ACM Copyright Notice

© ACM 1991
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Published in: ***Proceedings of the ACM SIGSOFT Conference on Software for Critical Systems (SIGSOFT'91),*** December 1991

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

Cite as:

Joanne Atlee and John Gannon. 1991. State-based model checking of event-driven system requirements. In Proceedings of the conference on Software for Critical systems (SIGSOFT '91). ACM, New York, NY, USA, 16-28.

BibTex:

```
@inproceedings{Atlee:1991:SMC:125083.123047,
 author = {Atlee, Joanne and Gannon, John},
 title = {State-based Model Checking of Event-driven System Requirements},
 booktitle = {Proceedings of the Conference on Software for Critical Systems},
 series = {SIGSOFT '91},
 year = {1991},
 pages = {16--28}
}
```

DOI: http://dx.doi.org/10.1145/125083.123047

State-Based Model Checking of Event-Driven System Requirements

Joanne Atlee
John Gannon

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

In this paper, we demonstrate how model checking can be used to verify safety properties for event-driven systems. We present a technique for transforming event-oriented software requirements into state-based structures, which we can then analyze using a state-based model checker. 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)[21]. Designers must be able to formally analyze requirements before system design begins.

The requirements specification is a behavioral specification of the system's activities; it describes the system's modes of operation and specifies 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 are techniques that have been used successfully to verify safety properties in hardware systems [4,9]. 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. Section 2 reviews event-oriented requirements specifications and state-based model checking, and presents a method 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 event-oriented requirements and state-based systems, and present a method by which properties of an event-oriented system can be simulated by a state machine.

2.1 Event-Driven Requirements Specifications

Our studies concentrated on techniques for analyzing SCR-style (Software Cost Reduction) requirements specifications [1,16,17]. A system's behavioral requirements are abstractly specified as a set of finite *mode*-machines that execute concurrently:

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

The system is in exactly one mode of each mode class at all times. Informally, the modes and transitions in each mode class form a mode-machine that describes one aspect of a system's behavior, and the transitional behavior of the entire system is defined by the composition of all the system's mode-machines.

Conditions are boolean state variables, and a system's state space is the set of all possible combinations of its conditions' values. By partitioning the state space

Current Mode	Request1	Request2	**New Mode**
Empty	@T	–	InUse1
	–	@T	InUse2
InUse1	@F	f	Empty
	@F	t	InUse2
InUse2	f	@F	Empty
	t	@F	InUse1

Table 1: Mode transition table for monitor.

into modes, we reduce the size of the requirements specification. The result is a higher-level specification that abstracts away details that do not contribute to a system's local behavior; the values of all variables are not important at all times.

In SCR requirements specifications, variable values are only important when they can effect mode transitions. A mode transition is activated by the occurrence of an *event*, which represents the point in time when a condition's value changes. For example, event

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

occurs when condition *Cond1* becomes true. Similarly, event @F(*Cond1*) signifies a time when condition *Cond1* becomes false. The occurrence of an event might depend on the values of other conditions. For example, event

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

occurs when condition *Cond1* becomes true *while* condition *Cond2* is true. More complex events can be created from simpler events and conditions using boolean operators.

A mode *transition condition* specifies the event that triggers the transition. Two transitions from the same mode are simultaneously enabled if their trigger events occur at the same instant. In such a case, the mode-machine is nondeterministic, and the activation of either transition (but not both) satisfies the requirements.

Table 1 is a requirements specification for a monitor that encapsulates data shared by two processes. The specification consists of a single mode class comprised of three modes: Empty, meaning that the data protected by the monitor is not being used; InUse1, meaning that **process1** currently has access to the shared data; and InUse2, meaning that **process2** has access to the shared data. The system always starts in mode Empty. Conditions *Request1* and *Request2* indicate whether or not **process1** and **process2**, respectively, have requested access to the protected data. Since the monitor can only be in one mode at a time, mutual exclusion is ensured.

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 specification condition (state variable). Upper-case letters '@T' and '@F' signify that the condition must change (to new value true or false, respectively) to activate the mode transition. Lower-case letters ('t' and 'f') signify that the condition must have a particular value (true or false, respectively) for the event to occur. If a condition does not affect a particular mode transition, then its value in the table is marked with a hyphen ('–'). For example, if the system is in mode Empty when condition *Request1* becomes true, it can transition into mode InUse1 without regard to the value of condition *Request2*. The system leaves mode InUse1 when *Request1* becomes false. If *Request2* is true when the system leaves mode InUse1, then the system will transition into mode InUse2; otherwise the system will transition into mode Empty. The two transitions from mode Empty can be simultaneously enabled if both processes request access to the shared data at the same time. The requirements are therefore nondeterministic, and the system designer (not the requirements designer) will be responsible for deciding how to choose between the two transitions.

The tabular format of SCR-style requirements specification is both easy to write and easy to understand. Even so, a requirements designer will often augment the behavioral specification with a set of global constraints on the system's behavior. The purpose of the global constraints is to give a compact view of a system's invariant properties that may otherwise be difficult to extract from a behavioral specification. Sometimes, the global constraints are explicitly included in the requirements document, though they may not be expressed mathematically; their format can range from logical formulas [19] to natural language sentences [1]. Other times, they are not included in the requirements document, but are implicit assumptions made by the requirements designer [26,27]. The safety assertions of our monitor example are:

$$\begin{aligned}&\textsc{Empty} \Rightarrow (\neg \mathit{Request}1 \wedge \neg \mathit{Request}2)\\&\textsc{InUse}1 \Rightarrow \mathit{Request}1\\&\textsc{InUse}2 \Rightarrow \mathit{Request}2\end{aligned}$$

These formulas state that whenever the system is in a particular mode (Empty), certain system conditions have invariant values (*Request1* and *Request2* are false).

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 that already exists in the behavioral specification. We want to use this redundancy to detect inconsistencies between a system's requirements specification and its expected invariant properties.

2.2 State-Based Model Checking

We used an improved version of Clarke's EMC model checking system [6], called MCB [3], as our model checker. 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 the MCB 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).

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). This 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 [6] and are simply summarized below:

1) Every atomic proposition[1] 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 monitor example are represented by the following CTL formulas:

$$AG(Empty \rightarrow (\sim Request1 \ \& \ \sim Request2))$$
$$AG(InUse1 \rightarrow Request1)$$
$$AG(InUse2 \rightarrow Request2)$$

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

MCB 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, presumably, in the system requirements.

2.3 Mapping SCR Requirements onto CTL Structures

In our experiments, we mapped SCR-style mode-machines onto representative CTL structures. This section describes how to transform such requirements into CTL machines.

Most elements of the SCR requirements model correspond naturally to elements of CTL structures:

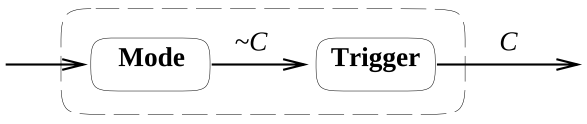
Requirements	CTL structure
modes	states
mode transitions	state transitions
conditions	input variables

There is no natural modeling of events because of the differences between mode transitions and state transitions. CTL state transitions occur based on the current state and the current values of the input conditions. Mode transitions, on the other hand, occur simultaneously with events; the system spends zero time in a mode once one of its transitions 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.

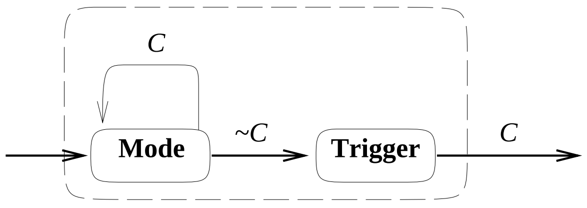
To model events, we can represent modes as two CTL states: a *mode state* and a *trigger state*. The mode state represents the system *in a mode*, and all transitions into the original system mode are modeled as transitions into the representative mode state. The trigger state represents the system *leaving a mode*, all transitions leaving the original mode are represented as transitions leaving the associated trigger state. We annotate the trigger state's transitions with the original mode's transition conditions, and we annotate the transition from the mode state into the trigger state with the *negation* of the transitions' trigger events. For example, for mode transition condition T(A)WHEN[B], the the mode state would be annotated with transition condition $\sim A$, and the trigger state would be annotated with transition condition $A \& B$. This representation captures the event of condition A changing from value false (in the mode state) to value true (in the trigger state).

Unfortunately, this intuitive representation does not model all system modes: in the above example, condition A is represented as always being false in the mode state, which may not be accurate. The following set of examples demonstrate how modes that have complex transition conditions can be transformed into state-based representations.

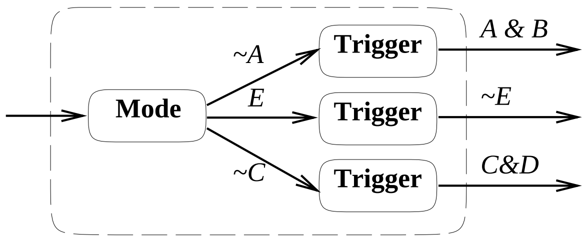
Example A: If a transition condition is a simple event @T(C) and condition C is never true upon entering the mode, then we can model the event by annotating the mode state with transition condition $\sim C$ and annotating the trigger state with transition condition C:



Example B: In the above example, our mode representation forces condition C to be false in the mode state. The input conditions in a CTL state must always satisfy one of the state's transition conditions. Therefore, since the mode state in the above example has only one transition, with transition condition ($\sim C$), condition C must always be false. This is an invalid representation if condition C can be true when the system enters the mode (the transition is only triggered if C *becomes* true while in the mode). To allow condition C to be true in the mode state, but still force C to be false immediately before the system leaves the mode, we add a transition from the mode state back to itself and annotate it with condition C:

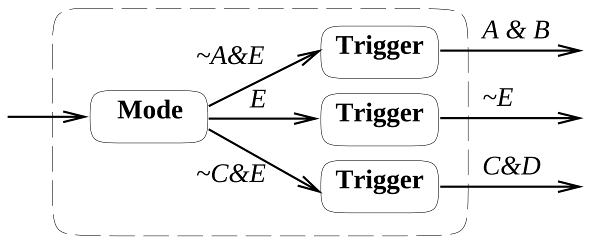


Example C: If a mode has multiple outgoing transitions, multiple trigger states may be needed to distinguish between events. For example, if a mode's transition conditions are @T(A)WHEN[B], @T(C)WHEN[D], and @F(E), then three trigger states are needed to represent the three distinct events @T(A), @T(C), and @F(E):



This representation forces condition A to be false in the mode state, *if* the system exits the mode because event @T(A)WHEN[B] occurs; conditions C and E can have either value in the mode state, given this sequence of events. Likewise, condition C must be false in the mode state if the system leaves the mode when event @T(C)WHEN[D] occurs; conditions A and E can have either value in the mode state, given this sequence of events. There is no need for a transition from the mode state back to itself, since the mode state's transition conditions allow the conditions A, C, and E to be either true or false in the mode state.

Example D: If condition E (in Example C) is always true whenever the system enters the mode, then our representation of the mode is not accurate. If E is true upon mode entry, then it can never have value false, since the event of E becoming false would trigger a transition out of the mode. However, the above representation allows condition E to have any value in the mode state. The mode representation can be corrected by appending condition E to all of the transitions from the mode state to the trigger states:



This representation ensures that condition E is invariantly true when the system is in the mode.

Figure 1 contains the CTL machine representation of the monitor requirements specification presented in Table 1. Monitor mode EMPTY is represented by a mode state EMPTY and two trigger states EMPTY ENABLED. The states' attributes indicate which system mode is represented by the state (EMPTY), plus whether or not the state is a trigger state (ENABLED). The trigger states' transition conditions are copied directly from the specification table. For example, the left-most EMPTY ENABLED state is annotated with transition condition *Request1*, representing the first transition leaving mode EMPTY in Table 1. The mode states' transition conditions represent the negation of the mode's $@T$ and $@F$ transition conditions. For example, state EMPTY is annotated with two transition conditions $\sim$*Request1* and $\sim$*Request2*, which indicate that neither transition out of mode EMPTY is enabled.

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

1. Create an input condition for each state variable in the requirements specification.[2]

[2] One can also create input conditions to represent the values of first-order predicate conditions, such as integer ranges and timing constraints, as long as there is a finite number of such predicates.

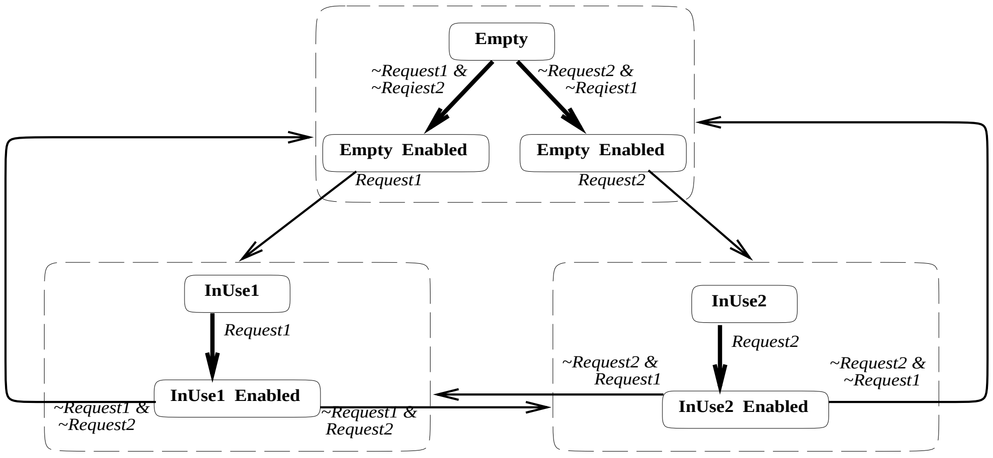


Figure 1: MCB Machine for monitor specifications.

2. Create an attribute proposition for each system mode in the requirements specification.
3. Create attribute *Enabled*, which will be used to indicate trigger states.
4. For each system mode described in the requirements:
   - Create a mode state and annotate it with the associated attribute proposition.
   - Create a unique trigger state for each condition C whose changing value (@T(C) or @F(C)) triggers a mode transition. Annotate each trigger state with the both the attribute representing the system mode and the attribute *Enabled*.
   - Annotate each trigger state with all of the mode transitions activated by a particular condition changing value. For example, annotate one trigger state with all of the transitions enabled by event @T(C).
   - Annotate the mode state with transitions leading to all of the trigger states; the enabling condition of each transition is the negation of the event that the destination trigger state represents. For example, if a trigger state represents those mode transitions activated by event @T(C), then the enabling condition of the transition from the mode state to that trigger state is $\sim C$, thereby capturing the event of condition C becoming true.
   - If all of a mode's transitions are activated by the same event @T(C) (or event @F(C)), and if condition C can be true (false) on mode entry, then add a transition from the mode state back to itself and annotate it with condition C ($\sim C$). Otherwise, the mode representation will restrict the value of condition C to being invariantly false (true) in the mode state.
   - For all simple mode transition conditions (that have no WHEN conditions) in which the condition always enters the mode negated (i.e., if the trigger event is @T(C) and C is always false upon mode entry), append the negation of the trigger event ($\sim C$) to all transition conditions between the mode state and the trigger states. Otherwise, the mode representation will allow condition C to have any value in the mode state, which would not be accurate.
5. For each mode transition from mode M1 to M2 in the requirements specification
   - find the particular trigger state of M1 that represents the mode transition's trigger event @T(C),
   - create a state transition from that trigger state to the mode state of mode M2, and
   - associate the enabling condition of the mode transition with the new state transition

All of the CTL structures presented in this paper are represented graphically, as in Figure 1. The ASCII notation used by the model checker is described in [3].

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

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	–	–	

Table 2: Mode Transitions for Automobile Cruise Control

of SCR requirements [16]; in this paper the specifications appear in the new SCR mode table format[10], which is easier to read and understand. We transformed these systems' requirements into CTL structures, 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 [19]. 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 is on and the cruise control system is on, controlling the automobile's speed.
OVERRIDE	Ignition is on and the cruise control system is on, but not controlling the automobile's speed.

The system always starts in mode OFF.

Table 2 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 vehicle when the 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
Resume	Cruise control lever is set to RESUME

The requirements document for the cruise control system also lists the following safety properties[3]:

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

While constructing the CTL machine, we noticed that the mode transition table in Table 2 contained implicit information, making it difficult to map SCR requirements onto CTL structures automatically.

The first type of implicit information consists of interrelationships between conditional values. For example, the cruise control lever cannot be simultaneously

[3]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*.

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

Table 3: Mode Transitions for adjusted cruise control specifications

set at values *Activate*, *Deactivate* and *Resume*. Therefore, if a mode transition is dependent upon condition *Activate* being true, then it is also dependent upon conditions *Deactivate* and *Resume* being false. The fact that these additional conditions do not normally appear in a mode transition table shows the difference between the minimal amount of information needed to specify a system's requirements and the additional information needed to verify these same requirements. We detected the following relationships between system conditions:

$Running \Rightarrow Ignited$
$Activate \Rightarrow (\neg Deactivate \wedge \neg Resume)$
$Deactivate \Rightarrow (\neg Activate \wedge \neg Resume)$
$Resume \Rightarrow (\neg\, Activate \wedge \neg\, Deactivate)$

A modified specification of the cruise control system is shown in Table 3. The additional mode transition conditions due to the above variable interrelationships appear in italics.

The second type of implicit information involves sequences of instantaneous mode transitions. For example, if the system is in mode Cruise, and the driver uses his brakes at the same time as the engine fails, then transitions to modes Inactive and Override are both enabled. If the system nondeterministically transitions to the latter mode, then (since mode transitions take zero time to complete) the system will be in mode Override at the instant the engine fails, causing a subsequent transition to mode Inactive. Thus, despite the apparent non-determinism in this scenario, the system will actually spend zero time in the intermediate mode Override and will always end up in mode Inactive.

A STATEMATE specification for a similar cruise control system [25] avoids sequences of instantaneous mode transitions 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[4]. 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 Resume. 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 3 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 appear as bold italic characters in Table 3.

The CTL machine that corresponds to the adjusted

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

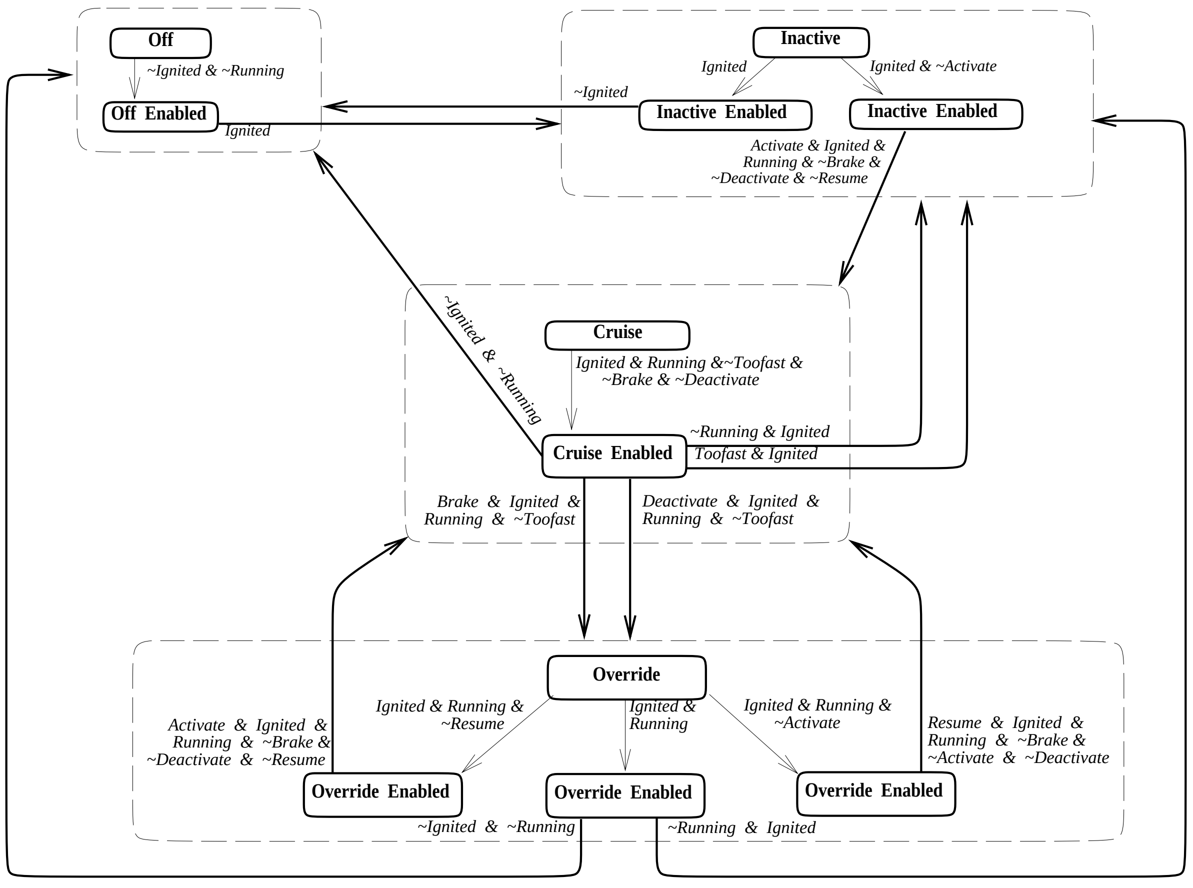

Off
~Ignited & ~Running
Off Enabled
Ignited
~Ignited
Inactive
Ignited
Ignited & ~Activate
Inactive Enabled
Inactive Enabled
Activate & Ignited & Running & ~Brake & ~Deactivate & ~Resume
~Ignited & ~Running
Cruise
Ignited & Running &~Toofast & ~Brake & ~Deactivate
Cruise Enabled
~Running & Ignited
Toofast & Ignited
Brake & Ignited & Running & ~Toofast
Deactivate & Ignited & Running & ~Toofast
Override
Ignited & Running & ~Resume
Ignited & Running
Ignited & Running & ~Activate
Activate & Ignited & Running & ~Brake & ~Deactivate & ~Resume
Resume & Ignited & Running & ~Brake & ~Activate & ~Deactivate
Override Enabled
Override Enabled
Override Enabled
~Ignited & ~Running
~Running & Ignited

Figure 2: MCB Machine for adjusted cruise control specifications

cruise control specifications is shown in Figure 2.

Analysis

Once the CTL machine has been created, we need to verify that the machine and the propositional values were entered correctly. To do this, we rephrase the requirements specifications as safety properties and use the MCB model checker to prove that they are true with respect to the constructed machine.

For each CTL state, we verified that the system properties associated with that state (displayed as state annotations in Figure 2) were true whenever the system was in that state. The following formulas represent the properties for the CRUISE and CRUISEENABLED states:

$AG(Cruise \rightarrow (Ignited \,\&Running\,\& \sim Toofast \,\& \sim Brake$
$\& \sim Deactivate))$
$AG((Cruise\,\&Enabled) \rightarrow ((\sim Ignited \,\& \sim Running) \mid$
$(\sim Running\,\&Ignited) \mid (Toofast\,\&Ignited) \mid$
$(Brake\,\&Ignited\,\&Running\,\& \sim Toofast) \mid$
$(Deactivate\,\&Ignited\,\&Running\,\& \sim Toofast)$

The next set of formulas were listed in the requirements document as invariants of the system:

$AG(Off \rightarrow \sim Ignited)$
$AG((Inactive \mid Cruise \mid Override) \rightarrow Ignited)$
$AG(Inactive \rightarrow (Ignited \,\& (\sim Running \mid \sim Activate))$
$AG(Cruise \rightarrow (Ignited \,\& Running \,\& \sim Brake))$
$AG(Override \rightarrow (Ignited \,\& Running))$

One of these invariants did not hold:
$AG(Inactive \rightarrow (Ignited \,\& (\sim Running \mid \sim Activate))$
If a driver depresses the brake when the engine is running in INACTIVE mode and then sets the cruise control lever to *Activate*, the specifications prevent a mode 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 invariant also did not hold:
$AG(Inactive \rightarrow (Ignited \,\&(\sim Running \mid Brake \mid \sim Activate))$
Consider the above scenario, where the system is in mode 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 con-

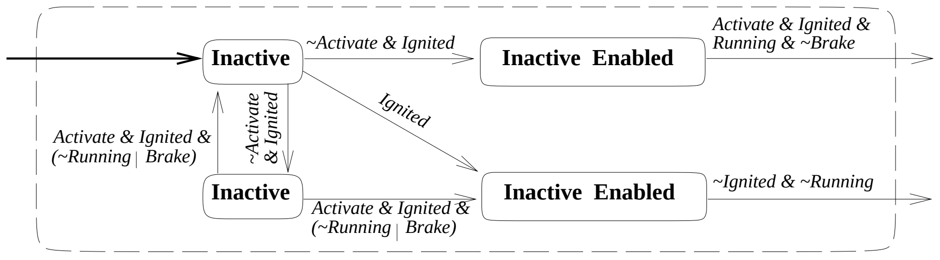

Inactive
~Activate & Ignited
Inactive Enabled
Activate & Ignited & Running & ~Brake
Ignited
~Activate & Ignited
Activate & Ignited & (~Running | Brake)
Inactive
Inactive Enabled
~Ignited & ~Running
Activate & Ignited & (~Running | Brake)

Figure 3: Modified representation of mode INACTIVE.

trol lever in the *Activate* position, the system will still remain in mode 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 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 set to *Activate*). These properties can be expressed by the following formula

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

which states that if the cruise control lever is not set to *Activate* in mode INACTIVE, 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*, the engine *Ignited* and *Running* and the *Brake* released. If all of the properties were to be true in the next state, then the transition from mode INACTIVE to mode CRUISE would be triggered and the system would actually be in mode CRUISE in the next state.

The MCB model checker determines that the above formula is true with respect to the system described in Figure 2, but only because our representation of the cruise control system has one CTL state representing the INACTIVE mode state (making it impossible for any two consecutive states to both be annotated with attribute INACTIVE). Our representation only characterizes the conditions that cause mode transitions; it does not characterize the conditions under which a system remains in a mode.

In order to represent the conditions that keep the system in a particular mode, we need to multiply the number of mode states representing that mode. Consider the modified representation of mode INACTIVE shown in Figure 3. The mode is entered via the top INACTIVE mode state. The transition from the upper mode state to the upper trigger state represents the event that activates the transition to mode CRUISE; this representation is attained by annotating the mode state with transition condition $\sim$*Activate*[5] and annotating the trigger state with transition condition

Activate & Ignited & Running & ~Brake

The transition from the upper mode state to the lower mode state, on the other hand, represents the event of *Activate* becoming true but not triggering the transition to mode CRUISE; this representation is attained by annotating the upper mode state with transition condition $\sim$*Activate* and annotating all of the lower mode state's transitions with enabling condition

Activate & (~Running | Brake)

The intended safety property for mode INACTIVE is really an invariant of the mode transition from INACTIVE to CRUISE rather than of the INACTIVE mode itself: the formula states that if event @T(*Activate*) occurs and the WHEN conditions for the transition to mode CRUISE are satisfied, then the system does not remain in mode INACTIVE. To verify this kind of problem, we need to represent the circumstances in which the system *does* remain in the mode, which the above representation of mode INACTIVE does. Even so, the above representation only describes the conditions under which the system remains in mode INACTIVE when condition *Activate* becomes true. Additional mode states would be needed to represent the status of the system when other conditions change value; and in the worse case, we would need 2^n mode states to completely represent the behavior of a system having n input conditions. While such a complete representation of the system would allows us to verify more expressive formulas, we lose the reduced state space that the mode abstraction provided. Thus, there is a trade off between

[5] The mode state is also annotated with transition condition *Ignited* to indicate that the transition to mode OFF has not been activated.

Current Mode	*Inside HysRange*	*WithIn Limits*	*SlfTest Pressed*	*SlfTest Interval*	*Test Interval*	*Reset Interval*	*Shutdown LockTime*	**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

Table 4: Mode Transitions for Water Level Monitoring System.

the size of the system representation and the expressiveness of the formulas that can be checked against the system.

Given our original representation of the cruise control system (which only characterizes the conditions under which the system changes modes), the strongest invariant for the INACTIVE *mode* that can be verified with the MCB model checker is

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

which states that if the system is in mode INACTIVE, then all we know is that the ignition is on.

3.2 Case Study: Water-Level Monitor

The requirements specification of a water-level monitoring system (WLMS) used in this study comes from [26]. The system consists of two mode classes, one that describes system behavior when the system is operating correctly, and one that describes the behavior when the system has failed. The **FAILURE** mode class is simple and uninteresting, and exists independently of the **OPERATING** mode class; therefore, we only studied the **OPERATING** mode class. Table 4 contains the mode transition table. **OPERATING** is comprised of four modes:

OPERATING	The system is running
STANDBY	The system has not been running for at least 200ms, but has not failed
SHUTDOWN	The system stopped running within the last 200ms
TEST	The system is being tested

The system always starts in mode STANDBY.

The propositional variables that appear in the table are defined as follows

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

CTL Machine Construction

Like the cruise control specifications, the WLMS mode transition table contained implicit information that we needed to make explicit before constructing the CTL machine. Additional conditions were added to satisfy the following relationships between system conditions:

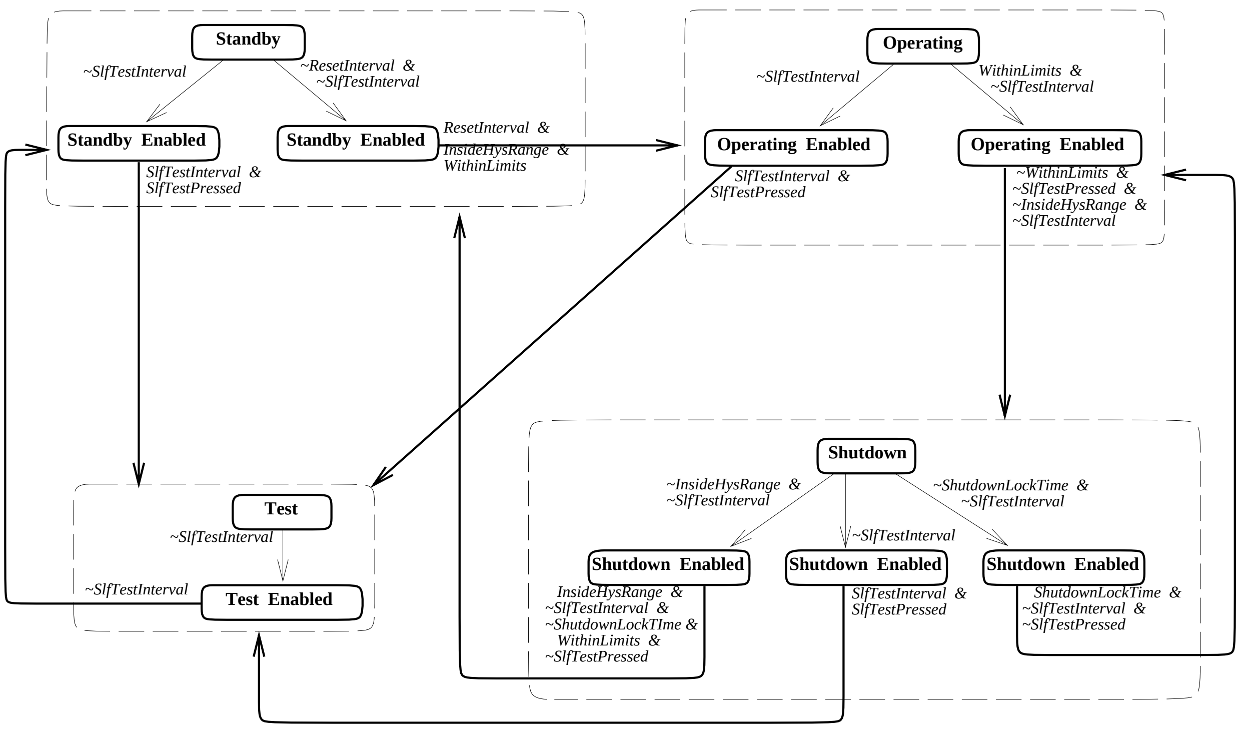
$InsideHysRange \Rightarrow WithinLimits$
$SlfTestInterval \Rightarrow SlfTestPressed$

The added conditions appear in italics in the mode transition table.

The WLMS specification does not allow sequences of simultaneous mode transitions. The model regards the current mode as an additional WHEN condition of any transition condition, and the WHEN conditions of an event must hold for some finite period of time before the event occurs. Therefore, no further adjustments to the requirements specification were needed. The resulting CTL machine is displayed in Figure 4.

Analysis

First, we tested to see if we had entered the specifications correctly, by verifying that the properties associated with each state (as depicted in Figure 4) held whenever the system was in that state. Then we tried to verify some system invariants. Although, the software requirements document for the WLMS system did not include a list of safety properties, we inferred that the following properties should be invariant:


Standby
~SlfTestInterval
~ResetInterval & ~SlfTestInterval
Standby Enabled
Standby Enabled
ResetInterval & InsideHysRange & WithinLimits
SlfTestInterval & SlfTestPressed
Operating
~SlfTestInterval
WithinLimits & ~SlfTestInterval
Operating Enabled
Operating Enabled
SlfTestInterval & SlfTestPressed
~WithinLimits & ~SlfTestPressed & ~InsideHysRange & ~SlfTestInterval
Test
~SlfTestInterval
Test Enabled
~SlfTestInterval
Shutdown
~InsideHysRange & ~SlfTestInterval
~SlfTestInterval
~ShutdownLockTime & ~SlfTestInterval
Shutdown Enabled
Shutdown Enabled
Shutdown Enabled
InsideHysRange & ~SlfTestInterval & ~ShutdownLockTIme & WithinLimits & ~SlfTestPressed
SlfTestInterval & SlfTestPressed
ShutdownLockTime & ~SlfTestInterval & ~SlfTestPressed

$$AG\,((SlfTestInterval \,\&\, {\sim}Test \,\&\, {\sim}TestEnabled) \rightarrow AX(Test))$$
$$AG\,((Operating \rightarrow (WithinLimits \mid SlfTestPressed))$$

The first formula asserts that if the **SlfTst** button has been pressed for 500 ms or more and the system is not currently in mode Test (or its trigger state TestEnabled), then the next state is Test. This formula is false because if the machine is in mode Standby and both of its transitions are simultaneously enabled, the CTL machine may nondeterministically choose the transition to the Operating mode, rather than the one to mode Test. This nondeterminism violates an intended invariant property [27].

The second formula is more critical. It states that if the system is in mode Operating, then either the water level is *WithinLimits* or the **SlfTst** button is being pressed. This property is not invariant. If the system is in mode Operating and the **SlfTst** button is being pressed, then the system will remain in mode Operating if the water level rises above or falls below the limits. But if the **SlfTst** button is released before the transition into mode 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.

These two safety violations only occur under carefully timed conditions. As a result, they went undetected even though the system was implemented.

4 Discussion

In both case studies, we used the model checker to disprove the invariance of required safety properties. We showed that a safety constraint in the cruise control specifications had been stated incorrectly; and in the water-level monitoring system, we found two properties that were intended to be invariant but were not enforced by the requirements.

The remainder of this section addresses issues about our state-based analysis of event-oriented requirements 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.

- We transformed system requirements consisting of one mode class into CTL structures. However, a system's requirements may consist of multiple mode classes, and as such would be modeled by the concurrent execution of several CTL machines.

One must unite these machines into a single 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, but 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, [5] describes a symbolic model checker that will be capable of checking very large CTL structures.

- For this paper, we constructed the CTL machines manually. In the future, we intend to investigate how this task can be automated. It is not clear at this point how much input will be needed from the requirements designer to create machines that accurately model modes with complex transition conditions.

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 [20,22,23]. 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 [7,13,15]. STATEMATE [14], 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, completeness, 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 [18] 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 being enforced in the system's requirements. One can also use the verifier to perform reachability tests but not to verify system properties[6].

Hierarchical multi-state (HMS) machines [11,12] 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 (thereby explicitly representing all execution paths) 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 [2]: 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. Using this technique, we were able to detect errors in both of the requirements documents we studied. We intend to investigate how one might automatically map SCR-style specifications onto CTL machines. We are also investigating how functional and timing requirements can be combined in a uniform model of a system's requirements specification, that can subsequently be analyzed and verified automatically.

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. This research is supported by grants from the Naval Research Laboratory and the Air Force Office of Scientific Research.

References

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

[2] Bledsoe, W. and L. Hines. "Variable elimination and chaining in a resolution-based prover for inequalities." *Proc. 5th Conf. Automated Deduction: Lecture Notes in Computer Science*, W. Bibel and R. Kowalski, Eds., New York, Springer-Verlag, 1980, pp. 70-87.

[6] At present, the Modechart language only consists of modes, transitions between modes, and timing properties of transitions. When state variables are supported, the verifier will be better able to analyze functional behavior.

[3] Browne, M. "Automatic Verification of Finite State Machines Using Temporal Logic." Ph.D. Thesis, Carnegie Mellon University, 1989.

[4] Browne, M., E. Clarke, and D. Dill. "Automatic Verification of Sequential Circuits Using Temporal Logic." *IEEE Trans. on Computers*, Vol. C-35, No. 12 (December 1986), pp. 1035-1044.

[5] Burch, J., E. Clarke, K. McMillan, D. Dill and J. Hwang. "Symbolic Model Checking: 10^{20} States and Beyond.", *Proc. of the 5th Ann. Sym. on Logic in Comp. Sci*, June 1990.

[6] Clarke, E., E. Emerson, and A. Sistla. "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications." *ACM Trans. on Prog. Lang. and Sys.*, Vol. 8, No. 2, April 1986, pp. 244-263

[7] Degl'Innocenti90, M., G. Ferrari, G. Pacini, F. Turini. "RSF: A Formalism for Executable Requirements Specifications." *Trans. on Software Engineering.*, Vol. 16, No. 11 (November 1990), pp. 1235-1246.

[8] Emerson, E. and A. Halpern. " 'Sometimes' and 'Not Never' Revisited: On Branching Time Verses Linear Time Temporal Logic." *Journal of the ACM*, Vol. 33, No. 1 (January 1986), pp. 151-178.

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

[10] Faulk, S. "State Determination in Hard-Embedded Systems." Ph.D. Thesis University of North Carolina, Chapel Hill, North Carolina, June 1989.

[11] Gabrielian, A. and R. Iyer. "Integrating automata and temporal logic: a framework for specification of real-time systems." *Proc. Unified Computation Laboratory*, 1990.

[12] Gabrielian, A. "HMS Machines: A Unified Framework for Specification, Verification and Reasoning for Real-Time Systems." *Proc. Office Naval Research 3rd Annual Workshop on Foundations of Real-Time Computing*, 1990, pp. 341-358

[13] Ghezzi, C. D. Mandrioli, A. Morzenti. "TRIO, a logic language for executable specifications of real-time systems.", *J. of Systems and Software*, Vol. 12, No. 2 (May 1990), pp. 107-123.

[14] Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, M. Trakhtenbrot. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems." *IEEE Trans. on Software Engineering*, Vol. 16, No. 4 (April 1990), pp. 403-414.

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

[16] Heninger, K. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications." *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 1 (January 1980), pp. 2-12.

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

[18] Jahanian F. and A. Mok. "Safety Analysis of Timing Properties in Real-Time Systems." *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 9 (September 1986), pp. 890-904.

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

[20] Manna, Z. and Pnueli, A. "Tools and Rules for the Practicing Verifier." Tech. Rep STAN-CS-90-1321, Stanford University, Stanford CA, July 1990.

[21] Parnas, D. and J. Madey. "Functional Documentation for Computer Systems Engineering." Tech. Rep. TR-90-287, Queen's University, Kingston, Ontario, September 1990.

[22] Ostroff, J. and W. Wonham. "Modeling, Specifying, and Verifying Real-time Embedded Computer Systems." *Proc. IEEE Real-Time Systems Sym.*, 1987, pp. 124-132.

[23] Pnueli, A. "The Temporal Logic of Programs." *Proc. of 18th Annual Sym. on the Foundation of Comp. Sci.*, 1977, pp. 46-57.

[24] Sistla, A., and E. Clarke. "The Complexity of Propositional Linear Temporal Logics." *Journal of ACM*, Vol. 32, No. 3 (July 1985), pp. 733-749.

[25] Smith, S. and S. Gerhart. "STATEMATE and Cruise Control: A Case^2 Study." *Proc. COMPAC '88, 12th Int. IEEE Computer Software and Application Conference.*, 1988, pp.49-56

[26] van Schouwen, J. "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems." Tech. Rep. 90-276, Queen's University, Kingston, Ontario, May 1990.

[27] van Schouwen, J. Private communications.