A Logic-Model Semantics for SCR Software Requirements

Joanne M. Atlee[*] Michael A. Buckley

Department of Computer Science
University of Waterloo
Waterloo, Ontario

Abstract. This paper presents a simple logic-model semantics for Software Cost Reduction (SCR) software requirements. Such a semantics enables model-checking of native SCR requirements and obviates the need to transform the requirements for analysis. The paper also proposes modal-logic abbreviations for expressing conditioned events in temporal-logic formulae. The Symbolic Model Verifier (SMV) is used to verify that an SCR requirements specification enforces desired global requirements, expressed as formulae in the enhanced logic. The properties of a small system (an automobile cruise control system) are verified, including an invariant property that could not be verified previously. The paper concludes with a discussion of how other requirements notations for conditioned-event-driven systems could be similarly checked.

Keywords: Reactive Systems, Software Requirements, Formal Semantics, Model Checking.

1 Introduction

Precise notations have been developed to specify unambiguous requirements. The use of these notations helps to ensure that the requirements designer considers and documents all cases of appropriate system behavior. Of these notations, the Software Cost Reduction (SCR) notation [1, 10, 11], Statecharts [7], and the Requirements State Machine Language (RSML) [14] notation have received considerable attention because they have been used to specify the software requirements of large, real-world applications (the A-7E aircraft, the avionics system for the Lavi fighter aircraft, and the Traffic Alert and Collision Avoidance System (TCAS), respectively). While there are many syntactic and semantic differences among these notations, all three were developed to specify *conditioned-event-driven*, reactive systems; such systems are controlled by events conditioned on the values of other system variables.

A previously proposed technique [3] for model checking conditioned-event-driven software requirements entails transforming the requirements into an 'equivalent' Moore machine representation [12] and analyzing this alternate representation using the MCB model checker [4]. The Moore machine representation does not accurately model the semantics of SCR conditioned events. As a result, the technique is not complete: some properties of the specification may not be verifiable.

This paper demonstrates how SCR software requirements can be translated into logic formulae that encode a temporal logic model. Such a logic model can be analyzed with the Symbolic Model Verifier (SMV) [5, 15], which determines whether a set of Computational Tree Logic (CTL) [6] formulae are modeled by the logic model. The paper also introduces modal logic abbreviations for representing SCR conditioned events in CTL formulae.

Properties of an existing SCR requirements specification (for an automobile cruise control system) are expressed as formulae in the extended logic and are verified with respect to the system's SCR software requirements using the SMV model checker. This specification was analyzed previously, using the state-based analysis approach mentioned above. Symbolic analysis of the system reveals the same discrepancies between the systems' properties and requirements as were revealed using the state-based approach; and the same corrections to the requirements resolves the discrepancies. In addition, the symbolic analysis is able to successfully specify and verify a system invariant of the cruise control specification that could not be verified using the state-based approach.

The paper concludes with a discussion of how other requirements notations for conditioned-event-driven systems could be analyzed symbolically.

[*]This work was supported by the Natural Sciences and Engineering Research Council under grant OGP0155243.

2 Operational Semantics of SCR

Before describing a logic-model semantics for the SCR requirements notation, let us first review its operational semantics. The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of a general Software Cost Reduction project [1, 11]. A complete SCR requirements specification describes behavioral, functional, precision, and timing requirements of the software system under development. The logic and logic-model semantics presented in this paper are for analysis of behavioral requirements only.

2.1 SCR Behavioral Requirements

An SCR requirements specification models a system-to-be-developed as a set of conditioned-event-driven, state-transition machines. The machines' environment is abstracted as a set of *monitored state variables* and *controlled state variables* [16, 17]. Monitored variables model environment phenomena that affect the behavior of the machines, whereas controlled variables model environment phenomena that are controlled (i.e., modified) by the machines. For example, a thermostat that regulates the air temperature of a room would have monitored variables for the temperature setting, the actual room-temperature, and the position of the On/Off switch; and it would have controlled variables for the power switches of the furnace and air conditioner.

The input language of each machine is a set of *conditioned events.* A *condition* is a predicate on monitored variables, a *primitive event* is a change in the value of a condition, and a *conditioned event* is a primitive event whose occurrence depends on the values of other conditions [10]. Let conditions SwitchIsOn and TooCold represent the predicates [On/Off switch = On] and [RealTemp < (SetTemp -3°C)], respectively. Primitive events @T(SwitchIsOn) and @F(SwitchIsOn) represent condition SwitchIsOn *becoming true* and *becoming false*, respectively. Conditioned event

@T(TooCold) WHEN [SwitchIsOn]

describes the event of condition TooCold *becoming true* while condition SwitchIsOn *remains true.* Formally, conditioned event @T(TooCold) WHEN [SwitchIsOn] occurs at time t if and only if primitive event @T(TooCold) occurs at time t and condition SwitchIsOn is true for some non-zero interval of time leading upto and including time t [1].[1] In the above event, TooCold is called the *triggering event* and SwitchIsOn is called the event's *WHEN condition.* Since a primitive event is a type of conditioned event (i.e., a conditioned event with vacuously true WHEN conditions), the term *conditioned event* will henceforth refer to both primitive and conditioned events.

The state of the monitored environment is defined by the current values of the conditions. The state space is the set of possible combinations of conditions' values. Because the behavior of the system-under-development is rarely affected by the values of all the conditions at once, the state space is partitioned into sets of states, called *modes of operation* or simply *modes.* A *mode class* defines a set of modes that partition the monitored environment's state space. A subset of these modes are designated possible *initial modes*; the initial state of the environment uniquely determines which of these is the initial mode when the system starts to execute. Transitions between pairs of modes (called *mode transitions*) are activated by conditioned events. If a conditioned event can trigger two or more transitions from the same mode, then the mode class is non-deterministic.

Figure 1 is an SCR requirements specification of a simple thermostat. The top of the mode transition table is labeled with conditions that affect the thermostat's behavior: SwitchIsOn, TooCold, TooHot [(SetTemp + 3°C) < RealTemp], and TempOk [(SetTemp - 3°C) $\leq$ RealTemp $\leq$ (SetTemp + 3°C)]. System behavior is expressed as four modes of operation: OFF, INACTIVE (the thermostat is regulating the room's air temperature, but the temperature does not need adjusting), HEAT (the thermostat is trying to heat the air), and AC (the thermostat is trying to cool the air). Below the mode transition table is the specification of the system's initial mode, whose value may depend on the values of environmental conditions when the system is started.

The specification of a mode class's transition relation has a tabular format. Each row in the table specifies a conditioned event that activates a transition from the mode on the left to the mode on the right. A table entry of "@T" (or "@F") under column C1 represents a triggering event @T(C1) (or @F(C1)); a table entry of "t" (or "f") represents a WHEN condition WHEN[C1] (or WHEN[$\sim$C1]). If the value of a condition C1 does not affect the occurrence of a conditioned event, then the table entry is marked with a hyphen ("–"). For example, the first row in Figure 1 specifies a transition from mode OFF to mode HEAT due to the occurrence of conditioned event

@T(SwitchIsOn) WHEN[TooCold]

[1] SCR semantics actually propose three definitions for a conditioned event occurrence and allow the requirements designer to decide which definition best suits each event [1]. For a given conditioned event @T(C1) WHEN [C2], if primitive event @T(C1) occurs at time t, then the conditioned event occurs at time t if and only if there exists a time interval $\epsilon > 0$ such that

1) C2 is true throughout closed interval [$t-\epsilon$, t], or
2) C2 is true throughout interval [t, $t+\epsilon$], or
3) C2 is true throughout interval [$t-\epsilon$, $t+\epsilon$]

This paper uses the first definition for all conditioned event occurrences.

MODECLASS Thermostat:

Current Mode	SwitchIsOn	TooCold	TempOk	TooHot	**New Mode**
Off	@T	t	–	–	Heat
	@T	@T	–	–	
	@T	–	t	–	Inactive
	@T	–	@T	–	
	@T	–	–	t	AC
	@T	–	–	@T	
Inactive	@F	–	–	–	Off
	t	@T	–	–	Heat
	t	–	–	@T	AC
Heat	@F	–	–	–	Off
	t	–	@T	–	Inactive
AC	@F	–	–	–	Off
	t	–	@T	–	Inactive

Initial Mode: Off (~SwitchIsOn)
Inactive (SwitchIsOn & TempOk)
Heat (SwitchIsOn & TooCold)
AC(SwitchIsOn & TooHot)

Assumptions:
TooCold – TempOk – TooHot

Goals:
If mode=Off, then ~SwitchIsOn
If mode=Inactive, then SwitchIsOn and TempOk
If mode=Heat, then SwitchIsOn and TooCold
If mode=AC, then SwitchIsOn and TooHot
If mode=Off, then ~@T(SwitchIsOn)

Figure 1: SCR behavioral specification of simple thermostat.

If during time interval $[t - \epsilon, t)$ the system is in mode OFF, the switch is in the Off position and the temperature is too cold; and if at time t the switch is moved to the On position while the temperature remains too cold; then the system is in mode HEAT at time t.

An *SCR behavioral requirements document* contains the specification of one or more mode classes. At all times, the system is in exactly one mode of each mode class. Each mode class specifies one aspect of the system's behavior, and the system's global behavior is defined to be the composition of the specification's mode classes. The resultant composite is called the *reachability graph* of the original specification.

2.2 Environmental Assumptions

An SCR requirements document also specifies any assumptions of the behavior of the environment. Similar to the NAT relation in Parnas's 4-variable model of system requirements [16], an *assumption* specifies constraints on the values of conditions, imposed by laws of nature or by other mode classes in the system. As such, assumptions are invariant constraints that hold in all states of the specification's reachability graph.

The syntax and semantics of assumption specifications are described in [2]; for the purposes of this paper, symbol "|" denotes an exclusive-or operator, "–" denotes an ordering on conditions related by exclusive-or[2], and "$->$" denotes implication. The assumption specified in Figure 1 states that exactly one of {TooCold, TempOk, TooHot} is true at all times, and that the temperature cannot rise (or fall) instantaneously from TooCold to TooHot (or vice versa).

2.3 System Goals

Finally, an SCR requirements specification often includes a set of *goals* that the system is required to meet. These goals are not additional constraints on the required behavior; it is expected that the SCR tabular specification enforces these goals. Specified goals are redundant information that are included in the specification because the reader might not deduce these properties from the tabular specifications. Most of the goals specified in the thermostat example are mode invariants:

If the system is in mode OFF, *then condition SwitchIsOn is false.*

[2]Assumptions relating conditions by "|" and "–" both specify enumerations. The difference is that conditions related by "–" are constrained as to when they can become true: the *ith* condition in a "–" enumeration can become true only if either the $i - 1st$ or the $i + 1st$ condition (if it exists) is becoming false.

Other goals express global behavioral requirements on the occurrence of an event:

Event @T(SwitchIsOn) cannot occur within mode Off.

3 Logic-model Semantics of SCR

This section presents a logic-model representation of SCR behavioral requirements that precisely models the operational semantics described above. Such a logic-model semantics can facilitate the development of mathematically sound and complete techniques for analyzing and verifying SCR requirements.

3.1 SCR logic model

System modes and environmental conditions can be represented by propositional variables. A propositional variable representing a condition is *true* if and only if the condition is *true*. Similarly, a propositional variable representing a mode is *true* if and only if the mode is the current mode of its mode class. An *interpretation* maps propositional variables to truth values. Because the values of conditions and current modes vary over time, the logic model consists of a *set* of interpretations and a transition relation among the interpretations.

Formally, a logic model of an SCR mode class is a tuple $\langle S, S_0, V, I, R\rangle$, where

- S is a finite set of states, where each state $s \in S$ represents a distinct interpretation.
- $S_0 \subset S$ is the set of possible initial states
- V is a finite set of propositional variables representing the specification's modes and environmental conditions.
- $I : S \rightarrow 2^V$ is the interpretation function that assigns truth values to propositional variables in each state.
- $R \subseteq (S \times S)$ is a binary transition relation on S.

We express the logic model of an SCR requirements specification as a set of temporal logic formulae. We will use symbols $\neg$, $\wedge$, $\vee$, and $\rightarrow$ to represent logic connectives 'not', 'and', 'or', and 'implies' respectively. Because we will need to refer to the values of modes and conditions in pairs of consecutive states, we let n be an unary modal-logic *nextstate* operator: $n(\text{C1})$ is *true* in the current state if and only if variable C1 is *true* in the next state.

SCR behavioral requirements

An SCR requirements specification defines a set of initial modes and a mode transition relation. Since a mode is an abstraction of a set of states, the set of initial modes represents a set of initial states and the mode transition relation maps a set of states to a set of successor states.

SCR behavioral requirements can be encoded as the conjunction of several temporal logic formulae. The set of initial states S_0 is represented by a formula, over conditions and modes, that is *true* with respect to the interpretations of states belonging to S_0. Consider the SCR requirements specification presented in Figure 1. The initial mode of the system depends on the initial values of the environmental conditions. The following formula specifies the logic formula that holds in the initial state S_0.

$$\begin{aligned} S_0 \;\rightarrow\; & ((\neg\text{SwitchIsOn} \rightarrow \text{Off}) \;\wedge \\ & ((\text{SwitchIsOn} \wedge \text{TempOk}) \rightarrow \text{Inactive}) \;\wedge \\ & ((\text{SwitchIsOn} \wedge \text{TooCold}) \rightarrow \text{Heat}) \;\wedge \\ & ((\text{SwitchIsOn} \wedge \text{TooHot}) \rightarrow \text{AC})) \end{aligned}$$

The transition relation R is expressed as a set of formulae must *true* with respect to the interpretations of states belonging to the transitions' source and destination modes. Each row in the mode transition table is expressed as a logic implication whose antecedent is the current mode and conditioned event, and whose consequent is the specified next mode. The conditioned event is modeled as a conjunction of conditions in the current and next states: the event's triggering conditions are unsatisfied in the current state, the triggering conditions are satisfied in the next state, and the event's WHEN conditions are satisfied in both states. For example, the first row in the mode transition table is represented by the following formula.

$$\begin{aligned} (\text{Off} \wedge \neg\text{SwitchIsOn} \wedge \text{TooCold} \wedge n(\text{SwitchIsOn}) \\ \wedge\; n(\text{TooCold})) \rightarrow n(\text{Heat}) \end{aligned}$$

An SCR mode transition table also states, implicitly, that if an occurring conditioned event does *not* trigger any of the transitions leaving the current mode, then the current mode remains the same. To capture this latter behavior, we add a formula to R for each mode, which specifies that if the mode is *true* in the current state and if none of the transitions leaving the mode is activated, then the system remains in the current mode. The following formula describes the conditions under which the thermostat remains in mode Heat.

$$\begin{aligned} (\text{Heat} \wedge \neg(\text{SwitchIsOn} \wedge n(\neg\text{SwitchIsOn})) \;\wedge \\ \neg(\text{SwitchIsOn} \wedge \neg\text{TempOk} \wedge n(\text{SwitchIsOn}) \\ \wedge\; n(\text{TempOk}))) \;\rightarrow\; n(\text{Heat}) \end{aligned}$$

The transition relation R is the conjunction of the mode transitions' formulae. If an SCR specification consists of several mode classes, then the specification's transition relation is the conjunction of the mode classes' transition relations.

The disjunction of the antecedents of all the implications encoding a mode transition relation forms a tautology. Thus, the transition relation of each mode class is total. Figure 2 shows part of the logic model for the SCR requirements of the thermostat specified in Figure 1.

$$
\begin{array}{lll}
S_0 & \rightarrow & ((\neg \text{SwitchIsOn} \rightarrow \text{Off})\, \mathit{wedge}\, ((\text{SwitchIsOn} \wedge \text{TempOk}) \rightarrow \text{Inactive}) \wedge \\
 & & ((\text{SwitchIsOn} \wedge \text{TooCold}) \rightarrow \text{Heat}) \wedge ((\text{SwitchIsOn} \wedge \text{TooHot}) \rightarrow \text{AC})) \\
\bigwedge & & \\
(& \multicolumn{2}{l}{((\text{Off} \wedge \neg \text{SwitchIsOn} \wedge \text{TooCold} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooCold})) \rightarrow n(\text{Heat})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Off} \wedge \neg \text{SwitchIsOn} \wedge \neg \text{TooCold} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooCold})) \rightarrow n(\text{Heat})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Off} \wedge \neg \text{SwitchIsOn} \wedge \text{TempOk} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TempOk})) \rightarrow n(\text{Inactive})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Off} \wedge \neg \text{SwitchIsOn} \wedge \neg \text{TempOk} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TempOk})) \rightarrow n(\text{Inactive})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Off} \wedge \neg \text{SwitchIsOn} \wedge \text{TooHot} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooHot})) \rightarrow n(\text{AC})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Off} \wedge \neg \text{SwitchIsOn} \wedge \neg \text{TooHot} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooHot})) \rightarrow n(\text{AC})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Off} \wedge \neg(\neg \text{SwitchIsOn} \wedge \text{TooCold} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooCold})) \wedge} \\
 & \multicolumn{2}{l}{\qquad \neg(\neg \text{SwitchIsOn} \wedge \neg \text{TooCold} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooCold})) \wedge} \\
 & \multicolumn{2}{l}{\qquad \neg(\neg \text{SwitchIsOn} \wedge \text{TempOk} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TempOk})) \wedge} \\
 & \multicolumn{2}{l}{\qquad \neg(\neg \text{SwitchIsOn} \wedge \neg \text{TempOk} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TempOk})) \wedge} \\
 & \multicolumn{2}{l}{\qquad \neg(\neg \text{SwitchIsOn} \wedge \text{TooHot} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooHot})) \wedge} \\
 & \multicolumn{2}{l}{\qquad \neg(\neg \text{SwitchIsOn} \wedge \neg \text{TooHot} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TooHot}))) \rightarrow n(\text{Off})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Heat} \wedge \text{SwitchIsOn} \wedge n(\neg \text{SwitchIsOn})) \rightarrow n(\text{Off})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Heat} \wedge \text{SwitchIsOn} \wedge \neg \text{TempOk} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TempOk})) \rightarrow n(\text{Inactive})) \wedge} \\
 & \multicolumn{2}{l}{((\text{Heat} \wedge \neg(\text{SwitchIsOn} \wedge n(\neg \text{SwitchIsOn})) \wedge} \\
 & \multicolumn{2}{l}{\qquad \neg(\text{SwitchIsOn} \wedge \neg \text{TempOk} \wedge n(\text{SwitchIsOn}) \wedge n(\text{TempOk}))) \rightarrow n(\text{Heat})) \wedge} \\
 & \multicolumn{2}{l}{\ldots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)}
\end{array}
$$

Figure 2: SCR logic model of thermostat.

Environmental Assumptions

The formula in Figure 2 represents the *unconstrained* transition relation of the thermostat specification. This transition relation allows all possible assignments of truth values to the system's conditions and modes. For example, there are no constraints on the number of modes that can be *true* in any logic-model state. Furthermore, the SCR specification's environmental assumptions are not represented in the logic model.

Environmental assumptions can be represented as logic formulae:

$$
\begin{array}{ll}
\bigwedge & (\text{Off} \vee \text{Heat} \vee \text{Inactive} \vee \text{AC}) \\
\bigwedge & (\text{Off} \rightarrow (\neg \text{Heat} \wedge \neg \text{Inactive} \wedge \neg \text{AC})) \\
\bigwedge & (\text{Heat} \rightarrow (\neg \text{Off} \wedge \neg \text{Inactive} \wedge \neg \text{AC})) \\
\bigwedge & (\text{Inactive} \rightarrow (\neg \text{Off} \wedge \neg \text{Heat} \wedge \neg \text{AC})) \\
\bigwedge & (\text{AC} \rightarrow (\neg \text{Off} \wedge \neg \text{Heat} \wedge \neg \text{Inactive})) \\
\bigwedge & (\text{TooCold} \vee \text{TempOk} \vee \text{TooHot}) \\
\bigwedge & (\text{TooCold} \rightarrow (\neg \text{TempOk} \wedge \neg \text{TooHot})) \\
\bigwedge & ((\neg \text{TooCold} \wedge n(\text{TooCold})) \rightarrow \\
 & \qquad (\text{TempOk} \wedge n(\neg \text{TempOk}))) \\
\bigwedge & \\
 & \ldots
\end{array}
$$

Because the environmental assumptions hold invariantly and constrain the specification's transition relation, each of the assumptions' representative formulae is conjoined with the formula representing the mode transition table.

3.2 Operational vs. logic-model semantics

A logic-model representation of an SCR requirements specification deviates from its operational semantics in two respects. First, SCR operational semantics forbids the simultaneous occurrence of two or more primitive events, because its operational model assumes that the implemented system will react to events one at a time, regardless of when the events occur. Traditionally, implementations of reactive systems queue primitive events as their occurrences are detected; when a primitive event reaches the head of the queue, the system checks the values of other conditions to determine if a significant conditioned event (e.g., one that might activate a mode transition) has occurred [8].

An unconditional restriction on the occurrence of simultaneous events may violate environmental assumptions. For example in the thermostat example, it is assumed that (always) exactly one of the conditions TooCold, TempOk, and TooHot is *true*, and that the temperature cannot rise (or fall) instantaneously from TooCold to TooHot (or vice versa). From this assumption, one should be able to infer that the occurrence of event @T(TooCold) implies the simultaneous occurrence of @F(TempOk). If the above inference is invalid due to constraints on simultaneous events, then one would not be able to infer from the verified goal

If the system is in mode HEAT*, then TooCold is true.*

that the following goal is also true

If the system is in mode HEAT*, then TempOk is false.*

Rather than incorporating a restriction on simultaneous events into the semantics of an SCR logic model, we express the restriction as an environmental assumption of the specification. This arrangement allows one to specify and analyze a system that does not assume events will be reported sequentially (such as the thermostat example). In addition, it allows the specification of tailored restrictions that model both the sequential occurrence of independent events and the simultaneous occurrence of related events.

The second difference between operational and logic-model semantics of SCR specifications pertains to the value of WHEN conditions at the time of a conditioned event. In an SCR logic-model, WHEN conditions must be satisfied both immediately before and during the occurrence of a conditioned event. According to the latest operational semantics of SCR, a WHEN condition must be satisfied immediately before the occurrence of a conditioned event, but its value at the time of the event is unknown [10, 9]. Given the above restriction on the occurrence of simultaneous events, one can to infer the value of most WHEN conditions: WHEN conditions that are unrelated to the triggering event have the same value during the event as they had immediately before the event. However, WHEN conditions that are related to the triggering event may or may not be changing value along with the triggering event.

We chose to model the older conditioned-event semantics [1] because it is easier to write a correct SCR specification given these semantics. If one wants to write a specification that guarantees a particular formula f is always *true* in a particular mode Mode, one needs to specify that f is always *true* upon entry into Mode and that the system always exits mode Mode if f becomes *false*. Such a specification is difficult (if not impossible) to write if one cannot infer from a WHEN condition WHEN[f] on a transition into the mode that f is true when the mode is entered.

4 Model Checking

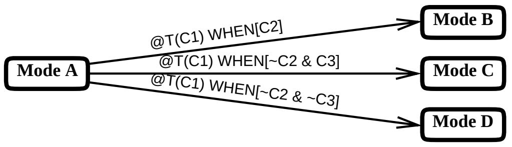
In the previous section, we showed how an SCR behavioral specification and its environmental assumptions can be represented as a logic model, where the logic model is expressed as a temporal logic formula. If the goals of an SCR-specified system can be expressed as logic formulae, then a model checker can automatically verify that the intended goals are modeled by the SCR specification:

$\forall \boldsymbol{s_i} \in \boldsymbol{S_0}$:
  $\boldsymbol{s_i}$, **Assumptions** $\wedge$ **SCR requirements** $\models$ **Goals**

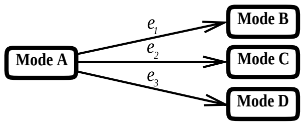
That is, the goals are evaluated with respect to a state's interpretation, environmental assumptions, and behavioral requirements; and the model checker verifies that the goals are true at every initial state.

4.1 Previously proposed techniques

A number of researchers have suggested that SCR requirements specifications be translated into a notation for purely event-driven systems (e.g., a process algebra) and that this alternate representation be verified using a model checker for event-driven specifications. If a conditioned-event-driven specification is transformed into an event-driven specification, then a unique event name must be created for each distinct conditioned event in the original specification. In such a transformation, relationships between conditioned events are lost. Consider the following set of mode transitions, in which triggering event @T(C1) occurs under different conditions.



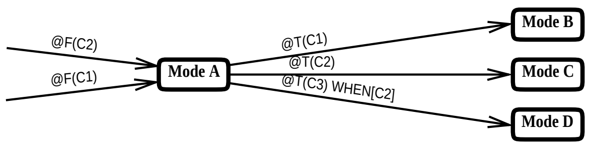
The result of transforming the above conditioned-event-driven specification into an event-driven specification appears below.



A side effect of renaming each conditioned-event is that the components of the original conditioned-events are no longer represented in the specification. As a result, one cannot verify invariant properties that refer to the events' individual components. In the original specification, the occurrence of event @T(C1) in ModeA causes the system to leave ModeA; this property cannot be verified in the transformed specification.

As an alternative [2, 3], it has been proposed that SCR requirements specifications be translated into a Moore machine representation [12] and that this alternate representation be verified using the MCB model checker [4]. States are annotated with a set of current modes (one for each mode class), and state transitions are annotated by the conditions that enable mode transitions. MCB evaluates formulae with respect to the annotations of a state and one of the state's exiting transitions. For a formula to be invariantly true, it must be true with respect to the annotations of all the machine's <state, transition> pairs. The problem with this approach is that such a representation models an SCR specification's *mode-space*, whereas the model checker was developed to analyze a specification's *state-space*.

Consider the following set of mode transitions.



An invariant property of the specification is

- *If the system is in mode* ModeA, *then either C1 or C2 is false.*

MCB would evaluate this formula with respect to the above specification as not invariantly *true*. Mode

ModeA paired with the mode transition from ModeA to ModeD does not enforce this property, since C1 and C2 can be *true* and still satisfy the mode transition's conditioned event. However, an exploration of the specification's state space would show that a state in which mode ModeA and conditions C1 and C2 are all *true* is not reachable: one of C1 or C2 is always *false* upon entry into mode ModeA, and the event of either condition becoming *true* activates a transition leaving ModeA.

The above technique attempts to compensate for the lack of state-space information by modeling the conditions under which a system remains in a mode. The invariant of each mode is calculated and added to all transitions exiting the mode. In addition, a self-looping transition is created for each mode and is annotated with the mode's calculated invariant. Thus, an evaluation of a mode invariant would find that all transitions leaving the mode's state are annotated with the calculated invariant; and evaluation would determine that the specification satisfies the invariant. However, this technique is only capable of deriving mode invariants that are conjunctions of environmental conditions and negated environmental conditions; it cannot derive arbitrary mode invariants.

4.2 Model checking using SMV

The Symbolic Model Verifier (SMV) [5, 15] accepts as input a system's behavioral specification and a set of logic formulae, and determines whether or not the logic formulae hold in the specification. An SMV behavioral specification consists of the specification's initial state and its transition relation (see Figure 3). The representation of an SMV behavioral specification is equivalent to an SCR logic model. Thus, SMV analysis of an SCR logic model is both sound and complete.

SMV representation of an SCR logic model

An SCR requirements specification, including environmental assumptions and goals to be verified, is automatically translated into an SMV specification.

The variables representing the specification's modes and conditions are defined in the VAR section. A boolean variable is declared for each condition, and an enumeration variable is declared for each mode class[3]. The values of a mode class variable are the names of the class's modes. Variable *pSwitchIsOn* will be discussed below.

The SCR logic model is defined in the ASSIGN section. The system's initial conditions are declared using init() statements. The logic model's mode transition relations are translated into next() statements that assign the next values of the mode class variables. Because the next value of a mode class variable depends on the current and next values of other variables, this assignment is specified using a case expression. Each branch in a case expression consists of a boolean expression and a value, separated by a colon. If a branch's expression is *true*, then the assigned variable has the associated value in the next state. A final branch with boolean expression 1 is equivalent to an else clause; such a boolean expression is evaluated *true* if none of the above branches' expressions is satisfied.

Each row in an SCR mode transition table is translated into a branch in the case expression of the mode class variable's next() statement. The branch's boolean expression specifies the current mode and the occurrence of a conditioned event; and the next value specifies the mode in the next state. The else clause at the end of the case expression represents the case where a conditioned event occurs but does not activate a mode transition.

next() and case statements are also used to express environmental assumptions. Assignments to environmental variables (that specify a set of allowable next values, rather than assigning *the* next value of the variable) enumerate the possible sequences of variable values. These assignments constrain the SCR mode transition relations. In the thermostat example, next() and case statements are used to restrict the possible next value of variable Enum1, which models the assumption that the environment cannot change instantaneously from being TooCold to TooHot (or vice versa).

SMV representation of system goals

The SPEC section lists the formulae to be model checked against the SMV behavioral specification; the SPEC formulae are expressed in the Computational Tree Logic (CTL) branching-time temporal logic [6]. Branching-time temporal formulae are evaluated with respect to a particular state in the logic model based on the set of paths emanating from the state. Because *the* future path of the system's execution is unknown, temporal operators are quantified over the set of possible futures (e.g., a property p is true in *some* next state or in *all* next states).

The syntax and semantics for CTL formulas are defined in [6] and are simply summarized below:

1. Every propositional variable is a CTL formula.
2. If f and g are CTL formulae, then so are: $!f$, $f\&g$, $f \mid g$, $f \rightarrow g$, AXf, EXf, $A[fUg]$, $E[fUg]$, AFf, EFf, AGf, EGf.

The symbols ! (*not*), & (*and*), | (*or*), and $\rightarrow$ (*implies*) are logical connectives and have their usual meanings. The temporal logic formulae are evaluated with respect to the current state in the logic model. For example X is the *nextstate* operator, and formula $EX\phi$ ($AX\phi$) is

[3]Conditions that are assumed to be related by exclusive-or are also represented by an enumeration variable.

```
MODULE main
VAR
  Thermostat : {Off, Inactive, Heat, AC};
  Enum1 : {TooCold, TempOk, TooHot};
  SwitchIsOn : boolean;
  pSwitchIsOn : boolean;
ASSIGN
  init(Thermostat) := Off;
  init(SwitchIsOn) := 0;
  next(Thermostat) := case
    Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TempOk & next(Enum1=TempOk) : Inactive;
    Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & !Enum1=TempOk & next(Enum1=TempOk) : Inactive;
    Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TooCold & next(Enum1=TooCold) : Heat;
    Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & !Enum1=TooCold & next(Enum1=TooCold) : Heat;
    Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & Enum1=TooHot & next(Enum1=TooHot) : AC;
    Thermostat = Off & !SwitchIsOn & next(SwitchIsOn) & !Enum1=TooHot & next(Enum1=TooHot) : AC;
    Thermostat = Inactive & SwitchIsOn & next(!SwitchIsOn) : Off;
    Thermostat = Inactive & !Enum1=TooCold & next(Enum1=TooCold) : Heat;
    Thermostat = Inactive & !Enum1=TooHot & next(Enum1=TooHot) : AC;
    Thermostat = Heat & SwitchIsOn & next(!SwitchIsOn) : Off;
    Thermostat = Heat & !Enum1=TempOk & next(Enum1=TempOk) : Inactive;
    Thermostat = AC & SwitchIsOn & next(!SwitchIsOn) : Off;
    Thermostat = AC & !Enum1=TempOk & next(Enum1=TempOk) : Inactive;
    1 : Thermostat;
  esac;
  init(pSwitchIsOn) := 0; next(pSwitchIsOn) := SwitchIsOn;
  next(Enum1) := case
    Enum1=TooCold : {TooCold, TempOk};
    Enum1=TooHot : {TooHot, TempOk};
    Enum1=TempOk : {TooCold, TempOk, TooHot};
  esac;
SPEC
  AG((Thermostat=Off)->!SwitchIsOn)
  AG((Thermostat=Inactive) -> (SwitchIsOn & Enum1=TempOk))
  AG((Thermostat= Heat) -> (SwitchIsOn & Enum1=TooCold))
  AG((Thermostat= AC) -> (SwitchIsOn & Enum1=TooHot))
  AG((Thermostat=Off) -> (!SwitchIsOn | pSwitchIsOn))
  AG((!pSwitchIsOn & SwitchIsOn) -> !(Thermostat=Off))
```

Figure 3: SMV specification of thermostat.

true in state s_i if formula ϕ is true in some (in every) successor state of s_i. U is the *until* operator, and formula $E[\phi\ U\ \psi]$ $(A[\phi\ U\ \psi])$ is true in state s_i if along some (every) path emanating from s_i there exists a future state s_j at which ψ holds and ϕ is true until state s_j is reached. F is the *future* operator, and $EF\phi$ $(AF\phi)$ is true in state s_i if along some (every) path from s_i there exists a future state in which ϕ holds. Finally, G is the *global* operator, and $EG\phi$ $(AG\phi)$ is true in state s_i if ϕ holds in every state along some (every) path emanating from s_i.

The following are some of the thermostat system's goals, expressed as CTL formulae:

$$AG(Off \rightarrow !SwitchIsOn)$$
$$AG(Heat \rightarrow (SwitchIsOn \wedge TooCold))$$
$$AG(Inactive \rightarrow (SwitchIsOn \wedge TempOk))$$
$$AG(AC \rightarrow (SwitchIsOn \wedge TooHot))$$

All of the above formulae state properties that hold invariantly when the system is in a particular mode. For example, if the system is in mode HEAT it is invariantly *true* that the SwitchIsOn and the temperature is TooCold.

The following goal of the thermostat example is more difficult to express as a CTL formula because it refers to the occurrence of a conditioned event:

Event @T(SwitchIsOn) cannot occur within mode OFF.

Since conditioned events are represented by consecutive logic-model states, a CTL formula referring to the occurrence of a conditioned event must refer to the values of the event's conditions in two states.

$$AG(!SwitchIsOn \rightarrow !EX(SwitchIsOn \wedge Off))$$

If SwitchIsOn is *false*, there is no next state in which SwitchIsOn is becoming *true* and mode Off is *true*.

To ease the phrasing of CTL formulae that refer to the occurrence of conditioned events, we propose unary logic connectives @T, @F and WHEN to express propositional formulae that are *becoming true*, *becoming false*, and maintaining a particular value, respectively. Presumably, we would simulate the new @T, @F, and WHEN connectives using their definitions: evaluation of their operands in the current and next states. SMV, however, can only check formulae with respect to the current values of variables. Thus, we simulate the @T, @F and WHEN connectives by evaluating their operands in the previous and current states; additional SMV variables are declared that record the previous values of the connectives' operands. Let v be an arbitrary SMV variable representing a SCR mode or an environmental condition, and let f be an arbitrary propositional formula (i.e., a CTL formula with no modal operators). The proposed connectives have the following definitions.

@T(v)	iff	!(pv) $\wedge$ v
@F(v)	iff	pv $\wedge$!(v)
WHEN[v]	iff	pv $\wedge$ v
@T(f)	iff	!(pf) $\wedge$ f
@F(f)	iff	pf $\wedge$!(f)
WHEN[f]	iff	pf $\wedge$ f

where pv represents the SMV variable that records the previous value of variable v, and pf is the result of substituting pv_n for each occurrence of v_n in f, for every n. The above invariant is more simply expressed when formulated using the new connectives:

$$AG((@T(SwitchIsOn) \rightarrow !Off))$$

The automated translation from SCR to SMV copies CTL goals expressed in the SCR specification to the SPEC section of the SMV specification. If the CTL goals use any of the proposed SCR connectives, then the translation declares an additional variable pv for each mode or condition variable v that appears as an operand in one of the proposed connectives. For example to check the above invariant, a variable pSwitchIsOn is declared in the thermostat specification whose value is defined to be the value of SwitchIsOn in the previous state: the `next()` value of pSwitchIsOn is set to the current value of SwitchIsOn. In addition, the translation replaces all occurrences of SCR connectives with their definitions: propositional formulae over current and previous-value variables.

Symbolic model checking

Model checkers based on reachability analysis exhaustively test each explicit state in the specification's state space to determine if a goal is modeled by the specification. In symbolic model checking, explicit logic model states are not tested. Rather, the model checker builds a formula that is true in a state if and only if that state satisfies the goal.

If a goal is a propositional formula, then the formula representing the set of states satisfying that goal is the goal itself. Propositional formula ϕ represents the set of all states whose interpretations evaluate ϕ to be *true*. Similarly, propositional formula $\phi \wedge \psi$ represents the set of all states whose interpretations evaluate both ϕ and ψ to be *true*.

If a goal states a property about the next state, then the model checker uses the logic model's transition relation to build the appropriate formula. For example, if formula $EX\phi$ represents the set of states that the transition relation R maps into ϕ, then formula $R^{-1}(\phi)$ (where R^{-1} is the inverse relation of R) represents the set of states in which goal $EX\phi$ holds. Formula $!(R^{-1}(!\phi))$ represents the set of states in which goal $AX\phi$ holds.

All of the other CTL modal operations can be defined as fixed points of the EX and AX operators. For example,

$$EF(f) = minY.(f \vee EX(Y))$$

is a fixed point definition of EF. Y_0 is *false* (the empty set of states); Y_1 is the set of states satisfying f (i.e., the set ot states that can reach a state satisfying f via 0 applications of the transition relation); Y_2 is the set of states $f \vee EX(f)$ (i.e., the set of states that can reach a state satisfying f via 0 or 1 applications of the transition relation); Y_3 is the set of states $f \vee EX(f) \vee EX(EX(f))$ (i.e., the set of states that can reach a state satisfying f via 0, 1, or 2 applications of the transition relation); etc. The computation terminates when the least fixed point is reached; that is after n iterations, where n is the smallest integer such that $Y_{n-1} = Y_n$.

AG has a greatest fixed point definition:

$$AG(f) = maxY.(f \wedge AX(Y))$$

Y_0 is *true* (the set of all states); Y_1 is the set of states satisfying f (i.e., the set of states for which f is true after 0 applications of the transition relation); Y_2 is the set of states satisfying $f \wedge AX(f)$ (i.e., the set of states for which f is *true* after 0 and 1 applications of the transition relation); Y_3 is the set of states satisfying $f \wedge AX(f) \wedge AX(AX(f))$ (i.e., the set of states for which f is *true* after 0, 1, and 2 applications of the transition relation); etc. The computation terminates when the greatest fixed point is reached; that is after n iterations, where n is the smallest integer such that $Y_{n-1} = Y_n$.

All of the formulae in the SPEC section of the thermostat specification have been verified to be *true* with respect to the system's behavioral specification.

5 Case Study

We used the SMV model checker to analyze the software requirements for an automobile cruise control system. This specification had been analyzed previously using a state-based model checker [2, 3], but that technique failed to verify one of the properties of the specification. In this case study, the property is successfully verified.

The SCR specification for the automobile cruise control system was originally presented in [13] and made deterministic in [3]. Figure 4 contains the deterministic version of the specification. The system's conditions indicate whether the ignition is on, the engine is running, the automobile is traveling too fast to be controlled, the

MODECLASS CruiseControl:

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

Initial Mode: Off

Assumptions:

Running –> Ignited
Activate | Deactivate | Resume

Goals:

Off –> !Ignited
!Off –> Ignited
Inactive –> (Ignited & (!Running | !Activate))
Cruise –> (Ignited & Running & !Brake)
Override –> (Ignited & Running)

Figure 4: SCR behavioral specification of the cruise control system.

brake pedal is being pressed, and whether the cruise control lever is set at Activate, Deactivate, or Resume. The modes of the cruise control are: OFF, INACTIVE (ignition is on, but cruise control is not), CRUISE, and OVERRIDE (cruise control is on but is not controlling the vehicle's speed). The system starts in mode OFF.

The set of environmental assumptions at the bottom of the specification describes constraints on the values of the conditions. The first assumption states that the ignition must be on if the engine is running. The second assumption states that exactly one of the conditions representing the cruise control lever position (Activate, Deactivate, or Resume) is true at any time.

The SCR requirements specification for the cruise control includes a set of goals that the SCR tabular specification is expected to enforce[4]. For example, if the system is in mode OFF, then the automobile's ignition must be off. The system invariant for mode INACTIVE is more complex: if the system is in mode INACTIVE, then the ignition is on, and either the engine is not running or cruise control has not been activated.

As stated above, these formulae were analyzed with respect to the SCR specification presented in Figure 4 using a state-based model checker. This technique transformed the SCR conditioned-event-driven specification into a state-based representation, and then model checked this alternate representation using the MCB model checker [4, 6]. The analysis detected several discrepancies between the formulae and the tabular specification. Most of these discrepancies could be resolved by strengthening the conditioned events of some of the mode transitions in the tabular specification: property ~Ignited needed to be added to the system's initial conditions, and WHEN condition ~Toofast needed to be added to all mode transitions into CRUISE.

However, the mode invariant for mode INACTIVE could not be verified. It was argued that the intended invariant was Ignited and either the engine is not running, the brake is on, the car is traveling too fast, or the cruise control lever was activated but not at a time when the other cruise control conditions held. These properties were expressed as the following CTL formula

$$\begin{aligned} AG\,((&\mathit{Inactive} \wedge {\sim}\mathit{Activate} \wedge \mathit{Ignited} \wedge \mathit{Running} \\ &\wedge {\sim}\mathit{Toofast} \wedge {\sim}\mathit{Brake}) \rightarrow \\ &{\sim}EX(\mathit{Inactive} \wedge \mathit{Activate} \wedge \mathit{Ignited} \wedge \mathit{Running} \\ &\wedge {\sim}\mathit{Toofast} \wedge {\sim}\mathit{Brake})) \end{aligned}$$

which states that if the system is in mode INACTIVE,

[4]The third system invariant is actually (Inactive $\rightarrow$ Ignited $\wedge$ ((~Running) $\vee$ (~StartIncr))). StartIncr is true when the driver has held the cruise control lever in the Activate position for a period of time, and therefore StartIncr $\rightarrow$ Activate.

```
MODULE main
VAR
  CruiseControl : {Off, Inactive, Cruise,Override};
  Enum1 : {Activate, Deactivate, Resume};
  Ignited, EngRunning, Toofast, Brake : boolean;
  pEnum1 : {Activate, Deactivate, Resume};
  pIgnited, pEngRunning, pToofast, pBrake : boolean;
ASSIGN
  init(CruiseControl) := Off;
  init(Ignited) := 0;
  init(EngRunning) := 0;

  next(CruiseControl) := case
    CruiseControl=Off & !Ignited & next(Ignited) : Inactive;
    CruiseControl=Inactive & Ignited & next(!Ignited) : Off;
    CruiseControl=Inactive & Ignited & EngRunning & !Toofast & !Brake & !(Enum1=Activate) & next(Ignited & EngRunning & !Toofast &
      !Brake & (Enum1=Activate)) : Cruise;
    CruiseControl=Cruise & Ignited & next(!Ignited) : Off;
    CruiseControl=Cruise & Ignited & EngRunning & next(Ignited & !EngRunning) : Inactive;
    CruiseControl=Cruise & Ignited & !Toofast & next(Ignited & Toofast) : Inactive;
    CruiseControl=Cruise & Ignited & EngRunning & !Toofast & !Brake & next(Ignited & EngRunning & !Toofast & Brake): Override;
    CruiseControl=Cruise & Ignited & EngRunning & !Toofast & !(Enum1=Deactivate) &
      next(Ignited & EngRunning & !Toofast & (Enum1=Deactivate)) : Override;
    CruiseControl=Override & Ignited & next(!Ignited) : Off;
    CruiseControl=Override & Ignited & EngRunning & next(Ignited & !EngRunning) : Inactive;
    CruiseControl=Override & Ignited & EngRunning & !Toofast & !Brake & !(Enum1=Activate) &
      next(Ignited & EngRunning & !Toofast & !Brake & (Enum1=Activate)) : Cruise;
    CruiseControl=Override & Ignited & EngRunning & !Toofast & !Brake & !(Enum1=Resume) &
      next(Ignited & EngRunning & !Toofast & !Brake & (Enum1=Resume)) : Cruise;
    1 : CruiseControl;
  esac;

  init(pEnum1) := Deactivate; next(pEnum1) := Enum1;
  init(pIgnited) := 0;        next(pIgnited) := Ignited;
  init(pEngRunning) := 0;     next(pEngRunning) := EngRunning;
  init(pToofast) := 0;        next(pToofast) := Toofast;
  init(pBrake) := 0;          next(pBrake) := Brake;

  next(Ignited) := case
    next(EngRunning) : 1;
    1 : {0,1};
  esac;

SPEC
  AG((CruiseControl=Off) -> !Ignited)
  AG(!(CruiseControl=Off) -> Ignited)
  AG((CruiseControl=Inactive) -> Ignited)
  AG((CruiseControl=Cruise) -> (Ignited & EngRunning & !Toofast & !Brake & !(Enum1=Deactivate)))
  AG((CruiseControl=Override) -> (Ignited & EngRunning))
  AG((CruiseControl=Inactive & Ignited & EngRunning & !Toofast & !Brake & !(Enum1=Activate)) ->
      !EX(CruiseControl=Inactive & Ignited & EngRunning & !Toofast & !Brake & (Enum1=Activate)))
  AG((CruiseControl=Inactive) -> (Ignited & !(pIgnited & pEngRunning & !pToofast & !pBrake & !(pEnum1=Activate) &
      Ignited & EngRunning & !Toofast & !Brake & (Enum1=Activate))))
```

Figure 5: SMV specification of automobile cruise control system.

and the cruise control WHEN conditions are satisfied but the cruise control lever is not set to Activate; then the system cannot transition to a state in which the system is still in mode INACTIVE with the cruise control lever now set to Activate and the other cruise control conditions still satisfied. If these properties were true in the next state, then the transition from INACTIVE to CRUISE would have been triggered and the system would actually be in mode CRUISE. SCR event operators can be used to express this property more succinctly:

$$AG(Inactive \rightarrow (Ignited \wedge \sim\!(@T(Activate)\ WHEN\ [Ignited \wedge Running \wedge \sim\!Toofast \wedge \sim\!Brake])))$$

If the system is INACTIVE, then the ignition is on and the conditioned event that activates the transition into mode CRUISE has not occurred since the system entered mode INACTIVE.

The SMV input for the *corrected* SCR behavioral specification and the *corrected* formulation of system goals are shown in Figure 5. All of the formulae declared in the SPEC section were successfully verified with respect to the specification's logic model, including the two formulae expressing the complex mode invariant for INACTIVE.

6 Conclusion

We have presented a logic-model semantics for SCR behavioral requirements specifications and have proposed new CTL modal logic operators for expressing formulae over modes, conditions, and events. It has been shown that CTL formulae can be verified with respect to an SCR logic model by translating an SCR specification into SMV input and using the SMV symbolic model checker. Using this technique, we have been able to express and verify formulae about SCR events and WHEN conditions. In particular, we have been able to verify an invariant property of an existing SCR specification that had not been automatically verified before.

While the analysis technique presented in this paper is sound and complete, the problem of symbolically model checking CTL formulae is in PSPACE [15]. Whether or not SMV model checking is more efficient than an exhaustive search of the system state space depends on the regularity of the specification. Additional empirical studies need to be done to determine whether the sparseness of SCR mode transition tables[5] qualifies as a form of regularity. We have just started a study to determine the feasibility of symbolically model checking larger SCR specifications.

The analysis technique presented in this paper could be adapted to other notations for specifying conditioned-event-driven software requirements. Consider the SCR, Statecharts and RSML requirements notations. These notations differ as to whether or not enabling conditions should be true before the occurrence of a conditioned event. According to SCR logic-model semantics, a conditioned event's enabling conditions (i.e., its WHEN conditions) must hold immediately before and during the occurrence of a conditioned event. In Statecharts and RSML, the enabling conditions need only hold at the time of the event's occurrence. Thus the logic model of a boolean Statecharts or RSML specification would be simpler than an SCR logic model. The specification's transition relation would only refer to the previous values of the triggering events and the current values of the triggering events and the enabling conditions; the previous values of the enabling conditions would be irrelevant. Likewise, a logic similar to the extended CTL logic presented here could be developed for expressing formulae that refer to Statecharts and/or RSML conditioned events. However, the SMV symbolic model checker can only analyze non-hierarchical specifications with boolean and enumeration variables. Thus, to apply this analysis technique to a Statecharts or an RSML specification, the specification must either be restricted to a subset of the notations' features or the specification would have to be preprocessed to produce a finite, flat representation for analysis.

[5]Most mode transitions depend on only a small subset of the system's conditions.

Acknowledgements

We would like to thank John Gannon, Paulo Alencar, Dominic Duggan, and Ken Braithwaite for their comments on previous drafts of this paper. We also greatly appreciate the feedback we received from the participants of the Fourth International SCR Workshop. The technical content and presentation of this work benefited a great deal from their input.

References

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

[2] J. Atlee. *Automated Analysis of Software Requirements.* PhD thesis, Department of Computer Science, University of Maryland, 1992.

[3] J. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[4] M. Browne. *Automatic Verification of Finite State Machines Using Temporal Logic.* PhD thesis, Department of Computer Science, Carnegie Mellon University, 1989.

[5] J. Burch, E. Clarke, K. McMillan, D. Dill, and J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond". In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 428–439, June 1990.

[6] E. Clarke, E. Emerson, and A. 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] D. Harel. "Statecharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[8] C. Heitmeyer. "Tools for Analyzing Requirements: A Formal Foundation", December 1994. Presented at the Fourth International SCR Workshop.

[9] C. Heitmeyer. Private communications, September 1995.

[10] C. Heitmeyer, B. Labaw, and D. Kiskis. "Consistency Checking of SCR-Style Requirements Specifications". In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, pages 56–65, March 1995.

[11] 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.

[12] J. Hofcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Co., 1979.

[13] J. Kirby. Example NRL/SCR Software Requirements for an Automobile Cruise Control and Monitoring System. Technical Report TR-87-07, Wang Institute of Graduate Studies, July 1987.

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

[15] K. McMillan. *Symbolic Model Checking: An approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

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

[17] J. van Schouwen. The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems. Technical Report TR-90-276, Department of Computing and Information Science, Queen's University, Kingston, Ontario, May 1990.