AUTOMATIC ANALYSIS OF CONSISTENCY BETWEEN IMPLEMENTATIONS AND REQUIREMENTS *

Marsha Chechik John Gannon

Computer Science Department
University of Maryland
College Park, MD 20742
{chechik,gannon}@cs.umd.edu

Date: July 28, 1995

Abstract

Formal methods like model checking can be used to demonstrate that safety properties of embedded systems are enforced by the system's requirements. Unfortunately, proving these properties provides no guarantee that they will be preserved in an implementation of the system. We have developed a tool, called Analyzer, which helps discover instances of inconsistency and incompleteness in implementations with respect to requirements.

Analyzer uses requirements information to automatically generate properties which ensure that required state transitions appear in a model of an implementation. A model is created through abstract interpretation of an implementation annotated with assertions about values of state variables which appear in requirements. Analyzer determines if the model satisfies both automatically-generated and user-specified safety properties.

This paper presents a description of our implementation of Analyzer and our experience in applying it to a small but realistic problem.

---

* *This research is supported in part by the Air Force Office of Scientific Research under contract F49620-93-1-0034.*

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. Automated techniques can be used to derive a finite-state representation of a set of requirements, and determine if it is a model for system safety properties expressed as temporal logic assertions[4]. We present a complementary technique which automatically compares properties derived from a set of requirements with a finite state representation of its implementation.

Requirements for embedded systems often describe a system as a set of concurrently executing state machines ([2, 15, 24, 13]) which respond to events in their environment. An implementation is consistent with its requirements if the implementation's state transitions are enabled by the same events as those of the requirements, all the requirement's state transitions appear in the implementation, and the requirement's safety properties hold in the implementation.

Generally, these properties are judged during code inspections conducted by teams of reviewers. Reviewers successfully discover 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 of the system hold. We developed a prototype tool, called Analyzer[5, 6], which automatically determines if an implementation is consistent with its requirements. The inputs to the tool are a requirements specification and a C source program annotated with comments describing the values of variables which appear in the requirements. As Figure 1 illustrates, our tool automatically generates a set of checks equivalent to temporal logic formulas to ensure the consistency of an implementation with its requirements. The implementation is abstracted into a finite-state machine (FSM), and a special-purpose model checking algorithm determines if the FSM is a model of the properties.

In this paper we describe the tool and show results of a case study in which we analyzed an implementation of a water-level monitoring system (WLMS) [31] and discovered several latent errors. The rest of the paper is organized as follows: Section 2 presents our requirements specification format. In Section 3, we describe the types of global system properties which are automatically generated from the specifications. Section 4 discusses the process of building a finite-state model of the implementation. In Section 5, we present an algorithm to check automatically-generated and user-defined system properties. Section 6 discusses our approach to processing programs with multiple procedures. Section 7

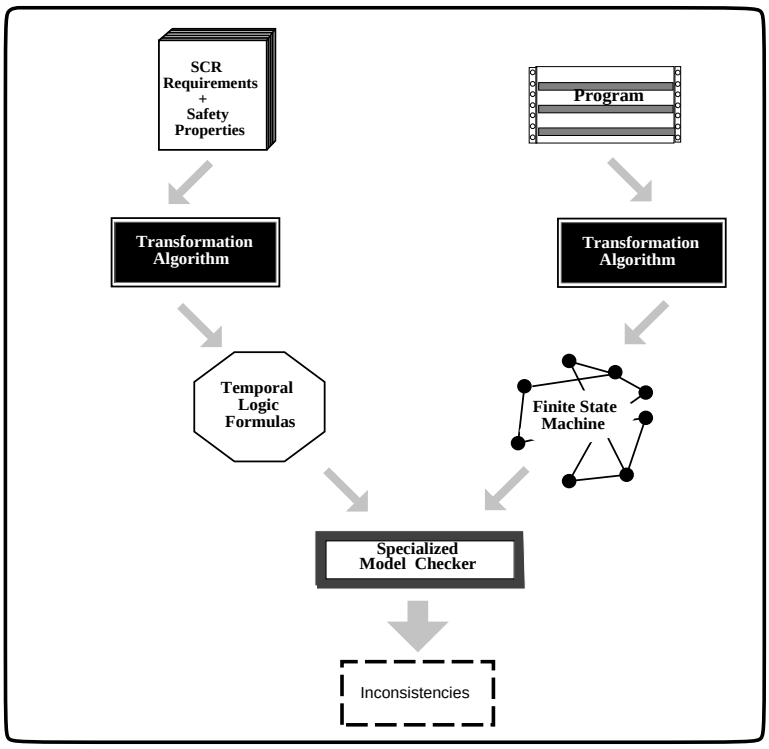

Figure 1: Algorithm to solve the problem.

presents the WLMS case study. Finally, in Section 8, we compare the approach taken in Analyzer with related work and discuss limitations of our approach.

2 Requirements Notation

This section describes the requirements specification format. Software Cost Reduction (SCR) requirements[2, 14, 16, 17] model a system as a set of concurrently executing state machines, where each machine interacts with its environment's state variables. Each state machine represents one *mode class*, whose states are *modes* and whose transitions occur in response to *events*. Modes within a mode class are disjoint, and the system is in exactly one mode of each mode class at all times. Changes to its *monitored* state variables may cause the system to change its mode or to alter values of its *controlled* variables (see Figure 2). A condition is a predicate on monitored or mode class variables. An event occurs when the value of a condition changes. For example, @T(A) WHEN [B] occurs if a condition A changes its value from False to True while a condition B is True. We refer to A and B as Triggering and When conditions, respectively.

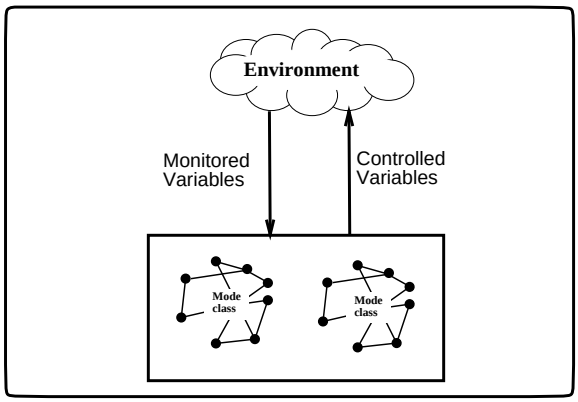

Figure 2: SCR Requirements notation.

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 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.

Table 1 shows a mode transition table for a simple system. We use this example throughout the paper, referring to it as the Simple System Example (SSE). This system has one mode class M with modes M1, M2, and M3; three monitored variables A, B, and C; and two controlled variables D and E. Mode class M starts in mode M1, and all variables except B and E are initially False. M transitions from M1 to M2 when C becomes True (indicated by "@T") while B is True (indicated by "t"), and transitions to M3 when B becomes False ("@F"). Entries marked by "–" are generally considered "don't care" values, although some values can be inferred from relationships between variables (see below).

Current Mode	A	B	C	**New Mode**
M1	–	t	@T	M2
	–	@F	–	M3
M2	@T	@T	–	M1
	t	@F	–	M3
M3	@T	–	–	M1
	–	f	@F	M2

Initial: M1 if (∼A & B & ∼C)

Table 1: Mode transition table for SSE.

The values of controlled variables change in response to events when the system is in particular modes. Table 2 shows an event table for a controlled variable D. This variable starts with value False and becomes True when the system is in mode M1 and an event @T(C) occurs.

Mode	Triggering Event	
M1	@T(C)	–
M3	–	@F(C)
D =	True	False

Initial: False

Table 2: Event table for a controlled variable D.

Our analysis tool processes a simplified version of SCR specifications in ASCII format, using boolean variables to represent predicates on monitored and controlled variables. In addition to mode transition and event tables, our requirements format also includes declarative information about system variables which often appears in other sections of an SCR requirements document. This information records relationships between monitored or controlled variables. Relationships help requirements designers eliminate redundant information and increase the clarity of specifications[3]. They also improve readability and reduce the effort involved in annotating an implementation. The following is a list of the relationships which we implemented:

- An *equivalence* (B = C) is a relationship which indicates that B and C change their values at the same time.
- An *implication* (B -> C) is a relationship which indicates that when B is True, C should be True; and when C is False, B should be False. In specifying SSE, we use a relationship E -> ∼D to indicate that whenever E is True, D is False.
- A *strict implication* relationship (B ->> C) is similar to implication except that when B becomes True, C should already True, and when C becomes False, B should already be False.
- A *timeline* relationship (T1 < T2) exists between variables which represent different lengths of time during which a particular condition holds. For example, T1 indicates that a condition is True for a shorter length of time than T2. T1 must be True when T2 becomes True, and must become False when T2 becomes False.
- An *enumeration* relationship (A | B | C) indicates that exactly one of the conditions is True at all times.

- A *range enumeration* relationship (A — B — C) is an enumeration relationship with an extra restriction that changes can affect only adjacent conditions, e.g., when A becomes True, B becomes False and C remains False.

The requirements designer may also specify system safety properties. Such properties duplicate information stored in the transition tables, but capture the desired information more compactly. We allow the following properties to be specified (P, P_1 and P_2 represent propositional logic formulae; M is a system mode):

- *smi(M, P)* (strong mode invariant) is satisfied when P is an invariant of a mode M. For SSE, we specify smi(M=M1, A) to indicate that A is True when the system is in mode M1 of mode class M.
- *wmi(M, P)* (weak mode invariant) is similar to *smi*. Conditions which change values as the result of triggering events may still be part of a weak mode invariant. For example, if conditions A and B are False when mode M1 is entered, and the event @T(A) WHEN ∼B causes a transition from M1, then the strong and the weak mode invariants are smi(M1, ∼B) and wmi(M1, ∼A & ∼B).
- *cause(P_1, P_2)* is satisfied if whenever P_1 is True, either P_2 is True or the next transition of the system will establish a state in which P_2 holds. For example, a property cause(M=M3 & ∼C, D) in SSE specifications indicates that the system will reach a state where D is True on all paths where the system is in mode M3 and C is False.
- *strcause(P_1, P_2)* is satisfied if whenever P_1 is True, the next transition of the system will establish a state in which P_2 holds.
- *reach(P)* is satisfied if the system can reach a state in which Property holds. reach properties are usually used to ensure that other types of properties are not satisfied vacuously. For example, specifying reach(M=M3) for SSE indicates that the system should eventually transit to mode M3.

A full specification of SSE can be found in Appendix A.

3 Implementation Properties

An implementation is consistent with its requirements if it implements all state transitions specified in the requirements, does not implement any state transitions which are not specified in the requirements, and satisfies all user-defined safety properties. A requirements specification can be translated into a list of properties which capture this notion

of consistency. To prove that an implementation is consistent with its requirements, we demonstrate that the implementation is a model of *all* of all these properties. We express system properties as Computation Tree Logic (CTL)[8] formulae. CTL is a propositional branching time logic, whose temporal operators permit explicit quantification over all possible futures. Table 3 summarizes some CTL operators.

CTL Operator	Description
$\mathcal{A}\bigcirc$(f)	f holds in every immediate successor
$\mathcal{E}\bigcirc$(f)	f holds in some immediate successors
$\mathcal{A}\Diamond$(f)	f eventually holds on all paths
$\mathcal{E}\Diamond$(f)	f eventually holds on some paths
$\mathcal{A}\Box$(f)	f holds on all paths

Table 3: CTL Operators.

3.1 Automatically-Generated Safety Properties

The properties that an implementation of SSE would have to satisfy appear in Figure 3. Properties P_1 - P_3 ensure that the only transitions to a mode are those specified in

$$
\begin{aligned}
P_1 &= \mathcal{A}\Box(M1 \vee (\mathcal{E} \bigcirc M_1 \rightarrow (M1 \vee (M_2 \,\&\, @T(A) \,\&\, @T(B)) \vee (M_3 \,\&\, @T(A)))))\\
P_2 &= \mathcal{A}\Box(\mathcal{E} \bigcirc M_2 \rightarrow (M_2 \vee (M_1 \,\&\, B \,\&\, @T(C)) \vee (M_3 \,\&\, \sim B \,\&\, @F(C))))\\
P_3 &= \mathcal{A}\Box(\mathcal{E} \bigcirc M_3 \rightarrow (M_3 \vee (M_1 \,\&\, @F(B)) \vee (M_2 \,\&\, A \,\&\, @F(B))))\\
P_4 &= \mathcal{E}\Diamond(M_1 \,\&\, B \,\&\, @T(C) \,\&\, \mathcal{E} \bigcirc M_2)\\
P_5 &= \mathcal{E}\Diamond(M_1 \,\&\, @F(B) \,\&\, \mathcal{E} \bigcirc M_3)\\
P_6 &= \mathcal{E}\Diamond(M_2 \,\&\, @T(A) \,\&\, @T(B) \,\&\, \mathcal{E} \bigcirc M_1)\\
P_7 &= \mathcal{E}\Diamond(M_2 \,\&\, A \,\&\, @F(B) \,\&\, \mathcal{E} \bigcirc M_3)\\
P_8 &= \mathcal{E}\Diamond(M_3 \,\&\, @T(A) \,\&\, \mathcal{E} \bigcirc M_1)\\
P_9 &= \mathcal{E}\Diamond(M_3 \,\&\, \sim B \,\&\, @F(C) \,\&\, \mathcal{E} \bigcirc M_2)
\end{aligned}
$$

Figure 3: Properties to ensure consistency with SCR mode tables.

the requirements. For example, P_2 asserts that if the system will be in mode M_2 in its next state, then either it must already be in that mode or one of the requirements' transitions is occurring: the system is in mode M_1 with an event $@T(C)$ `WHEN`$[B]$, or in M_3 with an event $@F(C)$ `WHEN`$[\sim B]$. P_2 was obtained from composing the rows in the SCR tables which have M_2 in their right columns. P_1 slightly differs from P_2 since it captures the fact that the system starts in mode M_1. Such properties are called "only

legal transitions" (OLT). There is one such property generated for every requirements mode (of every modeclass). Properties P_4 - P_9 ensure that all transitions specified in the requirements appear in implementations and are called "all legal transitions" (ALT) properties. P_4 and P_5 correspond to the two transitions leaving mode M_1 and can be obtained by composing the rows of the SCR tables which have M_1 in their left columns. There is one ALT property for every row of every mode transition table specified in the requirements. Properties generated for the controlled variable D are shown in Figure 4. P_{10} and P_{11} are OLT properties which ensure that D changes value only as a result of specified events, and P_{12} - P_{13} are ALT properties which ensure that all changes to D specified in the requirements appear in the implementation. There are two OLT properties generated for each controlled variable, reflecting changes of value to True and False, respectively, and one ALT property generated for each row of every event table.

$$
\begin{aligned}
P_{10} &= \mathcal{A}\Box(\sim D \ \vee \ (\mathcal{E} \bigcirc \sim D \rightarrow (\sim D \ \vee \ (@F(C) \ \& \ M3)))) \\
P_{11} &= \mathcal{A}\Box(\mathcal{E} \bigcirc D \rightarrow (D \ \vee \ (@T(C) \ \& \ M1))) \\
P_{12} &= \mathcal{E}\Diamond(D \ \& \ @F(C) \ \& \ M3 \ \& \ \mathcal{E} \bigcirc \sim D) \\
P_{13} &= \mathcal{E}\Diamond(\sim D \ \& \ T(C) \ \& \ M1 \ \& \ \mathcal{E} \bigcirc D)
\end{aligned}
$$

Figure 4: Properties for the controlled variable D.

3.2 User-Specified Safety Properties

The five types of properties that the user can specify in our requirements format can be easily translated into CTL formulae, as shown in Figure 5. This Figure clearly identifies the difference between *smi* and *wmi* properties. Property $smi(M, P)$ means that for every state of the system P holds whenever the system is in mode M. In contrast, property $wmi(M, P)$ means that P holds only in those states where the system is in mode M and remains in M in at least one of its next states.

$$
\begin{aligned}
U_1 = smi(M, P) &= \mathcal{A}\Box(\sim M \vee P) \\
U_2 = wmi(M, P) &= \mathcal{A}\Box((\sim M \vee P) \vee \mathcal{A} \bigcirc \sim M) \\
U_3 = cause(P_1, P_2) &= \mathcal{A}\Box(\sim P_1 \vee (P_2 \vee \mathcal{A} \bigcirc P_2)) \\
U_4 = strcause(P_1, P_2) &= \mathcal{A}\Box(\sim P_1 \vee \mathcal{A} \bigcirc P_2) \\
U_5 = reach(P_1) &= \mathcal{E}\Diamond(P_1)
\end{aligned}
$$

Figure 5: Formulae for user-specified properties.

These types of user-specified properties frequently appear in requirements documents, and during our case studies we did not need to verify any other types of properties. Re-

quirements specifications of realistic size and complexity (like [2]), however, contain global properties which cannot be expressed using only these assertions. Allowing the user to specify a richer set of properties is relatively easy, although not every CTL-expressible formula can be verified during our analysis. We defer a further discussion of this issue until Section 8.

4 Creating a Program Abstraction

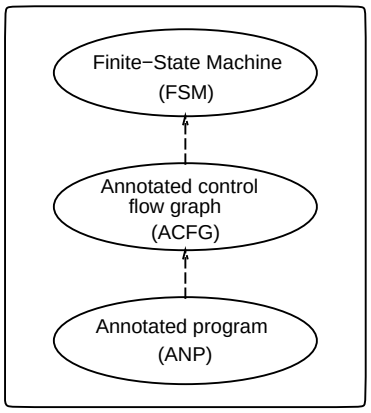

Figure 6: Layers of abstraction.

In this section we present a way to create an abstraction of a program which we use for our analysis. We need a way to establish a correspondence between the implementation and information specified in the requirements. Our solution involves annotating the program and, assuming that the annotations correctly reflect what the code does, using these annotations together with control-flow information of the program to create the necessary abstraction (see Figure 6). Afterwards, we use this abstraction to check system properties.

4.1 Annotating an Implementation

Annotations are user-specified comments which record mode transitions and changes to the values of controlled and monitored variables. Annotations are interleaved with the program statements, but start with @@ to distinguish them from the program text. Annotations capture local rather than invariant program properties, and therefore are relatively easy to specify.

We distinguish between controlled, monitored, and mode class variables. Monitored and controlled variables have boolean values; mode class variables have values which form

enumerated types whose constant values are the modes of the mode class. Thus our system states consist of a finite number of variables whose types contain only a finite number of values.

An *Initial annotation* indicates the starting state of each mode class. It unconditionally assigns values to variables. This annotation corresponds to initialization information specified in the requirements. An *Update annotation* assigns values to variables, identifying points at which the program changes its state. Analyzer uses Update annotations to compute sets of values that requirements variables may have at these points (i.e., possible system states). An *Assert annotation* asserts that variables must have particular values in the current system state. Static analysis usually gives imprecise results because system states are aggregated. Assert annotations reduce the amount of information in the system state to what the programmer *knows* to be true.

The usual objections to inserting annotations in the code are that this process greatly increases the amount of work during the implementation step and that the properties described by annotations may be complex. Annotations generally correspond to assignments of constants to variables or predicates on a variable's value. Thus, blocks of code which compute new values of variables often do not need to be annotated. Our experience indicates that the usual code/annotation ratio is about 10/1. The properties described by annotations are simple since they describe local state changes rather than invariant properties.

Figure 7a contains a fragment of an implementation of the Water-Level Monitoring system which is the subject of our case study. We presented the code with some typical user comments in order to compare the comments and our annotations (Figure 7b). Although the first portion of the code manipulates variables to compute a new system state, it is not annotated. The remaining part, consisting of predicates and assignments of constant values to variables, is heavily annotated.

Although most of our experience came from annotating existing code, we envision a code development process in which annotations are inserted while the program is being written.

4.2 Creating an Annotated Flow Graph

We construct a CFG of the program from the source text and annotations, and then propagate the state information about the requirements variables throughout the CFG. Figure 8a shows a small fragment of the implementation of the SSE. We have omitted all program variables and kept only the program control flow and annotations. This is the

```
while(TRUE) {
  ...
  GET_MODE(OperatingClass, OPCM);
  WATER_LEVEL(&LVLCM);
  INLIM=((LVLCM<(float)DPAR_HWL) &&
         (LVLCM>(float)DPAR_LWL));
  GET_TIME(SHUTER, &SHUTTM);
  if (ERR_VAR_F(CLK_ERR)==CLK_OVERFLOW) {
    RESET_TIMER(SHUTER);
    SHUTTM=0;
  }
  if (MODE_EQ(OPCM, Shutdown) && INLIM &&
     (SHUTTM<DPAR_SLT) && !PRESSED(SlfTst)){
    /* we are in mode Shutdown and water */
    /* is within limits */

     SET_MODE (OperatingClass, Operating);

  }
  ...
  GET_MODE(OperatingClass, OPCM);
  if (!MODE_EQ(OPCM, Test))) &&
      (PRESSED(SlfTst))){
    /* we are not in Test and self-test */
    /* button is pressed */
    SET_MODE(OperatingClass, Test);

    INIT_DRVS();
  }
  ...
}
```

a)

```
while(TRUE) {
  ...
  GET_MODE(OperatingClass, OPCM);
  WATER_LEVEL(&LVLCM);
  INLIM=((LVLCM<(float)DPAR_HWL) &&
         (LVLCM>(float)DPAR_LWL));
  GET_TIME(SHUTER, &SHUTTM);
  if (ERR_VAR_F(CLK_ERR)==CLK_OVERFLOW) {
    RESET_TIMER(SHUTER);
    SHUTTM=0;
  }
  if (MODE_EQ(OPCM, Shutdown) && INLIM &&
     (SHUTTM<DPAR_SLT) && !PRESSED(SlfTst)){
    @@ Assert Normal(Shutdown);
    @@ Update WithinLimits & ~SlfTstPressed
       & ~ShutdownLockTime200;
    SET_MODE (OperatingClass, Operating);
    @@ Update Normal(Operating);
  }
  ...
  GET_MODE(OperatingClass, OPCM);
  if (!MODE_EQ(OPCM, Test))) &&
      (PRESSED(SlfTst))){
    @@ Assert ~Normal(Test);
    @@ Update SlfTstPressed;
    SET_MODE(OperatingClass, Test);
    @@ Update Normal(Test);
    INIT_DRVS();
  }
  ...
}
```

b)

Figure 7: Annotations in WLMS code.

starting point for the analysis of the SSE.

Definition 4.1 *A CFG of an annotated program P (ACFG) is a directed graph $G = \langle V, E, V_0 \rangle$, where*

V is a finite set of nodes corresponding to decisions, joins and annotations of P.
$E \subseteq V \times V$ 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.

We interpret annotations in an ANP to create a *set-based approximation* of attainable values for each requirements' variable[10, 11]. We compute two sets of information for our analysis: *reaching values* (RVs) and *conditions* (Conds). The RV of a variable at a node is a set of values that a variable may attain if the control reaches the node. RVs are computed by interpreting Update and Initial annotations. The Cond of a variable at a node is a set of values the variable must attain if control reaches the node. Conds are computed by interpreting all annotations.

We use a simple abstraction function α to map the finite sets of values that variables may and must attain to abstract values representing these respective sets:

$$\alpha : 2^{V_c} \rightarrow V_a$$

where V_c is a set of concrete values that a variable may (or must) attain if control reaches some program node and V_a is a set of abstract values used to represent RVs and Conds for each variable. Values of V_a are partially ordered via set inclusion.

A concretization function $\gamma : V_a \rightarrow 2^{V_c}$:

$$\gamma(a) = \bigvee c,\ \forall (c \in V_c) \wedge (\alpha(c) = a)$$

is an inverse of α, and gives the concrete form of an abstract value, treating variables with abstract values corresponding to multiple concrete values as having each of these values. For example, if the abstract value of A is {True,False}, then A can have concrete values True or False.

Definition 4.2 *A system state SS at a node n is a set of triples (i, v, c) such that*
$i \in R$, where R is the set of all controlled, monitored and mode class variables;
$v \in V_a$ is an abstract value representing the RV *of i at the node n; and*
$c \in V_a$ is an abstract value representing the Cond *of i at the node n.*

For a system state SS, let

$$\begin{aligned}
\texttt{RV}(SS) &= \{(i,v) \mid (i \in R) \wedge (v \in V_a) \wedge (\exists c \in V_a \ s.t.\ (i,v,c) \in SS)\} \\
\texttt{Cond}(SS) &= \{(i,c) \mid (i \in R) \wedge (v \in V_a) \wedge (\exists v \in V_a \ s.t.\ (i,v,c) \in SS)\} \\
\texttt{RV}(SS,i) &= \{v \mid (i,v) \in \texttt{RV}(SS)\} \\
\texttt{Cond}(SS,i) &= \{c \mid (i,c) \in \texttt{Cond}(SS)\}
\end{aligned}$$

To evaluate programs abstractly, we define the meaning of *union* and *widening* operations ([10]) for system states. These operations are used to combine state information at

join nodes of condition and loop statements, respectively. The *union* operation combines `RV` and `Cond` values of variables using the usual set union operation. If SS_1 and SS_2 are two system states, then their union SS_r is defined as follows:

$$\begin{aligned} SS_r = SS_1 \bigcup SS_2 \quad = \quad & \{(i, v, c) \mid (i \in R) \ \wedge \ (v \in V_a) \ \wedge \ (c \in V_a) \ \wedge \\ & (v = RV(SS_1, i) \cup RV(SS_2, i)) \ \wedge \\ & (c = Cond(SS_1, i) \cup Cond(SS_2, i))\} \end{aligned}$$

Thus, our system states form a complete $\bigcup$-lattice under the partial ordering of set inclusion. Given that all variables in R have a finite number of abstract values, our system states do not have an infinite increasing chain of values[10], and thus we can define our *widening* operation to be the same as *union*. Thus, we no longer need to distinguish between the two types of join nodes, which simplifies our analysis significantly.

Our computation of system states at each node of the $ACFG$ is similar to that of reaching definitions via dataflow techniques ([1, 7]). `gen` sets capture new values generated at each node in an $ACFG$. The sets are empty except for nodes corresponding to annotations, which contain variable-value pairs for each variable. These pairs are produced from values in annotations and from information about the relationships between variables. Variables which are unrelated to those appearing in annotations, have empty-set values in the `gen` set. Two different `gen` sets are calculated, $\texttt{gen}_u$ and $\texttt{gen}_a$, to compute `RV`s and `Cond`s. These sets are identical except at nodes corresponding to Assert annotations where $\texttt{gen}_u$ is an empty set and $\texttt{gen}_a$ contains variable-value pairs derived from the annotation.

`kill` sets contain values that variables no longer have after each node. Two different `kill`s, $\texttt{kill}_u$ and $\texttt{kill}_a$, are computed from the following template by replacing `gen` with $\texttt{gen}_u$ and $\texttt{gen}_a$ respectively.

$$\texttt{kill}(n) = \{(i, v_1) \mid ((i, v_2) \in \texttt{gen}(n)) \wedge ((v_2 = \bot \rightarrow v_1 = \bot) \vee (v_2 \neq \bot \rightarrow v_1 = \top - v_1))\},$$

where "–" is set difference and $\top$ indicates the set of all possible values for i.

Figure 8b shows the $ACFG$ for the code fragment in Figure 8a. `gen` and `kill` sets computed at each node are shown in bold face on this figure. We did not include variables whose values are empty sets. The first Assert generates `Cond` information that mode class M has to be either in mode M1 or M3, and thus the value M2 is killed. The second Assert (on the left branch) generates the value True for B (killing False), and the Update generates the value True for C, affecting both $\texttt{gen}_a$ and $\texttt{gen}_u$ sets. The right branch is similar. The Update on the join generates the value True for D and M2 for M. Since D is related to E via implication, this Update also generates the value False for E.

We compute `in`(n) and `out`(n) attributes whose values are system states before and after a node n, respectively (SS(n) = `out`(n)), via a least fixed point algorithm using the

following system of equations:

$$\begin{aligned} \texttt{in}(n) &= \bigcup_{\forall\ k,\ s.t.\ (k,n)\ \in\ E} \texttt{out}(k) \\ \texttt{RV}(\texttt{out}(n)) &= \texttt{gen}_\texttt{u}(n) \cup (\texttt{RV}(\texttt{in}(n)) - \texttt{kill}_\texttt{u}(n)) \\ \texttt{Cond}(\texttt{out}(n)) &= \texttt{gen}_\texttt{a}(n) \cup (\texttt{Cond}(\texttt{in}(n)) - \texttt{kill}_\texttt{a}(n)) \end{aligned}$$

where union and difference are the usual set operations. Figure 8b shows `Cond`s and `RV`s in italic and regular fonts, respectively. Nodes of the graph which affect the computation of reaching values are shaded. Assume that processing starts with `RV` and `Cond` sets being {(A, {}), (B, {True, False}), (C, {}), (D, {True, False}), (E, {False}), (M, {M1,M2,M3})}. The first Assert changes the value of M in `Cond` to {M1,M3}. The Assert on the left branch restricts the value of B to {True} in the `Cond` set, whereas the Update changes the value of C to {True} in both the subsequent `Cond` and `RV` sets. The right branch is similar. At the join, we union the possible values for variables in `RV`s and `Cond`s. C and B are {True} on the left branch and {False} on the right, and so their resulting values are {True, False} in both `RV` and `Cond` sets.

4.3 Constructing a Finite-State Machine

We create a Finite-State Machine FSM from an $ACFG$ (see Figure 6) and use it for verification of the system properties. In an $ACFG$ ($G = \langle V, E, V_0 \rangle$), let $U \subseteq V$ and $I \subseteq V$ be sets of nodes containing Update and Initial annotations, respectively. (U and I are disjoint.)

Definition 4.3 *An FSM over a program P is a Kripke structure $M = \langle A,\ S,\ L,\ N,\ s_0 \rangle$, where*

A is a set of system states;
$S = U \cup I$ is a finite set of nodes;
L: $S \rightarrow A$ is a function labeling each node with a system state which holds in that node;
$N \subseteq S \times S$ is a total transition relation[1], i.e. $\forall x \in S\ \exists y \in S$ s.t. $(x,\ y) \in N$. N is obtained by connecting nodes of S s.t. there is an Update-clear path between them in $ACFG$; and
$s_0 \in S$ is an entry node.

Definition 4.4 *A path is an infinite sequence of nodes $s_0,\ s_1,\ s_2,\ \ldots$, s.t. $N(s_i,\ s_{i+1})$ is true for every i. A path is feasible if it can be taken during some execution of the program.*

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

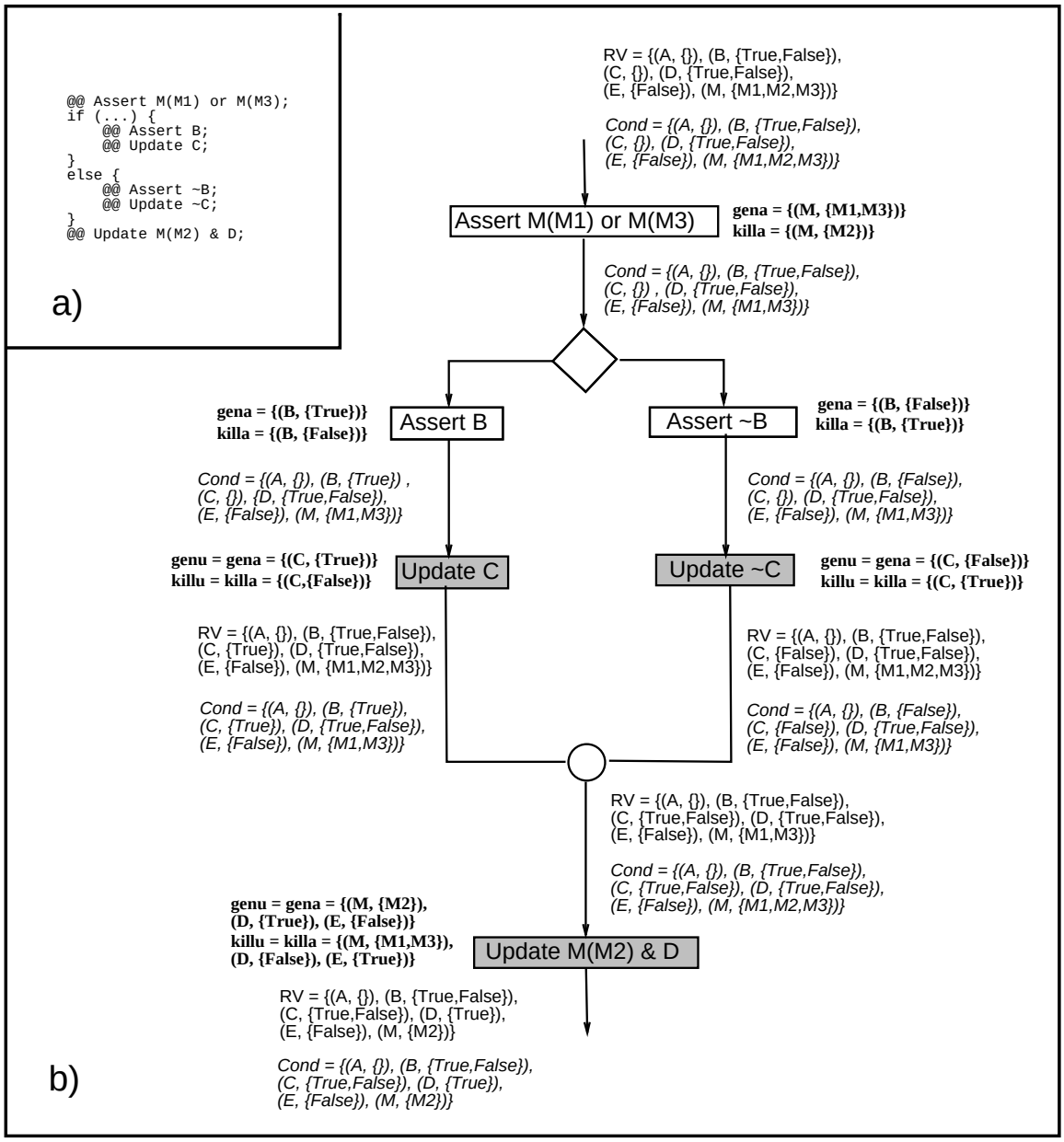

Figure 8: Computation of `Cond`s and `RV`s. a) A code fragment. b) A corresponding $ACFG$.

We combine `RV` and `Cond` information for each node of the FSM, resulting in `Info` sets.

$$\texttt{Info}(n) = \{(i, r) \mid (i \in R) \wedge (r \in V_a) \wedge ((i, v, c) \in SS(n)) \wedge (r = v \cap c)\},$$

where $\cap$ is the usual set intersection. This definition can be viewed as computing $\texttt{Info}(n) = \texttt{Cond}(n) \sqcap \texttt{RV}(n)$. We verify an implicit property P_{14}, indicating that each variable *may*

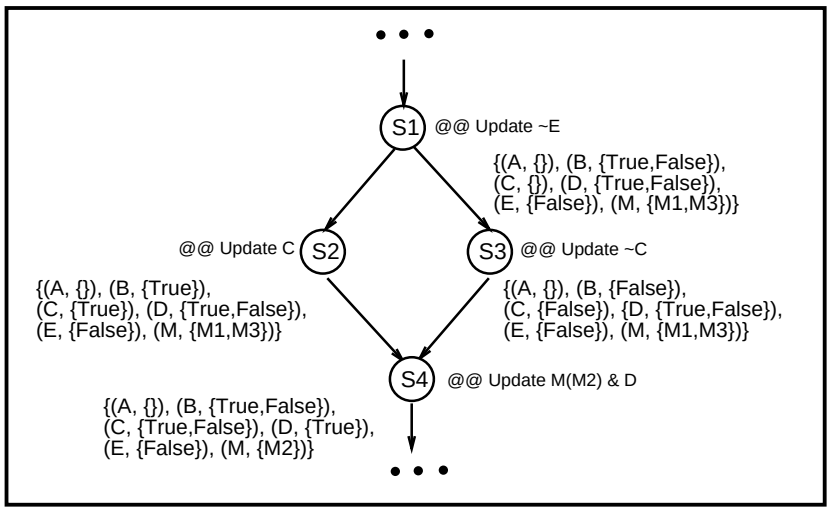

Figure 9: Finite-state abstraction.

attain at least one of the values it *must* attain:

$$P_{14} = \mathcal{A}\Box(\forall i \in R\ \exists(i,v,c) \in SS(n)\ \exists(i,r) \in \texttt{Info}(n)\ s.t.\ c \neq \{\} \rightarrow r \neq \{\})$$

This is the "assertion satisfied in current state" (ASCS) property, saying that if a `Cond` value for some variable is different from {}, then there is a value other then {} resulting from computation of `Info` for this variable.

Figure 9 shows a fragment of an FSM which abstracts the code in Figure 8, with `Info` sets shown at every state. In this figure, all states correspond to nodes containing Update annotations in the $ACFG$ in Figure 8b.

5 Verifying Properties

In this section we describe how automatically-generated and user-specified properties are verified.

5.1 Formal Model

We use the final abstraction, FSM, to verify properties about the system. To understand which of these properties are preserved in the correctly annotated program, we need to formalize our discussion, using notation similar to that in [33][2]:

[2]Unfortunately, it is impossible to analyze the actual program behavior when annotations are missing or wrong. We are currently exploring ways to explicitly connect annotations and code.

Let $\mathcal{M}$ be a class of models. In our case, $\mathcal{M}$ contains *ANP*, *ACFG*, and *FSM*.

Let $\mathcal{P}$ be a class of specification formulas, i.e., all automatically-generated and user-specified properties. These have a well-defined interpretation, i.e.

$$\forall M \in \mathcal{M}, \forall p \in \mathcal{P}, (M \models p) \vee (M \not\models p)$$

Let ϕ: $\mathcal{M} \rightarrow \mathcal{M}$ be a mapping from one model to another.

We note that we utilize an abstraction function α to go from the concrete domain V_c to the abstract domain V_a in *ANP*. Thus, we define a separate model, ANP_a, representing *ANP* on abstract values. The following is the process we undertook to create *FSM* from *ANP*:

$$ANP \xrightarrow{\alpha} ANP_a \xrightarrow{\phi_1} ACFG \xrightarrow{\phi_2} FSM,$$

where ϕ_1 and ϕ_2 are transformation functions defined in addition to the abstraction function α.

We verify all properties in *FSM* and are interested in their interpretation in *ANP*. In our case, it is hard to determine the correct interpretation of results of verifying properties since *FSM* was an abstraction designed specifically to bridge the gap in event granularity between requirements and an implementation. We further note that some of the paths in *ANP* are infeasible, and thus not all nodes in the resulting *FSM* can occur during the actual runs of the program. Let FSM_r and FSM_{rc} represent *FSM* produced by the actual runs of the program on abstract and concrete values of variables, respectively, and we will interpret the results with respect to FSM_{rc}. The interpretation process goes as follows:

$$FSM \xrightarrow{\phi_3} FSM_r \xrightarrow{\gamma} FSM_{rc},$$

where ϕ_3 is a transformation function defined in addition to the concretization function γ. We further define a combined transformation function $\phi = \phi_3 \circ \gamma$, and thus *FSM* $= \phi(FSM_{rc})$. Once we establish that a property holds (does not hold) on FSM_{rc}, we conclude that it holds (does not hold) on *ANP*. However, results of property verification on *FSM* are interpreted differently for FSM_{rc}, depending on the property being verified, due to the presence of infeasible paths and the fact that γ is not one-to-one.

Definition 5.1 *A map ϕ is falseness-preserving with respect to specification formula p, written FP(ϕ,p), iff $\forall M \in \mathcal{M}$, if $M \not\models p$, then $\phi(M) \not\models p$.*

Definition 5.2 *A map ϕ is truth-preserving with respect to specification formula p, written TP(ϕ,p), iff $\forall M \in \mathcal{M}$, if $M \models p$, then $\phi(M) \models p$.*

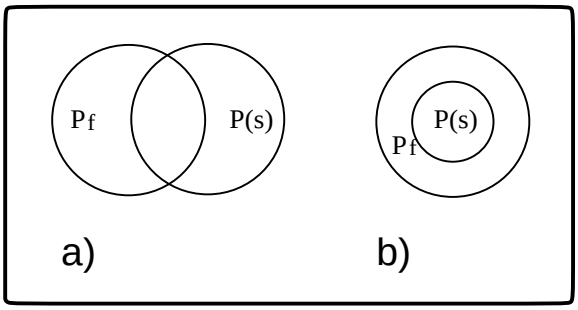

Figure 10: a) Optimistic and b) Pessimistic interpretations of a formula holding in a state.

Definition 5.3 *A map ϕ is not consistent with respect to specification formula p iff*

$$(\sim TP(\phi, p)) \wedge (\sim FP(\phi, p)).$$

We say that a property p is pessimistic for a map ϕ if TP(ϕ, p) and optimistic if FP(ϕ, p).

In our case, the map ϕ: $FSM \rightarrow FSM_{rc}$ is truth-preserving with respect to some properties and falseness-preserving with respect to some others. OLT properties involve quantification over every possible state. Thus, if one of these properties is preserved in the model, it is preserved in the annotated program. The map ϕ should then be falseness-preserving with respect to these properties. The ASCS property is also universally quantified and thus we need to define ϕ so that it is falseness-preserving with respect to it. However, ALT properties are existentially quantified, i.e., if there a state satisfying these properties in the model, there might not be one in the actual program because the paths on which these properties hold in the model might be infeasible. Thus, ϕ should be truth-preserving with respect to these properties.

When we determine if a state is a model of a formula, we should make sure that ϕ remains consistent for every formula, i.e., ϕ is truth-preserving or falseness-preserving for every formula. We define two interpretations for a formula holding in a state, one for pessimistic and another for optimistic properties. Variables in states have values represented by sets, so we treat atomic (or negated atomic) propositions in formulas as assertions about sets of values, e.g., P is treated as $P = \{True\}$ and $\sim M(M1)$ is treated as $M = \{M2, M3\}$. To check if a state s is a model of an atomic proposition P, we compare P's asserted value (P_f) with its value in state s ($P(s)$). For optimistic analysis, if there are common values between P_f and $P(s)$, i.e., $P_f \cap P(s) \neq \emptyset$ (see Figure 10a), we assume that P_f holds in s. In pessimistic analysis we want to find the maximum number of errors, and thus we consider P_f to hold in a state s only if $(P(s) \neq \{\}) \wedge (P(s) \subseteq P_f)$, i.e., all values of $P(s)$

are present in P_f (see Figure 10b). This way, during the interpretation step,

$$\forall p \in \mathcal{P}, ((FP(\phi, p) \wedge FSM \models p) \rightarrow (FSM_{rc} \models p)) \wedge$$
$$((TP(\phi, p) \wedge FSM \not\models p) \rightarrow (FSM_{rc} \not\models p)),$$

i.e., pessimistic properties which hold in FSM, also hold in FSM_{rc}, and optimistic properties violated in FSM, are also violated in FSM_{rc}.

5.2 Checking Automatically-Generated Properties

To verify the automatically-generated properties, we take advantage of their rather restricted forms and verify all of them in a single traversal of the FSM. These properties involve state transitions which occur only in response to events. To calculate a node's event, we traverse the FSM backwards until a node containing an Update annotation is found on each branch. We form conjunctions of the Triggering and When conditions in these Updates. For example, consider calculating the event which results in the transition to M(M2) at node S_4 in Figure 9. Predecessors on this node are S_2 and S_3. The Triggering conditions for transitions from S_2 to S_4 and from S_3 to S_4 consist of the information specified in the corresponding $\texttt{gen}_u$ sets. The When conditions are the `Info` sets at S_2 and S_3. The set I of transitions and events discovered by Analyzer is shown in Figure 11.

Transitions from $M(M1)$ to $M(M2)$ (or from $M(M3)$ to $M(M2)$):

$$I_1(I_2) : @T(C) \quad \texttt{WHEN} \quad [B = \{True\} \,\&\, D = \{True, False\} \,\&\, E = \{False\}]$$
$$I_3(I_4) : @F(C) \quad \texttt{WHEN} \quad [B = \{False\} \,\&\, D = \{True, False\} \,\&\, E = \{False\}]$$

Transitions from D to D (or from $\sim D$ to D):

$$I_5(I_6) : @T(C) \quad \texttt{WHEN} \quad [B = \{True\} \,\&\, E = \{False\} \,\&\, M = \{M1, M3\}]$$
$$I_7(I_8) : @F(C) \quad \texttt{WHEN} \quad [B = \{False\} \,\&\, E = \{False\} \,\&\, M = \{M1, M3\}]$$

Figure 11: Events and transitions discovered for node S_4.

Our event calculations change the semantics of SCR events in two ways. Although an SCR event occurs only when a value of a condition on a monitored variable changes, we assume that every Update annotation marks an event. This analysis exaggerates the number of events we calculate since an Update may not change a value of its variable, but this treatment simplifies searches for events in FSM. Secondly, SCR state transitions occur instantaneously with the events that trigger them, while our state transitions may be triggered by events associated with Update annotations which are textually remote

from the annotation marking the state transition. This difference is the result of analyzing implementations in which variables triggering events and those recording state transitions change values in different statements, each of which is separately annotated.

Verification of ASCS Property

The ASCS property is checked for each state during the FSM traversal which computes `Info` sets. A programmer might intentionally annotate his program with an assertion which "contradicts" `RVs` at some node, thus violating the ASCS property. This is done to explicitly mark a node in the FSM as unreachable to ensure that it is not considered in the verification of other properties. Thus, when Analyzer finds a violation of the ASCS property in a node, it marks the node as unreachable and skips it during the rest of the property verification step. The ASCS property is universally quantified, and thus ϕ is pessimistic with respect to it, i.e. more errors could be reported than are present in the annotated program.

Verification of OLT Properties

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

$$P_2 = \mathcal{A}\Box(\mathcal{E} \bigcirc M_2 \rightarrow (M_2 \ \vee \ (M_1 \ \& \ B \ \& \ @T(C)) \ \vee \ (M_3 \ \& \sim B \ \& \ @F(C))))$$

All such properties have the general form $P = \mathcal{A}\Box(\mathcal{E} \bigcirc F \rightarrow (F \vee G))$. Since $\mathcal{A}\Box(\mathcal{E} \bigcirc F \rightarrow (F \vee G)) = \mathcal{A}\Box(\mathcal{A} \bigcirc (\sim F) \vee (F \vee G))$, these properties are universally quantified and thus can be verified pessimistically, thus possibly identifying illegal transitions even if they actually occur on infeasible paths. We need to examine only those states in which F is False and in whose successors F is True (nodes S_2 and S_3 in our example). Thus, we check the formula only for predecessors of states labeled with Update annotations for F. In our example, the calculated events I_1 and I_4 establish P_2 for S_2 and S_3, respectively. We check all OLT properties in one traversal of the FSM by the following algorithm:

Algorithm 1

For every node y reachable from s_0 in the depth-first order of the FSM
  If y contains an Update annotation establishing F then
    For every x s.t. $(x, y) \in N$, where N is the successor relation
      If $x \models \sim F$ then
        If $x \models \sim G$ then

Report error

Continue

THEOREM 1. *The successful execution of Algorithm 1 indicates that a property* $P = \mathcal{A}\Box(\mathcal{E} \bigcirc F \rightarrow (F \vee G))$ *holds in our model, i.e., if the algorithm reports no error, then* $FSM \models P$.

PROOF. We will prove a contrapositive of the above statement, i.e., if $FSM \not\models P$, then the algorithm reports an error.

Assume that $FSM \not\models P$. Then $\exists z \in S$ s.t. z is reachable from s_0 and $z \models \sim P$, i.e. $z \models \mathcal{E} \bigcirc F$ but $z \not\models F$ and $z \not\models G$. Then $\exists w, (z, w) \in N \wedge w \models F$ which means that w contains an annotation establishing F. The algorithm starts with s_0, comes to w during its depth-first traversal, finds z (since $(z, w) \in N$), checks that $z \models \sim F$ and $z \models \sim G$, and reports an error.

So, if $FSM \not\models P$ then the algorithm reports an error. □

Verification of ALT Properties

The following are some of the ALT properties for SSE:

$$\begin{aligned} P_4 &= \mathcal{E}\Diamond(M_1 \;\&\; B \;\&\; @T(C) \;\&\; \mathcal{E} \bigcirc M_2) \\ P_9 &= \mathcal{E}\Diamond(M_3 \;\&\; \sim B \;\&\; @F(C) \;\&\; \mathcal{E} \bigcirc M_2) \\ P_{13} &= \mathcal{E}\Diamond(\sim D \;\&\; T(C) \;\&\; M1 \;\&\; \mathcal{E} \bigcirc D) \end{aligned}$$

Properties P_4 and P_9 hold in the model because the node S_4 satisfies them via transitions I_1 and I_4, respectively (see Table 11). Analyzer marks ALT properties indicating that they have been satisfied if the implementation's calculated event contains values that would cause a requirement's state transition. Any properties remaining unmarked at the end of analysis are reported as errors. This analysis is optimistic since it considers state transitions to satisfy a property even if they occur on infeasible paths. Thus, we might not report all unimplemented transitions. We also use an optimistic interpretation for a formula holding in a state. For example, while verifying property P_{13}, we find that the node S_4 satisfies it by our treatment of transition I_6. This node is reachable from the starting node, and thus the property is considered to hold.

5.3 Verifying User-Specified Properties

Recall that a user can specify five types of properties - reach, smi, wmi, strcause and cause. Each of these properties is verified in a separate traversal of the FSM.

- reach(P) properties are existentially quantified and thus are verified optimistically, i.e., Analyzer searches the state space for a state where P holds, and if one is found, the entire property is considered to hold.
- During the verification of smi(M, P) = $\mathcal{A}\Box(\sim M \vee P)$ properties, Analyzer takes all states where $\sim M$ does not hold, and determines if P holds in each of these states. As soon as an error is discovered, the process terminates. These properties are verified pessimistically, i.e., if there is one state which violates the invariant (although it can be on an unreachable path), the entire property is considered violated. All remaining types of properties are also universally quantified, and so their verification is pessimistic.
- During the verification of wmi(M, P) = $\mathcal{A}\Box((\sim M \vee P) \vee \mathcal{A} \bigcirc \sim M)$ properties, Analyzer looks for the nodes s.t. M holds in the node and in at least one of its successors, and checks that P holds in this node. The verification stops as soon as an error is discovered.
- During the verification of cause(P_1, P_2) properties, Analyzer starts with all states where $\sim P_1$ fails and checks if P_2 holds there; if not, it looks at all successor states and reports an error if P_2 does not hold in either of these.
- Verification of strcause(P_1, P_2) properties is done similarly to cause(P_1, P_2) except that only successor states are examined.

Figure 12 summarizes the basic steps of our analysis.

6 Processing Functions

In Section 4, we presented our algorithm for intraprocedural analysis of annotated programs, which creates an $ACFG$. However, since programs usually consist of several procedures, we needed to extend our analysis technique to process these programs. We perform interprocedural analysis using an adaptation of a technique called *cloning* [9]. A similar algorithm was described in [32]. This technique enables Analyzer to process programs with cycles in their call graphs (recursion), to analyze each called function only a constant

1. Start with an annotated program. Build its CFG.
2. Abstract it to $ACFG$:
   Compute `gen` and `kill` sets for each node containing an annotation.
   Propagate `RV`s and `Cond`s throughout the graph.
3. Abstract $ACFG$ to FSM:
   Remove all nodes except those containing Update or Initial annotations
   For each node, compute `Info` = `Cond` $\sqcap$ `RV` and check the ASCS property.
      If violated, report violation and mark the node as unreachable.
4. Generate $\mathcal{P}$ - a set of properties to ensure that the implementation is consistent with the requirements.
5. Check that $FSM \models \mathcal{P}$:
   For each reachable event changing a mode class or controlled variable:
      Compute Triggering and When conditions from all predecessors.
      Verify that each of the transitions is specified in requirements.
      Mark these transitions as used.
   Report unused mode transitions, if any.
7. Verify user-specified properties of the system.

Figure 12: Summary of the algorithm.

number of times, and to achieve reasonable precision in the analysis. Our algorithm clones a CFG of a function each time it is called in a new calling context (i.e., new `RV`s or `Cond`s). We verify each copy of a function once with its own set of reaching values and conditions. Since we ignore functions without annotations and the number of possible calling contexts is finite (and hopefully small), we avoid many potential problems with the combinatorial growth of the CFG of the entire program.

Definition 6.1 *An $ACFG$ of an annotated program P in the presence of function calls is a directed graph $G = \langle V, E, V_0 \rangle$, where*

V is a finite set of nodes corresponding to decisions, joins, function calls and annotations of P.

$E \subseteq V \times V$ is a set of directed edges, s.t. $(v_1, v_2) \in E \leftrightarrow v_2$ can immediately follow v_1 in some execution sequence; and

V_0 is an entry node.

We use an example in Figure 13b to illustrate how the computation of `RV`s is performed in the presence of function calls. This program uses recursion to decrement a counter until

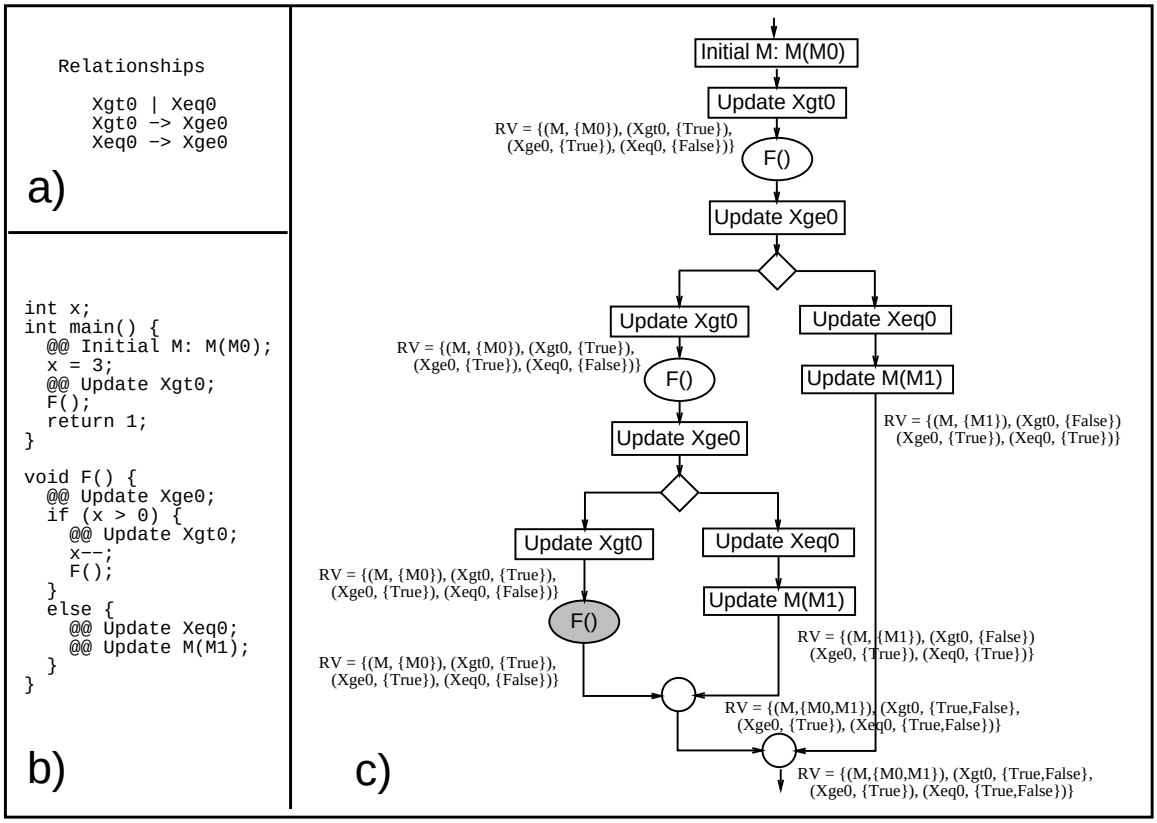

Figure 13: Computing RVs in the presence of function calls. a) Some relationships between the monitored variables. b) An implementation. c) An $ACFG$ generated for this example.

it reaches zero. The three monitored variables, Xgt0, Xeq0, Xge0, stand for "X greater than 0", "X equal to 0" and "X greater than or equal to 0", respectively. These are connected via relationships shown in Figure 13a. Figure 13c presents the $ACFG$ produced for this example. Ellipses indicate function calls. While computing RVs and encountering a call to F from the main routine, Analyzer creates a copy of the F's $ACFG$ in the environment where the initial information, `in(RV)`, is {(M, {M0}), (Xgt0, {True}), (Xge0, {True}), (Xeq0, {False})}, which is used as the `RV` set at the start of the function, and the computation is continued through the $ACFG$ of F. When the function calls itself (the shaded node in Figure 13b), the `RV` of the call site is identical to that of the original call, so Analyzer skips the recursive call and sets the output information, `out(RV)`, of the call site to its `in(RV)`. Processing of the non-recursive branch of F yields the context {(M, {M1}), (Xgt0, {False}), (Xeq0, {True}), (Xge0, {True})}. When these two sets are merged at join node at the end of F, F's `out(RV)` set is {(M, {M0, M1}), (Xgt0, {True,False}), (Xge0, {True}), (Xeq0, {True,False})}. This also becomes the `out(RV)` set of main's call to F.

A similar computation occurs for condition propagation except that both `RV` and `Cond` information should match in order for the recursive call to be skipped. The algorithm for processing function calls appears in Appendix B. After `RV` and `Cond` information is calculated, Analyzer abstracts $ACFG$ to FSM and uses it to verify the system properties, as described in Sections 4 and 5.

7 Case Study

A Water-Level Monitoring System (WLMS) monitors and displays the water level in a container. It also raises visual and audio alarms and shuts off its pump when the level is out of range or when the monitoring system fails. Two push buttons, SelfTest and Reset, permit the operator to test the system and return it to normal operation. A complete description of this system can be found in [31]. 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 modeclass Normal is shown in Table 5.

The requirements included four user-defined safety properties, identical to those used in [4]. They are shown in Table 6. If the SelfTest button has been pressed for 500ms or more, the system is either in mode Test or will be in mode Test after its next transition. When the system is in mode Standby, the SelfTest button has not been pressed for 500ms. If the system is in mode Operating, then either the water level is WithinLimits or the SelfTest button has been pressed long enough to cause a transition to mode Test. If the system is in mode Shutdown, then either the water level is outside the hysteresis water-level range and the system has been in mode Shutdown for less than 200 ms, or the SelfTest button has been pressed but not long enough to cause a transition to mode Test.

The WLMS was originally implemented by roughly 1300 lines of FORTRAN and Assembler code. To analyze the program, we translated it into C and replaced its PC interface routines with an Xlib interface. We annotated the program with 32 Update annotations corresponding to monitored variables, 30 Update annotations corresponding to mode class and controlled variables, and 33 Assert annotations. Out of 54 functions in the implementation, only eight had annotations.

After we eliminated annotation errors in the implementation, Analyzer reported a num-

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

Table 4: WLMS Modes.

Current Mode	Inside HysR	Within Limits	SlfTst Pressed	SlfTst Pressed 500	In Test 1400	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)

Table 5: Mode transition table for mode class Normal.

ber of inconsistencies between the requirements and the implementation (see Table 7). These numbers overestimate the actual errors in the implementation. First of all, some mode transitions and controlled variable value changes resulted in a number of OLT properties violations. For example, five illegal mode transitions generated 34 violation messages because several illegal transitions were detected at each location. Also, all the mode transition problems can be attributed to three principal causes: the wrong monitored variable was checked to enable mode transitions (WithinLimits rather than InsideHysR), the times

Properties
cause(SlfTstPressed500, Normal(Test))
wmi(Normal=Standby, ∼SlfTstPressed500)
wmi(Normal=Operating, (∼SlfTstPressed500 & (WithinLimits OR SlfTstPressed)))
wmi(Normal=Shutdown, (∼SlfTstPressed500 & ((∼InsideHysR & ∼ShutdownLockTime200) OR ∼SlfTstPressed)))

Table 6: Properties of the WLMS.

Property Type	Violations	Locations
OLT properties for modeclasses	34	5
OLT properties for controlled variables	56	5
ALT properties for modeclasses	10	
ALT properties for controlled variables	26	

Table 7: Results of analyzing the WLMS.

that the operator pressed the SelfTest and Reset buttons were not calculated or checked, and no transitions to a mode corresponding to complete system failure were implemented. Most of the illegal assignments to the controlled variables occurred because the order of triggering events in the implementation differed from that in the requirements.

Each of the four safety properties was verified by Analyzer. These results are not particularly interesting because all the safety properties included negated monitored variables that were never assigned True values in the implementation (i.e., InsideHysR and SlfTstPressed500). Thus most formulas hold trivially. The third formula:

```
wmi(Normal=Operating, ~SlfTstPressed500 & (WithinLimits OR SlfTstPressed)))
```

was verified since WithinLimits was set to True before each transition to Operating, and whenever WithinLimits and SlfTstPressed were set to False, the system immediately transitioned to mode Shutdown.

8 Discussion and Conclusion

This section compares our approach with related work and discusses potential improvements in our analysis method.

8.1 Related Work

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

In Inscape[28, 29, 30], complex logical formulas are abstracted to simple predicates which may be primitive or defined in terms of other predicates (like our relationships). 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 its calculated and the specified conditions.

Quick Defect Analysis (QDA)[18, 20] also uses a simplified specification language. Partial specifications 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[19] enriches QDA's specification language so assertions also describe event occurrences and hypotheses assert that the implementation's events occur in certain sequences.

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)[25, 26, 27]. 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[21, 22, 23] 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. [7] 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*[12], a subset of CTL* in which only a 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.

8.2 Conclusion and Future Work

We have defined a notion of consistency between SCR-style requirements and an annotated program. We have also presented a technique to ensure this notion of consistency, implemented in a tool called Analyzer. Analyzer creates a finite-state abstraction of an annotated program and checks it against a set of properties automatically generated from the requirements or specified by the user. This approach can be applied to small but realistic systems, as indicated by the case study that was presented here.

The algorithm which creates a finite-state model of a program uses only annotations and the control-flow of the program. Thus, it might happen that the annotations are correct, but the corresponding constructs of the program are incorrect or missing. Also, a program has to be fully (and correctly) annotated to obtain maximum benefits from the analysis. We are currently looking at means of connecting requirements and program variables. The user will provide declarative information describing the dependencies between implementation and requirements variables. We hope that this approach will enable us to find missing or incorrect annotations and, in many cases, insert necessary annotations automatically.

The language for user-specified safety properties is very restricted. Let ϕ be a mapping from an annotated program to FSM. ϕ is truth-preserving with respect to CTL formulas quantified over all possible paths and falseness-preserving with respect to formulas quantified over some paths. Therefore, ϕ is not consistent with respect to an arbitrary CTL formula p, i.e., verifying p on FSM gives no information about its validity on the annotated program. So, assuming that a formula does not include negated quantifiers (i.e. $\sim \mathcal{A}\Box(P) = \mathcal{E}\Diamond(\sim P)$, we can verify it only if it does not contain mixed quantifiers. Young and

Taylor[34] reach the same conclusion. All properties discussed in earlier sections contained just one quantifier, and thus could be processed. We plan to extend Analyzer to be able to verify arbitrary consistent CTL formulae.

Acknowledgements

We would like to thank Rich Gerber and Bill Pugh for many valuable technical contributions to this work.

References

[1] A. Aho, R. Sethi, and J. Ulman. *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] M. Chechik and J. Gannon. "Automatic Verification of Requirements Implementations". In *Proceedings of the 1994 ISSTA*, pages 1–14, Seattle, Washington, August 1994.

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

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

[8] 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.

[9] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. "Procedure Cloning". In *Proceedings of IEEE International Conference on Computer Languages*, pages 96–105, April 1992.

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

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

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

[13] 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.

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

[15] 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.

[16] K. Heninger. "Software Requirements for the A-7E Aircraft". Technical Report NRL Report 3876, Naval Research Laboratory, Washington, DC, 1978.

[17] 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.

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

[19] W.E. Howden and G.M. Shi. "Linear and Structural Event Sequence Analysis". Submitted to ISSTA'96, June 1995.

[20] 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.

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

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

[23] Daniel Jackson. "Aspect: Detecting Bugs with Abstract Dependences". (submitted to Transactions on Software Engineering and Methodology), November 1993.

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

[25] 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.

[26] 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.

[27] 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.

[28] Dewayne E. Perry. "Software Interconnection Models". In *Proceedings of the 9th International Conference on Software Engineering*, pages 61–69. IEEE Computer Society Press, 1987.

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

[30] 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.

[31] 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.

[32] Ben Wegbreit. "Property Extraction in Well-Founded Property Sets". *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.

[33] Michal Young. "How to Leave Out Details: Error-Preserving Abstractions of State-Space Models". In *Proceedings of the Workshop on Software Testing*, pages 63–70, 1988.

[34] Michal Young and Richard N. Taylor. "Rethinking the Taxonomy of Fault Detection Techniques". In *Proceedings of the 11th International Conference on Software Engineering*, pages 53–62, May 1989.

A SSE Requirements specification

```
MONITORED
  A, B, C

CONTROLLED
  D, E

MODECLASS M
  INITIAL M1 (~A & B & ~C & D)
  MODE M1
    M2 @T(C) WHEN [B]
    M3 @F(B)
  MODE M2
    M1 @T(A) & @T(B)
    M3 @F(B) WHEN [A]
  MODE M3
    M1 @T(A)
    M2 @F(C) WHEN [~B]
```

```
CONTROLLED VARIABLES
  VARIABLE D
    INITIAL False
    True @T(C) WHEN [M=M1]
    False @F(C) WHEN [M=M3]

  VARIABLE E
    INITIAL True
    False @T(C) WHEN [M=M1]
    True @F(C)

RELATIONSHIPS
  E -> ~D

PROPERTIES
  smi (M=M1, A)
  reach (M=M3)
  cause (M=M3 & ~C, D)
```

All such specifications adhere to the same format: monitored and controlled variables are listed separately, followed by a mode transition table for each mode class. Event tables for controlled variables, a list of relationships and a list of user-specified properties complete the specification.

B Algorithm to Process Function Calls

We store separate copies of CFGs of each function for different calling contexts. Every call cite contains a pointer to a CFG of a copy of a function it calls, or a null pointer if it calls a function without annotations. To distinguish between copies of the functions, we associate a list of contexts with each of them. Each context contains the following information: `in` and `out` which contain system states at the start and the end of the function; `graph` which is a pointer to the CFG of the function with this calling environment; and `color` which

indicates the processing status: either WHITE (unprocessed), GRAY (partially processed), or BLACK (processed). Initially, `ins` and `outs` are empty, and each context is marked WHITE. In the presentation of the algorithm, when these fields have a two argument (e.g., `in(RV, cl)`), they represent reaching value or condition information about the node corresponding to the function call, and when they have three arguments (e.g., `in(RV, ct, fn)`), they represent information about the body of the function in a particular context. Let $\mathcal{PR}$ be a set of functions which needs to be processed, i.e. those which contain annotations or call other functions which need to be processed. Figure 14 contains the algorithm.

1. Compute RVs. When a node calls a function in $\mathcal{PR}$, check its list of contexts.
If there exists a context ct s.t. in(RV,ct,fn) is in(RV,cl) then
    If color(ct,fn) is BLACK (already processed) then
    out(RV,cl) ← out(RV,ct,fn)
    Elseif color(ct,fn) is GRAY (recursion) then
    out(RV,cl) ← in(RV,ct,fn)
    /* skip the call */
Else
    New context nct ← $CopyCFG$ (graph(ct,fn))
    graph(cl) ← graph(nct,fn)
    color(nct,fn) ← GRAY
    in(RV,nct,fn) ← in(RV,first block(fn)) ← in(RV,cl)
    Recursively propagate RVs through the function
    color(nct,fn) ← BLACK;
    out(RV,last block(fn)) ← out(RV,cl) ← out(RV,nct,fn)
2. Mark all copies of all functions WHITE.
3. Compute Conds. If a node calls a function in $\mathcal{PR}$, check its list of contexts.
If there is a context ct s.t. in(RV,ct,fn) is in(RV,cl) and
in(Cond,ct,fn) is in(Cond,cl) then
    If color(ct,fn) is BLACK (already processed) then
        graph(cl) ← graph(ct,fn)
        out(Cond,cl) ← out(Cond,ct,fn)
    Elseif color(ct, fn) is GRAY (recursion) then
    out(Cond,cl) ← in(Cond,ct,fn)
Else
    New context nct ← $CopyCFG$ (graph (ct,fn))
    /* copy graph together with RVs */
    graph(cl) ← (graph(nct,fn)
    color(nct,fn) ← GRAY
    in(Cond,nct,fn) ← in(Cond,fist block(fn)) ← in(Cond,cl)
    Recursively compute conditions in the function
    color(nct,fn) ← BLACK
    out(Cond,last block(fn)) ← out(Cond,cl) ← out(Cond,nct,fn)

Figure 14: Algorithm to Process Function Calls.