Automatic Analysis of Consistency between Requirements and Designs

Marsha Chechik
Department of Computer Science
University of Toronto
Toronto, ON M5S 3G4
chechik@cs.toronto.edu

John Gannon *
Computer Science Department
University of Maryland
College Park, MD 20774
gannon@cs.umd.edu

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. Most of the proposed techniques suffer from exponential explosion of the number of states in the generated state spaces. One way to cope with this explosion is to develop and apply "light-weight" formal methods that will attempt to achieve scalability by checking an *abstraction* of the system for only *some* non-trivial properties. This paper describes methods and tools for automatically analyzing the consistency of software requirements and detailed designs in low-degree polynomial time. Requirements describe systems as state machines with event-driven transitions using a specification language that is easy to read and scalable to large systems. We define requirements and detailed designs to be consistent 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 existing systems.

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

*This work was supported by the Air Force Office of Scientific Research under contract F49620-93-1-0034 and UTRS Connaught Fund award 72008220.

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 trying to verify that a program corresponds to its specifications were made by Hoare, Mills and Dijstra in 1960s. Their methods typically suffered from necessity to guess program invariants - an impossible task for all but the smallest programs. To overcome this problem, researches suggested verifying *some* properties. Theorem-proving[41] allows to check that a property is implied by the program. This approach is very effective, although requires considerable skill and time investment. Another approach is to check properties via state exploration (model-checking[13]). This approach is very effective for verifying hardware and distributed systems, like protocols. However, the size of the state-space grows exponentially to the number of variables in the problem. Software problems are typically too large to be analyzed using model-checking. The first attempt to model-check requirements was made by Atlee and Gannon[4]. Since then, model-checking has been often used in requirements engineering. Programs, however, cannot usually be verified directly, and various researchers have been proposing checking *abstractions* of programs[51, 33, 26]. 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 ([2, 23, 37, 20]) which respond to events in their environment. Designs are often expressed in a program design language (PDL)[7] 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[36]. 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 this paper, we define a notation which can be used as a PDL and describe a prototype tool, called CORD[9, 10], 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

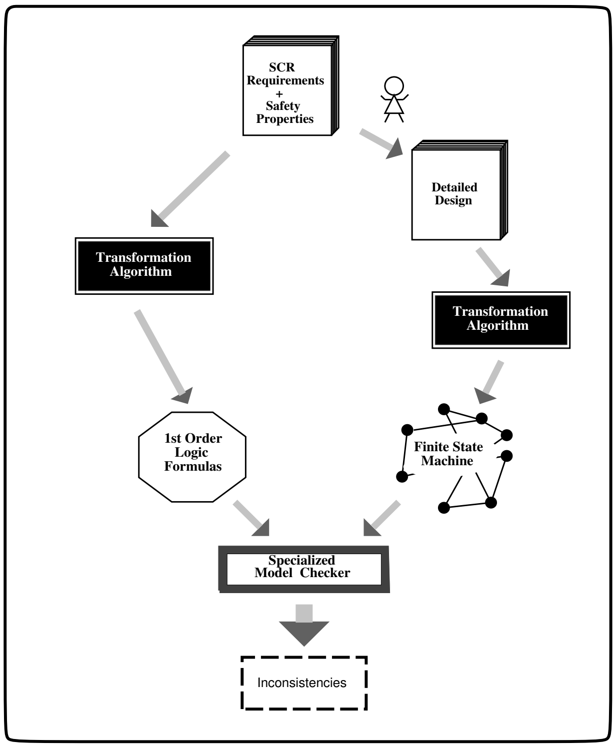

Figure 1: Algorithm to solve the problem.

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 Figure 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 FSM is a model of the formulas.

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 to checking automatically-generated properties. Section 7 presents a case study during which we analyzed an existing detailed design of a water-level monitoring system[49] with respect to its SCR requirements. Finally, Section 8 discusses limitations of our approach and compares it to related work.

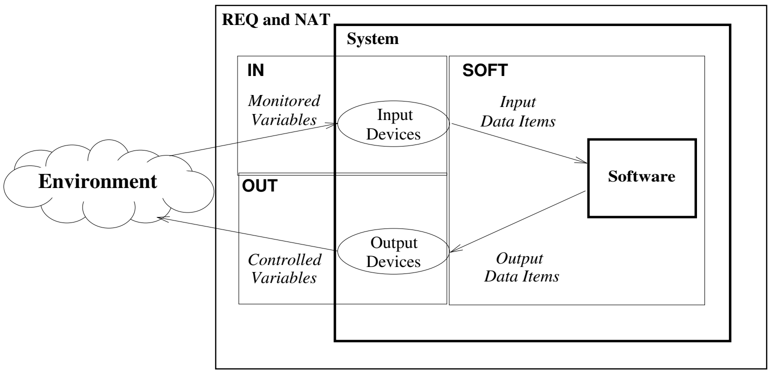

Figure 2: Four Variable Model.

2 Requirements Notation

The SCR requirements notation was developed by a research group at the Naval Research Laboratory as part of the Software Cost Reduction project [2, 25]. 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[3, 23, 18, 42, 49], some of which[23, 43] became bases of tools automating consistency and completeness checks[22] and simulations[15, 44] of requirements. In this section, we describe SCR behavioral requirements and environmental assumptions.

2.1 Four-Variable Model

The SCR requirements notation is based on the Four Variable Model[42, 43]. This model describes required system functions, timing, and accuracy as a set of mathematical relations on four sets of variables: monitored and controlled variables, and input and output data items. Figure 2 illustrates the approach. The environment contains *monitored* and *controlled* variables. Monitored variables are quantities that influence system behavior, and controlled variables are quantities that the system regulates. The required behavior of the system is specified as a black box, with monitored and controlled variable as its inputs and outputs, respectively. Two relations, NAT and REQ, from monitored to controlled variables, describe the system's behavior, specifying constraints imposed by nature and the system, respectively. REQ describes the ideal behavior of a system. Input devices (i.e., sensors) measuring monitored variables and output devices (i.e., actuators) regulating controlled variables approximate the values of the variables. The relations IN and OUT specify the accuracies of these mappings: IN from monitored variables to input data items, and OUT from output data items to controlled variables. SOFT is the relation between input and output data items.

An SCR requirements document can describe both the system and the software requirements by specifying REQ and SOFT, respectively. The rest of this paper is concerned only with software requirements. Thus we use the term *monitored variables* to describe monitored variables or input data items, and *controlled variables* to describe controlled variables or output data items. The SCR model is based on event-driven state-transition systems. We assume that time is discrete (because a sensor samples data periodically). 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 input and the output data items. This relation is called SOFTREQ[24]: in the Four Variable Model, SOFT represents the software implementation, whereas SOFTREQ represents the software requirements specification. $\mathcal{E}$ is a subset of NAT that represents constraints on input data items.

2.2 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 conditions are boolean variables. So, for the purposes of specifying and checking the system, we assume that all conditions are boolean.

A machine's behavior is defined and controlled by changes to the values of conditions, called *events*. We assume that conditions can change value at most once during a single time unit. Events are detectable only at the point in time in which they occur. For example, events @T(a) and @F(a) represent a condition *a becoming true* and *becoming false*, respectively. a is the event's *triggering condition*. All events in the form @T(a) or @F(a) are called *primitive events*.

Compound events can be specified using boolean operators, connecting primitive events or multiple conditions. Some formal semantics, e.g. [23], exclude simultaneous events. We assume that simultaneous events can only occur between variables related by environmental assumptions, defined below. A *conditioned event* @T(a) WHEN [b] describes the point of time when *a becomes true* while *b remains true*. Since time is discrete, we can reason about the current and the previous state. If "current" is the state of the system at time t, then "previous" is the state at time $t - \epsilon$ for some well-defined ϵ - a unit of measurement of time. The logical statement represented by a conditioned event is defined by[24]:

$$@\mathrm{T}(a) \text{ WHEN } [b] = \neg\, a \wedge a\prime \wedge b,$$

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. b is called the event's WHEN *condition*.

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. However, 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. An SCR requirements document contains the specification of one or more mode classes. At all times, the system is in exactly one mode of each mode class. Each mode class specifies one aspect of the system's behavior, and the system's global behavior is defined to be the composition of the specification's mode classes. Transitions between pairs of modes are activated by conditioned events. If a conditioned event can trigger two or more transitions from the same mode, then the mode class is non-deterministic. 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 mode. We can think of a mode class as an enumerated type whose values are modes of that mode class.

SCR requirements use tables to define the values of controlled variables and mode transitions. There is one table for each controlled variable and for each mode class. Each entry in a condition table defines a value of a controlled variable as a function of a system's modes and events. Each entry in a mode transition table maps a mode and an event to another mode in the same mode class[1]. Controlled variables are outputs of the system, and we assume that other variables cannot depend of them. Thus, controlled variables cannot be part of an event.

Table 1 shows a mode transition table for a simplified Water-Level Monitoring System(SWLMS). A switch controls whether the system is on or off. If the system is on and its sensors detect too much (too little) water, a pump is turned on for a fixed period to remove (add) some water. If the sensor or the pump fails, the system enters an error state. This simplified version of the system has no error recovery, so there are no transitions from the error state. This system has one mode class MC with modes Off, Operating, and Error; four monitored variables SwitchOn, PumpFail, TooHigh, and TooLow; and a single controlled variables PumpOn. 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

[1]In addition to controlled and monitored variables, SCR requirements often contain terms - results of intermediate computation. We do not currently handle terms, although our analysis technique can be easily augmented to do so. SCR also contains *condition* rather than *event* tables which allow the specification of a value of a variable or a term w.r.t. a set of conditions. In most cases, condition tables can be textually translated into event tables.

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 "don't care" values, although some values can be inferred from environmental assumptions about variables (see below).

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

Table 1: Mode transition table for mode class MC of SWLMS.

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

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

Initial: false

Table 2: Event table for controlled variable PumpOn.

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;
T: $R \rightarrow 2^V$ is a mapping of variables to ranges of their values:
$\forall\ r_j \in M \cup C,\ T(r_j) = \{true,\ false\}$
$\forall\ md_j \in MD,\ T(md_j)=\{m_{j,1},\ m_{j,2}, \ldots,\ m_{j,n}\}$, *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
$I : V^{MD \cup M} \rightarrow V^{MD \cup C}$ 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:

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

2.3 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[3]. 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–>> 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 non-strict 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–>> ¬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.

3 Detailed Design

A Program Design Language (PDL)[7] 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[6]. 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, i.e., with IN and OUT relations (see Section 2 and [43]).

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[7] 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[29, 48].

For sequential designs, we define three 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.

- An *Update annotation* assigns values to variables, identifying points at which the program changes its state.

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

- 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, the latter start with @@. For complete syntax of PDL, see [8].

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 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

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

Figure 3: Design of SWLMS.

monitors, tests their values and then decides to change either its mode or the values of several 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[21].

It is often necessary to hand-optimize designs for better performance or to reduce redundancy by "factoring out" common actions for different states. Figure 3 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[27, 29], where the code is annotated with comments specifying user assertions (representing known information) and hypothesis (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. Simple automatic tests, checking consistency between annotations and source code, can be performed[11].

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 the $\mathcal{B}$.*

This is a very restricted definition. Typically, artifacts are considered consistent with requirements when they implement at least what is specified. SCR was developed to specify high-assurance systems and 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 2.2 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 considered in determining if an event caused a mode change: the state in which the monitored variable had its original value, the state in which the monitored variable was assigned a new value, and the state in which the mode change occurred. Generally, we do not know the value assigned to a monitored variable until after the outcome of a test. Thus, monitored variables may have either the value true or the value false in a state; the exact value not being known until an arc representing a successful test outcome is traversed. As the result, we define formulas as being true on arcs as well as in states. A transition between m_i and m_j triggered by @T(a) WHEN [b] is formalized as

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

where a condition represents its value on the previous edge, a primed condition represents its value on the current edge, and the double-primed condition represents its value in the adjacent state. Figure 4 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 pred(n) and succ(n) returning a list of all predecessors and successors of n, respectively (this list can be empty). We express all properties using statements about three-state sequences $(p \in \text{pred}(n), n, s \in \text{succ(n)})$. For example, a property "there exists a transition from

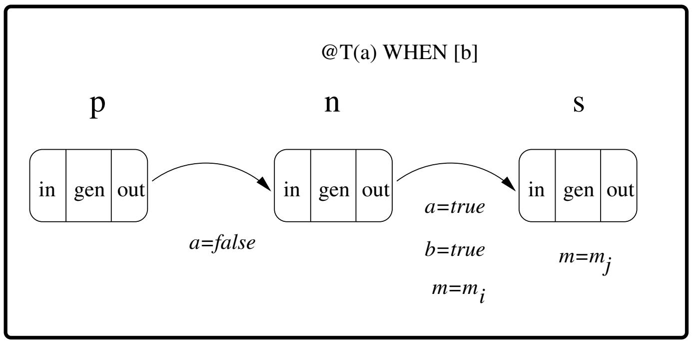

Figure 4: Pictorial representation for a mode transition.

$m = m_i$ to $m = m_j$ on @T(a) WHEN [b]" is expressed as

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

For brevity, we will write (p, n, s) to indicate $(p \in \text{pred}(n), n, s \in \text{succ(n)})$. Properties quantified on all of (p, n, s) are considered vacuously true when pred(n) or succ(n) are empty. A number of properties generated from SCR have the notion of an event in them. We say that an event @T(a), where a is a boolean variable, has occurred on a 3-state sequence (p, n, s), i.e. $(p, n, s) \models$ @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 MC to be in mode Off in its next state is if MC is currently in Off, or if a transition from mode Operating occurs in response to an event @F(SwitchOn) WHEN ¬PumpFail. This property was obtained by composing the rows of the MC mode transition table which have Off in their right columns (in this case, only row three). We write this property as

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

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

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

Figure 5: OLT properties for SWLMS.

part of our notion of consistency and are called "only legal transitions" (OLT) properties. OLT properties generated from requirements of SWLMS are shown in Figure 5. Property P_1 was generated from row three of Table 1; P_2 from row one; and P_3 from a composition of rows two and four. Properties P_4 and P_5 were generated for the controlled variable PumpOn from Table 2. There are two OLT properties generated for each controlled variable, reflecting changes of value to false (P_4) and to true (P_5).

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

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

Such properties ensure that all transitions specified in the requirements (potentially) 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 Figure 6. 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

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

Figure 6: ALT properties for SWLMS.

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 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"). Figure 7 depicts the steps needed to create and check an abstraction:

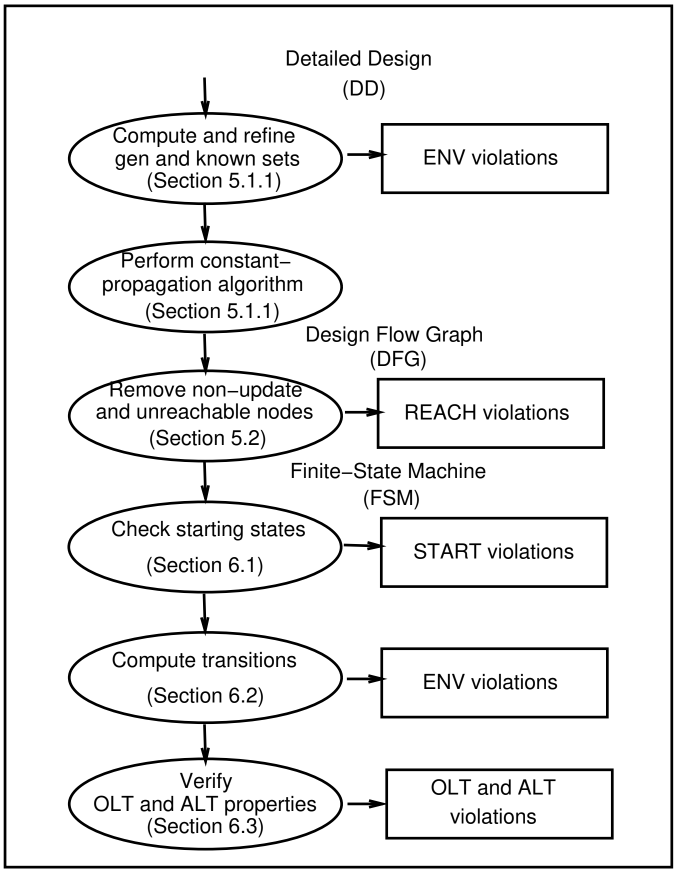

Figure 7: Analysis roadmap.

- 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.
- 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*
v_{r_j} is a set of values associated with the variable r_j at the node n, and
$\forall\ r_j \in R,\ v_{r_j} \in 2^{T(r_j)}$ (v_{r_j} is a set consisting of values in the domain of r_j).

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 all paths leading to this node, the variable is true.
{false}	On all paths leading to this node, the variable is false.
{true, false}	On some path the variable is true, and on some other it is false.

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 $\mathrm{repl}(r_j, \mu, s)$ to replace the current value of r_j by μ in s:

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

and a special state EMPTY:

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

Operations on states include "$\sqcup$" (union), "$\sqcap$" (intersection), "=" (equality), "$\sqsupset$" (superset), "$-$" (difference) and "$\sqsupseteq$" (superset or equal to). States form a complete $\sqcup$-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[50, 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 contain a disjunction of several clauses. For these nodes, `known` sets are lists of several states, one for each clause. Operations on states are trivially extended to handle sets of system states. In the examples below, we omit variable-value pairs in which the value is {}. For an annotation @@Update MC=Operating & PumpFail=true,

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

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

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

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

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

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 [8].

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 *meet* operator for combining information coming into the node is $\sqcup$, so

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

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

Annotation at node n	Transfer function
Initial	`out`(n) = `gen`(n)
Update, Read	`out`(n) = repl(`in`(n), `gen`(n))
Assert (single disjunct)	`out`(n) = `in`(n) $\sqcap$ `known`(n)
none	`out`(n) = `in`(n)

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

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

Our framework is strictly monotonic, i.e., `in` and `out` sets on an iteration of our algorithm have more values (or at least as many) for each variable as `in` and `out` sets on the previous iteration. Since all variables in R have a finite number of abstract values, our 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 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 (Figure 3). Figure 8 shows a fragment of this DFG corresponding to lines 38-45 of the SWLMS design. `gen` and `known` sets computed at each node are shown in bold font in this figure; variables with values {} 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,

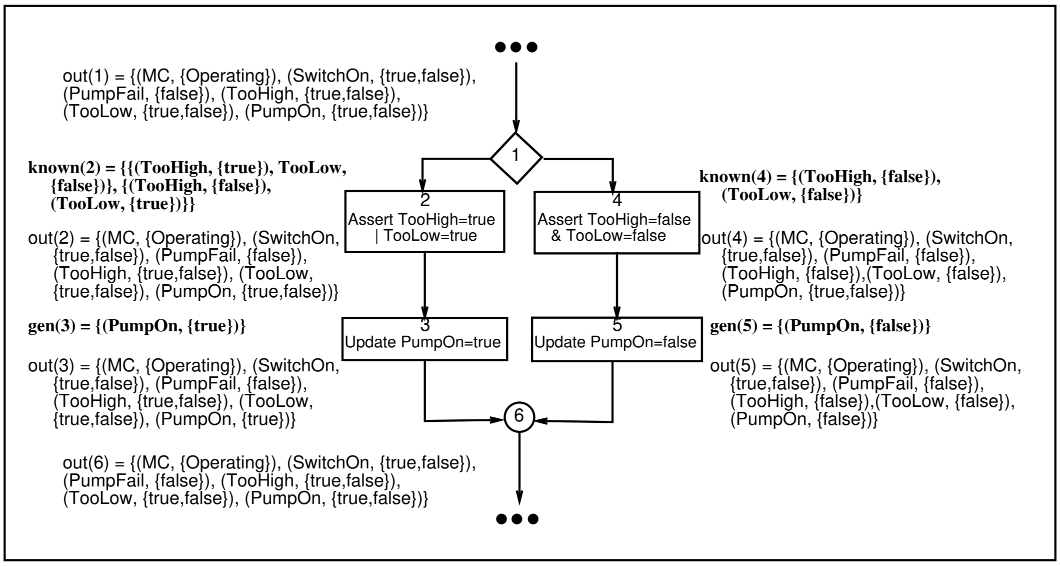

Figure 8: A fragment of DFG of SWLMS.

TooHigh is false, and vice versa. The resulting `known` set consists of two disjuncts, one for each possibility. The Update on that branch (node 3) generates the value {true} for PumpOn. The Assert on the right branch (node 4) generates the value {false} for TooLow and TooHigh, and the following Update (node 5) generates the value {false} for PumpOn. We did not include `in` sets in Figure 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 the possible values for each variable in the `out` sets of the predecessor nodes (nodes 3 and 5).

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 contains just 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*

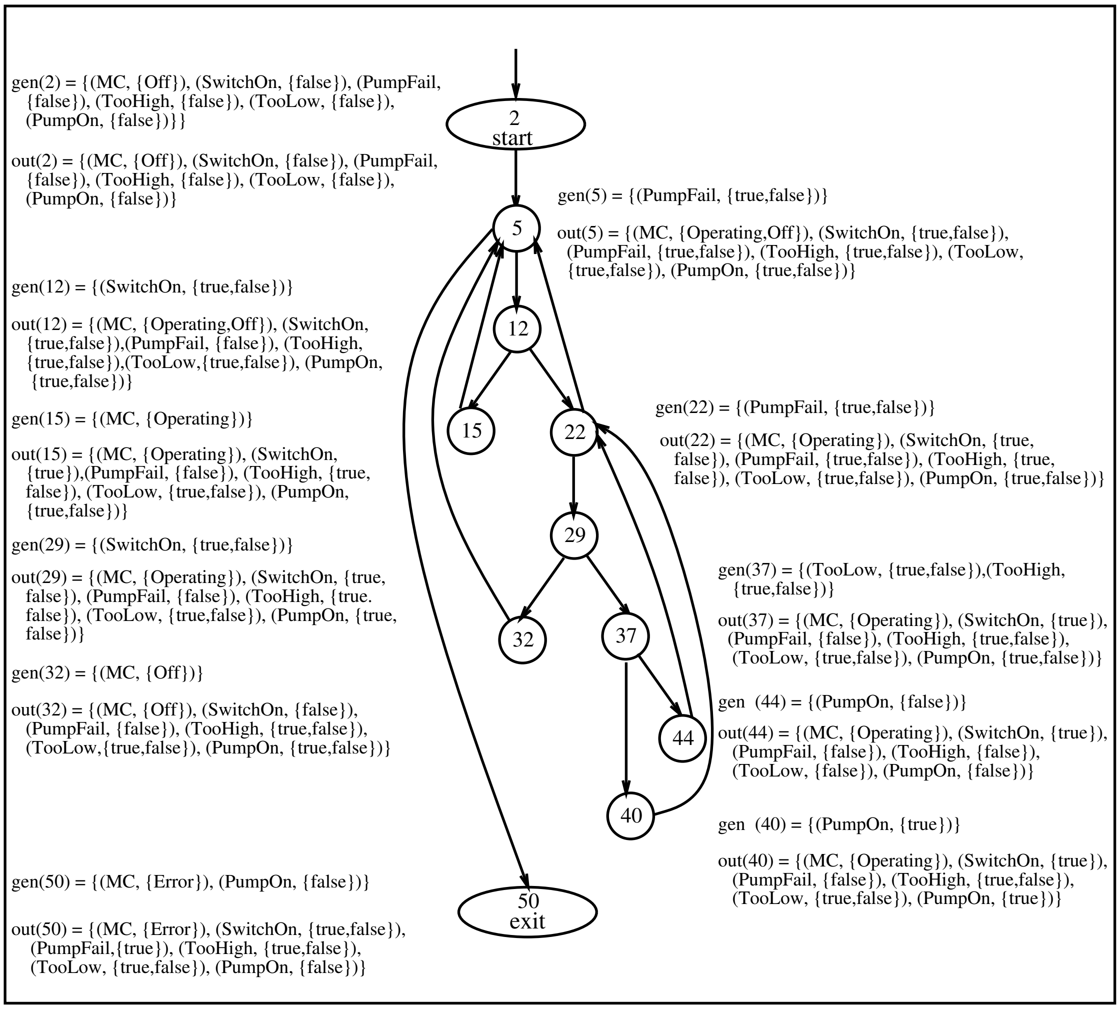

Figure 9: Finite-state abstraction.

A is a set of labels;
$S = U \cup I$ is a finite set of nodes;
L: $S \to A$ is a function associating each node with a label;
$N \subseteq S \times S$ is a transition relation.
N is obtained by connecting nodes of S s.t. there is an Update&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 (A) annotations, and connect all predecessors of a removed node to the node's successors. Let $S = I \cup U \cup A$ 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):

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

If this 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 $|V|$, and the number of nodes in the resulting FSM is bounded above by $|S|$, the number of state changes in the design.

Figure 9 shows the FSM created for the SWLMS design in Figure 3, 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 40 and 44 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[13] process states whose variables have scalar values. Transforming our FSM to correspond to an acceptable input for an existing model-checker would have resulted in an exponential increase in the number of nodes in the FSM. So, we developed our own technique to verify properties.

The semantics of an SCR event, given in Section 2.2, 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 a formula f *holds in a state* s if f holds in $\mathtt{out}(s)$, i.e., $\mathtt{out}(s) \models f$. Also, we say that f holds on an edge between nodes n and s if $(\mathtt{out}(n) \sqcap \mathtt{in}(s)) \models f$. By construction of the FSM, if a formula holds on exit of the node but does not hold on entrance, then it has been generated at this node, i.e. for a node n,

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

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*.

Definition 6.1 *A property is checked pessimistically if all of its violations in the design are detected, but the analysis incorrectly identifies violations at points in the design at which* *the property actually holds. A property is checked optimistically if all detected violations are present in the design, but the analysis is unable to find all violations of the property.*

6.1 Checking Starting States

We use exact checking 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 Figure 9). `gen`(2) is

$$\begin{array}{c}\{(\mathrm{MC}, \{\mathrm{Off}\}), (\mathrm{SwitchOn}, \{\mathrm{false}\}), (\mathrm{PumpFail}, \{\mathrm{false}\}),\} \\ (\mathrm{TooHigh}, \{\mathrm{false}\}), (\mathrm{TooLow}, \{\mathrm{false}\}), (\mathrm{PumpOn}, \{\mathrm{false}\})\end{array}$$

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 does not invalidate the verification of automatically-generated properties. For OLT properties, we want to check if all transitions in the design are present in the requirements, and overestimating the transitions might cause the tool to report some false negatives, which is a correct treatment of pessimistic analysis. For ALT properties, we want to check if all transitions in the requirements are present in the design, and overestimating the transitions might cause the tool to report some false positives, which is a correct treatment of optimistic analysis. Table 3 summarizes our analysis. For example, when a transition in the requirements is not implemented in the design and is not computed by our tool (row 4), then the tool reports a violation of 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

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

Table 3: Results of analysis.

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 `in`(s) and the `out` sets of the predecessors of n. The algorithm to compute transitions leading to a node s is shown in Figure 10. The output of this algorithm is a set of transitions $\{(n,$ `to`, `from`, `trigger`, `when`$)\}$, where

- n is a predecessor of s;
- `to` and `from` are in the form $r_j = v\prime_{r_j}$ and $r_j = v_{r_j}$, respectively, indicating the change of value of r_j from v_{r_j} to $v\prime_{r_j}$ 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.

The function 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, returning a disjunction of @T(r_j), @F(r_j) or none(r_j). The last value indicates that no event involving r_j has occurred between the two nodes. We use "none" as a symbolic constant to indicate that some variable did not change its value. If s_1 is a node containing the Initial annotation, events returns none(r_j).

Algorithm 1

Inputs: Finite state machine (FSM) and
node s for which to compute the transitions.
Outputs: A set of transitions to node s. Each transition is in the form (n, `to`, `from`, `trigger`, `when`), where $n \in \text{pred}(s)$ and `to` $\neq$ `from`.
Algorithm:

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

Figure 10: Algorithm for computing transitions.

Otherwise, for a boolean r_j, events(r_j, s_1, s_2) is defined as follows:

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

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

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

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

In our SWLMS example, consider calculating the event which causes MC to be set to Error at node 50 of the FSM in Figure 9. Figure 11 shows a fragment of this FSM. `when` is `out`(5) $\sqcap$ `in`(50), namely,

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

`gen`(5) (node 5 is a predecessor of 50) is (PumpFail, {true,false}), so in order to compute triggs, we call events for PumpFail:

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

`gen`(5) $\sqcap$ `when` = {(PumpFail, {true})}, which indicates that the designer assumed that PumpFail is true in node 50. Since PumpFail is {false} in nodes 2, 15 and 32, and {true,false} in node 22,

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

The events yields @T(PumpFail) $\vee$ 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`(50)). On the first iteration of the loop, `to` is (MC=Error) and `from` is (MC=Off); on the next iteration, `from` becomes (MC=Operating). The result is four transitions, I_1-I_4, for each combination of triggers and starting values of MC. These transitions are shown in Figure 12.

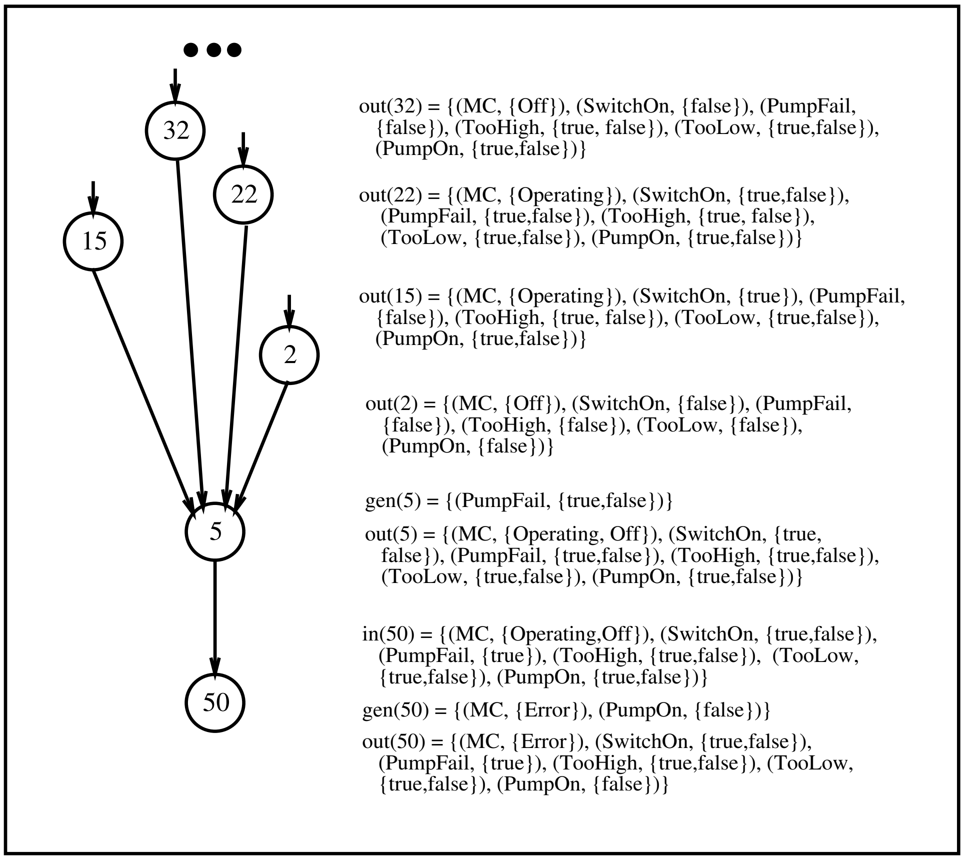

Figure 11: Calculating transitions for SWLMS.

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 algorithms shown below appear in [8].

Verification of OLT Properties

OLT properties have the general form

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

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

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, {})}]

Figure 12: Transitions discovered for node 50.

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 Figure 13.

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., @T(a)). Our algorithm to compute transitions also results in a conjunction of simple Triggering conditions. To check that $\texttt{trigger} \rightarrow \texttt{trcond}_j$, we check that each conjunct in $\texttt{trcond}_j$ is present in `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, PumpOn = true is treated as (PumpOn, {true}), and $\neg$(MC = Operating) means that MC can be either Off or Error and is treated as (MC, {Off, Error}). If a variable $r_k \in R$ is not part of $\texttt{whcond}_j$, then it was specified as a "don't care" condition in the tables. The value for this variable is considered to be a set of all of its attainable values, i.e., it is treated as $(r_k, T(r_k))$. One of the OLT properties for SWLMS is

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

Algorithm 2

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.
      }
   }
```

Figure 13: Algorithm for verifying OLT properties.

In this property,

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

For all nodes other than 50, MC$\neq$Error, so P_3 holds vacuously. The transitions generated for node 50 are I_1-I_4, as shown in Figure 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 `trigger`s for transitions I_2 and I_4 are NONE, indicating that no Triggering conditions were found, so CORD reports an error message:

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

Verification of ALT Properties

All ALT properties have the general form

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

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 Figure 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, $v_{new} \neq v_{old}$.

P_7 is an ALT property for SWLMS:

$$\begin{aligned} P_7 \quad = \quad & \exists(p,n,s),\ (s \models (\text{MC=Error})) \ \wedge\ ((p,n,s) \models \text{@T(PumpFail)}) \ \wedge \\ & ((n,s) \models (\text{MC=Off})) \end{aligned}$$

In this property,

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

Transitions generated for node 50, which is reachable from the start state, enable the algorithm to mark P_7 as satisfied: the `from` part of I_1 is MC=Off, the `to` part is MC=Error, `trcond` $\rightarrow$ `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

Algorithm 3

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 s0 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 Pi s.t.
            (r = vnew) = to AND
            (r = vold) = from AND
            trcond → trigger AND
            whcond ⊑ when
        Then mark Pi
Report all unmarked ALT properties
```

Figure 14: Algorithm for verifying ALT properties.

SCR requirements notation and subsequently implemented[49]. 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 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 [49]. WLMS has two mode classes, Normal and Failure, whose modes are described in Table 4. 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 5. 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 2 mode classes with 3 and 4 modes each, 18 monitored and 7 controlled variables. The number of potential states for this system is

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

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

Table 4: WLMS Modes.

To build a design, we reverse engineered an existing implementation of the WLMS, originally consisting of roughly 1300 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 [8]. 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 annotations right, and thus a quick response was appreciated.

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

Table 5: Mode transition table for mode class Normal.

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

Table 6: Results of analyzing the WLMS.

After we eliminated annotation errors, CORD reported a number of inconsistencies between the requirements and the design (see Table 6). Messages produced by CORD are organized in two columns: "Messages" indicates the actual number of messages, and "Violations" indicates the number of invalid mode transitions or changes of values of controlled variables. These 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. 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 though ALT properties were checked optimistically, we found that a large portion of the specification was not implemented.

8 Conclusion and Related Work

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[45, 46, 47], 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.

LCLint[17] 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 [16] 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)[27, 29] 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[28] 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)[38, 39, 40]. 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[30, 31, 32] 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. [12] 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 $\forall$CTL*[19], 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[14], checks that two levels of specifications (or a specification and an implementation) exhibit exactly the same behavior. COSPAN[35, 34] 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 Naval Research Lab recently undertook a verification effort very similar to ours [5]. 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 non-deterministic 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 @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

We 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 clarity of presentation. Connie Heitmeyer and Stuart Faulk helped us in understanding the SCR model. Discussions with Rich Gerber helped to shape this work. Finally, we are grateful to anonymous referees for thorough reading of the previous version of this paper and for various technical suggestions.

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, Naval Research Laboratory, March 1988.

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

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

[5] Ramesh Bharadwaj and Connie Heitmeyer. "Verifying SCR Requirements Specifications Using State Exploration". In *Proceedings of 1st ACM SIGPLAN Workshop on the Automated Analysis of Software*, Paris, France, January 1997.

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

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

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

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

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

[11] M. Chechik and V.S. Sudha. "Using SCR to Comment and Verify Code for Event-Driven Systems". Technical report, University of Toronto, November 1997.

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

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

[14] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. "The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems". *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[15] P. Clements, C. Heitmeyer, B. Labaw, and A. Rose. "MT: A Toolset for Specifying and Analyzing Real-time Systems". In *Proceedings of Real-Time Systems Symposium*, Raleigh, NC, December 1993.

[16] David Evans. "Static Detection of Dynamic Memory Errors". In *Proceedings of PLDI'96: SIGPLAN Conference on Programming Language Design and Implementation*, May 1996.

[17] David Evans, John Guttag, James Horning, and Yang Meng Tan. "LCLint: A Tool for Using Specifications to Check Code". In *Proceedings of FSE'94: Foundations of Software Engineering*, 1994.

[18] Stuart Faulk. *"State Determination in Hard-Embedded Systems"*. PhD thesis, University of North Carolina, Chapel Hill, 1989.

[19] O. Grumberg and D.E. Long. "Model Checking and Modular Verification". In *Proceedings of CONCUR'91: 2nd International Conference on Concurrency Theory*, 1991.

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

[21] Mats P.E. Heimdahl and David J. Keenan. "Generating Code from Hierarchical State-Based Requirements". In *Proceedings of the IEEE International Symposium on Requirements Engineering (RE'97)*, Annapolis, Maryland, January 1997.

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

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

[24] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. "SCR*: A Toolset for Specifying and Analyzing Requirements". In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 109–122, June 1995.

[25] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications". *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[26] Gerard Holzmann. Keynote address: "Designing Executable Abstractions". In *Proceedings of 2nd Workshop on Formal Methods in Software Practice*, March 1998.

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

[28] W.E. Howden and G.M. Shi. "Linear and Structural Event Sequence Analysis". In *Proceedings of ISSTA'96*, May 1996.

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

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

[31] Daniel Jackson. "Abstract Analysis with Aspect". In *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 19–27, June 1993.

[32] Daniel Jackson. "Aspect: Detecting Bugs with Abstract Dependences". *Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

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

[34] R. P. Kurshan. "COSPAN". In *Proceedings of CAV'96*, 1996.

[35] R.P. Kurshan. "Automata-Theoretic Verification of Coordinating Processes". Technical report, AT&T Bell Laboratories, August 1992.

[36] D.A. Lamb. *Software Engineering: Planning for Change*. Prentice-Hall, 1988.

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

[38] Kurt M. Olender and Leon J. Osterweil. "Cesar: A Static Sequencing Constraint Analyzer". In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 66–74, December 1989.

[39] Kurt M. Olender and Leon J. Osterweil. "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation". *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[40] Kurt M. Olender and Leon K. Osterweil. "Interprocedural Static Analysis of Sequencing Constraints". *ACM Transactions of Software Engineering and Methodology*, 1(1):21–52, January 1992.

[41] S. Owre, N. Shankar, and J. Rushby. "User Guide for the PVS Specification and Verification System (Draft)". Technical report, Computer Science Lab, SRI International, Menlo Park, CA, 1993.

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

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

[44] D.L. Parnas and Y. Wang. "Simulating the Behavior of Software Modules by Trace Rewriting Systems". *IEEE Transactions on Software Engineering*, 20(10):750–759, 1994.

[45] Dewayne E. Perry. "Version Control in the Inscape Environment". In *??*, pages 142–149, 1987.

[46] Dewayne E. Perry. "The Inscape Environment.". In *Proceedings of the 11th International Conference on Software Engineering*, pages 60–68, Pittsburgh PA, May 1989.

[47] Dewayne E. Perry. "The Logic of Propagation in The Inscape Environment". In *Proceedings of the 3rd Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 114–121, Key West, Florida, December 1989.

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

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

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

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