Using Model Checking to Analyze Requirements and Designs

Joanne Atlee* Marsha Chechik John Gannon†

Abstract

Precise notations have been developed to specify unambiguous requirements, and ensure that all cases of appropriate system behavior are considered and documented. Using one such notation, we have developed techniques to automatically analyze software artifacts at early stages of the software development life cycle. We use model checking as our verification technique because it can be fully automated and can check properties of large systems. This paper describes model checking and summarizes our efforts to use it to analyze software requirements and designs. We prove that requirements model system safety properties and that designs model consistency properties derived from requirements by creating abstractions of these software artifacts and using model checking to determine if the abstractions are models of the properties. We present results from a case study in which we analyzed the requirements and design of a small but realistic system.

1 Introduction

Precise documentation of software requirements has several potential benefits[27]: designers know what they are to build; reviewers can check that customers' intentions are met; testers can formulate test cases independently from the system's implementation; and maintainers can use the original requirements to learn about the system before making their changes. Several formal requirements notations (e.g., the Software Cost Reduction (SCR) notation[2, 20, 22], the Requirements State Machine Language[24], and Statecharts[18]) have been used to specify the requirements of large, real-world avionics applications (the A-7E aircraft, the FAA's Traffic Collision and Avoidance System, and the Lavi fighter's avionics system, respectively). These requirements notations describe systems as sets of concurrently executing state machines which respond to events in their environments.

The keys to winning acceptance for employing precise documentation during system development include demonstrating that its use improves software quality, amortizing the

*Atlee is at Department of Computer Science, University of Waterloo, Waterloo, Ontario, jmatlee@watdragon.uwaterloo.ca.

†Chechik and Gannon are at Department of Computer Science, University of Maryland, College Park, Maryland, {chechik,gannon}@cs.umd.edu.

cost of its creation across several different analysis activities, and reducing the cost of analysis through automation. Our research has focused on developing techniques that use formal methods to enable automatic analysis of program artifacts at early stages of the software development life cycle[4, 6, 28, 5, 10, 11].

In this paper, we summarize our work to analyze program requirements and designs. We use model checking[14] because it can be fully automated and can check properties of large systems. Developers are more likely to understand a proof technique like model checking, which is based on search and which produces counter examples when proofs fail, than a technique based inductive theorem proving. Model checking has been successfully applied to verifying and debugging hardware designs (e.g., [8, 7, 13]). More recently it has been used to analyze software artifacts. Model checking has been used to detect design flaws in software architecture designs[1] and Z specifications[23], and to prove properties of cache coherence protocols[30] and concurrent Ada programs[9]. The key to success in these endeavors is creating an appropriate abstraction of a system so that results obtained from analyzing the abstraction also apply to the system.

We analyze safety properties of software requirements and designs by creating models from such artifacts and using model checking techniques to determine if the safety properties are true for the model. We have developed automated techniques to translate SCR requirements into a logical model. We represent a system's safety assertions as logical formulas in a branching-time temporal logic, Computation Tree Logic[15] (CTL), and use existing CTL model checkers[8, 25] to check our models.

Analyzing the safety properties of system's requirements, however, fails to tell us whether or not its implementation preserves these properties. To verify that a system design is consistent with its requirements, we would like to ensure that the design's state transitions are enabled by the same events as those of the requirements, and the requirement's safety properties also hold in the design. To judge global properties like these, we need to determine the possible system states which exist at different program points. The detailed bookkeeping necessary to do this exceeds the capabilities of human reviewers for all but small implementations. We present a language for specifying detailed designs and an analysis technique to create a model of a design through abstract interpretation of the language constructs. We also show how to use requirements information to automatically generate properties which ensure that required state transitions appear in the design and systems goals hold, and how these properties are checked against the design model.

The rest of the paper is organized as follows: Section 2 introduces the requirements specification format. Section 3 explains basic principles behind model checking. In Section 4 we show how to create a logic-model semantics that precisely models the operational semantics of SCR modes and mode transitions, so that system goals can be translated into temporal logic formulas and model checking techniques used to verify that these formulas hold in the requirements. Section 5 presents our techniques for verifying designs: how to generate temporal logic properties from SCR tables, how build finite-state abstractions of designs, and how to model check the properties using a special-purpose model checker.

Section 6 describes a case study in which we analyzed the requirements and design of a Water-Level Monitoring System[29]. We present our conclusions in Section 7.

2 SCR Requirements

The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of the Software Cost Reduction project [2, 22]. A complete SCR requirements specification contains behavioral, functional, precision, and timing requirements of a software system as well as assumptions about the environment in which the system will operate. In this section, we describe SCR behavioral requirements and environmental assumptions, and the system goals they are meant to ensure.

2.1 Behavioral Requirements

An SCR requirements specification models a system as a set of event-driven, state-transition machines. The machines' environment is abstracted as a set of *monitored state variables* and *controlled state variables* [26, 29]. Changes to monitored variables may cause the system to change its mode or to alter the values of its controlled variables.

The input language of each machine is a set of *conditioned events*. A *condition* is a predicate on monitored or mode class variables, an *event* when the value of a condition changes. Let condition SwitchOn represent predicate [On/Off switch = On], and condition PumpFail represent predicate [Pump failure = true]. Primitive events @T(SwitchOn) and @F(SwitchOn) represent condition SwitchOn *becoming true* and *becoming false*, respectively. Conditioned event

@T(SwitchOn) WHEN [$\sim$ PumpFail]

describes the event SwitchOn *becomes true* while PumpFail *remains false*. Formally, conditioned event @T(SwitchOn) WHEN [PumpFail] occurs at time t if and only if primitive event @T(SwitchOn) occurs at time t and condition PumpFail is true for some non-zero interval of time leading up to and including time t [2]. SwitchOn is called the *triggering event* and PumpFail is called the event's WHEN *condition*.

A state of the monitored environment is defined by the current values of the conditions, and the state space is the set of possible combinations of values of conditions. However, the behavior of the system is rarely affected by the values of all the conditions at once. A *mode class* defines a set of states, called *modes*, that partition the monitored environment's state space. One mode is designated as the *initial mode*. Assumptions about the initial state of the environment are specified with the initial mode. Transitions between pairs of modes are activated by conditioned events. If a conditioned event can trigger two or more transitions from the same mode, then the mode class is non-deterministic.

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

Table 1 shows a mode transition table for a Simplified Water-Level Monitoring System (SWLMS). A switch controls whether the system is on or off. If the system is on and its sensors detect too much (too little) water, a pump is turned on for a fixed period to remove (add) some water. If the sensor or the pump fails, the system enters an error state. This simplified version of the system has no error recovery, so there are no transitions from the error state. This system has one mode class MC with modes Off, Operating, and Error; four monitored variables SwitchOn, PumpFail, TooHigh, and TooLow; and a single controlled variable PumpOn. Below the mode transition table is the specification of the system's initial mode. Mode class MC starts in mode Off, and all monitored variables are initially false. The specification of a mode class's transition relation has a tabular format. Each row in the table specifies a conditioned event that activates a transition from the mode on the left to the mode on the right. For example, a table entry of "@T" (or "@F") under a column represents a triggering event for the condition represented by the variable labelling the column, a table entry of "t" (or "f") represents a WHEN condition for the condition. If the value of a condition does not affect the occurrence of a conditioned event, then the table entry is marked with a hyphen ("–"). If during time interval $[t - \epsilon, t)$ the system is in mode Off, the switch is in the Off position and the pump is operating; and if at time t the switch is moved to the On position while the pump continues to operate; then the system is in mode Operating at time t.

Current Mode	SwitchOn	PumpFail	**New Mode**
Off	@T	f	Operating
	–	@T	Error
Operating	@F	f	Off
	–	@T	Error

Initial: Off (~SwitchOn & ~PumpFail & ~TooHigh & ~TooLow)
Assumptions: TooLow– >>~ TooHigh

Table 1: Mode transition table for SWLMS.

Values of controlled variables change in response to events when the system is in particular modes. Table 2 shows an event table for the controlled variable PumpOn, which represents the pump being turned on or off. This variable starts with value false and becomes true when the system is in mode Operating and either event @T(TooHigh) or @T(TooLow) occurs.

Mode	Triggering Event	
Operating	@T(TooHigh)	–
	@T(TooLow)	–
	–	@F(TooHigh)
	–	@F(TooLow)
	–	@T(PumpFail)
Off	–	@T(PumpFail)
PumpOn =	True	False

Initial: False

Table 2: Event table for controlled variable PumpOn.

2.2 Environmental Assumptions

An SCR requirements document also specifies assumptions of the behavior of the environment. Similar to the *NAT* relation in Parnas's 4-variable model of system requirements [26], an *assumption* specifies constraints on the values of conditions, imposed either by laws of nature or by other mode classes in the system. As such, assumptions are invariant constraints that must hold in all system states.

The syntax and semantics of assumption specifications are described in [3]; for the purposes of this paper, symbol “|” denotes exclusive-or, “$- >$” denotes implication, “$- >>$” denotes strict implication, and “$<$” denotes an ordering on the lengths of time that conditions are true[1]. The assumptions specified in Figure 1 state that the water level cannot be too high and too low at the same time.

2.3 System Goals

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

- *If the system is in mode* Off*, then conditions* SwitchOn *and* PumpFail *are false.*

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

- *If event @T(*SwitchOn*) occurs, the system cannot remain in mode* Off*.*

[1] In $a - >> b$ the state space in which a is true is a strict subset of the state space in which b is true. As with implication, whenever a is true, b must also be true. Because of the relationship between their state spaces, b must be true whenever a is changing value, and a must be false whenever b changes value.

Temporal requirements on state changes can also be specified.

- *When the water level becomes too high while the system is in mode* Operating*, the pump will immediately be turned on.*

3 Model Checking

Temporal logic permits us to make statements about changes in time, e.g., that a formula may be true at some point in the future. Computational tree logic (CTL) is a propositional branching time logic, whose operators permit explicit quantification over all possible futures[15]. The syntax for CTL formulas is summarized below:

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

Note that temporal operators occur only in pairs in which a quantifier A (always) or E (exists) is followed by F (future), G (global), U (until), or X (next).

The value of a formula is defined with respect to a model $M = (V, S, s_0, R, I)$ where V is a set of propositional variables, S is a set of states, $s_0 \in S$ is the start state, R is a transition relation, and I is a set of interpretations specifying which propositions are true in each state. Temporal logic formulas are evaluated with respect to a state in the model. For example, formula EXf (AXf) is true in state s_i if formula f is true in some (every) successor state of s_i. Formula $E[f\ U\ g]$ $(A[f\ U\ g])$ is true in state s_i if along some (every) path emanating from s_i there exists a future state s_j at which g holds and f is true until state s_j is reached. EFf (AFf) is true in state s_i if along some (every) path from s_i there exists a future state in which f holds. Finally, EGf (AGf) is true in state s_i if f holds in every state along some (every) path emanating from s_i.

We capture these ideas more formally with the following definitions. If formula f is true in state s of model M, we write $M, s \models f$. A formula f is true for the model, if it is true in the model's start state, i.e., $M, s_0 \models f$. When we are concerned with a single model, we abbreviate $M, s \models f$ as $s \models f$. Let $p \in V$ be a proposition, $s \in S$ be a state, and f be a formula.

$$
\begin{array}{rll}
s \models p & \text{iff} & (I(s))(p) \\
s \models \sim f & \text{iff} & s \not\models f \\
s \models f \vee g & \text{iff} & s \models f \text{ or } s \models g \\
s_0 \models EX\ f & \text{iff} & \text{for some path } (s_0, s_1, \ldots)), s_1 \models f \\
s_0 \models AX\ f & \text{iff} & \text{for all paths } (s_0, s_1, \ldots)), s_1 \models f \\
s_0 \models E(fUg) & \text{iff} & \text{for some path } (s_0, s_1, \ldots)), \text{ for some } i\ s_i \models g \text{ and for all j < i } s_j \models f \\
s_0 \models A(fUg) & \text{iff} & \text{for all paths } (s_0, s_1, \ldots)), \text{ for some } i\ s_i \models g \text{ and for all j < i } s_j \models f
\end{array}
$$

Several abbreviations will also be used to write formulas. These include abbreviations for propositional formulas, abbreviations for commonly used *until* operations (e.g., reachability and invariance), and universally quantified temporal formulas (defined in terms of existentially quantified temporal formulas).

$$\begin{array}{lcl}
(f \wedge g) & \equiv & \sim(\sim f \vee \sim g) \\
(f \rightarrow g) & \equiv & (\sim f \vee g) \\
AXf & \equiv & \sim EX(\sim f) \\
EFf & \equiv & E[\text{true}\ U\ f] \\
AFf & \equiv & A[\text{true}\ U\ f] \\
EGf & \equiv & \sim AF(\sim f) \\
AGf & \equiv & \sim EF(\sim f)
\end{array}$$

The abbreviation for invariance, AG, is used often in this paper. $AG(f)$ is true in state s_i if for all paths $(s_i, s_{i+1}, \ldots)$, formula f holds in all states.

Model checking determines the value of a formula f for a particular model by computing the set of states in which the formula is true, i.e., $\{s \mid s \models f\}$. For example formula EFf represents the set of states from which a state satisfying f can be reached in *some* number of state transitions.

$$f \quad \vee \quad EXf \quad \vee \quad EX(EXf) \quad \vee \ldots$$

where f is the set of states from which a f–state can be reached in zero state transitions, EXf is the set of states from which a f–state can be reached in one state transition, $EX(EXf)$ is the set of states from which a f–state can be reached in two state transitions, etc. This set of states can be computed using the following least fixpoint definition:

$$EFf = \mu.Y(f \quad \vee \quad EXY))$$

starting with false as the value for Y.

By way of example, consider checking $s_0 \models EF(\sim a \wedge \sim b)$ for model M in Figure 1.1. The first iteration computes the set of states in which the formula $(\sim a \wedge \sim b)$ holds, $\{s_3\}$. This set, which is shaded in Figure 1.2, corresponds to computing $(\sim a \wedge \sim b) \vee EX false$. During the second iteration, s_2 is added to the set because the formula is true in its successor, s_3. At this point, we have computed the set of states satisfying $(\sim a \wedge \sim b) \wedge EX(\sim a \wedge \sim b)$. The third iteration adds s_0 and s_1 to the set since each state has a successor which is already a member of the set. Thus far we have computed $(\sim a \wedge \sim b) \wedge EX((\sim a \wedge \sim b) \wedge EX(\sim a \wedge \sim b))$. Since no new states are added during the fourth iteration, we reach a fixed point and the algorithm terminates.

Automating model checking is quite easy, except that the entire state space of the model must be constructed before the fixed-point algorithms can be applied. Model checking can be performed symbolically by manipulating quantified boolean formulas without constructing a model's state space[25]. To perform symbolic model checking, sets of states and transition relations are represented by formulas, and set operations are defined in terms of formula manipulations.

As before, a system state represents an interpretation (i.e., truth assignment) for a set of propositional variables V. Formulas represent sets of states that satisfy the formula. For example, in Figure 1.1 the formula a represents the set of states in which a is true ($\{s_0, s_1\}$), the formula b represents the set of states in which b is true ($\{s_1, s_2\}$), and the formula $a \vee b$ represents the union of these sets ($\{s_0, s_1, s_2\}$). State transitions are described by relations over two sets of

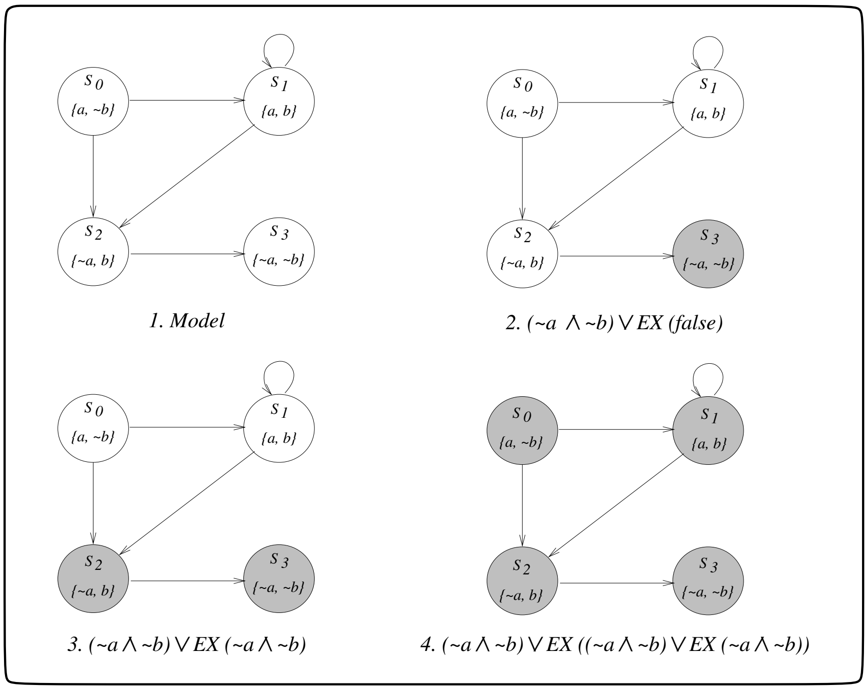

Figure 1: Model checking $s_0 \models EF(\sim a \wedge \sim b)$

variables, V and V'. The variables in V represent values in a transition's source state, while those in V' represent values in the destination state. For example, the transition from s_2 to s_3 in Figure 1.1 is described by the formula $(\sim a \wedge b \wedge \sim a' \wedge \sim b')$. This describes a transition from a state containing $\{\sim a, b\}$ to another state containing $\{\sim a, \sim b\}$. The formula $(a \wedge b')$ describes the transitions from s_0 to s_1 and s_2, and from s_1 to s_2 and to itself. The formula describing the transition relation R for Figure 1.1 is $(a \wedge b') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')$.

The meaning for CTL formulas can be redefined in terms of sets of states.

$$\begin{array}{rll}
s \models p & \text{iff} & s \in p \text{ where } p \in V \\
s \models \sim f & \text{iff} & s \in \neg p \\
s \models f \vee g & \text{iff} & s \in (f \vee g) \\
s \models EXf & \text{iff} & s \in (\exists V'(R \wedge f(V/V'))) \\
s \models AXf & \text{iff} & s \in \sim (\exists V'(R \wedge \sim f(V/V'))) \\
s \models E(fUg) & \text{iff} & s \in \mu Y(g \vee (f \wedge EX\ Y)) \\
s \models A(fUg) & \text{iff} & s \in \mu Y(g \vee (f \wedge AXY))
\end{array}$$

A CTL formula f is evaluated for a model by deriving a *propositional logic expression* that describes the set of states satisfying the CTL formula for the model. The model checker verifies that the interpretation of the model's initial state s_0 satisfies the expression.

For example, we check $s_0 \models EX(\sim a \wedge \sim b)$ by computing a propositional logic formula representing the states which have successors where $(\sim a \wedge \sim b)$ is true. For this, we use the formula $\exists a', b'(R \wedge (f(a, b/a', b')))$, where R represents the transition relation, f is the formula being checked $(\sim a \wedge \sim b)$, and $(f(a, b/a', b')$ represents substitutions of a' and b' for corresponding free occurrences of a and b in f (effectively changing f to be a formula about next states' truth assignments, $\sim a' \wedge \sim b'$). In doing the evaluation, we replace formulas of the form $\exists v(f)$ by $(f(v/true) \vee f(v/false))$.

$$
\begin{aligned}
&EX(\sim a \wedge \sim b) = \\
&\exists a', b'(((a \wedge b') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')) \wedge ((\sim a \wedge \sim b)(a, b/a', b'))) = \\
&\exists a', b'(((a \wedge b') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')) \wedge (\sim a' \wedge \sim b')) = \\
&\exists a', b'((a \wedge b' \wedge \sim a' \wedge \sim b') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b' \wedge \sim a' \wedge \sim b')) = \\
&\exists a', b'((false) \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')) = \\
&\exists a', b'(\sim a \wedge b \wedge \sim a' \wedge \sim b') = \\
&\exists a'((\sim a \wedge b \wedge \sim a' \wedge \sim true) \vee (\sim a \wedge b \wedge \sim a' \wedge \sim false)) = \\
&\exists a'(\sim a \wedge b \wedge \sim a') = (\sim a \wedge b \wedge \sim true) \vee (\sim a \wedge b \wedge \sim false) = \sim a \wedge b
\end{aligned}
$$

The computed propositional logic formula represents the set of states whose interpretations satisfy $EX(\sim a \wedge \sim b)$ is true, i.e., $\{s_2\}$. Theorem $s_0 \models EX(\sim a \wedge \sim b)$ is true if the values of a and b in s_0 (a=true and b=false) satisfy $\sim a \wedge b$, but this is not the case.

In the following example, we calculate $EF(\sim a \wedge \sim b)$ for the model in Figure 1.1.

1. $(\sim a \wedge \sim b) \vee EX false = (\sim a \wedge \sim b)$
2. $(\sim a \wedge \sim b) \vee EX(\sim a \wedge \sim b) = (\sim a \wedge \sim b) \vee (\sim a \wedge b) = \sim a$
3. $$
\begin{aligned}
&(\sim a \wedge \sim b) \vee EX(\sim a) = \\
&(\sim a \wedge \sim b) \vee \exists a', b'(((a \wedge b') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')) \wedge ((\sim a)(a/a'))) = \\
&(\sim a \wedge \sim b) \vee \exists a', b'(((a \wedge b') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')) \wedge \sim a') = \\
&(\sim a \wedge \sim b) \vee \exists a', b'((a \wedge b' \wedge \sim a') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim b')) = \\
&(\sim a \wedge \sim b) \vee \exists a'(((a \wedge true \wedge \sim a') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim true)) \\
&\qquad\qquad \vee((a \wedge false \wedge \sim a') \vee (\sim a \wedge b \wedge \sim a' \wedge \sim false))) = \\
&(\sim a \wedge \sim b) \vee \exists a'((a \wedge \sim a') \vee (\sim a \wedge b \wedge \sim a')) = \\
&(\sim a \wedge \sim b) \vee ((a \wedge \sim true) \vee (\sim a \wedge b \wedge \sim true)) \vee ((a \wedge \sim false) \vee (\sim a \wedge b \wedge \sim false)) = \\
&(\sim a \wedge \sim b) \vee ((false) \vee (false)) \vee ((a) \vee (\sim a \wedge b)) = \\
&(\sim a \wedge \sim b) \vee (\sim a \wedge b) \vee a = true
\end{aligned}
$$
4. $(\sim a \wedge \sim b) \vee EX(true) = true$

Note that the formulas at the end of each step correspond to the set of states that are shaded after that step. The first formula $(\sim a \wedge \sim b)$ corresponds to $\{s_3\}$, the second $(\sim a)$ to $\{s_2, s_3\}$, and true to the set of all states.

4 Model-Checking Requirements

In this section, we define a logic-model semantics for SCR specifications that precisely models the operational semantics of SCR modes and mode transitions. System goals are translated into temporal logic formulas, and model checking is used to analyze if the requirements are a model of each of the formulas.

4.1 SCR Logic Model of Mode Transitions

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

Formally, a logic model of an SCR mode class is a tuple $\langle V, S, s_0, R, I \rangle$, where

- V is the set of modes and conditions.
- S is the set of possible states.
- $s_0 \subset S$ is the set of possible initial states
- $R \subset (S \times S)$ is a binary transition relation on S.
- I is an interpretation function. $I(s)$ assigns a truth value to each mode and condition in state s.

A conditioned event is modeled by any pair of states s_i and s_j related by R (i.e., $(s_i, s_j) \in R$), in which the event's triggering conditions are unsatisfied in state s_i, the triggering conditions are satisfied in state s_j, and the event's WHEN conditions are satisfied in both states. If the conditioned event triggers a mode transition, then the pair of states must also model the mode change: in state s_i, the source mode must be true and the destination mode false; and in state s_j, the source mode must be false and the destination mode true.

Our logic-model representation of an SCR requirements specification deviates from its operational semantics in two respects. First, SCR operational semantics forbids the simultaneous occurrence of two or more primitive events, because its operational model assumes that the implemented system will react to events one at a time, regardless of when the events occur. Traditionally, implementations of reactive systems queue primitive events as their occurrences are detected; when a primitive event reaches the head of the queue, the system checks the values of other conditions to determine if a significant conditioned event (e.g., one that might activate a mode transition) has occurred [19].

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

The second difference between operational and logic-model semantics of SCR specifications pertains to the value of WHEN conditions at the time of a conditioned event. In an SCR logic-model, WHEN conditions must be satisfied both immediately before and during the occurrence of a conditioned event. According to the latest operational semantics of SCR, a WHEN condition must be satisfied immediately before the occurrence of a conditioned event, but its value at the time of the event is unknown [21]. Given the above restriction on the occurrence of simultaneous events, one can to infer the value of most WHEN conditions: WHEN conditions that are unrelated to the triggering event have the same value during the event as they had immediately before the event. However, WHEN conditions that are related to the triggering event may or may not be changing value along with the triggering event.

We chose to model the older conditioned-event semantics [2] because we found it easier to write a correct SCR specification given these semantics. If one wants to write a specification that guarantees a particular formula f is always true in a particular mode M, one needs to specify that f is always true upon entry into M and that the system always exits mode M if f becomes false. Such a specification is difficult (if not impossible) to write if one cannot infer from a WHEN condition WHEN[f] on a transition into the mode that f is true when the mode is entered.

4.2 Behavioral Requirements

We express the logic model of SCR mode transition requirements as a temporal logic formula. A logic-model state is expressed as a conjunction of the conditions and modes interpreted to be true in that state. The transition relation R is expressed as a formula over conditions and modes in both the current and the next state.

Consider the SCR mode transition table presented in Table 1. The following formula specifies the logic formula that holds in the initial state s_0. Since only mode transitions are being modelled, we have simplified the initial state to include only those monitored variables which affect mode transitions.

$s_0 \rightarrow (\text{Off} \wedge \sim \text{SwitchOn} \wedge \sim \text{PumpFail})$

Each row in the mode transition table is expressed as a conjunction of the current mode, the conditioned event, and the next mode. For example, the first row in the mode transition table is represented by the following formula.

$\text{Off} \wedge \sim\!\text{SwitchOn} \wedge \sim\!\text{PumpFail} \wedge \text{SwitchOn}' \wedge \sim\!\text{PumpFail}' \wedge \text{Operating}'$,

where a unprimed and the primed versions of a variable indicate the value of this variable in the current and the next states, respectively. Each row in a mode transition table specifies an element in the logic model's transition relation. The set of mode transitions is expressed as a disjunction of the transitions' formulas. Finally, the table also states implicitly that if a conditioned event occurs that does *not* trigger any of the transitions leaving the current mode, then the current mode remains the same. To capture this latter behavior, we add a new transition for each mode. The new transition states that the mode is true in the current state, all of the conditioned events triggering transitions leaving the mode are false in the next state, and the mode remains true in the next state. The resultant transition relation is a tautology; thus, an SCR logic model has a total transition relation. Figure 2 shows a partial logic model for the SCR mode transitions of the SWLMS specified in Table 1.

If an SCR specification consists of several mode classes, then the logic model of the specification is a conjunction of the logic models of the mode classes.

$$
\begin{array}{l}
s_0 \;\rightarrow\; (\mathrm{Off} \wedge \sim \mathrm{SwitchOn} \wedge \sim \mathrm{PumpFail}) \\
\bigwedge \\
(\quad (\mathrm{Off} \wedge \sim \mathrm{SwitchOn} \wedge \sim \mathrm{PumpFail} \wedge \mathrm{SwitchOn}' \wedge \sim \mathrm{PumpFail}' \wedge \mathrm{Operating}') \bigvee \\
\quad (\mathrm{Off} \wedge \sim \mathrm{PumpFail} \wedge \mathrm{PumpFail}' \wedge \mathrm{Error}') \bigvee \\
\quad (\mathrm{Off} \wedge \sim (\mathrm{SwitchOn} \wedge \sim \mathrm{PumpFail} \wedge \mathrm{SwitchOn}' \wedge \sim \mathrm{PumpFail}') \wedge \\
\qquad\qquad \sim (\sim \mathrm{PumpFail} \wedge \mathrm{PumpFail}') \wedge \mathrm{Off}') \bigvee \\
\quad (\mathrm{Operating} \wedge \mathrm{SwitchOn} \wedge \sim \mathrm{PumpFail} \wedge \sim \mathrm{SwitchOn}' \wedge \sim \mathrm{PumpFail}' \wedge \mathrm{Off}') \bigvee \\
\quad (\mathrm{Operating} \wedge \sim \mathrm{PumpFail} \wedge \mathrm{PumpFail}' \wedge \mathrm{Error}') \bigvee \\
\quad (\mathrm{Operating} \wedge \sim (\mathrm{SwitchOn} \wedge \sim \mathrm{PumpFail} \wedge \sim \mathrm{SwitchOn}' \wedge \sim \mathrm{PumpFail}') \wedge \\
\qquad\qquad \sim (\sim \mathrm{PumpFail} \wedge \mathrm{PumpFail}') \wedge \mathrm{Operating}') \bigvee \\
\quad (\mathrm{Error} \wedge \mathrm{Error}') \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad)
\end{array}
$$

Figure 2: SCR logic model of SWLMS mode transition table.

4.3 Environmental Assumptions

The formula in Figure 2 represents the *unconstrained* transition relation of the SWLMS. For example, the formula expresses no constraints on the number of modes that can be true in any logic-model state. Furthermore, the SCR specification's environmental assumptions are not represented in the logic model.

Environmental assumptions can be represented as logic formulas.

$$
\begin{array}{ll}
\bigwedge & (\mathrm{Off} \vee \mathrm{Operating} \vee \mathrm{Error}) \bigwedge (\mathrm{Off} \rightarrow (\sim \mathrm{Operating} \wedge \sim \mathrm{Error})) \bigwedge \\
& (\mathrm{Operating} \rightarrow (\sim \mathrm{Off} \wedge \sim \mathrm{Error})) \bigwedge (\mathrm{Error} \rightarrow (\sim \mathrm{Off} \wedge \sim \mathrm{Operating})) \bigwedge \\
& (\mathrm{TooLow} \rightarrow \sim \mathrm{TooHigh}))
\end{array}
$$

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

4.4 System Goals and Model Checking

Most of the SWLMS system's goals can be easily expressed as CTL formulas and proved using the SMV model checker[25]. The following formulas state properties that hold invariantly when the system is in a particular mode. For example, if the system is in mode Operating, it is invariantly true that the switch is on and the pump is working.

$AG(\text{Off} \rightarrow (\sim\text{SwitchOn} \wedge \sim\text{PumpFail}))$

$AG(\text{Operating} \rightarrow (\text{SwitchOn} \wedge \sim\text{PumpFail}))$

$AG(\text{Error} \rightarrow \text{PumpFail})$

The last formula is false, and the SMV checker produces a counter example in which the system remains in mode Error while PumpFail changes value from true to false. We could rewrite the last formula as $AG(\text{PumpFail} \rightarrow \text{Error})$, which can be verified.

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

*If event @T(*SwitchOn*) occurs, the system cannot remain in mode* Off.

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

$AG(\sim\text{SwitchOn} \rightarrow \sim EX(\text{SwitchOn} \wedge \text{Off}))$

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

To ease the phrasing of CTL formulas that refer to the occurrence of conditioned events, we use unary logic connectives @T and @F to express propositional formulas that are *becoming true* and *becoming false*, respectively. We could simulate the new connectives using their definitions: evaluation of their operands in the current and next states. SMV, however, can only check formula with respect to the current values of variables. Therefore, the SMV behavioral requirements must evaluate events using the current and next values of the their operands. Let f be an arbitrary propositional CTL formula (i.e., a CTL formula with no modal operators). The connectives have the following definitions.

$@\text{T}(f) \quad \text{iff} \quad \sim f \wedge f'$

$@\text{F}(f) \quad \text{iff} \quad f \wedge \sim f'$

The above invariant is more simply expressed when formulated using the new connectives.

$AG((@T(\text{SwitchOn}) \rightarrow \sim EX(\text{Off})))$

5 Model Checking Designs

Once we verify that system goals hold in a set of requirements, we are interested in showing that these and other properties are preserved in a design (and further in an implementation) of the system. The rest of this section presents a technique for discovering instances of inconsistency and incompleteness in detailed designs of programs with respect to their requirements.

5.1 Consistency

We say that a detailed design is consistent with its requirements if it

1) implements all state transitions specified in the requirements,
2) does not implement any state transitions which are not specified in the requirements, and
3) satisfies all assumptions and system goals.

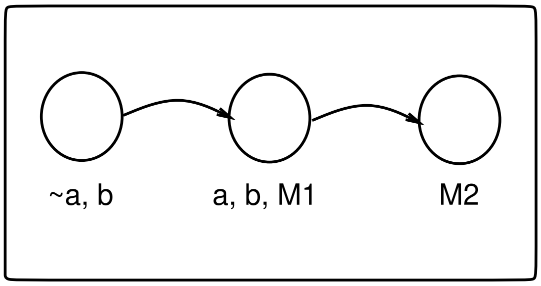

Figure 3: Behavioral model of a mode transition.

A requirements specification can be translated into a list of properties which capture this notion of consistency. To prove that a design is consistent with its requirements, we demonstrate that the design is a model of *all* these properties. In order to facilitate that, we have to make some changes to the original SCR semantics. First, no program can implement mode transitions which occur instantaneously with event detections. Instead, event detections and mode transitions usually occur in different statements. We modify the SCR semantics discussed in Section 2 accordingly. For example, a behavioral model of a transition between modes M1 and M2 triggered by a conditioned event @T(a) WHEN [b], is as shown in Figure 3. We capture the meaning of this transition in a formula $\sim a \wedge a\prime \wedge b \wedge b\prime \wedge M1\prime \wedge M2\prime\prime$, where the unprimed version of condition a denotes a in the current state, the primed version denotes a in the next state, and the double-primed version denotes a in the next next state. Second, the notion of "next state" used in the requirements indicates either *all* or *some* next states. Since we are interested in generating properties which can be captured in a real program, we want to make a clear distinction between *all* and *some* next states by using CTL operators EX and AX.

5.1.1 Automatically-generated safety properties

Properties that a design of SWLMS would have to satisfy appear in Figures 4 and 5. Property P_0 is generated from environmental assumptions of the system and is used to check that these assumptions are preserved. We call this property "constraints are met" (CAM). Properties P_1 - P_3 ensure that the only transitions to a mode are those specified in the requirements. Such properties are called "only legal transitions" (OLT). There is one property of this type generated for every requirements mode (of every mode class). Property P_2 asserts that if the system will be in mode Operating in its next state, then it either must already be in that mode or the transition from Off on @F(SwitchOn) WHEN [~PumpFail] is occuring. P_2 is obtained by composing the rows in the mode transition table of mode class MC which have Operating in their right columns. P_1 slightly differs from P_2 since it captures the fact that MC starts in mode Off. Properties P_4

$$
\begin{aligned}
P_0 &= AG((\text{Off} \vee \text{Operating} \vee \text{Error}) \ \wedge \ (\text{Off} \rightarrow (\sim \text{Operating} \wedge \ \sim \text{Error})) \wedge \\
&\quad (\text{Operating} \rightarrow (\sim \text{Off} \wedge \ \sim \text{Error})) \ \wedge \ (\text{Error} \rightarrow (\sim \text{Operating} \wedge \ \sim \text{Off})) \wedge \\
&\quad (\text{TooLow} \rightarrow \sim \text{TooHigh})) \\
P_1 &= AG(\text{Off} \vee (EX(\text{Off}) \rightarrow (\text{Off} \vee \\
&\quad (\text{Operating} \ \& \ @F(\text{SwitchOn}) \ \& \ \sim\text{PumpFail})))) \\
P_2 &= AG(EX(\text{Operating}) \rightarrow (\text{Operating} \vee \\
&\quad (\text{Off} \ \& \ @T(\text{SwitchOn}) \ \& \ \sim\text{PumpFail}))) \\
P_3 &= AG(EX(\text{Error}) \rightarrow (\text{Error} \vee \\
&\quad (\text{Operating} \ \& \ @T(\text{PumpFail})) \ \vee \ (\text{Off} \ \& \ @T(\text{PumpFail})))) \\
P_4 &= EF(\text{Off} \ \& \ @T(\text{SwitchOn}) \ \& \ \sim\text{PumpFail} \ \& \ EX(\text{Operating})) \\
P_5 &= EF(\text{Off} \ \& \ @T(\text{PumpFail}) \ \& \ EX(\text{Error})) \\
P_6 &= EF(\text{Operating} \ \& \ @F(\text{SwitchOn}) \ \& \ \sim\text{PumpFail} \ \& \ EX(\text{Off})) \\
P_7 &= EF(\text{Operating} \ \& \ @T(\text{PumpFail}) \ \& \ EX(\text{Error}))
\end{aligned}
$$

Figure 4: Assumptions and properties for mode class.

- P_7 ensure that all transitions specified in the requirements appear in designs and are called "all legal transitions" (ALT) properties. P_4 and P_5 correspond to the two transitions leaving mode Off and are obtained by composing the rows of the MC mode transition table which have Off in their left columns. There is one ALT property for every row of every mode transition table specified in the requirements. Properties generated for the controlled variable PumpOn are shown in Figure 5. P_8 and P_9 are OLT properties which ensure that PumpOn changes value only as a result of specified events, and P_{10} - P_{15} are ALT properties which ensure that all changes to PumpOn specified in the requirements appear in the design. There are two OLT properties generated for each controlled variable, reflecting changes of value to true and false, respectively, and one ALT property generated for each row of every table.

5.1.2 System Goals

We use a shorthand notation for stating system goals. The following are the types of goals that can be specified and trivially translated into CTL formulas (P, P_1 and P_2 represent propositional logic formulas).

- *smi(M, P)* $= AG(\sim M \vee P)$ (strong mode invariant) is satisfied if P holds in every state where the system is in mode M.

- *wmi(M, P)* $= AG((\sim M \vee P) \vee AX(\sim M))$ (weak mode invariant) is satisfied if P holds when the system is in mode M, but not while the system is *exiting* mode M. For example, wmi(Operating, SwitchOn) holds even though an event @F(SwitchOn) occurs when the system is in mode Operating, since the system leaves this mode in all of its next states[2].

[2] The CTL translations clearly identify the difference between *smi* and *wmi* properties. Property $smi(M, P)$ means that for every state of the system P holds whenever the system is in mode M. In

$$
\begin{aligned}
P_8 &= AG(\sim\text{PumpOn} \;\vee\; EX(\sim\text{PumpOn}) \rightarrow (\sim\text{PumpOn} \;\vee \\
&\quad (@F(\text{TooHigh}) \;\&\; \text{Operating}) \;\vee\; (@F(\text{TooLow}) \;\&\; \text{Operating}) \;\vee \\
&\quad (@T(\text{PumpFail}) \;\&\; \text{Operating}) \;\vee\; (@T(\text{PumpFail}) \;\&\; \text{Off}))))\\
P_9 &= AG(EX(\text{PumpOn}) \rightarrow (\text{PumpOn} \;\vee\; (@T(\text{TooHigh}) \;\&\; \text{Operating}) \;\vee \\
&\quad (@T(\text{TooLow}) \;\&\; \text{Operating})))\\
P_{10} &= EF(\text{PumpOn} \;\&\; @F(\text{TooHigh}) \;\&\; \text{Operating} \;\&\; EX(\sim\text{PumpOn}))\\
P_{11} &= EF(\text{PumpOn} \;\&\; @F(\text{TooLow}) \;\&\; \text{Operating} \;\&\; EX(\sim\text{PumpOn}))\\
P_{12} &= EF(\text{PumpOn} \;\&\; @T(\text{PumpFail}) \;\&\; \text{Operating} \;\&\; EX(\sim\text{PumpOn}))\\
P_{13} &= EF(\text{PumpOn} \;\&\; @T(\text{PumpFail}) \;\&\; \text{Off} \;\&\; EX(\sim\text{PumpOn}))\\
P_{14} &= EF(\sim\text{PumpOn} \;\&\; @T(\text{TooHigh}) \;\&\; \text{Operating} \;\&\; EX(\text{PumpOn}))\\
P_{15} &= EF(\sim\text{PumpOn} \;\&\; @T(\text{TooLow}) \;\&\; \text{Operating} \;\&\; EX(\text{PumpOn}))
\end{aligned}
$$

Figure 5: Properties for controlled variable PumpOn.

- *cause*$(P_1,\ P_2) = AG(\sim P_1 \vee (P_2 \vee AX(P_2)))$ is satisfied if whenever P_1 is true, either P_2 is true or the next transition of the system will establish a state in which P_2 holds. For example, the property cause(Operating & TooHigh, PumpOn) in SWLMS specifications indicates that as soon as the system reaches a state where it is in mode Operating and the water level is too high, the pump is immediately turned on in all next states.

- *reach*$(P) = EF(P)$ is satisfied if the system can reach a state in which Property holds. reach properties are usually used to ensure that mode invariants are not satisfied vacuously.

These types of system goals frequently appear in requirements documents, and during our case studies we did not need to verify any other types of properties. Requirements specifications of realistic size and complexity (such as [2]), however, contain global properties which cannot be expressed using only these assertions. Allowing the user to specify a richer set of properties is relatively easy, although not every CTL-expressible formula can be verified during our analysis (see [12]).

5.2 Design Language

To create a design, programmers use a mixture of control-flow constructs, comments and *annotations* - special statements which record mode transitions and changes to the values of controlled and monitored variables. These changes are local (rather than invariant) and therefore are relatively easy to specify. An *Initial annotation* indicates the starting state of each mode class. It unconditionally assigns values to variables. This annotation corresponds to initialization information specified in the requirements. An *Update annotation* assigns values to variables, identifying

contrast, property $wmi(M, P)$ means that P holds only in those states where the system is in mode M and remains in M in at least one of its next states. The concept of a weak mode invariant was not needed at the requirements level because mode transitions are considered to occur at the same time as their triggering events.

points at which the program changes its state. An *Assert annotation* reflects a programmer's knowledge that variables have particular values in the current system state. Static analysis usually gives imprecise results because system states are aggregated. Assert annotations improve the precision of analysis based on what the programmer *believes* to be true.

A designer might also use other statements in addition to annotations and control-flow constructs. We do not process the former since they do (should) not reflect changes of values of requirements variables. To differentiate between statements and annotations, the later start with @@.

Designs of programs implementing SCR requirements typically consist of an Initial annotation marking the starting state, and a loop in which the system reads relevant monitors, tests their values and then changes its mode and the values of controlled variables. Figure 6 shows the design of the Simplified Water-Level Systen from Section 2 written by following this method. Assert annotations are inserted at locations where predicates are tested, and Update annotations are inserted where program changes occur. Annotations in the form @@ Update Top(V) are used to indicate that V has been assigned some unknown value (true or false). Mode transitions and changes to the values of controlled variables occur after events have been detected. In Figure 6, controlled variables are updated on lines 17, 34, 38 and 44. Such a design could have been automatically generated. Unfortunately, generating designs for complex systems with several mode classes is not a trivial task, and we make no attempts to automate it. Experienced designers might want to hand-optimize their designs by "factoring out" common actions for different states. Figure 7 shows the result of such a hand-optimization. Here we noticed that the pump can be turned on only when the system is in mode Operating, and thus the system reads the water level and determines the state of the pump in the inner while loop, exiting it when the switch is turned Off.

For the remainder of this section we show how we can check consistency between our designs and requirements. After a detailed design is completed and verified, it can be further refined to build an implementation.

5.3 Creating an Abstraction

The Detailed Design(DD) is used to create a Design Flow Graph (*DFG*) which is abstracted into a Finite-State Machine (*FSM*) (see Figure 8). Afterwards, the *FSM* is used to check system properties. We construct a Design Flow Graph (DFG) from annotations and control-flow information and then propagate the state information about the requirements variables throughout the DFG.

Definition 5.1 *A Design Flow Graph DFG of a program design DD is a directed graph* $G = \langle S, E, s_0 \rangle$, *where*

S is a finite set of nodes corresponding to splits, joins and annotations of DD.

$E \subseteq S \times S$ is a set of directed edges, s.t. $(s_1, s_2) \in E$ iff s_2 can immediately follow s_1 in some execution sequence; and

$s_0 \in S$ is an entry node.

Let $U \subseteq S$ and $I \subseteq S$ be sets of nodes containing Update and Initial annotations, respectively. (Sets U and I are disjoint.)

```
{ ...
  @@ Initial Off & ~SwitchOn & ~PumpFail & ~TooHigh & ~TooLow & ~PumpOn;
  while (1) {
    if (IN_OFF(System)) {
      @@ Assert Off;  /* the system is in mode Off */
      READ_SWITCH();  /* read switch monitor       */
      @@ Update Top(SwitchOn);
      if (SWITCH_ON() && !PUMP_FAIL()) {
        @@ Assert SwitchOn & ~PumpFail;
        @@ Update Operating;
      }
      else {
        READ_DEVICE();    /* read state of the pump */
        @@ Update Top(PumpFail);
        if (PUMP_FAIL()) {
          @@ Assert PumpFail;
          @@ Update Error & ~PumpOn;
        }
      }
    }
    else if (IN_OPERATING(System)) {
      @@ Assert Operating;
      READ_SWITCH();
      @@ Update Top(Switch);
      if (!SWITCH_ON() && !PUMP_FAIL()) {
        @@ Assert ~SwitchOn & ~PumpFail;
        @@ Update Off;
      }
      else {
        GET_WATER_LEVEL (&Water);    /* compute water level */
        @@ Update Top(TooHigh) & Top(TooLow);
        if (IS_HIGH(Water) || IS_LOW(Water)) {
          @@ Assert TooHigh | TooLow;
          @@ Update PumpOn;
        }
        else {
          @@ Assert ~TooHigh & ~TooLow;
          @@ Update ~PumpOn;
        }
        READ_DEVICE();
        @@ Update Top(PumpFail);
        if (PUMP_FAIL()) {
          @@ Assert PumpFail;
          @@ Update Error & ~PumpOn;
        }
      }
    }
  }
  ...
}
```

Figure 6: Initial SWLMS design.

We interpret annotations in the design to create a *set-based approximation* of attainable values for each requirements' variable[16, 17]. The details of building a DFG are given in [12]. The *DFG* that we create contains nodes which do not reflect state changes (e.g., decision nodes, joins and

```
{ ...
  @@ Initial Off & ~SwitchOn & ~PumpFail & ~TooHigh & ~TooLow & ~PumpOn;
  while (1) {
    @@ Assert ~PumpFail;   /* assume no device failures */
    READ_SWITCH();         /* read switch monitor */
    @@ Update Top(SwitchOn);
    if (SWITCH_ON() && IN_OFF(System)) {
      @@ Assert SwitchOn & Off;
      @@ Update Operating;
    }
    else if (IN_OPERATING(System)) {
      @@ Assert Operating;
      while (1) {
        READ_SWITCH();
        @@ Update Top(SwitchOn);
        if (!SWITCH_ON()) {
          @@ Assert ~SwitchOn;
          @@ Update Off;
          break;
        }
        GET_WATER_LEVEL (&Water); /* compute water level */
        @@ Update Top(TooHigh) & Top(TooLow);
        if (IS_HIGH(Water) || IS_LOW(Water)) {
          @@ Assert TooHigh | TooLow;
          @@ Update PumpOn;
        }
        else {
          @@ Assert ~TooHigh & ~TooLow;
          @@ Update ~PumpOn;
        }
      }
    }
    READ_DEVICE();            /* read device monitor */
    @@ Update Top(PumpFail);
    if (PUMP_FAIL()) {
      @@ Assert PumpFail;
      @@ Update Error & ~PumpOn;
    }
  }
  ...
}
```

Figure 7: Final SWLMS design.

nodes containing Assert annotations). We use the *DFG* to construct a Finite-State Machine (*FSM*). We start with all nodes of the *DFG* and remove those not representing state changes. The resulting *FSM* is then used as a model for verifying system properties.

Definition 5.2 *A Finite-State Machine (FSM) over a detailed design DD is a Kripke structure* $M = \langle V, S, s_0, R, I\rangle$, *where*

V *is a set of modes and conditions;*

$S = U \cup I$ *is a finite set of nodes;*

$s_0 \in S$ *is an entry node;*

$R \subseteq S \times S$ *is a total transition relation*[3], *i.e.* $\forall x \in S\ \exists y \in S$ *s.t.* $(x, y) \in R$;

[3] We add loops at terminal states since our specifications describe only infinite behaviors.

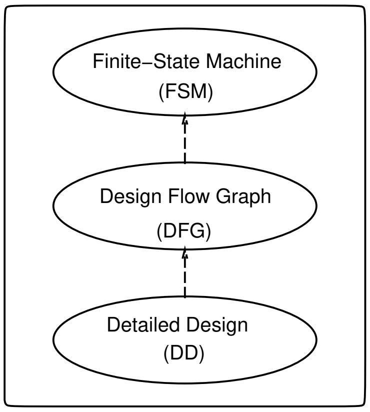

Figure 8: Layers of abstraction.

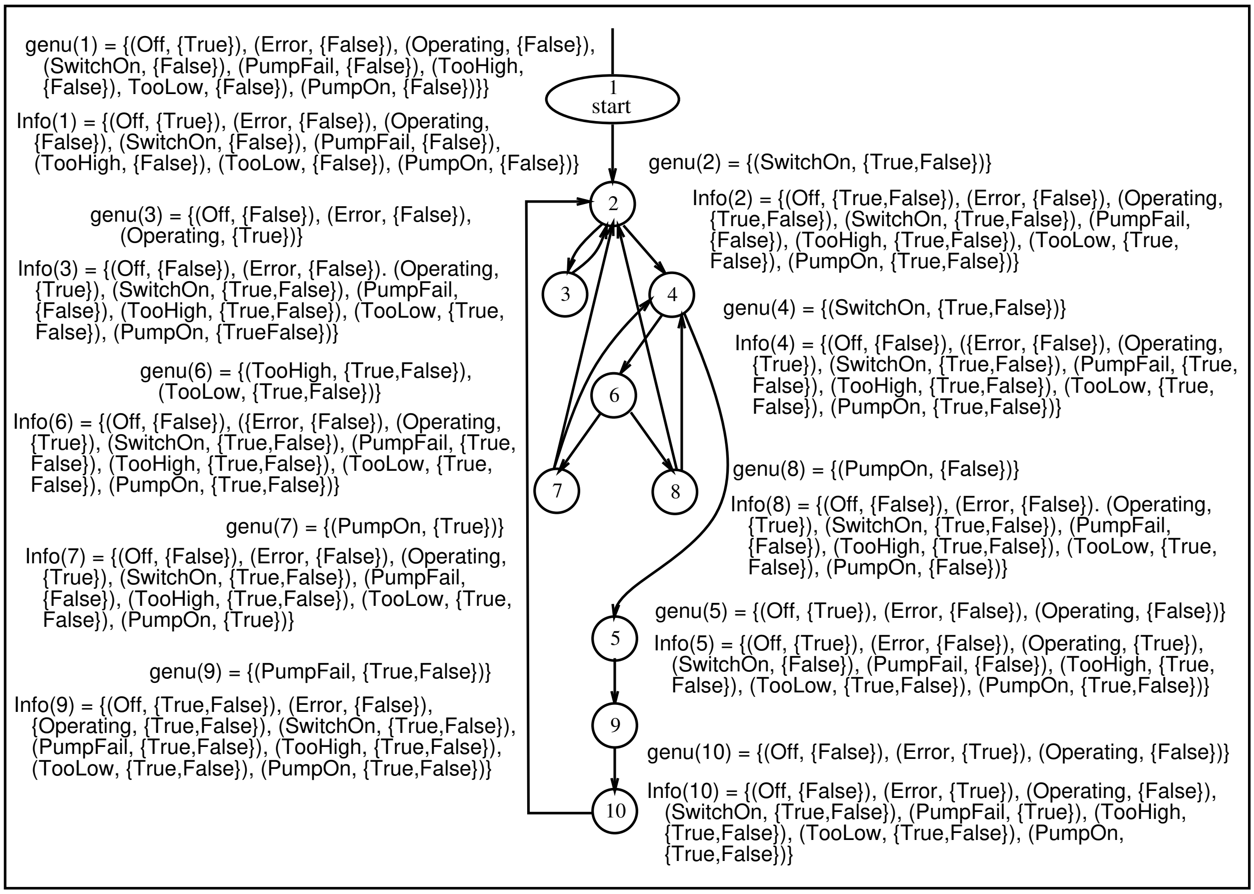

Figure 9: Finite-state abstraction.

R is obtained by connecting nodes of S s.t. there is an Update-clear path between them in DFG*; and*

I: $S \rightarrow 2^V$ *is a function associating each node with a truth assignment for each* *variable in V.*

When constructing this machine, we connect all predecessors of a removed node in *DFG* to all of this node's successors. Figure 9 shows the *FSM* which abstracts the SWLMS design in Figure 6. A label for each node consists of its `Info` and $\texttt{gen}_u$ sets. For a state s, we use the notation $\texttt{gen}_u(s)$ and $\texttt{Info}(s)$ to refer to these sets separately. $\texttt{gen}_u$ sets specify changes made to state information at the node (generated by Update statements), and `Info` sets contain all information about system variables available at this point.

5.4 Verifying Properties

In this section we describe how automatically-generated properties and system goals are verified. During the process described in the previous section, we took a detailed design and built a *FSM* from it, combining all possible values of each variable at every program point and removing nodes which do not correspond to state changes. Unfortunately, since our analysis is static, we are unable to determine which paths through the resulting *FSM* are feasible, and thus we cannot verify properties exactly, that is, we cannot claim that a property is violated if and only if we report a violation. We have built our own model checker to facilitate property verification since our method for constructing finite-state abstractions from designs produces sets of values for each program variable. Available model checkers were not able to interpret these sets correctly. Since we knew which properties we were interested in verifying, we were able to optimize the verification, so that automatically-generated properties can be checked in a single traversal of the *FSM*; however, separate traversals are required for user-specified system goals.

Definition 5.3 *A design is analyzed pessimistically if all (plus perhaps some nonexistent) violations present in the design are reported. A design is analyzed optimistically if all reported violations (plus perhaps some others) are present in the design.*

Thus, if a pessimistic analysis reports no violations of property P, then we know that P holds in the design; and if an optimistic analysis reports any violation of P, then we know that P does not hold in the design. A property P is verified inconsistently if no valid conclusion can be made when a violation of P is reported or when no violations of P are reported.

The OLT properties (like P_1 - P_3 in Figure 4) involve quantification over every possible state. Since reported violations might occur on infeasible paths, we analyze these properties pessimistically. ALT properties (like P_4 - P_7), on the other hand, are existentially quantified. Since a state satisfying these properties in the model might occur on an infeasible path, we analyze these properties optimistically.

During property verification, we need to determine if a first-order formula holds in a state (for example, to verify a formula $AG(a)$, we need to check that a holds in every state), making sure that no properties are verified inconsistently. Variables in states have values represented by sets, so we treat atomic (or negated atomic) propositions in formulas as assertions about sets of values, e.g., PumpOn is treated as PumpOn = $\{True\}$ and $\sim$Operating is treated as

$$\text{Operating} = \{False\} \ \& \ \text{Off} = \{True, False\} \ \& \ \text{Error} = \{True, False\}$$

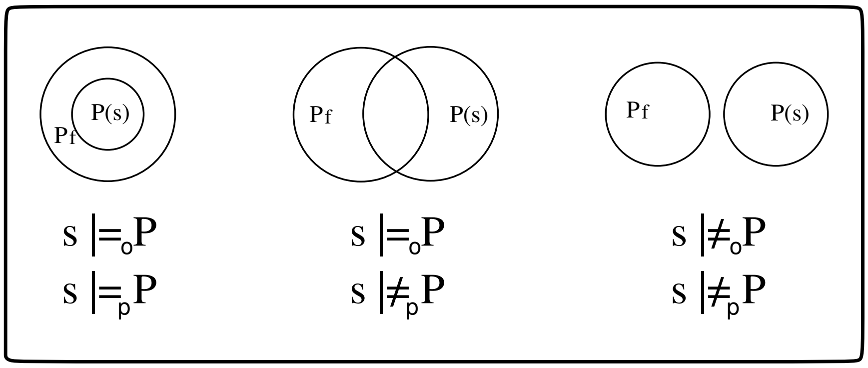

Figure 10: Interpretations of a proposition P holding in state s.

because of environmental assumptions.

We define two interpretations for a proposition P holding in a state s, one for pessimistic interpretation ($s \models_p P$) and another for optimistic interpretation ($s \models_o P$). To check if a state s is a model of a proposition P, we compare P's asserted value (P_f) with its value in state s ($P(s)$) (see Figure 10). For optimistic analysis, if there are common values between P_f and $P(s)$, i.e., $P_f \cap P(s) \neq \emptyset$, we assume that P_f holds in s. In pessimistic analysis we do not want to overlook any errors, and thus we consider P_f to hold in a state s only if $(P(s) \neq \{\}) \wedge (P(s) \subseteq P_f)$, i.e., all values of $P(s)$ are present in P_f.

For a temporal formula P we write $FSM, s_0 \models P$ ($FSM, s_0 \not\models P$) to indicate that *FSM* is (is not) a model of P. Finally, for both types of analysis, we assume that an event @T(a)(@F(a)) occurs at a node containing an annotation which sets a to true(false).

5.4.1 Checking automatically-generated properties

To verify the automatically-generated properties, we take advantage of their rather restricted forms and verify *all* these properties in a *single* traversal of the *FSM*. The CAM property is checked pessimistically by examining every state of the *FSM*. The remaining properties involve state transitions which occur only in response to events. To calculate a node's event, we look for predecessors of the node and construct triggering and WHEN conditions from corresponding $\texttt{gen}_u$ and `Info` sets, respectively. For example, consider calculating the event which results in transition to Error at node 10 of Figure 9. Node 10 corresponds to line 37 of the design in Figure 7. Its predecessor is node 9. The triggering condition for the transition from 9 to 10 is computed from the information specified in $\texttt{gen}_u(9)$ set: @T(PumpFail) & @F(PumpFail). The WHEN conditions are generated from `Info`(9), resulting in the set of events in Figure 11. Controlled variables are not part of events; thus, PumpOn is excluded from the event's conditions. Error, Off and Operating in node 9 become starting modes for a transition. Finally, since PumpFail is restricted to {True} in node 10 because of the Assert annotation at line 36, only the triggering condition @T(PumpFail) could have occured.

Verification of OLT Properties

$$\text{Transition from Off to Error:}$$
$$\begin{aligned} I_1 : @T(\text{PumpFail}) \quad \text{WHEN} \quad & [\text{SwitchOn} = \{True, False\} \ \& \ \text{TooHigh} = \{True, False\} \\ & \& \ \text{TooLow} = \{True, False\} \ \& \ \text{Off} = \{True, False\} \\ & \& \ \text{Operating} = \{True, False\} \ \& \ \text{Error} = \{False\}] \end{aligned}$$

$$\text{Transition from Operating to Error:}$$
$$\begin{aligned} I_2 : @T(\text{PumpFail}) \quad \text{WHEN} \quad & [\text{SwitchOn} = \{True, False\} \ \& \ \text{TooHigh} = \{True, False\} \\ & \& \ \text{TooLow} = \{True, False\} \ \& \ \text{Off} = \{True, False\} \\ & \& \ \text{Operating} = \{True, False\} \ \& \ \text{Error} = \{False\}] \end{aligned}$$

Figure 11: Events and transitions discovered for node 9.

One of the OLT properties automatically generated from the requirements of SWLMS was

$$\begin{aligned} P_3 \quad = \quad & AG(EX(\text{Error}) \rightarrow (\text{Error} \vee \\ & (\text{Operating} \ \& \ @T(\text{PumpFail})) \ \vee \ (\text{Off} \ \& \ @T(\text{PumpFail})))) \end{aligned}$$

All such properties have the general form $P = AG(EX(F) \rightarrow (F \vee G))$. Since $AG(EX(F) \rightarrow (F \vee G)) = AG(AX(\sim F) \vee (F \vee G))$, these properties are universally quantified and thus can be verified pessimistically, possibly identifying illegal transitions even if they actually occur on infeasible paths. Thus, we use a pessimistic interpretation of a property holding in a state. We need to examine only those states in which F is false and in whose successors F is true (node 9 in our example). We check the formula only for predecessors of states labeled with Update annotations for F. In our example, both calculated events (I_1 and I_2) establish P_3 for node 9. We check all OLT properties in one traversal of *FSM* using the algorithm in [12].

Verification of ALT Properties

The following are some of the ALT properties for SWLMS:

$$\begin{aligned} P_5 \quad &= \quad EX(\text{Off} \ \& \ @T(\text{PumpFail}) \ \& \ EF(\text{Error})) \\ P_7 \quad &= \quad EX(\text{Operating} \ \& \ @T(\text{PumpFail}) \ \& \ EF(\text{Error})) \end{aligned}$$

These properties hold in the model because node 9 satisfies them via transitions I_1 and I_2, respectively (see Figure 11). We mark ALT properties indicating that they have been satisfied if the design's calculated event contains values that would cause a requirement's state transition. Any properties remaining unmarked at the end of analysis are reported as errors. This analysis is optimistic since it considers state transitions to satisfy a property even if they occur on infeasible paths. Thus, we might not report all unimplemented transitions. We use an optimistic interpretation for a property holding in a state to ensure that no extra errors are identified.

5.4.2 Verifying system goals

Each of the system goals specified in the requirements is verified in a separate traversal of *FSM*. Reach properties are existentially quantified CTL formulas, and are verified optimistically. All other properties are universally quantified CTL formulas, and are verified pessimistically. We start by verifying the system goals stated in Section 4.4, plus one new goal involving a controlled variable.

- wmi (Operating, (SwitchOn & ~PumpFail))
- wmi (Off, (~SwitchOn & ~PumpFail))
- wmi(Error, PumpFail)
- cause (@T(SwitchOn), ~Off)
- cause (Operating & @T(TooHigh), PumpOn)

Each of these properties is verified in a separate search of the *FSM*. For example wmi(M,P) is equivalent to $AG((\sim M \;\vee\; P) \vee AX(\sim M))$. Thus, we search the *FSM* for the nodes such that M holds in the node and in at least one of its successors, check that P holds in this node, and stop as soon as an error is discovered.

As was the case in our previous analysis, wmi(Error, PumpFail) fails to hold. The system transits to mode Error in node 6 (line 37 on Figure 7). Figure 9 shows that node 2 (line 6) is a successor of node 6 in the *FSM*. Since `Info`(2) contains (PumpFail, $\{False\}$), we report that the property is violated at node 2. The alternative invariant wmi(PumpFail,Error) holds for the model.

6 Case Study

To demonstrate our analysis techniques on more realistic applications, we conducted a case study on a Water-Level Monitoring System (WLMS) which had been specified using SCR requirements and subsequently implemented[29]. Our usual analysis process starts by merging information from environmental assumptions into mode transition tables, and continues by trying to prove that the logic model derived from this combined information is a model of the system goals using model checking. This analysis often results in changes to the mode transition tables and/or to the system goals. After this we proceed to create a design corresponding to the new mode transition tables and any controlled variables' event tables. We make changes to our design to ensure that it is consistent with the properties automatically generated from the requirements and the stated system goals. We deviated from this process in our case study by reverse engineering a design from an existing implementation in order to determine what, if any, errors might be detected with our analysis technique.

6.1 The Application

A Water-Level Monitoring System (WLMS) monitors and displays the water level in a container. It also raises visual and audio alarms, and shuts off its pump when the level is out of range or when the monitoring system fails. Two push buttons, SelfTest and Reset, permit the operator to test the system and return it to normal operation. WLMS has two mode classes, Normal and Failure, whose modes are described in Table 3.

Mode Class	Mode	Meaning
Normal	Operating	The system is running properly.
	Shutdown	The water level is out of range and the system will be shut-down unless conditions change.
	Standby	The system is waiting for the operator to push a button to select test or operating mode.
	Test	The system is not operating, but controlled variables are being checked.
Failure	AllOK	No device failures.
	BadLevDev	The water level cannot be measured.
	HardFail	Unrecoverable failure.

Table 3: WLMS Modes.

Conditions indicate whether the water level in the container is within its limits (WithinLimits) and its more stringent hysteresis limits (InsideHysR). Other conditions indicate the lengths of time that buttons have been pressed (SelfTestPressed500 - SelfTest button pressed for 500ms) or that the system has been in a mode (InTest14000 – in mode Test for 14 seconds). Table 4 summarizes the environmental assumptions that we deduced from descriptions of conditions in the requirements.

Environmental Assumptions
InsideHysR $- >>$ WithinLimits
SelfTestPressed $<$ SelfTestPressed500
ResetPressed $<$ ResetPressed3000
InTest0 $<$ InTest2000 $<$ InTest4000 $<$ InTest14000

Table 4: Environmental assumptions about WLMS conditions.

The system starts in mode Standby of mode class Normal and mode AllOK of mode class Failure. A mode transition table for mode class Normal is shown in Table 5.

Controlled variables are set to trigger alarms and to display the water level to the operator. Table 6 shows the settings for the controlled variable LowWindowOn which represents the annunciation window labeled "Water Level Low." When LowWindowOn is true, the annunciator

Current Mode	Inside HysR	Within Limits	SelfTest Pressed	SelfTest Pressed 500	In Test 14000	Reset Pressed 3000	Shutdown LockTime 200	**New Mode**
Standby	t	–	–	–	–	@T	–	Operating
	–	–	–	@T	–	–	–	Test
Operating	–	@F	f	–	–	–	–	Shutdown
	–	–	–	@T	–	–	–	Test
Shutdown	@T	–	f	–	–	–	f	Operating
	–	–	f	–	–	–	@T	Standby
	–	–	–	@T	–	–	–	Test
Test	–	–	–	–	@T	–	–	Standby

Initial: Standby (~SelfTestPressed500 & ~ResetPressed3000 & ~InTest14000 & ~ShutdownLockTime200)

Table 5: Original mode transition table for mode class Normal.

displays a value. This happens when the water level falls below its limits (i.e., LevelLow) or when the water level cannot be measured (BadLevDev) and a certain amount of time has passed (FlashOff500).

Mode	Triggering Event	
Operating $\wedge$ AllOK	@T(LevelLow)	@T(Inmode) WHEN [~LevelLow]
Shutdown $\wedge$ AllOK	@T(LevelLow)	
Test $\wedge$ AllOK	@T(InTest2000)	@T(InTest4000)
BadLevDev	@F(FlashOff500)	@T(FlashOff500)
LowWindowOn =	True	False

Initial: True

Table 6: Event table for controlled variable LowWindowOn.

The WLMS requirements document did not contain a list of system goals. We inferred that the following four properties should be invariant and corroborated this with the requirements designer.

1. If the SelfTest button has been pressed for 500ms or more, the system is either in mode Test or will be in mode Test after its next transition.

2. When the system is in mode Standby, the SelfTest button has not been pressed for 500ms.

3. If the system is in mode Operating, then the SelfTest button has not been pressed for 500ms, and either the water level is within limits or the SelfTest button is being pressed.

4. If the system is in mode Shutdown, then the SelfTest button has not been pressed for 500 time units (or the system would have transitioned into the Test mode); and either the system has been in the Shutdown mode for less that 200 time units, during which the water-level has remained outside the hysteresis water-level range, or the SelfTest button is being pressed (indicating an imminent transition to mode Test).

6.2 Requirements Analysis

The first step in our requirements analysis is to add information from the environmental assumptions to mode transition tables. The relationships InsideHysR $- >>$ WithInLimits and SelfTestPressed $<$ SelfTestPressed500 add information to seven transitions. For example, consider the transition from Operating to Shutdown. InsideHysR must already be false if WithinLimits is becoming false, and SelfTestPressed500 must be false since SelfTestPressed is false. Table 7 is a mode transition table with the environmental assumption information added.

Current Mode	Inside HysR	Within Limits	SelfTest Pressed	SelfTest Pressed 500	In Test 14000	Reset Pressed 3000	Shutdown LockTime 200	**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: Standby (~SelfTestPressed500 & ~ResetPressed3000 & ~InTest14000 & ~ShutdownLockTime200)

Table 7: Detailed mode transition table for mode class Normal.

From the detailed mode transition table, we create a logic model and translate our invariants in CTL formulas.

1. $AG((\text{SelfTestPressed500} \wedge \sim\text{Test}) \rightarrow AX(\text{Test}))$

2. $AG(\text{Standby} \rightarrow \sim\text{SelfTestPressed500})$

3. $AG(\text{Operating} \rightarrow (\sim\text{SelfTestPressed500} \wedge (\text{WithinLimits} \vee \text{SelfTestPressed})))$

4. AG(Shutdown → (∼SelfTestPressed500 ∧ ((∼InsideHysR ∧ ∼ShutdownLockTime200) ∨ ∼SelfTestPressed)))

These assertions are processed using the techniques in Section 3 since $AG(f)$ is just $\sim EF(f)$. The SMV model checker found counter-examples to each of these invariants.

The first formula failed because the two transitions leaving mode Standby can be simultaneously enabled if both SelfTestPressed500 and ResetInterval become true at the same time while their respective WHEN conditions are also true. To make the first formula hold, we gave priority to the transition to Test by adding the WHEN condition ∼SelfTestPressed500 to the transition from Standby to Operating.

The second formula is not invariant because SelfTestPressed500 is not always false on entry to Standby. If the operator presses the SelfTest button when the system is in mode Test, the button may have been pressed long enough to make SelfTestPressed500 true before the system transitions from mode Test to mode Standby. Adding the WHEN condition ∼SelfTestPressed500 to the transition between modes Test and Standby ensures that SelfTestPressed500 is always false, but may cause the system to remain in mode Test indefinitely. If the operator is pressing the SelfTest button when InTest14000 becomes true, not only will the transition leaving mode Test not be activated, but it will never be activated since InTest14000 will never be satisfied again. To avoid this, we add a second transition from Test to Standby that is activated if the operator releases the SelfTest button after the system has already spent 14 seconds in mode Test.

The failure of the third formula is critical because it is meant to ensure that the system does not remain in mode Operating with the water level outside its limits. If the system is in mode Operating and the SelfTest button is being pushed, the system will remain in this mode even if the water level goes outside its limits because of the expectation that the next mode transition should be to mode Test. However, if the button is released before 500 milliseconds pass, the system remains in Operating regardless of the water level. In addition, the transition from Operating to Shutdown is disabled because WithinLimits has already become false and can no longer be detected as a triggering condition for an event. The transition from Operating to Shutdown should depend on the button not being pressed long enough to enable a different transition from Operating (to Test), rather than just on the button being pressed. Thus we replace ∼SelfTestPressed with ∼SelfTestPressed500 in this transition's WHEN condition. We also change the invariant to

$$AG(\text{Operating} \rightarrow \text{WithinLimits})$$

since we do not care if the button is being pressed.

The fourth formula fails for the same reason as the third one did. InsideHysR or Shutdown-LockTime200 becoming true will fail to trigger events if the SelfTest button is depressed. We make the same replacements in the WHEN conditions for the events these conditions trigger as we did for the transition from Operating to Shutdown, and change the invariant to

$$AG(\text{Shutdown} \rightarrow (\sim\text{InsideHysR} \wedge \sim\text{ShutdownLockTime200})).$$

6.3 Design Analysis

From the corrected requirements in Table 8 and invariants, we would usually proceed to create a design. However, in this case study we were interested in finding errors in the existing system.

Current Mode	Inside HysR	Within Limits	SelfTest Pressed	SelfTest Pressed 500	In Test 14000	Reset Pressed 3000	Shutdown LockTime 200	**New Mode**
Standby	t	t	–	f	–	@T	–	Operating
	–	–	t	@T	–	–	–	Test
Operating	f	@F	–	f	–	–	–	Shutdown
	–	–	t	@T	–	–	–	Test
Shutdown	@T	t	–	f	–	–	f	Operating
	–	–	–	f	–	–	@T	Standby
	–	–	t	@T	–	–	–	Test
Test	–	–	f	f	@T	–	–	Standby
	–	–	@F	f	t	–	–	Standby

Initial: Standby (~SelfTestPressed500 & ~ResetPressed3000 & ~InTest14000 & ~ShutdownLockTime200)

Table 8: Corrected mode transition table for mode class Normal.

Since the existing implementation used the original transition tables and invariants, we did not modify them for our study either. To build a design, we reverse engineered an existing implementation of WLMS, originally consisting of roughly 1300 lines of FORTRAN and Assembler code. The resulting design was about 700 lines long, with 19 Update annotations corresponding to monitored variables, 30 Update annotations corresponding to mode class and controlled variables, and 60 Assert annotations. Out of 54 functions in the original program, only eight had state changes and thus were included into the design.

Property Type	Violations	Locations
OLT properties for mode classes	4	3
OLT properties for controlled variables	53	7
ALT properties for mode classes	9	
ALT properties for controlled variables	28	
User-defined properties	1	

Table 9: Results of analyzing the WLMS.

After we eliminated annotation errors in the design, we used the original mode transition table (Table 5, which had also been used by the original programmer) to perform our analysis. Our tool reported a number of inconsistencies between the requirements and the design (see Table 9). These numbers overestimate the actual errors in the design. First, some mode transitions and controlled variable value changes resulted in a number of OLT property violations. For example, three illegal mode transitions generated four violation messages because several illegal transitions were detected at one location. All of the mode transition problems can be attributed to three principal causes: the wrong monitored variable was checked to enable mode transitions (WithinLimits rather than InsideHysR), the times that the operator pressed the SelfTest and Reset buttons were not calculated or checked, and no transitions to a mode corresponding to the complete system failure were implemented. Most of the illegal assignments to the controlled variables occurred because the order of triggering events in the design differed from that in the requirements.

We started with the same four invariants as in our requirements analysis, but adapted them to our tool's assertion notation.

1. cause(@T(SelfTestPressed500), Test)
2. wmi(Standby, ~SelfTestPressed500)
3. wmi(Operating, (~SelfTestPressed500 & (WithinLimits | SelfTestPressed)))
4. wmi(Shutdown, (~SelfTestPressed500 &
   ((~InsideHysR & ~ShutdownLockTime200) | ~SelfTestPressed)))

Each of the first three user-defined properties was verified by Analyzer. These results are not particularly interesting because all the safety properties included a negated monitored variable (SelfTestPressed500) that was never assigned a value true in the design. Thus, the first two formulas hold trivially. The third formula was verified since WithinLimits was set to true before each transition to Operating, and whenever WithinLimits and SelfTestPressed were set to false, the system immediately transitioned to mode Shutdown. Analyzer reported a violation of the fourth property because InsideHysR received the value {True,False} as a result of its relationship to WithinLimits. This value was propagated throughout the graph, and, since wmi properties are verified pessimistically, Analyzer reported a potential fault. In fact, InsideHysR is never explicitly set to true in the code, and thus this property holds in the design.

7 Conclusion

This paper presented our work on using model checking to verify requirements and designs which is part of an on-going effort to develop automated techniques that use formal methods to verify program artifacts.

We have presented a logic-model semantics for SCR behavioral requirements specifications and have proposed modal logic operators for expressing formulas over modes, conditions and events. This enables us to verify CTL-expressed system goals with respect to an SCR logic model using the SMV model checker. We have also defined a notion of consistency between SCR-style requirements and a detailed design and presented an automated technique that uses transition tables to generate temporal logic formulas capturing this notion. We have shown how to create a finite-state abstraction of a detailed design and model check it against these formulas. We have applied these techniques to analyze the requirements and design of a Water-Level Monitoring System, uncovering several errors.

Acknowledgements

This research was supported by the Air Force Office of Scientific Research under contract F49620-93-1-0034 and by the Natural Sciences and Engineering Research Council under grant OGP0155243. We thank David Parnas and John van Schouwen for providing the requirements and an implementation of the Water-Level Monitoring System. Rich Gerber and Bill Pugh gave us valuable technical suggestions about model checking and abstract interpretation. We also benefited from discussions with Connie Heitmeyer about the semantics of SCR, and with Stuart Faulk about creating designs from SCR specifications.

References

[1] R. Allen and D. Garlan. "Formalizing Architectural Connection". In *Proceedings of the Sixteenth International Conference on Software Engineering*, May 1994.

[2] 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.

[3] J. Atlee. *"Automated Analysis of Software Requirements"*. PhD thesis, University of Maryland, College Park, Maryland, December 1992.

[4] J.M. Atlee and J. Gannon. "State-Based Model Checking of Event-Driven System Requirements". *IEEE Transactions on Software Engineering*, pages 22–40, January 1993.

[5] Joanne M. Atlee and Michael A. Buckley. "A Logic-Model Semantics for SCR Software Requirements". In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, January 1996. To appear.

[6] Joanne M. Atlee and John Gannon. "Analyzing Timing Requirements". In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 117–127, Cambridge, MA, June 1993.

[7] G. Barrett. "Model Checking in Practice: The t9000 Virtual Channel Processor". *IEEE Transactions on Software Engineering*, 21(2):69–78, February 1995.

[8] M. Browne. *"Automatic Verification of Finite State Machines Using Temporal Logic"*. PhD thesis, Carnegie Mellon University, 1989.

[9] Tevfik Bultan, Jeffrey Fischer, and Richard Gerber. "Compositional Verification by Model Checking for Counter-Examples". In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, January 1996. To appear.

[10] M. Chechik and J. Gannon. "Automatic Verification of Requirements Implementations". In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–14, Seattle, Washington, August 1994.

[11] M. Chechik and J. Gannon. "Automatic Analysis of Consistency Between Implementations and Requirements: A Case Study". In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 123–131, June 1995.

[12] M. Chechik and J. Gannon. "Automatic Analysis of Consistency Between Requirements and Designs". Technical report CS-TR-3394.1, Dept. of CS, University of Maryland, College Park, October 1995.

[13] Edmind M. Clarke, Orna Grumberg, and David E. Long. "Model Checking and Abstraction". In *Proceedings of the Ninth Annual Symposium on Principles of Programming Languages*, pages 343–354, August 1992.

[14] E.M. Clarke and E.A. Emerson. *"Synthesis of synchronization skeletons for branching time temporal logic"*, volume 131 of *Lecture Nodes in Computer Science*, chapter Logic of Programs: Workshop, Yorktown Heights, NY, May 1981. Springer-Verlag, 1981.

[15] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[16] Patrick Cousot and Radhia Cousot. "Static Determination of Dynamic Properties of Programs". In *Proceedings of the "Colloque sur la Programmation"*, April 1976.

[17] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of the 4th POPL*, pages 238–252, Los Angeles, California, 1977.

[18] David Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

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

[20] C. Heitmeyer and B. Labaw. "Consistency Checks for SCR-Style Requirements Specifications". Technical Report NRL Report 93-9586, Naval Research Laboratory, November 1993.

[21] C. Heitmeyer, B. Labaw, and D. Kiskis. "Consistency Checking of SCR-Style Requirements Specifications". In *Proceedings of RE'95 International Symposium of Requirements Engineering*, March 1995.

[22] 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.

[23] Daniel Jackson and Craig A. Damon. "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector". In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA)*, January 1996. To appear.

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

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

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

[27] D.L. Parnas. "Some Theorems We Should Prove". In *Proceedings of 1993 International Conference on HOL Therem Proving and Its Applications*, Vancouver, BC, August 1993.

[28] Tirumale Sreemani and Joanne M. Atlee. "Feasibility of Model Checking Software Requirements: A Case Study". Technical Report CS-95-48, Department of Computer Science, University of Waterloo, November 1995.

[29] A. 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, Queen's University, Kingston, Ontario, May 1990.

[30] Jeannette Wing and Mandana Vaziri-Farahani. "Model Checking Software Systems: A Case Study". In *Proceedings of the 3rd Symposium on the Foundations of Software Engineering*, pages 128–139, October 1995.