Automatic Analysis of Consistency between Requirements and Designs

Marsha Chechik, *Member*, *IEEE Computer Society*, and
John Gannon, *Senior Member*, *IEEE*

Abstract—Writing requirements in a formal notation permits automatic assessment of such properties as ambiguity, consistency, and completeness. However, verifying that the properties expressed in requirements are preserved in other software life cycle artifacts remains difficult. The existing techniques either require substantial manual effort and skill or suffer from exponential explosion of the number of states in the generated state spaces. "Light-weight" formal methods is an approach to achieve scalability in *fully automatic* verification by checking an *abstraction* of the system for only *certain* properties. This paper describes light-weight techniques for automatic analysis of consistency between software requirements (expressed in SCR) and detailed designs in low-degree-polynomial time, achieved at the expense of using imprecise data-flow analysis techniques. A specification language SCR describes the systems as state machines with event-driven transitions. We define detailed designs to be consistent with their SCR requirements if they contain exactly the same transitions. We have developed a language for specifying detailed designs, an analysis technique to create a model of a design through data-flow analysis of the language constructs, and a method to automatically generate and check properties derived from requirements to ensure a design's consistency with them. These ideas are implemented in a tool named CORD, which we used to uncover errors in designs of some existing systems.

Index Terms—SCR requirements, static analysis, formal specification, finite-state abstraction, data-flow analysis.

1 Introduction

The keys to winning acceptance for employing formal methods during system development include demonstrating that their use improves software quality, amortizing the cost of their creation across several different analysis activities, and reducing the cost of their application through automation. Software quality can be improved by eliminating errors arising from inconsistencies within the description of a system or between two different descriptions of a system.

The problem of checking consistency between different program artifacts has been worked on since the early days of computer science. Attempts at verifying that a program corresponds to its specifications were first made by Hoare, Mills, and Dijkstra in 1960s. Their methods typically suffered from the necessity to guess program invariants—a difficult task for all but the smallest programs. Theorem-proving [53], allows for checking if a property is implied by the program. This approach, although it requires considerable skill and time investment, is useful in ensuring correctness of software and has been applied in a variety of verification efforts, e.g., [16], [4], [59], [63]. Another approach is to check properties via state exploration (*model-checking* [18]). This approach is typically limited to finite-state systems but its main attractiveness is that the verification is fully automated. Model-checking has been effectively applied to verifying hardware [22], [19], [15], [49] and distributed systems, including network and security protocols [34], [47], [48], [43], [3]. Model-checking has also started to be applied to requirements engineering [6], [23], [60], [7], [64]. However, the size of the state-space grows exponentially compared to the number of variables in the problem, making all but the most trivial programs too large to analyze. Various researchers have been proposing checking *abstractions* of programs [66], [42], [35]. Unfortunately, coming up with useful abstractions and interpreting counter-examples remains difficult.

Motivated by the necessity to create highly scalable analysis techniques, we have developed a low-degree-polynomial-time approach to check low-level designs against requirements, summarized in this paper.

Requirements for embedded systems often describe a system as a set of concurrently executing state machines (see [2], [30], [46], [28]) which respond to events in their environment. Designs are frequently expressed in a program design language (PDL) [9] consisting of a concrete outer syntax of basic statement types and an inner syntax of comments. We define a design to be consistent with its requirements if the design's state transitions are enabled by the same events as those of the requirements and all the requirement's state transitions appear in the design. In a large project, these properties may be checked by design and code inspections conducted by human reviewers [45]. This process may be effective in catching local inconsistencies, but the bookkeeping tasks needed to determine all the possible system states at a particular program point make it difficult to ensure that global properties hold. In

- *M. Chechik is with the Department of Computer Science, University of Toronto, 10 King's College Rd., Toronto, ON, Canada M5S 3G4. E-mail: chechik@cs.toronto.edu.*
- *J. Gannon was with the Department of Computer Science, University of Maryland, College Park, MD 20774.*

Manuscript received 4 May 1998; revised 22 July 1999; accepted 19 May 2000.
Recommended for acceptance by M. Young and A. Andrews,
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106806.

0098-5589/01/$10.00 © 2001 IEEE

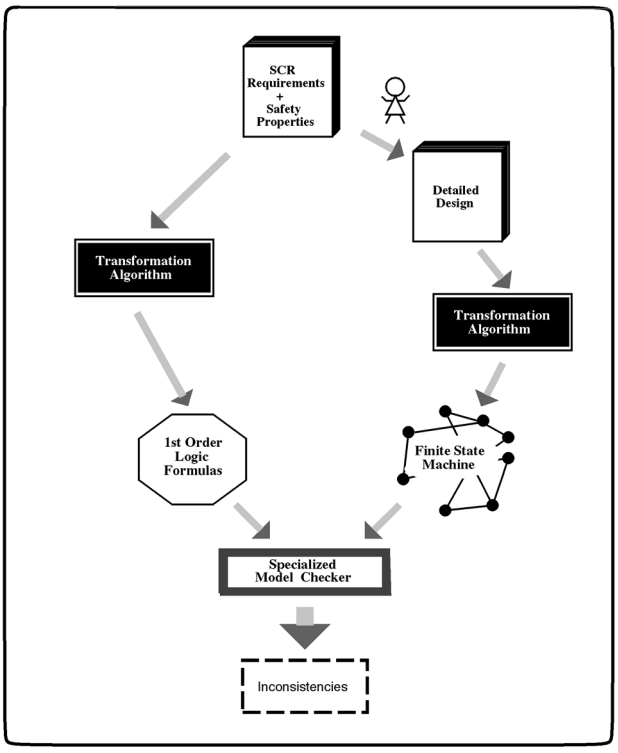

Fig. 1. Verification using CORD.

this paper, we define a notation which can be used as a PDL and describe a prototype tool, called CORD [12], [13], [11], which automatically determines if a design written in this notation is consistent with its requirements.

The inputs to our tool are a tabular requirements specification and a detailed design written in the PDL. The PDL's outer syntax is comprised of C control constructs, and its inner syntax is a set of special comments which describe local properties of values of requirements variables. As Fig. 1 illustrates, the tool generates a set of first-order logic formulas corresponding to our notion of consistency between a design and its requirements and a finite-state machine (FSM) abstraction of the design. A special-purpose model checking algorithm determines if the formulas hold in the abstraction.

The rest of the paper is organized as follows: Section 2 describes the formal model of SCR requirements. Section 3 presents a language for creating detailed designs. Section 4 presents our notion of consistency between SCR requirements and designs. Section 5 describes a process of building a finite-state machine from the detailed design. Section 6 describes an approach for checking automatically generated properties. Section 7 presents a case study during which we analyzed an existing detailed design of a water-level monitoring system [62] with respect to its SCR requirements. Finally, Section 8 summarizes our approach and compares it to related work.

2 Requirements Notation

The SCR requirements notation was developed by a research group at the US Naval Research Laboratory as part of the Software Cost Reduction project [2], [33]. The project resulted in SCR requirements and design standards, as well as guidelines for software development using SCR. 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. The specification language is precise, can be understood by engineers and software developers, and is easy to use and modify. The initial language lacked an underlying formal semantics. A number of semantics have been proposed [5], [30], [26], [54], [62], some of which became bases of tools performing consistency and completeness checks [31] and enabling simulations [21], [55] of requirements. In this section, we briefly describe SCR behavioral requirements and environmental assumptions. A more formal description can be found in [31].

The SCR model is used to specify reactive systems. The environment contains *monitored* and *controlled* variables. Monitored variables are quantities that influence the system behavior, and controlled variables are quantities that the system regulates. The required behavior of the system is specified as a black box by two relations, REQ and NAT,[1] both from monitored to controlled variables. Relation NAT specifies the natural constraints on the system behavior, imposed by physical laws and the system environment, e.g., "the system can never encounter temperatures less than $0°C$." REQ describes the ideal behavior of the system to be built. The system is considered correct if it behaves like REQ in all cases specified by NAT.

The SCR model is based on discrete-time event-driven state-transition systems. The software samples the sensors periodically or receives their values through interrupts. Changes to monitored variables may cause the system to change its state or to alter the values of its controlled variables.

Definition 2.1. *SCR requirements $\mathcal{R}$ is a tuple $\mathcal{R} = \langle \mathcal{B}, \mathcal{E} \rangle$, where $\mathcal{B}$ is a set of behavioral requirements and $\mathcal{E}$ is a set of environmental assumptions.*

$\mathcal{B}$ is the *required* relation between the monitored and the controlled quantities, and $\mathcal{E} \subseteq NAT$ represents constraints on these quantities.

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. For example, conditions SwitchOn and PumpFail can represent predicates [On/Off switch = On] and [Pump failure = true], respectively. Representing conditions as predicates allows us to assume that all monitored and controlled entities are Boolean variables. A *state* of the monitored environment is a mapping of variables to values, and a *state space* is the set of possible combinations of values of variables. We assume that the system takes one "unit of time" to move between its states. The behavior of the system is rarely affected by the values of all the variables at once. A *mode class* defines a set of states, called *modes*, that partition the monitored environment's state space. Each mode class specifies one aspect of the system's behavior, and the system's global behavior

1. These definitions have been coined by Parnas and Madey in [54].

is defined to be the composition of the specification's mode classes. At all times, the system is in exactly one mode of each mode class. One mode of each mode class is designated as the *initial mode*. Assumptions about the initial state of the environment are specified with the initial modes. In addition to Boolean entities, we use expressions in the form $mc = m$ as conditions indicating that the system is in mode m of modeclass mc; thus, we can think of a mode class as an enumerated type whose values are modes of that mode class.

The system moves between its modes and changes values of its controlled variables in response to *events*— changes in the environment. We use notation "@T" and "@F" to denote various events. A *primitive event* is an event @T/@F (a), where a is a condition. For example, @T(Running) and @F(Running) represent a condition Running *becoming true* and *becoming false*, respectively, whereas @T(Operating=Off) represents a mode class Operating moving to mode Off. A *simple conditioned event* is denoted $@\mathrm{T}(a)$ WHEN [b], where $@\mathrm{T}(a)$ is a primitive event, and b is a simple condition or a conjunction of simple conditions, called the event's WHEN *condition*. Any primitive event $@\mathrm{T}(a)$ can be represented as $@\mathrm{T}(a)$ WHEN [*true*]. A *conditioned event* is composed of simple conditioned events connected by logical connectors $\wedge$ and $\vee$. A simple conditioned event can be represented by a logical expression [32]:

$$@\mathrm{T}(a) \text{ WHEN } [b] = \neg\, a \wedge a' \wedge b,$$

where the unprimed version of a condition a denotes the value of a in the previous state, and the primed version denotes the value of a in the current state. *Compound events* can be specified using Boolean operators, connecting primitive events, or multiple conditions. Some formal semantics, e.g., [30], exclude simultaneous events. We assume that simultaneous events can occur only between variables related by environmental assumptions, defined below. Any event can be expressed as a logical statement; an event occurs if the logical statement that the event represents evaluates to true for a pair of consecutive states. For example, the conditioned event @T(Running) WHEN [Operating=Off] can be rewritten as

$$\neg\ \mathrm{Running} \wedge \mathrm{Running}' \wedge \mathrm{Operating} = \mathrm{Off}.$$

This event occurs if in the previous state, Operating is Off and Running is false, whereas, in the current state, Running is true. Finally, we assume that controlled variables cannot be part of an event since they are outputs of the system and other entities cannot depend on them.

SCR requirements use tables to define changes of values of controlled variables and mode transitions. There is one table for each controlled variable and for each mode class. Each entry in a controlled variable table defines a value of the corresponding controlled variable as a function of a system's modes and events. Such tables are called *event tables*. Each entry in a mode transition table maps a mode and an event to another mode in the same mode class.[2] In cases when a conditioned event can trigger two or more transitions, the exact transition is chosen nondeterministically.

A more formal definition of SCR behavioral requirements is given below:

Definition 2.2. *Behavioral requirements $\mathcal{B}$ expressed in SCR is a tuple $\mathcal{B} = \langle R, V, T, MT, CV, L, I\rangle$, where*

- *$R = \langle M, C, MD\rangle$ is a set of system variables, M is a set of monitored, C is a set of controlled, and MD is a set of mode class variables.*
- *V is a set of values that variables in R can attain. Thus, V^M, V^C, and V^{MD} indicate the respective values that monitored, controlled, and mode class variables can attain.*
- *T: $R \rightarrow 2^V$ is a mapping of variables to ranges of their values:*

  $$\forall\, r_j \in M \cup C, T(r_j) = \{true, false\}$$
  $$\forall\, md_j \in MD, T(md_j) = \{m_{j,1}, m_{j,2}, ..., m_{j,n}\},$$

  where $m_{j,k}$ is a mode in the mode class md_j.
- *L is the set of events,*
- *$MT : V^{MD} \times L \rightarrow V^{MD}$ is a relation describing mode transitions,*
- *$CV : V^{MD} \times L \rightarrow V^C$ is a relation describing changes of values of controlled variables, and*
- *$I : V^M \rightarrow V^{MD \cup C}$ is a relation describing the initial state of the system.*

Since all our variables have finite values, our state space is also finite. Monitored and controlled variables are Boolean, and mode class variables are enumerated types. If $|T(r)|$ indicates the number of values that a variable $r \in R$ can attain, the state space S described by the requirements is bounded above by:

$$S = 2^{|M \cup C|} \times \Pi_{1 \le j \le |MD|} |T(md_j)|.$$

We will use SCR to specify requirements of 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 variables PumpOn. Table 1 shows a mode transition table for SWLMS. Mode class MC starts in mode Off, and all monitored variables are initially false. MC transitions from Off to Operating when SwitchOn becomes true (indicated by "@T") while PumpFail is false (indicated by "f") and transitions to Error when PumpFail becomes true. Once it enters mode Operating, MC remains there until SwitchOn becomes false ("@F") while PumpFail is false, or until PumpFail becomes true. Entries marked by "–" are the "don't care" values, although some values can be inferred from environmental assumptions about variables (see below).

2. In addition to controlled and monitored variables, the SCR method described in [31] can also include terms—results of intermediate computations. We do not currently handle terms, although our analysis technique can be easily augmented to do so. The SCR method also contains *condition* rather than *event* tables which allow the specification of a value of a variable or a term with regard to a set of conditions. Condition tables can be textually translated into event tables.

TABLE 1
Mode Transition Table for Mode Class MC of SWLMS

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

Initial: Off ($\neg$SwitchOn $\wedge$ $\neg$PumpFail $\wedge$ $\neg$TooHigh $\wedge$ $\neg$TooLow)
Assumptions: TooLow–>> $\neg$TooHigh

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 MC is in mode Operating and either event @T(TooHigh) or @T(TooLow) occurs.

2.2 Environmental Assumptions

An *assumption* specifies constraints on the values of variables, 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. For example, the water in a container cannot be too high and too low at the same time, and buttons can be either pressed or released but not both. Many environmental assumptions can be expressed by declaring relationships between conditions [5]. Some sample relationships are discussed below.

implication (a–> b). The state space in which a is true is a subset of the state space in which b is true.

strict implication (a–$\gg$ b). This assumption is similar to implication except that when a becomes true, b should already be true, and when b becomes false, a should already be false.

timeline ($a < b$). Conditions a and b represent lengths of time that a particular environmental condition has been true, where b represents a longer length of time than a. The timeline assumption is a strict implication assumption whose contrapositive is nonstrict implication. Therefore, a must already be true when b becomes true and must become false when b becomes false.

enumeration ($a_0 \mid a_1 \mid \cdots \mid a_n$). The state space is partitioned such that exactly one member of the enumeration is true in each partition.

The assumptions about environmental conditions can be found in the section of an SCR requirements document that describes the system's controlled and monitored variables. A human requirements analyzer is responsible for accurately interpreting the environmental assumptions as they are stated in the specification and translating them into the appropriate syntax. The assumption specified in Table 1: TooLow–$\gg$ $\neg$TooHigh, states that the water level cannot be too high and too low at the same time, i.e., if TooHigh is true, TooLow is false, and vice versa.

We could do away with some of the environmental assumptions if we choose to represent variables as enumerated types. For example, having a variable valued TooLow, TooHigh, and Neither would automatically satisfy the above assumption. However, some of the other assumptions, like timeline, are not satisfied this easily.

TABLE 2
Event Table for Controlled Variable PumpOn

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

Initial: false

3 Detailed Design

A Program Design Language (PDL) [9] is a language used to specify designs. We have defined our own PDL to reason about designs for SCR requirements. Our PDL was motivated by the following factors:

- Designs should look like real programs, i.e., specify control flow via programming language constructs for sequence, selection, and iteration.
- Designs should capture the *essence* of what is happening in the code, rather than details [8]. When writing designs, we want to reason about requirements-level variables rather than implementation-level structures.
- Finally, designs should be able to deal with sensors and actuators of the system [54].

After a detailed design is complete and verified, it can be further refined into an implementation.

3.1 Design Constructs

Typically, control-flow-based PDLs [9] 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 PDL's inner syntax consists of *annotations*—special statements describing values of requirements variables. The use of annotations was inspired by Howden's work on QDA [38], [61].

We define four types of annotations:

- 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, assigning a value to every SCR variable.[3]
- 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

3. This is checked automatically by CORD.

current state. Our analysis usually gives imprecise results because we utilize aggressive state folding. Assert annotations reduce the amount of information in the state to what the programmer *believes* to be true.

- A *Read annotation* indicates that a variable has been assigned some input value. Given that all values are equally possible, the semantics of Read is that the variable receives the value corresponding to the union of all values of its type (denoted $\top$).

Variables in Update and Read 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, our verification tool, CORD, assumes that the latter start with @@. For the complete syntax of our PDL, see [10].

Consider the following design fragment:

```
READ_DEVICE();
@@ Read PumpFail;
if(PUMP_FAIL()) {
   @@ Assert PumpFail=true;
   break;
}
@@ Assert PumpFail=false;
...
```

In this fragment, the function READ_DEVICE() is called to determine the status of a pump. The Read annotation reflects this action for the requirement's variable PumpFail. The function PUMP_FAIL() determines if the value read corresponds to the failure of the pump. In the Then clause, we assert that the pump did fail (i.e., the value of PumpFail is true rather than $\top$), and exit an enclosing loop. Otherwise, we assert that the value of PumpFail is false rather than $\top$ and continue processing the next statement.

3.2 How to Write Designs

Designs of programs implementing SCR requirements are very stylized: An Initial annotation marks the starting state and is followed by a loop in which the system reads relevant monitors, tests their values, and then decides to change either its mode or the values of some controlled variables. Read annotations correspond to reading monitors, Assert annotations are used to document the results of testing predicates, and Update annotations mark changes in the program's state. Potentially, such designs can be automatically generated [29].

It is often necessary to hand-optimize designs for better performance or to reduce redundancy by "factoring out" common actions for different states. Fig. 2 shows the result of such an optimization for 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.

CORD can also be used to verify consistency of existing programs. To do so, we can annotate existing code with annotations 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 7). This approach is similar to that of QDA [36], [38], where the code is annotated with comments specifying user assertions (representing known information) and hypotheses (representing information to be verified).

In annotating implementations, we inevitably run into the problem of verifying the consistency of the requirements with that of the annotations rather than the source code. If annotations are done carefully, their correctness provides some assurance about the correctness of the source code. However, in order to suppress diagnostic messages from our analysis, a programmer may add or change annotations without making corresponding changes in the source code. Annotations and source code may also "diverge" as modifications to the program are being made. We have certainly noticed that ourselves: We often modify the annotations to get CORD to give us a correct answer, forgetting to update the source code. To remedy this problem, we have recently implemented a tool SAC [14], [11] that checks consistency between annotations and source code. The programmer specifies correspondences between annotation and code variables; these correspondences are not necessarily one-to-one. We will refer to code variables in the correspondences as *relevant* variables. The tool goes through the annotated code, checking the following four conditions:

1. Every Assert annotation corresponds to a test of the appropriate source code variable(s).
2. Every Update/Read annotation corresponds to an assignment of the appropriate source code variable(s).
3. Every change of a relevant source code variable corresponds to an Update/Read annotation with the appropriate annotation variable(s).
4. Every test of a relevant source code variable corresponds to an Assert annotation with the appropriate annotation variable(s).

The closer the relationship between annotation and source code variables is to one-to-one, the less spurious warnings SAC gives. Note that SAC does not check if variables are assigned correct *values*; instead, it just checks if an assignment to an appropriate variable takes place.

4 Consistency with SCR Requirements

Definition 4.1. *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. *$\mathcal{A}_\mathcal{E}$ does not implement any state transitions which are not specified in $\mathcal{B}$, and*
3. *$\mathcal{A}_\mathcal{E}$ implements all state transitions specified in $\mathcal{B}$.*

```
 1: { ...
 2:   @@ Initial MC=Off & SwitchOn=false & PumpFail=false &
         TooHigh=false & TooLow=false & PumpOn=false;
 3:   while(1) {
 4:     READ_DEVICE();
 5:     @@ Read PumpFail;
 6:     if (PUMP_FAIL()) {
 7:       @@ Assert PumpFail=true;
 8:       break;
 9:     }
10:     @@ Assert PumpFail=false; /* assume no device failures */
11:     READ_SWITCH();            /* read switch monitor */
12:     @@ Read SwitchOn;
13:     if (SWITCH_ON() && IN_OFF(System)) {
14:       @@ Assert SwitchOn=true & MC=Off;
15:       @@ Update MC=Operating;
16:     }
17:     else if (IN_OPERATING(System)) {
18:       @@ Assert MC=Operating;
19:       while(1) {
21:         READ_DEVICE();
22:         @@ Read PumpFail;
23:         if (PUMP_FAIL()) {
24:           @@ Assert PumpFail=true;
25:           break;
26:         }
27:         @@ Assert PumpFail=false;
28:         READ_SWITCH();
29:         @@ Read SwitchOn;
30:         if (!SWITCH_ON()) {
31:           @@ Assert SwitchOn=false;
32:           @@ Update MC=Off;
33:           break;
34:         }
35:         @@ Assert SwitchOn=true;
36:         GET_WATER_LEVEL (&Water); /* compute water level */
37:         @@ Read TooHigh & TooLow;
38:         if (IS_HIGH(Water) || IS_LOW(Water)) {
39:           @@ Assert TooHigh=true | TooLow=true;
40:           TURN_PUMP(ON);
41:           @@ Update PumpOn=true;
42:         }
43:         else {
44:           @@ Assert TooHigh=false & TooLow=false;
45:           TURN_PUMP(OFF);
46:           @@ Update PumpOn=false;
47:         }
48:       }
49:     }
50:   }
51:   @@ Assert PumpFail=true;
52:   @@ Update MC=Error & PumpOn=false;
53: }
```

Fig. 2. Design of SWLMS.

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 was 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.

A property capturing the first part of our definition of consistency, the fact that requirements and design have the same starting state, can be obtained from I. I is a conjunction of initial conditions associated with each mode class and controlled variable. If s_0 is the initial state of a model of the design, then this property, called START, is

$$\text{START} = s_0 \models I.$$

There is one START property generated for an SCR specification. In Section 2.1, we introduced a simple Water-Level Monitoring System (SWLMS). For SWLMS, the START property is

$$s_0 \models (\text{MC} = \text{Off} \wedge \neg\, \text{SwitchOn} \wedge \neg\, \text{PumpFail} \wedge \neg\, \text{TooHigh} \ \wedge\ \neg\, \text{TooLow} \wedge \neg\, \text{PumpOn}).$$

This property was generated from initial conditions of Tables 1 and 2.

The remaining parts of our definition of consistency deal with events. We use states to denote points at which variables change values. Thus, three states need to be

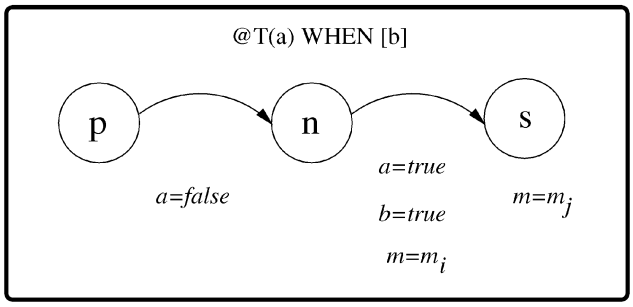

Fig. 3. Pictorial representation for a mode transition.

considered in determining if an event caused a mode change: the state in which the variable involved in the triggering condition had its original value, the state in which this variable was assigned a new value, and the state in which the mode change occurred. Generally, we might know which value has been assigned to a variable more exactly after this variable has been tested. For example, consider lines 38-47 of the design in Fig. 2. The exact values of TooHigh and TooLow, read on line 36, are not known until the test on line 38 has been performed. In particular, the exact values of variables are available at the arcs emanating from the test node. As the result, we define formulas as being true on arcs as well as in states. A transition between modes m_i and m_j of modeclass m, triggered by $@\mathrm{T}(a)$ 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. Fig. 3 gives a pictorial representation of this semantics.

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, s), we use the notation $(n, s) \models f$ to indicate that f is true on an edge between n and s. We also assume that for each state n we have functions $\mathrm{pred}(n)$ and $\mathrm{succ}(n)$ returning a list of all predecessor and successor states of n, respectively (this list can be empty). We express all properties using statements about three-state sequences $(p \in \mathrm{pred}(n), \ n, \ s \in \mathrm{succ(n)})$. For example, a property "there exists a transition from $m = m_i$ to $m = m_j$ on $@\mathrm{T}(a)$ WHEN [b]" is expressed as

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

For brevity, we will write (p, n, s) to indicate $(p \in \mathrm{pred}(n), \ n, s \in \mathrm{succ(n)})$. Properties quantified on all of (p, n, s) are considered vacuously true when $\mathrm{pred}(n)$ or $\mathrm{succ}(n)$ are empty. A number of properties generated from SCR have the notion of an event in them. We say that an event $@\mathrm{T}(a)$, where a is a Boolean variable, has occurred on a three-state sequence (p, n, s), i.e., $(p, n, s) \models @\mathrm{T}(a)$, if

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

A set of properties capturing the second and the third parts of our definition of consistency can be obtained by composing rows and columns of SCR tables. 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, in SWLMS, we have a property asserting that the only way for mode class 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

$$P_1 = \forall(p, n, s), \ (s \models (\mathrm{MC} = \mathrm{Off})) \rightarrow (((n, s) \models (\mathrm{MC} = \mathrm{Off})) \\ \vee \ (((p, n, s) \models @\mathrm{F(SwitchOn)}) \\ \wedge ((n, s) \models ((\mathrm{PumpFail} = \mathrm{false}) \wedge (\mathrm{MC} = \mathrm{Operating}))))).$$

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 second part of our notion of consistency and are called "only legal transitions" (OLT) properties. OLT properties generated from requirements of SWLMS are shown in Fig. 4. 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

$$P_6 = \exists(p, n, s), \ (s \models (\mathrm{MC} = \mathrm{Operating})) \\ \wedge \ ((p, n, s) \models @\mathrm{T(SwitchOn)}) \ \wedge \ ((n, s) \models ((\mathrm{MC} = \mathrm{Off}) \\ \wedge \ (\mathrm{PumpFail} = \mathrm{false}))).$$

Such properties ensure that all transitions specified in the requirements should appear in the design, capturing the last 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 Fig. 5. 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.

The total number of properties is proportional to the total number of rows in SCR tables.

5 Creating the Abstraction

We construct a Design-Flow Graph (DFG) from annotations and control-flow information and compute an approximation of the possible system states at each node of the DFG using data-flow analysis techniques. The DFG is abstracted into a finite-state machine (FSM) by removing nodes that

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

Fig. 4. OLT properties for SWLMS.

are unreachable or do not correspond to annotations that indicate a state change. Section 6 describes how the FSM is compared to properties generated from requirements.

During the analysis, five kinds of properties are checked: START ("starting state is correct"), OLT ("only legal transitions"), ALT ("all legal transitions"), ENV ("environmental assumptions are preserved"), and REACH ("all nodes are reachable"). Fig. 6 depicts the steps needed to create and check an abstraction:

- Compute the information generated by each annotation. Use environmental assumptions to add values generated by related variables and report any violations of the assumptions caused by inconsistent annotations.

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

Fig. 5. ALT properties for SWLMS.

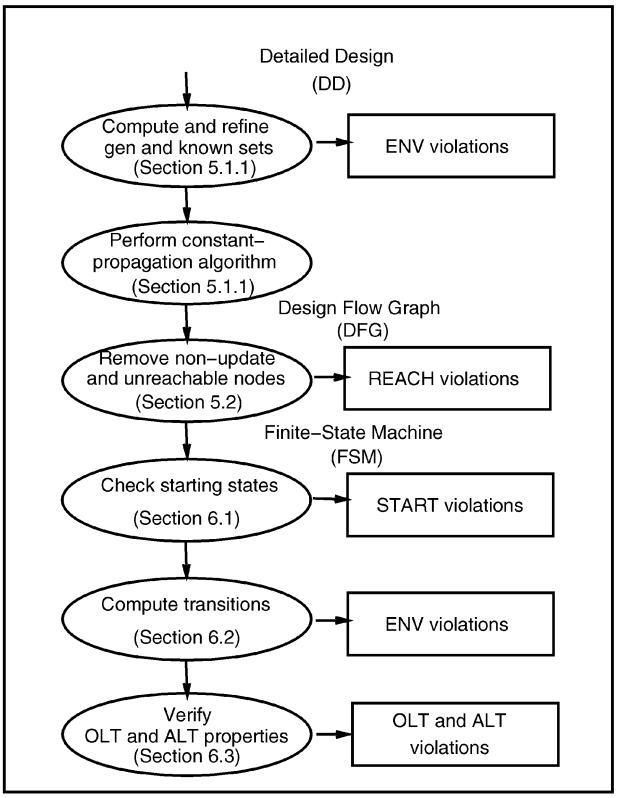

Fig. 6. Analysis roadmap.

- Propagate values throughout the DFG.
- Remove nodes not corresponding to Initial, Read, and Update annotations to create a FSM. Report and remove nodes which cannot be reached.
- Check that the initial state of the design is correct (START property). If an error is found, stop the processing.
- For each FSM node, compute the set of events causing transitions from predecessor nodes and verify OLT and ALT properties.

5.1 Design Flow Graphs

Definition 5.1. *A Design-Flow Graph (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 the design.*
- *$E \subseteq V \times V$ 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.*

A state is associated with every DFG node. Each variable in a state is associated with a set of values it may attain if the control reaches that node.

Definition 5.2. *A 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 $\forall\ r_j \in R$, $v_{r_j} \in 2^{T(r_j)}$ (v_{r_j} is a set consisting of values in the domain of r_j).*

The values for controlled and monitored variables are {}, {true}, {false}, and {true, false}, which form a $\cup$-lattice on set-inclusion. These values have the following meanings:

{ }. On all paths leading to this node, the value of the variable is unknown.

{true}. On some paths leading to this node, the variable is true. On others it may be unknown.

{false}. On some paths leading to this node, the variable is false. On others it may be unknown.

{true, false}. On some paths the variable is true, and on some others it is false. It may be unknown on some paths.

The values of mode class variables also form a $\cup$-lattice on set-inclusion.

We require that there is exactly one variable-value pair for each variable in the requirements and, thus for a 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:

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

and a special state EMPTY:

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

Operations on states include "$\sqcup$" (union), "$\sqcap$" (intersection), "=" (equality), "$\sqsupset$" (superset), "$\setminus$" (difference) and "$\sqsupseteq$" (superset or equal to). These operations are defined in Fig. 7. Note that we overloaded the function repl to take two states as parameters. States form a complete $\cup$-lattice under the partial ordering of inclusion $\sqsupset$.

5.1.1 Computing Values of States

Our computation of 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 [65], [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.

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

- For nodes corresponding to Update, Read, and Initial annotations, `gen` sets contain variable-value pairs with the specified value ($\top$ for Read), 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 also include a disjunction of several clauses. For these nodes, `known` contains sets of

$$
\begin{array}{lcl}
ss_1 \sqcup ss_2 & \equiv & \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, ss_1) \cup v(r_j, ss_2)\} \\
ss_1 \sqcap ss_2 & \equiv & \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, ss_1) \cap v(r_j, ss_2)\} \\
ss_1 \setminus ss_2 & \equiv & \{(r_j, \mu_j) \mid \forall r_j \in R, \mu_j = v(r_j, ss_1) \setminus v(r_j, ss_2)\} \\
ss_1 = ss_2 & \equiv & \forall r_j \in R, v(r_j, ss_1) = v(r_j, ss_2) \\
ss_1 \sqsupseteq ss_2 & \equiv & \forall r_j \in R, v(r_j, ss_1) \supseteq v(r_j, ss_2) \\
ss_1 \sqsupset ss_2 & \equiv & (ss_1 \sqsupseteq ss_2) \ \wedge \ (ss_1 \neq ss_2) \\
\mathrm{repl}(ss_1, ss_2) & \equiv & \forall r_j \in R, \ \text{if } v(r_j, ss_2) \neq \{\} \\
 & & \mathrm{repl}(r_j, v(r_j, ss_2), ss_1)
\end{array}
$$

Fig. 7. Operations on system states for set-based approximation.

states, one for each clause. Operations on states are extended in a natural way to handle sets of states. In the examples below, we omit variable-value pairs in which the value is {}. For an annotation @@Update MC=Operating & PumpFail=true,

$$\texttt{gen} = \{(\mathrm{MC}, \{\mathrm{Operating}\}), (\mathrm{PumpFail}, \{\mathrm{true}\})\}.$$

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

$$\texttt{known} = \{\{(\mathrm{SwitchOn}, \{\mathrm{false}\})\}, \{(\mathrm{PumpFail}, \{\mathrm{true}\})\}\}.$$

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

$$\texttt{known} = \{(\mathrm{TooLow}, \{\mathrm{true}\}), (\mathrm{TooHigh}, \{\mathrm{false}\})\}.$$

Environmental assumptions can also be used to make sure that contradictory variable values are not asserted. Errors are considered violations of the ENV property. Details of this processing are presented in [10].

The system state for each DFG node is the value we compute for its out set. We initialize in and out sets of every node to EMPTY and propagate information throughout the DFG until a least fixed point is reached. A *join* 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, Read	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., $\texttt{known}(n) = \{d_1, d_2, ..., d_k\}$, 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 at the end of each iteration of our algorithm have at least as many values associated with each variable as at the beginning. Since all variables in R have a finite number of abstract values, our states do not have an infinite increasing chain of values, and the fixed point can be achieved in a finite number of steps. The height of the lattice of states is the length of the longest increasing chain of values, i.e., the maximum number of times that information for each node can be changed before the fixed point is achieved, is

$$\mathcal{H} = \Sigma_{r_j \in R} |T(r_j)| + 1.$$

Thus, the entire computation of in and out sets for the DFG can take at most

$$O(|V|^2 \times (\Sigma_{r_j \in R} |T(r_j)| + 1))$$

steps, where $|V|$ is the number of nodes in the DFG.

5.1.2 Example: DFG of SWLMS

Consider computing DFG for the design of SWLMS (Fig. 2). Fig. 8 shows a fragment of this DFG corresponding to lines 38-47 of the SWLMS design. gen and known sets computed at each node are shown in bold font in this figure; variables with values {} are 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 assumptions, 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 Fig. 8 because these sets are equal to the out sets of their predecessors for nodes 2-5 and to their out sets for nodes 1 and 6. To compute out sets, CORD uses the multiple disjunct transfer function for Assert nodes. Each of the 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 of the possible values for each variable in the out sets of the predecessor nodes (nodes 3 and 5).

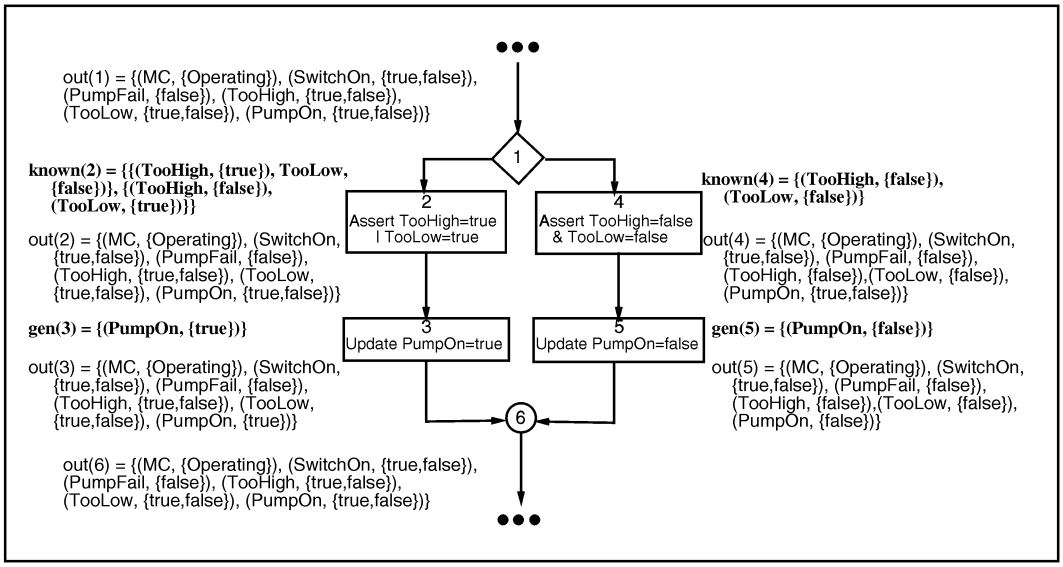

Fig. 8. A fragment of DFG of SWLMS.

5.2 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 only contains reachable nodes representing state changes. The resulting FSM is used as a model for verifying system properties.

Definition 5.3. *A Finite-State Machine (FSM) over a program design is a structure* $M = \langle A, S, L, N, s_0 \rangle$*, where*

- A *is a set of labels,*
- $S = U \cup I \cup Rd$ *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, where* N *is obtained by connecting nodes of* S*, s.t., there is an Update- and Read-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 (I), Update (U), and Read (Rd) annotations, and connect all predecessors of a removed node to the node's successors. Let $S = I \cup U \cup Rd$ be the set of these nodes. We assume that all variables are initialized by an Initial annotation, so for every node, we check an implicit property (REACH):

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

Note that this property could have been checked after the DFG had been constructed. However, checking it while building the FSM limits the reported violations to lines containing annotations. If the REACH property is violated in a node, an error is reported and the node is removed from the FSM. The running time to build the FSM is proportional to $|E|$, the number of edges in the DFG, and the number of nodes in the resulting FSM is bounded above by $|S|$, the number of state changes in the design.

Fig. 9 shows the FSM created for the SWLMS design in Fig. 2 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, Read, or Initial annotation can be found. For example, nodes 41 and 46 correspond to @@Update PumpOn=true and @@Update PumpOn=false, respectively. The algorithm for computing `out` sets ensures that the effects of Assert annotations are preserved in system states, even though Assert nodes themselves are removed.

6 Verifying Properties

Our method for constructing finite-state abstractions produces sets of values for each program variable. Model checkers [18] 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 for verifying properties.

The semantics of an SCR event, given in Section 2.1, indicates that some formulas need to hold on states or edges. However, our FSM consists of just states, with `in`, `gen` and `out` sets. Formally, we say that an atomic formula f (f: variable = value) *holds in a state* s if f holds in $\texttt{out}(s)$, i.e., $\texttt{out}(s) \models f$. Also, we say that f holds on an edge between nodes n and s if $(\mathbf{out}(n) \sqcap \mathbf{in}(s)) \models f$. By construction of the FSM, if a formula holds on the exit from the node but does not hold on the 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.

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. Properties may be checked *optimistically* or *pessimistically*.

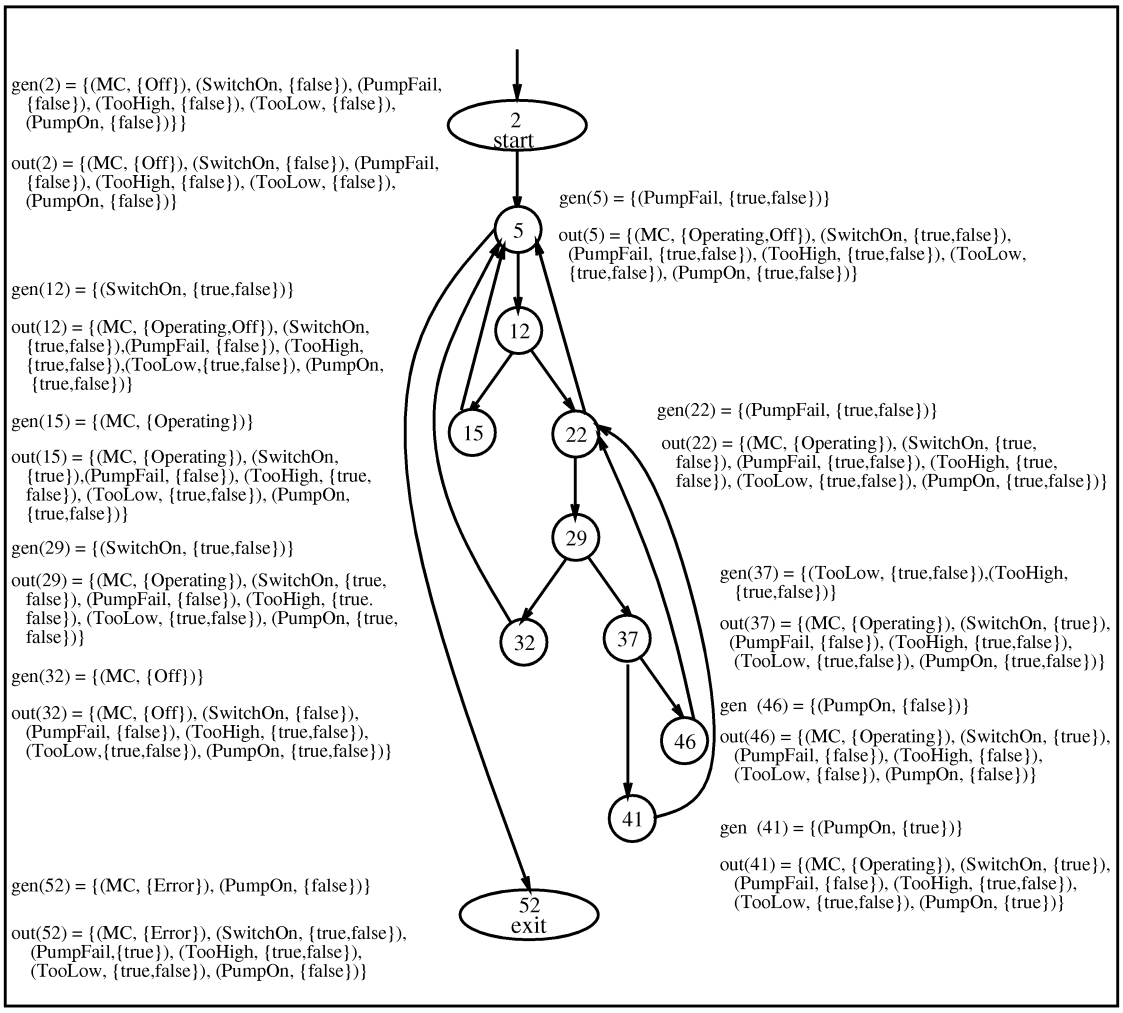

Fig. 9. Finite-state abstraction.

Definition 6.1. *A property is checked pessimistically if all of its violations in the design are detected, but the analysis may incorrectly identify 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 may be unable to find all violations of the property.*

6.1 Checking Starting States

We use exact checking so that the initial conditions of the SCR requirements and the design are the same, i.e., an error is reported if and only if there is a violation. By construction of the FSM, s_0, the starting state, is a node corresponding to the Initial annotation of the design. Thus, verifying the START property reduces to checking that the variable-value assignment of the gen set of the starting state is a model of the requirements' initial condition. However, if the START property is violated, we report an error and halt the rest of the analysis since in this case the behavior of the rest of the system is undefined.

For example, the START property of the SWLMS is

$$s_0 \models (\mathrm{MC} = \mathrm{Off} \wedge \neg\mathrm{SwitchOn} \wedge \neg\mathrm{PumpFail} \wedge \neg\mathrm{TooHigh} \wedge \neg\mathrm{TooLow} \wedge \neg\mathrm{PumpOn}).$$

In the FSM corresponding to the design of the SWLMS, s_0 is node 2 (see Fig. 9). gen(2) is

$$\{(\mathrm{MC}, \{\mathrm{Off}\}), (\mathrm{SwitchOn}, \{\mathrm{false}\}), (\mathrm{PumpFail}, \{\mathrm{false}\}), \} \\ (\mathrm{TooHigh}, \{\mathrm{false}\}), (\mathrm{TooLow}, \{\mathrm{false}\}), (\mathrm{PumpOn}, \{\mathrm{false}\}).$$

So, we check that this assignment satisfies

$$\mathrm{MC} = \mathrm{Off} \wedge \neg\mathrm{SwitchOn} \wedge \neg\mathrm{PumpFail} \\ \wedge \neg\mathrm{TooHigh} \wedge \neg\mathrm{TooLow} \wedge \neg\mathrm{PumpOn},$$

which it does.

6.2 Computing Transitions

OLT and ALT properties involve state transitions which occur only in response to events. 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 occurs from state-folding during construction of the FSM, described in Section 5, and 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

TABLE 3
Results of Analysis

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

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.[4] 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 a 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.

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 Fig. 10. 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,
- `trigger` is a 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.

In summary, the algorithm constructs transitions into node s by joining `when` conditions from each node n, a predecessor of s, with triggering conditions from the changes between n and its predecessors. The running time of this algorithm is

$$O(|S|^2 \times \Sigma_{r_j \in R}|T(r_j)| + k \times (\Sigma_{r_j \in R}|T(r_j)|)^2),$$

where k is the number of disjuncts in triggs. In our experience, $k < 10$ and does not depend on the size of the specification or the design. The maximum number of transitions generated by this algorithm for node s is

$$Trans_s = O(|S| \times k \times \Sigma_{r_j \in M \cup C}(|T(r_j)|)^2).$$

The function $\texttt{events}(r_j, s_1, s_2)$ checks values of a variable r_j in states s_1 and s_2 to determine if an event involving r_j has occurred. If s_1 is a node containing the Initial annotation, `events` returns $\mathrm{none}(r_j)$. "none" here is a symbolic constant to indicate that the variable did not change its value. Otherwise, for a Boolean r_j, $\texttt{events}(r_j, s_1, s_2)$ is defined in Table 4. Thus, for a Boolean variable r_j, `events` returns a disjunction of $@\mathrm{T}(r_j)$, $@\mathrm{F}(r_j)$ or $\mathrm{none}(r_j)$.

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

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

Finally, a function $\texttt{partof}(r_j, \texttt{trigger})$ 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 52 of the FSM in Fig. 9. Fig. 11 shows a fragment of this FSM. `when` is $\texttt{out}(5) \sqcap \texttt{in}(52)$, namely,

$$\begin{aligned}&\{(\mathrm{SwitchOn}, \{\mathrm{true}, \mathrm{false}\}), (\mathrm{TooHigh}, \{\mathrm{true}, \mathrm{false}\}), \\ &(\mathrm{TooLow}, \{\mathrm{true}, \mathrm{false}\}), (\mathrm{MC}, \{\mathrm{Operating}, \mathrm{Off}\}), \\ &(\mathrm{PumpOn}, \{\mathrm{true}, \mathrm{false}\}), (\mathrm{PumpFail}, \{\mathrm{true}\})\}.\end{aligned}$$

$\texttt{gen}(5)$ (node 5 is a predecessor of 52) is (PumpFail, {true,false}), so in order to compute triggs, we call `events` for PumpFail:

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

4. OLT properties are "universal," whereas ALT properties are "existential", and in principle can be considered to be the "dual" of the former. Thus, if OLT properties are to be checked pessimistically, ALT properties will be checked optimistically.

```
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 ∈ pred(s) and to ≠ from.
Algorithm:
         all_transitions = {}
         For each n ∈ pred(s) {
            /* triggs is a logical expression ⋀_{r_j∈R\C} ⋁_k exp_{j,k} */
            triggs = true
            when = out(n) ⊓ in(s)
            For each r_j ∈ R \ C, where v(r_j, gen(n)) ≠ {}
               /* Controlled variables cannot be part of the triggering condition */
               triggs = triggs ∧ events(r_j, ⋃_{∀p∈pred(n)} out(p), gen(n) ⊓ when)
            /* Rewrite triggs to be in form ⋁_k ⋀_{r_j∈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 ∈ 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 ∈ R \ M s.t. v(r_j, gen(s)) ≠ {} {
                  /* remove values changed between source and destination nodes */
                  when[k] = repl(r_j, when[k], {})
                  For each value_l ∈ v(r_j, gen(s)) {
                     to = (r_j = value_l) /* final value of r_j */
                     For each value_m ∈ v(r_j, out(n) ⊓ in(s)) {
                        from = (r_j = value_m) /* starting value of r_j */
                        If value_l ≠ value_m /* otherwise there is no transition */
                           all_transitions = all_transitions ∪
                                    (n, to, from, trigger[k], when[k])
                     }
                  }
               }
            }
         }
```

Fig. 10. Algorithm for computing transitions.

gen(5) ⊓ when = {(PumpFail, {true})}, which indicates that the designer assumed that PumpFail is true in node 52. Since PumpFail is {false} in nodes 2, 15, and 32, and {true,false} in node 22,

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

The call to events yields @T(PumpFail) ∨ none(PumpFail). The second disjunct corresponds to paths on which PumpFail becomes true on line 22 and is set to true again 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(52)). On the first iteration of the loop, to is (MC=Error) and from is

TABLE 4
Definition of $\texttt{events}(r_j, s_1, s_2)$

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

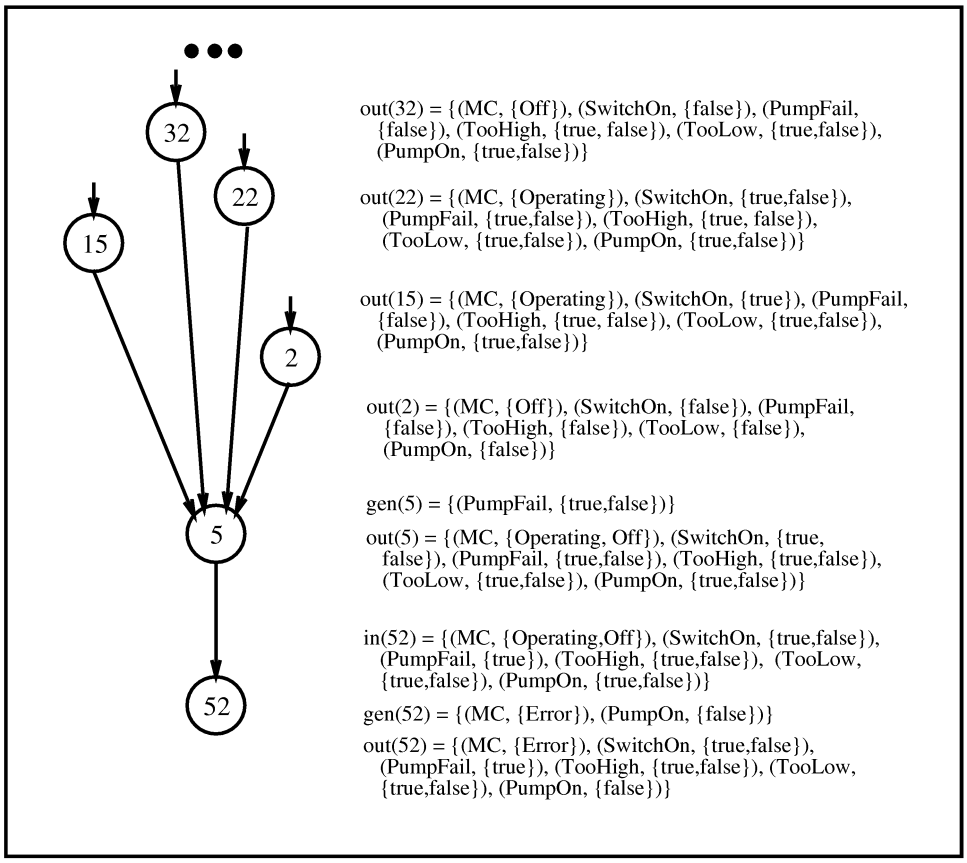

Fig. 11. Calculating transitions for SWLMS.

(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 Fig. 12.

6.3 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 the algorithms shown below appear in [10].

6.3.1 Verification of OLT Properties

OLT properties have the general form

$$\begin{aligned} P_i = \forall(p,n,s),\ (s \models (r = v_{new})) \rightarrow (((n,s) \models (r = v_{new})) \\ \vee \bigvee_j (((p,n,s) \models \texttt{trcond}_j) \\ \wedge ((n,s) \models ((r = v_{j,old}) \wedge \texttt{whcond}_j)))), \end{aligned}$$

where v_{new} and $v_{j,old}$ 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}. OLT properties are verified pessimistically and their violations are reported as soon as they are discovered. Since we compute a number of transitions for a single Update or Read annotation, we can report a number of OLT violations for a given line in the design, as outlined by the algorithm in Fig. 13. If $Trans$ is the total number of transitions computed for the FSM ($Trans \leq Trans_s \times |S|$), P_{olt}, the set of OLT properties, and D, the maximum number of disjuncts in P_{olt}, then the running time of the algorithm is

$$O(|P_{olt}| \times D \times \Sigma_{r_j \in R} |T(r_j)| \times Trans).$$

We take advantage of the fact that all properties have been generated from SCR tables and, thus, $\texttt{trcond}_j$ consists of a conjunction of one or more simple Triggering conditions (e.g., $@\mathrm{T}(a)$). Our algorithm for computing 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 `trigger`.

Before checking that $\texttt{when} \sqsubseteq \texttt{whcond}_j$, we first represent $\texttt{whcond}_j$ as a set of variable-value pairs. For example, $\mathrm{PumpOn} = \mathrm{true}$ is treated as (PumpOn, {true}), and $\neg(\mathrm{MC} = \mathrm{Operating})$ 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 $(r_k, T(r_k))$. One of the OLT properties for SWLMS is

$$\begin{aligned} &P_3 = \\ &\forall(p,n,s), (s \models (\mathrm{MC} = \mathrm{Error})) \rightarrow (((n,s) \models ((\mathrm{MC} = \mathrm{Error})) \\ &\vee ((p,n,s) \models @\mathrm{T}(\mathrm{PumpFail})) \wedge ((n,s) \models (\mathrm{MC} = \mathrm{Operating}))) \\ &\vee\ ((p,n,s) \models @\mathrm{T}(\mathrm{PumpFail})) \wedge ((n,s) \models (\mathrm{MC} = \mathrm{Off})))). \end{aligned}$$

I_1:		Transition from MC=Off to MC=Error:
	@T(PumpFail)	WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false}), (PumpFail, {}), (MC, {})}]
I_2:		Transition from MC=Off to MC=Error:
	NONE	WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false}), (PumpFail, {true}), (MC, {})}]
I_3:		Transition from MC=Operating to MC=Error:
	@T(PumpFail)	WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false}) (PumpFail, {}), (MC, {})}]
I_4:		Transition from MC=Operating to MC=Error:
	NONE	WHEN [{(SwitchOn, {true,false}), (TooHigh, {true,false}), (TooLow, {true,false}), (PumpOn, {true,false}), (PumpFail, {true}), (MC, {})}]

Fig. 12. Transitions discovered for node 52.

In this property,

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

For all nodes other than 52, MC $\neq$ Error, so P_3 holds vacuously. The transitions generated for node 52 are I_1-I_4, as shown in Fig. 12. For I_1, the `from` part is $(r = v_{2,old})$, `trigger` $\rightarrow$ $\texttt{trcond}_2$, and `when` $\sqsubseteq$ $\texttt{whcond}_2$, so no errors are reported. The case for I_3 is similar: the `from` part is $(r = v_{1,old})$, `trigger` $\rightarrow$ $\texttt{trcond}_1$, and `when` $\sqsubseteq$ $\texttt{whcond}_1$. The `triggers` for transitions I_2 and I_4 are NONE, indicating that no Triggering conditions were found, so CORD reports an error message:

Inputs: A set $\{P_i\}$ of OLT properties, where
$P_i = \forall(p, n, s),\ (s \models (r = v_{new})) \rightarrow (((n, s) \models (r = v_{new})) \vee \bigvee_j (((p, n, s) \models \texttt{trcond}_j) \wedge ((n, s) \models ((r = v_{j,old}) \wedge \texttt{whcond}_j))))$,
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 ≠ 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 → trcond_j AND
                    when ⊑ whcond_j
                Then found = true
            If not found
                report a violation of P_i at node n.
        }
    }
```

Fig. 13. Algorithm for verifying OLT properties.

Inputs: A set $\{P_i\}$ of ALT properties, where
$P_i = \exists(p, n, s),\ (s \models (r = v_{new})) \wedge ((p, n, s) \models \texttt{trcond})$
$\wedge((n, s) \models ((r = v_{old}) \wedge \texttt{whcond}))$,
Finite state machine
Outputs: Error messages indicating violations of P_is 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` $\rightarrow$ `trigger` AND
`whcond` $\sqsubseteq$ `when`
Then mark P_i
Report all unmarked ALT properties

Fig. 14. Algorithm for verifying ALT properties.

```
Error on line 52 of function main
in mode class MC:
  no triggering condition for transition
  from mode(s) {Operating,Off} to mode(s)
  {Error}
```

None of the messages reported by CORD for SWLMS are spurious, but this is not true in general. Spurious messages come from checking that `when` $\sqsubseteq$ `whcond`, and often constitute a fair share of all reported messages (see Section 7).

"Don't care" conditions in SCR turned out to be ideal for set-based approximation that we use as the basis of CORD. Attempts to implement *exact analysis*—a technique that keeps track of intervariable dependencies [10]—quickly showed that not only does the running time of the algorithm go from polynomial to exponential, the rate of spurious messages does not decrease. The reason is that many states of the system result from infeasible paths, and the exact analysis lists every combination that is not correct, including variables which are not relevant to a property being verified, overwhelming the user.

6.3.2 Verification of ALT Properties

All ALT properties have the general form

$$P_i = \exists\,(p, n, s),\ (s \models (r = v_{new})) \wedge ((p, n, s) \models \texttt{trcond}) \wedge ((n, s) \models ((r = v_{old}) \wedge \texttt{whcond})),$$

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 verified optimistically, so 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 Fig. 14. For ALT properties, we translate "don't care" conditions in `whcond` to empty sets, so `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, $v_{new} \neq v_{old}$.

If P_{alt} is the set of ALT properties, then the running time of this algorithm is

$$O(|P_{alt}| \times \Sigma_{r_j \in R}|T(r_j)| \times Trans).$$

Consider verifying property P_7 of the SWLMS:

$$P_7 = \exists\,(p, n, s),\ (s \models (\mathrm{MC} = \mathrm{Error})) \wedge\ ((p, n, s) \models @\mathrm{T}(\mathrm{PumpFail}))\ \wedge\ ((n, s) \models (\mathrm{MC} = \mathrm{Off})).$$

In this property,

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

Transitions generated for node 52, 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` $\rightarrow$ `trigger`, and `whcond` $\sqsubseteq$ `when`.

7 Case Study

To demonstrate our analysis technique on a more realistic application, we conducted a case study of a Water-Level Monitoring System (WLMS) which had been specified using SCR requirements notation and, subsequently, implemented [62]. To create the design, we reverse engineered an existing implementation of WLMS in order to determine what types of errors can be detected with our methods.

A WLMS is a safety monitoring device which serves as a component of a steam generator application. It ensures that the water level is within a specified range while the stream generator is in operation. Inside this allowable water-level range, there is a specified hysteresis water-level

TABLE 5
WLMS Modes

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

range. The area inside the allowable water-level range but outside the hysteresis range is used as a buffer to keep the generator from toggling on and off when the water level is near the limits of its allowable range. The system also raises visual and audio alarms and shuts off its pump when the level is out of range or when the monitoring system fails. In addition, there are two buttons on the system control console: the SelfTest button permits the operator to test the system, and the Reset button permits the operator to return the monitoring system (and the stream generator) to normal operation, provided that the water level is inside the hysteresis range. A complete description of this system can be found in [62]. WLMS has two mode classes, Normal and Failure, whose modes are described in Table 5. The system starts in mode Standby of mode class Normal and mode AllOK of mode class Failure. Monitored variables indicate the water level in the container (both that it is within its limits and its more stringent hysteresis range, InsideHysR $->$ WithinLimits), the lengths of time that buttons have been pressed (SlfTstPressed $<$ SlfTstPressed500) or that the system has been in a mode (InTest $<$ InTest2000 $<$ InTest4000 $<$ InTest14000), and device failures. Controlled variables are set to trigger alarms and to display the water level to the operator. A mode transition table for mode class Normal is shown in Table 6. Environmental assumptions which appear in this table have been deduced from descriptions of the conditions in the requirements.

All together, the system specification consists of two mode classes with three and four modes each, 18 monitored and seven controlled variables. The number of potential states for this system is

$$S = 2^{18+7} \times 3 \times 4 = 2^{25} \times 12 = 4.026 \times 10^8.$$

To build a design, we reverse engineered an existing implementation of the WLMS, originally consisting of roughly 1,300 lines of FORTRAN and Assembler code. First, we translated it into C and changed the Assembler routines for manipulating the screen with a GUI written in Xlib. Then, we manually reverse engineered the implementation. This process entailed determining the meaning of the code in terms of requirements and capturing programmer assumptions and state changes in our PDL. The resulting design was about 300 lines long, with 45 Update, 22 Read, and 56 Assert annotations. Out of 54 functions in the original program, only eight had state changes and, thus, were included into the design. The complete design can be found in [10]. For this design, the DFG and the FSM contained 216 and 64 states, respectively.

We ran the analysis on a SPARC 5 with 110 MHz microSPARC II CPU and 64 megabytes of memory. The time to analyze the design, as measured by the Unix **time** command was 12.0s (real), 8.5s (user) and 1.3s (system). We found it was necessary to rerun the analysis a number of times before we could get the annotations right (SAC [14], [11] did not exist yet) and, thus, a quick response was appreciated.

After we eliminated annotation errors, CORD reported a number of inconsistencies between the requirements and the design (see Table 7). Messages produced by CORD are organized in three columns: "Messages" indicates the total number of reported messages, "Spurious Messages" indicates messages that did not correspond to errors, and "Violations" indicates the number of invalid mode transitions or changes of values of controlled variables. Even after subtracting spurious messages from all messages, the numbers overestimate the actual errors in the design. Some mode transitions and controlled variable value changes resulted in a number of OLT properties violations. For example, eight illegal mode transitions generated 15 violation messages because several illegal transitions were detected at each location.

In this case study, all of the mode transition problems can be attributed to four principal causes:

1. The wrong monitored variable was checked to enable mode transitions (WithinLimits rather than InsideHysR).
2. The times that the operator pressed the SelfTest and Reset buttons were not calculated or checked.
3. Some events were computed several statements before mode transitions triggered by them occurred.
4. 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. Finally, even

TABLE 6
Mode Transition Table for Mode Class Normal

Current Mode	Inside HysR	Within Limits	SlfTst Pressed	SlfTst 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 (¬SlfTstPressed500 ∧ ¬ResetPressed3000 ∧ ¬InTest14000 ∧ ¬ShutdownLockTime200)

Assumptions: InsideHysR -> WithinLimits
SlfTstPressed < SlfTstPressed500
ResetPressed < ResetPressed3000
LDTestVal | LD0
InTest0 < InTest2000 < InTest4000 < InTest14000

though ALT properties were checked optimistically, we found that a large portion of the specification was not implemented. The WLMS design and the messages reported by CORD are available in [10].

The case study showed that CORD can be applied to the analysis of designs of realistic systems. It can find subtle errors quickly and effectively, giving easy to interpret messages. The code of WLMS has not been implemented with as careful of a definition of events as we use in this work. Thus, it led to several event-related errors. A more careful way of treating events in designs can significantly reduce the number of errors identified by CORD and lead to better-quality programs.

8 Conclusion and Related Work

In this section, we describe related work and summarize the paper.

8.1 Related Work

CORD contains features similar to those in several other static analysis systems. To simplify the verification of properties of programs, these systems restrict the forms of their formal specification notations or create abstract models from programs that could be analyzed with state-exploration rather than theorem-proving techniques.

In Inscape [56], [57], [58], complex logical formulas are abstracted to simple predicates which may be primitive or defined in terms of other predicates (like our environmental assumptions). Predicates form pre- and postconditions used to specify implementations. A programmer constructs an implementation with an editor that analyzes the implementation's control flow and operation invocations to calculate its pre- and postconditions. During the calculation, Inscape uses pattern matching and simple deduction to determine if the precondition of an operation has been satisfied before its invocation. If not, unsatisfied predicates are propagated backwards through the control-flow graph until Inscape finds operations satisfying them. The predicates of an operation's postconditions are propagated forward through the graph so that they might satisfy a subsequent operation's precondition. To determine if an implementation is correct, Inscape compares the calculated and the specified conditions.

TABLE 7
Results of Analyzing the WLMS

Property Type	Messages	Spurious Messages	Violations
REACH property	10		
OLT properties for mode classes	21	6	8
OLT properties for controlled variables	51	14	12
No events found	15		
ALT properties for mode classes	13		
ALT properties for controlled variables	21		

LCLint [25] is a tool developed to do static checks on C source code using LCL specifications. When no specifications are provided, LCLint behaves like lint, detecting uninitialized variables and incorrect parameter-passing. However, with more specifications, it is able to detect violations of abstraction boundaries, undocumented uses of global variables, incorrect dependencies between variables, etc. LCLint was designed to provide "useful" information, so some of the checks are neither sound nor complete. The tool also has extensive facilities to control the kinds of messages reported to the user. More recent work [24] extends LCLint to detect a broad class of pointer-related errors, like misuses of null pointers, uses of dead storage, memory leaks, and dangerous aliasing. This approach takes advantage of annotations which make certain assumptions, like "pointer cannot have the value NULL," explicit at interface points.

Quick Defect Analysis (QDA) [36], [38] also uses a simple annotation language. Annotations called hypotheses are embedded in comments to describe properties that objects should have at particular program points. Other comments contain assertions about properties of objects. An interpreter builds an abstract model of the implementation from the assertions and the implementation's control flow graph. Hypotheses are verified with respect to this model. More recent work [37] enriches QDA's specification language so assertions also describe event occurrences, and hypotheses assert that the implementation's events occur in certain sequences. However, QDA annotations do not correspond to program units, like functions, loops, etc. Some do not even describe observable properties, but rather record intentions of the programmer about a role of a variable. The QDA is similar to our verification of implementations, where we annotate existing code with events corresponding to changes and tests of values of requirements variables. However, our approach is more formal, taking advantage of complete system specification and being able to interpret the results as strictly optimistic or strictly pessimistic.

The Cecil specification language permits the description of sequencing constraints on user-definable program events (e.g., definitions or uses of variables, operation invocations, etc.) by anchored, quantified regular expressions (AQREs) [50], [51], [52]. After a user specifies a mapping from programming language constructs to Cecil events, the Cesar analyzer uses dataflow analysis techniques to determine if the implementation meets Cecil constraints.

Aspect's [39], [40], [41] specification notation permits users to write pre- and postconditions about the data dependencies of an operation. Dataflow analysis is used to compute an upper bound on the data dependencies of the implementation. If an asserted dependency is missing, an error is reported.

Clarke et al. [17] also create abstract, finite state models of programs, and use model checking techniques to verify formulas. Programs written in a special finite-state programming language are translated into relational expressions characterizing the program's initial state and transition relation. To reduce the size of the model, users define mappings of implementation values to abstract values and symbolically execute operations on the values. The model checking approach is pessimistic for formulas expressed in $\forall \mathrm{CTL}^*$ [27], a subset of CTL in which only universal path quantification is allowed. The authors also identify a large class of temporal formulas for which the verification results are exact, i.e., formulas hold in the model iff they hold in the original program.

All of the methods outlined above do not assume existence of *complete* specifications and, thus, can be applied to a variety of different systems. Our method takes advantage of a complete, fully-developed, SCR specification. On one hand, this means that CORD can be applied only to event-driven systems—the kinds of systems that can be effectively specified using SCR notation. And, of course, results of the analysis are only as good as the specification. On the other hand, our analysis makes sure that the design implements *exactly* the same transitions as specified. A number of approaches are similar to ours by nature. Equivalence checking in process algebras, e.g., as implemented in the Concurrency Workbench [20], checks that two levels of specifications (or a specification and an implementation) exhibit exactly the same behavior. COSPAN [44] uses L-automata to check two levels of specifications (or a specification and an implementation) for language containment properties. Both approaches have been used to formally verify communication protocols and circuit designs, as well as other hardware and software systems.

A group of researchers at the US Naval Research Lab recently undertook a verification effort similar to ours [7]. The goal of this work was to use a linear-time model-checker SPIN to check consistency of SCR requirements. Promela (an input language for SPIN) is a C-like language with nondeterministic guarded IF statements. Rather than using analysis to determine when events occurred, their implementation explicitly keeps track of *previous* values of variables. Then, an event $@\mathrm{T}(a)$ occurs when previous value of a is false and current value is true. This technique allows a more natural treatment of SCR events, but does not help in analyzing existing code.

8.2 Summary

We have defined a notion of consistency between an SCR-style requirements document and a detailed design. We have also presented a technique to check that this notion of consistency is satisfied, implemented in a tool called CORD. CORD creates a finite-state abstraction of a detailed design and checks it against a set of properties automatically generated from the requirements. The analysis takes a low-degree polynomial time.

We believe that our techniques are highly-scalable and envision CORD being used in a software development process in which SCR specifications are first written and checked for consistency and completeness. Then, a design is developed using our PDL. A detailed design is automatically verified for consistency with the requirements. Afterwards, a real implementation is written around the PDL statements. Consistency checking between code and design statements is assured through code reviews or via an automated procedure. This process gives a developer some assurance that the code implements the system behavior specified in the requirements. Finally, the implementation is thoroughly tested.

ACKNOWLEDGMENTS

The authors would like to thank Bill Pugh for finding fundamental errors in the original version of this paper. He proposed viewing our data-flow analysis as a constant propagation problem, which improved the clarity of our presentation. Connie Heitmeyer and Stuart Faulk helped us in understanding the SCR model. Discussions with Rich Gerber helped to shape this work. They are also very grateful for the anonymous referees thorough readings of previous versions of this paper. They gave many good technical suggestions and helped improve the presentation of this paper. This work was supported by the Air Force Office of Scientific Research under contract F49620-93-1-0034 and UTRS Connaught Fund award 72008220.

REFERENCES

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

[2] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore, "Software Requirements for the A-7E Aircraft," technical report, US Naval Research Lab., Mar. 1988.

[3] S. Anderson and G. Bruns, "The Formalization and Analysis of a Communications Protocols," *Formal Aspects of Computing,* vol. 6, pp. 92–112, 1994.

[4] G. Archinoff, R. Hohendorf, A. Wassyng, B. Quigley, and M. Borsch, "Verification of the Shutdown System Software at the Darlington Nuclear Generation Station," *Proc. Int'l Conf. Control and Instrumentation in Nuclear Installations,* May 1990.

[5] J. Atlee, "Automated Analysis of Software Requirements," PhD thesis, Univ. of Maryland, College Park, MD, Dec. 1992.

[6] J.M. Atlee and J. Gannon, "State-Based Model Checking of Event-Driven System Requirements," *IEEE Trans. Software Eng.,* pp. 22–40, vol. 19, no. 1, Jan. 1993.

[7] R. Bharadwaj and C. Heitmeyer, "Model Checking Complete Requirements Specifications Using Abstraction," *J. Automated Software Eng.,* vol. 6, no. 1, Jan. 1999.

[8] F.P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer,* pp. 10–19, Apr. 1987.

[9] S.H. Caine and E.K. Gordon, "PDL: A Tool for Software Design," *Proc. Nat'l Computer Conf.,* vol. 44, pp. 271–276, 1975.

[10] M. Chechik, *Automatic Analysis of Consistency between Requirements and Designs,* PhD thesis, Univ. of Maryland, College Park, MD, Dec. 1996.

[11] M. Chechik "$\mathrm{SC(R)}^3$: Towards Usability of Formal Methods," *Proc. CASCON'98,* pp. 177–191, Nov. 1998.

[12] M. Chechik and J. Gannon, "Automatic Verification of Requirements Implementations," *Proc. 1994 Int'l Symp. Software Testing and Analysis (ISSTA),* pp. 1–14, Aug. 1994.

[13] M. Chechik and J. Gannon, "Automatic Analysis of Consistency Between Implementations and Requirements: A Case Study," *Proc. 10th Ann. Conf. Computer Assurance,* pp. 123–131, June 1995.

[14] M. Chechik and V.S. Sudha, "Checking Consistency between Source Code and Annotations," CSRG Technical Report 373, Dept. of Computer Science, Univ. of Toronto, 1998.

[15] G. Chehaibar, H. Garavel, L. Mounier, N. Tawbi, and F. Zulian, "Specification and Verification of the Powerscale Bus Arbitration Protocol: An Industrial Experiment with LOTOS," *Proc. Joint Int'l Conf. Formal Description Techniques for Distributed Systems and Comm. Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV '96,* 1996.

[16] E. Clarke, S. German, and X. Zhao, "Verifying the SRT Division Algorithm Using Theorem Proving Techniques," *Proc. Eighth Int'l Conf. Computer-Aided Verification,* pp. 111–122, July 1996.

[17] E.M. Clarke O. Grumberg, and D.E. Long, "Model Checking and Abstraction," *IEEE Trans. Programming Languages and Systems,* vol. 19, no. 2, 1994.

[18] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems,* vol. 8, no. 2, pp. 244–263, Apr. 1986.

[19] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. Long, K.L. McMillan, and L.A. Ness, "Verification of the Futurebus+ Cache Coherence Protocol," *Formal Methods in System Design,* vol. 6, pp. 217–232, 1995.

[20] R. Cleaveland, J. Parrow, and B. Steffen, "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 1, pp. 36–72, Jan. 1993.

[21] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose, "MT: A Toolset for Specifying and Analyzing Real-time Systems," *Proc. Real-Time Systems Symp.,* Dec. 1993.

[22] D.L. Dill, A.J. Drexler, A.J. Hu, and C.H. Yang, "Protocol Verification as a Hardware Design Aid," *IEEE Int'l Conf. Computer Design: VLSI in Computers and Processors,* pp. 522–525, 1992.

[23] S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton, "Experience Using Lightweight Formal Methods for Requirements Modeling," *IEEE Trans. Software Eng.,* vol. 24, no. 1, pp. 4–14, Jan. 1998.

[24] D. Evans, "Static Detection of Dynamic Memory Errors," *Proc. PLDI'96: SIGPLAN Conf. Programming Language Design and Implementation,* May 1996.

[25] D. Evans, J. Guttag, J. Horning, and Y. Meng Tan, "LCLint: A Tool for Using Specifications to Check Code," *Proc. FSE'94: Foundations of Software Eng.,* 1994.

[26] S. Faulk, "State Determination in Hard-Embedded Systems," PhD thesis, Univ. of North Carolina, Chapel Hill, NC, 1989.

[27] O. Grumberg and D.E. Long, "Model Checking and Modular Verification," *Proc. CONCUR'91: Second Int'l Conf. Concurrency Theory,* 1991.

[28] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Trans. Software Eng.,* vol. 16, no. 4, pp. 403–414, Apr. 1990.

[29] M.P.E. Heimdahl and D.J. Keenan, "Generating Code from Hierarchical State-Based Requirements," *Proc. IEEE Int'l Symp. Requirements Eng. (RE'97),* Jan. 1997.

[30] C. Heitmeyer, B. Labaw, and D. Kiskis, "Consistency Checking of SCR-Style Requirements Specifications," *Proc. RE'95 Int'l Symp. Requirements Eng.,* Mar. 1995.

[31] C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw, "Automated Consistency Checking of Requirements Specifications," *ACM Trans. Software Eng. and Methodology,* vol. 5, no. 3, pp. 231–261, July 1996.

[32] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw, "SCR*: A Toolset for Specifying and Analyzing Requirements," *Proc. 10th Ann. Conf. Computer Assurance,* pp. 109–122, June 1995.

[33] K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," *IEEE Trans. Software Eng.,* vol. 6, no. 1, pp. 2–12, Jan. 1980.

[34] G.J. Holzmann, "Practical Methods for Formal Validation of SDL Specifications," *Computer Comm.,* vol. 15, no. 2, pp. 129–134, Mar. 1992.

[35] G. Holzmann, "Designing Executable Abstractions," *Proc. Second Workshop Formal Methods in Software Practice,* Mar. 1998.

[36] W.E. Howden, "Comments Analysis and Programming Errors," *IEEE Trans. Software Eng.,* vol. 16, no. 1, pp. 72–81, Jan. 1990.

[37] W.E. Howden and G.M. Shi, "Linear and Structural Event Sequence Analysis," *Proc. ISSTA'96,* May 1996.

[38] W.E. Howden and B. Wieand, "QDA—A Method for Systematic Informal Program Analysis," *IEEE Trans. Software Eng.,* vol. 20, no. 6, pp. 445–462, June 1994.

[39] D. Jackson, *Aspect: A Formal Specification Language for Detecting Bugs,* PhD thesis, MIT, Cambridge, Mass., June 1992.

[40] D. Jackson, "Abstract Analysis with Aspect," *Proc. 1993 Int'l Symp. Software Testing and Analysis (ISSTA),* pp. 19–27, June 1993.

[41] D. Jackson, "Aspect: Detecting Bugs with Abstract Dependences," *Trans. Software Eng. and Methodology,* vol. 4, no. 2, pp. 109–145, Apr. 1995.

[42] D. Jackson and C.A. Damon, "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *Proc. 1996 Int'l Symp. Software Testing and Analysis (ISSTA),* Jan. 1996.

[43] J. Mitchell, M. Mitchell, and U. Stern, "Automated Analysis of Cryptographic Protocols Using Murϕ," *Proc. IEEE Symp. Security and Privacy,* pp. 141–151, 1997.

[44] R.P. Kurshan, *Computer-Aided Verification of Coordinating Processes : The Automata-Theoretic Approach,* Princeton Series in Computer Science, Princeton Univ. Press, 1995.
[45] D.A. Lamb, *Software Engineering: Planning for Change,* Prentice-Hall, 1988.
[46] N.G. Leveson, M.P.E. Heimdahl, H. Hildreth, and J.D. Reese, "Requirements Specification for Process-Control Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 9, pp. 684–707, Sept. 1994.
[47] E. Madelaine and D. Vergamini, "Specification and Verification of a Sliding Window Protocol in LOTOS," *Proc. IFIP TC6/WG6.1 Fourth Int'l Conf. Formal Description Techniques for Distributed Systems and Comm. Protocols,* pp. 495–510, Nov. 1991.
[48] W. Marrero, E. Clarke, and S. Jha, "Model Checking for Security Protocols," *Proc. DIMACS Workshop Design and Formal Verification of Security Protocols,* 1997.
[49] K.L. McMillan and J. Schwalbe, "Formal Verification of the Gigamax Cache Consistency Protocol," *Shared Memory Multiprocessing,* N. Suzuki, ed., MIT Press, 1992.
[50] K.M. Olender and L.J. Osterweil, "Cesar: A Static Sequencing Constraint Analyzer," *Proc. ACM SIGSOFT '89 Third Symp. Software Testing, Analysis, and Verification (TAV3),* pp. 66–74, Dec. 1989.
[51] K.M. Olender and L.J. Osterweil, "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation," *IEEE Trans. Software Eng.,* vol. 16, no. 3, pp. 268–280, Mar. 1990.
[52] K.M. Olender and L.K. Osterweil, "Interprocedural Static Analysis of Sequencing Constraints," *ACM Trans. Software Eng. and Methodology,* vol. 1, no. 1, pp. 21–52, Jan. 1992.
[53] S. Owre, N. Shankar, and J. Rushby, "User Guide for the PVS Specification and Verification System(Draft)," technical report, Computer Science Lab, SRI Int'l, Menlo Park, Calif., 1993.
[54] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems," *Science of Computer Programming,* vol. 25, pp. 41–61, 1995.
[55] D.L. Parnas and Y. Wang, "Simulating the Behavior of Software Modules by Trace Rewriting Systems," *IEEE Trans. Software Eng.,* vol. 20, no. 10, pp. 750–759, 1994.
[56] D.E. Perry, "Version Control in the Inscape Environment," *Proc. Ninth Int'l Conf. Software Eng.,* pp. 142–149, 1987.
[57] D.E. Perry, "The Inscape Environment," *Proc. 11th Int'l Conf. Software Eng.,* pp. 2–12, May 1989.
[58] D.E. Perry, "The Logic of Propagation in The Inscape Environment," *Proc. Third Symp. Software Testing, Analysis, and Verification (TAV3),* pp. 114–121, Dec. 1989.
[59] N. Shankar, "Computer-Aided Computing," *Mathematics of Program Construction,* Bernhard Möller, ed., vol. 947, pp. 50–66, 1995.
[60] T. Sreemani and J.M. Atlee, "Feasibility of Model Checking Software Requirements: A Case Study," *Proc. 11th Ann. Conf. Computer Assurance (COMPASS '96),* June 1996.
[61] C. Vail, "Program Verification via Abstraction using Incremental Operational Specifications," PhD thesis, Univ. of California, San Diego, 1991.
[62] 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 Univ., Kingston, Ontario, May 1990.
[63] C.J. Walter, P. Lincoln, and N. Suri, "Formally Verified On-Line Diagnosis," *IEEE Trans. Software Eng.,* vol. 23, no. 11, pp. 684–721, Nov. 1997.
[64] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, and D. Notkin, "Model Checking Large Software Specifications," *IEEE Trans. Software Eng.,* vol. 24, no. 7, pp. 498–520, July 1998.
[65] M.N. Wegman and K. Zadeck, "Constant Propagation," *Optimization in Compilers,* F. Allen, B. Rosen, and K. Zadeck, eds., 1991.
[66] J. Wing and M. Vaziri-Farahani, "Model Checking Software Systems: A Case Study," *Proc. Third Symp. Foundations of Software Eng.,* pp. 128–139, Oct. 1995.

Marsha Chechik received the MS (1994) and PhD (1996) degrees in computer science from the University of Maryland, College Park. Since 1996, she has been an assistant professor in the Department of Computer Science at the University of Toronto. Her main research interests are in the application of formal methods to improve the quality of software. She is also interested in requirements engineering, static analysis and testing, and computer security. Dr. Chechik is a member of the ACM and the IEEE Computer Society.

John Gannon (1948-1999) received the bachelor's and master's degrees from Brown University and the PhD degree in computer science from the University of Toronto. He joined the University of Maryland faculty in 1975 and became the chairman of the Department of Computer Science in 1995. Dr. Gannon's research was in the specification, analysis, and testing of software systems. He was a member of the editorial board of *Transactions on Software Engineering* between 1992-1998. His public service included membership on the board of directors of the Computing Research Association, chairing the Graduate Record Examination Board computer science committee, and serving as a program officer with the National Science Foundation. Dr. Gannon was a fellow of the ACM and a senior member of the IEEE.

▷ **For further information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.