REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE 1997	3. REPORT TYPE AND DATES COVERED Final AFOSR-TR-97-0611

4. TITLE AND SUBTITLE

Analyzing Safety Properties of Requirements

5. FUNDING NUMBERS

Grant F496209310034

F49620-93-1-0034

6. AUTHOR(S)

Joanne Atlee, Marsha Chechik, and John Gannon

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

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

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFOSR/NM
110 Duncan Avenue Room B115
Bolling AFB DC 20332-8080

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

19971204 190

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release;
distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

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 report 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.

14. SUBJECT TERMS

Formal methods, Requirement analysis, Model checking

15. NUMBER OF PAGES
34

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT unclassified	20. LIMITATION OF ABSTRACT UL

Analyzing Safety Properties of Requirements

Joanne Atlee
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

Marsha Chechik
Dept. of Computer Science
University of Toronto
Toronto, Ontario, M5S 1A4

John Gannon
Dept. of Computer Science
University of Maryland
College Park, Maryland 20742

1 Introduction

Precise documentation of software requirements has several potential benefits[31]: 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[3, 21, 23], the Requirements State Machine Language[27], and Statecharts[19]) 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 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[5, 7, 6, 14, 15].

In this report, we summarize our work to analyze program requirements and designs. We use model checking[17] 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., [10, 8, 16]). More recently it has been used to analyze software artifacts. Model checking has been used to detect design flaws in software architecture designs[2] and Z specifications[26], and to prove properties of cache coherence protocols[35] and concurrent Ada programs[11]. 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 requirements into a logical model. We represent a system's safety assertions as logical formulas in a branching-time temporal logic, Computation Tree Logic[18] (CTL), and use existing CTL model checkers[10, 28] 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 report is organized as follows: Section 2 introduces the SCR 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 6 presents our techniques for verifying designs: how to generate first-order logic properties from SCR tables, how build finite-state abstractions of designs, and how to check the properties using a special-purpose model checker. Section 10 describes a case study in which we analyzed the requirements and design of a Water-Level Monitoring System[33]. We present our conclusions in Section 11.

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 [3, 23]. 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 SCR requirements, environmental variables are monitored, and their values are translated to input data values for a a set of finite state machines (FSMs). The FSMs record the system's states and set the values of output data items. The values of output data items control variables in the system's environment[29, 33].

2.1 Behavioral Requirements

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 [~ 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 [3]. 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 [29], 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 [4]; for the purposes of this report, 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[18]. 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 report. $AG(f)$ 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., $\{s \mid s \models f\}$. 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[28]. 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.

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 [20].

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 [22]. 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 [3] 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 1 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.

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

Figure 1: SCR logic model of SWLMS mode transition table.

4.3 Environmental Assumptions

The formula in Figure 1 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.

$$
\begin{array}{l}
\bigwedge \quad (\text{Off} \vee \text{Operating} \vee \text{Error}) \bigwedge (\text{Off} \rightarrow (\sim \text{Operating} \wedge \sim \text{Error})) \bigwedge \\
\qquad (\text{Operating} \rightarrow (\sim \text{Off} \wedge \sim \text{Error})) \bigwedge (\text{Error} \rightarrow (\sim \text{Off} \wedge \sim \text{Operating})) \bigwedge \\
\qquad (\text{TooLow} \rightarrow \sim \text{TooHigh}))
\end{array}
$$

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[28]. 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.

6 Consistency with SCR Requirements

Definition 6.1 *Let a set of SCR requirements $\mathcal{R} = \langle \mathcal{B}, \mathcal{E} \rangle$ be given. A program artifact $\mathcal{A}$ constrained by environmental assumptions $\mathcal{E}$ (called $\mathcal{A}_\mathcal{E}$) is consistent with its requirements $\mathcal{R}$ if*

1. *$\mathcal{A}_\mathcal{E}$ and $\mathcal{B}$ have the same starting state;*
2. *$\mathcal{A}_\mathcal{E}$ implements all state transitions specified in the $\mathcal{B}$; and*
3. *$\mathcal{A}_\mathcal{E}$ does not implement any state transitions which are not specified in $\mathcal{B}$*

This is a very restricted definition. Typically, artifacts are considered consistent with requirements when they implement at least what is specified. SCR was developed to specify high-assurance systems, and intended to capture all allowed system behaviors. Indeed, it is clearly a fault if an artifact implements an unspecified transition to a state representing a failure.

SCR tables can be transformed into a list of properties which capture this notion of consistency. To prove that an artifact is consistent with its requirements, we demonstrate that it is a model of *all* these properties. We express these properties as first-order logic formulas.

We use states to denote points at which variables change values. Thus, three states need to be considered in determining if an event caused a mode change: the state in which the monitored variable had its original value, the state in which the monitored variable was assigned a new value, and the state in which the mode change occurred. Generally, we do not know the value assigned to a monitored variable until after the outcome of a test. Thus, monitored variables may have either the value true or the value false in a state; the exact value not being known until an arc representing a successful test outcome is traversed. As the result, we define formulas as being true on arcs as well as in states. A transition between m_i and m_j triggered by @T(a) WHEN [b] is formalized as

$$(a = \text{false}) \ \wedge \ (m' = m_i) \ \wedge \ (b' = \text{true}) \ \wedge \ (a' = \text{true}) \ \wedge \ (m'' = m_j),$$

where a condition represents its value on the previous edge, a primed condition represents its value on the current edge, and the double-primed condition represents its value in the adjacent state. Figure 2 gives a pictorial representation of this semantics.

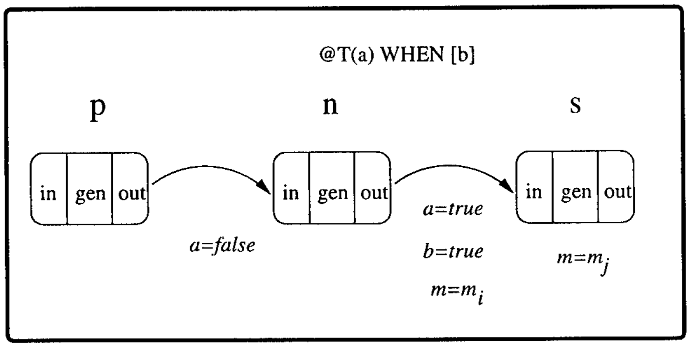


Figure 2: Pictorial representation for a mode transition.

For a state n and a formula f, we use the notation $n \models f$ to indicate that f is true in n. For a pair of states (n, p), we use the notation $(n, p) \models f$ to indicate that f is true on an edge between n and p. We also assume that for each state n we have functions pred(n) and succ(n) returning a list of all predecessors and successors of n, respectively. (This list can be empty.) Thus, we can express a property "there exists a transition from $m = m_i$ to $m = m_j$ on @T(a) WHEN [b]" as

$$\exists n, \ \exists s \in \text{succ}(n), \ \exists p \in \text{pred}(n), \ (s \models (m = m_j)) \wedge$$
$$((n, s) \models ((a = \text{true}) \ \wedge \ (b = \text{true}) \ \wedge \ (m = m_i))) \ \wedge \ ((p, n) \models (a = \text{false}))$$

A number of properties generated from SCR have the notion of an event in them. We say that an event @T(a), where a is a boolean variable, has occurred on an edge (n, s), i.e. $(n, s) \models$ @T(a), if

$$((n, s) \models (a = \text{true})) \ \wedge \ (\exists p \in \text{pred}(n), \ (p, n) \models (a = \text{false}))$$

A set of properties capturing first two parts of our definition of consistency can be obtained by composing rows and columns of SCR tables. In Section 2.1 we introduced a simple Water-Level Monitoring System (SWLMS). We use the SWLMS requirements to illustrate the kinds of properties which are generated to capture our notion of consistency with SCR requirements. For example, we have a property asserting that the only way for mode MC to be in mode Off in its next state is if MC is currently in Off, or if a transition from mode Operating occurs in response to an event @F(SwitchOn) WHEN ¬PumpFail. This property was obtained by composing the rows of the MC mode transition table which have Off in their right columns (in this case, only row three). We write this property as

$$\begin{aligned} P_1 \quad = \quad & \forall n, \ n \models (\text{MC=Off}) \rightarrow (\forall p \in \text{pred}(n), \ ((p, n) \models (\text{MC=Off})) \\ & \vee \ ((p, n) \models (\text{@F(SwitchOn)} \wedge (\text{PumpFail=false}) \wedge (\text{MC=Operating})))) \end{aligned}$$

Properties quantified on all members pred(n) or succ(n) are considered vacuously true when the corresponding list is empty.

$$\begin{aligned}
P_1 &= \forall n,\ n \models (\text{MC=Off}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Off}) \\
&\quad \vee (p,n) \models (\text{@F(SwitchOn)} \wedge (\text{PumpFail=false}) \wedge (\text{MC=Operating}))) \\
P_2 &= \forall n,\ (n \models (\text{MC=Operating}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Operating}) \\
&\quad \vee (p,n) \models (\text{@T(SwitchOn)} \wedge (\text{PumpFail=false}) \wedge (\text{MC=Off}))) \\
P_3 &= \forall n,\ (n \models (\text{MC=Error}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Error}) \ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Operating})) \ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Off}))) \\
P_4 &= \forall n,\ (n \models (\text{PumpOn=false}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{PumpOn=false}) \ \vee \\
&\quad (p,n) \models (\text{@F(TooHigh)} \wedge (\text{MC=Operating})) \ \vee \\
&\quad (p,n) \models (\text{@F(TooLow)} \wedge (\text{MC=Operating})) \ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Operating})) \ \vee \\
&\quad (p,n) \models (\text{@T(PumpFail)} \wedge (\text{MC=Off}))) \\
P_5 &= \forall n,\ (n \models (\text{PumpOn=true}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{PumpOn=true}) \ \vee \\
&\quad (p,n) \models (\text{@T(TooHigh)} \wedge (\text{MC=Operating})) \ \vee \\
&\quad (p,n) \models (\text{@T(TooLow)} \wedge \text{MC=Operating}))
\end{aligned}$$

Figure 3: OLT properties for SWLMS.

We generate properties similar to P_1 for each value of controlled variables and every mode in the right columns of mode transition tables. These properties capture the third part of our notion of consistency and are called "only legal transitions" (OLT) properties. OLT properties generated from requirements of SWLMS are shown in Figure 3. Property P_1 was generated from row three of Table 1; P_2 from row one; and P_3 from a composition of rows two and four. Properties P_4 and P_5 were generated for the controlled variable PumpOn from Table 2. There are two OLT properties generated for each controlled variable, reflecting changes of value to false (P_4) and to true (P_5).

Another property asserts that there exists a transition from mode Off to mode Operating on an event @T(SwitchOn) WHEN [PumpFail=false]. This property corresponds to the first row of Table 1. We express this property as

$$\begin{aligned}
P_6 &= \exists n,\ \exists p \in \text{pred}(n),\ n \models (\text{MC=Operating}) \wedge \\
&\quad (p,n) \models ((\text{MC=Off}) \wedge \text{@T(SwitchOn)} \wedge (\text{PumpFail=false}))
\end{aligned}$$

Such properties ensure that all transitions specified in the requirements (potentially) appear in the design, capturing the second part of our notion of consistency. We call them "all legal transitions"(ALT) properties. One ALT property is generated for every row of transition tables for mode classes and controlled variables. Other ALT properties for SWLMS are shown in Figure 4. Properties P_6-P_9 were generated from Table 1 (rows 1-4, respectively). Properties P_{10}-P_{15} were generated from Table 2 (rows 1-6, respectively). Of course, these properties mean that there *may be* a path to a transition. Although we are able to find unreachable states, we are not always able to find and eliminate infeasible paths.

7 Detailed Design

A Program Design Language (PDL)[12] is a language used to specify designs. Typically, PDLs[12] are defined by an *outer syntax* of control structures and *inner syntax* of other statements. Our PDL's outer syntax is a set of C-like control structures. Our inner syntax consists of *annotations* - special statements describing values of requirements variables. Our use of annotations was inspired by Howden's work on QDA[25, 32].

For sequential designs, we defined three types of annotations:

$$
\begin{aligned}
P_6 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Operating}) \land \\
&\quad (p,n) \models ((\mathrm{MC{=}Off}) \land \mathrm{@T(SwitchOn)} \land (\mathrm{PumpFail{=}false})) \\
P_7 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Error}) \ \land\ (p,n) \models ((\mathrm{MC{=}Off}) \\
&\quad \land \mathrm{@T(PumpFail)}) \\
P_8 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Off}) \land \\
&\quad (p,n) \models ((\mathrm{MC{=}Operating}) \land \mathrm{@F(SwitchOn)} \land (\mathrm{PumpFail{=}false})) \\
P_9 &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{MC{=}Error}) \land \\
&\quad (p,n) \models ((\mathrm{MC{=}Operating}) \land \mathrm{@T(PumpFail)}) \\
P_{10} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \land \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \land \mathrm{@F(TooHigh)} \land (\mathrm{MC{=}Operating})) \\
P_{11} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \land \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \land \mathrm{@F(TooLow)} \land (\mathrm{MC{=}Operating})) \\
P_{12} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \land \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \land \mathrm{@T(PumpFail)} \land (\mathrm{MC{=}Operating})) \\
P_{13} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}false}) \land \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}true}) \land \mathrm{@T(PumpFail)} \land (\mathrm{MC{=}Off})) \\
P_{14} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}true}) \land \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}false}) \land \mathrm{@T(TooHigh)} \land (\mathrm{MC{=}Operating})) \\
P_{15} &= \exists n,\ \exists p \in \mathrm{pred}(n),\ n \models (\mathrm{PumpOn{=}true}) \land \\
&\quad (p,n) \models ((\mathrm{PumpOn{=}false}) \land \mathrm{@T(TooLow)} \land (\mathrm{MC{=}Operating}))
\end{aligned}
$$

Figure 4: ALT properties for SWLMS.

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

The syntax of Update annotations is described below:

Annotation Syntax	Meaning
Update A=true	A receives the value {true}.
Update A=false	A receives the value {false}.
Update A=Top	A receives the value corresponding to the union of all values of its type.
Update M=M1	Mode Class M receives the value {M1}.

Either syntax can be used when writing designs. Variables in Update annotations may be combined using the & (AND) operator, indicating that all variables receive their values at the same time. The syntax of Assert annotations is the same as that of Update annotations, except that variables may also be combined using the | (OR) operator, indicating that the programmer knows (or assumes) that at least one of the disjuncts is true.

We do not process statements other than annotations and control flow constructs since they do not reflect changes of values of requirements variables. To differentiate between statements and annotations, the latter start with @@.

The following design fragment shows sample uses of annotations:

```
ReadDeviceMonitor();     /* read value */
@@ Update PumpFail=Top;
if (pump_failed()) {   /* test value */
    @@ Assert PumpFail=true;
    @@ Update MC=Error;
}
else {
   @@ Assert PumpFail=false;
   ...
}
```

In this fragment, the function ReadDeviceMonitor() is called to determine the status of a pump. The Update annotation records the outcome of this call by setting the value of the requirement's variable PumpFail to Top. The function pump_failed() determines if the value read corresponds to failure of the pump. We assert that the Then clause will be executed only if the pump did fail (i.e., the value of PumpFail is true rather than Top). In this case, the system should transit to mode Error. In the Else branch, we assert that the value of PumpFail is false rather than Top.

Figure 5 shows a design for the SWLMS. Here we notice that the pump can be turned on only when the system is in mode Operating. So, the design has an inner WHILE loop, in which the system reads the water level and determines the state of the pump. The system exits the loop when the pump fails or when the switch is turned Off. Since the SWLMS has no error recovery, transitions to mode Error are done outside the main WHILE loop.

We can also verify the consistency of existing programs with their requirements by annotating source code with comments corresponding to changes and tests of values of requirements variables. We followed this approach to verify an implementation of the Water-Level Monitoring System (see Section 10). This approach raises the a problem of verifying the consistency of requirements with annotations rather than with the actual code. However, if annotations are done carefully, their consistency can raise our confidence about that between the source code and requirements. Annotations and source code may also "diverge" as modifications to the program are being made.

8 Creating the Abstraction

We construct a Design-Flow Graph (DFG) from annotations and control-flow information of the design and then propagate the state information throughout the DFG, in a manner similar to data-flow analysis[1]. The rest of this section describes the process we use to construct a set-based approximation of attainable values of requirements variables for each node of a DFG. The process consists of the following steps:

- Compute information generated by each annotation.
- Detail this information using environmental assumptions and check for violations of the assumptions.
- Propagate this information throughout the DFG.

The DFG is abstracted into a Finite-State Machine (FSM) which is then used to check system properties. Figure 6 gives a roadmap of the analysis process described in the remainder of this paper.

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

Figure 5: Design of SWLMS.

Definition 8.1 *A Control-Flow Graph of a program design PD (DFG) is a directed graph* $G = \langle V,\ E,\ V_0 \rangle$, *where*

V is a finite set of nodes corresponding to splits, joins and annotations of PD.
$E \subseteq V \times V$ is a set of directed edges, s.t. $(v_1,\ v_2) \in E$ iff v_2 can immediately follow v_1 in some execution sequence; and
$V_0 \in V$ is an entry node.

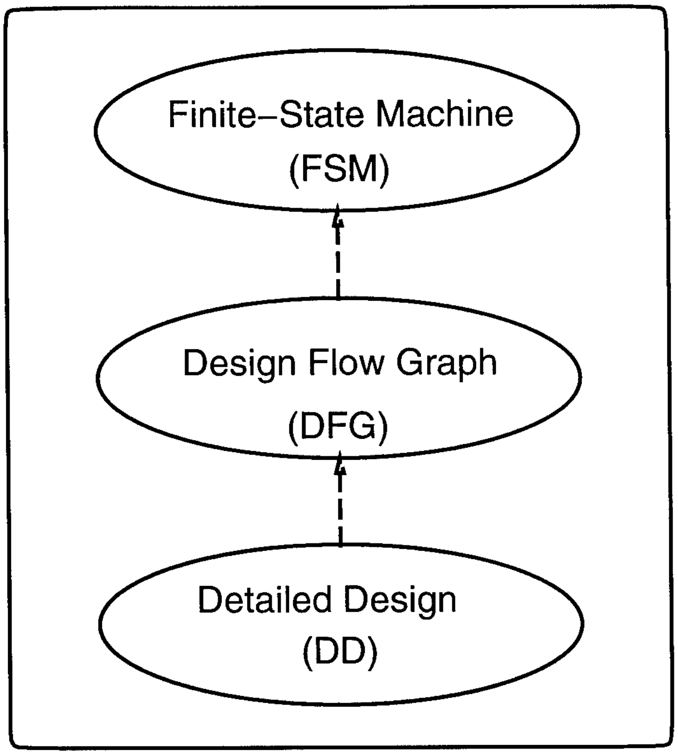


Figure 6: Analysis roadmap.

We interpret annotations in the design to create a *set-based approximation* of attainable values for each requirements' variable at each node of the DFG. This means that we are interested in only those variables which are specified in the requirements, and that each variable is associated with a set of values it may attain if the control reaches that node.

Definition 8.2 *A system state at node n of a design implementing an SCR specification $\mathcal{R}$ is a set of variable-value pairs $\{(r_j, v_{r_j}) \mid r_j \in R\}$, where*
v_{r_j} is a set of values associated with the variable r_j at the node n, and
$\forall\ r_j \in R$, $v_{r_j} \in 2^{T(r_j)}$ (v_{r_j} is a set consisting of values in the domain of r_j).

We require that there is exactly one variable-value pair for each variable in the requirements, and thus for a system state s we can define a function $v(r_j, s)$ which returns the value of r_j in s:

$$v(r_j, s) \equiv \mu \text{ s.t. } (r_j, \mu) \in s$$

In addition, we define a function $\text{repl}(r_j, \mu, s)$ to replace the current value of r_j by μ in s:

$$\text{repl}(r_j, \mu, s) \equiv (s - \{(r_j, v(r_j, s))\}) \cup \{(r_j, \mu)\}$$

For controlled and monitored variables ($r_j \in M \cup C$), these values are $\{\}$, $\{\text{true}\}$, $\{\text{false}\}$ and $\{\text{true, false}\}$, which form a $\cup$-lattice on set-inclusion. These values have the following meaning:

{ } On any path leading to this node, the value of the variable is unknown.

{true} On all paths leading to this node, the variable is true.

{false} On all paths leading to this node, the variable is false.

{true, false} On some path the variable is true, and on some other it is false.

For mode classes ($r_j \in MD$), the values also form a ⊔-lattice on set-inclusion.

Operations on system states are: "⊔" (union), "⊓" (intersection), "=" (equality), "⊐" (superset), "−" (difference) and "⊒" (superset or equal to). We also define a special system state EMPTY:

$$s = \text{EMPTY} \equiv \forall r_j \in R,\ v(r_j, s) = \{\}$$

8.1 Constant Propagation

Our computation of system states at each node of the DFG is similar to that of *constant propagation* - a compiler technique whose goal is to discover values that are constant for all possible executions of a program and to propagate these constant values as far forward through the program as possible[34, 1]. For every node n in the graph, we keep the following sets of variable-value pairs:

gen(n) Pairs with values generated in the annotation at node n.

known(n) Pairs with values assumed by the designer at node n.

in(n) Pairs that may exist when control reaches n.

out(n) Pairs that may exist when control leaves n.

8.2 Computing gen and known Sets

gen and **known** sets for each node are computed using the following rules:

- For nodes corresponding to Update and Initial annotations, **gen** sets contain variable-value pairs with the specified value, and with empty set values for all other variables.
- For nodes corresponding to Assert annotations, **known** sets contain variable-value pairs with the specified value, and with empty set values for all other variables.
- For all other nodes, **gen** and **known** sets are EMPTY.

An Assert annotation may contain a disjunction of several clauses. For these nodes, **known** sets are lists of several system states, one for each clause. Operations on system states are trivially extended to handle sets of system states. In the examples below, we omit variable-value pairs in which the value is {}. For an annotation @@Update MC=Operating & PumpFail=true,

$$\textbf{gen} = \{(\text{MC}, \{\text{Operating}\}), (\text{PumpFail}, \{\text{true}\})\}$$

For an annotation @@Assert SwitchOn=false | PumpFail=true,

$$\textbf{known} = \{\{(\text{SwitchOn}, \{\text{false}\})\}, \{(\text{PumpFail}, \{\text{true}\})\}\}$$

Once the initial **gen** and **known** sets are constructed, we use environmental assumption information (i.e., the $\mathcal{E}$ part of the SCR requirements) to make these sets more precise. For example, environmental assumption TooLow −>> ¬TooHigh is used to add information to the **known** set for an annotation @@Assert TooLow=true, resulting in

$$\textbf{known} = \{(\text{TooLow}, \{\text{true}\}), (\text{TooHigh}, \{\text{false}\})\}$$

Environmental assumptions can also be used to make sure that contradictory variable settings (like @@Assert TooLow=true and TooHigh=true) are not made. Errors are considered violations of the ENV property - "environmental assumptions are preserved". Details of this processing are presented in [13].

8.3 Propagating Information

We initialize in and out sets of every node to EMPTY. Then we propagate information throughout the DFG until a least fixed point is reached. A *meet* operator for combining information coming into the node is $\sqcup$, so

$$\texttt{in}(n) = \sqcup_{\forall\, k,\, \text{s.t.}\, (k,n)\, \in\, E}\texttt{out}(k)$$

A set F of *transfer functions* describing the transformation between in and out sets at each node, is defined as follows:

Annotation at node n	Transfer function
Initial	out(n) = gen(n)
Update	out(n) = repl(in(n), gen(n))
Assert (single disjunct)	out(n) = in(n) ⊓ known(n)
none	out(n) = in(n)

If the known set for a node n containing an Assert annotation consists of several disjuncts, i.e., known(n) = $\{d_1, d_2, ..., d_k\}$, then

$$\texttt{out}(n) = \sqcup_{1 \leq i \leq k}(\texttt{in}(n) \sqcap d_i)$$

Our framework is strictly monotonic, i.e., in and out sets on an iteration of our algorithm have more values (or at least as many) for each variable as in and out sets on the previous iteration. Since all variables in R have a finite number of abstract values, our system states do not have an infinite increasing chain of values, and the fixed point can be achieved in a finite number of steps. We assume that all variables are initialized by the Initial annotation. A node n is considered *unreachable* if

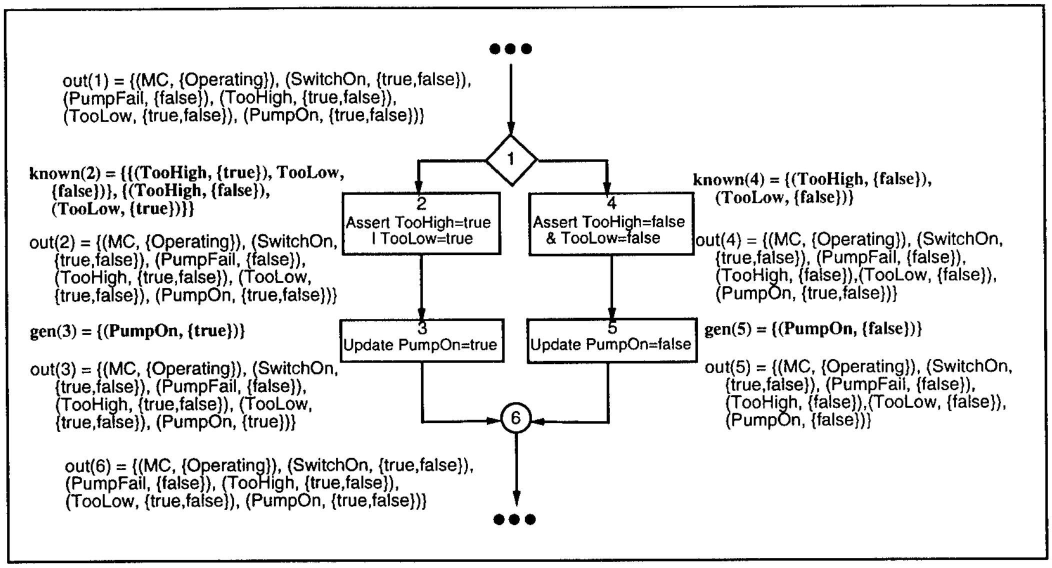
$$\exists r_j \in R \text{ s.t. } v(r_j, \texttt{out}(n)) = \{\}.$$

8.4 Example - DFG of SWLMS

Consider computing DFG for the design of SWLMS (Figure 5). Figure 7 shows a fragment of this DFG corresponding to lines 38-45 of the SWLMS design. gen and known sets computed at each node are shown in bold font in this figure; variables with values {} omitted. The Assert on the left branch (node 2) generates information that either TooLow or TooHigh is true. If TooLow is true, then, via environmental restrictions, TooHigh is false, and vice versa. The resulting known set consists of two disjuncts, one for each possibility. The Update on that branch (node 3) generates the value {true} for PumpOn. The Assert on the right branch (node 4) generates the value {false} for TooLow and TooHigh, and the following Update (node 5) generates the value {false} for PumpOn. We did not include in sets in Figure 7 since the in sets for nodes 2-5 are equal to the out sets of their predecessor nodes; and the in sets for nodes 1 and 6 are equal to their out sets. To compute out sets, CORD uses the multiple disjunct transfer function for Assert nodes. Each of disjuncts of known(2) is intersected with in(1), and the union of the results is computed. So, the values for TooHigh and TooLow in out(2) become {true, false}. The Update annotation PumpOn=true (node 3) changes the value of PumpOn in out(3) to {true}. The right branch is processed similarly. At the join, we compute the union the possible values for each variable in the out sets of the predecessor nodes (nodes 3 and 5).

8.5 Constructing a Finite-State Machine

Our DFG contains nodes which do not reflect state changes (e.g., decision nodes, joins and nodes containing Assert annotations) and possibly some unreachable nodes. We construct a Finite-State Machine (FSM) which contains just reachable nodes representing state changes. The resulting FSM is used as a model for verifying system properties. In a DFG ($G = \langle V, E, V_0 \rangle$), let $U \subseteq V$ and $I \subseteq V$ be disjoint sets of nodes containing Update and Initial annotations, respectively.



out(1) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
1
known(2) = {{(TooHigh, {true}), TooLow, {false})}, {(TooHigh, {false}), (TooLow, {true})}}
2
Assert TooHigh=true I TooLow=true
4
Assert TooHigh=false & TooLow=false
known(4) = {(TooHigh, {false}), (TooLow, {false})}
out(2) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
out(4) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {false}),(TooLow, {false}), (PumpOn, {true,false})}
gen(3) = {(PumpOn, {true})}
3
Update PumpOn=true
5
Update PumpOn=false
gen(5) = {(PumpOn, {false})}
out(3) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true})}
out(5) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {false}),(TooLow, {false}), (PumpOn, {false})}
6
out(6) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}

Figure 7: A fragment of DFG of SWLMS.

Definition 8.3 *A Finite-State Machine (FSM) over a program design PD is a structure $M = \langle A, S, L, N, s_0 \rangle$, where*

A is a set of labels;
$S = U \cup I$ is a finite set of nodes;
L: $S \rightarrow A$ is a function associating each node with a label;
$N \subseteq S \times S$ is a transition relation.
N is obtained by connecting nodes of S s.t. there is an Update-clear path between them in DFG; and
$s_0 \in S$ is an entry node.

To build a FSM from a DFG we remove all nodes except those corresponding to Initial and Update annotations and connect all predecessors of a removed node to the node's successors. For every node, we check an implicit property (REACH):

$$\forall n \in S (\forall r_j \in R,\ v(r_j, out(n)) \neq \{\})$$

If this property is violated in a node, an error is reported and the node is removed from the FSM.

Figure 8 shows the FSM created for the SWLMS design in Figure 5, depicting **out** and **gen** sets for every node. The number of each node of the FSM indicates the line of the design at which the corresponding Update or Initial annotation can be found. For example, nodes 40 and 44 correspond @@Update PumpOn=true and @@Update PumpOn=false, respectively. The algorithm for computing **out** sets described in Section 8.3 ensures that the effects of Assert annotations are preserved in system states, even though Assert nodes themselves are removed.

All states in the resulting FSM correspond to state changes. This FSM is then used to verify properties generated from requirements.

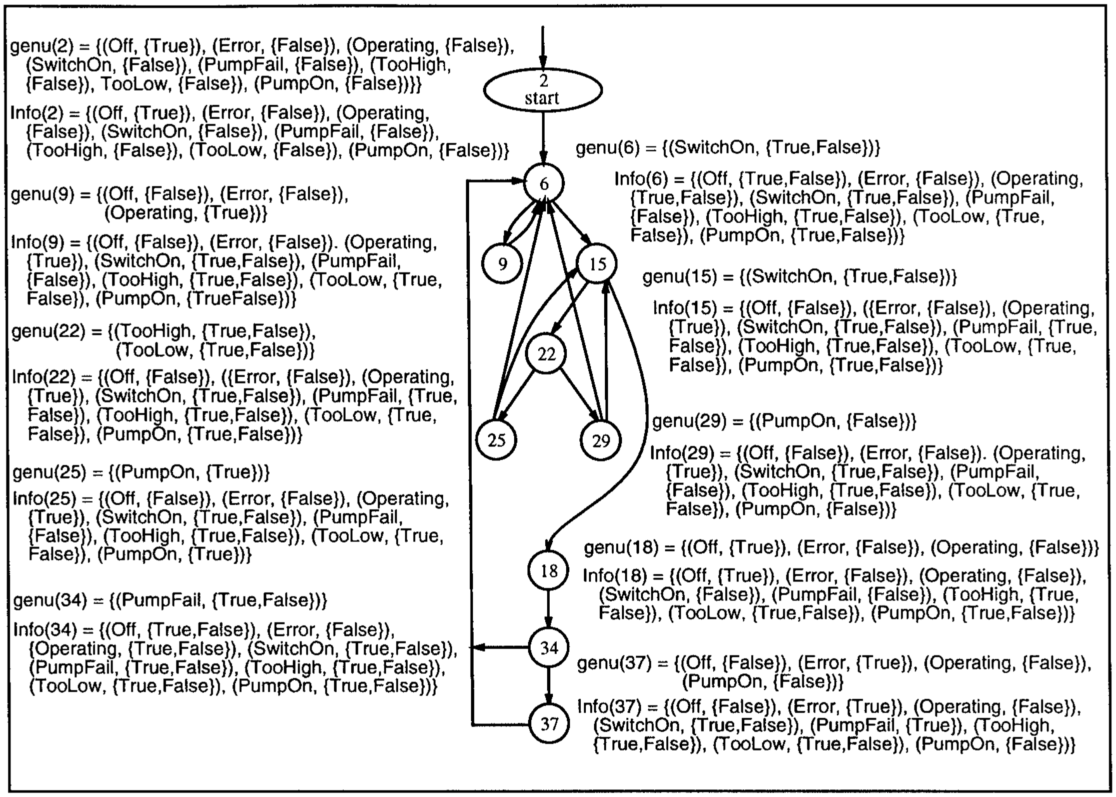

genu(2) = {(Off, {True}), (Error, {False}), (Operating, {False}), (SwitchOn, {False}), (PumpFail, {False}), (TooHigh, {False}), TooLow, {False}), (PumpOn, {False})}}
Info(2) = {(Off, {True}), (Error, {False}), (Operating, {False}), (SwitchOn, {False}), (PumpFail, {False}), (TooHigh, {False}), (TooLow, {False}), (PumpOn, {False})}
genu(9) = {(Off, {False}), (Error, {False}), (Operating, {True})}
Info(9) = {(Off, {False}), (Error, {False}). (Operating, {True}), (SwitchOn, {True,False}), (PumpFail, {False}), (TooHigh, {True,False}), (TooLow, {True, False}), (PumpOn, {TrueFalse})}
genu(22) = {(TooHigh, {True,False}), (TooLow, {True,False})}
Info(22) = {(Off, {False}), ({Error, {False}), (Operating, {True}), (SwitchOn, {True,False}), (PumpFail, {True, False}), (TooHigh, {True,False}), (TooLow, {True, False}), (PumpOn, {True,False})}
genu(25) = {(PumpOn, {True})}
Info(25) = {(Off, {False}), (Error, {False}), (Operating, {True}), (SwitchOn, {True,False}), (PumpFail, {False}), (TooHigh, {True,False}), (TooLow, {True, False}), (PumpOn, {True})}
genu(34) = {(PumpFail, {True,False})}
Info(34) = {(Off, {True,False}), (Error, {False}), {Operating, {True,False}), (SwitchOn, {True,False}), (PumpFail, {True,False}), (TooHigh, {True,False}), (TooLow, {True,False}), (PumpOn, {True,False})}
2
start
6
9
15
22
25
29
18
34
37
genu(6) = {(SwitchOn, {True,False})}
Info(6) = {(Off, {True,False}), (Error, {False}), (Operating, {True,False}), (SwitchOn, {True,False}), (PumpFail, {False}), (TooHigh, {True,False}), (TooLow, {True, False}), (PumpOn, {True,False})}
genu(15) = {(SwitchOn, {True,False})}
Info(15) = {(Off, {False}), ({Error, {False}), (Operating, {True}), (SwitchOn, {True,False}), (PumpFail, {True, False}), (TooHigh, {True,False}), (TooLow, {True, False}), (PumpOn, {True,False})}
genu(29) = {(PumpOn, {False})}
Info(29) = {(Off, {False}), (Error, {False}). (Operating, {True}), (SwitchOn, {True,False}), (PumpFail, {False}), (TooHigh, {True,False}), (TooLow, {True, False}), (PumpOn, {False})}
genu(18) = {(Off, {True}), (Error, {False}), (Operating, {False})}
Info(18) = {(Off, {True}), (Error, {False}), (Operating, {False}), (SwitchOn, {False}), (PumpFail, {False}), (TooHigh, {True, False}), (TooLow, {True,False}), (PumpOn, {True,False})}
genu(37) = {(Off, {False}), (Error, {True}), (Operating, {False}), (PumpOn, {False})}
Info(37) = {(Off, {False}), (Error, {True}), (Operating, {False}), (SwitchOn, {True,False}), (PumpFail, {True}), (TooHigh, {True,False}), (TooLow, {True,False}), (PumpOn, {False})}

Figure 8: Finite-state abstraction.

9 Verifying Properties

Our method for constructing finite-state abstractions produces sets of values for each program variable. Model checkers process states whose variables have scalar values. Transforming our FSM to correspond to an acceptable input for an existing model-checker would have resulted in an exponential increase in the number of nodes in the FSM. So, we developed our own technique to verify properties. We cannot verify properties exactly, i.e., claim that a property is violated if and only if we find a violation. So, we have carefully designed our verification algorithms so that the results can be correctly interpreted.

9.1 Optimism and Pessimism

We divide properties into two categories: those checked *optimistically* and *pessimistically.* A property is checked pessimistically if all of its violations in the design are detected, but the analysis incorrectly identifies violations at points in the design at which the property actually holds. A property is checked optimistically if all detected violations are present in the design, but the analysis is unable to find all violations of the property.

Most of properties that we are interested in verifying involve state transitions which occur only in response to events. Thus, we need to develop techniques to compute events and transitions for both optimistic and pessimistic analysis. Our techniques overestimate the number of transitions in the design, i.e., if a transition is present in the design, we compute it, but some of the computed transitions might not be present in the design. Overestimation of the number of transitions does not invalidate the verification of automatically-generated properties. For OLT properties, we want to check if all transitions in the design are present in the requirements, and overestimating the transitions might cause the tool to report some false negatives, which is a correct treatment of pessimistic analysis. For ALT properties, we want to check if all transitions in the requirements are present in the design, and overestimating the transitions might cause the tool to report some false positives, which is a correct treatment of optimistic analysis. Table 3 summarizes our analysis. For example, when a transition in the requirements is not implemented in the design and is not computed by our tool (row 4), then the tool reports a violation of the corresponding ALT property and does not report a violation of an OLT property. Our computation finds all transitions present in the design. Thus the cases described by rows 3 and 7 in Table 3 cannot occur, so the analysis results are not defined for them.

Row	Transition exists			Analysis reports	
	Requirements	Computation	Design	ALT properties	OLT properties
1	T	T	T	no violation	no violation
2	T	T	F	false positive	no violation
3	T	F	T	–	–
4	T	F	F	violation	no violation
5	F	T	T	no violation	violation
6	F	T	F	no violation	false negative
7	F	F	T	–	–
8	F	F	F	no violation	no violation

Table 3: Results of analysis.

9.2 Mapping Between Events and Transitions in Requirements and Our Model

The semantics of an SCR event, given in Section 2.1, indicates that some conditions need to hold on edges. However, our FSM consists of states, with `in`, `gen` and `out` sets. Formally, we say that a formula f *holds in a state* s if f holds in $\texttt{out}(s)$, i.e., $\texttt{out}(s) \models f$. We say that f is *generated in a state* s if $\texttt{gen}(s) \models f$. Finally, we say that f holds on an edge between nodes n and s if $(\texttt{out}(n) \sqcap \texttt{in}(s)) \models f$. By construction of the FSM, if a formula holds on exit of the node but does not hold on entrance, then it has been generated at this node, i.e. for a node n,

$$(\texttt{out}(n) \models f) \wedge (\texttt{in}(n) \not\models f) \rightarrow (\texttt{gen}(n) \models f)$$

We also note that an event occurs at a node if a value of some variable on an edge entering the node is different from its value on an edge leaving the node. Thus, the variable is changed in the node's `gen` set.

9.3 Algorithm to Compute Transitions

To determine if an event occurred at a node s, we use information generated at each predecessor node n and check it against $\texttt{in}(s)$ and the `out` sets of the predecessors of n. The algorithm to compute transitions leading to a node s is shown in Figure 9. The output of this algorithm is a set of transitions $\{(n, \texttt{to}, \texttt{from}, \texttt{trigger}, \texttt{when})\}$, where

- n is a predecessor of s;
- `to` and `from` are in the form $r_j = v'_{r_j}$ and $r_j = v_{r_j}$, respectively, indicating the change of value of r_j from v_{r_j} to v'_{r_j} for a transition between n and s;

```
Inputs:    Finite state machine (FSM) and
           node $s$ for which to compute the transitions.
Outputs:   A set of transitions to node $s$. Each transition is in the form (n, to, from,
           trigger, when), where $n \in$ pred($s$) and to $\neq$ from.
Algorithm:
           all_transitions = {}
           For each $n \in$ pred(s) {
              /* triggs is a logical expression $\bigwedge_{r_j \in R-C} \bigvee_k exp_{j,k}$ */
              triggs = true
              when = out($n$) $\sqcap$ in($s$)
              For each $r_j \in R - C$, where $v(r_j, \text{gen}(n)) \neq \{\}$
                 /* Controlled variables cannot be part of the triggering condition */
                 triggs = triggs $\wedge$ events($r_j$, $\bigcup_{\forall p \in \text{pred}(n)}$out($p$), gen($n$) $\sqcap$ when)
              /* Rewrite triggs to be in form $\bigvee_k \bigwedge_{r_j \in R} exp_{k,j}$ */
              For each of the $k$ disjuncts $d_k$ of triggs {
                 Evaluate $d_k$ interpreting none($r_j$) as true.
                 If $d_k$ = true, then replace $d_k$ with special value NONE
                 trigger[$k$] = $d_k$
                 when[$k$] = when
                 /* remove trigger values from WHEN condition */
                 For each $r_j \in R - C$
                    If partof($r_j$, $d_k$)
                       when[$k$] = repl($r_j$, when[$k$], {})
                 /* compute to and from values for mode class and controlled variables */
                 For each $r_j \in R - M$ s.t. $v(r_j, \text{gen}(s)) \neq \{\}$ {
                    /* remove values changed between source and destination nodes */
                    when[$k$] = repl($r_j$, when[$k$], {})
                    For each value$_l$ $\in v(r_j, \text{gen}(s))$ {
                       to = ($r_j$ = value$_l$) /* final value of $r_j$ */
                       For each value$_m$ $\in v(r_j, \text{out}(n) \sqcap \text{in}(s))$ {
                          from = ($r_j$ = value$_m$) /* starting value of $r_j$ */
                          If value$_l$ $\neq$ value$_m$ /* otherwise there is no transition */
                             all_transitions = all_transitions $\cup$
                                      ($n$, to, from, trigger[$k$], when[$k$])
                       }
                    }
                 }
              }
           }
```

Figure 9: Algorithm for computing transitions.

- trigger is logical expression - a disjunction of simultaneously occurring triggering conditions, or NONE, if no events can occur; and
- when is a set of variable-value pairs indicating the When condition for this transition. We assume that controlled variables cannot be part of Triggering conditions, and that variables which are part of a Triggering condition cannot be part of a corresponding When condition.

The function events(r_j, s_1, s_2) checks values of a variable r_j in system states s_1 and s_2 to determine if an event involving r_j has occurred, returning a disjunction of @T(r_j), @F(r_j) or none(r_j). The last value indicates that no event involving r_j has occurred between the two nodes. We use "none" as a symbolic constant to indicate that some variable did not change its value. When the second system state refers to the initial node of the FSM, i.e., the node containing the Initial annotation, we assume that all values assigned to variables in this annotation signify events. For the initial node and a boolean variable r_j, events(r_j, $s_1 = \{\}$, s_2) is defined as

$v(r_j, s)$	events(r_j, {}, s)
{true}	@T(r_j)
{false}	@F(r_j)
{true,false}	@T(r_j) $\vee$ @F(r_j)

If the second system state refers to a non-initial node, then, for a boolean r_j, events(r_j, s_1, s_2) is defined as follows:

$v(r_j, s_1) \setminus v(r_j, s_2)$	{true}	{false}	{true,false}
{true}	none(r_j)	@F(r_j)	@T(r_j) $\vee$ none(r_j)
{false}	@T(r_j)	none(r_j)	@F(r_j) $\vee$ none(r_j)
{true,false}	@T(r_j) $\vee$ none(r_j)	@F(r_j) $\vee$ none(r_j)	@T(r_j) $\vee$ @F(r_j) $\vee$ none(r_j)

For a mode class mc, events(mc, s_1, s_2) returns a disjunction of all possible event *combinations* which could occur between the two nodes. A combination of events is a conjunction of events which occur simultaneously. For example, if $v(mc, s_1) = \{\text{m1}, \text{m2}\}$ and $v(mc, s_2) = \{\text{m2}, \text{m3}\}$, then

$$\begin{aligned}\text{events}(mc, s_1, s_2) \quad = \quad & ((\text{@T(mc=m2)} \wedge \text{@F(mc=m1)}) \;\vee\; \text{none(mc)} \;\vee \\ & (\text{@T(mc=m3)} \wedge \text{@F(mc=m1)}) \;\vee \\ & (\text{@T(mc=m3)} \wedge \text{@F(mc=m2)}))\end{aligned}$$

Finally, a function partof(r_j, trigger) returns true if trigger contains a conjunct corresponding to r_j and false otherwise.

In our SWLMS example, consider calculating the event which causes MC to be set to Error at node 50 of the FSM in Figure 8. Figure 10 shows a fragment of this FSM. when is out(5) $\sqcap$ in(50), namely,

{(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}),
(MC, {Operating,Off}), (PumpOn, {true,false}), (PumpFail, {true})}.

gen(5) (node 5 is a predecessor of 50) is (PumpFail, {true,false}).

$$\text{triggs} = \text{events(PumpFail}, \bigcup_{\forall p \in \{2,15,22,32\}} \text{out}(p),\ \text{gen}(5) \sqcap \text{when})$$

Since PumpFail is {false} in nodes 2, 15 and 32, and {true,false} in node 22,

$$\bigcup_{\forall p \in \{2,15,22,32\}} \text{out}(p) = \{(\text{PumpFail}, \{\text{true,false}\})\}.$$

gen(5) $\sqcap$ when = {(PumpFail, {true})}, which indicates that the designer assumed that PumpFail is true in node 50. The events function is called for PumpFail and yields @T(PumpFail) $\vee$ none(PumpFail). The second disjunct corresponds to paths on which PumpFail becomes true on line 22 and is again set to true on line 5. The Triggering condition has two disjuncts, @T(PumpFail) and NONE. For the event triggered by @T(PumpFail), we replace PumpFail's and MC's values with {} in when[k] (since MC is in gen(50)). On the first iteration of the loop, to is (MC=Error) and from is (MC=Off); on the next iteration, from becomes (MC=Operating). The result is four transitions, I_1-I_4, for each combination of triggers and starting values of MC. These transitions are shown in Figure 11.

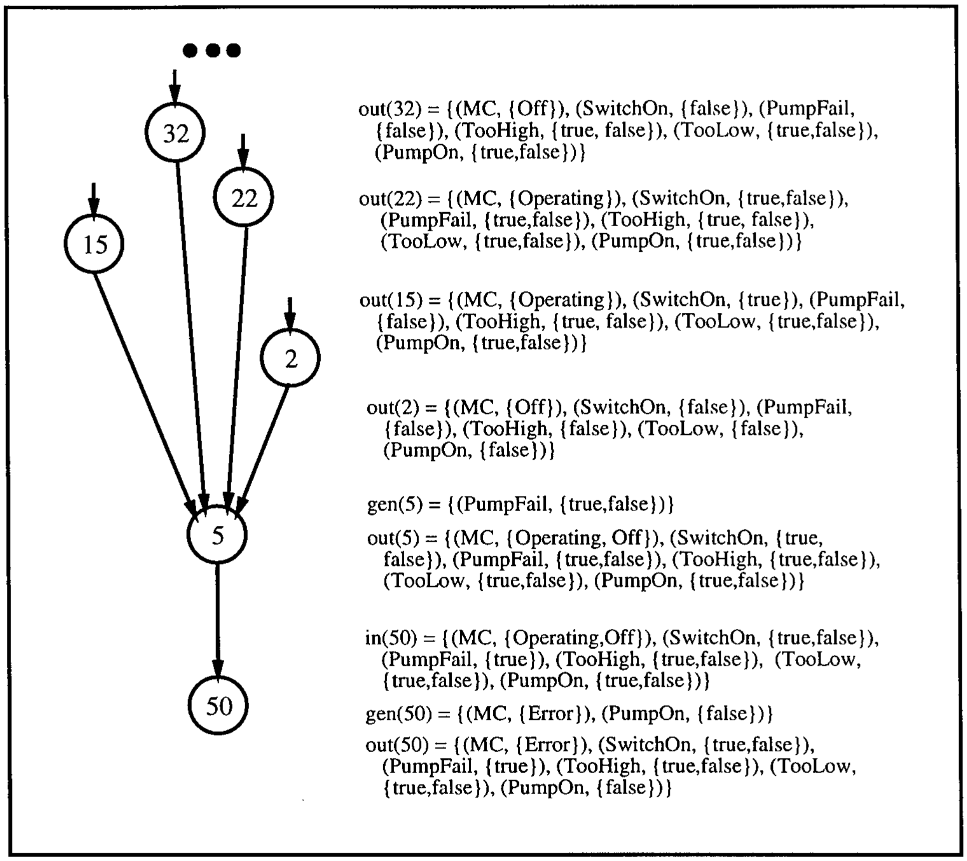

32
22
15
2
5
50
out(32) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}
out(22) = {(MC, {Operating}), (SwitchOn, {true,false}), (PumpFail, {true,false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}
out(15) = {(MC, {Operating}), (SwitchOn, {true}), (PumpFail, {false}), (TooHigh, {true, false}), (TooLow, {true,false}), (PumpOn, {true,false})}
out(2) = {(MC, {Off}), (SwitchOn, {false}), (PumpFail, {false}), (TooHigh, {false}), (TooLow, {false}), (PumpOn, {false})}
gen(5) = {(PumpFail, {true,false})}
out(5) = {(MC, {Operating, Off}), (SwitchOn, {true, false}), (PumpFail, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
in(50) = {(MC, {Operating,Off}), (SwitchOn, {true,false}), (PumpFail, {true}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false})}
gen(50) = {(MC, {Error}), (PumpOn, {false})}
out(50) = {(MC, {Error}), (SwitchOn, {true,false}), (PumpFail, {true}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {false})}

Figure 10: Calculating transitions for SWLMS.

9.4 Checking Automatically-Generated Properties

Once the FSM has been created, ALT and OLT properties are checked in a single traversal of the FSM. Proofs of correctness of algorithms shown below appear in [13].

Verification of OLT Properties

OLT properties have the general form $P_i = \forall n,\ (n \models (r = v_{new})) \rightarrow (\forall p \in \text{pred}(n),$ $(p,n) \models (r = v_{new}) \vee \bigvee_j (p,n) \models ((r = v_{j,old}) \wedge \texttt{trcond}_j \wedge \texttt{whcond}_j))$, where v_{new} and v_{old} are the new and the old values, respectively, for the mode class or controlled variable r. $\texttt{trcond}_j$ and $\texttt{whcond}_j$ are conjuncts representing the Triggering and the When conditions of the jth row of the table entry corresponding to a change of r_j's value from $v_{j,old}$ to v_{new}. Since OLT properties are to be verified pessimistically, we want to ensure that if an OLT property is violated in the design, our analysis catches the violation. OLT violations are reported as soon as they are discovered. Since we compute a number of transitions for a single Update annotation, we can report a number of OLT violations for a given line in the design, as outlined by the algorithm in Figure 12.

We take advantage of the fact that all properties have been generated from SCR tables, and thus

I_1:
@T(PumpFail) Transition from MC=Off to MC=Error:
WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false}),
(PumpFail, {}), (MC, {})}]

I_2:
NONE Transition from MC=Off to MC=Error:
WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false}),
(PumpFail, {true}), (MC, {})}]

I_3:
@T(PumpFail) Transition from MC=Operating to MC=Error:
WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false})
(PumpFail, {}), (MC, {})}]

I_4:
NONE Transition from MC=Off to MC=Error:
WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}),
(TooLow, {true,false}), (PumpOn, {true,false}),
(PumpFail, {true}), (MC, {})}]

Figure 11: Transitions discovered for node 50.

$\texttt{trcond}_j$ consists of a conjunction of one or more simple Triggering conditions (e.g., @T(a)). Our algorithm to compute transitions also results in a conjunction of simple Triggering conditions. To check that $\texttt{trigger} \rightarrow \texttt{trcond}_j$, we check that each conjunct in $\texttt{trcond}_j$ is present in $\texttt{trigger}$.

Before checking that $\texttt{when} \sqsubseteq \texttt{whcond}_j$, we first need to represent $\texttt{whcond}_j$ as a set of variable-value pairs. For example, PumpOn = true is treated as (PumpOn, {true}) and ¬(MC = Operating) means that MC can be either Off or Error and is treated as (MC, {Off, Error}). If a variable $r_k \in R$ is not part of $\texttt{whcond}_j$, then it was specified as a "don't care" condition in the tables. The value for this variable is considered to be a set of all of its attainable values, i.e., it is treated as $(r_k, T(r_k))$. For example, if a boolean variable TooHigh is a "don't care" for some transition, then the corresponding $\texttt{whcond}_j$ set contains (TooHigh, {true,false}). One of the OLT properties for SWLMS is

$$\begin{aligned} P_3 \;=\; & \forall n,\ (n \models (\text{MC=Error}) \rightarrow (\forall p \in \text{pred}(n),\ (p,n) \models (\text{MC=Error}) \ \vee \\ & (p,n) \models \text{@T(PumpFail)} \wedge (\text{MC=Operating}) \ \vee \\ & (p,n) \models \text{@T(PumpFail)} \wedge (\text{MC=Off})) \end{aligned}$$

In this property,

$$\begin{array}{lcl} r & = & \text{MC} \\ v_{new} & = & \text{Error} \\ v_{1,old} & = & \text{Operating} \\ v_{2,old} & = & \text{Off} \\ \texttt{trcond}_1 \text{ and } \texttt{trcond}_2 & = & \text{@T(PumpFail)} \\ \texttt{whcond}_1 \text{ and } \texttt{whcond}_2 & = & \{\text{(SwitchOn, \{true,false\}), (TooHigh, \{true,false\}),} \\ & & \text{(TooLow, \{true,false\}), (PumpOn, \{true,false\})}\} \end{array}$$

For all nodes other than 50, MC≠Error, so P_3 holds vacuously. The transitions generated for node 50 are I_1-I_4, as shown in Figure 11. For I_1, the $\texttt{from}$ part is $(r = v_{2,old})$, $\texttt{trigger} \rightarrow \texttt{trcond}_2$ and $\texttt{when} \sqsubseteq \texttt{whcond}_2$, so, no errors are reported. The case for I_3 is similar: the $\texttt{from}$ part is $(r = v_{1,old})$, $\texttt{trigger} \rightarrow \texttt{trcond}_1$ and $\texttt{when} \sqsubseteq \texttt{whcond}_1$. $\texttt{trigger}$s for transitions I_2 and I_4 are NONE, indicating that no Triggering condition was found, so CORD reports an error message:

```
Inputs:    A set $\{P_i\}$ of OLT properties, where
           $P_i = \forall n,\ (n \models (r = v_{new})) \rightarrow (\forall p \in \text{pred}(n), (p, n) \models (r = v_{new}) \vee$
           $\bigvee_j (p, n) \models ((r = v_{j,old}) \wedge \texttt{trcond}_j \wedge \texttt{whcond}_j)),$
           Node $n$ in finite state machine FSM
Outputs:   Error messages indicating violations of $P_i$'s at $n$.
Algorithm:
           Compute a set of transitions for node $n$.
           For each transition ($p$, to, from, trigger, when) s.t. to $\neq$ from
              If trigger is equal to NONE
                 Report error "no triggering conditions"
              Else {
                 For each property $P_i$ s.t. to = $(r = v_{new})${
                    found = false
                    For each disjunct $P_{i,j}$
                       If from = $(r = v_{j,old})$ AND
                          trigger $\rightarrow$ $\texttt{trcond}_j$ AND
                          when $\sqsubseteq$ $\texttt{whcond}_j$
                       Then found = true
                    If not found
                       report a violation of $P_i$ at node $n$.
                 }
              }
```

Figure 12: Algorithm for verifying OLT properties.

```
Error on line 50 of function main in mode class MC:
   no triggering condition for transition
   from mode(s) {Operating,Off} to mode(s) {Error}
```

Verification of ALT Properties

All ALT properties have the general form $P_i = \exists n,\ \exists p \in \text{pred}(n),\ n \models (r = v_{new}) \wedge (p, n) \models ((r = v_{old}) \wedge \texttt{trcond} \wedge \texttt{whcond})$, where v_{old} and v_{new} are the new and the old values, respectively, for the mode class or controlled variable r. **trcond** and **whcond** are conjuncts representing the Triggering and the When conditions for this transition. ALT properties are to be verified optimistically, so we want to ensure that if the analysis finds a violation of an ALT property, this property does not hold in the design. We might not report all unimplemented transitions. Once transitions are computed for a given node, we look through the list of ALT properties and mark those which are satisfied by this transition. Any properties remaining unmarked at the end of analysis are reported as errors. An algorithm to check ALT properties is outlined in Figure 13. For ALT properties, we translate "don't care" conditions in **whcond** to empty sets, so that **whcond** $\sqsubseteq$ **when** returns true if the computed When condition contains at least the variable-value pairs specified in P_i's **whcond**. Our model of SCR guarantees that there are no transitions in which the source and the destination are the same, i.e. for each P_i, $v_{new} \neq v_{old}$.

P_7 is an ALT property for SWLMS:

$P_7 \ = \ \exists n,\ \exists p \in \text{pred}(n),\ n \models (\text{MC=Error}) \wedge (p, n) \models ((\text{MC=Off}) \wedge \text{@T(PumpFail)})$

In this property,

```
Inputs:     A set $\{P_i\}$ of ALT properties, where
            $P_i = \exists n,\ \exists p \in \text{pred}(n),\ n \models (r = v_{new}) \wedge$
            $(p, n) \models ((r = v_{old}) \wedge \texttt{trcond} \wedge \texttt{whcond})$
            Finite state machine
Outputs:    Error messages indicating violations of $P_i$s at $n$.
Algorithm:
            Unmark all ALT properties
            For each node $n$ reachable from $s_0$ in depth-first order
               Compute a set of transitions for $n$.
               For each transition ($p$, to, from, trigger, when)
                  If there is an unmarked property $P_i$ s.t.
                     $(r = v_{new})$ = to AND
                     $(r = v_{old})$ = from AND
                     trcond $\rightharpoonup$ trigger AND
                     whcond $\sqsubseteq$ when
                  Then mark $P_i$
            Report all unmarked ALT properties
```

Figure 13: Algorithm for verifying ALT properties.

$$\begin{aligned}
r &= \text{MC} \\
v_{old} &= \text{Off} \\
v_{new} &= \text{Error} \\
\texttt{trcond} &= \text{@T(PumpFail)} \\
\texttt{whcond} &= \{(\text{SwitchOn}, \{\}), (\text{TooHigh}, \{\}), (\text{TooLow}, \{\}), (\text{PumpOn}, \{\})\}
\end{aligned}$$

Transitions generated for node 50, which is reachable from the start state, enable the algorithm to mark P_7 as satisfied: the from part of I_1 is MC=Off, the to part is MC=Error, trcond $\rightharpoonup$ trigger and whcond $\sqsubseteq$ when.

10 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[33]. 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.

10.1 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 4.

Mode Class	Mode	Meaning
Normal	Operating	The system is running properly.
	Shutdown	The water level is out of range and the system will be shutdown 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 4: 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 5 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 5: 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 6.

Controlled variables are set to trigger alarms and to display the water level to the operator. Table 7 shows the settings for the controlled variable LowWindowOn which represents the annunciation window labeled "Water Level Low." When LowWindowOn is true, the annunciator 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).

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

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 6: Original mode transition table for mode class Normal.

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

Initial: True

Table 7: Event table for controlled variable LowWindowOn.

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).

10.2 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 8 is a mode transition table with the environmental assumption information added.

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

1. *AG*((SelfTestPressed500 ∧ ~Test) → *AX*(Test))
2. *AG*(Standby —~SelfTestPressed500)
3. *AG*(Operating — (~SelfTestPressed500 ∧ (WithinLimits ∨ SelfTestPressed)))
4. *AG*(Shutdown — (~SelfTestPressed500 ∧ ((~InsideHysR ∧ ~ShutdownLockTime200) ∨ ~SelfTestPressed)))

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 8: Detailed mode transition table for mode class Normal.

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 ShutdownLockTime200 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

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

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 9: Corrected mode transition table for mode class Normal.

10.3 Design Analysis

From the corrected requirements in Table 9 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. 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 300 lines long, with 32 Update annotations corresponding to monitored variables, 31 Update annotations corresponding to mode class and controlled variables, and 56 Assert annotations. Of the 54 functions in the original program, only eight had state changes and thus were included into the design.

Property Type	Messages	Violations
REACH property	10	
OLT properties for mode classes	15	8
OLT properties for controlled variables	37	12
No events found	15	
ALT properties for mode classes	13	
ALT properties for controlled variables	21	

Table 10: Results of analyzing the WLMS.

After we eliminated annotation errors in the design, we used the original mode transition table (Table 6, 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 10). The columns labeled "Messages" and "Violations" indicate the number of reported and actual invalid state transitions. These numbers overestimate the actual errors in the design. All of the mode transition problems can be attributed to four 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, state changes did not always immediately follow their triggering events, 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.

11 Conclusion

This report 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 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.

References

[1] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools, Chapter 10.* Addison Wesley, 1988.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] Frederick P. Brooks. "No Silver Bullet: Essence and Accidents of Software Engineering". *IEEE Computer*, pages 10–19, April 1987.

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

[11] 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.

[12] S.H. Caine and E.K. Gordon. "PDL: A Tool for Software Design". In *Proceedings of the National Computer Conference*, volume 44, pages 271–276, 1975.

[13] M. Chechik. *"Automated Analysis of Consistency between Requirements and Designs"*. PhD thesis, University of Maryland, College Park, Maryland, December 1996.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] W.E. Howden. "Comments Analysis and Programming Errors". *IEEE Transactions on Software Engineering*, 16(1):72–81, January 1990.

[25] W.E. Howden and B. Wieand. "QDA – A Method for Systematic Informal Program Analysis". *IEEE Transactions on Software Engineering*, 20(6):445–462, June 1994.

[26] 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.

[27] 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.

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

[29] 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.

[30] David Lorge Parnas and Jan Madey. "Functional Documents for Computer Systems". *Science of Computer Programming*, 25:41–61, 1995.

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

[32] C. Vail. *"Program Verification via Abstraction using Incremental Operational Specifications"*. PhD thesis, University of California, San Diego, 1991.

[33] 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.

[34] Mark N. Wegman and Kenneth Zadeck. "Constant Propagation". In Fran Allen, Barry Rosen, and Kenneth Zadeck, editors, *Optimization in Compilers*. ACM Press, 1991. (forthcoming).

[35] 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.