Using the B-Toolkit to Ensure Safety in SCR Specifications

Indrakshi Ray* Paul Ammann

Department of Information and Software Systems Engineering
George Mason University, Fairfax, Virginia
Email: {indrakshi, pammann}@gmu.edu

Abstract

SCR (Software Cost Reduction) specifications are useful for specifying event-driven systems. To use SCR effectively for critical applications, automated verification of safety properties is important. Towards this end, Atlee and others proposed model checking and symbolic model checking approaches. The fact that these approaches are sometimes problematic motivates our alternative approach based on theorem proving. Theorem proving, in general, is a difficult task; however, the regular structure of the proof obligations generated from SCR specifications suggests that relatively unsophisticated theorem provers can discharge many of these obligations. As a feasibility study, we use the B-Toolkit to detect safety violations in an example SCR specification. The B-Toolkit is a good choice because it is commercially available and supports verified refinement to executables in a commercial programming language (C). We convert the mode transition table in the example SCR specification to an AMN (Abstract Machine Notation) specification and analyze the result with the B-Toolkit. The B-Toolkit generates 120 proof obligations of which 113 are automatically discharged by the theorem prover. The remaining 7 proof obligations are, in fact, not theorems and correspond to the 3 problems in the SCR specification detected by the model checking approaches. For the corrected SCR specification, the B-Toolkit automatically discharges all proof obligations. The example shows that even simple theorem provers are a viable approach to automated analysis for SCR specifications.

Contact Author: Paul Ammann, ISSE Department, Mail Stop 4A4 S&T II Room 430, George Mason University, Fairfax, VA 22030. 703 993–1660; Fax: 703 993–1638; pammann@gmu.edu

---

*Partially supported by a George Mason University Graduate Research Fellowship Award

1 Introduction

The use of formal methods has been advocated [PST91] to reduce the likelihood of introducing errors in the early stages of system development. SCR (Software Cost Reduction) [Hen80] is one such method used to formally capture and document the requirements of a software system. The SCR method and its derivatives have been used to specify a variety of practical systems [HM83, vSPM93, FFK+94]. To use SCR effectively for safety-critical applications, automated analysis of SCR specifications with respect to external safety properties is necessary. Towards this end, we propose a theorem proving based approach to verify safety properties with respect to SCR mode transition tables. As a feasibility study, we use the B-Toolkit to verify safety properties with respect to an example SCR mode transition table.

The research in automated checking of SCR specifications includes the consistency checker of Heitmeyer, Jeffords, and Labaw [HJL96] and the model checking approaches first developed by Atlee [AG93, Atl94, AB96, ORS95]. The consistency checker [HJL96] analyzes application independent properties such as syntax, type mismatch, missing cases, circular dependencies and so on, but not application dependent properties such as safety and security. Atlee [AG93, Atl94, AB96] addresses the application dependent property of safety in SCR mode transition tables by expressing an SCR mode transition table as a logic model, expressing the safety properties of the specification as logic formulae, and using a model checker to determine if the formulae hold in the model. Owre, Rushby, and Shankar [ORS95] describe how the model checker in PVS can be used to verify safety properties in SCR mode transition tables.

We provide an alternative approach based on theorem proving for verifying whether an SCR mode transition table satisfies a given set of safety properties specified as invariants. Theorem proving is often a difficult task, however, the regular structure of the proof obligations associated with safety maintenance in SCR specifications suggests that relatively unsophisticated theorem provers can discharge many of these obligations. As a feasibility study we use the B-Toolkit, an existing software development tool, to detect safety violations in an example SCR specification. We convert the mode transition table of the SCR specification into a set of operations in Abstract Machine Notation (AMN), the specification language recognized by the B-Toolkit, state the safety properties as invariants, and apply the analysis procedures in the B-Toolkit. The PO Generator, the proof obligation generator of the B-Toolkit, generates 120 proof obligations for the example application, of which 113 are successfully discharged by the AutoProver, the theorem prover of the B-Toolkit. The remaining 7 proof obligations are not theorems and correspond to the 3 problems in the underlying SCR specification. We reapply our approach to the corrected SCR specification; this time 123 proof obligations are generated all of which are successfully discharged. The fact that AutoProver, which has limited theorem proving capabilities, is capable of discharging the proof obligations demonstrate that even simple theorem provers can be used for automated safety analysis of SCR specifications.

The rationale for our approach is as follows. First, the binary decision diagrams (BDDs) used in model checking have computational anomalies. For example, the order in which variables are considered can have a dramatic effect on the computational time required [SA96]. Therefore, alternate methods are desirable, and, to our knowledge, no one else has reported on the effectiveness of automated theorem proving for evaluating safety in SCR mode transition tables. Note that the study in [ORS95] used only the model checking capabilities of PVS. Second, the use of a comprehensive method such as the one embodied by the B Toolkit is beneficial. In addition to analyzing safety properties, the tools provided by the B-Toolkit can be used to check other consistency properties, animate the AMN specification to help the developer decided whether the original SCR specification conforms to the requirements, and formally refine the AMN specification to executable code in a commercial language.* In brief, it advantageous for safety analysis to be part of an integrated tool set for software development. Third, the present AutoProver in the B Toolkit is relatively weak, which has the effect of strengthening our results. Despite its relative lack of sophistication, the B Toolkit AutoProver discharges the necessary obligations in our example without any intervention from the user. Pratten [Pra95] discusses the use of the more powerful theorem prover PVS [ORSvH95] for AMN specifications where AutoProver is inadequate; such an approach would be a useful reserve for treating SCR specifications where AutoProver runs into difficulty.

The remainder of the paper is organized as follows. Section 2 discusses work related to automated analysis of SCR specifications. Section 3 briefly describes SCR specifications with the help of an example application. Section 4 discusses the transformation procedure that transforms an SCR specification to an equivalent AMN specification. Section 5 describes how an AMN specification is analyzed for maintenance of safety properties specified as invariants on a single state.† Section 6 describes the results obtained by applying our approach to an SCR specification satisfying all the safety properties. Section 7 discusses how our approach can be extended to verify safety properties

*Presently the B Toolkit supports refinement to C.

†We do not address safety properties about reachability because the AMN specification language presently does not support temporal logic.

constraining consecutive states. Section 8 concludes the paper.

2 Related Work

In [HJL96] the authors present a formal consistency checking analysis for SCR requirements specifications. SCR requirements specifications are expressed in the form of event tables, condition tables and mode transition tables. Consistency checking involves checking these tables for application independent properties such as type errors, nondeterminism, missing cases and so on. Consistency checking is prerequisite to evaluating application dependent properties such as safety, security and timing. Our work focuses on verifying the application dependent property of safety in SCR mode transition tables.

Owre, Rushby and Shankar [ORS95] describe how the PVS can be used to check for both application independent as well as application dependent properties. The application independent checks include checks for disjointness, coverage, and well-formedness of SCR tables. The PVS model checker checks for application specific properties such as safety. The authors apply their approach on the cruise control example of Atlee and Gannon [AG93]; the results are consistent with those of alternative model checking approaches [AG93, AB96].

Atlee and Gannon [AG93] were the first to propose model checking approaches to analyze the maintenance of safety properties in SCR mode transition tables. They analyze a cruise control system and a water level monitoring system. The SCR specifications are converted to an extended finite state machine called a CTL machine [BCD86, CES86], and the safety properties for the system are written as CTL formulae. A model checker [Bro89] determines whether or not the properties hold. In both the case studies performed by Atlee and Gannon [AG93] discrepancies were found between the SCR specifications and the safety assertions. One drawback[‡] of this approach is that only safety properties consisting of conjunction of input conditions and negated input conditions can be verified.

Atlee addressed this problem in subsequent works [Atl94, AB96, SA96] where she proposed the notion of an SCR logic model for expressing the semantics of SCR mode transition tables. The SCR specifications are checked using an SMV symbolic model checker [McM92]. The results obtained on applying this approach to the cruise control system [Atl94, AB96] were consistent with those obtained using the earlier [AG93] approach. Also a safety property of the cruise control system that

[‡]the authors made this trade-off to avoid the state explosion problem

could not be verified using model checking approach [AG93] could now be verified. In a subsequent work [SA96] Sreemani and Atlee demonstrate the feasibility of symbolic model checking real life applications by verifying safety properties of A-7E aircraft.

The mode invariants in an SCR mode transition table can be viewed as implying a set of safety properties. Jin [Jin96] propose an algorithm for finding mode invariants in a given SCR mode transition table. This algorithm is based on the fact that mode invariant is the conjunction of the conditions causing the system to enter a mode and the negation of the conditions that cause the system to leave the mode. In our approach we do not find the set of mode invariants from a given SCR specification; we verify whether a given set of mode invariants hold in an SCR mode transition table. We assume that a mode invariant is expressed as any propositional logic formula; this is in contrast with Jin who assumes that mode invariants are conjunctions of input conditions and negated input conditions and consequently cannot detect mode invariants of the form $A \vee B$ where A and B are two input conditions. Jin's approach [Jin96] has not yet been automated.

The previous approaches are focussed on verifying the correctness of SCR specifications. However, the task of validating whether the operating system complies with the requirements still remains. Towards this end, Blackburn, Busser, and Fontaine [BBF] propose an approach for deriving specification-based tests for SCR specifications. The authors describe how an SCR specification can be modeled by a specification in the T-VEC system which in turn can be tested using the test vector generator of the T-VEC system. The test vector generator automatically determines test inputs, expected outputs and the set of tests which help test a given specification.

3 SCR Specifications

An important feature of the SCR specification is the use of tabular notations to represent various aspects of the system being modeled. Tabular notations, providing a clear and concise representation of the behavior of a large system, makes the semantics intuitively clear. SCR specifications are tabular notations of three varieties: condition tables, event tables, and mode transition tables. Following Atlee and Gannon, we focus on mode transition tables for analyzing invariant maintenance. The structure of a mode transition table will be described after presenting some background on SCR.

SCR requirements specifications describe the behavior of event driven mode machines. The input to these machines is a set of boolean environmental conditions termed *conditions*. The possible system states are divided into disjoint classes called *modes*, where states belonging to the same mode have some characteristic in common. A set of modes describing the behavior of some aspect of the system comprises a *modeclass.* A change in the value of an input condition is termed an *event.* Events are represented as follows:

- *Trigger* conditions: @T(Cond1) – denotes the event that trigger condition Cond1 becomes true. @F(Cond1) is the event that trigger condition Cond1 becomes false.

- *WHEN* conditions: @T(Cond1) WHEN [Cond2] or @F(Cond1) WHEN [Cond2] – denotes the event of trigger condition Cond1 changing while WHEN condition Cond2 holds.

Trigger conditions are unsatisfied before the event and are satisfied at the event. WHEN conditions are satisfied before as well as at the event. Note that we use the same semantics as used by Atlee [AG93, Atl94, AB96]; we differ from the SCR operational semantics [HJL96] where the WHEN condition is satisfied before the event but its value at the event is unspecified.

An event may cause a mode transition to take place. A *mode transition* takes the system from one mode to another. The mode transitions in a system and the events causing these transitions are listed in a mode transition table. The next section uses an example application to describe the structure of a mode transition table and other aspects of the SCR specifications that are relevant to our current work.

3.1 An Example SCR Specification

For an example, we choose a variant of the cruise control problem used in [AG93, Atl94]. At this stage we wish to demonstrate the feasibility of theorem proving approach for verifying SCR specifications; hence we take a very simple example SCR specification. An SCR mode transition table for this system appears in Table 1.

The states in the Cruise Control can be divided into modes *Off*, *Inactive*, *Cruise*, and *Override*. The system is in mode *Off* when the ignition is off. When the ignition is on and the cruise control is not, the system is in mode *Inactive*. In mode *Cruise*, the ignition and cruise control are on and are controlling the vehicle's speed. In mode *Override*, the ignition and cruise control are on but not controlling the vehicle's speed. The inputs to the Cruise Control System are *Ignited*, *Running*, *TooFast*, *Brake*, *Activate*, *Deactivate*, and *Resume*.

The first four inputs describe respectively if the ignition is on, if the engine is running, if the

Current Mode	*Ignited*	*Running*	*TooFast*	*Brake*	*Activate*	*Deactivate*	*Resume*	New Mode
Off	@T	—	—	—	—	—	—	*Inactive*
Inactive	@F	—	—	—	—	—	—	*Off*
Inactive	t	t	—	f	@T	—	—	*Cruise*
Cruise	@F	—	—	—	—	—	—	*Off*
Cruise	t	@F	—	—	—	—	—	*Inactive*
Cruise	t	t	@T	—	—	—	—	*Inactive*
Cruise	t	t	f	@T	—	—	—	*Override*
Cruise	t	t	f	—	—	@T	—	*Override*
Override	@F	—	—	—	—	—	—	*Off*
Override	t	@F	—	—	—	—	—	*Inactive*
Override	t	t	—	f	@T	—	—	*Cruise*
Override	t	t	—	f	—	—	@T	*Cruise*

Initial State : Mode = *Off*

Table 1: Mode Transition Table for Cruise Control Example

automobile is going too fast, if the brake is pressed. The last three inputs describe the position of the cruise control lever.

The columns in the center of the table represent input conditions. Each row of the table describes an event that cause a transition from the current mode, shown on the left, to the new mode, shown on the right. A table entry of @T or @F under the column for input x indicates that x is a trigger condition. A table entry of 't' or 'f' under column for input x signifies that x is a WHEN condition. An entry of '—' under column for an input x indicates that the value of x is a 'don't care' for that event.

For example, Row 3 of Table 1 specifies that the mode transition from current mode *Inactive* to new mode *Cruise* takes place with the following event: @T(*Activate*) WHEN [$Ignited \wedge Running \wedge \neg Brake$]. This states that if the system is in mode *Inactive* and *Ignited* and *Running* are true and *Brake* and *Activate* are false and the control lever is set to *Activate*, then in the new state the system is in mode *Cruise*, *Activate* is true, and remaining inputs are unchanged.

The SCR requirements specification for the cruise control system provides the safety invariants shown below. The second invariant states that if the system is not in mode *Off*, then *Ignited* must be true. In other words, the second invariant states that if the system is in any of the modes *Inactive*, *Cruise* or *Override*, then *Ignited* must be true. Note that the last three invariants ensure that *Ignited* is true if the system is in any of the modes *Inactive*, *Cruise* or *Override*. As the second invariant is implied by the last three invariants, it is redundant and can be omitted.

Mode	Safety Invariant
Off	$\neg$ *Ignited*
$\neg$ *Off*	*Ignited*
Inactive	*Ignited* $\wedge$ (($\neg$ *Running*) $\vee$ ($\neg$ *Activate*))
Cruise	*Ignited* $\wedge$ *Running* $\wedge$ $\neg$ *Brake*
Override	*Ignited* $\wedge$ *Running*

Note that some input conditions are related. Specifically, the following relationship holds between the input conditions signifying the position of the cruise control lever: *Activate* | *Deactivate* | *Resume*. Unlike [Atl94], we assume that *Running*, *Ignited* and *Toofast* are independent; consequently Row 6 in Table 1 differs from Row 6 of Table 1 in [Atl94]. This modification was necessary to make Table 1 deterministic. This Cruise Control example will subsequently be referred to as *CruiseControl*1.

4 Transforming SCR Tables To AMN Specifications

This section describes a procedure to transform a SCR mode transition table to an Abstract Machine Notation (AMN) specification. This procedure will henceforth be referred to as Transformation Procedure 1. (In Section 7 we describe another procedure identified as Transformation Procedure 2.) In the following subsections we describe the assumptions, the input and output of Transformation Procedure 1. The output produced by the Transformation Procedure 1 can be analyzed using the B-Toolkit.

4.1 Assumptions for Transformation Procedure 1

Our transformation procedure relies on the following assumptions:

1. Inputs are independent.

   In other words, an input can change without requiring a simultaneous change in some other input. This assumption often does not hold in typical SCR tables. However, a simple rewrite may remedy the defect. For example an enumerated type can replace a collection of interdependent boolean variables, as illustrated by *Activate*, *Deactivate* and *Resume* inputs of the cruise control example in Section 3. We discuss this after stating the other assumptions.

2. Inputs change one at a time.[§]

[§]This assumption is referred to in SCR operational semantics [HJL96] as the One-Input Assumption.

In other words, if two inputs change concurrently, the specification serializes these changes in an arbitrary order.

3. The specification contains a single mode class.

   Multiple mode classes could easily be handled separately. Our method does not address interactions between mode classes.

4. Safety properties are specified as invariants which constrain a single state, as opposed to properties that constrain future states in terms of the current state.

   This limitation is related to the B-Toolkit which does not support predicates on history of operations. ¶ Later we generalize this restriction and handle safety properties constraining consecutive states.

The syntax of the SCR tables accepted by our transformation procedure is different than in [AG93, Atl94, AB96]. In the following we discuss the differences. First, the dependent variables are represented by a single variable. In the new rewritten form, instead of having three columns for the cruise control position, we have only one column. Second, we show the exact transformation in the new table. For enumerated types which are not boolean the notation '@T' cannot be used because it does not signify the exact transformation. Hence we show the exact transformation of variables involved in the trigger and WHEN conditions; that is, we show the actual values of these variables both before the event and at the event. Third, for each row where the trigger condition is represented by an enumerated type variable which can take n values, we specify $n-1$ rows showing the exact transformation. For example, Row 3 in Table 1 has the following trigger condition: '@T(*Activate*)'. Now since the cruise control position can take on three values, corresponding to this row we specify two rows in the new table, one where the transformation is from *Resume* to *Activate*, and the other where the transformation is from *Deactivate* to *Activate*. The modified table is given in figure 2.

4.2 Inputs to the Transformation Procedure 1

The Transformation Procedure 1 requires as input the following items.

¶The use of an alternative specification language having a trace based semantics like Object Z would have enabled us to overcome this shortcoming. We did not use Object Z because it does not provide the necessary tool support.

Mode		Condition 1		Condition 2		Condition 3		Condition 4		Condition 5	
Old	New	Old	New	Old	New	Old	New	Old	New	Old	New
Off	*Inact*	*NIgnit*	*Ignit*	–	–	–	–	–	–	–	–
Inact	*Off*	*Ignit*	*NIgnit*	–	–	–	–	–	–	–	–
Inact	*Cruis*	*Ignit*	*Ignit*	*Run*	*Run*	–	–	*NBrk*	*NBrk*	*Res*	*Act*
Inact	*Cruis*	*Ignit*	*Ignit*	*Run*	*Run*	–	–	*NBrk*	*NBrk*	*Deact*	*Act*
Cruis	*Off*	*Ignit*	*NIgnit*	–	–	–	–	–	–	–	–
Cruis	*Inact*	*Ignit*	*Ignit*	*Run*	*NRun*	–	–	–	–	–	–
Cruis	*Inact*	*Ignit*	*Ignit*	–	–	*NToofst*	*Toofst*	–	–	–	–
Cruis	*Ovrde*	*Ignit*	*Ignit*	*Run*	*Run*	*NToofst*	*NToofst*	*NBrk*	*Brk*	–	–
Cruis	*Ovrde*	*Ignit*	*Ignit*	*Run*	*Run*	*NToofst*	*NToofst*	–	–	*Act*	*Deact*
Cruis	*Ovrde*	*Ignit*	*Ignit*	*Run*	*Run*	*NToofst*	*NToofst*	–	–	*Res*	*Deact*
Ovrde	*Off*	*Ignit*	*NIgnit*	–	–	–	–	–	–	–	–
Ovrde	*Inact*	*Ignit*	*Ignit*	*Run*	*NRun*	–	–	–	–	–	–
Ovrde	*Cruis*	*Ignit*	*Ignit*	*Run*	*Run*	–	–	*NBrk*	*NBrk*	*Res*	*Act*
Ovrde	*Cruis*	*Ignit*	*Ignit*	*Run*	*Run*	–	–	*NBrk*	*NBrk*	*Deact*	*Act*
Ovrde	*Cruis*	*Ignit*	*Ignit*	*Run*	*Run*	–	–	*NBrk*	*NBrk*	*Act*	*Res*
Ovrde	*Cruis*	*Ignit*	*Ignit*	*Run*	*Run*	–	–	*NBrk*	*NBrk*	*Deact*	*Res*

Initial State: $mode = Off$

Table 2: Modified Mode Transition Table of CruiseControl1

1. The modes comprising the mode class being modeled.
2. For each independent input condition, the list of possible values each condition can take.
3. The mode invariants of the mode class being modeled.
4. The initial values (if any) of the mode or input conditions.
5. The modified mode transition table, for example, the table shown in figure 2.

4.3 Outputs of the Transformation Procedure 1

The output produced by the Transformation Procedure 1 is an AMN specification. An AMN specification consists of two parts, namely a static part and a dynamic part. The static part contains a declaration of mathematical objects needed to model the state. These objects include basic sets, variables and constants. The static part also includes invariants which constrain the values of the variables. The dynamic part is composed of the initialization operation and the operations defined on the state variables. Once a system has been specified, the consistency of the system can be checked by ensuring that the invariants are satisfied by the values of the variables which are changed according to the operations specified in the dynamic part.

4.3.1 Sets

Each variable specified in the static part should belong to some basic set. Basic sets can be predefined (for example, set of natural numbers) or they can be defined by the user (for example, enumerated sets). For the AMN specification generated from the SCR table we define enumerated sets. The enumerated set $MODETYPE$ represents all possible values of mode. Similarly, for each independent input condition we specify an enumerated set which lists the possible values the condition can take. For the sake of uniformity we have defined new enumerated sets for conditions that are boolean. The sets required for modeling $CruiseControl1$ are given in figure 1.

4.3.2 Variables

In order to model the SCR mode transition table in AMN we require the following variables.

1. One state variable for each independent input in the SCR table.
2. One state variable to represent the mode.

The variables of $CruiseControl1$ are shown in figure 1.

4.3.3 Invariant

The invariant is a logical property that constrains the values of the variables. Moreover, it should contain enough information to relate each variable to its basic set. The invariant in the AMN specification is the conjunction of the safety invariants of the SCR specification and the type information of the variables. The invariant of $CruiseControl1$ is given in figure 1.

4.3.4 Initialization

The initialization operation consists of establishing the initial state. For each condition or mode in the SCR Table having an initial value, the corresponding variable in the AMN specification must be initialized with the same value. The initialization operation, as well as other operations, are defined using the notion of substitution. The notation ':=' represents substitution in AMN. When multiple variables are initialized, parallel substitution must be used. The individual substitutions in a parallel substitution are separated by '$||$' operator. When a condition is not initialized, a clause must be included to specify that the initial value of the variable is of the correct type. The notation '$:\in$' is used to specify that a variable is being assigned to some unspecified element of a given set – in this case, the underlying type. The initialization operation of $CruiseControl1$ is shown in figure 1.

4.3.5 Operations

The operations in an AMN specification are defined as *pre-conditioned substitution*, which means the substitution is carried out only if the predicate denoting the precondition of the operation holds. If the operations are carried out when the predicates are not satisfied, anything might happen. The substitution does not refer to variables which do not change in the course of the operation. The substitutions comprise the *body* of the operation [SH94]. The predicate that holds after the substitutions specified in the body have been carried out is referred to as the postcondition of the operation.

The operations generated from the SCR table are classified into three classes, which we term *explicit* operations, *rowWise remainder* operations and *notInTable remainder* operations. An explicit operation is associated with a row in the mode transition table; it corresponds to the mode transition operation. A rowWise remainder operation is also associated with a row; it corresponds to the operation which occurs when the mode transition cannot take place because one or more WHEN conditions are not satisfied. A notInTable remainder operation corresponds to those cases where an input changes, but there is no applicable row in the mode transition table. The semantics of the mode transition table is that the mode changes only for the explicit operations and remains unchanged for the rowWise remainder and notInTable remainder operations. The motivation for the remainder operations is totality. That is, we need to supply an operation for every state admitted by the invariants. The explicit rows in the SCR table supply partial operations. The rowWise remainder operations and notInTable remainder operations are intended to make the partial operations complete. We require that remainder operations be enumerated explicitly so that the invariants can be analyzed with respect to the changed input.

In the following paragraphs we explain the specification of the different types of operation. The precondition of the explicit operation consists of three components which are generated from the current mode, the trigger condition and the WHEN condition. The body consists of simultaneous

MACHINE

$CruiseControl1$

SETS

$MODETYPE = \{Off, Inact, Cruis, Ovrde\};$

$COND1TYPE = \{Ignit, NIgnit\};$

$COND2TYPE = \{Run, NRun\};$

$COND3TYPE = \{Toofst, NToofst\};$

$COND4TYPE = \{Brk, NBrk\};$

$COND5TYPE = \{Act, Deact, Res\}$

VARIABLES

$mode, cond1, cond2, cond3, cond4, cond5$

INVARIANT

$(mode = Off) \Rightarrow (cond1 \neq Ignit)) \wedge$

$((mode = Inact) \Rightarrow (cond1 = Ignit) \wedge (cond2 \neq Run \vee cond5 \neq Act)) \wedge$

$((mode = Cruis) \Rightarrow ((cond1 = Ignit) \wedge (cond2 = Run) \wedge (cond4 \neq Brk))) \wedge$

$((mode = Ovrde) \Rightarrow ((cond1 = Ignit) \wedge (cond2 = Run))) \wedge$

$mode : MODETYPE \wedge cond1 : COND1TYPE \wedge cond2 : COND2TYPE \wedge$

$cond3 : COND3TYPE \wedge cond4 : COND4TYPE \wedge cond5 : COND5TYPE$

INITIALISATION

$mode := Off \mid\mid cond1 :\in COND1TYPE \mid\mid cond2 :\in COND2TYPE \mid\mid$

$cond3 :\in COND3TYPE \mid\mid cond4 :\in COND4TYPE \mid\mid cond5 :\in COND5TYPE$

OPERATIONS

$Row3Op \mathrel{\widehat{=}}$

PRE

$mode = Inact \wedge cond5 = Res \wedge$

$cond1 = Ignit \wedge cond2 = Run \wedge cond4 = NBrk$

THEN

$mode := Cruis \mid\mid cond5 := Act$

END;

$Row3RemOp \mathrel{\widehat{=}}$

PRE

$mode = Inact \wedge cond5 = Res \wedge$

$\neg\, (cond1 = Ignit \wedge cond2 = Run \wedge cond4 = NBrk)$

THEN

$cond5 := Act$

END;

$NotInTabOp15 \mathrel{\widehat{=}}$

PRE

$mode = Inact \wedge cond2 = Run$

THEN

$cond2 := NRun$

END

END

Figure 1: AMN Specification $CruiseControl1$ Generated by Transformation Procedure 1

substitution of the variables associated with the mode and the trigger condition.

The precondition of the rowWise remainder operation consists of three components which are generated from the current mode, the trigger condition and the negation of the WHEN condition. The body consists of a single substitution of the variable representing the trigger condition. *Row3Op*, *Row3RemOp* in figure 1 are the explicit operation, the rowWise remainder operation respectively corresponding to Row 3 of Table 2.

The precondition of the notInTable remainder operation consists of two components,namely, the current mode and the trigger condition. The body consists of a substitution involving the variable representing the trigger condition. When the system is in mode *Inactive* and the event @F(*Running*) occurs, the corresponding notInTable remainder operation, denoted by *NotInTabOp*15, is shown in figure 1. The transformation procedure generates 16 explicit operations, 12 rowWise remainder operation, 40 notInTable operations. *Row1Op*, *Row2Op* denote the explicit operations corresponding to Row 1, Row 2 respectively. *Row4RemOp* denotes the rowWise remainder operation corresponding to Row 4. *NotInTab*1, *NotInTab*2, ..., denote the notInTable operations. Due to space constraints we do not list all the operations of *CruiseControl*1 in figure 1.

4.3.6 Number of Operations

Let $x_1, ..., x_m$ denote the inputs and $n_1, ..., n_m$ denote the number of possible values for each input. The total number of operations is the summation of the number of explicit operations, the number of rowWise remainder operations and the number of notInTable remainder operations. The number of explicit operations equals the number of rows in the mode transition table. The number of rowWise remainder operations equals the number of rows in the mode transition table having WHEN conditions; the number of rowWise remainder operations is therefore bounded by the number of rows in the mode transition table. The number of notInTable remainder operations in the worst case equals the number of modes multiplied by $\sum_{i=1}^{m} n_i(n_i - 1)$. (The above formula is derived as follows. Consider the first input condition x_1; x_1 can take on n_1 values. The number of possible operations in which x_1 changes from one value to another is given by $n_1(n_1 - 1)$. Since there are m input conditions, the total number of notInTable operations for each mode equals $\sum_{i=1}^{m} n_i(n_i - 1)$). Note that the number of operations is polynomial, not combinatorial, in the various parameters of the mode transition table.

5 Invariant Analysis of an AMN Specification

An AMN specification produced as output of the Transformation Procedure 1 must be analyzed for consistency. When the B-Method is used to write AMN specifications, the user is given specific guidelines as to how to proceed with the analysis. The guidelines are given in the form of specific proof obligations that must be successfully discharged. In this section we describe the proof obligations associated with the maintenance of the invariant. These proof obligations must be satisfied to ensure that the invariant holds in every state reached by the machine.

Note that these analysis rules are built into the B-Toolkit. The PO Generator of the B-Toolkit automatically generates these proof obligations. The proof obligations can be discharged using the AutoProver of the B-Toolkit or by any other external theorem prover such as PVS.

5.1 Initialization

First, we have to prove that the initialization establishes the invariant. For this we need to discharge the following proof obligation.

$$[init]I$$

This means we need to prove that the predicate obtained by applying the substitution *init* to the invariant I evaluates to *true*.

For example, in $CruiseControl1$ we must prove that

$$[mode := Off]I$$

where I represents the invariant shown in figure 1. To prove this we must show that the predicate, obtained by replacing every occurrence of *mode* in the invariant I by *Off*, evaluates to true. In $CruiseControl1$ this predicate does not evaluate to true and the initialization fails to establish the invariant. We discuss this later when presenting our results of the analysis.

5.2 Operations

Next, we prove that the invariant is preserved by each operation. An operation in an AMN specification is defined as a pre-conditioned substitution; the proof obligation associated with each operation is defined as follows.

$$I \wedge pre\_op \Rightarrow [op]I$$

The above expression means that if the invariant I and the precondition of the operation pre_op hold then the predicate obtained by applying the substitution op in the invariant I also holds.

For example, the operation $NotInTabOp15$ generates the following proof obligation.

$$I \wedge mode = Inact \wedge cond2 = Run \Rightarrow [cond2 := NRun]I$$

where I denotes the invariant in figure 1. This proof obligation can be successfully discharged ensuring that $NotInTabOp15$ maintains the invariant.

Note that the proof obligations associated with initialization and operations can be discharged using the mathematical laws of first-order logic and set-theory. The Autoprover of the B-Toolkit discharges the proof obligations using the library of mathematical rules.

5.3 Invariant Analysis of CruiseControl1

The AMN specification $CruiseControl1$ (whose partial specification is shown in figure 1) is analyzed using the B-Toolkit. The B-Toolkit PO Generator generates all the proof obligations associated with the AMN specification. The proof obligations are discharged by the AutoProver of the B-Toolkit. 120 proof obligations are generated of which 113 can be successfully discharged. (Note that to simplify the proof, the B-Toolkit generates proof obligations for each conjunct of the invariant rather than the whole invariant. As a result, the number of proof obligations generated exceeds the number of operations.) In the following subsections we discuss the proof obligations associated with initialization and operations that cannot be successfully discharged.

5.3.1 Initialization

1 initialization proof obligation cannot be discharged. This proof obligation is required to ensure that the condition ($cond1 \neq Ignit$) is true in the initial state. This means that the initial state fails to establish the invariant; specifically, the conjunct of the invariant associated with mode *Off* cannot be established.

5.3.2 Operations

The operations whose proof obligations cannot be successfully discharged are given in figure 2.

$Row1Op$ is the mode transition operation associated with Row 1 of Table 2 which occurs when the system is in mode *Off* and the ignition is turned on. This event takes the system to mode

$NotInTabOp16 \mathrel{\widehat{=}}$

PRE

$mode = Inact \wedge$

$cond2 = NRun$

THEN

$cond2 := Run$

END

$Row1Op \mathrel{\widehat{=}}$

PRE

$mode = Off \wedge$

$cond1 = NIgnit \wedge$

THEN

$mode := Inact \parallel$

$cond1 := Ignit$

END

$Row7Op \mathrel{\widehat{=}}$

PRE

$mode = Cruis \wedge$

$cond1 = Ignit \wedge$

$cond3 = NToofst$

THEN

$mode := Inact \parallel$

$cond3 := Toofst$

END

$Row3RemOp \mathrel{\widehat{=}}$

PRE

$mode = Inact \wedge$

$cond5 = Res \wedge$

$\neg ((cond1 = Ignit) \wedge$

$(cond2 = Run) \wedge$

$(cond4 = NBrk))$

THEN

$cond5 := Act$

END

$Row4RemOp \mathrel{\widehat{=}}$

PRE

$mode = Inact \wedge$

$cond5 = Deact \wedge$

$\neg ((cond1 = Ignit) \wedge$

$(cond2 = Run) \wedge$

$(cond4 = NBrk))$

THEN

$cond5 := Act$

END

$Row8RemOp \mathrel{\widehat{=}}$

PRE

$mode = Cruis \wedge$

$cond4 = NBrk \wedge$

$\neg ((cond1 = Ignit) \wedge$

$(cond2 = Run) \wedge$

$(cond3 = NToofst))$

THEN

$cond4 := Brk$

END

Figure 2: Problematic Operations in AMN Specification of Cruise Control

Inactive with the ignition turned on. We do not know the values of other input conditions, that is, whether the engine is running or the setting of the cruise control lever. As expected, the proof obligation, which derives the condition $(cond2 \neq Run \vee cond5 \neq Act)$ from the conjunction of the invariant and the precondition of $Row1Op$, cannot be discharged.

$Row7Op$ describes the mode transition that takes place when the system is in mode *Cruise* and *Toofast* becomes true. This event takes the system to mode *Inactive* with *Toofast* being true. To ensure the satisfaction of the invariant, we must be able to derive $(cond5 \neq Act)$ from the conjunction of the invariant and the precondition of $Row7Op$. Since this is not possible, the associated proof obligation cannot be discharged.

$Row3RemOp$ corresponds to the rowWise remainder operation of Row 3; it specifies the operation that occurs when the system is in mode *Inactive*, *Activate* becomes true and the mode transition to *Cruise* cannot take place because one or more WHEN conditions are not satisfied. As a result,the system remains in mode *Inactive* with the cruise control lever set to *Activate*. The conjunction of invariant and the precondition does not imply $(cond2 \neq Run)$ which is necessary to ensure the maintenance of the invariant; hence the corresponding proof obligation cannot be discharged. For similar reasons a proof obligation associated with $Row4RemOp$ cannot be discharged.

$Row8RemOp$ is the remainder operation corresponding to Row 8. This describes the operation that takes place when the system is in mode *Cruise* and the brake is pressed and one or more of the WHEN conditions specified in Row 8 is not satisfied. The system therefore remains in mode *Cruise* with the brake pressed on. Note that the postcondition of this operation is *false*. The proof obligation necessary to ensure invariant maintenance is as follows: the conjunct of invariant and precondition must imply *false*. This proof obligation also cannot be discharged.

$NotInTabOp16$ is the operation which corresponds to *Running* becoming true when the system is in mode *Inactive*. After the operation, the system remains in mode *Inactive* with *Running* being true. The conjunction of invariant and the precondition does not imply $(cond5 \neq Act)$ which is required to ensure the satisfaction of the invariant; hence, the associated proof obligation cannot be discharged.

5.3.3 Summary of Results

Thus, from our analysis, we conclude the invariants of the following mode are not satisfied in the Cruise Control Specification: *Off*, *Inactive* and *Cruise*. The initialization check reveals that the invariant is not satisfied in the initial state when the system is in mode *Off*. The postconditions

Current Mode	*Ignited*	*Running*	*TooFast*	*Brake*	*Activate*	*Deactivate*	*Resume*	New Mode
Off	@T	—	—	—	—	—	—	*Inactive*
Inactive	@F	—	—	—	—	—	—	*Off*
Inactive	t	t	f	f	@T	—	—	*Cruise*
Cruise	@F	—	—	—	—	—	—	*Off*
Cruise	t	@F	—	—	—	—	—	*Inactive*
Cruise	t	t	@T	—	—	—	—	*Inactive*
Cruise	t	t	f	@T	—	—	—	*Override*
Cruise	t	t	f	—	—	@T	—	*Override*
Override	@F	—	—	—	—	—	—	*Off*
Override	t	@F	—	—	—	—	—	*Inactive*
Override	t	t	f	f	@T	—	—	*Cruise*
Override	t	t	f	f	—	—	@T	*Cruise*

Initial State : Mode = *Off*, $\neg$ *Ignited*

Table 3: Mode Transition Table for *CruiseControl*2

of problematic operations involved modes *Inactive* and *Cruise*; thus the invariants of these modes do not always hold.

6 The Corrected Cruise Control Specification

In this section, we give an alternative cruise control specification which we verify using our techniques. The alterations are done according to the suggestions in [AG93]. This altered specification will be referred to as *CruiseControl*2.

The mode invariants are listed as follows.

Mode	Safety Invariant
Off	$\neg$ *Ignited*
Inactive	*Ignited*
Cruise	*Ignited* $\wedge$ *Running* $\wedge$ $\neg$ *Toofast* $\neg$ *Brake*
Override	*Ignited* $\wedge$ *Running*

The mode invariants of *Inactive* and *Cruise* are changed. *Inactive* has only one mode invariant which is *Ignited*. $\neg$ *Toofast* is now a mode invariant of *Cruise*. Since $\neg$ *Toofast* is a mode invariant of *Cruise*, the mode transition table is modified to include $\neg$ *Toofast* as a WHEN condition for all transitions that cause the system to enter mode *Cruise*.

The mode transition table is modified to include two initial conditions, namely, *Mode* = *Off* and $\neg$ (*Ignited*). The corresponding SCR table is shown in figure 3.

We apply Transformation Procedure 1 to this example and analyze the resulting AMN specification using the B-Toolkit. 124 proof obligations are generated all of which can be successfully discharged.

7 Safety Properties Constraining Consecutive States

The Transformation Procedure 1 presented in Section 4 assume that the SCR safety properties are specified as invariants (Assumption 4 in Section 4.1). In this section we briefly outline a technique which allows us to overcome this restriction to a limited extent.

We have another transformation procedure, known as Transformation Procedure 2, which takes as input the list of modes and conditions, the mode transition table, the invariants, and produces as output an AMN specification. The Transformation Procedure 2 is applied on a variation of the Cruise Control and the result is analyzed using the B-Toolkit. These details are described below.

7.1 Transformation Procedure 2

Assumptions 1,2 and 3 stated in Section 4.1 also hold for Transformation Procedure 2. Assumption 4 is relaxed to include invariants constraining consecutive states.

The inputs include the list of conditions, the list of modes, the list of initialization conditions, the invariants and the mode transition table.

The output is an AMN specification; however the AMN specification differs from that produced by Transformation Procedure 1. In the following paragraphs we describe the structure of the AMN specification produced as output.

<u>Sets</u> The basic sets are defined as in Transformation Procedure 1.

<u>Variables</u> In this approach the number of variables is twice of that required in Transformation Procedure 1. Each independent condition has two variables associated with it – one, which is prefixed by *old*, represents the value before the event takes place; the other, prefixed by *new*, represents the value at the occurrence of the event. Similarly two variables, namely, *oldmode* and *newmode*, is necessary to represent mode. A similar approach using shadow variables has been used in [Atl94].

<u>Invariant</u> The specification of the invariant differs from that of Transformation Procedure 1. For each invariant in the SCR specification defined over a single state, we now require two conjuncts of the invariant in the corresponding AMN specification; one involving the *old* variables and the

other involving *new* variables. For an invariant defined over consecutive states, a single conjunct in the AMN specification is required. As before, the type information must be specified as conjuncts of the invariant.

Initialization In an SCR specification if initial values are specified for a mode or input condition, the *old* as well as the *new* variable corresponding to that mode or input condition must be initialized in the AMN specification.

Operations From a given mode transition table, the number of operations generated by Transformation Procedure 2 is identical to that of Transformation Procedure 1. As before these operations are classified into explicit, rowWise remainder and notInTable remainder operations. The specifications of these operations are however different. The variables referenced in the precondition involve before state; these are therefore specified using *old* values. The variables referenced in the body of operations involve after state variables; hence the substitution made in these operations involve the *new* variables. Suppose a condition x does not change during the course of an operation; in such a case we require the following substitution $newx := oldx$ to be included in the body.

7.2 Applying Transformation Procedure 2 to CruiseControl3

The *CruiseControl*2 specified in Section 6 is modified to include the invariant given below. This modified example will be referred to as *CruiseControl*3.

$$AG(\mathit{Inactive} \wedge \neg\, \mathit{Activate} \wedge \mathit{Ignited} \wedge \mathit{Running} \wedge \neg\, \mathit{Toofast} \wedge \neg\, \mathit{Brake}) \Rightarrow$$
$$\neg\, EX(\mathit{Inactive} \wedge \mathit{Activate} \wedge \mathit{Ignited} \wedge \mathit{Running} \wedge \neg\, \mathit{Toofast} \wedge \neg\, \mathit{Brake}))$$

The above notation, used in [AG93], implies that if the system is in mode *Inactive* and the WHEN conditions (*Ignited*, *Running*, $\neg$ *Toofast*, $\neg$ *Brake*) are satisfied when the cruise control lever is changed to *Activate*, then the system cannot go to a state in which the WHEN conditions hold, *Activate* is true and the mode is still *Inactive*.

The AMN specification generated by applying the transformation procedure 2 to *CruiseControl*3 is given in Figure 3. For space constraints, we did not list all the operations.

7.3 Analyzing the AMN Specification of CruiseControl3

The AMN specification generated by applying the Transformation Procedure 2 to *CruiseControl*3 is analyzed using the B-Toolkit. All the 269 proof obligations generated by the PO Generator can be successfully discharged by the AutoProver.

MACHINE

$CruiseControl3$

SETS

$MODETYPE = \{Off, Inact, Cruis, Ovrde\};$

$COND1TYPE = \{Ignit, NIgnit\};$

$COND2TYPE = \{Run, NRun\};$

$COND3TYPE = \{Toofst, NToofst\};$

$COND4TYPE = \{Brk, NBrk\};$

$COND5TYPE = \{Act, Deact, Res\}$

VARIABLES

$oldmode, newmode, oldcond1, newcond1, oldcond2, newcond2,$

$oldcond3, newcond3, oldcond4, newcond4, oldcond5, newcond5$

INVARIANT

$((oldmode = Off) \Rightarrow (oldcond1 \neq Ignit)) \wedge$

$((oldmode = Inact) \Rightarrow (oldcond1 = Ignit)) \wedge$

$((oldmode = Cruis) \Rightarrow ((oldcond1 = Ignit) \wedge (oldcond2 = Run) \wedge (oldcond3 \neq Toofst) \wedge (oldcond4 \neq Brk))) \wedge$

$((oldmode = Ovrde) \Rightarrow ((oldcond1 = Ignit) \wedge (oldcond2 = Run))) \wedge$

$((newmode = Off) \Rightarrow (newcond1 \neq Ignit)) \wedge$

$((newmode = Inact) \Rightarrow (newcond1 = Ignit)) \wedge$

$((newmode = Cruis) \Rightarrow ((newcond1 = Ignit) \wedge (newcond2 = Run) \wedge (newcond3 \neq Toofst) \wedge (newcond4 \neq Brk))) \wedge$

$((oldmode = Inact \wedge oldcond1 = Ignit \wedge oldcond2 = Run \wedge oldcond3 \neq Toofst \wedge oldcond4 \neq Brk \wedge oldcond5 \neq Act) \Rightarrow$

$\neg (newmode = Inact \wedge newcond1 = Ignit \wedge newcond2 = Run \wedge newcond3 \neq Toofst \wedge newcond4 \neq Brk \wedge newcond5 = Act)) \wedge$

$oldmode : MODETYPE \wedge newmode : MODETYPE \wedge oldcond1 : COND1TYPE \wedge newcond1 : COND1TYPE \wedge$

$oldcond2 : COND2TYPE \wedge newcond2 : COND2TYPE \wedge oldcond3 : COND3TYPE \wedge newcond3 : COND3TYPE \wedge$

$oldcond4 : COND4TYPE \wedge newcond4 : COND4TYPE \wedge oldcond5 : COND5TYPE \wedge newcond5 : COND5TYPE$

INITIALISATION

$newmode := Off \mid\mid newcond1 := NIgnit \mid\mid oldmode := Off \mid\mid oldcond1 := NIgnit$

$newcond2 :\in COND2TYPE \mid\mid newcond3 :\in COND3TYPE \mid\mid newcond4 :\in COND4TYPE \mid\mid newcond5 :\in COND5TYPE$

$oldcond2 :\in COND2TYPE \mid\mid oldcond3 :\in COND3TYPE \mid\mid oldcond4 :\in COND4TYPE \mid\mid oldcond5 :\in COND5TYPE$

OPERATIONS

$Row1Op \mathrel{\widehat{=}}$

PRE

$oldmode = Off \wedge oldcond1 = NIgnit$

THEN

$newmode := Inact \mid\mid newcond1 := Ignit \mid\mid$

$newcond2 := oldcond2 \mid\mid newcond3 := oldcond3 \mid\mid$

$newcond4 := oldcond4 \mid\mid newcond5 := oldcond5$

END

END

Figure 3: AMN Specification $CruiseControl3$ Generated by Transformation Procedure 2

8 Summary and Conclusion

In this paper we analyze SCR specifications with a two-step approach for verifying safety properties that are either invariants on a single state or a pair of consecutive states. The first step of our approach is to transform an SCR mode transition table to an equivalent form in AMN. The transformation procedure is straight forward; our prototype tool is a few hundred lines of C. The second step is to analyze the AMN specification using the B-Toolkit and verify the maintenance of the invariants. The results obtained for an example SCR mode transition table are consistent with those reported for model checking approaches.

Since our approach is based on theorem proving, it does not suffer from the state explosion problem inherent in model checking approaches. The main drawback of theorem proving is its inherent difficulty. However, the proof obligations necessary for ensuring safety properties in SCR specifications have a relatively uniform structure, and it is possible that theorem proving approaches are well suited to this task. Our experience with the example SCR specification with the relatively weak theorem prover in the B Toolkit is a positive indication in this direction.

The number of proof obligations to be discharged is one measure of the complexity of our approach. The number of proof obligations generated is polynomial in the various parameters of the mode transition table, which makes our method a candidate for scaling to large systems.

There are additional advantages to using the B-Toolkit. The specification animators in the B-Toolkit help check if the specification conforms to the requirements. The B-Toolkit also provides rapid prototyping facilities. Finally the code generators in the B-Toolkit help to implement the specification. At each stage of the refinement process, the associated proof obligations give assurance that safety and other desirable properties are preserved [HL94]. In future we plan to investigate whether the B-Method is adequate for modeling other aspects of SCR specifications.

References

[AB96] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 280–292, San Diego, CA, January 1996.

[AG93] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[Atl94] J. M. Atlee. Native model-checking of SCR requirements. In *In Proceedings of the Fourth International SCR Workshop*, November 1994.

[BBF] M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-Style specifications. In Preparation.

[BCD86] M. Browne, E. Clarke, and D. Dill. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, C-35(12):1035–1044, December 1986.

[Bro89] M. Browne. *Automated Verification of Finite State Machines Using Temporal Logic.* PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1989.

[CES86] E. 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.

[FFK+94] S. R. Faulk, L. Finneran, J. Kirby Jr., S. Shah, and J. Sutton. Experience applying the CoRE method to the lockheed C-130J. In *Proceedings of the 9th Annual Conference on Computer Assurance*, pages 3–8, Gaithersburg, MD, June 1994.

[Hen80] K. L. Heninger. Specifying software requirements for complex systems. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.

[HJL96] 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.

[HL94] H. Haughton and K. Lano. Testing and safety analysis of AMN specifications. In D. Till, editor, *Proceedings of the Refinement Workshop*, pages 69–79, London, January 1994.

[HM83] C. L. Heitmeyer and J. Mclean. Abstract requirements specifications: A new approach and its application. *IEEE Transactions on Sofware Engineering*, SE-9(5):580–589, September 1983.

[Jin96] Z. Jin. Deriving mode invariants from scr specifications. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, pages 515–521, Montreal, Canada, October 1996.

[McM92] K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

[ORS95] S. Owre, J. Rushby, and N. Shankar. Analyzing tabular and state-transition requirements specifications in PVS. Technical Report CSL-95-12, Computer Science Laboratory SRI International, June 1995. Revised April, 1996.

[ORSvH95] S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Pra95] C.H. Pratten. An introduction to proving AMN specifications with PVS and the AMN-PROOF tool. In Henri Habrias, editor, *Proceedings of Z Twenty Years on – What is its Future*, pages 149–165. IRIN-IUT de Nantes, October 1995.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[SA96] T. Sreemani and J. M. Atlee. Feasibility of model checking software requirements: A case study. In *COMPASS '96: Proceedings of the Eleventh Annual Conference on Computer Assurance*, pages 77–88, Gaithersburg, MD, June 1996.

[SH94] A. C. Storey and H. Haughton. A strategy for the production of verifiable code using the B method. In *FME '94 : Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 346–365. Springer-Verlag, 1994.

[vSPM93] A. J. van Schouwen, D. L. Parnas, and J. Madey. Documentation of requirements for computer systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 198–207, San Diego, CA, January 1993. IEEE Computer Society Press.