IEEE Copyright Notice

Published in: ***IEEE Transactions on Software Engineering,*** Vol. 19, No. 1, January 1993

"State-Based Model Checking of Event-Driven Systems Requirements"

Cite as:

J. M. Atlee and J. Gannon, "State-based model checking of event-driven system requirements," in *IEEE Transactions on Software Engineering*, vol. 19, no. 1, pp. 24-40, Jan. 1993.

BibTex:

```
@ARTICLE{210305,
author={J. M. {Atlee} and J. {Gannon}},
journal={IEEE Transactions on Software Engineering},
title={State-based model checking of event-driven system requirements},
year={1993},
volume={19},
number={1},
pages={24-40},
month={Jan},}
```

DOI: http://dx.doi.org/10.1109/32.210305

State-Based Model Checking of Event-Driven System Requirements

Joanne M. Atlee
University of Waterloo
Waterloo, Ontario

John Gannon*
University of Maryland
College Park, Maryland

August 28, 1992

Abstract

In this paper, we demonstrate how model checking can be used to verify safety properties for event-driven systems. SCR tabular requirements describe required system behavior in a format that is intuitive, easy to read, and scalable to large systems (e.g., the software requirements for the A7 aircraft). Model checking of temporal logics has been established as a sound technique for verifying properties of hardware systems. We have developed an automated technique for formalizing the semi-formal SCR requirements and for transforming the resultant formal specification onto a finite structure that a model checker can analyze. This technique was effective in uncovering violations of system invariants in both an automobile cruise control system and a water-level monitoring system.

1 Introduction

A *software requirements document* is usually the first specification of a system's required behavior. Errors in this document are difficult and expensive to correct if propagated to the design phase (or worse, to the implementation) [24]. Designers must be able to formally analyze requirements before system design begins.

A requirements specification is a behavioral specification of the system's activities; it describes the system's modes of operation and the events that cause the system to change modes. The specification often includes a set of safety assertions that must also be enforced. These assertions are invariant properties of the system, so they should also be properties of the requirements specification. As such, they are redundant information that can be used to verify that the requirements are internally consistent.

Temporal logic and model checking have been used to verify safety properties in hardware systems [6, 11]. The hardware system is portrayed as a logical model, and safety assertions are represented as logical formulas. One assumes that if a formula is true in the model, then the safety assertion holds in the hardware system. One reason that this verification technique is so promising is that model checking can be automated for some temporal logics.

This paper demonstrates the feasibility of using model checking to analyze safety properties of software requirements. Our analysis technique combines the SCR requirements specification language [1, 19, 18] with the CTL model checker [8, 5]. SCR requirements are intuitive, easy to write and change, and scalable to large systems (e.g., the software requirements for the A7 aircraft [1]). The CTL model checker is an

*This work was supported by the Office of Naval Research under Contract N00014-91-K-2029 and by the Air Force Office of Scientific Research under Contract AFOSR 90-0031.

automated analyzer that has been used successfully to verify properties of hardware systems [6]. The result is a formal method whose specification language is easy read and understand, and whose analysis is automated.

Section 2 of this paper reviews event-oriented requirements specifications and state-based model checking, and presents our algorithm for constructing state-based models of event-oriented system requirements. Section 3 describes two case studies of event-oriented requirements and their safety properties. Section 4 contains a discussion of problems and solutions that arose during the case studies.

2 Analysis Technique

In this section, we briefly describe the models of SCR software requirements and CTL machines. We also present our automated technique for formalizing the software requirements and for transforming the event-driven requirements into a state-based CTL machine that can subsequently be analyzed.

2.1 Event-Driven Requirements Specifications

SCR requirements were developed by a research group at NRL as part of a general Software Cost Reduction project [1, 19, 18]. The model they developed is based on compositional, event-driven, *mode*-machines:

- A *mode* is a set of system states that share a common property.
- A *modeclass* is a set of modes, and the union of the modes in a modeclass must cover the system's state space.
- The system is in exactly one mode of each modeclass at all times.
- A *mode transition* occurs between modes in the same modeclass as a result of system state changes.
- Mode transitions are specified by *conditions* and *events*, which comprise the machine's input language.

Informally, each modeclass describes one aspect of the system's behavior, and the global behavior of the entire system is defined by the composition of all the system's modeclasses.

Conditions and events

The input to the machines is the set of environmental *conditions* (e.g., whether a button is being pressed). All conditions are boolean, although first-order predicate conditions that can be represented by a finite number of boolean conditions (such as integer ranges and timing constraints) are also expressible.

A system's behavior is defined and controlled by *changes* to the conditions' values. A change to a condition's value is an *event*, and events are only detectable at the point in time at which they occur. For example, event

$$@\mathrm{T}(\mathit{Cond1})$$

specifies the point in time when the value of condition *Cond1* changes from false to true. Similarly, event @F(*Cond1*) specifies the time when condition *Cond1* becomes false. In these events, we call condition *Cond1* the events' *triggered condition.* The occurrence of an event might also depend on the values of other conditions. For example,

$$@\mathrm{T}(\mathit{Cond1})\ \mathrm{WHEN}\ [\mathit{Cond2}]$$

describes the event of condition *Cond1* becoming true *while* condition *Cond2* is also true. In the above event, we call condition *Cond2* the event's *WHEN condition.* More complex events can be created from simpler events and conditions using boolean operators.

SCR semantics propose three definitions for an event occurrence and allow the requirements designer to decide which definition pertains to each event [1]. For a given event @T(A) WHEN [B] there is a

Temperature Control System:

Current Mode	Running	BelowDesiredTemp	TempOK	AboveDesiredTemp	New Mode
Off	@T	–	t	–	Inactive
	@T	t	–	–	Heat
	@T	–	–	t	AC
Inactive	@F	–	–	–	Off
	–	@T	–	–	Heat
	–	–	–	@T	AC
Heat	@F	–	–	–	Off
	–	–	@T	–	Inactive
AC	@F	–	–	–	Off
	–	–	@T	–	Inactive

Initial Mode: Off (~Running)

Table 1: Mode Transitions for temperature control system.

defined time ϵ, such that if $t_{@T(A)}$ is the time that triggered condition A becomes true, then the event occurs if

1) B is true throughout closed interval $[t_{@T(A)} - \epsilon, t_{@T(A)}]$, or
2) B is true throughout interval $[t_{@T(A)}, t_{@T(A)} + \epsilon]$, or
3) B is true throughout interval $[t_{@T(A)} - \epsilon, t_{@T(A)} + \epsilon]$

We use the first definition for all event occurrences. This definition allows us to distinguish between WHEN conditions (which must be satisfied both before and at the time of the event) and triggered conditions (which must be unsatisfied immediately before the event and satisfied at the time of the event).

Modes and transitions

The system state is defined by the current values of the system conditions, and the system state space is the set of all combinations of the conditions' values. To reduce of the size of the state space, sets of system states are collected into *modes*. The use of modes abstracts away details that do not contribute to the system's behavior; the values of all system conditions are not important at all times. In fact, the conditions' values are only important when they can affect mode transitions.

A *mode transition* occurs between modes in the same modeclass as a result of the occurrence of an event. A transition's *transition event* specifies the event that triggers the transition. Mode transitions occur at the same time as their transition event and, like the event, take zero time units to complete. Two transitions from the same mode are simultaneously enabled if their transition events occur at the same time. In such a case, the modeclass is nondeterministic, and the activation of either transition (but not both) satisfies the requirements.

Table 1 is a requirements specification for a temperature control system. The specification consists of a single modeclass comprised of four modes: OFF, meaning that the system is turned off; INACTIVE, meaning that the system is on, but neither the heater nor the air conditioner is on; and HEAT and AC, meaning that either the heater or the air conditioner, respectively, is on and controlling the temperature. Condition *Running* indicates whether or not the system has been turned on, and conditions *BelowDesiredTemp*, *TempOK*, and *AboveDesiredTemp* represent the current temperature.

The initial state of the system is mode OFF, in which the system is not *Running*. The system is undefined if *Running* initially true. Each row in the table specifies the event causing the transition from the mode on the left to the mode on the right. Each column in the center of the table represents a system

condition. A table entry containing an upper-case letter ('@T' or '@F') signifies that the condition is a triggered condition of the transition event, and must change value (to true or false, respectively) to activate the mode transition. A table entry containing a lower-case letter ('t' and 'f') signifies that the condition is a WHEN condition of the transition event, and must have a particular value (true or false, respectively) both immediately before and at the time of the event occurrence. If a condition is neither a triggered condition nor a WHEN condition of a transition event, then the corresponding table entry is marked with a hyphen ('–').

If the temperature control system is in mode OFF and starts *Running* when the temperature is within desired limits, then it will enter mode INACTIVE. The AC comes on when the system is INACTIVE and the temperature rises above the desired temperature, or when the system starts *Running* while the temperature is above the desired temperature. The AC cycles off when the temperature falls to within $3^\circ F$ of the desired temperature. Transitions into and out of mode HEAT resemble those of mode AC.

Some of the global constraints that one would expect to hold in a temperature control system are:

OFF $\Rightarrow \sim$*Running*
INACTIVE $\Rightarrow$ (*Running & TempOK*)
HEAT $\Rightarrow$ (*Running & BelowDesiredTemp*)
AC $\Rightarrow$ (*Running & AboveDesiredTemp*)
(*Running&BelowDesiredTemp* $\Rightarrow$ (Heat | O(Heat))[1]
(*Running&AboveDesiredTemp* $\Rightarrow$ (AC | O(AC))

The first four formulas state that whenever the system is in a particular mode, certain system conditions have invariant values. For example, if the system is in mode HEAT, then the system is *Running* and the temperature is *AboveDesiredTemp*. The last two formulas state that if certain conditions hold then either the system is in a particular mode or the next system transition will be into that mode. For example, if the system is *Running* and the temperature is *BelowDesiredTemp*, then either the HEAT is on or the HEAT will come on imminently.

SCR-style specifications and global assertions provide different views of a system's requirements. Modes and mode transitions specify system properties that hold *under certain conditions*, whereas global assertions specify properties that must *always* hold. Thus, the global assertions are redundant information which already exists in the behavioral specification. We use this redundancy to ensure that an SCR tabular requirements specification enforces a system's invariant properties.

2.2 Formalizing SCR requirements

To enhance readability of SCR-style requirements, redundant information is often excluded from the tabular requirements. A row in a mode transition table specifies only the minimal set of triggered and WHEN conditions the system designer needs to consider when determining whether or not the transition event has occurred. To the requirements and system designers, a condition value '–' in a transition event means that the value of the condition is not important in the detection of this event occurrence. However, to an analyzer, a condition value '–' in a transition event means that the event can occur regardless of the value of the condition. If a condition is restricted to a certain value (or set of values), then this knowledge should be explicitly stated in the requirements specification. Otherwise, invariant properties that depend on this missing information cannot be automatically verified.

The first type of information missing from the tabular requirements pertains to relationships between condition values. For example, the temperature in a room cannot simultaneously be *AboveDesiredTemp*, *TempOK*, and *BelowDesiredTemp*. If a mode transition is dependent upon condition *AboveDesiredTemp*

[1] The symbol O is a modal logic operator often used as a *nextstate* operator. It is used here to specify what should be true in the next system state.

Operating:

Current Mode	Running	BelowDesiredTemp	TempOK	AboveDesiredTemp	New Mode
Off	@T	**f**	t	**f**	Inactive
	@T	t	**f**	**f**	Heat
	@T	**f**	**f**	t	AC
Inactive	@F	–	–	–	Off
	@F	*@T*	*@F*	*f*	
	@F	*f*	*@F*	*@T*	
	–	@T	**@F**	**f**	Heat
	–	**f**	**@F**	@T	AC
Heat	@F	–	–	–	Off
	@F	*@F*	*@T*	*f*	
	–	**@F**	@T	**f**	Inactive
AC	@F	–	–	–	Off
	@F	*@F*	*@T*	*f*	
	–	**f**	@T	**@F**	Inactive

Initial Mode: Off (~Running)

Table 2: Mode transition tables with transitions due to simultaneous events.

being true, then it is also dependent upon conditions *TempOK* and *BelowDesiredTemp* being false. Our transformation algorithm accepts a set of condition relationships and propagates the information throughout the mode transition tables (the details of this process are described in [2]). It is important to note that these changes are not additional restrictions on the requirements; information from other parts of the requirements document (from the section that describes the conditions) is incorporated into the mode transition tables. The additional transition conditions in the temperature control system, due to the relationships between the temperature conditions, appear as bold characters in Table 2.

The second step in the formalization of SCR software requirements involves sequences of instantaneous mode transitions [2]. SCR mode transitions are instantaneous. Also, the system need not spend a minimum amount of time in a mode before exiting the mode. Therefore, if a mode's transition conditions are enabled at the time the mode is entered, a subsequent transition will occur and the system will effectively spend no time in the intermediate mode. For example, if the temperature control system is in mode HEAT, and somebody turns the system off at the same time as the temperature rises to *TempOK*, then transitions to modes INACTIVE and OFF are both enabled. If the system nondeterministically enters the former mode, then (since mode transitions take zero time to complete) the system will enter mode INACTIVE at the instant the system is turned off, causing a subsequent transition to mode OFF. Our transformation algorithm follows all sequences of simultaneous mode transitions (HEAT-INACTIVE-OFF) and adds them to the requirements specification as new distinct mode transitions (HEAT-OFF), whose transition event is the conjunction of the transition events of the transitions that compose the sequence (@F(*Running*) & @T(*TempOK*)). Mode transitions added to the temperature control system to represent sequences of instantaneous transitions appear in italics in Table 2.

Next, the transformation algorithm detects all instances of nondeterminism in the specification and issues a warning message for each instance [2]. For example, in Table 2, all of the italicized transition events trigger multiple nondeterministic mode transitions; event (@F(*Running*) & @T(*TempOK*)) in mode HEAT triggers transitions into both OFF and INACTIVE. If nondeterminism is detected in the specification,

Operating:

Current Mode	Running	BelowDesiredTemp	TempOK	AboveDesiredTemp	New Mode
Off	@T	**f**	t	**f**	Inactive
	@T	t	**f**	**f**	Heat
	@T	**f**	**f**	t	AC
Inactive	@F	–	–	–	Off
	t	@T	**@F**	**f**	Heat
	t	**f**	**@F**	@T	AC
Heat	@F	–	–	–	Off
	t	**@F**	@T	**f**	Inactive
AC	@F	–	–	–	Off
	t	**f**	@T	**@F**	Inactive

Initial Mode: Off (~Running)

Table 3: Mode transition tables for deterministic temperature control system.

the requirements designer must decided whether or not the nondeterminism should be allowed. The designer may purposely specify nondeterministic requirements so that nonessential design decisions can be delayed. Furthermore, unlike any of the formalization described so far, converting a nondeterministic requirements specification into a deterministic specification changes the semantics of the specification and forces additional restrictions on the designs that satisfy the requirements. Therefore, the designer is responsible for deciding whether and how a nondeterministic specification should be made deterministic. The temperature control system can be made nondeterministic by forcing condition *Running* to be a WHEN condition of all transitions among modes INACTIVE, HEAT, and AC (see Table 3). One of the results of this change to the specification is that there are no sequences of instantaneous mode transitions.

Finally, if the specification consists of multiple, concurrent modeclasses, the transformation algorithm composes the transition tables representing the different modeclasses into a single global tabular specification [2]. The resultant specification is in a form that can be formally analyzed. To use a particular analysis tool, one needs to transform the global tabular specification into the appropriate representation that the tool will accept. Our approach is to convert this specification into a CTL machine, which can then be analyzed using the CTL model checker.

2.3 State-Based Model Checking

If a system's behavioral requirements can be represented as a finite structure, and if the safety assertions can be expressed as propositional temporal logic formulas, then a model checker can be used to determine if the structure is a model of the logic formulas (and by implication, that the safety assertions hold in the requirements specification). We used an improved version of Clarke's EMC model checking system [8], called MCB [5], as our model checker.

Informally, the system is expressed as an extended finite state machine, in which each state is annotated with transition conditions (*input condition* values) and *attributes* (properties distinct from input conditions). The machine is in exactly one current state at all times. Once every time unit, one of the state's transitions is activated, leaving the machine in a possibly new current state. The values of the input conditions determine which of the current state's transitions is enabled. Since a transition is activated every time unit, at least one of the current state's transitions must be enabled at all times. This means that the disjunction of the current state's transition conditions must always be true. If more than

one transition is enabled, then one is nondeterministically chosen and activated[2].

This state machine can serve as a temporal logic model of a system, and we can test whether safety properties phrased as temporal formulas hold in the model. The formulas are expressed in a propositional branching time logic called computational tree logic (CTL), whose operators permit explicit quantification over all possible futures. The syntax and semantics for CTL formulas are defined in [8] and are simply summarized below:

1) Every atomic proposition[3] is a CTL formula.
2) If f and g are CTL formulas, then so are: $\sim f$, $f\&g$, $f \mid g$, AXf, EXf, $A[fUg]$, $E[fUg]$, AFf, EFf, AGf, EGf.

The symbols $\sim$ (*not*), & (*and*), and | (*or*) are logical connectives and have their usual meanings. Formula AXf (EXf) means that formula f holds in every (in some) immediate successor of the current state. U is the *until* operator, and formula $A[fUg]$ ($E[fUg]$) means that along every (some) path there exists a future state s_i in which g holds and f is true until state s_i is reached. The formula AFf (EFf) means that along every (some) path there exists some future state in which f holds. The formula AGf (EGf) means that f holds in every state along every (some) path.

Safety assertions are invariant, so the formulas we want to check are of the form AGf. The safety assertions for our temperature control system, described in section 2.1, are represented by the following CTL formulas:

$AG(Off \rightarrow \sim Running)$
$AG(Inactive \rightarrow (Running \ \& \ TempOK))$
$AG(Heat \rightarrow (Running \ \& \ BelowDesiredTemp))$
$AG(AC \rightarrow (Running \ \& \ AboveDesiredTemp))$
$AG((Running \ \& \ BelowDesiredTemp) \rightarrow (Heat \mid AX(Heat)))$
$AG((Running \ \& \ AboveDesiredTemp) \rightarrow (AC \mid AX(AC)))$

The model checker accepts a CTL machine and a CTL formula, and determines whether or not the formula holds in the machine. If the model checker determines the formula f is true, then the safety property holds in the CTL state machine and also in the system requirements.

2.4 Mapping SCR requirements onto CTL machines

This section describes how to transform formalized SCR requirements into CTL machines. Most elements of the SCR requirements model correspond naturally to elements of CTL machines:

SCR requirements	CTL machine
modes	states
mode transitions	state transitions
conditions	input variables
events	?

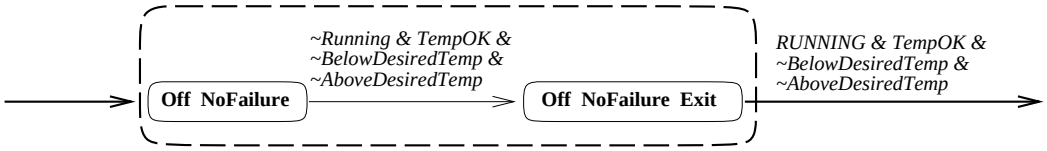
There is no natural modeling of events in a CTL machine. CTL state transitions occur based on the current state and the current values of the input conditions. Mode transitions, on the other hand, occur at the time of their transition events; the system spends zero time in a mode once one of its transitions

[2]Note that above definition of a CTL machine differs from the one used in the earlier EMC version of the model checker [8]. Previously, states were annotated with attributes, which consisted of both input conditions and output propositions. The state transitions were unconditional. To model conditional transitions, one had to create multiple instances of the source state and assign different values to the copies' input conditions. In the new definition, all states annotated with the same set of output propositions are combined in a single state, and the transition conditions are related to the a state's transitions rather than to the state.

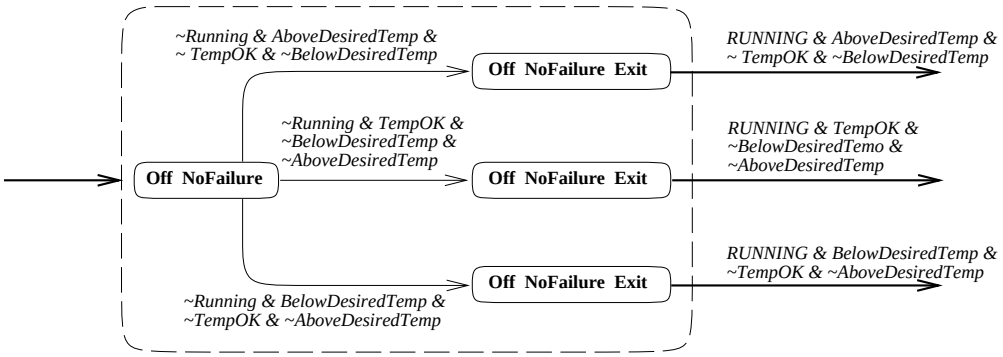
[3]The set of atomic propositions is the union of the set of input conditions and the set of state attributes.

has been activated. Therefore, we need to be able to detect changes in condition values and ensure that state transitions are activated by these value changes.

To model an event, we can represent a mode as two CTL states: a *CTL mode state* and a *CTL exit state*.



The CTL mode state represents the system *in the mode*, and the CTL exit state represents the system *leaving the mode* due to the occurrence of an event. Both the CTL mode state and its exit state are annotated with the names of the mode's component modes. The exit state is also annotated with an additional state attribute EXIT, to indicate that it is a CTL exit state. The transition from the CTL mode state to its exit state is annotated with the event's WHEN conditions and the *negations* of the event's triggered conditions. The transition from the exit state (to the CTL mode state of the destination mode) is annotated with the event's triggered and WHEN conditions. This representation captures the property that the event's triggered conditions must become satisfied at the time of the mode transition; it also captures the property that the WHEN conditions must be satisfied both before and at the time of the transition. (To make the graphical CTL representation more readable, the triggered conditions from the exit state appear in upper-case characters, to distinguish them from the WHEN conditions.) Multiple CTL exit states are needed to represent the events of multiple transitions from the same mode.

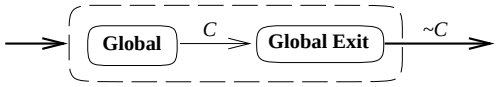


Unfortunately, this intuitive representation does not model all desired properties of a mode. In particular, the invariant properties of the CTL mode state are not always equivalent to the invariant properties of the SCR mode that it represents. *Mode invariants* are the invariant properties of a system mode. A mode invariant must be true when the system enters the mode, must remain true while the system stays in the mode, and must either be true or become false when the system leaves the mode. For example in the temperature control system, condition *Running* is always false upon entering mode OFF and the mode is always exited when *Running* becomes true. Therefore, $\sim$*Running* is an invariant property of mode OFF. *State invariants*, on the other hand, are conditions that are always true when the CTL machine is in that state. A state's output propositions are state invariants since they are properties of the state. In addition, since one of the state's transitions must be enabled at all times, any input condition that is a transition condition of all the state's transitions is also a state invariant.

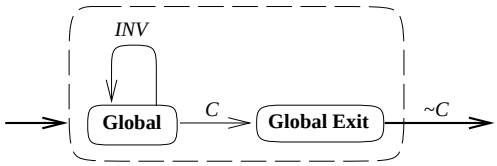
The following examples show the circumstances under which our CTL representation of modes and transitions does not always accurately model mode invariants. They also describe how the representation

can be refined to handle these cases.

Example A: The model checking algorithm assumes that for each CTL state some outgoing transition is always enabled. Therefore, the disjunction of a state's transition conditions must always be true. This assumption can cause a formula that is stronger than the mode's invariant to be a provable invariant of the CTL mode state. Consider our CTL representation for a mode whose sole transition event is @F(C):

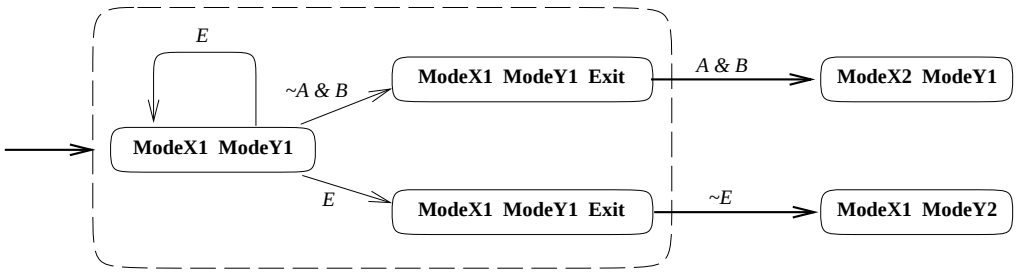


Since some transition from the current CTL state must be satisfied at all times, and since the CTL mode state in the above example has only one transition, its transition condition C is a state invariant of the CTL mode state. If condition C can be false when the system enters the mode, then C is not an invariant of the mode and there is a discrepancy between the invariant properties of the mode and the invariant properties of its representative CTL mode state. To ensure that the disjunction of a CTL mode state's transition conditions is no stronger than the mode's invariant, we add a transition from the CTL mode state back to itself and annotate it with the mode invariant:



If C is a mode invariant, then the state invariant of the CTL mode state will not change with the addition of this transition. If C is not a mode invariant, then since the new transition is not annotated with it, C will not be a verifiable state invariant.

Example B: Another problem with the representation is that a state invariant of a CTL mode state may not be as strong as the mode invariant of its associated mode. Consider a specification consisting of multiple modeclasses; the global mode-machine specifies the system's global modes (each the composition of one component mode from each modeclass) and the transitions between global modes. A transition event of a global transition is the conjunction of the transition events of a set of simultaneously occuring mode transitions. There is no guarantee that every global transition event will contain the source global mode's invariant. If it does not, then the mode invariant of the global mode will not be verifiable. Consider our CTL representation for a global mode consisting of two component modes:



There is a mode transition from ModeX1 to ModeX2 if event @T(A) WHEN [B] occurs, and another mode transition from ModeY1 to ModeY2 if event @F(E) occurs. If condition E is always true upon entering ModeY1, then it is an invariant property of both the mode ModeY1 and the global mode

ModeX1/ModeY1. Condition E is not a state invariant of the CTL mode state, however, since there is a transition from the state whose event does not require the truth of E. In the CTL model, this transition implies that it is possible to be in mode ModeX1/ModeY1 and for condition E to be false in that state. To ensure that the state invariant of a CTL mode state is at least as strong as the global mode's invariant, we add the global mode invariant to the transition conditions of all state transitions from the CTL mode state to its exit states:

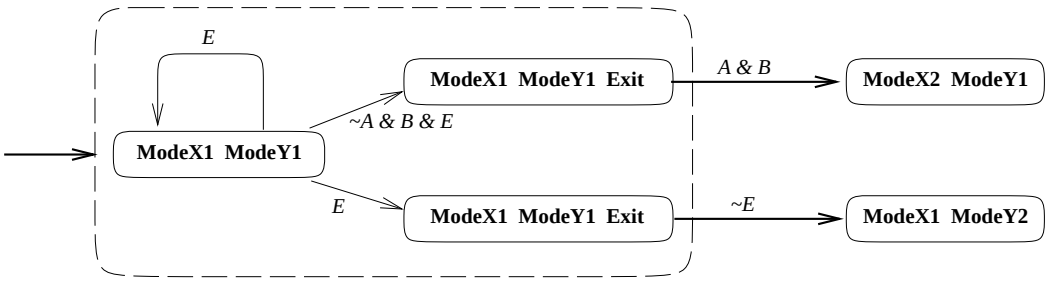


Figure 1 contains the CTL machine representation of the temperature control system presented in Table 1. Mode Off is represented by one CTL mode state Off and three CTL exit states Off Exit. The states' attributes indicate which original system mode is represented by the state (Off), plus whether or not the state is an exit state (Exit). The exit states' transitions are copied directly from the specification table. For example, the center Off Exit state is annotated with transition condition

$$Running \ \& \ TempOK \ \& \ \sim BelowDesiredTemp \ \& \ \sim AboveDesiredTemp$$

which represents the first transition leaving mode Off in Table 1. The CTL mode states' transition conditions consist of the mode transitions' when conditions and the negations of their triggered conditions. For example, the state transition from the CTL mode state Off to the center exit state Off Exit is annotated with $\sim Running$ (the negation of the triggered condition of the event) and with $TempOK \ \& \ \sim BelowDesiredTemp \ \& \ \sim AboveDesiredTemp$ (the when conditions of the event).

We now present our algorithm for transforming SCR specifications into CTL machines:

1. Create a CTL input condition for each environmental condition in the requirements specification.
2. Create a CTL output proposition for each system mode in the original SCR requirements specification.
3. Create output proposition Exit, which will be used to indicate CTL exit states.
4. For each mode in the SCR requirements:
   (a) Create a CTL mode state and annotate it with the name of the SCR mode.
   (b) Create a state transition from the CTL mode state back to itself, and annotate this transition with the mode's invariant properties.
   (c) Create a unique CTL exit state for each transition from the mode. Annotate each exit state with the name of the mode plus the attribute Exit.
   (d) For each CTL exit state and its associated mode transition, create a state transition from the CTL mode state to the CTL exit state and annotate the state transition with both the when conditions and the *negations* of the triggered conditions of the mode transition event. Also annotate this transition with the mode's invariant properties.

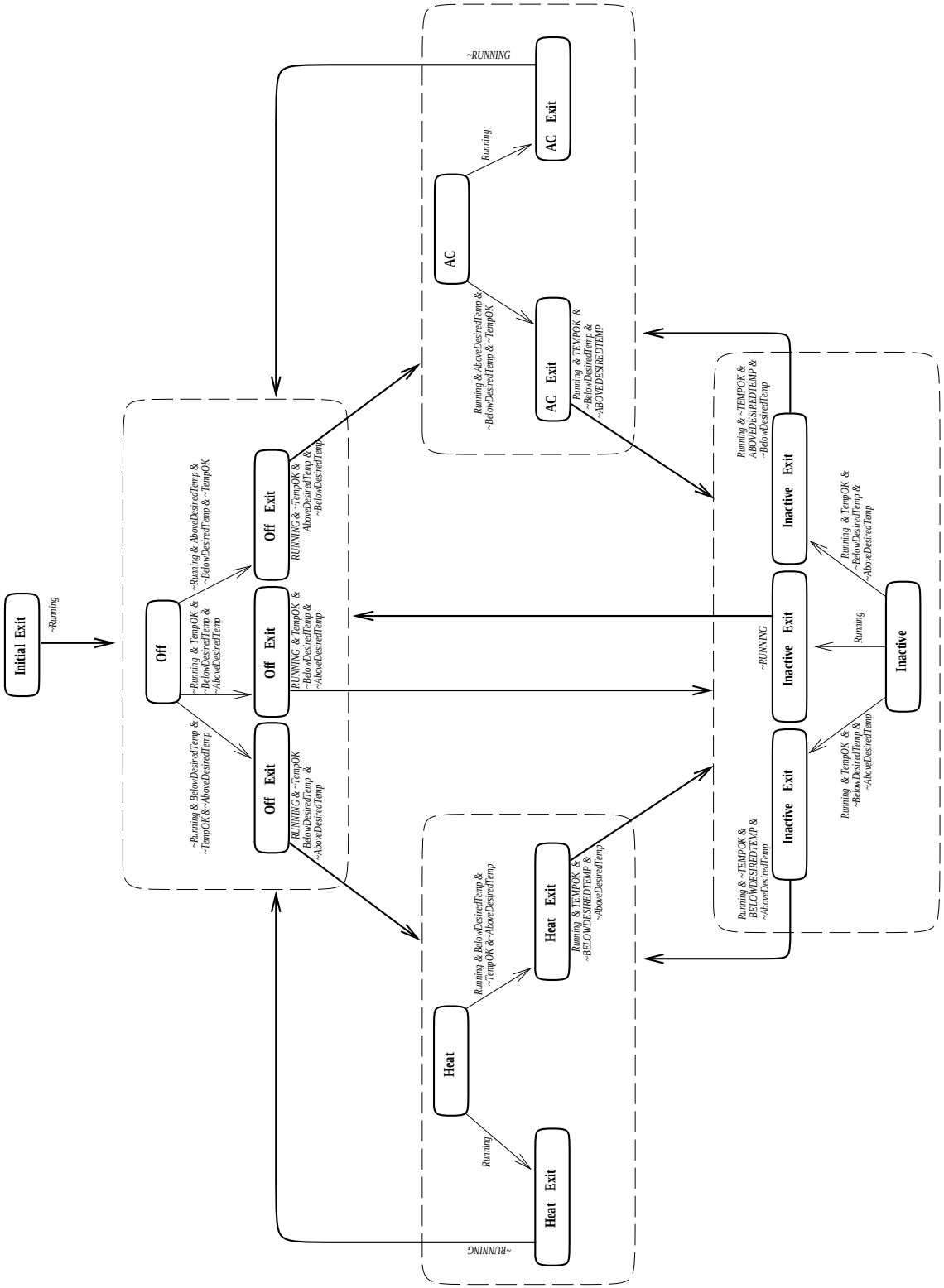


Figure 1: CTL representation of temperature control system.

(e) For each CTL exit state and its associated mode transition, create a state transition from the exit state to the CTL mode state of the destination mode of the mode transition. Annotate this transition with the triggered conditions and the WHEN conditions of the mode transition event.

5. Create a unique CTL exit state that is only annotated with output propositions INITIAL (to indicate that it is the initial state of the system) and EXIT (to indicate it is a CTL exit state). For each initial mode, create a state transition from this exit state to the CTL mode state of the initial mode; annotate the state transition with the initial conditions of the initial mode.

2.5 Putting it together

Sections 2.3 and 2.4 describe our transformation for formalizing a semi-formal SCR requirements document and transforming it into a CTL machine representation. The steps of the methodology, and who is responsible for performing each step, are summarized below.

1. The requirements designer provides the tabular specifications, and the initial conditions under which a mode is the initial mode of a modeclass.

2. The (human) requirements analyzer declares the relationships that hold among the environmental conditions. These relationships are manually transcribed from the section of the requirements document that describes the environmental conditions.

3. The transformation algorithm details the tabular requirements with respect to the declared condition relationships.

4. The transformation algorithm derives all possible sequences of instantaneous mode transitions and explicitly adds these sequences to the tables as 'new' single mode transitions.

5. The transformation algorithm detects all instances of nondeterminism in the detailed specification and issues a warning message for each instance.

6. The requirements designer decides whether any of the detected instances of nondeterminism invalidate the intended requirements. For each instance of undesired nondeterminism, the requirements designer prioritizes the nondeterministic mode transitions and manually modifies the specification to enforce these priorities.

7. The transformation algorithm composes the mode transition tables (represented by multiple modeclasses) into a global SCR specification.

8. The transformation algorithm constructs a representative CTL machine from the global SCR specification.

A more elaborate description of each of the above steps, and proofs that these steps preserve certain system properties, can be found in [2].

The above algorithm is part of our automated transformation algorithm that accepts an SCR tabular requirements specification and creates a representative CTL machine that can be analyzed using the CTL model checker. All of the CTL machines presented in this paper are represented graphically, as in Figure 1. The ASCII notation used by the model checker is described in [5].

Cruise Control:

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
	–	@F	–	–	–	–	–	Inactive
	–	–	@T	–	–	–	–	
	–	–	–	@T	–	–	–	Override
	–	–	–	–	–	@T	–	
Override	@F	–	–	–	–	–	–	Off
	–	@F	–	–	–	–	–	Inactive
	t	t	–	f	@T	–	–	Cruise
	t	t	–	f	–	–	@T	

Initial Mode: Off

Table 4: Mode transitions for automobile cruise control

3 Case Studies

We used our analysis technique to analyze two requirements documents, one for an automobile cruise control system and one for a water-level monitoring system. The requirements specifications of the two systems were originally specified using an alternate version of SCR requirements [19]; in this paper the specifications appear in the new SCR mode table format [12], which is easier to read and understand. We transformed these systems' requirements into CTL machines, rephrased the required safety properties as logical formulas, and verified the formulas using the MCB model checker. In both studies, we found discrepancies between the systems' requirements specifications and their safety assertions.

3.1 Case Study: Automobile Cruise Control

The cruise control specifications used in this study come from [21]. The possible states of the cruise control are partitioned into four modes:

OFF	Ignition is off
INACTIVE	Ignition is on, but cruise control is not on
CRUISE	Ignition and cruise control are on, controlling the vehicle's speed
OVERRIDE	Ignition and cruise control are on, but not controlling the vehicle's speed

The system always starts in mode OFF.

Table 4 contains the mode transition table. It shows the events and conditions which cause the cruise control system to transition from one mode to another. The table uses the following conditions:

Ignited	Ignition is on
Running	Engine is running
Toofast	Cruise control is unable to decelerate the automobile when speed is above the desired speed
Brake	Brake is on
Activate	Cruise control lever is set to ACTIVATE
Deactivate	Cruise control lever is set to DEACTIVATE

Cruise Control:

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

Initial Mode: Off

Table 5: Detailed mode transitions for adjusted cruise control specifications.

Resume Cruise control lever is set to RESUME

The requirements document for the cruise control system also lists the following safety properties[4]:

Mode	Safety Property
OFF	$\neg Ignited$
$\neg$OFF	$Ignited$
INACTIVE	$Ignited \wedge ((\neg Running) \vee (\neg Activate))$
CRUISE	$Ignited \wedge Running \wedge \neg Brake$
OVERRIDE	$Ignited \wedge Running$

Whenever the system is in a particular mode, the associated safety assertion must hold. For example, if the system is in mode OFF, then the automobile's ignition must be off. The safety assertion 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 the cruise control has not been activated. If any of these conditions does not hold, then the system should not be in the INACTIVE mode.

CTL Machine Construction

First, the transformation algorithm formalizes the software requirements. The following relationships were found in the requirements descriptions of the system conditions.

$Running \Rightarrow Ignited$

$Activate \mid Deactivate \mid Resume$

[4]The third safety assertion is actually (Inactive $\rightarrow Ignited \wedge ((\neg Running) \vee (\neg StartIncr))$), which uses a condition ($StartIncr$) not present in the requirements table. The assertion printed below is weaker than the original safety assertion: $StartIncr$ is true when the driver has held the cruise control lever in the $Activate$ position for a period of time, and therefore $StartIncr \rightarrow Activate$.

The first relationship states that the ignition must be on if the engine is running; the second relationship describes the cruise control lever conditions as members of an enumerated type, of which exactly one member is always true. Our transformation algorithm accepts condition relationships in the above format and propagates the information throughout the mode transition tables. For example, whenever the value of *Activate* is becoming true in the tables, the values of *Deactivate* and *Resume* are changed to either false (**f**) or becoming false (**@F**). A modified specification of the cruise control system is shown in Table 5. The additional mode transition conditions due to the above variable interrelationships appear as bold characters.

The next step of the formalization process explicates sequences of instantaneous mode transitions. For example, if the system is in mode CRUISE and the driver depresses the brake pedal at the same time as the engine fails, then the system can transition from CRUISE into OVERRIDE (because the brakes are on) and immediately transition from OVERRIDE into INACTIVE (because the engine has failed). Our transformation algorithm finds each sequence of simultaneous mode transitions (CRUISE-OVERRIDE-INACTIVE) and adds it to the requirements specification as a new distinct mode transition (CRUISE-INACTIVE), whose transition event is the conjunction of the transition events of the transitions that compose the sequence (@F(*Running*) & @T(*Brake*).

This process reveals hidden instances of unintended nondeterminism in the specification. For example, the occurrence of the new compound event (@F(*Running*) & @T(*Brake*)) enables transitions to both INACTIVE and OVERRIDE. The specification should only allow a transition into OVERRIDE if the engine continues *Running*.

A STATEMATE specification for a similar cruise control system [26] avoids this type of unintended nondeterminism by prioritizing the system's modes. STATEMATE specifications model a system's behavior as a hierarchical state machine, where transitions from the same state are prioritized based on the level of the destination state. In the STATEMATE cruise control specification, the top level of the system consists of two states, ENGINE-OFF and ENGINE-ON[5]. State ENGINE-ON represents an internal state machine that describes the system's behavior when the engine is on; it includes states CRUISE-ACT and CRUISE-INACT. (State CRUISE-ACT corresponds to mode CRUISE, and state CRUISE-INACT corresponds to mode OVERRIDE). Since state ENGINE-OFF is at a higher level in the specification than state CRUISE-INACT, the transition from CRUISE-ACT to ENGINE-OFF always takes precedence over the transition from CRUISE-ACT to CRUISE-INACT.

We can capture the precedence information represented as STATEMATE hierarchies by adding conditions to mode transitions. Table 5 contains an adjusted cruise control specification in which some of the transitions have additional constraints on their enabling conditions to prevent them from being enabled when a transition of higher priority is enabled. For example, a transition from CRUISE to OVERRIDE can only occur:

- if the ignition is on (disabling the transition to OFF),
- the engine is running (disabling a transition to INACTIVE), and
- the automobile is not going too fast (disabling the other transition to INACTIVE)

The additional conditions added to prevent nondeterminism appear in italics in Table 5. There are no sequences of instantaneous mode transitions in the adjusted specification.

The CTL machine that corresponds to the adjusted cruise control specifications is shown in Figure 2.

Analysis

Once the CTL machine has been created, we can verify that the tabular specifications and the relationships between conditions were entered correctly. To do this, we rephrase the requirements specifications as liveness properties, and use the MCB model checker to prove that they are true with respect to the

[5]The STATEMATE specification does not distinguish between the ignition being on and the engine running.

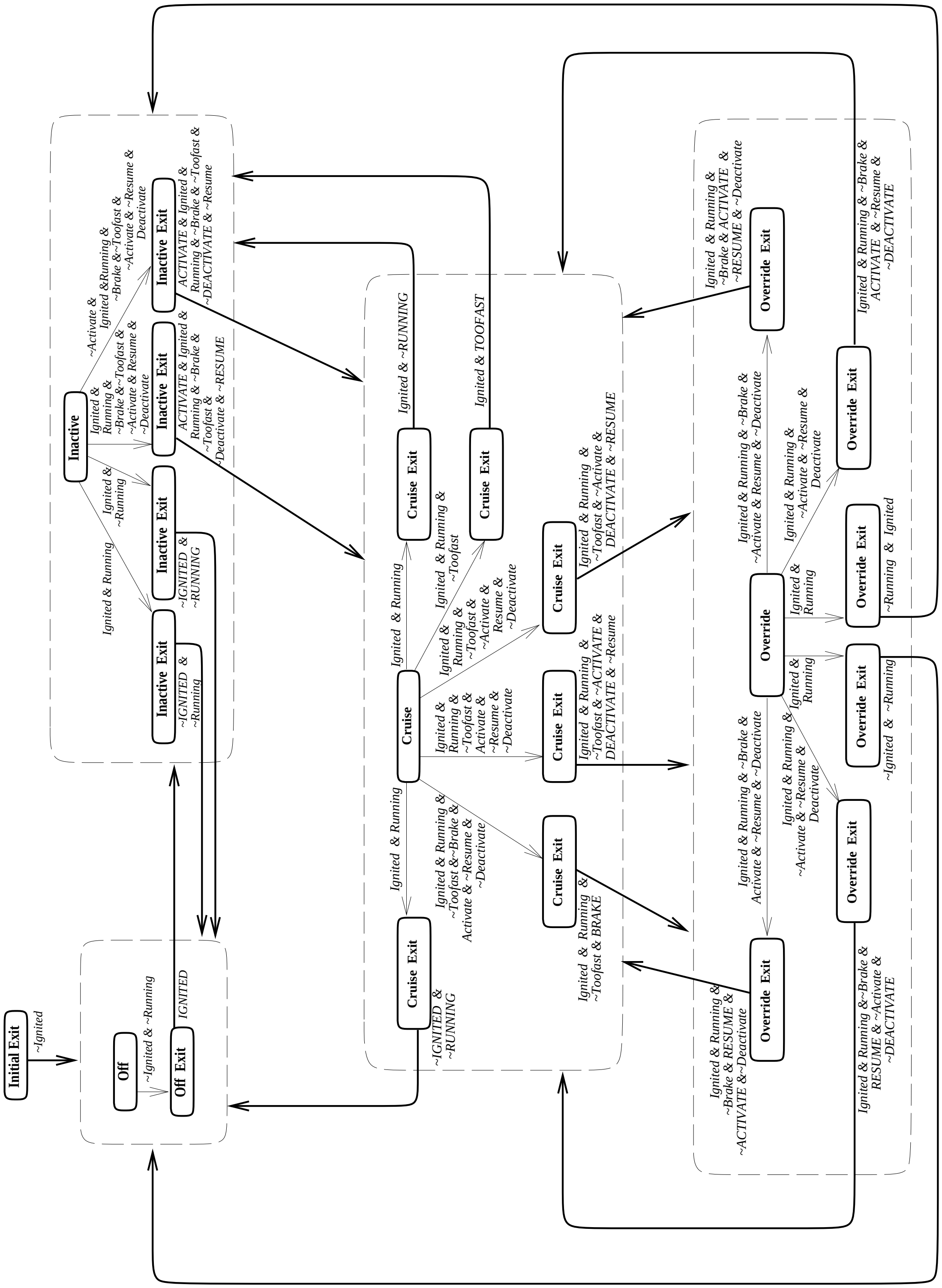


Figure 2: CTL machine for adjusted cruise control specifications.

constructed machine. For each mode transition, we create two CTL formulas that state if the CTL machine is in the transition's source mode and the transition event occurs, then the machine will transition into the transition's destination mode. The following formulas represent the first mode transition from CRUISE to OVERRIDE in Table 5; output proposition EXIT is used to distinguish between the system being in the mode (~EXIT) and the sytem exiting the mode (EXIT).

$$AG((Cruise \;\&\; \sim Exit \;\&\; Ignited \;\&\; Running \;\&\; \sim Toofast \;\&\; \sim Brake) \rightarrow$$
$$AX((Cruise \;\&\; Exit \;\&\; Ignited \;\&\; Running \;\&\; \sim Toofast \;\&\; \sim Brake) \rightarrow AX(Override)))$$
$$EF((Cruise \;\&\; \sim Exit \;\&\; Ignited \;\&\; Running \;\&\; \sim Toofast \;\&\; \sim Brake) \;\&$$
$$EX(Cruise \;\&\; Exit \;\&\; Ignited \;\&\; Running \;\&\; \sim Toofast \;\&\; \sim Brake))$$

The first formula states that if the system is in the source mode ($Cruise \;\&\; \sim Exit$) and the transition's triggered condition is negated ($\sim Brake$) and the transition's WHEN conditions hold ($Ignited \;\&\; Running \;\&\; \sim Toofast$); and if in the next state, the system is leaving the source mode ($Cruise \;\&\; Exit$) with the transition's triggered and WHEN conditions satisfied ($Ignited \;\&\; Running \;\&\; \sim Toofast \;\&\; \sim Brake$); then in the state after that, the system will be in the destination mode ($Override$). The second formula states that the hypothesis of the first formula is not false, thereby precluding the possibility that the first formula is determined to be true because it is vacuously true.

Being able to verify that the specification has been entered correctly is a secondary feature. Most importantly, we can verify that the CTL machine enforces the specification's intended invariant properties. The next set of formulas are the required properties that were listed in the requirements document, restated as CTL formulas[6]

1. $AG((Off \;\&\; \sim Exit) \rightarrow \sim Ignited)$
2. $AG(((Inactive \mid Cruise \mid Override) \;\&\; \sim Exit) \rightarrow Ignited)$
3. $AG((Inactive \;\&\; \sim Exit) \rightarrow (Ignited \;\&\; (\sim Running \mid \sim Activate)))$
4. $AG((Cruise \;\&\; \sim Exit) \rightarrow (Ignited \;\&\; Running \;\&\; \sim Brake))$
5. $AG((Override \;\&\; \sim Exit) \rightarrow (Ignited \;\&\; Running))$

The first formula states that the ignition is off whenever the system is in CTL mode state OFF. This formula was found not to be invariant. The system always exits OFF when *Ignited* becomes true. However, *Ignited* is not always false upon entering OFF; since the system can initially start in mode OFF under any initial conditions. If the initial conditions are changed such that the system starts in OFF only if $\sim Ignited$ (and the system is otherwise undefined), then the first formula becomes invariantly true.

The fourth formula states that whenever the system is in CTL mode state CRUISE, then the ignition is on, the engine is running, and the brake is not being pressed. This formula was also found not to be invariant, because $\sim Brake$ is not an invariant property of CRUISE. The transition that causes the system to leave mode CRUISE when the brake is pressed is not unconditional (see Table 5); if the vehicle is going *Toofast*, then the transition is not enabled. $\sim Brake$ would be an invariant property of CRUISE if WHEN condition $\sim Toofast$ were removed from the transition or if $\sim Toofast$ were an invariant property of CRUISE. The presence of WHEN condition $\sim Toofast$ in this transition is needed to prevent the system from transitioning into OVERRIDE when it should be transitioning into INACTIVE. (The transitions into INACTIVE have priority over those into OVERRIDE.) Besides, the system really should not be in CTL mode state CRUISE if the automobile is going too fast. To make $\sim Toofast$ an invariant property of CRUISE, we must modify the specification so that *Toofast* is false whenever the system enters CRUISE (i.e., we force $\sim Toofast$ to be a WHEN condition in all transitions entering CRUISE). Since the system

[6] Again, the output proposition EXIT has been added to the formulas to specify that the system is in the mode and not exiting the mode.

unconditionally leaves mode CRUISE when *Toofast* becomes true, $\sim Toofast$ is a mode invariant of CRUISE in the modified specifications. This means that $\sim Brake$ is also an invariant property of the CRUISE in the modified specifications, and that the fourth formula stated above is invariant.

Lastly, the third formula to be verified:

$$AG\,((Inactive\ \&\ \sim Exit) \rightarrow (Ignited\ \&\ (\sim Running \mid \sim Activate))$$

was also found not be invariant. This formula states that if the cruise control is INACTIVE, then the ignition is on and either the engine is not running or the cruise control has not been activated. However, if the cruise control is INACTIVE, and a driver depresses the brake when the engine is running and then sets the cruise control lever to *Activate*, there is no transition to CRUISE because the brake is on. Thus it is possible to be in mode INACTIVE when the ignition is on, the engine is running, and the cruise control has been activated.

In fact, a presumably corrected version of this formula also did not hold:

$$AG\,((Inactive\ \&\ \sim Exit) \rightarrow (Ignited\ \&\ (\sim Running \mid Brake \mid \sim Activate))$$

Consider the above scenario, where the cruise control is INACTIVE, the engine is *Running*, the *Brake* is depressed, and the driver sets the cruise control lever to *Activate*. The system remains in mode INACTIVE because the *Brake* is being pressed. If the driver then releases the brake but continues to hold the cruise control lever in the *Activate* position, the cruise control will still remain INACTIVE and the invariant will be violated.

We believe the intended invariant properties of mode INACTIVE are that the ignition is on and either

- the engine is not running
- the brake is on
- the cruise control lever has not been activated, or
- the cruise control was activated, but not at a time when the other cruise control conditions held.

The mode invariant needs to be changed to address this fourth case, which differs from the first three in that it deals with the values of the variables *at a particular time* (when the cruise control lever is changed and set to *Activate*). These properties can be expressed by the following formula

$$AG\,((Inactive\ \&\ \sim Exit\ \&\ \sim Activate\ \&\ Ignited\ \&\ Running\ \&\ \sim Brake) \rightarrow \sim EX(Inactive\ \&\ \sim Exit\ \&\ Activate\ \&\ Ignited\ \&\ Running\ \&\ \sim Brake))$$

which states that if the system is in mode INACTIVE and the cruise control WHEN conditions are satisfied (the engine *Ignited* and *Running* and the *Brake* released) 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 all of the properties were true in the next state, then the transition from INACTIVE to CRUISE would be triggered and the system would actually be in mode CRUISE.

The MCB model checker determines that the above formula is false with respect to our CTL machine depicted in Figure 2, but this is due to a weakness in our model. Our representation only characterizes the conditions that cause mode transitions; it does not characterize the conditions under which a system remains in a mode. According to our definition, a mode invariant is the set of properties that are always satisfied upon entering the mode and whose failure causes the system to unconditionally leave the mode. Using this definition, the mode invariant of INACTIVE is

$$AG\,((Inactive\&Exit) \rightarrow Ignited)$$

This formula states that if the system is in mode INACTIVE, then all we know is that the ignition is on. This formula is verifiably invariant.

Water Level Monitoring System:

Current Mode	InsideHysRange	WithinLimits	SlfTstPressed	SlfTestInterval	TestInterval	ResetInterval	ShutdownLockTime	New Mode
Standby	t	**t**	–	–	–	@T	–	Operating
	–	–	**t**	@T	–	–	–	Test
Operating	**f**	@F	f	**f**	–	–	–	Shutdown
	–	–	**t**	@T	–	–	–	Test
Shutdown	@T	**t**	f	**f**	–	–	f	Operating
	–	–	f	**f**	–	–	@T	Standby
	–	–	**t**	@T	–	–	–	Test
Test	–	–	–	–	@T	–	–	Standby

Initial Mode: Standby (~SlfTestInterval & ~TestInterval & ~ResetInterval & ~ShutdownLockTime)

Table 6: Mode transitions for water level monitoring system.

3.2 Case Study: Water-Level Monitor

The requirements specification of a water-level monitoring system (WLMS) used in this study comes from [27]. The system consists of two modeclasses, one that describes system behavior when the system is operating correctly, and one that describes the behavior when the system has failed. The **FAILURE** modeclass is simple and uninteresting, and exists independently of the **OPERATING** modeclass; therefore, we only studied the **OPERATING** modeclass. Table 6 contains the mode transition table. **OPERATING** is comprised of four modes:

OPERATING	The system is running
STANDBY	The system has been stopped, but has not failed
SHUTDOWN	The water level has fallen outside allowable limits, which may cause the system to halt if the water level is not restored to within the hysteresis range within 200ms
TEST	The system is being tested

The system always starts in mode STANDBY.

The system conditions that appear in the table are defined as follows

WithinLimits	LowLimit<WaterLevel<HighLimit
InsideHysRange	(LowLimit+0.5cm)<WaterLevel<(HighLimit-0.5cm)
SlfTestPressed	**SelfTst** button is being pressed
SlfTestInterval	**SelfTst** button pressed constantly for > 500ms
TestInterval	System in TEST mode for > 14s
ResetInterval	**Reset** button pressed constantly for > 3s
ShutdownLockTime	System in SHUTDOWN mode for > 200ms

The last four conditions are timing conditions; all of them must be false when the system is initiated.

CTL Machine Construction

Like the cruise control specifications, the WLMS mode transition table needed to be formalized before it could be analyzed. The following two relationships could be deduced from the descriptions of the

conditions, found in the section of the requirements document that specified the environmental state variables:

$InsideHysRange \Rightarrow\!\!> WithinLimits$
$SlfTestPressed < SlfTestInterval$

The first relationship states that if the water level is inside the hysteresis water-level range, then it is certainly within the allowable water-level range. The second relationship states that conditions *SlfTestPressed* and *SlfTestInterval* represent the same environmental state variable (the **SelfTest** button), but that condition *SlfTestInterval* represents the state variable holding its value for a longer period of time than does condition *SlfTestPressed*. The changes to the tabular specifications due to the above condition-relationships appear in boldface in the mode transition table (Table 6).

The requirements designer of the water-level monitoring system used a specification model that does not allow sequences of simultaneous mode transitions. The model regards the current mode as an additional WHEN condition of any transition event. Since the WHEN conditions of an event must hold for some finite period of time before the event occurs, no mode transition is immediately enabled upon entry into the transition's source mode.

The transformation algorithm warns that the two transitions leaving mode STANDBY can be simultaneously enabled, indicating that the specification is nondeterministic. However, since the requirements specification never listed determinism as one of the system's requirements, we left the specifications nondeterministic. The resulting CTL machine is displayed in Figure 3.

Analysis

First, we tested to see if we had entered the tabular specifications correctly by verifying the CTL-formula representation of all the mode transitions. Then we tried to verify expected invariant properties of the system. The software requirements document for the WLMS system did not include a list of safety properties. We inferred from the system description that the following properties should be invariant.

1. $AG\,((SlfTestInterval\ \&\ {\sim}Test) \rightarrow AX(Test))$
2. $AG\,((Standby\ \&\ {\sim}Exit) \rightarrow {\sim}SlfTestInterval)$
3. $AG\,((Operating\ \&\ {\sim}Exit) \rightarrow ({\sim}SlfTestInterval\ \&\ (WithinLimits \mid SlfTestPressed)))$
4. $AG\,((Shutdown\ \&\ {\sim}Exit) \rightarrow ({\sim}SlfTestInterval\ \&$
   $((\sim InsideHysRange\ \&\ \sim ShutdownLockTime) \mid \sim SlfTestPressed)))$

We corroborated our inferred invariants with the requirements designer [28].

The first formula states that if the **SelfTst** button has been pressed for 500 ms or more and the system is not currently in mode TEST, then the system will be in mode TEST after the next transition. This formula is not invariant. Refering to Table 6, if the machine is in STANDBY and both of the allowable transitions are simultaneously enabled, the CTL machine may nondeterministically choose the transition to OPERATING rather than the one to TEST. Thus, the system's nondeterminism poses problems for the specification. To correct this, we made the specification deterministic by giving priority to the transition into TEST (giving priority to the transition into OPERATING would violate the formula). Adding WHEN condition $\sim$*SlfTestInterval* to the transition from STANDBY to OPERATING ensures that the system will always transition into TEST when the **SelfTest** has been pressed long enough, thereby making the first formula a system invariant.

The second formula is an apparent consequence of the first formula. It states that $\sim$*SlfTestInterval* is a mode invariant of STANDBY. However, this formula was found not to be invariant. The system unconditionally exits mode STANDBY when *SlfTestInterval* becomes true; however it is not invariantly

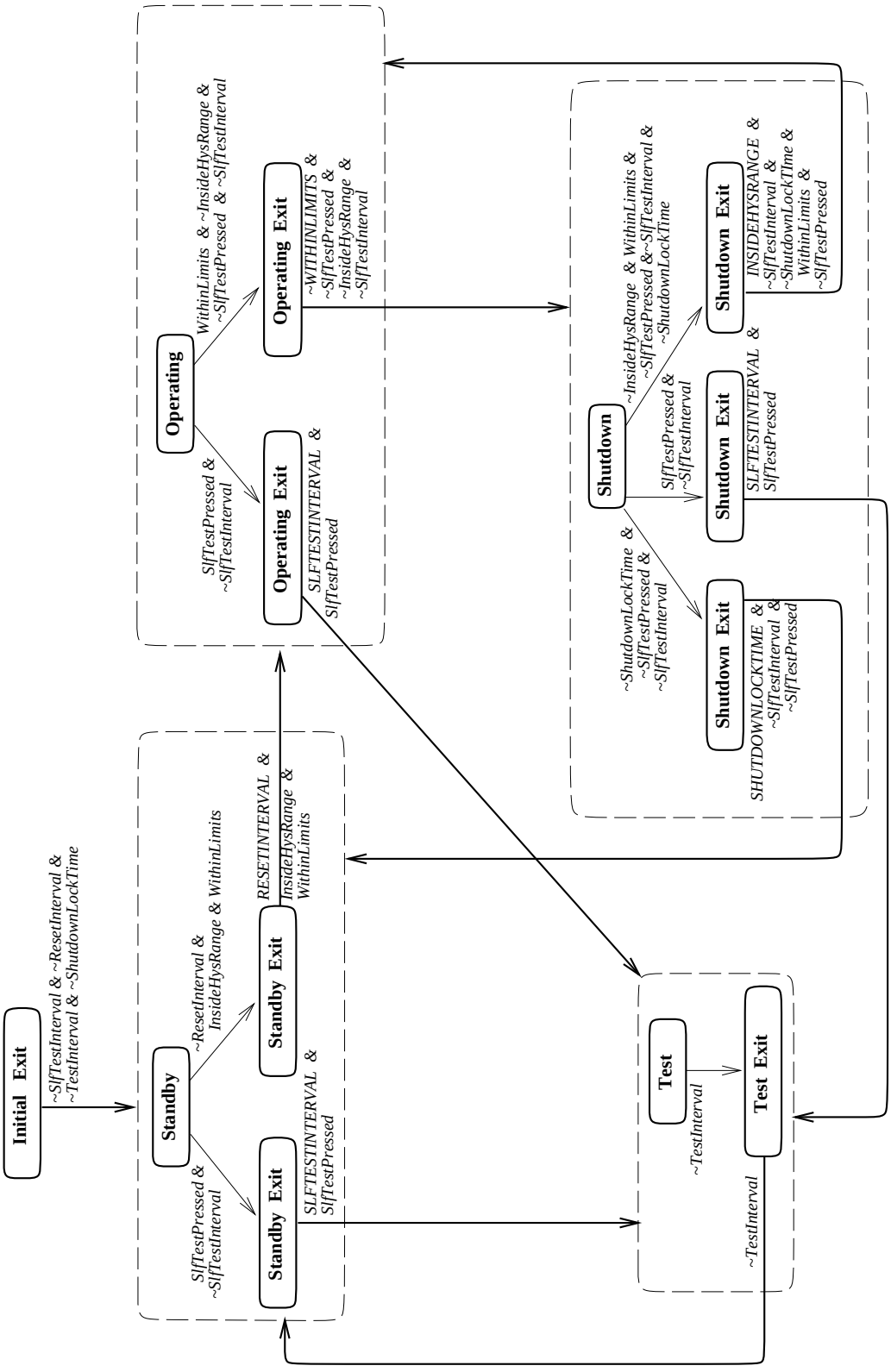


Figure 3: CTL machine for adjusted water-level monitoring system.

true that *SlfTestInterval* is false upon entry into STANDBY. If the operator presses the **SelfTst** button when the system is in mode TEST, it is possible the button will have been pressed long enough to make *SlfTestInterval* true before the system leaves mode TEST and enters STANDBY. Adding WHEN condition $\sim$*SlfTestInterval* to the transition from TEST to STANDBY will ensure that *SlfTestInterval* is always false upon entering STANDBY and will make the second formula invariant. However, this additional WHEN condition can cause the system to deadlock in mode TEST; if the operator is pressing the **SelfTst** button when *TestInterval* becomes true, not only will the transition leaving TEST not be activated but it will never be activated, since its triggered condition will never be satisfied again. To avoid this we add a second transition from TEST to STANDBY (not shown in Table 6) that is activated when the operator releases the **SelfTst** button if the system has been in mode TEST for more than 14 seconds (the transition event has triggered condition @F(*SlfTestPressed*) and WHEN condition *TestInterval*). The two new mode transitions ensure that *SlfTestInterval* is invariantly false in mode STANDBY.

The third formula is more critical. It states that if the system is OPERATING, then the **SelfTst** button has not been pressed for more than 500 ms (if at all) and either the water level is *WithinLimits* or the **SelfTst** button is being pressed. This property was also found not to be invariant. If the system is OPERATING and the **SelfTst** button is being pressed, then the transition into mode SHUTDOWN is disabled; under these circumstances, the system will remain in the OPERATING mode if the water level rises or falls outside the allowable limits. However, if the **SelfTst** button is released before the system transitions into TEST, then the system remains in the OPERATING mode even though the water level is not *WithinLimits*. In addition, the transition from OPERATING to SHUTDOWN is now disabled because the event of the water level crossing the *WithinLimits* boundary has already occurred and can no longer be detected. The transition from OPERATING to SHUTDOWN should not depend on whether the **SelfTst** button is being pressed but rather on whether it has been pressed long enough to enable the transition into TEST. Thus, $\sim$*SlfTestInterval* should be a WHEN condition in the transition from OPERATING to SHUTDOWN, and WHEN condition $\sim$*SlfTestPressed* should be removed. If we make this change to the tabular specifications, then we need to change the intended mode invariant to

$$AG\,((Operating\ \&\ \sim Exit) \rightarrow (WithinLimits\&\ \sim SlfTestInterval)$$

since whether or not the button is being pressed is no longer important.

The fourth formula is also a safety-critical property. It states that if system is in mode SHUTDOWN, then either the water-level is still outside the hysteresis water-level range and the system has been in mode SHUTDOWN for less than 200 ms or the **SelfTst** button is being pressed. Again, the system will ignore events @T(*InsideHysRange*) and @T(*ShutdownLockTime*) if the **SelfTst** button is being pressed; and if this button is then released before the system transitions into TEST, then the system could deadlock in global mode SHUTDOWN. As before, the transitions from SHUTDOWN to OPERATING and STANDBY should not depend on whether the **SelfTst** button is being pressed but rather on whether it has been pressed long enough to activate the transition into TEST. Thus, we add $\sim$*SlfTestInterval* as a WHEN condition to these transitions and remove WHEN condition $\sim$*SlfTestPressed*; we also modify the intended mode invariant so that it no longer references *SlfTestPressed*.

$$AG\,((Shutdown\ \&\ \sim Exit) \rightarrow (\sim InsideHysRange\ \&\ \sim ShutdownLockTime\&\ \sim SlfTestInterval))$$

4 Discussion

In both case studies, we used the model checker to disprove the invariance of several required safety properties. It has been observed [29] that many of the discrepancies in the cruise control specification would have been avoided if the system had originally been specified using the more recent SCR tables [12] (which we used in this paper) rather than the original SCR tables [19]; the columns of conditions in the tables help remind the requirements designer of all the conditions that need to be considered.

Other discrepancies between the tabular specifications and the intended global assertions were subtle and would have been difficult to notice by inspection even if the requirements had been stated using the better specification language. In our original experiments, in which the transformation from the requirements to the CTL machine was done by hand, two intended mode invariants (those for modes OFF and CRUISE) were thought to be enforced by the requirements [3]; these discrepancies were not detected until the algorithm was automated.

Furthermore, all of the discrepancies in the water-level monitoring system involved unexpected combinations of events (e.g. the operator presses conflicting buttons at the same time, or presses the **SelfTst** button when the system is already in TEST mode) or race conditions (e.g., the water level drops below desired limits after the **SelfTst** button has been pressed but before it has been pressed long enough for the system to enter mode TEST). As a result, these discrepancies went unnoticed even though the system was implemented and tested.

The remainder of this section addresses issues about our analysis technique and discusses related work.

4.1 Observations

The MCB model checker has proven useful for verifying safety properties in the case studies presented. There are, however, limitations to this approach.

- The MCB model checker is restricted to the study of finite state machines (FSM). The system being developed need not be a FSM, but the requirements specification that describes the transitional behavior of the system must be expressible as a FSM.

- In this paper, we analyzed system requirements consisting of one modeclass. However, a system's requirements may consist of multiple modeclasses, and as such would be modeled by the concurrent execution of several CTL machines. Such a specification is composed into a single global specification and transformed into a global CTL machine, in which each state represents the combined current states of the machine's components. This leads to an exponential increase in the number of states. However, since we start with a relatively small number of states (CTL states represent system modes rather than system states), the size of the global machine might still be manageable. In addition, tools and techniques are being developed to tackle the larger problem. A symbolic model checker capable of checking very large CTL machines is described in [7]. Furthermore, [9] describes a compositional model checking method, whereby one can correctly assume that certain properties verified against the specification's components are also true of the global specification.

4.2 Related Work

Model checking is only one technique for analyzing requirements specifications. Other promising approaches include formal verification, executable specifications, and theorem proving.

One approach is to introduce a specification language based on temporal logic and provide a proof system for that logic [22, 23, 25]. The requirements designer specifies a system's behavior as a set of temporal logical formulas, and then proves system properties using the logic's proof rules. Such formal verification ensures a high degree of confidence in the validated system. It is a laborious process, however, that entails detailed mathematical analysis, most of it unautomated. As a result of the high cost, few real-world systems have been formally verified.

A more popular approach is to define an executable specification language, which allows the designer to run the specification and test that the specified system works correctly [10, 15, 16]. STATEMATE [17], for example, is a programming environment for graphically specifying reactive systems. In addition to simulation capabilities, the STATEMATE system offers a set of dynamic tests that can be performed

automatically: consistency, reachability, nondeterminism, and deadlock. However, none of these tests can be used to analyze or verify the functional behavior of the system being specified.

Modechart [20] is a variant of the STATEMATE language that was developed to incorporate timing requirements into a system's requirements specification. The primary use of this system is to confirm that timing constraints are enforced in the system's requirements. However, unless the components of the system are tightly coupled, the computation graph of the system (which explicitly represents all execution paths in the system) is unmanageablely large for all but the smallest specifications.

Hierarchical multi-state (HMS) machines [13, 14] is another specification formalism, in which the system behavior is expressed as a hierarchical state machine. One can model check an HMS machine by expanding the system into a computation tree and verifying a temporal logical formula with respect to the computation tree. Alternatively, one can use a variation of the resolution-based theorem proving technique introduced in [4]: the property to be verified is negated and added to the system specification as an extra state; if this extra state is unreachable, then the system property is invariant. Unfortunately, neither of these verification techniques has been automated.

5 Conclusion

We have presented a technique whereby SCR-style event-oriented requirement specifications can be modeled as state-based structures and analyzed using a state-based model checker. The result is a formal method whose specification language is intuitive and scalable, and whose analysis is automated. The only aspect of this methodology that requires any real mathematical aptitude is the phrasing of global properties as temporal logic formulas. Currently, we are investigating ways of extending the SCR requirements language and the CTL model checker so that timing requirements can also be specified and analyzed.

Acknowledgements

C. L. Heitmeyer and B. Labaw made valuable technical contributions to this work. D. Lamb, J. van Schouwen, and M. Zelkowitz helped in making this paper more presentable. V.S. Subrahmanian contributed his time and library of logic papers.

We would also like to thank the referees for their careful reading and perceptive comments.

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, University of Maryland, College Park, 1992.

[3] J. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". In *Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pages 16–28, 1991.

[4] W. Bledsoe and L. Hines. "Variable Elimination and Chaining in a Resolution-based Prover for Inequalities". In *Proceedings of 5th Conference on Automated Deduction: Lecture Notes in Computer Science*, pages 70–87, 1980.

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

[6] 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.

[7] 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.

[8] 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.

[9] E. Clarke, D. Long, and K. McMillan. "A Language for Compositional Specification and Verification of Finite State Hardware Controllers". *Proceedings of the IEEE*, 79(9):1283–1292, September 1991.

[10] M. Degl'Innocenti, G. Ferrari, G. Pacini, and F. Turini. "RSF: A Formalism for Executable Requirements Specifications". *IEEE Transactions on Software Engineering*, 16(11):1235–1246, November 1990.

[11] E. Emerson and E. Clarke. "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons". *Science of Computer Programming*, 2:241–266, 1982.

[12] S. Faulk. *State Determination in Hard-Embedded Systems.* PhD thesis, Department of Computer Science, University of North Carolina, Chapel Hill, 1989.

[13] A. Gabrielian and M. Franklin. "Multilevel Specification of Real-Time Systems". *Communications of the ACM*, 34(5):51–60, May 1991.

[14] A Gabrielian and R. Iyer. "Integrating Automata and Temporal Logic: A Framework for Specification of Real-Time Systems and Software". In *Proceedings of the Unified Computation Laboratory*, 1990.

[15] C. Ghezzi, D. Mandrioli, and A. Morzenti. "TRIO, a logic language for executable specifications of real-time systems". *Journal of Systems and Software*, 12(2):107–123, May 1990.

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

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

[18] C. Heitmeyer and B. Labaw. "Consistency Checks for SCR-Style Requirements Specifications". (in preparation).

[19] 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.

[20] F. Jahanian and A. Mok. "Safety Analysis of Timing Properties in Real-Time Systems". *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[21] 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.

[22] Z. Manna and A. Pnueli. "Tools and Rules for the Practicing Verifier". Technical Report STAN-CS-90-1321, Department of Computer Science, Stanford University, 1990.

[23] J. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press LTD., 1989.

[24] D. Parnas, D. Smith, and T. Pearce. Making Formal Software Documentation More Practical: A Progress Report. Technical Report TR-88-236, Department of Computing and Information Science, Queen's University, 1988.

[25] A. Pnueli. "The Temporal Logic of Programs". In *Proceedings of 18th Annual Symposium on the Foundation of Computer Science*, pages 46–57, 1977.

[26] S. Smith and S. Gerhart. "STATEMATE and Cruise Control: A Case^2 Study". In *Proceedings of COMPAC '88, 12th Int. IEEE Computer Software and Application Conference*, pages 49–56, 1988.

[27] 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.

[28] J. van Schouwen. Private communications., 1991. (May 1991 - August 1991).

[29] D. Weiss. Private communications., 1992. (August 1992).