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

Marsha Chechik

University of Toronto
Department of Computer Science

Abstract

This paper gives an overview of $SC(R)^3$ – a toolset designed to increase the usability of formal methods for software development. Formal requirements are specified in $SC(R)^3$ in an easy to use and review format, 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 using mathematically-precise ("formal") specifications in software projects [21]. 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 [9]. Although case-studies (e.g. [7]) have shown applicability and effectiveness of formal methods for various 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. [13, 16]). However, to facilitate a wide-spread use of formal methods, another, complimentary approach is necessary: to improve *usability* of the methods and the tools and to demonstrate the cost-effectiveness of applying them to software systems. We believe that the way to make formal methods more usable is by

- amortizing the cost of creating formal documentation throughout the software lifecycle, i.e. using this documentation for checking correctness of programs, generating test cases and test environments, etc.;
- using (or developing) easy to read and review notations (e.g., state-machines and tables);
- decreasing analysis cost through automation; and
- adopting existing technologies wherever possible.

We are interested in specifying and verifying event-driven systems, and chose SCR (Software

Cost Reduction) to be the requirements notation used in our project.

The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of the Software Cost Reduction project [11, 12]. 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.

Representing logical formulae using tables is a powerful way to visualize information which gained acceptance among many practitioners. Table structure makes specifications easy to write and review and allows for high-yield mechanical analysis. Tools were developed to check mode tables of SCR for correctness with respect to global properties using model-checking [24] and theorem-proving [19]. A group at the Naval Research Lab developed an industrial-quality tool called `SCR*` which allows specifying and reasoning about complex systems using SCR [10]. `SCR*` performs checks to ensure that the tables are complete and consistent. Tool-building is not limited to the SCR community. For example, David Parnas and his colleagues are working on methods and tools to document programs using tables [22, 20, 23, 1], and a group at Odyssey Research Associates are developing `Tablewise` - a tool to reason about decision tables [14]. However, none of these tools are aimed at *using* tabular requirements once they have been created.

During the past several years, we have been developing a number of tools that use SCR requirements throughout the various stages of software lifecycle, and have recently integrated them into a toolset called SC(R)^3 which stands for the SCR Requirements Reuse. SC(R)^3 allows users to specify their requirements through a visual interface, conducts simple syntactical checks, and invokes various tools to perform analysis of software artifacts. We have developed tools to check requirements for correctness and to verify consistency between annotated code and requirements, and will describe these activities using the Cruise Control system – a case study that we recently undertook.

The rest of this paper is organized as follows: Section 2 describes requirements of the Cruise Control system. Sections 3 and 4 outline techniques to analyze the consistency of the requirements and the correspondence between the code and the requirements, respectively. In Section 5 we summarize our work and outline future research directions.

2 Requirements of Cruise Control System

A Cruise Control system specified by Jim Kirby [15] 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.

Table 1 gives an overview of the different sections of an SC(R)^3 requirements specification. SC(R)^3 uses a simplified SCR method in which monitored and controlled variables have just boolean values (representing predicates on inputs and outputs), and results of intermediate computations (so called "terms") are not used. Monitored variables in our case study indicate the state of the ignition, brake, and acceleration, the buttons operating the cruise control system, and the speed of the car. Table 2 shows some monitored variables and the

[1] The value of THRESHOLD is not specified in [15]. That document suggests to use a "value chosen according to how wide a speed variation is regarded as acceptable."

Component	Description
monitored variables	quantities that influence the system behaviour
mode classes	sets of states (called modes) that partition the monitored environment's state space
controlled variables	quantities that the system regulates
assumptions	assumed properties of the environment
goals	properties that are required to hold in the system

Table 1: Components of a requirements specification in the SC(R)^3 notation.

Current Mode	`Ignition`	`Running`	`Toofast`	`Brake`	`Accel`	`b_Cruise`	`b_Resume`	`b_Off`	New Mode
`Off`	@T	–	–	–	–	–	–	–	`Inactive`
`Inactive`	@F	–	–	–	–	–	–	–	`Off`
	–	t	f	f	f	@T	–	–	`Cruise`
`Cruise`	@F	–	–	–	–	–	–	–	`Off`
	–	–	@T	–	–	–	–	–	`Inactive`
	–	@F	–	–	–	–	–	–	`Inactive`
	–	–	–	@T	–	–	–	–	`Override`
	–	–	–	–	@T	–	–	–	`Override`
	–	–	–	–	–	–	–	@T	`Override`
`Override`	@F	–	–	–	–	–	–	–	`Off`
	–	@F	–	–	–	–	–	–	`Inactive`
	–	–	f	f	f	–	@T	–	`Cruise`
	–	–	f	f	f	@T	–	–	`Cruise`

Initial Mode: `Off` (~`Ignition`)

Table 4: Mode transition table for mode class `CC` of the cruise control system.

predicates they represent.

The Cruise Control system has one mode class `CC`, whose modes are described in Table 3. The system is in exactly one mode of each modeclass at all times, so we can think of modeclasses as finite-state machines. The mode transition table of mode class `CC` is shown in Table 4. 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*, i.e., has a value false in the current state and a value true in the next, indicated by "@T" in the mode transition table. Once in mode `Inactive`, the system remains there until `Ignition` *becomes false* (indicated by "@F"), 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 (indicated by "t"), and `Toofast`, `Brake`, `Accel` are false (indicated by "f"). Values of monitored variables indicated by "–" are not relevant to that particular transition, e.g., a variable `Brake` in the transition from `Off` to `Inactive`. An SCR specification defines one mode transition table for each mode class of the system. Such tables are entered into SC(R)^3 using a mode class editor shown in Figure 1.

The Cruise Control system has a number of controlled variables to control the car throttle and display messages when the car is due for service or when it needs more oil. For example, a variable `ThrottleAccel` is true when the throttle is in the accelerating position and false

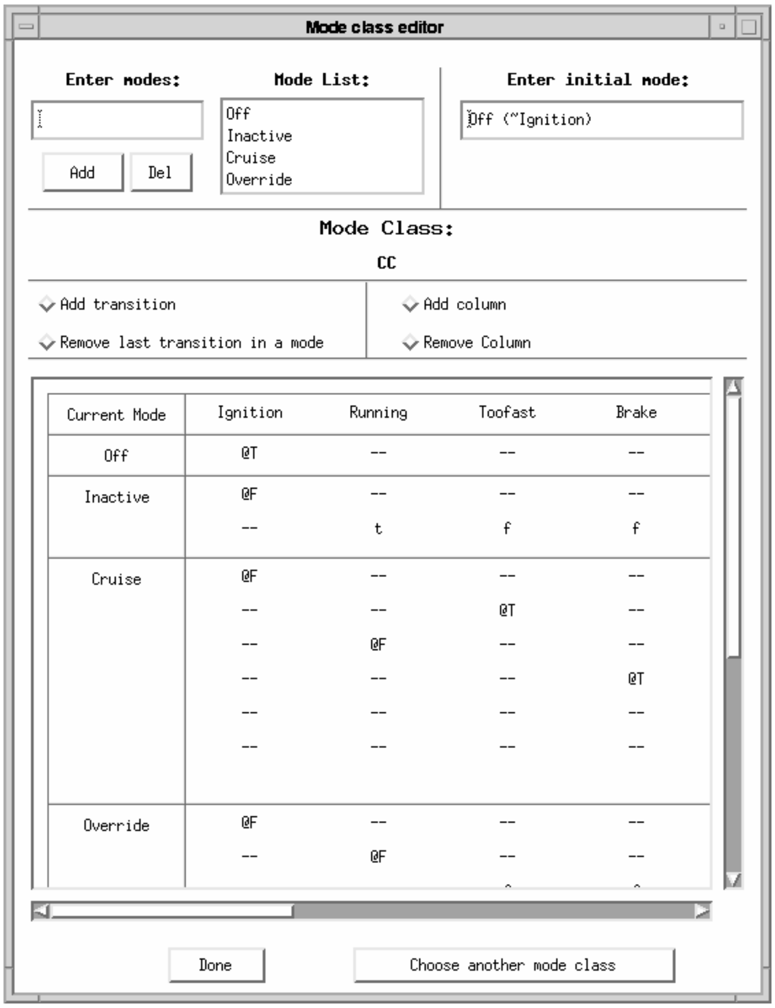

Figure 1: SC(R)^3 mode class editor.

otherwise. The condition table for this variable is shown in Table 5. `ThrottleAccel` is only evaluated in mode `Cruise`. The first row of the table specifies that `ThrottleAccel` should become true when the speed is too slow, and false when the system exits the mode `Cruise` (indicated as @F(Inmode)). When the system enters `Cruise` because the user pressed the resume button, the cruise control needs to maintain the previously set speed. Thus, the current speed is immediately evaluated, and if it is too slow, `ThrottleAccel` should become true, as indicated in the second row of the table. An SCR specification defines one table for each controlled variable of the system. The SC(R)^3 module for specifying these variables is shown in Figure 2.

SC(R)^3 allows to record assumptions of the requirements about the behavior of the environment. Assumptions specify constraints on the values of conditions, imposed by laws of nature or by other mode classes in the system. In particular, we can assume that the engine is running only if the ignition is on (`Running -->> Ignition`), the car can be going "too fast" only if it is running (`Toofast -->> Running`), and various boolean conditions are related by enumeration. The later cat-

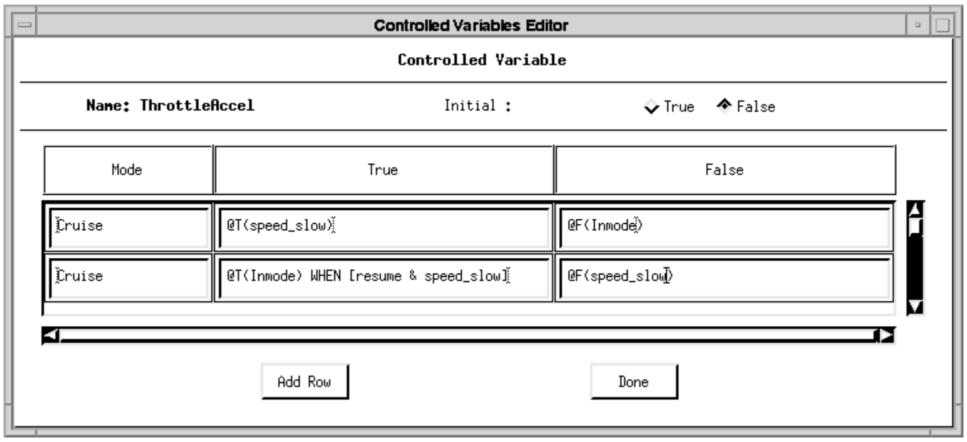

Figure 2: SC(R)^3 controlled variable editor.

Mode	True	False
`Cruise`	@T(`speed_slow`)	@F(Inmode)
`Cruise`	@T(Inmode) WHEN [`b_Resume` & `speed_slow`]	@F(`speed_slow`)

Initial: False

Table 5: Event table for the controlled variable `ThrottleAccel`.

Variable	Description
`Ignition`	ignition is on
`Running`	engine is running
`Toofast`	Sp_Vehicle > MAX_SPEED
`Brake`	brake pedal is being pressed
`Accel`	gas pedal is being pressed
`b_Cruise`	cruise button is being pressed
`b_Resume`	resume button is being pressed
`b_Off`	off button is being pressed
`speed_slow`	Sp_Vehicle < Sp_Desired - THRESHOLD[1]

Table 2: Select Cruise Control monitored variables.

egory includes predicates on the vehicle's speed (`speed_slow / speed_ok / speed_fast`) and buttons controlling the cruise control system (`b_Off / b_Cruise / b_Resume`).

The last section of requirements specification consists of the system goals. These are not con-

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 3: Modes of the mode class `CC`.

straints on the system behaviour but rather *putative theorems* – global properties that should hold in the system under specification, e.g. "the light will eventually become green", or "reversing a list twice gives us the original list". The language for specifying these properties in SC(R)^3 is an extension of CTL (Computational Tree Logic). CTL is a branching-time temporal logic [6] 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 $\sim f$ (negation), f & g (conjunction), and $f \mid g$ (disjunction). 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 addition to propositional formulae, our language allows to express properties involving SCR events, e.g.

> **[Property 1]** If the system is in mode `Override`, then it will react to the event `@F(Ignition)` by immediately going to the mode `Off`.

The semantics of events @T(a) (@F(a)) is that a is true (false) in the current state and false (true) in the previous. To ease the phrasing of CTL formulas that refer to the occurrence of conditioned events, we use unary logic connectives @T and @F to express the SCR notions of *becoming true* and *becoming false*.

Typically, the properties we are interested in specifying include invariants, reachability properties – these are important to ensure that the invariant properties are not true vacuously, and "progress" (bounded liveness) properties. For example, some of the invariant properties of the Cruise Control system is "whenever the system is in mode `Override` of modeclass `CC`, the system is running and the ignition is on", formalized as

$$AG(\texttt{CC} = \texttt{Override} \rightarrow (\texttt{Ignition} \,\&\, \texttt{Running}))$$

and "predicates representing the state of the Throttle are related by enumeration", formalized as

$$AG(\texttt{ThrottleOff} \mid \texttt{ThrottleMaintain} \\ \mid \texttt{ThrottleAccel} \mid \texttt{ThrottleDecel})$$

"Progress" properties are used to check that the system behaves according to our expectations. For example, one of the "progress" properties of the Cruise Control system is Property 1 given above, formalized as

$$AG(\texttt{CC} = \texttt{Cruise} \rightarrow \\ AX(@F(\texttt{Ignition}) \rightarrow \texttt{CC} = \texttt{Off}))$$

Note that in this property we use a logical connective @F.

3 Checking Consistency of Requirements

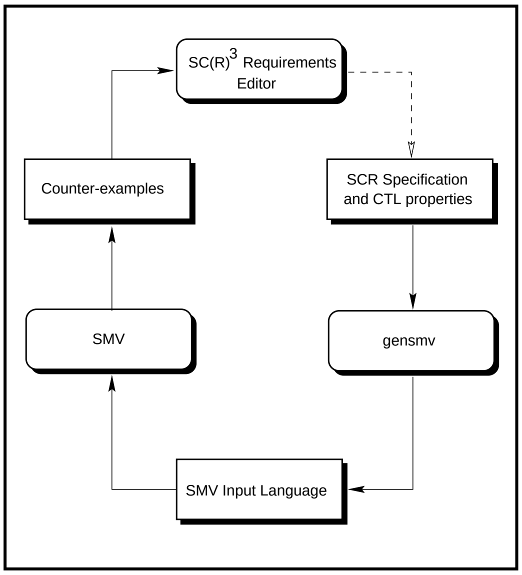

Figure 3: Requirements analysis with SC(R)^3.

System goals allow semantical checks on requirements, i.e. checks that these goals hold in the specification of the system. We chose to use *model-checking* [6] to perform these checks. A symbolic model-checker `SMV` developed by McMillan [17], uses state exploration to check if temporal properties hold in a finite-state model. However, we first needed to translate the behavioral specifications of SCR into a format accepted by `SMV`, which is done using a tool `gensmv`. The requirements analysis process is depicted in Figure 3, where tools and artifacts are represented by ellipses and boxes, respectively. In this section, we briefly describe `gensmv`, outline counter-example facilities of `SMV` and SC(R)^3, and present results of verification of the Cruise Control system.

3.1 Translating SCR Specifications

gensmv [24] was developed at the University of Waterloo to reason about mode transition tables. Before translating SCR specifications, gensmv details the mode transition tables with information derived from environmental assumptions. For example, an assumption Running --≫ Ignition adds information to the second transition from mode Inactive to mode Cruise (third row of Table 4): if Running is true, then Ignition should already be true.

Detailed SCR tables are translated into the SMV input language. In order to translate added logic connectives @T and @F to regular CTL, accepted by SMV, gensmv needs to store previous values of variables which can occur in events. This is facilitated by introducing additional variables in the SMV model. For example, in order to reason about a property involving @F(Ignition), we need an additional variable PIgnition, which is assigned the current value of Ignition before the next value of this variable is computed. Thus, Property 1 is translated into CTL as

$$AG(\texttt{CC} = \texttt{Cruise} \rightarrow AX((\texttt{PIgnition}\ \& \sim \texttt{Ignition}) \rightarrow \texttt{CC} = \texttt{Off}))$$

We extended gensmv to translate condition tables of controlled variables into the SMV modeling language [4]. This way, we are able to reason about the entire SC(R)^3 specification.

3.2 Counter-examples

During verification of CTL properties, SMV explores all possible behaviors of the model and either declares that the property holds or gives a counter-example. Since we wanted to make calls to SMV completely transparent to the user, and because of the introduction of extra variables into the SMV model, we found it necessary 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.
For example, if we try to check the property "pressing the Cruise button at any point when

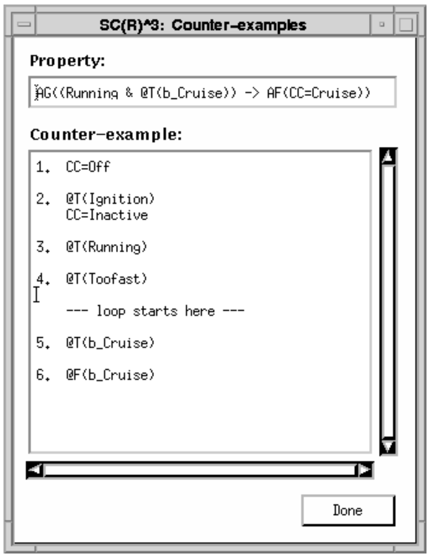

Figure 4: A counter-example generated during the requirements analysis.

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 4. 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.

3.3 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 SPARCstation-20 (2 75 MHz processors) with 256 megabytes of RAM using SMV version 2.4. The complete Cruise Control specification consists of 22 monitored and 13 controlled variables, translated into 47 SMV variables. Given that the size of the model grows exponentially to the number of variables in the system, we were not surprised that the initial runs of SMV did not complete after two days. However, we explored various

Test	User time	System time	BDD nodes	Total number of vars
Full specification				
-reorder	26487.25s	420.31s	34315278	47
Throttle properties				
Base	5.67s	0.2s	112687	26
-f -inc	11.06s	0.07s	18714	26
-reorder	5.69s	0.15s	75356	26
-f -inc -reorder	12.28s	0.1s	18630	26
All but throttle				
Base	8102.76s	50.59s	9780674	43
-f -inc	18783.8s	263.46s	3406360	43
-reorder	4237.56s	7.37s	4425504	43
-f -inc -reorder	13599.5s	12.54s	3406360	43

Table 6: Verification of the Cruise Control specification: experimental results.

ways to reduce the time and memory requirements necessary to verify the system. The following is the summary of our findings.

• The most effective technique is "slicing", i.e., removing variables that are not used in properties under verification. Currently, if a variable does not appear in any of the properties and no other variables depend on it, `SMV` still includes it into the models it builds. However, it is safe to remove such variables from consideration, thus greatly reducing sizes of the models. In SCR, there is a hierarchy of dependencies between requirements' variables, which is very easy to compute and take advantage of, although at the expense of producing separate `SMV` models. In SC(R)[3], the later process can only be done by hand so far, although we are developing an automatic model generation.

• Another technique is turning the variable reordering on. `SMV` uses binary decision diagrams (BDDs) to quickly manipulate logical expressions. Unfortunately, the size of the BDDs is extremely sensitive to the order of the variables used to build them [24]. `SMV` has an option[2]`--reorder` that allows it to heuristically compute and use a "better" variable reordering which is typically very effective. However, even with reordering turned on, `SMV` took a long time to verify the Cruise Control specification (26487.25 seconds of user time). To overcome this problem, we split the model into two: mode classes and controlled variables related to the throttle, and mode classes and all other controlled variables. Results of our experiments appear in Table 6. In this table, we list the user and the system time in seconds, and the total number of BDD nodes used during the verification, as reported by `SMV`. The last column of the table is the number of variables in the SMV model. We were unable to measure the running time in terms of real time, since `SMV` does not keep this information.

• A more controversial technique is building the state space incrementally, removing unreachable parts of the model (`SMV` options `--f` and `---inc`). This technique reduces the size of BDDs at the expense of the longer running time. Combining this option with `--reorder` typically yields a smaller number of BDD nodes than either of the options alone, but the verification time is slightly worse than by running `SMV` with with `--reorder` alone (see Table 6). We feel that both options should be turned on on a fast machine with a limited amount of memory, whereas just `--reorder` should be turned on on a slower machine.

The verification effort yielded a number of errors, most of which could be traced back to the original specification [15]. We found no errors in the mode transition table because this table has been analyzed earlier by Atlee and her colleagues [2][3]. However, we did find errors in controlled variable tables of the origi-

[2]Version 2.5 of `SMV` uses variable reordering by default.

[3]However, our study showed that a number of conditions used in Atlee's specification were not neces-

Throttle Setting				
Mode	**Condition**			
Cruise	speed_slow	speed_ok & ~Accel	speed_fast & ~Accel	Accel & ~ speed_slow
Throttle	Accel	Maintain	Decel	Off

Table 7: Condition table for variable Throttle.

nal specification. For example, the cruise control system includes a controlled variable OilOn 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). Thus, the vehicle could be traveling with the cruise control system turned off, not changing its modes and never setting OilOn to true. We modified the table for OilOn to set the variable to true when Omiles_low becomes true, regardless of the mode.

Another problem was found in the table for evaluating the throttle. In the notation used by Kirby, Throttle was an enumerated variable whose values are evaluated in mode Cruise as specified in Table 7. We found that condition ~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 check that, we modeled the variable Throttle in SMV and ran it against the property

$$\sim \texttt{Throttle} = \texttt{Off} \rightarrow AG(\sim \texttt{Throttle} = \texttt{Off})$$

which was verified, confirming our suspicion. Finally, the correctness of the specification is 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.

sary, 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.

4 Checking Correctness of Code

At some point in the future it will be possible to generate code from the black-box requirements specified in SCR. However, this code will likely require some hand-tuning, e.g., because it is too slow, or might even have to be rewritten from scratch. Mathematically precise software requirements, like SCR, can be used to reason about correctness of implementations. SC(R)^3 incorporates a tool called cord [3] that takes specially annotated source code and an SCR specification and checks for the correspondence between them. cord uses dataflow analysis instead of exhaustive state enumeration to enable effective verification in low-degree polynomial time. However, the analysis can sometimes be imprecise, i.e. "there *may be* a problem on this line of the code". In this section, we describe how to (correctly) annotate the code, outline the algorithms used in cord to check consistency between requirements and annotated code, and present results of verifying the implementation of the Cruise Control System.

4.1 Annotations

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 [3], are of three types. *Initial* annotations indicate the starting states of the program. They unconditionally assign values to variables and correspond to initialization information specified in the requirements. *Update* annotations assign values to controlled, mon-

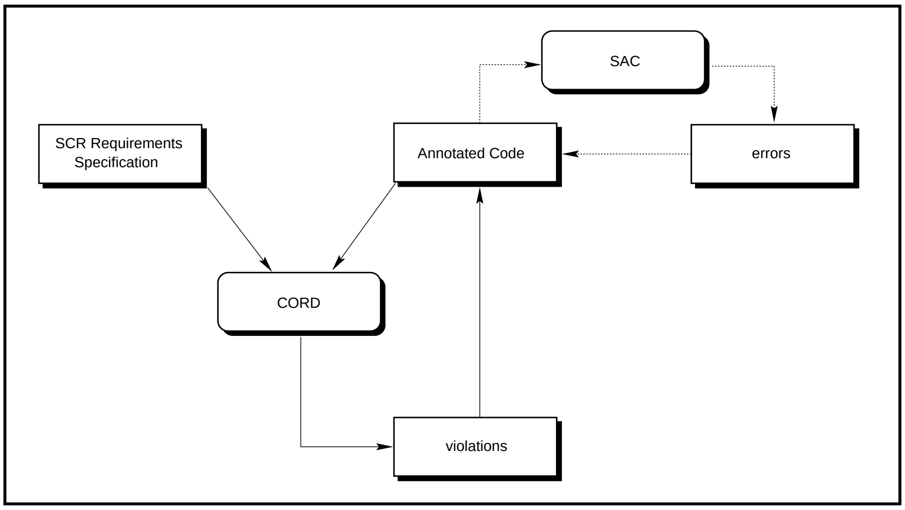

Figure 5: Code analysis with SC(R)^3.

itored or mode class variables[4]. These annotations identify points at which the program changes its state. *Assert* annotations assert that variables have particular values in the current system state. This makes our analysis more precise and serves as documentation of what the developer assumes about the system at a given point in the program.

Consider the following annotated code fragment, taken from the implementation of the Cruise Control system:[5]

```
if (!vIgnition) {
  @@ Assert ~Ignition;
  vIgnition = true;
  @@ Update Ignition;
  vCC = mInactive;
  @@ Update CC=Inactive;
}
else {
  @@ Assert Ignition;
  vIgnition = false;
  @@ Update ~Ignition;
  vCC = mOff;
  @@ Update CC=Off;
}
```

[4]Monitored and controlled variables have boolean values; mode class variables have values which form enumerated types whose constant values are modes of the mode classes.

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

In this fragment, code variables `vIgnition` and `vCC` correspond to requirements variables `Ignition` and `CC`. In the Then branch of the If statement we assert that `Ignition` is false, register that `Ignition` becomes true, marked by an Update annotation, and then change the mode of the system to `Inactive`. This is also marked by an Update annotation. The Else branch is similar.

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". Although annotations are easy to insert, we found ourselves frequently making mistakes, and also noticed that code maintenance greatly reduces the correspondence between the code and the annotations. Thus, the SC(R)^3 toolset includes a tool called `sac` [5] designed to check that the code is annotated correctly. To use the tool, the programmer creates a list of *correspondences* between variables in the requirements and the code. For example, the following is the specification of correspondences for the above code fragment:

```
correspondences:
{Ignition} -> {vIgnition};
{CC} -> {vCC};
```

These correspondences can be one-to-one (one requirements variable corresponds to one code variable), one-to-many (one requirements variable corresponds to several code variables), and many-to-one. `sac` ensures that an assignment to (a check of) a code variable is always followed by an Update (an Assert) of an appropriate requirements variable. The entire process of code verification with SC(R)^3 is depicted in Figure 5. Here, tools (`sac` and `cord`) and artifacts are represented by ellipses and boxes, respectively.

4.2 Analysis

Analysis done by `cord` is described in detail elsewhere [3]. In this section, we give a quick summary of this process.

`cord` checks that transitions implemented in the code are exactly the same as those specified in the requirements. These checks correspond to three types of properties: (1) the code and the specification start out in the same initial state; (2) the code implements *all* specified transitions; and (3) the code implements *only* specified transitions. Properties of types (2) and (3) are called ALT ("all legal transitions") and OLT ("only legal transitions"), respectively. For example, `cord` checks an ALT property that 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 4. One of the OLT properties is "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)`".

Verification is done via static analysis of the annotated code. A technique similar to *constant propagation* is used to create a finite-state abstraction of the annotations and control-flow program statements of the program. We use an aggressive state-folding strategy aimed at minimizing the number of states. This number is bound above by the number of annotations and control-flow structures in the code and is almost not affected by the number of variables in the specification. After the model has been created, we check it for consistency with the specification. Typically, the properties involve fairly short (2-3 states) fragments of the paths through the code, thus enabling very efficient checking. In addition, `cord` can verify invariant and reachability properties, find unreachable states, and check that environmental assumptions are not violated in the code.

Fast and highly scalable processing comes at a price of inexact verification. Abstraction used in `cord` leads to computing more behaviors than can be present in the code. Thus, OLT properties and invariants are checked *pessimistically*, i.e., some violations that are not present in the code can be reported. ALT and reachability properties are checked *optimistically*, i.e., some violations in the code can be overlooked. We believe that `cord` should be used as a debugging rather than a verification tool and found it to be extremely effective in discovering errors (see Section 4.3).

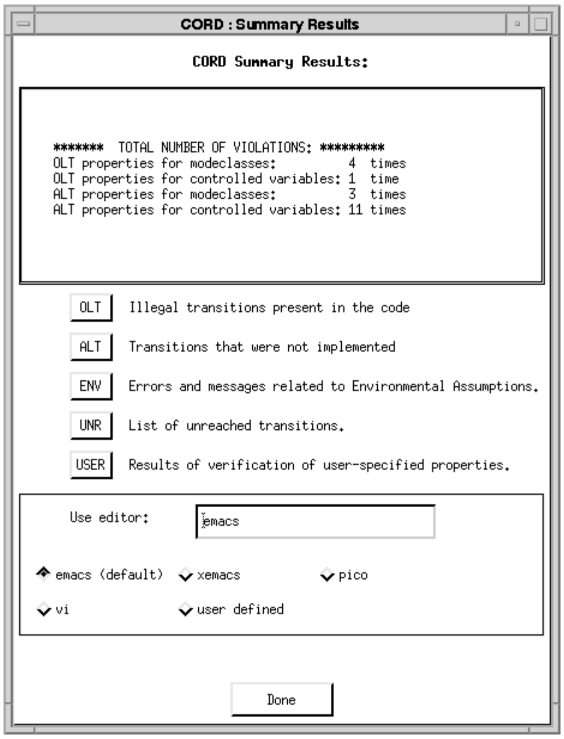

Figure 6: Summary of errors.

We had to resort to annotating code because our analysis techniques could only handle simple operations on booleans and enumerated types. A next generation of `cord` is currently being developed. For finite types, this version of the tool will be able to handle more interesting operations, like addition and comparison.

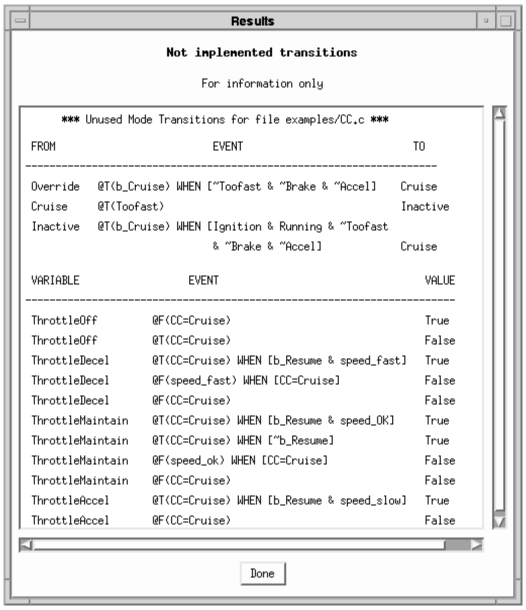

Figure 7: ALT errors.

It will also be able to approximate infinite types and perform operations on them using abstract interpretation [8]. This will allow us to analyze code directly rather than using annotations.

4.3 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 37 Update, 25 Assert 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 14 ALT and 5 OLT violations (see Figure 6). The ALT and the OLT violations are shown in Figures 7 and 8, respectively.

One OLT and one ALT violation were caused by an incorrect annotation overlooked by `sac`. `sac` does not consider values of variables when checking 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` computed that the system can be in all possible modes before this line, whereas only `Cruise` is possible. This problem arises because of the imprecise analysis used in `cord` and can be fixed by adding an extra Assert annotation. 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 is only enabled 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 `ThrottleOff`. All other ALT properties came from transitions that were not implemented in the code.

5 Conclusion

In this paper we presented SC(R)^3 – an integrated toolset for specification and reasoning about tabular requirements. Through a unified interface, SC(R)^3 allows to check software requirements for correctness and analyze consistency between annotated code and requirements. We are currently working on developing support for automatic code generation and are looking into ways of using SCR for generation of black-box test cases. We believe that SC(R)^3 is an important step towards increasing usability of formal methods: it attempts to replace free-form reasoning in logic by easy to write and review structured tables and amortizes the cost of creation of formal requirements through multiple automated analysis activities. We envision the following methodology for using 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.

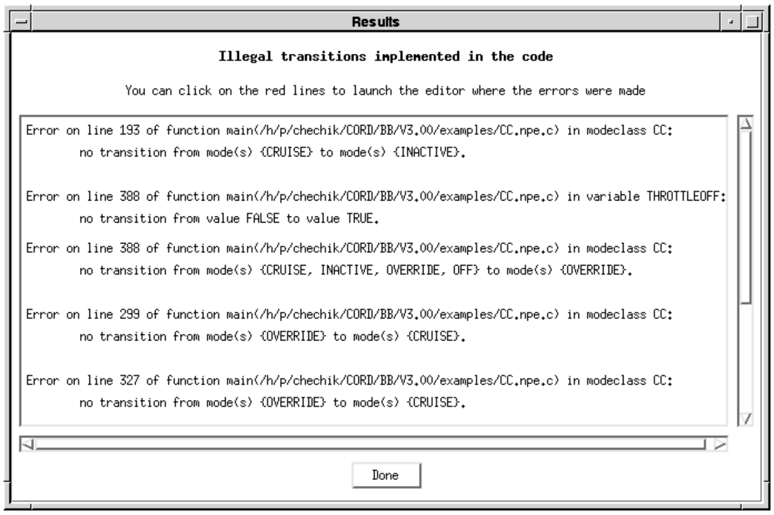

Figure 8: OLT errors.

2. SC(R)^3 automatically translates the SCR specification into the input to a model-checker, verifies properties and translates counter-examples into the SCR notation.

3. 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.

4. Using the specification, SC(R)^3 generates test cases. Although the mode logic is correct by construction, interface is likely to need testing.

5. The resulting implementation is analyzed for performance and optionally manually fine-tuned.

6. If manual fine-tuning was used, the code is automatically checked for consistency with its specifications.

7. Once the problems pointed out by the analysis are fixed, the code is tested using (a subset of) test cases generated in step (4) above.

In Figure 9, analysis activities performed within SC(R)^3 and software artifacts are shown inside ellipses and boxes, respectively. For example, the code generation activity takes SCR requirements as an input and generates an implementation in C. A dashed line signifies that the generated code and the code that is analyzed for correctness, represent the same artifact.

In SC(R)^3, requirements analysis is done via model-checking. Although model-checking is an effective verification technique which can be used without any user input, it has a number of limitations. Checking can often be prohibitively expensive and cannot usually be applied to infinite-state systems. Thus, we are limited to verifying just control (as opposed to data or timing) aspects of the specification. In order to use model-checking effectively, we had to reduce the expressive power of our input logic, which may not be feasible for many applications. We recognize that more sophisticated verification might be necessary and are planning to experiment with using a theorem-prover, e.g. PVS [18], to check complex type-checking and timing properties of systems.

Code verification in SC(R)^3 is done via a static-analysis tool `cord`. Although effective in finding errors in our case studies, `cord` is still a prototype tool in need of major improvements. We are currently redesigning it to pro-

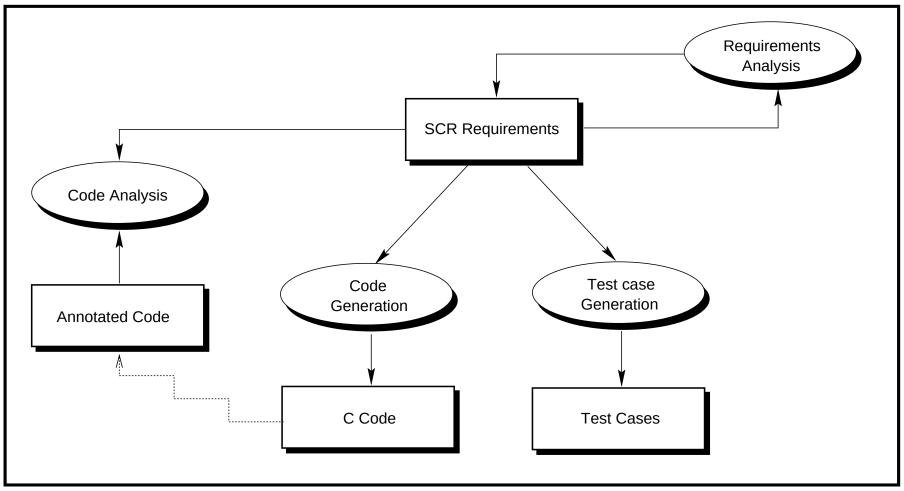

Figure 9: Pictorial description of activities that can be performed with SC(R)^3.

cess a more expressive annotation language and reason about infinite-domain variables. This would make `cord` able to verify implementations of unrestricted SCR specifications.

Acknowledgments

Joanne Atlee and her students developed `gensmv` and performed the initial analysis of the Cruise Control system. They also helped in shaping the look and feel of the GUI for specifying requirements. The author is also grateful to John Gannon and Rich Gerber for teaching her about formal methods, to Connie Heitmeyer and Stuart Faulk for many fruitful discussions about SCR, to David Wendland and Hicham Mouline who implemented and improved many aspects of the tools, to Sai Sudha for developing `sac`, and to Matthew Cwirko-Godycki for his help with the Cruise Control case study.

This research was supported in part by the NSERC Grant # OGP0194371 and by the University of Toronto's Connaught Fund.

About the Author

Marsha Chechik is an assistant professor in the Department of Computer Science at the University of Toronto. She can be reached at the address Department of Computer Science, University of Toronto, 10 King's College Rd., Toronto, ON M5S 3G4. Her e-mail address is `chechik@cs.toronto.edu`.

Dr. Chechik's interests are mainly in the application of formal methods to improve quality of software. She is also interested in other aspects of software engineering and in computer security.

References

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

[2] 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.

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

[4] M. Chechik and M. Cwirko-Godyski. "Cruise Control System: A Case-Study in

Using SCR in Development and Verification of Event-Driven Systems". Technical report, University of Toronto, June 1998. (in preparation).

[5] 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.

[6] 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.

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

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

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

[10] C. Heitmeyer, R. Jeffords, and B. Labaw. "Automated Consistency Checking of Requirements Specifications". *Transactions on Software Engineering and Methodology*, 1996.

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

[12] 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.

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

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

[15] 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.

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

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

[18] S. Owre, N. Shankar, and J. Rushby. "User Guide for the PVS Specification and Verification System (Draft)". Technical report, Computer Science Lab, SRI International, Menlo Park, CA, 1993.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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