mechanisms, the designer gains confidence in these specifications.
2. Once the specification is considered correct, SC(R)^3 generates an implementation of the mode logic and stubs for interface code, which are filled in by a human developer.
3. Using the specification, SC(R)^3 generates test cases. Although the mode logic is correct by construction, the interface is likely to need testing.
4. The resulting implementation is analyzed for performance and optionally manually fine-tuned.
5. If manual fine-tuning was used, the code is automatically checked for consistency with its specifications.
6. Once the problems pointed out by the analysis are fixed, the code is tested using (a subset of) test cases generated in step (3) above.

Currently the SC(R)^3 toolset only includes techniques to analyze correctness of requirements and to check the correctness of code. In this paper we discuss the specification and analysis activities supported by SC(R)^3 using the Cruise Control system as the example. We emphasize the reuse and the usability issues associated with the verification activities supported by SC(R)^3. The major contributions of the paper are:

1. We present a tool that allows to reuse formal requirements throughout several analysis activities. This tool takes advantage of the NRL-developed `SCRTool` to specify formal requirements, check their consistency, and create partial specifications, essential for effective push-button verification.
2. We present a requirements verification method alternative to the one used in `SCRTool` and discuss tradeoffs between the two.
3. We show how to encapsulate the verification technology to make sure that property specification and counter-examples remain in the language of SCR rather than in the language of the verification tool. The technique to translate property specifications is new and its application results in significantly smaller models. We also take advantage of `SCRTool`-specified environmental assumptions to model the environment during the verification.

4. We show how to analyze consistency between code and requirements.

The rest of this paper is organized as follows: Section 2 briefly reviews the SCR method, describes requirements of the Cruise Control System and shows how these requirements can be mapped into SCR. Section 3 discusses the techniques employed in SC(R)^3 to effectively verify requirements for correctness. Correctness criteria are expressed in a variant of temporal logic CTL [20] augmented with SCR events. Section 4 describes the method to analyze correctness of code. Code is first commented with *annotations* — statements about changes in requirement-level variables. SC(R)^3 then checks the correspondence between the code and the annotations, to ensure that the code is correctly annotated. Afterwards, it checks consistency between the code and the requirements. In Section 5 we survey related work. Finally, Section 6 summarizes the paper, discusses limitations of our work, and outlines future research directions.

2 Specifying Software Requirements

In this section we briefly overview the SCR development methodology and the SCR formal model and describe the Cruise Control System. For a more complete description of the SCR method and requirements model, please refer to the work of Heitmeyer and her group [41, 42].

2.1 SCR Requirements Model

The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of the Software Cost Reduction project [2, 46]. The project resulted in SCR requirements and design standards, as well as in guidelines for software development using SCR. A complete SCR requirements specification contains behavioral, functional, precision, and timing requirements of a software system, as well as assumptions about the environment in which the system will operate.

The SCR model is used to specify reactive systems by mapping a (non-deterministic) specification of the system environment to a (deterministic) system behavior. The system environment contains *monitored* and *controlled* quantities – quantities that the system monitors and controls, respectively. The environment non-deterministically produces a sequence of *input events* where each event is a change in some monitored quantity. It is also assumed that the system receives one input event at a time (*Single Input Assumption*). The system reacts to *input events*, changes in the environment, by changing its state and updating values of its controlled variables. To keep track of its states, the SCR specification uses two types of auxiliary variables: *terms* and *mode classes* whose values are called *modes*. One can think of mode classes as parallelly-executing state machines with modes as their states. The system is considered to be exactly in one mode of each mode class, and one mode of each mode class is designated as the initial mode. Terms are results of intermediate computations.

More formally, an SCR specification Σ is a 4-tuple $\Sigma = (S, S_0, E^m, T)$, where S is a set of states, $S_0 \subseteq S$ is the set of initial states, E^m is the set of input events, and T is the (deterministic) transform describing the allowed state transitions. T maps an input event and the current state to a new state. A *variable* is any monitored or controlled variable, mode class, or term. The SCR requirements model includes a set $RF = \{r_1, ..., r_n\}$ of all variable names, and a function TY that maps each variable name to a set of its legal values. Therefore, a *state* maps each variable

$r_i \in RF$ to its value in $TY(r_i)$ A *condition* is a predicate defined on a system state, whereas an *event* is a predicate defined on two system states that differ in the value of at least one variable. An event occurs if there is a change of value of a variable or a condition. The notation "@T(a)" denotes an event and is defined by [41]

$$@\mathrm{T}(a) = \neg a \wedge a\prime$$

where the unprimed version of a condition a denotes the value of a in the old state, and the primed version denotes the value of a in the new state. Notation "@F(a)" is then a short-cut denoting an event @T$(\neg a)$. We distinguish between *primitive events* – of the form @T/@F (a) where a is a condition, and *conditioned events*. A *simple conditioned event* is denoted @T(a) WHEN $[b]$, where @T(a) is a primitive event, otherwise called the event's *triggering condition*, and b is a simple condition or a conjunction of simple conditions, called the event's WHEN *condition*. A simple conditioned event can be represented by a logical expression [41]:

$$@\mathrm{T}(a) \text{ WHEN } [b] = \neg\, a \wedge a\prime \wedge b,$$

A *conditioned event* is composed of simple conditioned events connected by logical connectors AND ($\wedge$) and OR ($\vee$). *Compound events* can be specified using boolean operators, connecting primitive events or multiple conditions. We assume that simultaneous events can occur only between variables related by environmental assumptions, defined below. Any event can be expressed as a logical statement; an event occurs if the logical statement that the event represents evaluates to true for a a pair of consecutive states.

The transform T can be described using three types of tables: mode transition tables, condition and event tables. These tables define changes of values of requirement variables, and there is one table for each mode class, variable, and term. A condition table describes a controlled variable or a term as a function of a mode and a condition; an event table describes a controlled variable or a term as a function of a mode and an event. A mode transition table describes a mode as a function of another mode and an event.

To compute the "new" state, T uses values of variables in the "old" and in the "new" state. In order to ensure that the "new" state can always be computed, the SCR method requires that the partial order $D \subseteq RF \times RF$ of dependencies between its variables be established. We say that a variable r_i depends on variable r_j, i.e. $(r_i, r_j) \in D$ if $r_j\prime$ is used in a function defining $r_i\prime$ [41], that is, the new value of r_i depends on the event involving the change of r_j. `SCRTool` uses the dependency checker to ensure that there are no circular dependencies between SCR variables. In particular, it ensures that monitored variables may not depend on any other variables, and that no other variables may depend on controlled variables.

The SCR method also allows to specify assumptions about the environment. If the environmental assumptions hold, the specification should guarantee that the behavior of the system is correct. An *environmental assumption* specifies constraints on the values of variables, imposed either by laws of nature or by other mode classes in the system. As such, assumptions are invariant constraints that must hold in all system states. For example, the water in a container cannot be too high and too low at the same time, and buttons can be either pressed or released but not both. In this paper, we will use the following types of assumptions (the terminology follows [5]):

implication $(a \rightarrow b)$. The state space in which a is true is a subset of the state space in which b is true.

strict implication $((a \rightarrow b)$ AND $(@\mathrm{T}(a) \rightarrow (b$ AND $b'))$ AND $(@\mathrm{F}(b) \rightarrow ($NOT a AND NOT $a')))$. This assumption is similar to implication except that when a becomes true, b should already be true, and when b becomes false, a should already be false.

exclusion $(\forall i, 0 \leq i \leq n, (a_i \rightarrow$ NOT $(a_0$ OR $\cdots$ OR a_{i-1} OR a_{i+n} OR $\cdots$ OR $a_n)))$[1]. The state space is partitioned such that at most one member of the exclusion is true in each state. Note that this environmental assumption is best described using the XOR operator which `SCRTool` Version 1.7 of March 9, 1998 does not support.

2.2 Specifying the Cruise Control System

A Cruise Control system specified by Jim Kirby [56] is responsible for keeping an automobile traveling at a certain speed. The driver accelerates to the desired speed and then presses a button on the steering wheel to activate the cruise control. The cruise control then maintains the car's speed, remaining active until one of the following events occurs: (1) the driver presses the brake pedal; (2) the driver presses the gas pedal; (3) the car's speed becomes uncontrollable; (4) the engine stops running; (5) the driver turns the ignition off; (6) the driver turns the cruise control off. If any of the first three events listed above occur, the driver can re-activate the cruise control system at the previously set speed by pressing a 'resume' button.

In this case study we made a number of simplifications to the SCR method. We did not use terms or condition tables and represented monitored and controlled variables as booleans or enumerated types. The reason was that our code consistency analyzer (see Section 4) can handle only this subset of SCR. Monitored variables in our case study indicate the state of

Mode	**Description**
`Off`	vehicle's ignition is off
`Inactive`	vehicle's ignition is on, but the cruise control is not on
`Cruise`	the cruise control is on and can control the vehicle's speed
`Override`	the cruise control is on but cannot control the vehicle's speed

Table 1: Modes of the mode class `CC`.

the ignition, brake, and acceleration (represented using boolean variables `Ignition`, `Brake` and `Accel`), the buttons operating the cruise control system (boolean variables `b_Cruise`, `b_Resume` and `b_Off`), and the speed of the car. We refer to the speed to be maintained by the Cruise Control as `Sp_Desired`, the speed of the car as `Sp_Vehicle`, and define the following predicates on the value of the later:

`Toofast`	the car's speed is out of control: `Sp_Vehicle` > `MAX_SPEED`
`Speed = speed_fast`	`Sp_Vehicle` > `Sp_Desired` + `THRESHOLD`[2]

[1]Environmental assumptions cannot be specified using first-order formulas. Instead, exclusion is represented as a set of n properties, one for each variable.

[1]The value of THRESHOLD is not specified in [56]. That document suggests to use a "value chosen according to how wide a speed variation is regarded as acceptable."

Old Mode	Event	New Mode
`Off`	@T(`Ignition`)	`Inactive`
`Inactive`	@F(`Ignition`)	`Off`
	@T(`b_Cruise`) WHEN `Ignition` AND `Running` AND NOT (`Toofast`) AND NOT (`Brake`) AND NOT (`Accel`)	`Cruise`
`Cruise`	@F(`Ignition`)	`Off`
	@T(`Toofast`)	`Inactive`
	@F(`Running`)	`Inactive`
	@T(`Brake`)	`Override`
	@T(`Accel`)	`Override`
	@T(`b_Off`)	`Override`
`Override`	@F(`Ignition`)	`Off`
	@F(`Running`) WHEN `Ignition`	`Inactive`
	@T(`b_Resume`) WHEN `Ignition` AND `Running` AND NOT (`Toofast`) AND NOT (`Brake`) AND NOT (`Accel`)	`Cruise`
	@T(`b_Cruise`) WHEN `Ignition` AND `Running` AND NOT (`Toofast`) AND NOT (`Brake`) AND NOT (`Accel`)	`Cruise`

Initial Mode: `Off` WHEN NOT `Ignition`

Table 2: Mode transition table for mode class `CC` of the cruise control system.

`Speed = speed_slow`	`Sp_Vehicle` $<$ `Sp_Desired - THRESHOLD`
`Speed = ok`	(`Sp_Desired` $-$ `THRESHOLD` $\leq$ `Sp_Vehicle`) $\wedge$ (`Sp_Vehicle` $\leq$ `Sp_Desired` $+$ `THRESHOLD`)

In addition, the system keeps track of the number of miles the car traveled to determine if it is due for service or its oil or air filter need to be replaced.

Modes	**Events**			
`Cruise`	@T(`Speed=speed_slow`)	@T(`Speed=speed_ok`)	@T(`Speed=speed_fast`)	@F(`Inmode`)
`Cruise`	@T(`b_Resume`) AND @T(`Inmode`) WHEN (`Speed=speed_slow`)	@T(`b_Resume`) AND @T(`Inmode`) WHEN (`Speed=speed_ok`)	@T(`b_Resume`) AND @T(`Inmode`) WHEN (`Speed=speed_fast`)	@F(`Inmode`)
`Throttle'` =	`ThrottleAccel`	`ThrottleMaintain`	`ThrottleDecel`	`ThrottleOff`

Initial: `ThrottleOff`

Table 3: Event table for the controlled variable `Throttle`.

The environmental assumptions of the Cruise Control System are

- The engine is running only if the ignition is on, i.e. `Running` and `Ignition` are related by strict implication;
- The car can be going "too fast" only if it is running, i.e., `Toofast` and `Running` are related by strict implication; and
- Off, Cruise and Resume buttons cannot be pressed simultaneously, i.e., `b_Off`, `b_Cruise` and `b_Resume` are related by exclusion.

The Cruise Control system has one mode class `CC`, whose modes are described in Table 1. The mode transition table of mode class `CC` is shown in Table 2. Each row of the table specifies an event that activates a transition from the mode on the left to the mode on the right. The system starts in mode `Off` if `Ignition` is false, and transitions to mode `Inactive` when `Ignition` becomes true. Once in mode `Inactive`, the system remains there until `Ignition` becomes false at which point it switches to mode `Off`. The system also transitions from `Inactive` to `Cruise` if `b_Cruise` becomes true while `Running` is true and `Toofast`, `Brake` and `Accel` are false. Variables that are not explicitely present in the event are considered to be "don't care" conditions.

The Cruise Control system defines the following controlled variables: `Throttle` to control the car throttle, `Oil` and `Air` to indicate that the car needs an oil or an air filter change, respectively, and `Service` to indicate that the car is due for service. Throttle is active only when the system is in mode `Cruise`. The event table for variable `Throttle` is shown in Table 3. For example, `Throttle` assumes the value `ThrottleAccel` (indicating that the throttle is in the accelerating position) if the speed becomes too slow while the system is in mode `Cruise` (see first row of the table), or when the system returns to the mode `Cruise` (indicated by @T(`Inmode`)) because the Resume button has been pressed (@T(`b_Resume`)), and the speed has been determined to be too slow, as indicated in the second row of the table. Note that this event does not violate the *Single Input Assumption* of SCR: it is composed of the change of the monitored variable `b_Resume` that leads to the change in the modeclass. Based on the semantics of SCR, both of these events can be detected in one state.

Thus, the set RF of variable names in the Cruise Control System consists of 12 monitored variables `Ignition`, `Running`, `Brake`, `Accel`, `Toofast`, `b_Cruise`, `b_Resume`, `b_Off`, `Speed`, `Omiles`, `Amiles`, `Smiles`, the mode class `CC`, and four controlled variables `Throttle`, `Oil`, `Air` and `Service`. The type definitions include

$$
\begin{array}{lcl}
TY(\texttt{Ignition}) & = & \{\text{true, false}\} \\
TY(\texttt{Running}) & = & \{\text{true, false}\} \\
 & \ldots & \\
TY(\texttt{Speed}) & = & \{\texttt{speed\_fast}, \texttt{speed\_fast}, \texttt{speed\_ok}\} \\
TY(\texttt{CC}) & = & \{\texttt{Off}, \texttt{Inactive}, \texttt{Cruise}, \texttt{Override}\} \\
TY(\texttt{Omiles}) & = & \{\texttt{omiles\_low}, \texttt{omiles\_inter}, \texttt{omiles\_high}\} \\
TY(\texttt{Amiles}) & = & \{\texttt{amiles\_low}, \texttt{amiles\_inter}, \texttt{amiles\_high}\} \\
TY(\texttt{Smiles}) & = & \{\texttt{smiles\_low}, \texttt{smiles\_inter}, \texttt{smiles\_high}\}
\end{array}
$$

TY(`Throttle`) = {`ThrottleDecel`, `ThrottleAccel`,`ThrottleMaintain`, `ThrottleOff`}

From Tables 2 and 3 we can conclude that `CC` depends on $CCvars$ = {`Ignition`, `b_Cruise`, `Toofast`, `Running`, `Brake`, `Accel`, `b_Off`, `b_Resume`, `b_Cruise`}, and `Throttle` depends on `Speed`, `b_Resume`, and `CC`.

Since the Cruise Control System has been previously specified in SCR [56, 7], we had no problem using `SCRTool` to capture its requirements. In fact, this process was performed by an undergraduate student doing a summer project. The following were discovered while observing this student: (1) it was possible to teach the student how to use the tool effectively in about two hours; (2) the syntax checker was extremely useful, as the student struggled with numerous misspellings and variable omissions in our specification; (3) the two ways of specifying assertions, supported by `SCRTool`, namely, via the environmental assertion dictionary and the specification assertion dictionary, caused problems. The similarity of names and the lack of any restrictions on variables in these dictionaries confused the student who was not sure where to put environmental assumptions and where correctness claims about the system (described in Section 3).

3 Checking Consistency of Requirements

The SCR state-machine model presented in the previous section describes the *operational*, or *model-oriented* specification of the system [81]. An alternative, *property-oriented* specification describes the required properties of the system, e.g. "the light will eventually become green", or "reversing a list twice gives us the original list". Such properties are typically described in some temporal logic. Both types of specifications have their advantages: it is easier to reason about completeness of an operational specification, and such specifications are often executable. Property-oriented specifications, on the other hand, do not give an implementation bias and concisely represent requirements of the system. Combining both types of specifications into a *dual-language approach* [67] allows not only to take advantage of the best features of each type, but also to detect inconsistencies between them. Often these inconsistencies lead to modifications both in the operational model and in the statement of properties [67, 3]. For many specifications, such inconsistencies can be uncovered automatically using *model-checking* [20]. Model-checking is a technique which takes a (finite[3]) transition relation of the system and a set of temporal logic properties and checks if the properties hold.

Global system properties appear in the majority of SCR specifications starting with A-7 [46]. This gave rise to using model-checking to check consistency of SCR mode tables [6, 77, 42, 10]. `SCRTool` implements property verification via a model-checker Spin [49]. SC(R)^3 incorporates `SCRTool` and augments its verification capabilities by utilizing a different model-checker called `SMV` [63]. In this section we describe the verification supported by `SCRTool` in Section 3.1, then follow it with the verification added by SC(R)^3 in Section 3.2, and finish by discussing the relative applicability of both methods. In what follows we refer to `SCRTool` Version 1.7 of March 9, 1998.

[3] Modelcheckers for infinite state space are described in Section 5.

3.1 `SCRTool`-Supported Verification

Spin [49] is a tool for analyzing the logical consistency of concurrent systems using explicit state exploration. The system is specified in a guarded command language Promela [48], and its correctness claims are represented in the form of `assert` statements or Linear-Time Temporal Logic (LTL) properties [62].

The majority of global system properties can be characterized as *state invariants*, e.g. "whenever the system is in mode `Override` of modeclass `CC`, the system is running and the ignition is on", or *transitions invariants*, i.e., properties involving pairs of states, e.g., "whenever the system is in mode `Cruise` of modeclass `CC` and the event `@F(Ignition)` occurs, the system should immediately transition into mode `Off`" [42]. Such properties are easily expressed in the form of *assertions*, e.g.

```
ASSERT CC=Override => Running AND Ignition
ASSERT CC=Cruise AND @F(Ignition) => CC'=Off
```

The SCR "prime" notation allows to easily reason about variables in the next state.

The `SCRTool` tool translates SCR specifications into Promela by explicitly representing every variable in its old and its new state, so Promela variables X`_OLD` and X`_NEW` are created for each SCR variable X. The following is a fragment of Promela translation of the mode transition table of the Cruise Control system:

```
d_step { /* calculate in one "step" */
  if
   ...
   /* mode: Inactive, event: @F(Ignition) */
    :: (Ignition_OLD && (!Ignition_NEW)) && (CC_OLD == Inactive)
         -> CC_NEW = Off;
    /* mode: Cruise, event: @F(Ignition) WHEN Running */
    :: ((Ignition_OLD && (!Ignition_NEW)) && Running_OLD) && (CC_OLD == Cruise)
         -> CC_NEW = Off;
    ...
    :: else skip;
    fi;
}
```

In a single step, evaluate every condition and non-deterministically pick the one that evaluates to true. Follow it by the statement on the right hand side of $\rightarrow$. So, if in the old state, `CC=Inactive` and `Ignition` is true, and in the new state `Ignition` is false, then in the new state `CC` should also be `Off`. If none of the conditions are true, then no action takes place.

3.1.1 Abstraction

Like all model-checkers, verification algorithms in Spin are exponential in the number of variables in the model, thus creating a problem of state explosion. In addition, Spin can only process finite models. In order to combat these problems, SCR researchers proposed several abstraction techniques that reduce state space and make the specification finite [42, 10]. Two of the three abstraction methods outlined in [42] are:

• *Variable Restriction (or Slicing [80]): Remove Irrelevant Variables.* This method uses the set of variable names which occur in the formula being analyzed to eliminate unneeded variables from the analysis; those can be monitored variables or variables defined via tables. Assume that $\mathcal{O} \subseteq RF$ is the set of variables occurring in a formula under analysis q, and $\mathcal{O}^*$ is the reflexive and transitive closure of $\mathcal{O}$ under the dependency relation D of the SCR state machine Σ. Checking q on the abstraction that removes tables and input specification of variables in $RF - \mathcal{O}^*$ from Σ is both sound and complete [10].

For example, when checking the property `ASSERT CC=Override` $\Rightarrow$ `(Running` AND `Ignition)` on the model of the Cruise Control System, we eliminate all variables except those in the set *CCvars* (see Section 2.2). This method typically results in a smaller Promela model and thus significantly reduces time and memory required for the verification. `SCRTool` Variable Dependency browser allows users to pick the variables present in the property, then computes $\mathcal{O}^*$ and removes other variables from the specification.

• *Replace Detailed Variable with Abstract Variable.* This abstraction is based on replacing a large or infinite-domain variable with a set of predicates. For example, in the specification of the Cruise Control System we replaced the actual variable `Sp_Vehicle` by four predicates, `Toofast`, `Speed=speed_fast`, `Speed=speed_slow`, `Speed=speed_ok`. The technique is easily shown to be sound; and it is complete if every state in the equivalence class resulting from replacing variables by predicates is reachable from any other state in the equivalence class in a finite number of states [10]. This technique is very effective in reducing the state space of systems under analysis, although it can only be applied manually.

3.1.2 What is Missing?

The majority of global properties of SCR specifications encountered in practice can be represented using assertions [10, 77]. However, more complex properties are sometimes necessary. Such properties, e.g. fairness and liveness, can be part of the specification or be added by the analyst to help debug the SCR state machines. Such properties can be expressed in LTL and verified using Spin, although this approach suffers from several problems from the usability point of view:

- Users have to understand how SCR specifications are translated into Promela and ask their questions in terms of newly-created Promela variables X`_NEW` and X`_OLD` rather than in terms of the familiar SCR notation involving events and WHEN conditions. This process is error-prone and forces the user to understand another specification language.
- Spin takes LTL formulas one-at-a-time, thus making it difficult to keep track of which formulas are being verified. Even if a log of such formulas is kept,
  - the formulas cannot be reviewed by non-specification experts because they are no longer expressed in SCR, and
  - these formulas need to be fed into Spin one-by-one if a change to the specification is made and its properties need to be verified one more time.
- Finally, counter-examples generated by Spin are not given in terms of SCR events.

To sum up, the use of Spin in the `SCRTool` tool is not fully encapsulated, making it difficult to verify properties outside those that can be represented in terms of assertions.

In addition, we feel that Promela is not perfectly suited for representing SCR models: it has an asynchronous semantics, and constructs like `d_step` are necessary to ensure that all SCR variables change their values in one step. More importantly, it is difficult to explicitly model the environment: the state-machine semantics of Spin is just not general enough to represent complex relationships between input variables that are often encountered in practice [10].

3.2 Verification With SC(R)^3

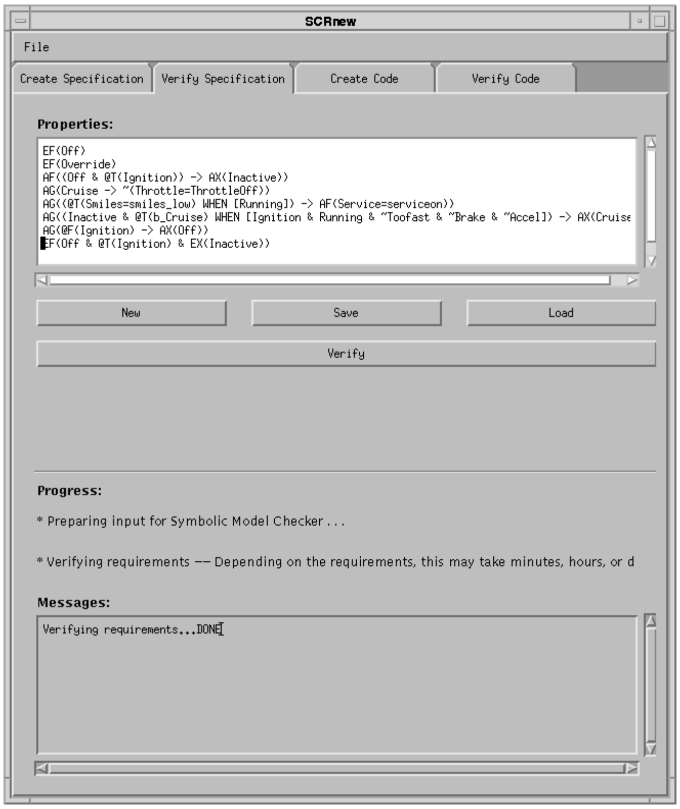

Figure 1: Requirements analysis with SC(R)^3.

We chose to use a synchronous model-checker `SMV` [63] to augment the verification techniques used in `SCRTool`. `SMV` was first used to analyze mode transition tables by Joanne Atlee and her colleagues in [7, 77]. They built a tool called `gensmv` that took an SCR specification consisting of boolean monitored variables and mode transition tables and translated it into the `SMV` notation. This method was shown to be effective in verifying properties of the A-7, whose state space can theoretically be as large as 2^{117} states [77].

SC(R)^3 borrowed many ideas from `gensmv`. However, it deviates from `gensmv` in several important ways: (1) it uses the complete SCR specification[4] expressed in the `SCRTool`-produced SSL [57] format; (2) it uses `SCRTool` for specification of environmental assumptions; (3) it translates properties involving events differently, resulting in creation of much smaller `SMV` models;

[4] `SMV` cannot handle infinite domain systems, thus we have to manually abstract infinite-domain variables into predicates and subranges.

and (4) it captures SMV counter-examples and translates them into the SCR notation, allowing to make the use of the model-checker completely transparent to the user.

Requirements are analyzed using SMV via the SC(R)^3 screen like the one shown in Figure 1. All global properties are specified in a single window and are later stored together with the SCR specification. In the remainder of this section we describe the verification facilities that are not part of SCRTool and illustrate them using the Cruise Control system.

3.2.1 Specifying Correctness Criteria

The language for specifying correctness properties in SC(R)^3 is an extension of CTL (Computational Tree Logic). CTL is a branching-time temporal logic [20] which allows quantification over some or all possible futures. CTL formulae are defined recursively: all propositional formulae are in CTL; if f and g are in CTL, so are $!f$(negation), $f \,\&\, g$ (conjunction), $f \mid g$ (disjunction), and $f \rightarrow g$ (implication). Furthermore, the universal (A) and the existential (E) quantifiers are used alongside the "next state" (X), "until" (U), "future" (F) and "global" (G) operators. Thus, the formula $AG(f \rightarrow g)$ means that it is invariantly true that f implies g.

In order to enable SCR users to express properties about conditioned events, we extend CTL with three unary logic connectives – @T, @F and WHEN, as proposed by Atlee and Buckley in [7]. However, we give these connectives a more familiar semantics:

$$
\begin{aligned}
\texttt{@T}(v) &= \neg v \wedge \texttt{next}(v) \\
\texttt{@F}(v) &= v \wedge \neg\texttt{next}(v) \\
\texttt{WHEN}(v) &= v
\end{aligned}
$$

For example, a property [**Property 1**]

> When the oil has not been changed long enough, the Oil indicator will be turned on, as long as the car is running.

can be expressed as $AG((\texttt{@T(Omiles = omiles\_low) WHEN [Running]}) \rightarrow AF(\texttt{Oil = oilon}))$ and a property [**Property 2**]

> From any mode, the system will react to turning the ignition off by going into state Off.

can be expressed as $AG(\texttt{@F(Ignition)} \rightarrow AX(\texttt{CC = Off}))$.

SCR connectives can be easily translated into Promela or LTL (see Section 3.1). However, translating them to CTL is more difficult due to the branching-time nature of this language: CTL quantifies over all possible futures and thus it is not clear if the event should happen in *all* next states (AX) or only in *some* (EX). It is essential to evaluate the event in the correct set of next states, the same ones as the rest of the property refers to. The algorithm for expanding events follows.

1. Recursively rewrite the property to remove $\rightarrow$ and ensure that only propositional formulas and primitive events are negated. This can be easily done, considering that all CTL operators except U ("until") have duals, e.g. $(!AG(f) = EF(!f))$ [20]. For properties involving U, stop and ask the user for help.

2. Attempt to automatically determine the quantification of the "next" state so that the intended semantics of the property is preserved. Let $@T(r_i = v_i)$ be an event on variable r_i[5] and f be the largest CTL subformula in which this event occurs *atomically* (not within the scope of any temporal operator). The algorithm to expand f follows:

Let Q_1 be the quantifier for the temporal operator encompassing f.
If f does not include any temporal operators
  Expand the event as $(!(r_i = v_i)\ \&\ Q_1X(r_i = v_i))$
  /* use Q_1 as the quantifier for the "next"state. */
Else, for each temporal operator Q_2T in f
  If $Q_1 \neq Q_2$, stop and ask the user for help.
  /* we will attempt to expand the event only for cases in which the quantifiers for the outer and the inner temporal operators are the same */
  Let b be the propositional formula within the scope of Q_2T
  If Q_1 is the universal quantifier
    Expand the event as $(!(r_i = v_i)\ \&\ AX(r_i = v_i))$
  Else /* trying to expand the existential quantification */
    Replace $[@\mathrm{T}(r_i = v_i)\ \&\ EX(b)]$ by $[!(r_i = v_i)\ \&\ EX((r_i = v_i)\ \&\ b)]$
    Replace $[@\mathrm{T}(r_i = v_i) \mid EX(b)]$ by $[(!(r_i = v_i) \mid EX(b))\ \&\ EX((r_i = v_i) \mid b)]$
    Replace $[@\mathrm{T}(r_i = v_i)\ \&\ EF(b)]$ by
      $[!(r_i = v_i)\ \&\ (EX((r_i = v_i)\ \&\ EF(b)) \mid (EX(r_i = v_i)\ \&\ b))]$
    Replace $[@\mathrm{T}(r_i = v_i) \mid EF(b)]$ by
      $[(!(r_i = v_i) \mid EF(b))\ \&\ (b \mid EX((r_i = v_i) \mid EF(b)))]$
    Replace $[@\mathrm{T}(\mathrm{r_i} = \mathrm{v_i})\ \&\ EG(b)]$ by
      $[!(r_i = v_i)\ \&\ EX((r_i = v_i)\ \&\ EG(b))\ \&\ b]$
    Replace $[@\mathrm{T}(r_i = v_i) \mid EG(b)]$ by
      $[(!(r_i = v_i) \mid EG(b))\ \&\ (EX(r_i = v_i) \mid b)\ \&\ EX((r_i = v_i) \mid EG(b))]$
    If the replacement cannot be made, ask user for assistance.

Proof outline:
We want to prove that the above algorithm is correct, that is, the resulting translation corresponds to the meaning expressed in extended-CTL. The proof proceeds by induction on the number of temporal operators in f.
Base case: There are no temporal operators within f. Since the event is not negated, its meaning is determined by the quantifier of f. For example, $AG(a\ \&\ @\mathrm{T}(b)) \Rightarrow AG(a\ \&\ (!b\ \&\ AX(b)))$
Inductive Hypothesis: Assume the algorithm is correct if f has n quantifiers.
Inductive Case: Let Q_2T be the $n+1$st temporal operator, and assume that it is equal to Q_1 and that the algorithm worked without consulting the user.

If Q_2 is a universal quantifier, the "next" operator in the event should be evaluated in all next states. In addition, since b will be evaluated in all next states, there is no need to bundle r_i with b.

[5]The treatment of $@\mathrm{F}(r_i = v_i)$ is similar.

If Q_2 is the existential quantifier. We point out that

$$EF(b) = b \mid EX(EF(b)) \tag{1}$$

$$EG(b) = b \,\&\, EX(EG(b)) \tag{2}$$

We will prove correctness of the expansion of $@\mathrm{T}(r_i = v_i) \mid EG(b)$. The rest are similar.

First we determine that the expansion of the event should contain the existential quantifier and want to ensure that the right "next" state is being used. Thinking of $(r_{i'} = v_i)$ as the value of r_i in the next state, we proceed:

$$\begin{aligned}
& @\mathrm{T}(r_i = v_i) \mid EG(b) \\
= \; & (!(r_i = v_i) \,\&\, (r_{i'} = v_i)) \mid EG(b) \\
= \; & (!(r_i = v_i) \mid EG(b)) \,\&\, ((r_{i'} = v_i) \mid EG(b)) \\
= \; & (!(r_i = v_i) \mid EG(b)) \,\&\, ((r_{i'} = v_i) \mid (b \,\&\, EX(EG(b)))) \\
= \; & (!(r_i = v_i) \mid EG(b)) \,\&\, ((r_{i'} = v_i) \mid b) \,\&\, ((r_{i'} = v_i) \mid EX(EG(b)))
\end{aligned}$$

In the first conjunct of the expansion of $EG(b)$, the exact "next" state is not important because b is evaluated in the current state, whereas in the second conjunct we should use the same "next" state, which can be accomplished by adding the disjunct $(r_i = v_i)$ to $EG(b)$. Thus, the algorithm returns $(!(r_i = v_i) \mid EG(b)) \,\&\, (EX(r_i = v_i) \mid b) \,\&\, EX((r_i = v_i) \mid EG(b))$. ■

The above algorithm has been implemented in SC(R)[3]. Using this algorithm, Property 1 is translated as follows:

$$\begin{aligned}
& AG((@\mathtt{T}(\mathtt{Omiles} = \mathtt{omiles_low})\ \mathtt{WHEN}\ [\mathtt{Running}]) \rightarrow AF(\mathtt{Oil} = \mathtt{oilon})) \\
= \; & AG((@\mathtt{T}(\mathtt{Omiles} = \mathtt{omiles_low}) \,\&\, \mathtt{Running}) \rightarrow AF(\mathtt{Oil} = \mathtt{oilon})) \\
= \; & AG(!@\mathtt{T}(\mathtt{Omiles} = \mathtt{omiles_low}) \mid !\mathtt{Running} \mid AF(\mathtt{Oil} = \mathtt{oilon})) \\
= \; & AG(!(!(\mathtt{Omiles} = \mathtt{omiles_low}) \,\&\, AX(\mathtt{Omiles} = \mathtt{omiles_low})) \mid !\mathtt{Running} \mid AF(\mathtt{Oil} = \mathtt{oilon})) \\
= \; & AG((!(\mathtt{Omiles} = \mathtt{omiles_low}) \,\&\, AX(\mathtt{Omiles} = \mathtt{omiles_low}) \,\&\, \mathtt{Running}) \rightarrow AF(\mathtt{Oil} = \mathtt{oilon}))
\end{aligned}$$

whereas Property 2 is translated as

$$\begin{aligned}
& AG(@\mathtt{F}(\mathtt{Ignition}) \rightarrow AX(\mathtt{CC} = \mathtt{Off})) \\
= \; & AG(!@\mathtt{F}(\mathtt{Ignition}) \mid AX(\mathtt{CC} = \mathtt{Off})) \\
= \; & AG(!(\mathtt{Ignition} \,\&\, AX(!\mathtt{Ignition})) \mid AX(\mathtt{CC} = \mathtt{Off})) \\
= \; & AG((\mathtt{Ignition} \,\&\, AX(!\mathtt{Ignition})) \rightarrow AX(\mathtt{CC} = \mathtt{Off}))
\end{aligned}$$

Note that the algorithm works automatically for the absolute majority of properties expressed for SCR, including one-state and two-state invariants, and reachability and liveness properties. The algorithm can also be generalized to cover a larger class of extended-CTL formulas, but so far we have not yet encountered formulas for which it was necessary to do so.

An alternative solution was given by Atlee and her colleagues who proposed keeping track of the previous values of variables involved in events [7]. If $\mathtt{p}(X)$ is the previous-state value of variable X, and if v is a propositional CTL formula (e.g., the one without temporal modalities), the connectives are defined as follows:

$$\begin{aligned}
@\mathtt{T}(v) &= \neg\mathtt{p}(v) \wedge v \\
@\mathtt{F}(v) &= \mathtt{p}(v) \wedge \neg v \\
\mathtt{WHEN}(v) &= \mathtt{p}(v)
\end{aligned}$$

Although this technique allows for translating arbitrary extended-CTL formulas, it suffers from a number of problems, the main one being the introduction of a large number of extra variables into the SMV model. Since the running time of `SMV` grows exponentially with the number of variables in its models, this approach severely limits the domain of problems that can be automatically analyzed. In addition, the properties are specified from the "new state" point of view. Under Atlee's approach, Property 2 should be expressed as $AG(@\mathtt{T}(\mathtt{Ignition}) \rightarrow (\mathtt{CC} = \mathtt{Off}))$. The SCR tables are specified from the perspective of the "old state", and thus we feel that our approach of specifying properties from the same perspective is more intuitive.

3.2.2 Translating SCR Specifications

The process of automatically translating SCR specifications into `SMV` includes using environmental assumptions to correctly model the behavior of the environment and using SCR tables to convert them into `SMV` state machines.

Environmental assumptions in the form of a logical expression involving values of variables in the old and in the new state are specified using the `SCRTool`. These assumptions are translated into `SMV` using the algorithm inspired by the work of Atlee in [5]. For example, the environmental assumption that `Running` and `Ignition` are related by strict implication gives rise to the following specification of the environment:

```
TRANS
  Running -> Ignition

TRANS
  ( (!Ignition & next(Ignition) & !Running & !next(Running)) |
    (Ignition & !next(Ignition) & !Running & !next(Running)) |
    (Ignition & next(Ignition) & Running & !next(Running)) |
    (Ignition & next(Ignition) & !Running & next(Running)) |
    ((Running)=(next(Running))) & ((Ignition)=(next(Ignition)))
  )
```

Here, `next(`X`)` refers to the value of X in the next state. In producing this specification, we expanded the definition of strict implication given in Section 2 to remove "$\rightarrow$" and translated the resulting two-state formula into the `SMV` notation.

Afterwards, SC(R)^3 produces a `TRANS` relation to model the SCR's *Single Input Assumption* (see Section 2.1). The `TRANS` relation involves all monitored variables in the system and indicates that only one of them can be different from its `next`-state value. The ability to specify the restriction on input variables using logic statements as part of the `TRANS` relation rather than via state-machines, as in Promela, increases the correctness of generated models and makes the verification more meaningful.

SC(R)^3 automatically translates SCR tables into their `SMV` state machine representation using the semantics of [41]. Events are expressed as explicit current-state/next-state expressions. For example, the mode transition table of the Cruise Control system can be translated as follows [7]:

```
next(CC) := case
    CC=Off & (!(Ignition) & (next(Ignition))) : Inactive;
    CC=Inactive & ((Ignition) & !(next(Ignition))) : Off;
    CC=Inactive & Ignition & Running & !(Toofast) & !(Brake) & !(Accel) &
```

```
                  (!(b_Cruise) & (next(b_Cruise))) : Cruise;
    CC=Cruise & ((Ignition) & !(next(Ignition))) : Off;
    CC=Cruise & (!(Toofast) & (next(Toofast))) : Inactive;
    CC=Cruise & ((Running) & !(next(Running))) : Inactive;
    CC=Cruise & (!(Brake) & (next(Brake))) : Override;
    CC=Cruise & (!(Accel) & (next(Accel))) : Override;
    CC=Cruise & (!(b_Off) & (next(b_Off))) : Override;
    CC=Override & ((Ignition) & !(next(Ignition))) : Off;
    CC=Override & Ignition & ((Running) & !(next(Running))) : Inactive;
    CC=Override & Ignition & Running & !(Brake) & !(Accel) & !(Toofast) &
                  (!(b_Resume) & (next(b_Resume))) : Cruise;
    CC=Override & Ignition & Running & !(Brake) & !(Accel) & !(Toofast) &
                 (!(b_Cruise) & (next(b_Cruise))) : Cruise;
    1 : CC;
esac;
```

Here, SMV starts evaluating conditions from the top, and as soon as one condition becomes true, the state machine assumes a new state. For example, when the system is in CC=Off and Ignition is false in the current state, and in the next state Ignition is true, then the next-state value of CC should be Inactive. Note that SMV executes the first transition whose guard is true rather than non-deterministically picking one, as in Promela. Since the SCR model assumes that the new state can always be computed deterministically, the Promela and the SMV models are equivalent. Also, SMV performs the transition synchronously, updating the values of all variables at the same time, thus allowing the state machines describing the variables to be listed in any order, rather than in the relative order of their dependencies, like in Promela [10]. SMV can also explicitly refer to the next state of variables, allowing us to avoid creating the new and the old values of all variables, as in the translation into Promela employed by SCRTool and thus achieving a much faster verification (see Section 3.2.3).

3.2.3 Verification and Counter-Examples

During the verification of CTL properties, SMV explores all possible behaviors of the model. Since the size of the model, in its worst case, grows exponentially to the number of variables, it is essential to reduce the number of variables. Simple abstraction like *variable restriction* (see Section 3.1.1) has a significant impact on the size of the model and thus on the verification time. This abstraction can be performed using SCRTool's Dependency Browser. The use of this abstraction results in creating several partial SCR specifications; however, unlike the Promela/Spin approach, we can still verify several properties for each partial specification in one run of SMV. For example, the 39 properties of the Cruise Control System (see Section 3.2.4) were split up into 20 properties of mode classes, 10 properties of Throttle, and 3 properties of Service, Oil and Air each. Most of these properties checked the general sanity of SCR tables, and only nine were the actual correctness properties.

SMV uses binary decision diagrams (BDDs) [12] to manipulate logical expressions. The size of BDDs is extremely sensitive to the variable ordering used to build them [77]. [77] reported that a good variable order can reduce the running time of SMV from days to seconds. Unfortunately, finding an optimal BDD ordering is NP-complete. The SMV version -- reorder uses hieristical techniques to reorder variables to achieve more compact BDDs. For all but the smallest systems, this option leads to a significantly better performance of SMV.

At the end of the verification, SMV either declares that a given property holds or gives a counter-example. Since we wanted to make calls to SMV completely transparent to the user, we designed SC(R)^3 to automatically capture SMV's counter-examples and translate them into the SCR format. This translation allows the user to easily determine where errors occur, without having to understand the intricacies of translation between the SCR and the SMV models.

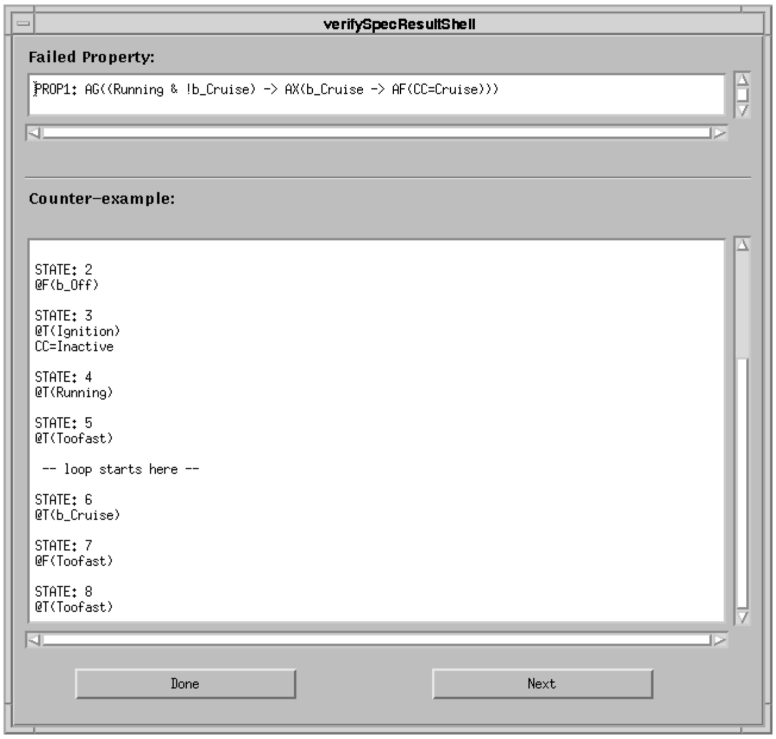

Figure 2: A counter-example generated during the requirements analysis.

For example, if we try to check the property "pressing the Cruise button at any point when the system is running will turn the cruise control system on", formalized as

$$AG((\texttt{Running} \ \& \ @T(\texttt{b\_Cruise})) \rightarrow AF(\texttt{CC} = \texttt{Cruise}))$$

SC(R)^3 reports a scenario that would violate this property, shown in Figure 2. Thus, the system does not react to changes in b_Cruise if the car is running too fast to be controlled by the automatic system.

3.2.4 Checking the Cruise Control System

We were successfully able to model and verify the automobile cruise control system using the SC(R)^3 toolset. The verification was performed on a moderately loaded Sun Enterprise450 with

Test	User time	System time	Real time	BDD nodes
Full specification — 17 variables				
Base	6.12s	0.03s	7.99s	232581
`--reorder`	5.75s	0.39s	7.91s	15150
Modeclass properties — 9 variables				
Base	0.03s	0.00s	0.06s	3393
`--reorder`	0.06s	0.01s	0.09s	3393
`Throttle` properties — 11 variables				
Base	0.05s	0.01s	0.09s	9546
`--reorder`	0.16s	0.01s	0.21s	5271
`Oil` properties — 11 variables				
Base	0.06s	0.03s	0.11s	10094
`--reorder`	0.09s	0.04s	0.14s	5236
`Air`, `Oil`, `Service` properties — 15 variables				
Base	2.22s	0.01s	2.25s	55194
`--reorder`	0.74s	0.03s	1.39s	17738

Table 4: Verification of the Cruise Control specification: experimental results.

Throttle Setting				
Mode	**Condition**			
`Cruise`	`Speed=speed_slow`	`Speed=speed_ok` AND NOT `Accel`	`Speed=speed_fast` AND NOT `Accel`	`Accel` AND NOT `Speed=speed_slow`
`Throttle`	`ThrottleAccel`	`ThrottleMaintain`	`ThrottleDecel`	`Throttle Off`

Table 5: Condition table for variable `Throttle`.

4X400 MHz CPUs and 4 GB of RAM using `SMV` version 2.5.3.1d of March 30, 1999. With the exact value of `Sp_Vehicle` abstracted into predicates `speed_ok, speed_fast, speed_slow`, the Cruise Control specification consists of 4 controlled and 12 monitored variables, translated into 17 `SMV` variables. The verification of the complete model without any optimization took 6.12 user, 0.03 user, and 7.99 real seconds, and resulted in creating 232581 BDD nodes. Applying the *variable restriction* abstraction reduced the verification time significantly, as shown in Table 4. In this table, we list the user, the real and the system time in seconds, and the total number of BDD nodes used during the verification, as reported by `SMV`. Since `CC` depends only on the variables from the set $CCvars = \{$`Ignition`, `b_Cruise`, `Toofast`, `Running`, `Brake`, `Accel`, `b_Off`, `b_Resume`, and `b_Cruise`$\}$ (see Section 2.2), the total number of variables in the `SMV` model for verifying properties of mode classes is 9. `Throttle` depends on `CC` (and transitively on all variables in *CCVars*), `b_Resume` (part of `CC`) and `Speed`; thus, the minimal `SMV` model for checking `Throttle` properties uses 11 variables. Properties of `Oil`, `Air` and `Service` are similar.

The verification effort yielded a number of errors, most of which could be traced back to the original specification [56]. We found no errors in the mode transition table because this table has been analyzed earlier by Atlee and her colleagues [7][6]. However, we did find errors in controlled variable tables of the original specification. For example, the cruise control system includes a controlled variable `Oil` which represents lighting up an indicator when the vehicle is due for an oil change, i.e., it has traveled a certain distance since its previous oil change. In Kirby's specification, this controlled variable is evaluated when we enter any mode other than `Off` (`@T(Inmode) when` [`Omiles_low`]). This problem was found using Property 1 given in Section 3.2.1. Thus, the vehicle could be traveling with the cruise control system turned off, not changing its modes and never setting `Oil` to `OilOn`. We modified the table for `Oil` to set the variable to `OilOn` when `Omiles_low` becomes true, regardless of the mode.

Another problem was found in the table for evaluating the throttle. [56] used a condition table to represent `Throttle`, as shown in Table 5. We found that the condition NOT `Accel` is redundant – the system exits mode `Cruise` as soon as `Accel` becomes true. We also suspected that the throttle might never become `Off`. To verify that, we checked the original specification against the (weak liveness) property

$$AG(!(\texttt{Throttle} = \texttt{ThrottleOff}) \rightarrow EF(\texttt{Throttle} = \texttt{ThrottleOff}))$$

if `Throttle` is not off, at least some paths will lead to a future state in which `Throttle` is off. This property ended up being false, confirming our suspicion. Note also that this property cannot be expressed using property assertions of `SCRTool`.

Finally, the correctness of Kirby's specification was conditional upon the time when the predicate on the vehicle's speed is evaluated. It works correctly only when `speed_ok`, `speed_slow` and `speed_fast` get evaluated *before* the system transitions into mode `Cruise`. Our event table (Table 3) fixes this problem.

The verification case study has been performed by an undergraduate student. We found that we had no problems training him in the effective use of the verification technology. We believe that the success of the study depended on the ability to express ad-hoc correctness criteria, on the availability of an automated counter-example generation, and the support for variable restriction in `SCRTool`.

3.3 SMV or Spin?

SC(R)^3 allows to perform verification either using Spin (through `SCRTool`) or using `SMV`. Each method has its own advantages. For example, it may be easier for users to specify assertions rather than express correctness properties in either CTL or LTL, and assertions are only supported in Spin. However, Promela models generated by `SCRTool` contain roughly twice the number of variables of the `SMV` models, because each SCR variable needs to be captured in its "new" and its "old" state. This makes verification with `SMV` significantly more feasible, especially when the properties are expected to hold in the model. For example, we tried running Spin Version 3.3.3. of 21 July 1999 to verify properties of the mode class of the Cruise Control System. These properties

[6] However, our study showed that a number of conditions used in Atlee's specification were not necessary, e.g., specifying that transitions from `Override` to `Cruise` occur only when `Running` and `Ignition` are true. These conditions are implied by other transitions in the mode table and did not need to be specified explicitly.

hold in the model, and `SMV` was able to verify them in less than a second. Verification with Spin did not terminate after 24 hours and was aborted. Spin also does not necessarily generate the shortest possible counter-example [10] because, unlike `SMV`, it uses depth-first search. Bharadwaj and Heitmeyer [10] report a study in which Spin generated a 25-state counter-example, whereas the shortest one was only 7 states. Interpreting such counter-examples and using them to fix SCR tables may be difficult for many users.

So, in our opinion, `SMV` is a better choice for verifying SCR models. We should note, however, that [10] reported that if a property does not hold, Spin reports a counter example much faster and using less memory than SMV. Thus, adding `SMV` verification to `SCRTool` gives the user an ability to choose the verification method based on the property to analyze and the expected outcome of the analysis.

4 Checking Correctness of Code

In principle, it is possible to generate code from the black-box requirements specified in SCR. However, it may not always be the right thing to do because SCR was specifically designed to allow specifying the system at a relatively high level of abstraction. In addition, the code is likely to implement more complex data structures than those found in SCR and may include parts that are not specified in SCR (e.g, user interfaces). Finally, it is likely that computer-generated code may be too slow and may have to be tuned for performance by hand [40]. For all these reasons, effective analysis of correctness of code w.r.t. its SCR requirements is an essential part of the formal development of software.

SC(R)^3 incorporates a tool called `cord` [16, 17] that takes specially annotated source code and an SCR specification. The code is considered correct if it contains exactly the same transitions, modulo the level of abstraction, as its SCR specification. In designing `cord`, our principles were:

- Enable relatively cheap and fully-automatic verification
- Check an abstraction of the system
- Report errors in easy-to-understand fashion
- Ensure that results of the verification can be correctly interpreted

Users annotate their code with statements indicating changes to requirements variables. `cord` uses data-flow analysis instead of exhaustive state enumeration to (conservatively) collect information about values of requirements variables, compute transitions between them, and then check these transitions against the specification. The results of the analysis are imprecise; for example, many of `cord`'s messages should be interpreted as "there *may be* a problem on this line of the program".

The verification is done in low-degree polynomial time, roughly proportional to the square of the number of lines in the program and to the sum of the number of SCR variables times the size of their domain [17]. The technique works only for a subset of SCR, which includes mode classes, and boolean and enumerated-type monitored and controlled variables.

In this section, we briefly describe the meaning of correctness w.r.t. SCR specifications, discuss the annotation language, outline the algorithms used in `cord` to check consistency between the requirements and the annotated code, and present results of verifying the implementation of the Cruise Control System. Since formal algorithms used in `cord` appear elsewhere [17, 16], we try to emphasize the *usability* of this approach.

4.1 Creating Annotated Code

`cord` can process arbitrary C programs as long as they are correctly annotated. Throughout the analysis, `cord` ignores the actual programming constructs and analyzes only the annotations and the control-flow of the program.

Code is annotated with special statements which describe changes in the system state. These changes involve local (rather than invariant) information and therefore are relatively easy to specify. The annotations, described in detail in [17], are of four types: An *Update annotation* assigns values to variables, identifying points at which the program changes its state. An *Assert annotation* reflects a programmer's knowledge that variables have particular values in the current state. Our analysis usually gives imprecise results because we utilize aggressive state folding. Assert annotations reduce the amount of information in the state to what the programmer *believes* to be true. A *Read annotation* indicates that a variable has been assigned some input value. Given that all values are equally possible, the semantics of Read is that the variable receives the value corresponding to the union of all values of its type (denoted $\top$).

Consider the following annotated code fragment, taken from the implementation of the Cruise Control system:[7]

```
if (event == BRAKE) {
  if (!vBrake) {
    @@ Assert ~Brake;
    vBrake = true;
    @@ Update Brake;
    deltaS -= ACCEL_BRAKE;
```

```
if (vCC == mCruise) {
  @@ Assert CC=Cruise;
  vCC = mOverride;
  @@ Update CC=Override;
  CruiseButton = vOff;
@@ Update b_Off & ~b_Cruise & ~b_Resume;
}
```

...

In this fragment, code variables `vBrake` and `vCC` correspond to requirements variables `Brake` and `CC`, respectively. A code variable `CruiseButton` implements the state of the Cruise button, and the implementor chose it to be an enumerated type, with values `vOff`, `vCruise`, `vResume` and `vNone`. This variable corresponds to three requirements variables `b_Off`, `b_Cruise` and `b_Resume`. If the brake pedal of the car is not pressed, the `BRAKE` event will reduce the speed of the car by `ACCEL_BRAKE`. The test and the assignment to `vBrake` are followed by an Assert and an Update annotation, respectively. The second If statement specifies that if the system is in mode Cruise, it should transition into mode `Override` and disable the cruise control.

In our experience, implementations of SCR requirements of reactive systems are quite stylized: an Initial annotation marks the starting state and is followed by a loop in which the system reads relevant monitors, tests their values, and then decides to change either its mode or the values of some controlled variables. Read annotations often follow reading of the monitors; Assert annotations are used to document the results of testing predicates; and Update annotations mark changes in the program's state.

As stated above, `cord` uses annotations and control-flow information of the code to check it for correctness. That is, "the analysis is as good as the annotations". If the code is annotated carefully, correctness of annotations provides some assurance about the correctness of the source code. However, in order to suppress diagnostic messages from our analysis, a programmer may

[7] Lines that start with @@ indicate annotations.

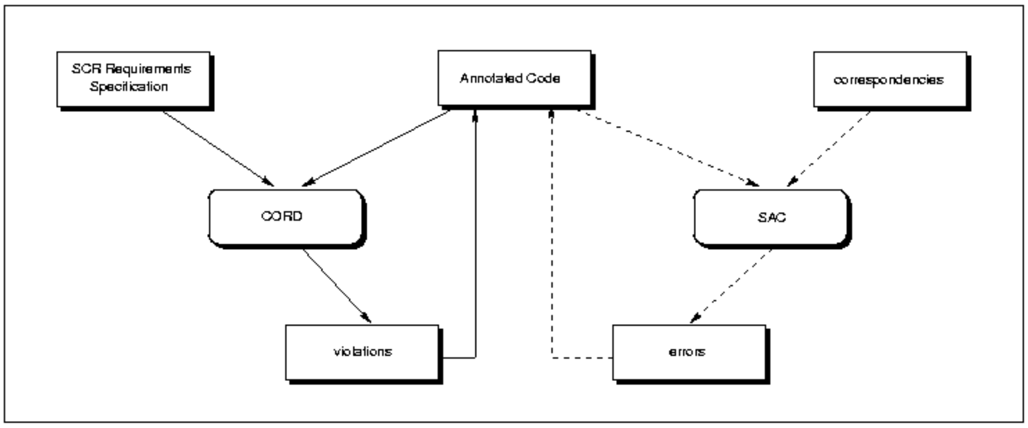

Figure 3: Code analysis with SC(R)^3.

add or change annotations without making corresponding changes in the source code. Annotations and source code may also "diverge" as modifications to the program are being made: it is possible to modify the annotations to get `cord` to give the correct answer, forgetting to update the actual source code. The SC(R)^3 toolset includes a tool called `sac` [18] designed to check that the code is annotated correctly. The entire process of the code analysis performed by SC(R)^3 is depicted in Figure 3. Here, tools (`sac` and `cord`) and artifacts are represented by ellipses and boxes, respectively. To use `sac`, the programmer creates a list of *correspondences* between variables in the requirements and the code, e.g. Figure 4 shows correspondences between the code and requirements variables of the Cruise Control System. Correspondences can be one-to-one (one code variable corresponds to one requirements variable, e.g. {`vBrake`} = {`Brake`}), one-to-many (one code variable corresponds to several requirements variables, e.g. {`CruiseButton`} = {`b_Cruise`, `b_Resume`, `b_Off`}), and many-to-one. `sac` goes through the annotated code, checking the following four conditions (we call code variables that correspond to some requirements variables *relevant*):

1. Every Assert annotation corresponds to a test of the appropriate source code variable(s).
2. Every Update/Read annotation corresponds to an assignment to the appropriate source code variable(s).
3. Every assignment to a relevant source code variable corresponds to an Update/Read annotation with the appropriate annotation variable(s).
4. Every test of a relevant source code variable corresponds to an Assert annotation with the appropriate annotation variable(s).

The meaning of the correspondences is: if *all* of the code variables in a correspondence change, all requirements variables should change, and vice versa. If only *some* of the variables change, `sac` issues warnings. Results of running `sac` on the code of the Cruise Control System is given in Figure 4. Running `sac` uncovered a number of errors: we forgot to test for a value of Throttle

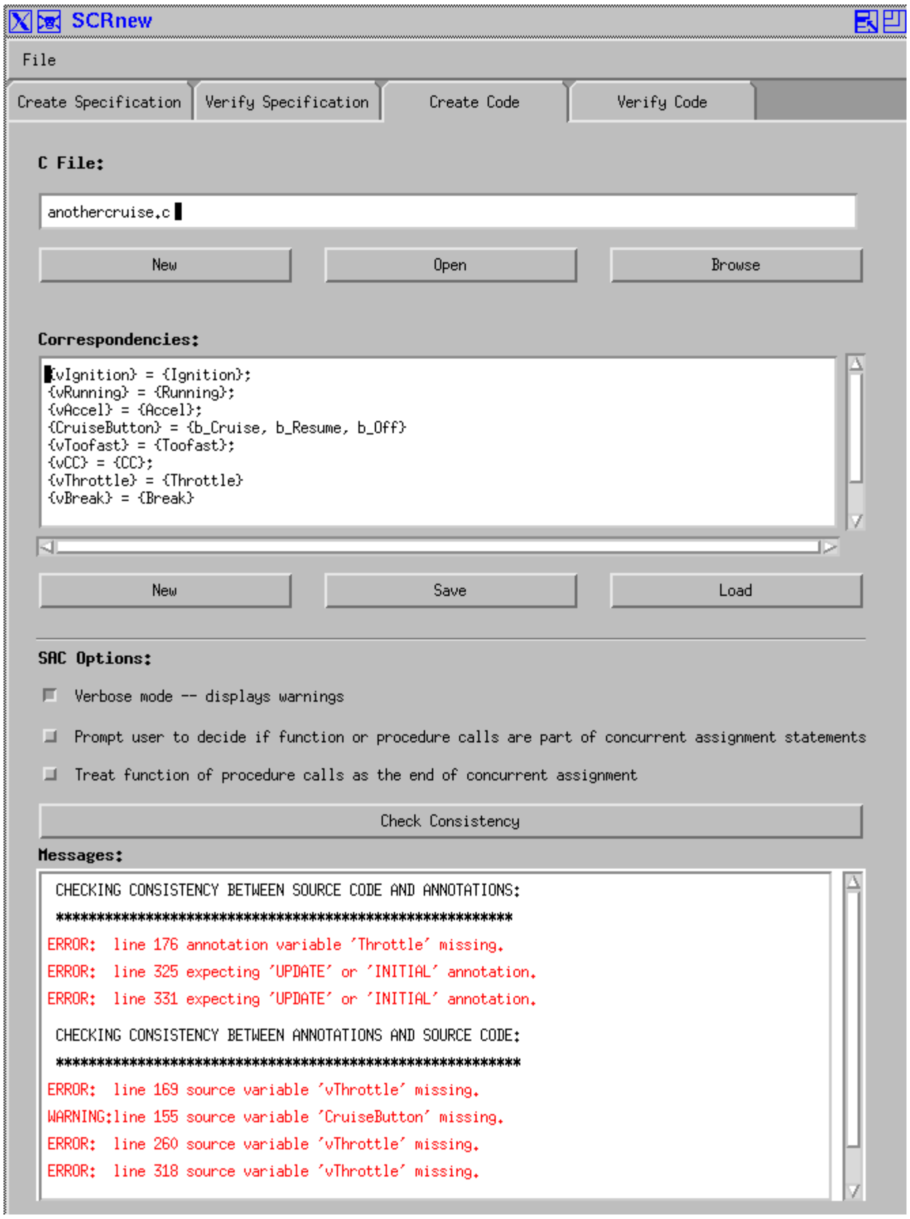

Figure 4: Checking annotations with SC(R)^3.

in the code, failed to include information about a changed variable into an annotation, in two places switched the relative order of annotations and source code (`sac` expects the annotation to follow the source code line), and used an Update instead of an Assert in one annotation.

The closer the relationship between annotation and source code variables is to one-to-one, the less spurious warnings `sac` gives. Note that `sac` does not check if variables are assigned correct *values*, so it will not complain if we follow a line `vCC=mCruise;` with `@@ Update CC=Off;` since it only checks for the assignment to the appropriate *variable*.

4.2 Correctness Criteria

A program artifact $\mathcal{A}$ is consistent with its SCR requirements $\Sigma = (S, S_0, E^m, T)$ if

1. $\mathcal{A}$ and Σ have the same starting state S_0;
2. $\mathcal{A}$ does not implement any state transitions which are not specified by the transform T; and
3. $\mathcal{A}$ implements all state transitions specified by T.

where a *state change* in the code occurs if one of requirements variables changes its value.

This is a very restricted definition. Typically, artifacts are considered consistent with requirements when they implement *at least* what is specified. Since SCR was developed to specify high-assurance systems, its models are intended to capture *all* allowed system behaviors. Indeed, it is clearly a fault if an artifact implements an unspecified transition to a state representing a failure.

Properties of types (2) and (3) are called OLT ("only legal transitions") and ALT ("all legal transitions"), respectively. For example, one of the ALT properties of the Cruise Control System is **[Alt1]**:

> A transition from mode `Cruise` to mode `Off` on event `@F(Ignition)` exists in the code.

This transition refers to the fourth row of Table 2. An ALT property is generated from every row of mode transition tables and every entry of event tables.

One of the OLT properties is **[Olt1]**:

> The only transitions into mode `Off` are from mode `Inactive` on event `@F(Ignition)`, or from mode `Cruise` on event `@F(Ignition)`, or from mode `Override` on event `@F(Ignition)`.

An OLT property is generated for every value of controlled variables and every mode class in the right column of mode transition tables.

The properties are transformed into a list of first-order logic formulas [17]. For a state n and a formula f, $n \models f$ indicates that f is true in n, and $\mathrm{succ}(n)$ returns the list of successor states. Then properties **[Alt1]** and **[Olt1]** are expressed as

$$\begin{aligned}
Alt_1 &= \exists n, \exists s \in \mathrm{succ}(n), (s \models (\mathtt{CC} = \mathtt{Off})) \ \wedge\ (n \models ((\mathtt{CC} = \mathtt{Cruise}) \ \wedge\ \mathtt{@F(Ignition)})) \\
Olt_1 &= \forall n, \forall s \in \mathrm{succ}(n), (s \models (\mathtt{CC} = \mathtt{Off})) \rightarrow ((n \models (\mathtt{CC} = \mathtt{Off})) \vee \\
&\qquad (n \models ((\mathtt{CC} = \mathtt{Inactive}) \ \wedge\ \mathtt{@F(Ignition)})) \vee \\
&\qquad (n \models ((\mathtt{CC} = \mathtt{Cruise}) \ \wedge\ \mathtt{@F(Ignition)})) \vee \\
&\qquad (n \models ((\mathtt{CC} = \mathtt{Override}) \ \wedge\ \mathtt{@F(Ignition)})))
\end{aligned}$$

Olt_1 signifies that for the system to be in mode `Off` in a given state, each of its predecessors should be either already in mode `Off`, or in mode `Inactive` when `@F(Ignition)` occurs, or in mode `Cruise` when `@F(Ignition)` occurs, or in mode `Override` when `@F(Ignition)` occurs.

The total number of properties is proportional to the combined number of rows in SCR tables. To prove that an artifact is consistent with its requirements, we demonstrate that it is a model of *all* these properties.

4.3 Analysis

Analysis done by `cord` is described in detail in [17]; here we give a quick summary of this process.

Verification is done via static analysis of the annotations and the control-flow information of the code. `cord` uses a technique similar to *constant propagation* to compute a set of values that every requirements variable can attain at each point in the program. More formally, a state at node n of the code implementing an SCR specification Σ is a set of variable-value pairs $\{(r_i, v_i) \mid r_i \in RF\}$ where $v_i \in 2^{TY(r_i)}$ (a set consisting of values in the domain of r_i). In each iteration of the algorithm, we propagate the information *generated* in nodes containing Update and Read annotations and the information *known* in nodes containing Assert annotations throughout the control-flow graph (CFG) of the program, until a fixpoint is achieved. If V is the number of nodes in the CFG (V equals to the number of annotations and control statements of the code), then the fixpoint can be achieved in at most

$$O(|V|^2 \times (\Sigma_{r_i \in RF}|TY(r_i)| + 1))$$

steps. This method also allows to identify *unreachable* states – states that did not change throughout constant propagation.

In constructing the abstraction of the system, we used state-folding to minimize the number of states. We had an option of using model-checking to check the resulting model against consistency properties. However, model checkers process states whose values have scalar values. Transforming our model into an acceptable input for an existing model-checker would have resulted in an exponential increase in the number of nodes. Instead, we recognized that, like SCR assertions, consistency properties involve short 1-3 state fragments of paths through the code, thus enabling very efficient checking. However, since we compute more behaviors than can be present in the code, we cannot claim that our verification is sound and complete, i.e. a property is violated if and only if we find a violation. Instead, we check properties *optimistically* or *pessimistically.*

A property is checked pessimistically if all of its violations in the code are detected; a property is checked optimistically if all detected violations are present in the code. Pessimistic and optimistic analysis are duals: if a property P is verified pessimistically, then $\neg P$ is verified optimistically. Under the *conservative* analysis, i.e., analysis that overestimates the behaviors present in the code, universally-quantified properties are typically verified pessimistically (since some of the paths may be infeasible in practice), and existentially-quantified properties are verified optimistically.

`cord` verifies OLT properties pessimistically and ALT properties optimistically. In addition, it verifies invariant (pessimistically) and reachability (optimistically) properties and checks that environmental assumptions are not violated in the code.

Once the constant propagation algorithm terminates, ALT and OLT properties are checked in a single traversal of the graph. The verification starts with computing transitions that can occur at each node in the CFG. Events are computed by comparing the current and the predecessor nodes. `cord` assumes that the predecessor node contains previous values of modeclass variables, WHEN conditions, and old values of triggering conditions, whereas the current node contains new values of mode class or controlled variables and new values of triggering conditions. The maximum number of transitions generated for each node is

$$Trans_v = O(|V| \times \Sigma_{r_i \in RF}(|TY(r_i)|)^2)$$

although in practice this number is significantly smaller.

For each node of the graph, `cord` uses computed transitions to check every OLT property and report violations as they are discovered. Since a number of transitions are computed for a single node, we can report several OLT violations for a given line of the code. If $Trans \leq Trans_v \times |V|$ is the total number of computed transitions, P_{olt} – the set of OLT properties, and D – the maximum number of disjuncts in P_{olt} (for example, the number of disjuncts in Olt_1 from Section 4.2 is four), then the running time of the algorithm for checking OLT properties is

$$O(|P_{olt}| \times D \times \Sigma_{r_i \in RF}|TY(r_i)| \times Trans)$$

Since ALT properties indicate a presence of a given transition in the graph, `cord` marks the ALT properties that correspond to the computed transitions. At the end of the traversal unmarked properties are reported as violations. The running time of this algorithm is

$$O(|P_{alt}| \times \Sigma_{r_i \in RF}|TY(r_i)| \times Trans)$$

where P_{alt} is the set of ALT properties.

The resulting algorithms are extremely effective in discovering errors (see Section 4.4). However, it seems that if we strive for strictly optimistic or strictly pessimistic verification, we would have trouble comparing the WHEN conditions computed from the code with the WHEN conditions of SCR tables. In fact, transitions in SCR tables seldom involve a large number of variables, with variables not explicitly listed in the condition considered "don't care"s. If a variable r_i is "don't care", it is treated as having *any* of its values ($TY(r_i)$) for the pessimistic verification of properties, and *neither* of its values ($\perp$) for the optimistic verification. Thus, SCR's "don't care" conditions turned out to be ideal for set-based approximation used as the basis for `cord`; a similar technique may not have worked this well for a different specification language.

4.4 Checking the Cruise Control System

We have implemented the Cruise Control system in C with an Xlib interface. The implementation was not originally annotated and consisted of 675 lines of code, out of which roughly 380 lines were used to implement the GUI of the system. The code was then annotated by another member of the group in about 40 minutes; the annotation effort was trivial and resulted in 34 Update, 28 Assert, 1 Read and 1 Initial annotation.

After fixing annotation errors reported by `sac`, we ran `cord` on the code of the Cruise Control system. The analysis took 7.62 seconds (3.45 user, 0.74 system, as reported by tshell's `time` command) on a moderately loaded SPARCstation-4 (85 MHz processor) with 64 megabytes of RAM, and resulted in reporting 5 ALT and 5 OLT violations (see Figure 5). The ALT and the OLT violations are shown in Figures 6 and 7, respectively.

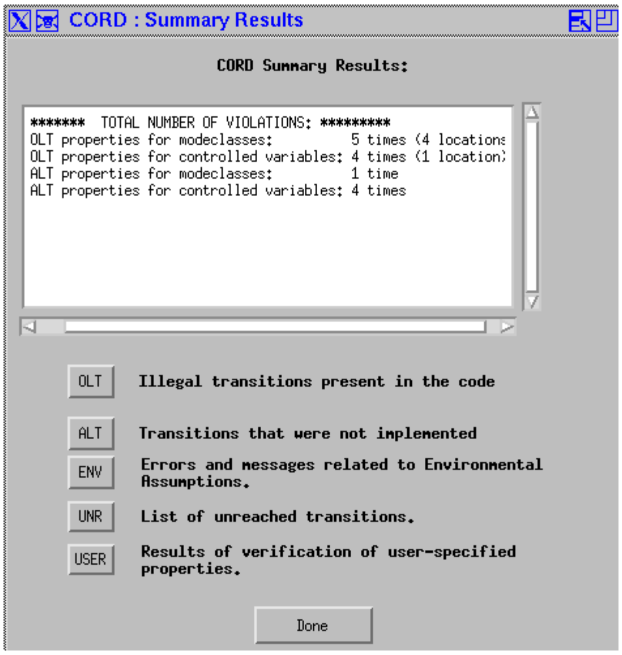

Figure 5: Summary of errors.

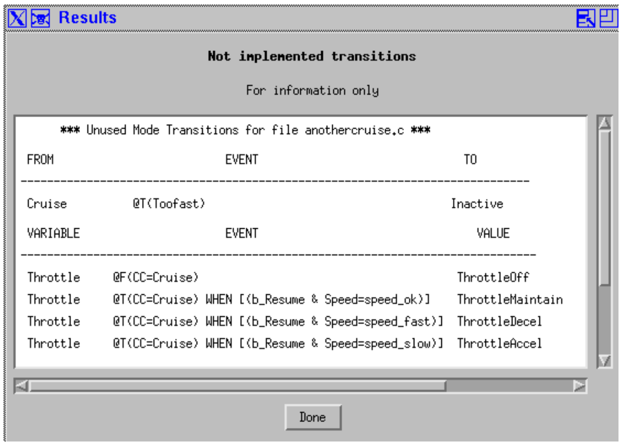

Figure 6: ALT errors.

One OLT and one ALT violation were caused by an incorrect annotation overlooked by `sac`. As we mentioned earlier, `sac` does not consider values of variables when checking the code for correct annotations, and did not report a violation when we annotated an assignment with `Accel` instead of `~Accel`. Another pair of OLT and ALT violations came from an incorrect transition in the code (line 388) – to mode `Override` instead of `Cruise`. In addition, `cord` concluded that the system can be in all possible modes before this line, whereas only `Cruise` was possible. This

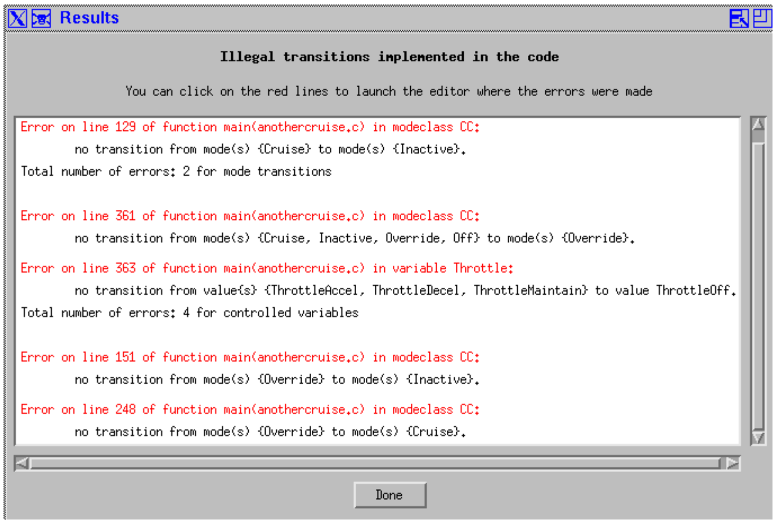

Figure 7: OLT errors.

problem arises because of the imprecise analysis used in `cord` and can be fixed by adding extra Assert annotations. A yet another pair of OLT and ALT violations came from a transition from `Override` to `Cruise`. The code did not check that this transition should be enabled only when `Toofast`, `Brake` and `Accel` are false. Out of the remaining two OLT violations, one was a false negative and the second was an incorrect triggering event for the variable `Throttle`. All other ALT properties came from transitions that were not implemented in the code.

5 Related Work

SC(R)^3 allows to reuse software requirements by combining several integrated analysis activities in a single toolset. The work reported in this paper is most closely related to Atlee and her colleagues [6, 7, 77] and to Heitmeyer, Bharadwaj and others [10, 41, 42] as described in previous sections. In this section we briefly describe several tools based on languages other than SCR, discuss attempts to analyze software requirements, and compare analysis capabilities of `cord` with those of similar static analysis systems.

5.1 Tools

Several tools include formal graphical modeling languages and combine multiple analysis activities.

STATEMATE [32, 35] allows to model reactive systems using StateCharts [33, 34] for describing system behavior, and the hierarchical language of Activity-charts for describing the system's activities and the data that flows between them. STATEMATE includes a variety of tools, including consistency analysis and simulation of the model, documentation generation, and code generation into C, Ada, VHDL and Verilog, and has been used in a number of avionics and telecommunications companies. STATEMATE does not include model checking capabilities, although efforts to analyze StateCharts have been made by other researchers [11].

RSML [60] is a variant of StateCharts, with somewhat different semantics and augmented with AND/OR diagrams. RSML has been used for specifying the Traffic Collision Avoidance System (TCAS II) [37]. RSML requirements can be analyzed for consistency using model-checking [15, 3] and theorem-proving [66, 38], and can be used for code generation [40].

In telecommunications, the standardized Specification Description Language SDL [53] is supported by several industrial-quality tools. For example, Telelogic's SDT [79] allows to amortize the investment into SDL specifications through syntax and semantics checking, symbolic debugging that checks if a certain behavior captured as a message-sequence chart (MSC) [54] holds in the model, generates C code, and includes a test derivation facility that allows to capture test cases in the form of MSCs. SDT can also perform limited verification, symbolically executing a model up to a certain depth, looking for deadlocks and synchronization violations. Although not a validation tool, SDT's verifier is very useful for debugging. The easy-to-use graphical notation of SDL and MSCs and the strong integration between the tools enable the effective use of this technology in many telecommunications companies around the world.

5.2 Model-checking and Theorem Proving

As we mentioned above, model-checking has been used to verify other state-machine-based models. In particular, since StateCharts has a more complex formal model than SCR, allowing for multiple simultaneous events and possible zero-time loops (infinite sequences of events that occur while the time does not elapse), verification of StateCharts is much more complex. In particular, the technique of variable abstraction that we rely on heavily, cannot be used because the variables do not form a partial order of dependencies. As the result, effective model-checking of StateCharts requires significant modifications to `SMV` to enable efficient BDD operations on integer variables [15].

Bultan and others [14, 13] tried to verify unabstracted SCR specifications using integer constraint sets. In this approach, SCR tables are represented as systems of affine constraints which are symbolically manipulated using BDDs and the Omega Library [75]. Although we believe that this technique have a lot of potential, its application to the verification of realistic systems is still infeasible.

Outside the abovementioned work, model-checking works only for finite-state systems which either limits its availability or creates the need for abstraction. An alternative technique is theorem-proving which represents specifications as a set of logical axioms and attempts to show that the desired property can be derived from these axioms. Although much more powerful, theorem-proving suffers from requiring a lot of time and expertise to apply, and from not generating counter-examples. The use of theorem-proving in analyzing SCR tables has been proposed in [68]. [4] discussed approaches of specifying proof templates to allow non-expert users to effectively reason about the correctness of SCR specifications.

5.3 Code Analysis

Here we only list the systems that use abstraction and state exploration to check properties.

LCLint [27, 26] performs static checks on C code using LCL specification [30]. It is able to detect violations of abstraction boundaries, undocumented uses of global variables, incorrect dependencies between variables, and detect a broad class of pointer-related errors, e.g. uses of dead storage, memory leaks, etc. Unlike `cord`, LCLint was designed to provide "useful" information, so some of the checks are neither sound nor complete.

Quick Defect Analysis (QDA) [51] also uses a simple annotation language. Annotations called hypotheses are embedded in comments to describe properties that objects should have at particular program points. Hypotheses can also include event sequences. Other comments contain assertions about properties of objects. Like `cord`, QDA can be applied to existing code if the code is annotated. However, our approach is more formal, taking advantage of complete system specifications and being able to interpret the results as strictly optimistic or strictly pessimistic.

Aspect's [55] specification notation permits users to write pre- and postconditions about the data dependencies of an operation. Dataflow analysis is used to compute an upper bound on the data dependencies of the implementation. If an asserted dependency is missing, an error is reported.

Clarke et al. [19] also create abstract, finite state models of programs, and use model checking techniques to verify formulas. Programs written in a special finite-state programming language are translated into relational expressions characterizing the program's initial state and transition relation. To reduce the size of the model, users define mappings of implementation values to abstract values and symbolically execute operations on the values. The model checking approach is pessimistic for formulas expressed in $\forall$CTL* [29], a subset of CTL in which only universal path quantification is allowed. The authors also identify a large class of temporal formulas for which the verification results are exact, i.e., formulas hold in the model iff they hold in the original program.

Recently, several techniques to analyze concurrency properties of Ada and Java programs have been developed. In particular, INCA tools [9, 73] translate Ada into Promela, and JCAT [52] and Java Path Finder [36] translate Java into Promela, for later verification by Spin. Unlike the others, Java Path Finder can handle a much larger portion of Java, including exceptions and inheritance. None of these tools provide any support for abstracting control and data states of the program.

The methods outlined above do not assume existence of *complete* specifications, and thus can be applied to a variety of different systems. Our method is much more SCR-specific and requires that complete SCR specifications be present. However, it does guarantee that the code implements exactly the same transitions as specified. A number of approaches are similar to ours in nature, including equivalence checking in process algebras, e.g. as implemented in the Concurrency Workbench [22]. This technique checks that two levels of specifications (or a specification and an implementation) exhibit exactly the same behavior. COSPAN [59] uses L-automata to check two levels of specifications (or a specification and an implementation) for language containment properties. Both approaches have been used to formally verify communication protocols and circuit designs, as well as other hardware and software systems.

6 Conclusion and Future Work

In this paper we argued that formal methods can be used effectively if the cost of their creation is amortized across several analysis activities and presented a tool called SC(R)^3 that incorporates several such activities. Through a tight integration with NRL-developed `SCRTool`, SC(R)^3 allows to create SCR specifications and check their internal consistency. In addition, SC(R)^3 augments `SCRTool`'s verification capabilities by encapsulating the symbolic model-checker `SMV`. Our techniques allow to model the environment correctly and, using the algorithm presented in the paper, to automatically create very compact models that are analyzed by `SMV` quickly and effectively. If a property fails, SC(R)^3 captures counter-examples generated by `SMV` and displays them in the SCR format. The paper also discusses cases for which each of the requirements verification activities is preferred. After requirements have been finalized and implemented, SC(R)^3 allows to check consistency between annotated code and requirements. Since the state-space of programs can be very large, the verification is done instead via data-flow analysis, resulting in imprecise but fast (polynomial in the number of lines in the program and the number of variables in the specification) analysis. The first version of SC(R)^3 is available for distribution at `http://www.cs.toronto.edu/~chechik/SCR3/scr3.html`.

In Section 1 we outlined a methodology for using the SC(R)^3 toolset throughout the software development cycle. However, a number of improvements to SC(R)^3 need to be made before this methodology is realizable:

- Verification of requirements should be better automated. In particular, this refers to being able to automatically abstract integer-domain variables into predicates and generate criteria guaranteeing that the abstraction is complete.
- Current requirements verification is limited to verifying mostly control, as opposed to timing or data, aspects of the specifications, although other researchers have started to reason about real-time in the domain of SCR [8, 43]. We need to be able to deal with real-time systems and to extend the verification technology to reasoning about data structures.
- Both `cord` and `sac` need a number of improvements to be able to analyze code against more complex SCR constructs. Such improvements should include dealing with SCR terms and conditions and, more importantly, reasoning about infinite-domain variables. In fact, a new generation of `cord` that allows direct reasoning about program variables, without using annotations, is currently being developed. This version of the tool will be able to approximate infinite types and perform arithmetic operations using abstract interpretation [23].
- We have developed [78] preliminary techniques for creating test cases from SCR specifications. However, we have neither automated them nor evaluated their effectiveness. More work in this area is planned.
- We need to create a tool that would generate provably-correct portions of code from SCR specifications. Given that the specifications are black-box, the complexity lies in being able to generate *usable* code by letting the designer make the necessary design decisions.

We hope that the future generation of SCR tools will become the de facto standard in projects that deal with event-driven systems, safety-critical or not.

Acknowledgments

Joanne Atlee and her students developed `gensmv` — the tool that was the prototype of `SSL2SMV`, and performed the initial analysis of the Cruise Control System. We are very grateful to John Gannon and Rich Gerber for teaching us about formal methods and to Connie Heitmeyer and Stuart Faulk for many fruitful discussions about SCR. The `SCR*` tool used as part of SC(R)^{3}is courtesy of Connie Heitmeyer's group at Naval Research Lab. Many people have participated in this project; in particularly, we would like to acknowledge the help of David Wendland, Hicham Mouline and Serene Wong for implementing and improving many aspects of the tools, to Sai Sudha for implementing `sac`, and Matthew Cwirko-Godycki and Virginie Lim for their help with testing the tool and performing the case study.

We are grateful to Joanne Atlee for providing comments on the previous version of this paper.

References

[1] Ruth Abraham. "Evaluating Generalized Tabular Expressions in Software Documentation". CRL Technical Report 267, McMaster University, February 1997.

[2] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. "Software Requirements for the A-7E Aircraft". Technical report, Naval Research Laboratory, March 1988.

[3] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon Reese. "Model Checking Large Software Specifications". In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE'96)*, October 1996.

[4] M. Archer, C. Heitmeyer, and S. Sims. "TAME: A PVS Interface to Simplify Proofs for Automata Models". In *Proceedings of User Interfaces for Theorem Provers*, July 1998.

[5] J. Atlee. *"Automated Analysis of Software Requirements"*. PhD thesis, University of Maryland, College Park, Maryland, December 1992.

[6] J.M. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, pages 22–40, January 1993.

[7] Joanne M. Atlee and Michael A. Buckley. "A Logic-Model Semantics for SCR Software Requirements". In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, January 1996.

[8] Joanne M. Atlee and John Gannon. "Analyzing Timing Requirements". In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 117–127, Cambridge, MA, June 1993.

[9] G.S. Avrunin, U.A. Buy, J.C. Corbett, L.K. Dillon, and J.C. Wielden. "Automated Analysis of Concurrent Systems with the Constrained Expression Toolset". *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[10] Ramesh Bharadwaj and Connie Heitmeyer. "Model Checking Complete Requirements Specifications Using Abstraction". *Journal of Automated Software Engineering*, 6(1), January 1999.

[11] Udo Brockmeyer and Gunnar Wittich. "Tamagotchis Need Not Die – Verification of STATEMATE Designs". In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98), LNCS 1384*. Springer Verlag, 1998.

[12] Randal E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams". *Computing Surveys*, 24(3):293–318, September 1992.

[13] Tevfik Bultan, Richard Gerber, and Christopher League. "composite model checking: Verification with type-specific symbolic representations.". Technical Report TRCS99-02, Department of Computer Science, University of California, Santa Barbara, January 1999.

[14] Tevfik Bultan, Richard Gerber, and William Pugh. "Model Checking Concurrent Systems with Unbounded Integer Variables: Symbolic Representations, Approximations and Experimental Results.". *ACM Transactions on Programming Languages and Systems*, 1999.

[15] William Chan, Richard J. Anderson, Paul Beame, David H. Jones, David Notkin, and William E. Warner. "Decoupling Synchronization from Local Control for Efficient Symbolic Model Checking of StateCharts". In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 142–151, May 1999.

[16] M. Chechik. *"Automatic Analysis of Consistency between Requirements and Designs"*. PhD thesis, University of Maryland, College Park, Maryland, December 1996.

[17] M. Chechik and J. Gannon. "Automatic Analysis of Consistency Between Requirements and Designs", May 1998. (accepted for publication in IEEE Transactions on Software Engineering).

[18] M. Chechik and V.S. Sudha. "Checking Consistency between Source Code and Annotations". CSRG Technical Report 373, Department of Computer Science, University of Toronto, 1998.

[19] Edmund M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". *IEEE Transactions on Programming Languages and Systems*, 19(2), 1994.

[20] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[21] E.M. Clarke and J. Wing. "Formal Methods: State of the Art and Future Directions". *ACM Computing Surveys*, 28(4):626–643, December 1996.

[22] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems". *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[23] Patrick Cousot and Radhia Cousot. "Static Determination of Dynamic Properties of Generalized Type Unions". *SIGPLAN Notices*, 12(3), March 1977.

[24] Steve Easterbrook and John Callahan. "Formal Methods for Verification and Validation of Partial Specifications: A Case Study". *Journal of Systems and Software*, 40(3), 1998.

[25] Steve Easterbrook, Robyn Lutz, Richard Covington, John Kelly, Yoko Ampo, and David Hamilton. "Experience Using Lightweight Formal Methods for Requirements Modeling". *IEEE Transactions on Software Engineering*, 24(1):4–14, January 1998.

[26] David Evans. "Static Detection of Dynamic Memory Errors". In *Proceedings of PLDI'96: SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

[27] David Evans, John Guttag, James Horning, and Yang Meng Tan. "LCLint: A Tool for Using Specifications to Check Code". In *Proceedings of FSE'94: Foundations of Software Engineering*, 1994.

[28] S. Faulk, L. Finneran, J. Kirby, S. Shah, and J. Sutton. "Experience Applying the CoRE Method to the Lockheed C-130J". In *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS'94)*, pages 3–8, June 1994.

[29] O. Grumberg and D.E. Long. "Model Checking and Modular Verification". In *Proceedings of CONCUR'91: 2nd International Conference on Concurrency Theory*, 1991.

[30] J.V. Guttag and J.J. Horning. "Introduction to LCL, A Larch/C Interface Language". Research Report 74, SCR, 1991.

[31] Anthony Hall. "Seven Myths of Formal Methods". *IEEE Software*, pages 11–19, September 1990.

[32] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems". *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[33] David Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[34] David Harel and Amnon Naamad. "The STATEMATE Semantics of StateCharts". *ACM Transactions on Software Engineering Methodology*, 5(4), October 1996. revised version of "The Semantics of StateCharts", TR, i-Logic.

[35] David Harel and Michal Politi. *Modeling Reactive Systems With Statecharts : The Statemate Approach.* McGraw Hill, 1998.

[36] K. Havelund and T. Pressburger. "Model Checking Java Programs Using Java Pathfinder". *International Journal on Software Tools for Technology Transfer*, 1999.

[37] Mats P.E. Heimdahl. "Lessons from the Analysis of TCAS II". In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'96)*, San Diego, CA, January 1996.

[38] Mats P.E. Heimdahl and Barbara J. Czerny. "Using PVS to Analyze Hierarchical State-Based Requirements for Completeness and Consistency". In *Proceedings of the IEEE High Assurance Systems Engineering Workshop (HASE'96)*, Niagara Falls, Canada, October 1996.

[39] Mats P.E. Heimdahl and Constance L. Heitmeyer. "Formal Methods For Developing High Assurance Computer Systems: Working Group Report". In *Proceedings of the Second International Workshop on Industrial-Strength Formal Techniques (WIFT'98)*, October 1998.

[40] Mats P.E. Heimdahl and David J. Keenan. "Generating Code from Hierarchical State-Based Requirements". In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'97)*, Annapolis, Maryland, January 1997.

[41] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. "Automated Consistency Checking of Requirements Specifications". *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[42] C.L. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. "Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications". *IEEE Transactions on Software Engineering*, 24(11), November 1998.

[43] C.L. Heitmeyer, J. Kirby, and B.G. Labaw. "Tools for Formal Specification, Verification, and Validation of Requirements". In *Proceedings of the 12th Annual Conference on Computer Assurance (COMPASS'97)*, June 1997.

[44] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. "SCR*: A Toolset for Specifying and Analyzing Requirements". In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 109–122, June 1995.

[45] K. Heninger. "Software Requirements for the A-7E Aircraft". Technical Report NRL Report 3876, Naval Research Laboratory, Washington, DC, 1978.

[46] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications". *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[47] Gerard Holzmann. "A Practical Method for Verifying Event-Driven Software". In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 597–607, May 1999.

[48] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[49] G.J. Holzmann. "The Model Checker SPIN". *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[50] D. N. Hoover and Zewei Chen. "TableWise: A Decision Table Tool". In John Rushby, editor, *Proceedings of COMPASS'95*, June 1995.

[51] W.E. Howden and G.M. Shi. "Linear and Structural Event Sequence Analysis". In *Proceedings of ISSTA'96*, May 1996.

[52] R. Iosef. "A Concurrency Analysis Tool for Java Programs". Master's thesis, Polytechnic University of Turin, August 1997.

[53] ITU-T. "ITU-T Recommendation Z.100: Specification and Description Language (SDL)". *ITU-T*, 1993.

[54] ITU-T. "ITU-T Recommendation Z.120: Message Sequence Chart (MSC)". *ITU-T*, 1993.

[55] Daniel Jackson. "Aspect: Detecting Bugs with Abstract Dependences". *Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[56] James Kirby. "Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System". Technical report, Wang Institute of Graduate Studies, October 1988. Revision 2.0 by Skip Osborne and Aaron Temin.

[57] James Kirby. "SCR* Tooset: The User Guide, 1997.

[58] R. P. Kurshan. "COSPAN". In *Proceedings of CAV'96*, 1996.

[59] R.P. Kurshan. *Computer-Aided Verification of Coordinating Processes : The Automata-Theoretic Approach.* Princeton Series in Computer Science. Princeton University Press, 1995.

[60] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese. "Requirements Specification for Process-Control Systems". *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[61] R.R. Lutz and H.-Y. Shaw. "Applying the SCR* Requirements Toolset to DS-1 Fault Protection". Technical Report JPL-D15198, Jet Propulsion Laboratory, Pasadena, CA, December 1997.

[62] Z. Manna and A. Pnueli. "Tools and Rules for the Practicing Verifier". Technical Report STAN-CS-90-1321, Department of Computer Science, Stanford University, 1990. Appeared in *Carnegie Mellon Computer Science: A 25 year Commemorative*.

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

[64] S.P. Miller. "Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR". In *Proceedings of the 2nd ACM Workshop on Formal Methods in Software Practice (FMSP'98)*, 1998.

[65] S.P. Miller and M. Strivas. "Formal Verification of the AAMP5 Microprocessor". In *Proceedings of the International Workshop on Industrial Strength Formal Techniques*, pages 2–17, 1995.

[66] M.P.E.Heimdahl and N.G.Leveson. "Completeness and Consistency in Hierarchical State-Based Requirements". *IEEE Transactions on Software Engineering*, SE-22(6):363–377, June 1996.

[67] J. Ostroff. "A Visual Toolset for the Design of Real-Time Discrete Event Systems". *IEEE Transactions on Control Systems Technology*, May 1997.

[68] Sam Owre, John Rushby, and Natarajan Shankar. "Integration in PVS: Tables, Types, and Model Checking". In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97), LNCS 1217*, pages 366–383, April 1997.

[69] D. Parnas and J. Madey. "Functional Documentation for Computer Systems Engineering(Version 2)". Technical Report CRL Report 237, McMaster University, Department of Electrical and Computer Engineering, 1991.

[70] David Lorge Parnas and Jan Madey. "Functional Documents for Computer Systems". *Science of Computer Programming*, 25:41–61, 1995.

[71] D.L. Parnas. "Some Theorems We Should Prove". In *Proceedings of 1993 International Conference on HOL Theorem Proving and Its Applications*, Vancouver, BC, August 1993.

[72] D.L. Parnas, J. Madey, and M. Iglewski. "Precise Documentation of Well-Structured Programs". *IEEE Transactions on Software Engineering*, 20(12):948–976, December 1994.

[73] Corina S. Pasareanu, Mathew B. Dwyer, and Michael Huth. "Assume-Guarantee Model Checking of Software: A Comparative Case Study". In *Proceedings of the 6th International SPIN Workshop (SPIN'6)*, September 1999. (to appear).

[74] Dennis K. Peters and David Lorge Parnas. "Using Test Oracles Generated from Program Documentation". *IEEE Transactions on Software Engineering*, 24(3):161–173, March 1998.

[75] William Pugh. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis". *Comm. of the ACM*, August 1992.

[76] S.Faulk, J. Brackett, P. Ward, and J. Kirby. "The CoRE Method for Real-Time Requirements". *IEEE Software*, 9(5):22–33, September 1992.

[77] Tirumale Sreemani and Joanne M. Atlee. "Feasibility of Model Checking Software Requirements: A Case Study". In *Proceedings of COMPASS'96*, Gaithersburg, Maryland, June 1996.

[78] D. Suydam and M. Chechik. "Generating Test Cases from SCR Specifications". Csrg technical report, Department of Computer Science, University of Toronto, 1997.

[79] Telelogic. "Telelogic SDT Home Page". `http://www.telelogic.com/solution/tools/sdt.asp`, September 1998.

[80] M. Weiser. "Program Slicing". *IEEE Transactions on Software Engineering*, SE-10(4), 1984.

[81] Jeannette Wing. "A Specifier's Introduction to Formal Methods". *IEEE Computer*, 23(9):8–23, September 1990.

[82] Andre Wong and Marsha Chechik. "Formal Modeling in a Commercial Setting: A Case Study". In *Proceedings of FM'99: World Congress in Formal Methods*, September 1999.