Automatic Verification of Requirements Implementation*

Marsha Chechik
John Gannon

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

Abstract

Requirements of event-based systems can be automatically analyzed to determine if certain safety properties hold. However, we lack comparable methods to verify that implementations maintain the properties guaranteed by the requirements. We have built a tool that compares implementations written in C with their requirements. Requirements describe events which cause state transitions. Implementations are annotated to describe changes in the values of their requirement's variables, and dataflow analysis techniques are used to determine the set of events which cause particular state changes. To show that an implementation is consistent with its requirements, we show that each event causing a change of state in the implementation appears in the requirements, and that all the events specified to cause state changes in the requirements appear in the implementation. The annotation language encourages programmers to describe local program behaviors. These behaviors are collected into system-level behaviors, which are compared to those in the requirements. Since our analysis is not based on program code, annotations can describe behaviors at any level of granularity. We illustrate the use of our tool with several different annotations of a temperature-control system.

1 Introduction

The keys to winning acceptance for employing formal methods during the 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. Generally, when two descriptions of a system are written, designers attempt to convince themselves that concepts are captured in the higher-level description and preserved in the lower-level description [9].

Formal methods can be used to demonstrate that safety properties of event-based systems are enforced by the system's requirements[3]. After building a finite-state structure representing the system's reachability graph, a model checker can automatically determine if the structure is a model of temporal logic formulas representing the system's safety properties. Unfortunately, proving these properties provides no guarantee that they will be preserved in an implementation of the system.

In Software Cost Reduction (SCR) requirements [2, 5, 6], systems are modelled as concurrently executing state machines. Transition tables describe changes in the states of the system and in the values of the system's variables. To show that an implementation is consistent with its requirements, we show that each event causing a change of state in the implementation appears in the requirements, and that all the events specified to cause state changes in the requirements appear in the implementation.

We have built a prototype tool, called Analyzer, which reads a requirements specification and an anno-

*This work was supported by the Air Force Office of Scientific Research under contract F49620-93-1-0034.

tated C source program implementing the requirements. Using dataflow analysis techniques, the tool computes the events which cause state transitions in the implementation, determines if the transitions are permitted by the requirements, and keeps track of which of the requirements transitions have appeared in the implementation.

Our tool uses concepts from comments analysis [7], Cesar/Cecil[10, 11], and Aspect[8]. In comments analysis, implementations are annotated with two types of comments: assertions that an object has one of a finite number of values, and assumptions about the object's values at particular control points. An interpreter traverses a graph representation of the program to build states representing the possible information flows in the program. These states are tested by a verifier to determine if the assumptions are satisfied by the assertions on all paths which reach the assumptions. The Cecil 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. After a user specifies a mapping from programming language constructs to Cecil events, the Cesar analyzer determines if the implementation meets the Cecil constraints. In Aspect, procedures are annotated with assertions that their results depend on certain inputs. Dataflow analysis is used to compute an upper bound on the dependencies of the implementation. If an asserted dependency is missing, an error is reported. In a manner similar to comments analysis, we analyze user-specified annotations and the control flow of the implementation to determine if assertions hold. Like Cesar/Cecil and Aspect, we use a formal description of our intentions (i.e., requirements) to determine if our solution meets our expectations.

The remainder of the paper is organized as follows: Section 2 presents SCR requirement specifications and uses them to describe a small system for temperature control. In Section 3, we describe our annotation language and demonstrate its use on an implementation of the sample system from the previous section. Section 4 describes how Analyzer works and shows examples of the types of errors it detects. In Section 5, we conclude with some remarks about the strengths and weaknesses of our approach.

2 SCR Requirements

SCR requirements model a system as a state machine which interacts with its environment's state variables. Changes to its *monitored* state variables may cause the system to change state and to alter the values of its *controlled* state variables.

A *modeclass* is a state machine whose states are *modes* and whose transitions occur in response to *events.* An event occurs when there is a change in the value of a condition, which is a predicate on state variables. An event @T(A) WHEN [B] occurs if a condition A changes from False to True while a condition B is True. Similarly, an event @F(A) WHEN [B] occurs if a condition A changes from True to False while a condition B is True. We refer to A and B as triggering and When conditions, respectively. Mode transitions are instantaneous and occur at the same time as their respective transition events. Complex systems are described by a finite set of concurrent state machines.

SCR requirements use tables to define the values of controlled variables and mode transitions. Condition tables define the values of state variables as a function of a system's modes and conditions. Each entry in a mode transition table maps a mode and an event to another mode in the same modeclass.

As an example, consider a temperature-control system that runs a heater and an air conditioner in order to keep the current temperature within a desired range. Such a system could be represented by a single mode class Operating with four modes: Off, Inactive, Heat, and AC, as described in Table 1. The monitored variables, described in Table 2, indicate the status of the system's on/off switch and the difference between the desired and the actual temperatures. The controlled variables (see Table 3) are switches which turn the air conditioner and the heater on and off. In order to make our analysis task easier, all of our monitored and controlled variables are boolean. We replace predicates on variables with enumerated types or discrete ranges of values by a finite number of boolean conditions. In our example, *AboveTemp, TempOK* and *BelowTemp* represent the three outcomes of comparing the desired and the actual temperatures. The values of controlled variables are specified by condition tables. Each row in the condition tables specifies an event which changes the value of the controlled variable. A table entry containing an upper-case letter ('@T' or '@F') indicates that the condition changes value (to True or False, respectively). An entry containing a lower-case letter ('t' and 'f') signifies that the condition must have a particular value (True or False, respectively) immediately before and at the time of the event occurrence. "Don't care" entries are marked with hyphens ('–'). Table 4 defines the values of the controlled variable *HeatOn*, which activates the heater. Initially, *HeatOn* is False. If the system is in INACTIVE and *BelowTemp* becomes True while *Running* is True, *HeatOn* is assigned the value True. Other events cause *HeatOn* to be assigned the value False. A similar condition table defines the values of *ACOn.*

Each row in the mode transition table (Table 5) specifies the event causing the transition from the mode on the left to the mode on the right. Initially, the sys-

Mode	Meaning
OFF	The system is off.
INACTIVE	The system is on, but neither the heater nor the air conditioner is on.
AC	The air conditioner is on.
HEAT	The heater is on.

Table 1: Modeclass Operating.

Variable	Meaning
Running	True if the system's on/off switch is on and false otherwise.
AboveTemp	ActualTemp $\geq$ (DesiredTemp+3°F).
TempOK	(DesiredTemp-3°F) $<$ ActualTemp $<$ (DesiredTemp+3°F).
BelowTemp	ActualTemp $\leq$ (DesiredTemp-3°F).

Table 2: Monitored variables.

tem is in OFF and *Running* is False. The system is undefined if the switch is initially on. When the system is in OFF and the switch is turned on, *Running* becomes True, and the system transitions from OFF to INACTIVE. AC is entered when the system is INACTIVE, and the temperature rises above the desired. When the temperature falls to within $3°F$ of the desired, the system exits AC and enters INACTIVE. Transitions in and out of HEAT are similar to those of AC.

In addition to these tables, we declare relationships between the controlled and monitored variables of the system to help requirements designers eliminate redundancy and increase the clarity of specifications [3]. These relationships improve readability and reduce the effort involved in writing the requirements as well as annotating the implementation. We implement three of the most frequently used relationships: implication, equivalence, and enumeration.

Implication ($\mathbf{X \rightarrow Y}$). Whenever X is asserted to be True, Y must also be True. Whenever Y is asserted to be False, X must also be False.

Equivalence ($\mathbf{X = Y}$). This defines a relationship between variables X and Y which is equivalent to X $\rightarrow$ Y AND Y $\rightarrow$ X. Whenever one variable is asserted to be True (False), the other is also True (False).

Enumeration ($\mathbf{X_0 \mid X_1 \mid \cdots \mid X_n}$). This defines a relationship between variables X_0, X_1, $\cdots X_n$ such that only one of the X_i's can be True at any time. Whenever one of X_i's is asserted to be True, all the others are False. When one of the variables is asserted to be False, no other variables change values.

3 Annotations

To use Analyzer, a programmer annotates his code to assign and test values of monitored, controlled, and modeclass variables. All annotations start with @@ and a command (one of COND, UPDATE, SET, INITIAL, or PERFORM INITIAL), and are followed by a list of expressions.

Monitored and controlled variables have boolean values; modeclass variables have values which form enumerated types whose constant values are the modes of the modeclass. Analyzer computes the set of possible values that each variable could have at different points of the computation.

3.1 SET annotations

SET annotations unconditionally change the values of monitored variables. For example,

```
@@ SET TempOK;
```

assigns the value True to TempOK. Each of AboveTemp and BelowTemp is assigned the value False if the variables have been declared to be related by enumeration.

3.2 UPDATE annotations

Controlled and modeclass variables cannot appear in SET annotations because their values change only in response to event occurrences. UPDATE annotations change values of these variables if Analyzer determines that the current system state contains the appropriate modes and events specified in the requirements. For example,

```
@@ UPDATE Operating(Heat) AND HeatOn;
```

specifies that the modeclass variable Operating and the controlled variable HeatOn should be assigned the values Heat and True, respectively. The current system state and its predecessors are evaluated to determine if

Variable	Meaning
ACOn	Set to True to activate the air conditioner and set to False to deactivate it.
HeatOn	Set to True to activate the heater and set to False to deactivate it.

Table 3: Controlled variables.

Mode	*Running*	*AboveTemp*	*TempOK*	*BelowTemp*	*HeatOn*
INACTIVE	t	–	–	@T	t
HEAT	t	–	–	@F	f
HEAT	@F	–	–	–	f

Initial value: f

Table 4: HeatOn Condition Table.

these assignments are legal. For each mode which is a possible value of the modeclass variable in the current system state, the state must contain an event which the requirements specify as causing a transition to the new mode. In our example, the mode transition table for Operating permits a transition to mode Heat only when Operating={Inactive}, Running=True, and BelowTemp becomes True. Thus, the current system state must contain these variable-value bindings. In addition, BelowTemp must have had the value False[1] in all of the predecessors of the current system state. Similar checks are performed for the assignment to HeatOn. If the current system state does not contain events enabling the mode transition or the change of value of the controlled variable, an error is reported.

Compound UPDATE annotations like the one above specify that Operating and HeatOn change values at the same time. If separate annotations had been written, e.g.,

```
@@ UPDATE Operating(Heat);
@@ UPDATE HeatOn;
```

then the variables change values at different times. The changes described by the compound annotation are triggered by the same event, while those described by the separate annotation are triggered by different events. In the latter case, HeatOn must only change value in response to an event triggered by the state change caused by the UPDATE annotation for Operating.

3.3 INITIAL annotations

Special initialization annotations make it possible to assign values to variables regardless of their previous values. For each modeclass, one INITIAL annotation is placed at the start of the module implementing the modeclass. In our example,

```
@@ INITIAL Operating: Operating(Off) AND
   NOT Running AND NOT ACOn AND NOT HeatOn;
```

assigns the value Off to the modeclass variable Operating and the value False to Running as specified in the mode transition table of the requirements. In addition, each of ACOn and HeatOn is assigned the value False in accordance with the condition tables of the requirements. The PERFORM INITIAL annotation is used to reset the system to its original state. We found this construct necessary in applications with more than one modeclass. The annotation

```
@@ PERFORM INITIAL Operating;
```

unconditionally resets the values of the variables in current state to those specified in the corresponding INITIAL annotation for the modeclass.

3.4 COND annotations

COND annotations assert that variables have particular values in the current system state. If the values specified in the COND annotation appear in the current system state, the test succeeds, and the analysis continues with the variable having only the values specified in the COND command. Otherwise, an error is reported, and the rest of the path is skipped. In the following example, the COND command is successful if the possible values of Operating in the current state include AC, Heat, and Inactive.

```
if (e == OFF) {
  @@ COND Operating(AC) AND Operating(Heat)
     AND Operating(Inactive);
  ...
}
```

[1]We also permit a special TrueOrFalse value which is explained in the section on Analysis.

Current Mode	*Running*	*AboveTemp*	*TempOK*	*BelowTemp*	New Mode
Off	@T	–	–	–	Inactive
Inactive	@F	–	–	–	Off
	t	@T	–	–	AC
	t	–	–	@T	Heat
AC	@F	–	–	–	Off
	t	–	@T	–	Inactive
Heat	@F	–	–	–	Off
	t	–	@T	–	Inactive

Initial mode: Off if ~*Running*

Table 5: Operating condition table.

If this is the case, the analysis continues with Operating ={AC, Heat, Inactive}.

4 Analysis

Analyzer reads two input files, an ASCII version of the requirements (Figure 1) and an annotated C program (Figure 2), and produces messages identifying illegal or unused mode transitions and changes of values of controlled variables. Invariants for each mode are also calculated.

We obtain control flow information from the gcc compiler in RTL format which we use to create a control flow graph (CFG) for the program.[2] Dataflow analysis techniques are used to convert the CFG to a basic block graph (BBG) in which each block contains a copy of the system state representing the effects of all possible executions of the program. We traverse the BBG to determine if its states are consistent with the requirements.

4.1 Building a basic block graph

System states contain one entry for each modeclass, controlled variable, and monitored variable. Modeclass values are sets of modes which the system can attain at a particular point in the computation. Controlled and monitored variables have one of the values Undefined, True, False, and TrueOrFalse (indicating that the value of the variable may be either True or False).

GEN and KILL sets[1] are computed for each SET, UPDATE, INITIAL, and PERFORM INITIAL command. For example, consider the annotation

```
@@ UPDATE Operating(Heat) and HeatOn;
```

The GEN set for this annotation contains HeatOn= True, Operating={Heat}, and undefined values for all other variables. The KILL set contains HeatOn= False, Operating={AC, Inactive, Off}, and undefined values for all other variables.

Each node of the CFG containing a SET, UPDATE, INITIAL, or PERFORM INITIAL annotation is marked as the leader of a basic block[1], and the basic blocks of the CFG are identified and stored in the BBG. An iterative algorithm is used to compute reaching definitions[1] for every basic block. The IN and OUT sets represent system states before and after the execution of the blocks. The algorithm uses set operations to manipulate GEN, KILL, IN, and OUT sets. Implementing set operations on modeclass variables is straightforward since their values are sets of modes. We define the effects of set operations on boolean values by allowing monitored and controlled variables to have values like TrueOrFalse (e.g., to represent the value of a variable in the union of two system states, one of which contained the variable with the value False and the other contained the variable with the value True). Tables 6 and 7 define the union and the intersection operations for monitored and controlled variables. The set difference is defined similarly.

4.2 Traversing a basic block graph

Each annotation is verified at most once during a depth-first traversal of a BBG. For each path, Analyzer builds a path condition and a current system state which are used during the analysis. Both the path condition and the current system state are empty at the start. Encountering an INITIAL annotation replaces the empty current system state with a copy of the state associated with the INITIAL block. When a COND annotation is processed, the values of the annotation's variables must appear in the current system state. Otherwise, an error is reported, and the traversal of this path is abandoned. If the test succeeds, the values of COND's variables replace the path condition, and the analysis continues. Processing a SET annotation alters values of

[2] Currently we create a CFG for each function defined in the program and replace each call to such a function by its entire CFG.

```
MONITORED
        Running, AboveTemp, TempOK, BelowTemp

CONTROLLED
        ACOn, HeatOn

MODECLASS Operating
        Initial Off (~Running)
        MODE Off
                Inactive @T(Running)

        MODE Inactive
                Off @F(Running)
                AC @T(AboveTemp) WHEN [Running]
                Heat @T(BelowTemp) WHEN [Running]
        MODE AC
                Off @F(Running)
                Inactive @T(TempOK) WHEN [Running]
        MODE Heat
                Off @F(Running)
                Inactive @T(TempOK) WHEN [Running]

CONTROLLED_VARIABLES
        VARIABLE ACOn
        Initial False
                True @T(AboveTemp) WHEN [Operating=Inactive & Running]
                False @F(AboveTemp) WHEN [Operating=AC & Running]
                False @F(Running) WHEN [Operating=AC]
        VARIABLE HeatOn
        Initial False
                True @T(BelowTemp) WHEN [Operating=Inactive & Running]
                False @F(BelowTemp) WHEN [Operating=Heat & Running]
                False @F(Running) WHEN [Operating=Heat]

RELATIONSHIPS
                AboveTemp | TempOK | BelowTemp
```

Figure 1: Sample mode transition table.

A \ B	Undefined	True	False	TrueOrFalse
Undefined	Undefined	True	False	TrueOrFalse
True	True	True	TrueOrFalse	TrueOrFalse
False	False	TrueOrFalse	False	TrueOrFalse
TrueOrFalse	TrueOrFalse	TrueOrFalse	TrueOrFalse	TrueOrFalse

Table 6: Computing A $\cup$ B.

the monitored variables in the current system state, but no verification is performed, and no errors are reported. When a block with an UPDATE annotation is reached, we verify that events changing values of modeclass or controlled variables may occur in the current system state and are specified in the requirements. If appropriate events exist, the UPDATE annotation changes the values of the variables in the current system state. Otherwise an error is reported, and the path is abandoned.

In describing how UPDATE annotations are verified, we only discuss the case of mode transitions because controlled variables are processed similarly. Assume that in the current state the value of the modeclass variable MClass is {M1, M2, M3}, and Analyzer is attempting to verify the annotation

```
@@ UPDATE MClass(M4);
```

Analyzer calculates the set of events which caused the last change in the system state. For each of M1, M2, and M3, Analyzer checks that the requirements specify M4 as a destination state for some member of the set of events.

The When conditions for the set of events can be determined by computing the intersection of the current path condition and the current system state. However, the triggering conditions for the set of events must be constructed by determining which variables changed values immediately prior to the UPDATE block. Starting from the UPDATE block, a breadth-first search of incoming arcs is conducted until another SET or UPDATE block is encountered on each arc. The intersection of the system states associated with these blocks (labeled P1 in Figure 6) yields a state containing variables which have common values (X=False). Another breadth-first search starting from the blocks identified in the previous step (P1) locates their immediate pre-

A \ B	Undefined	True	False	TrueOrFalse
Undefined	Undefined	Undefined	Undefined	Undefined
True	Undefined	True	Undefined	True
False	Undefined	Undefined	False	False
TrueOrFalse	Undefined	True	False	TrueOrFalse

Table 7: Computing A ∩ B.

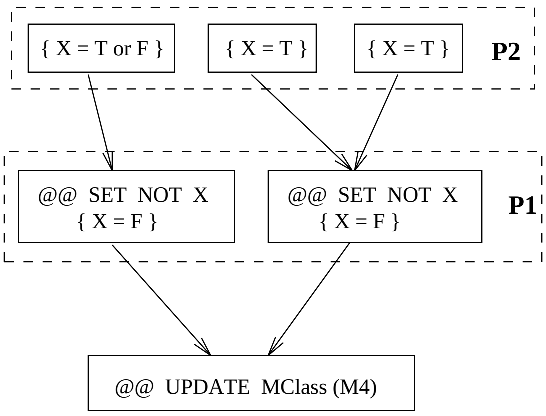

Figure 6: Analyzing an UPDATE Annotation

decessors (labeled P2). The value of X in each of the states in P2 is compared to that in its successors to determine if X's value changed.[3] If value changes are observed in all P1-P2 pairs of states examined, then a triggering event is observed, e.g., @F(X).

MClass's mode transition table is examined to ascertain that each of M1, M2, and M3 has a transition to M4 which is triggered by @F(X) with the computed When conditions. Mode transition table entries with such events are marked to indicate they have been exercised. An error is reported if any of M1, M2, or M3 cannot transition to M4 given the set of computed events and conditions.

When the traversal is complete, messages are printed identifying any unmarked mode transitions or changes of the values for the controlled variables. Invariants for each mode are computed by intersecting the values of all monitored and controlled variables from each system state in which the mode appears as a value for its modeclass variable.

4.3 Output of analysis

Analyzer reports four types of information: errors, unused mode transitions, unreached (or unreachable) modes, and mode invariants. For instance, errors are reported when a set of events constructed for an UPDATE annotation is empty. Assume that we omit the SET annotation (formerly on line 19) from the following part of the implementation:

```
16  else if (e == OFF) {
17    @@ COND Operating(Inactive) AND
      Operating(Heat) AND Operating(AC);
18    running = FALSE;
19    ac = heat = FALSE;
20    @@ UPDATE Operating(Off) AND NOT
      HeatOn AND NOT ACOn;
21  }
```

Analyzer finds no events for the UPDATE annotation. Figure 3 shows the output of Analyzer. Unused mode transitions and unchanged controlled variables are also reported because events triggered by @F(Running) never occur. The last piece of the output shows that Running is invariantly True in those states for which the requirements indicate that it should be able to become False.

Even when Analyzer computes a non-empty set of events, the requirements may not permit transitions on these events from all of the modes of the modeclass Operating in the current system state. Suppose we had mistakenly written the COND annotation on line 44 as Operating(AC) rather than Operating(Inactive).

```
43  else if (!heat && difftemp <= -3) {
44    @@ COND Operating(AC);
45    @@ SET BelowTemp;
46    heat = TRUE;
47    @@ UPDATE Operating(Heat) AND HeatOn;
48  }
```

The output of Analyzer is shown in Figure 4. At this point of the analysis, the current system state contains Operating=AC (from the COND annotation), and Analyzer computes the following event (ellipses represent possible additional When conditions):

```
@T(BelowTemp) AND @F(AboveTemp) AND
  @F(TempOK) WHEN Operating(AC) ...
```

Errors are reported because the requirements do not contain transitions to Heat from modes corresponding to the value of Operating in the current system state

[3] We consider the change from X=TrueOrFalse to X=False to be the same as from X=True to X=False.

(e.g., AC). Also, HeatOn becomes True in response to @T(BelowTemp) WHEN Inactive rather than in response to the computed event. Additional unused transitions are reported. Note that computed invariants no longer include Running=True.

Errors are also reported when the current path condition does not include the values specified in a COND annotation. Assume the COND annotation on line 32 is changed from Operating(Heat) to Operating(Heat) AND HeatOn.

```
31  else if (heat && absdifftemp < 3) {
32    @@ COND Operating(Heat) AND HeatOn;
33    @@ SET TempOK;
34    heat = FALSE;
35    @@ UPDATE Operating(Inactive) AND
36         NOT HeatOn; }
```

Analyzer then produces output shown in Figure 5. In this case, the condition on line 32 contradicted the current path condition, so lines 33-36 were skipped during the analysis. The UPDATE annotation on line 35 is reported as unreachable.

5 Discussion

Since we have only recently completed a prototype version of Analyzer, we do not have an extensive experience using it. However, it has discovered omitted cases in our implementations and sharpened our understandings of the definition of events.

In this section, we make some observations about how some of our design decisions impact Analyzer's error detection capabilities and discuss future work.

5.1 Conditions

Analyzer uses an iterative algorithm to compute the reaching definitions for each basic block. The resulting minimum fixed point computation for each block can overestimate the block's reaching definitions. Consider the following example:

```
1  x = 6;
2  while (true) {
3      if (x > 5)
4         @@ SET Y;
5      else
6          @@ SET NOT Y;
7  }
```

The set of reaching definitions computed for the basic block on line 4 is {Y = TrueOrFalse}, although only {Y = True} is actually possible. The algorithm assumes that each predicate may evaluate to either True or False and, thus, that any path in the BBG is executable.

We added COND annotations to our annotation language in order to permit programmers to indicate the conditions under which the paths are executable. COND annotations are similar to control comments in the QDA comments language[7] because they can be used to control Analyzer's execution. We also use COND annotations to reduce the number of values in the set of reaching definitions in a manner similar to abstract interpretation [4] which uses predicates to reduce the ranges of values of variables.

Analysis can be performed without any COND annotations. In this case, Analyzer reports additional errors and weaker invariants. Below is the fragment of our temperature-control system which resulted from removing the COND annotation for Operating(Off) on line 11:

```
10   if (e == ON) {
11       running = TRUE;
12       @@ SET Running;
13       @@ UPDATE Operating(Inactive);
14   }
```

This causes Analyzer to produce the following error messages:

```
Error on line 13 in modeclass Operating:
         no transition from mode(s) {Heat}
         to mode(s) {Inactive}.
Error on line 13 in modeclass Operating:
         no transition from mode(s) {AC}
         to mode(s) {Inactive}.
```

The error is reported because the system state contains values for Operating (i.e., Heat and AC) which do not have transitions to Inactive triggered by @T(Running).

In contrast, writing COND annotations which are too restrictive results in error messages which report unused rules. For example, unused rules are reported when we omit Operating(Heat) and Operating(Inactive) from line 17 of our implementation:

```
16     else if (e == OFF) {
17       @@ COND Operating(AC);
18       running = FALSE;
19       @@ SET NOT Running;
20       ac = heat = FALSE;
21       @@ UPDATE Operating(Off) AND
            NOT HeatOn AND NOT ACOn;
```

5.2 Events

Analyzer identifies *possible* rather than definite changes in the values of the variables as the triggering conditions for the events. Consider the following transition between system states:

$$\{\text{Operating} = \{\text{Heat, Inactive}\}, \\ \text{HeatOn} = \text{TrueOrFalse}\} \rightarrow \\ \{\text{Operating} = \{\text{Inactive}\}, \text{HeatOn} = \text{False}\}$$

As a result of the computation of reaching definitions, both Operating and HeatOn have two possible values in the origin state. To compute the triggering conditions, Analyzer compares each of the values in the origin state with the corresponding values in the destination state. The triggering conditions @T(Operating(Inactive)), @F(HeatOn) and @F(Operating(Heat)) are identified even though no variables change values on some execution paths. By discovering more events than may actually occur during execution, Analyzer may overlook errors resulting from missing events. A more conservative approach, considering only definite changes in values as triggering events, results in reporting false errors about illegal and unused transitions.

5.3 Future Work

We plan to perform a number of case studies to measure Analyzer's performance and to understand if our design decisions are reasonable. We are currently working on analyzing several implementations of a water-level monitoring system. While experimenting with the temperature-control system, we realized that it is possible that the annotations correspond to the requirements, but the program still does not work correctly due to an omitted statement. Analyzer never detects such problems because it only examines the annotations. Defining relationships between variables of the implementation and those of the annotations would enable us to check if an event actually occurs: we can verify that program variable(s) associated with this event also changed value(s)[10]. We are also investigating extending our analysis techniques to be able to verify mode transitions in real-time concurrent programs.

References

[1] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools.* Adisson-Welsley, 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.M. Atlee and J. Gannon. "State-based model checking of event-driven system requirements". *IEEE Transactions on Software Engineering*, pages 22–40, January 1993.

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

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

[6] 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.

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

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

[9] B.H. Liskov and S.N. Zilles. "Specification Techniques for Data Abstraction". *IEEE Transactions on Software Engineering*, SE-1(1):7–18, March 1975.

[10] 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.

[11] 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.

```
thermo(){
  boolean running = FALSE, heat = FALSE, ac = FALSE, inactive = FALSE;
  char ch; EventType e;
  int settemp = 65, currtemp, difftemp, absdifftemp;
  @@ INITIAL OPERATING: OPERATING(OFF) AND NOT RUNNING AND NOT ACON AND NOT HEATON;
  while (getevent(&e, &settemp)) {
    fscanf(stdin, "%d%ch", &currtemp, &ch);
    difftemp = currtemp - settemp;
    absdifftemp = difftemp < 0 ? -difftemp : difftemp;
    if (e == ON) {
       @@ COND OPERATING(Off);
       running = TRUE;
       @@ SET Running;
       inactive = TRUE;
       @@ UPDATE OPERATING(Inactive);
    }
    else if (e == OFF) {
      @@ COND NOT OPERATING(OFF);
      running = FALSE;
      @@ SET NOT RUNNING;
      ac = heat = inactive = FALSE;
      @@ UPDATE OPERATING(Off) AND NOT HEATON AND NOT ACON;
    }
    else if ( running ) {
      @@ COND RUNNING;
      if (ac && absdifftemp < 3) {
        @@ COND OPERATING(AC);
        @@ SET TempOK;
        ac = FALSE; inactive = TRUE;
        @@ UPDATE OPERATING(Inactive) AND NOT ACON;
      }
      else if (heat && absdifftemp < 3) {
        @@ COND OPERATING(Heat);
        @@ SET TempOK;
        heat = FALSE; inactive = TRUE;
        @@ UPDATE OPERATING(Inactive) AND NOT HEATON;
      }
      else if (inactive && difftemp > 3) {
        @@ COND OPERATING(Inactive);
        @@ SET AboveTemp;
        ac = TRUE; inactive = FALSE;
        @@ UPDATE OPERATING(AC) AND ACON;
      }
      else if (inactive && difftemp < 3) {
        @@ COND OPERATING(Inactive);
        @@ SET BelowTemp;
        heat = TRUE; inactive = FALSE;
        @@ UPDATE OPERATING(Heat) AND HEATON;
      }
    }
  }
}
```

Figure 2: Temperature-Control System Implementation

```
Error on line 20:
  no triggering events found for the following transitions:
  Modeclass Operating   : {Inactive, AC, Heat} -> {OFF}.
  Variable  ACOn        : TRUE -> FALSE.
  Variable  HeatOn      : TRUE -> FALSE.
                         .....

                **** Unused Transitions: ****
FROM                          EVENT                 TO
------------------------------------------------------
Heat            @F(Running)                         OFF
AC              @F(Running)                         OFF
Inactive        @F(Running)                         OFF

VARIABLE                      EVENT                 VALUE
----------------------------------------------------------
ACOn            @F(Running) WHEN [Operating(AC)]    FALSE
HeatOn          @F(Running) WHEN [Operating(Heat)]  FALSE

              **** Computed Invariants: ****
                   Modeclass Operating:
Mode Heat:       Mode AC:            Mode Inactive:    Mode OFF:
ACOn   = EITHER  ACOn   = TRUE       ACOn   = FALSE    AcOn   = FALSE
HeatOn = TRUE    HeatOn = EITHER     HeatOn = FALSE    HeatOn = FALSE
Running= TRUE    Running= TRUE       Running= TRUE
```

Figure 3: Example of an omitted event.

```
Error on line 47 in variable HeatOn:
        no transition from value FALSE to value TRUE.
Error on line 47 in modeclass Operating:
        no transition from mode(s) {AC} to mode(s) {HEAT}.
                         .....
                **** Unused Transitions: ****
FROM                EVENT                                        TO
--------------------------------------------------------------------
Inactive  @T(BelowTemp) WHEN [Running]                           Heat

VARIABLE            EVENT                                        VALUE
--------------------------------------------------------------------------
HeatOn    @T(BelowTemp) WHEN [(Operating(Inactive) AND Running)] TRUE

              **** Computed Invariants: ****
                   Modeclass Operating:
Mode Heat:        Mode AC:           Mode Inactive:    Mode OFF:
ACOn   = EITHER   ACOn   = TRUE      ACOn   = FALSE    ACOn   = FALSE
HeatOn = TRUE     HeatOn = EITHER    HeatOn = FALSE    HeatOn = FALSE
```

Figure 4: Example of unused mode transitions.

```
Error on line 32:
        Condition Operating(Heat) AND HeatOn does not satisfy the current
        state of the program.  Processing the rest of the block is aborted.
                         .....
                **** Unused Transitions: ****
FROM                EVENT                                     TO
--------------------------------------------------------------------
Heat            @T(TempOK) WHEN [Running]                     Inactive

 VARIABLE           EVENT                                        VALUE
--------------------------------------------------------------------------
 HeatOn       @F(BelowTemp) WHEN [(Operating(Heat) AND Running)] FALSE

             **** Unreachable Mode Transitions: ****
                      on line 35
```

Figure 5: Example of an invalid condition.