Automatic Analysis of Consistency between Implementations and Requirements: A Case Study*

Marsha Chechik John Gannon

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

Abstract

Formal methods like model checking can be used to demonstrate that safety properties of event-based 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. This paper describes a tool, called Analyzer, which discovers instances of inconsistency and incompleteness in implementations, and a case study of its use. Analyzer uses requirements information to check that all state transitions implemented in the code are exactly those specified in the requirements.

If programmers annotate their implementations with comments describing changes of values of requirements variables, our tool checks it against the requirements. Analyzer can also verify that the implementation is a model of user-specified safety properties. The tool can perform interprocedural analysis and verify properties which involve multiple state machines.

In our case study, we analyzed the requirements and the implementation of a water-level monitoring system and discovered instances of inconsistency (testing incorrect conditions) and incompleteness (missing timing constraints) in the implementation.

1 Background

Reactive systems can be specified by requirements which describe events causing state transitions. Formal methods can be used to demonstrate that safety properties are enforced by the system's requirements. However, we lack similar techniques to determine that an implementation of a system is consistent with its requirements, i.e., the implementation contains exactly the transitions specified in the requirements. We have developed a prototype tool, called Analyzer[4] which discovers instances of inconsistency and incompleteness in implementations by automatically establishing and verifying global properties from local annotations. 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 (see Figure 1). Using data-flow analysis on these comments and the program's control flow, the tool computes sets of possible values that each requirements variable can attain at every state of the program. This information is then used to compute the events which cause state transitions in the implementation. The tool checks that the implementation's state transitions are enabled by the same events as those of the requirements; that all the requirement's state transitions appear in the implementation, and that the requirement's safety properties hold in the implementation.

In this paper, we describe results of a case study in which we annotated and analyzed an implementation of a water-level monitoring system (WLMS)[14] (Section 2). To perform the study, we extended the capabilities of our tool to analyze realistic size programs containing function calls, and to verify invariant assertions. We present our algorithms for intraprocedural (Section 3) and interprocedural (Section 4) cases separately. We also present our techniques to verify assertions about the program under analysis (Section 5). We discovered that the WLMS implementation used different variables for making transition decisions than specified by the requirements, that certain timing conditions specified in the requirements were omitted from the implementation, and that safety invariants were maintained by the implementation. Section 6 presents a summary of our results and a comparison of our approach with other static analyzers.

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

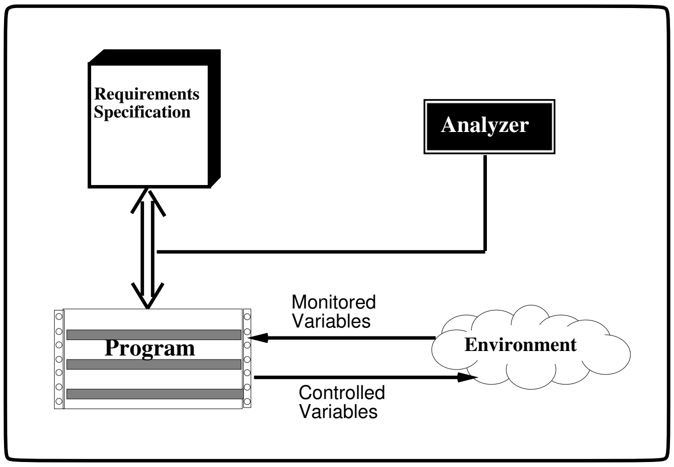

Figure 1: Overview of the problem.

2 Case Study

The 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, SelfTst and Reset, permit the operator to test the system and return it to normal operation.

2.1 WLMS Requirements

Our analysis tool uses a slightly restricted version of SCR specifications [2, 9, 8] represented in the ASCII format[4]. A system is modeled as a set of concurrently executing state machines, where each machine interacts with its environment's state variables. Changes to its *monitored* state variables may cause the system to change its state or to alter the values of its *controlled* state variables (see Figure 1). A state machine represents an SCR *mode class*, whose states are *modes* and whose transitions occur in response to *events*. Each mode transition is assumed to take one time unit. The values of controlled variables may also change in response to events, again taking one time unit.

WLMS has two mode classes, Normal and Failure, whose modes are described in Table 1. The system starts in mode Standby of mode class Normal and mode AllOK of mode class Failure.

We use boolean variables to represent predicates on monitored variables. 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. Table 2 shows some monitored variables and the predicates they represent.

Table 3 depicts transitions between the modes of mode class Normal. The system starts in the mode Standby and transitions to Operating after the Reset button has been pressed for three seconds while the water level is within its hysteresis limits. The system continues in mode Operating unless the water level limits are violated while the SelfTest button is not being pressed, or the operator initiates a system test by pushing the SelfTest button for more than 500 milliseconds.

Many requirements variables are dependent on each other. We record these dependencies using the following relationships:

- An *implication* (B $\longrightarrow$ C) is a relationship which indicates that when B is True, C should be True; and when C is False, B should be False.
- A *timeline* relationship (T1 < T2) exists between variables which represent different lengths of time that a particular condition holds. 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 (B1 | B2 | B3) indicates that exactly one of the conditions is True at all times.

We declare InsideHysR $\longrightarrow$ WithinLimits and SlfTstPressed < SlfTstPressed500 to be the relationships

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 1: WLMS Modes.

Variable	Predicate
WithinLimits	LowLimit $<$ WaterLevel $<$ HighLimit
InsideHysR	(LowLimit + 0.5cm) $<$ WaterLevel $<$ (HighLimit $-$ 0.5cm)
SlfTstPressed	SelfTst button is being pressed
SlfTstPressed500	SelfTst button is pressed constantly for 500ms or more

Table 2: Selected WLMS monitored variables.

between the monitored variables in the system.

In addition to relationships, the requirements designer might want to specify safety properties of the system. Such properties duplicate information stored in the mode transition tables, but capture the desired information compactly. We allow the following properties to be specified:

- inv(Property) is satisfied if Property holds in every system state. Although Property can be an arbitrary propositional logic formula, it is generally in the form (mode $\rightarrow$ property).
- reach(Property) is satisfied if the system can reach a state in which Property holds. reach is used to ensure that cause and invariant properties are not satisfied vacuously.
- cause(Property_1, Property_2) is satisfied if whenever Property_1 is True, the system either satisfies Property_2, or will next enter a state which satisfies Property_2.

Four safety properties, same as used in [3], were verified in this study. They are shown in Table 4. 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.

2.2 WLMS Implementation

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. After that, we annotated the program.

Annotations are user-specified comments which record mode transitions and changes of the values of controlled and monitored 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. Annotations specify an abstraction between the implementation and the requirements. We identified three types of annotations described in detail in [5]. An *Initial annotation* indicates the starting state of the program. It unconditionally assigns values to variables and corresponds to initialization information specified in the requirements. An *Update annotation* assigns values to controlled, monitored or mode class variables. This annotation identifies points at which the program changes its state. Analyzer uses Update an-

Current Mode	Inside HysR	WithIn Lmts	SlfTst Prssd	SlfTst Prssd500	InTest 14000	Reset Prssd3000	Shutdown LckTm200	**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 3: Mode transition table for mode class Normal.

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

Table 4: Properties of the WLMS.

notations to compute possible values of requirements variables (system state). An *Assert annotation* asserts that variables have particular values in the current system state and helps make the analysis more precise. Static analysis usually gives imprecise results due to aggregating system states, and so Assert annotations are used to indicate information that the programmer *knows* to be true at a particular point in the program. Since these annotations represent information that *must* be true (as opposed to system state which captures information that *may* be true), values of variables in Assert annotations should always appear in the system state.

We annotated the program with 62 Update (32 for monitored variables and 30 for controlled variables and mode classes) and 33 Assert annotations. Annotations capture local information about the state variables, and we were able to easily identify which program variables corresponded to requirements variables. For mode classes and controlled variables, Update annotations were typically inserted at statements which assigned values to corresponding program variables, and Assert annotations were inserted at If statements. Values of monitored variables were received from the input, and Update annotations for these variables usually had to be placed at If statements. This regular pattern of annotation placement gave us an idea to insert annotations automatically, using user-specified templates. This work is still in progress.

2.3 Improvements to Analyzer

During this case study, we found out that we need a richer annotation language and better analysis techniques than were implemented in [4] to deal with multiple mode classes. The following code fragment controls the WLMS audio alarm. When the If test fails, all we know about the state of the system is that it cannot be in one of the current modes. Thus, we need to be able to represent negations and disjunctions of modes.

```
if ((MODE_EQ(OPM, Operating)) && (ALOK)) {
  @@ Assert Normal(Operating) & Failure(AllOK);
  ...;
}
else {
  @@ Assert ~Normal(Operating) OR ~Failure(AllOK);
    if ((MODE_EQ(OPM, Shutdown)) && (ALOK)) {
      @@ Assert Normal(Shutdown) & Failure(AllOK);
      ...;
    }
}
```

Negating mode class variables simply produces a set of all the other possible modes of the mode class. For example, ∼Failure(AllOK) produces the value {Failure(BadLevDev), Failure(HardFail)}. However, dealing with the disjunction in the Assert annotation is more complex. Analyzer takes several ORed conditions in disjunctive normal form and stores them in a list. At join nodes, the assertion is then a pair-wise intersection of clauses of each predecessor list. In the example below,

```
if (..){
    @@ Assert A OR B;
}
else {
    @@ Assert C OR D;
}
```

the resulting condition on the join is

```
@@ Assert (A AND C) OR (A AND D) OR
          (B AND C) OR (B AND D);
```

From this example, one might infer that we get combinatorial growth in the number of clauses in our conditional expressions but that is not the case. Each of the clauses is a possible state of the system. Since the system can only be in a finite number of states, the size of each expression is limited to a (small) number of clauses.

The implementation contained functions, so we needed a way to perform interprocedural analysis. Out of 54 functions that the implementation contained, only 8 had annotations. An algorithm we devised is very precise and reasonably efficient (see Section 4).

2.4 Results of WLMS Analysis

Our analysis tool keeps sets of all attainable values of the requirements variables, and thus overestimates the state information. However, this approach resulted in remarkably fast (faster than compiling) and reasonably precise analysis. Table 5 summarizes inconsistencies between the requirements and the implementation, which were reported by Analyzer.
These numbers overestimate the actual errors in the implementation. 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 that the operator pressed the SelfTest and Reset buttons were not calculated or checked; and no transitions to a mode corresponding to the complete system failure were implemented. Most of the illegal assignments to the controlled variables occurred because the order of triggering events in the implementation was different than specified 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:

```
inv(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.

3 Implementation

This section sketches our analysis technique. For more detail, please refer to [5].

Analyzer builds a control flow graph CFG of the implementation and uses data-flow analysis ([1]). Figure 2 outlines the basic analysis steps. The reaching definition `RD` of a variable at a node is a set of values that the variable could attain if the control reaches this node. `RD`s are calculated for each node via a least fixed point computation. The fixed point computation may overestimate a basic block's `RD`s because it assumes that each predicate may evaluate to either True or False.

Assertion annotations are propagated to each node by calculating `Cond`s. Condition information `Cond` of a variable at a node is a value that this variable has to attain if the control reaches this node. This computation is done in a similar fashion, by computing a greatest fixed point. `Cond` information permits programmers to indicate the conditions under which paths are executable and reduces the number of values in the set of reaching definitions. To understand how `Cond`s restrict `RD`s, we need to consider an abstract interpretation framework[7]. In this framework, values of controlled and monitored variables and those of mode class variables each form lattices (Figure 3). Thus the states of the program also form a lattice.

The lattice in Figure 3a is used to compute `RD`s for boolean-valued variables. Initially, variables are in a *Undefined* state (bottom of the lattice, $\perp$). Update annotations change the values of variables to either True or False. At join nodes, where values of a variable in each of the predecessors of the join are unioned, unions of True and False values produce the value *TorF* (top of the lattice, $\top$). Conversely, for `Cond`s, $\top$ represents a *Known To Be Unknown* value, and $\perp$ represents a *Known To Be True And False* value, which is effectively unreachable. Table 6 shows how `RD` and `Cond` information is combined to compute values of boolean variables. For example, when a variable's `RD` value is TorF and its `Cond` value is True, we infer that the variable has value True. Entries marked with "X" indicate errors. An annotation error is reported if a variable with an undefined `RD` value is restricted by a True `Cond`

Inconsistencies	Number
Illegal mode transitions	3
Illegal assignments to controlled variables	5
Unexercised mode transitions	9
Unexercised assignments to controlled variables	25

Table 5: Results of analyzing the WLMS.

value. The two X's in the top row of the table are caught by the restriction that assertions cannot refer to undefined variables. (Step 4 of the algorithm in Figure 2.) The remaining X's are caught by checking that $\texttt{Cond} \sqsubseteq (\texttt{Cond} \sqcap \texttt{RD})$ in Step 6.

Figure 4 gives a sample computation of `RD`s and `Cond`s. (`Cond`s are shown in italic, and `RD`s in regular font.) In this example, M1 and M2 are modes of a mode class M, and A, B and C are either controlled or monitored variables. Nodes of the graph which affect the computation of reaching definitions are shaded. The processing starts with the `RD` set {A=TorF, B=TorF, C=undefined, M={M1}} which was generated as a result of an Initial annotation. The first Assert restricts values of A to True. The Assert on the left branch restricts the value of B to True, and the Update changes the value of C to True in both the subsequent `Cond` and `RD` sets. The right branch is similar. On the join, we union the possible values for variables in `RD` and `Cond`. For the `RD` set, C is True on the left branch and False on the right, so on the join the value of C is TorF. For the `Cond` set, C is True on the left branch and False on the right, so no information is available about it on the join, thus the value of C is *Known To Be Unknown*.

To compute events enabling a mode transition, we start at an Update annotation and work backwards through the branches of the graph until we reach a node containing Update annotation on each branch. Taking the union of the `gen` sets of these nodes produces the event's Triggering condition. Our definition of Triggering conditions is weaker than the corresponding SCR definition since we do not ensure that a variable changes its value as a result of annotations we encounter. In our example, during the computation of an event for transition into mode M2 of mode class M (marked by an arrow), we find nodes with annotations "Update C" and "Update ∼C". Thus the triggering condition is {@T(C), @F(C)}. Intersecting the `RD` and `Cond` sets at the join node produces the event's When condition. In this example, the When condition is {M={M1}, A=True, B=TorF, C=TorF}. Mode transitions in the requirements are marked as implemented if their When and Triggering conditions are subsets of the computed When and Triggering conditions.

4 Processing Functions

We process functions using a technique called *cloning* [6] which we adapted for our analysis. A similar algorithm was described in [15]. This technique enables Analyzer to process programs with cycles in their call graphs (recursion), to analyze each called function only a constant 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 reaching definitions or 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. We verify each copy of a function once with its own set of reaching definitions and conditions. Our algorithm is described in detail in [5]. Example below:

```
int x;                          void F() {
int main() {                       @@ Assert Xge0;
   @@ Initial M: M(M0);            if (x > 0) {
   x = 3;                             @@ Update Xgt0;
   @@ Update Xgt0;                    x--;
   F();                               F();
   return 1;                       }
}                                  else {
                                      @@ Update Xeq0;
                                      @@ Update M(M1);
                                   }
                                }
```

illustrates how the computation of reaching definitions is performed in the presence of function calls. While computing reaching definitions and encountering a call to F from the main routine, Analyzer creates a copy of the F's CFG in the environment where the initial information (`in`) is {M={M0}, Xgt0=True, Xge0=True}, and marks the environment gray. This `in` is used as the reaching definition at the start of the function, and

Algorithm

1. Build basic-block graph
2. Compute `gen` and `kill` ([1]) sets for each Update and Initial annotation
3. Compute `RD`s
4. For each Assert annotation, check that its variables are defined
5. Propagate `Cond`s throughout the graph
6. Traverse basic-block graph
   For each node
      Check that the assertion corresponds to the system state, that is, enforce
      $\texttt{Cond} \sqsubseteq (\texttt{Cond} \sqcap \texttt{RD})$
   For each Update annotation
      Compute an event using only potentially executable paths
      Check if the event can change the mode or the value of a controlled variable
      in the current state (according to the requirements)
7. Verify properties of the system
8. Output error messages and other information

End of Algorithm

Figure 2: Basic Analysis steps.

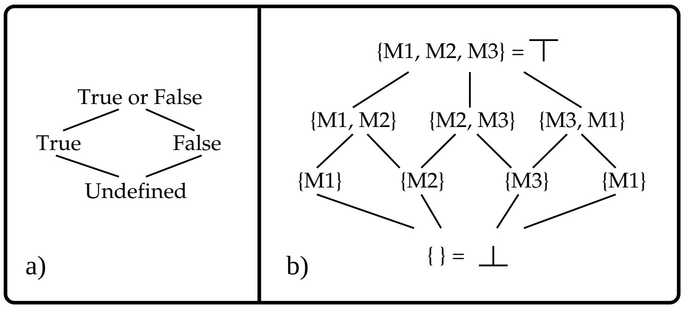

Figure 3: Lattice structure of requirements variables.

the computation is continued through the CFG of F. When a function calls itself, the reaching definitions of the call site are identical to that of the original call, so Analyzer skips the recursive call and sets the output information (`out`) of the call site to its `in`. Processing the non-recursive branch of F yields the context {M={M0}, Xeq0=True, Xge0=True}. When these two sets are merged at join node at the end of F, F's `out` set is {M={M0, M1}, Xgt0=True, Xge0=True, Xeq0=True}. This also becomes the `out` set of main's call to F. Similar computation occurs for condition propagation except that both the context and the condition should match in order for the recursive call to be skipped.

5 Verifying Safety Properties

Verifying a property like inv(M(M1) $\longrightarrow$ A) is more complicated than simply finding all the system states which satisfy M={M1} and checking that A is True in these states. Our analysis must ignore paths which are known to be infeasible (e.g., the Else clause in our sample program in which the invariant does not hold), ignore program states occurring between those which enable events and those in which a state transition occurs, and deal correctly with variables whose reaching definition values are TorF.

If TorF were considered as neither True nor False in verifying invariants, Analyzer would always report errors in the system states at join nodes. In the example below,

RD \ Cond	True	False	⊤
⊥	X	X	⊥
True	True	X	True
False	X	False	False
⊤	True	False	⊤

Table 6: Computation of an actual value of a boolean variable.

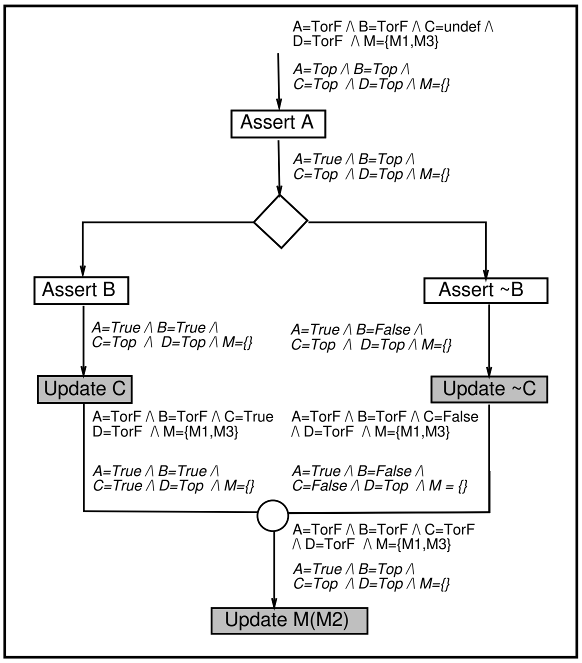

Figure 4: Computation of conditions and reaching definitions.

```
@@ Initial M: M(M0) and ~A;
if (...) {
    @@ Update A;
    @@ Update M(M1);
}
else {
    @@ Update M(M2);
}
```

the system state at the join node is {M={M1, M2}, A=TorF}, which violates inv(M(M1) $\longrightarrow$ A). If the value TorF were treated as being both True and False, the invariant would hold at the join node. This interpretation is safe since we can guarantee that all predecessors of the join node which violated the invariant would be identified. If A=TorF was obtained by merging states {M={M2}, A=True} and {M={M1}, A=False}, Analyzer detects that the latter state violates the invariant. If A=TorF was obtained by merging states with {M={M1}, A=True} and {M={M2}, A=False}, there is no error.

To verify a reachability property, Analyzer looks for a state which satisfies the property. If none is found, it considers the assertion to be violated. In this case, we must also deal with non-executable paths and TorF values. Here we treat TorF as *not* satisfying either of the possibilities to avoid overlooking errors. For example, we would not want to verify that reach (M(M1) & ∼A) was satisfied by the state at the join.

Finally, we can make the computation of invariants more precise by utilizing the conditions stored in each state. A state is considered to violate the invariant if an intersection of *each* of the condition disjuncts with the reaching definition violates the invariant.

6 Summary and Related Work

We have found Analyzer to be a useful prototype. It amortizes the cost of creating requirements and safety properties by using them to analyze implementations. We have demonstrated that Analyzer can detect inconsistencies in implementations which were developed to conform to SCR requirements.

Analyzer uses concepts from Quick Defect Analysis (QDA)[10] and Cesar/Cecil[11, 12, 13]. In QDA, 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 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 (AQREs). After a user specifies a mapping from programming language constructs to Cecil events, the Cesar analyzer determines if the implementation meets Cecil constraints.

Analyzer and QDA both use local, informal descriptions of state changes specified by annotations. These are probably easier for analysts to write than AQREs because they do not require any knowledge of predicate calculus. QDA and Cesar/Cecil both require users to annotate functions with summary information describing all their behaviors, while Analyzer chooses to analyze functions once for each new context. The Cesar/Cecil system greatest strength is that the events it analyzes are tied to source code statements, rather than annotations which describe state changes which may or may not be implemented by the code. We are currently exploring ways of creating mappings from source code variables to state variables to help determine where annotations are incorrect or have been omitted.

Acknowledgements

We thank Rich Gerber and Bill Pugh for many helpful comments.

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

[5] M. Chechik and J. Gannon. "Automatic Analysis of Consistency between Implementations and Requirements". Technical report CS-TR-3394, Dept. of CS, University of Maryland, College Park, January 1995. (in preparation).

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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