Coverage Metrics for Requirements-Based Testing [*]

Michael W. Whalen
Advanced Technology Center
Rockwell Collins Inc.
mwwhalen@rockwellcollins.com

Ajitha Rajan
Dept. of Comp. Sci. and Eng.
University of Minnesota
arajan@cs.umn.edu

Mats P.E. Heimdahl
Dept. of Comp. Sci. and Eng.
University of Minnesota
heimdahl@cs.umn.edu

Steven P. Miller
Advanced Technology Center
Rockwell Collins Inc.
spmiller@rockwellcollins.com

ABSTRACT

In *black-box* testing, one is interested in creating a suite of tests from requirements that adequately exercise the behavior of a software system without regard to the internal structure of the implementation. In current practice, the adequacy of black box test suites is inferred by examining coverage on an executable artifact, either source code or a software model.

In this paper, we define structural coverage metrics directly on high-level formal software requirements. These metrics provide *objective, implementation-independent* measures of how well a black-box test suite exercises a set of requirements. We focus on structural coverage criteria on requirements formalized as LTL properties and discuss how they can be adapted to measure finite test cases. These criteria can also be used to automatically generate a requirements-based test suite. Unlike model or code-derived test cases, these tests are immediately traceable to high-level requirements. To assess the practicality of our approach, we apply it on a realistic example from the avionics domain.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Verification

[*]This work has been partially supported by NASA Langley's Aviation Safety and Security Program (AvSSP) under contract NCC-01001.

1. INTRODUCTION

In *black-box* testing, one is interested in creating a suite of tests from requirements that adequately exercise the behavior of a software system without regard to the internal structure of the implementation. This is the preferred style of testing in safety-critical domains and is advocated by standards such as DO-178B [27]. Currently, there is no objective standard for directly determining the adequacy of a black-box test suite given a set of requirements. Instead, the adequacy of such suites are inferred by examining different coverage metrics on an executable artifact, either source code [4, 5] or software models [1, 22].

There are several problems with using the executable artifacts to measure the adequacy of black-box tests. First, it is an indirect measure: if an implementation is missing functionality, a weak set of black-box tests may yield structural coverage of the implementation and yet not expose defects of omission. Conversely, if black-box tests yield poor coverage of an implementation, an analyst must determine whether it is because (a) there are missing or implicit requirements, (b) there is code in the implementation that is not derived from the requirements, or (c) the set of tests derived from the requirements was inadequate. Finally, an executable artifact is necessary to measure the adequacy of the test suite. This may mean that the adequacy of a test suite cannot be determined until late in the development process.

Generally, requirements are defined informally as, for example, "shall" statements or use-cases. However, recent efforts (e.g., [19]) have formalized and verified software requirements using notations such as temporal logics [9] and synchronous observers [12]. Given formal requirements, it is possible to define meaningful coverage metrics *directly on the structure of the requirements.*

In this paper, we present coverage metrics on formal high-level software requirements. These metrics are desirable because they provide *objective, implementation-independent* measures of how well a black-box test suite exercises a set of requirements. Further, given a set of test cases that achieve a certain level of structural coverage of the high-level requirements, it is possible to measure model or code coverage to objectively assess whether the high-level requirements have been sufficiently defined for the system. This approach yields several objective measurements that are not possible with traditional testing techniques, and integrates and cross-checks several of the validation and verification activities.

In addition, using the coverage metrics and an executable formal model, it is possible to *autogenerate* requirements-based test cases from properties. This idea has also been explored by Tan et al. [28], and in several efforts at the model and/or source code level [1, 24, 11]. The benefit from generating tests from properties as opposed to models or code is that the generated test cases can be immediately traced back to a high-level requirement of interest. This aspect may be helpful to satisfy the testing guidelines of rigorous development processes such as DO-178B [27].

The idea of property metrics has also been explored theoretically by Tan et al. [28], using metrics based on *vacuity* [3, 16, 21]. Our work provides new metrics which we believe to be more practical for measuring coverage given black-box tests and discusses some of the implications of property-based testing. To evaluate the requirements coverage metrics and the practicality of autogenerating requirements-based tests, we have implemented a property-based test case generator and applied our techniques to a realistic system from the civil avionics domain.

The remainder of the paper is organized as follows. Section 2 presents a case example of the flight guidance system we use to illustrate our ideas. Our contributions to requirements-based testing and the test coverage criteria used are introduced in Section 3. Section 4 discusses how we generate requirements-based tests for the FGS example and the results obtained from running the tests. Related work is discussed in Section 5. Finally, Section 6 discusses the implications of the results and points to future work in requirements-based test case generation.

2. THE FLIGHT GUIDANCE SYSTEM

To illustrate our approach we will use an example from commercial avionics—a Flight Guidance System (FGS). A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this paper we focus on the mode logic of the FGS. The requirements and implementation model used in this paper are described in [18] and are similar to production systems created by Rockwell Collins Inc.

3. REQUIREMENTS-BASED TESTING

There is a close relationship between the high-level requirements and the properties captured for verification purposes. As an example, consider the requirement from the sample FGS shown in Table 1 defining how the Flight Director (FD) is turned on by the Autopilot (AP). The property states that it is globally true (G) that if the Onside FD is not on and the AP is not engaged, in the next instance in time (X) if the AP is engaged, then the Onside FD will also be on. In [19], the full set of informal requirements for the FGS were translated into 293 properties in Linear Time Temporal Logic (LTL) [9] and verified over an implementation model of the mode logic using the NuSMV model checker. In most cases, as in Table 1, the LTL property is very similar in structure to the natural language requirement and the translation was straightforward.

"If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged."

(a)

$$G((\neg Onside\_FD\_On \ \wedge \ \neg Is\_AP\_Engaged) \ \rightarrow X(Is\_AP\_Engaged \rightarrow Onside\_FD\_On))$$

(b)

Table 1: (a) Sample high-level requirement on the FGS (b) LTL property for the requirement

An analyst developing test cases from the informal requirements might derive the scenario in Table 2 to demonstrate that the requirement is met. Does this adequately cover this high-level requirement? Does passing such a test case indicate that the model has correctly captured the behavior required through this requirement? If not, what would additional test cases look like? The specification of the requirement as a property allows us to define several objective criteria with which to determine whether we have adequately tested the requirement. We hypothesize that coverage of such criteria can serve as a reliable measure of the thoroughness of the requirements-based testing activities.

1. Turn the Onside FD off
2. Disengage the AP
3. Engage the AP
4. Verify that the Onside FD comes on

Table 2: Manually developed requirements-based test scenario

To define suitable coverage metrics, it is possible to adapt several existing code and modeling coverage metrics, for example, [26, 23, 20, 11]. A distinction must be made, however, between coverage metrics over source code and coverage metrics over requirements. Metrics over code assume that Boolean expressions can take on both 'true' and 'false' values. When generating tests from requirements, we usually are interested in test cases exercising the different ways of satisfying a requirement (i.e., showing that it is true). Test cases that presume the requirement is 'false' are not particularly interesting; this is discussed in further detail in Section 3.1.2.

In this paper, we focus on structural coverage metrics over LTL. Nevertheless, there are several other notations that can be used to describe high-level requirements. For example, SCADE [29] and Reactis [25] use *synchronous observers* [12]. Synchronous observers are small specifications of high-level requirements written as state machines or in the same notation as the software specification, and they run "in parallel" with the model. The notion of requirements coverage can now be viewed as structural coverage over the observer. We will consider requirements formulated as synchronous observers and the notion of requirements coverage applied to them in our future work.

3.1 Structural Coverage over LTL Syntax

Structural coverage criteria defined for source code and for executable modeling languages can be adapted to fit propositional temporal logics such as CTL and LTL. For instance, decision coverage of the requirements would require a single test case that demonstrates that the requirement is satisfied, such as the manually developed one in Table 2. Typically, this single positive test case is too weak of a coverage since it is often possible to derive a useless test case. If we again consider our sample requirement and formalization from Table 1, we can satisfy the decision coverage metric by creating a test case that leaves the autopilot disengaged throughout the test and disregards the behavior of the flight director. Although this test case technically satisfies the property, it does not shed much light on the correctness of our model.

A better alternative would be to adopt one of the more rigorous structural coverage criteria such as vacuity coverage [3] or the Modified Condition/Decision Coverage (MC/DC) criterion [5, 13], for use in the requirements-based testing domain. In this section, we define a requirements coverage metric called *Unique First Cause Coverage* (UFC) that is adapted from MC/DC criterion.

MC/DC is a structural coverage metric that is designed to demonstrate the independent effect of basic Boolean conditions (i.e., subexpressions with no logical operators) on the Boolean decision (expression) in which they occur. A test suite is said to satisfy MC/DC if executing the test cases in the test suite will guarantee that:

- every point of entry and exit in the model has been invoked at least once,
- every basic condition in a decision in the model has taken on all possible outcomes at least once, and
- each basic condition has been shown to independently affect the decision's outcome

Note that each instance of a condition within a formula is treated separately: the formula $(A \wedge B) \vee A)$ has three basic conditions, and we must determine the independence of each instance of A separately. In this paper we use *masking* MC/DC [13] to determine the independence of conditions. In masking MC/DC, a basic condition is *masked* if varying its value cannot affect the outcome of a decision due to structure of the decision and the value of other conditions. To satisfy masking MC/DC for a basic condition, we must have test states in which the condition is not masked and takes on both 'true' and 'false' values.

3.1.1 MC/DC Coverage of Decisions

In masking MC/DC, the masking criteria are defined over Boolean operators (and, in the case of imperative programs, loops and selection statements). For example, given the expression $A \wedge B$, to show the independence of B, we must hold the value of A to true; otherwise varying B will not affect the outcome of the expression. When we consider decisions with multiple Boolean operators, we must ensure that the test results for one operator are not masked out by the behavior of other operators. For example, given $A \vee (B \wedge C)$ the tests for $B \wedge C$ will not affect the outcome of the decision if A is true.

It is straightforward to describe the set of required MC/DC assignments for a decision as a set of Boolean expressions. Each expression is designed to show whether a particular condition positively or negatively affects the outcome of a decision. That is, if the expression is true, then the corresponding condition is guaranteed to affect the outcome of the decision. Given a decision A, we define A^+ to be the set of expressions necessary to show that all of the conditions in A *positively* affect the outcome of A, and A^- to be the set of expressions necessary to show that the all of the conditions in A *negatively* affect the outcome of A.

We can define A^+ and A^- schematically over the structure of complex decisions as follows:

$x^+ = \{x\}$ (where x is a basic condition)
$x^- = \{\neg x\}$ (where x is a basic condition)

The positive and negative test cases for conditions are simply the singleton sets containing the condition and its negation, respectively.

$$(A \wedge B)^+ = \{a \wedge B \mid a \in A^+\} \cup \{A \wedge b \mid b \in B^+\}$$
$$(A \wedge B)^- = \{a \wedge B \mid a \in A^-\} \cup \{A \wedge b \mid b \in B^-\}$$

To get positive MC/DC coverage of $A \wedge B$, we need to make sure that every element in A^+ uniquely contributes to making $A \wedge B$ true while holding B true, and symmetrically argue for the elements of B^+. The argument for negative MC/DC coverage is the same, except we show that $A \wedge B$ is false by choosing elements of A^- and B^-.

$$(A \vee B)^+ = \{a \wedge \neg B \mid a \in A^+\} \cup \{\neg A \wedge b \mid b \in B^+\}$$
$$(A \vee B)^- = \{a \wedge \neg B \mid a \in A^-\} \cup \{\neg A \wedge b \mid b \in B^-\}$$

To get positive (negative) MC/DC coverage over $A \vee B$, we need to make sure that every element in A^+ (A^-) uniquely contributes to making $A \vee B$ true (false) while holding B false, and the symmetric argument for elements of B^+ (B^-).

$$(\neg A)^+ = A^-$$
$$(\neg A)^- = A^+$$

The positive and negative MC/DC coverage sets for $\neg A$ swap the positive and negative obligations for A.

Each of the expressions in the positive and negative sets can be seen as defining a constraint over a program or model state. The process of satisfying MC/DC involves determining whether each of these constraints is satisfied by some state that is reached by a test within a test suite.

3.1.2 Unique-First-Cause (UFC) Coverage

Since requirements captured as LTL properties define paths rather than states, we broaden our view of structural coverage to accommodate satisfying paths rather than satisfying states. The idea is to measure whether we have sufficient tests to show that all atomic conditions within the property affect the outcome of the property. We can define these test paths by extending the constraints for state-based MC/DC to include temporal operators. These operators describe the path constraints required to reach an acceptable state. The idea is to characterize a trace $\pi = s_0 \rightarrow s_1 \rightarrow \ldots$ in which the formula holds for states $s_0 \ldots s_{k-1}$, then passes through state s_k, in which the truth or falsehood of the formula is determined by the atomic condition of interest. For satisfying traces, we require that the formula continue to hold thereafter.

A test suite is said to satisfy UFC coverage over a set of LTL formulas if executing the test cases in the test suite will guarantee that:

- every basic condition in a formula has taken on all possible outcomes at least once
- each basic condition has been shown to independently affect the formula's outcome.

We define independence in terms of the shortest satisfying path for the formula. Thus, if we have a formula A and a path π, an atom α in A is the unique first cause if, in the first state along π in which A is satisfied, it is satisfied because of atom α. To make this notion concrete, suppose we have the formula $\mathrm{F}(a \vee b)$ and a path $P = s_0 \rightarrow s_1 \rightarrow \ldots$ in which a was initially true in step s_2 and b was true in step s_5. For path P, a (but not b) would satisfy the unique first cause obligation. It is possible to generalize this definition to *unique cause*, which states that an atom α is necessary for the satisfaction of A, but space concerns prevent us from covering both formulations in this paper.

The definition above takes the evenhanded view that all formulas will be *satisfiable*, that is, there are some traces in which the property is true and some in which it is false. Nevertheless, we eventually want a system in which all of the properties (requirements) are *valid*. Therefore, we are primarily concerned with the *positive* set of test cases for a formula. It is necessary, however, to define the negative UFC constraint sets since when an LTL formula, say f, is negated, the positive UFC constraints of $\neg f$ is evaluated as the negative UFC constraints of f.

For the formalization of UFC coverage of requirements expressed as LTL properties we will use the notational conventions that were defined above for Boolean expressions and extend them to include temporal operators in LTL.

We extend A^+ and A^- defined over states to define satisfying paths over LTL temporal operators as follows:

$\mathrm{G}(A)^+ = \{A \ \mathrm{U} \ (a \ \wedge \ \mathrm{G}(A)) \mid a \in A^+\}$
$\mathrm{F}(A)^- = \{\neg A \ \mathrm{U} \ (a \wedge \mathrm{G}(\neg A)) \mid a \in A^-\}$

$\mathrm{G}(A)$ is true if A is true along all states within a path. The $A \ \mathrm{U} \ (a \ \wedge \ \mathrm{G}(A))$ formula ensures that each element a in A^+ contributes to making A true at some state along a path in which A is globally true.

$\mathrm{F}(A)^-$ is the dual of $\mathrm{G}(A)^+$, so the obligations match after negating A and a.

$\mathrm{F}(A)^+ = \{\neg A \ \mathrm{U} \ a \mid a \in A^+\}$
$\mathrm{G}(A)^- = \{A \ \mathrm{U} \ a \mid a \in A^-\}$

The independent effect of $a \in A^+$ for the $\mathrm{F}(A)$ formula is demonstrated by showing that it is the first cause for A to be satisfied. Similar to the previous definition, $\mathrm{G}(A)^-$ is the dual of $F(A)^+$.

$\mathrm{X}(A)^+ = \{\mathrm{X}(a) \mid a \in A^+\}$
$\mathrm{X}(A)^- = \{\mathrm{X}(a) \mid a \in A^-\}$

The independent effects of $a \in A^+$ (resp. $a \in A^-$) are demonstrated by showing that they affect the formula in the next state.

$(A \ \mathrm{U} \ B)^+ =$
$\{(A \ \wedge \neg B) \ \mathrm{U} \ ((a \ \wedge \neg B) \wedge (A \ \mathrm{U} \ B)) \mid a \in A^+\} \ \cup$
$\{(A \ \wedge \neg B) \ \mathrm{U} \ b \mid b \in B^+\}$

For the formula $A \ \mathrm{U} \ B$ to hold, A must hold in all states until we reach a state where B holds. Therefore, positive UFC coverage for this would mean we have to ensure that every element in A^+ contributes to making A true along the path and every element in B^+ contributes to completing the formula.

The formula to the left of the union provides positive UFC over A in $(A \ \mathrm{U} \ B)$. Recall that an 'until' formula is immediately satisfied if B holds. Therefore, in order to show that some specific atom in A (isolated by the formula a) affects the outcome, we need to show that this atom is necessary *before* B holds. This is accomplished by describing the prefix of the path in which a affects the outcome as: $(A \ \wedge \neg B) \ \mathrm{U} \ (a \ \wedge \neg B)$. In order to ensure that our prefix is sound, we still want B to eventually hold, we add $(A \ \mathrm{U} \ B)$ to complete the formula.

The formula on the right of the union is similar and asserts that some $b \in B$ is the unique first cause for B to be satisfied.

$(A \ \mathrm{U} \ B)^- =$
$\{(A \wedge \neg B) \ \mathrm{U} \ ((a \wedge \neg B) \mid a \in A^-\} \ \cup$
$\{(A \wedge \neg B) \ \mathrm{U} \ (b \wedge \neg(A \ \mathrm{U} \ B)) \mid b \in B^-\}$

For the formula $A \ \mathrm{U} \ B$ to be falsified, there is some state in which A is false before B is true. The formula to the left of the union demonstrates that $a \in A^-$ uniquely contributes to the falsehood of the formula by describing a path in which A holds (and B does not — otherwise the formula $A \ \mathrm{U} \ B$ would be true) until a state in which both a and B are false.

The formula to the right demonstrates that $b \in B^-$ uniquely contributes to the falsehood of the formula by describing a path in which $A \ \mathrm{U} \ B$ eventually fails, but A holds long enough to contain state b falsifying B.

The formulations above define a rigorous notion of requirements coverage over execution traces. Since we have been working with linear time temporal logic, the definitions apply over infinite traces. Naturally, test cases must by necessity be finite; therefore, the notion of requirements coverage must apply to finite traces.

3.1.3 Adapting Formulas to Finite Tests

LTL is normally formulated over *infinite* paths while test cases correspond to *finite* paths. Nevertheless, the notion of coverage defined in the previous section can be straightforwardly adapted to consider finite paths as well. There is a growing body of research in using LTL formulas as monitors during testing [17, 10, 2], and we can adapt these ideas to check whether a test suite has sufficiently covered a property.

Manna and Pnueli [17] define LTL over *incomplete* models, that is, models in which some states do not have successor states. In this work, the operators are given a best-effort semantics, that is, a formula holds if all evidence along the finite path supports the truth of the formula. The most significant consequence of this formulation is that the next operator (X) is split into two operators: X! and X, which are *strong* and *weak* next state operators, respectively. The strong operator is always false on the last state in a finite path, while the weak operator is always true.

It is straightforward to define a formal semantics for LTL over finite paths. We assume that a state is a labeling $L : V \rightarrow \{T, F\}$ for a finite set of variables V, and that a finite path π of length k is a sequence of states $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_k$.

1. $\pi \models true$
2. $\pi \models p$ iff $|\pi| > 0$ and $p \in L(s_1)$
3. $\pi \models \neg A$ iff $\pi \not\models A$
4. $\pi \models (A \wedge B)$ iff $\pi \models A$ and $\pi \models B$
5. $\pi \models (A \vee B)$ iff $\pi \models A$ or $\pi \models B$
6. $\pi \models X(A)$ iff $|\pi| \leq 1$ or $\pi^2 \models A$
7. $\pi \models X!(A)$ iff $|\pi| > 1$ and $\pi^2 \models A$
8. $\pi \models G(A)$ iff $\forall 1 \leq i \leq |\pi|\ \pi^i \models A$
9. $\pi \models F(A)$ iff for some $1 \leq i \leq |\pi|\ \pi^i \models A$
10. $\pi \models A$ U B iff there is some $1 \leq i \leq |\pi|$ where $\pi^i \models B$ and $\forall j = 1..(i-1), \pi^j \models A$

Table 3: Semantics of LTL over Finite Paths

Test1:

Atom	Step1	Step2	Step3
a	t	t	f
b	t	f	t
c	f	f	f

Test2:

Atom	Step1	Step2
a	t	t
b	t	f
c	f	t

Table 4: Test Suite for Property $((a\ \vee\ b)$ **U** $c)$

We write π^i for the suffix of π starting with state s_i, and the length of the path as $|\pi|$. Given these definitions, the formal semantics of LTL over finite paths is defined in Table 3. As expected, these definitions correspond with the standard semantics except that they do not require that G properties hold infinitely (only over the length of the finite path), and do not require X properties to hold in the last state of a finite path.

The semantics in Table 3 are sensible and easy to understand but may be too strong for measuring test coverage. We may want to consider tests that show the independence of one of the atoms even if they are "too short" to discharge all of the temporal logic obligations for the original property. For example, consider the formula:

$$((a\ \vee\ b)\ \mathrm{U}\ c)$$

and the test cases in Table 4. Are these two test cases sufficient to show the independent effects of a, b, and c? From one perspective, test 1 is (potentially) a prefix of a path that satisfies $((a\ \vee\ b)$ U $c)$ and independently shows that a and b affect the outcome of the formula; the test case illustrates that the formula holds with only a or only b being true. Test 2 shows the independent effect of c. From another perspective (the perspective of the finite semantics described above), test 1 does not satisfy the formula (since the finite semantics in Table 3 requires that for the until formula to hold, c must become true in the path), so cannot be used to show the independent effect of any of the atoms.

The issue with these tests (and with finite paths in general) is that there may be *doubt* as to whether the property as a whole will hold. This issue is explored in [10], which defines three different semantics for temporal operators: *weak, neutral,* and *strong.* The *neutral* semantics are the semantics of [17] described in Table 3. The *weak* semantics do not require eventualities (F and the right side of U) to hold along a finite path, and so describe prefixes of paths that may satisfy the formula as a whole. The *strong* semantics always fail on G operators, and therefore disallow finite paths if there is any doubt as to whether the stated formula is satisfied.

Since we believe that the test cases in Table 4 adequately illustrate the independence of a and b, we slightly weaken our LTL obligations. Given a formula f, we are interested in a prefix of an accepting path for f that is long enough to demonstrate the independence of our condition of interest. Thus, we want the operators leading to this demonstration state to be *neutral*[1] , but the operators afterwards can be weak.

The strong and weak semantics are a coupled dual pair because the negation operator switches between them. In [10], the semantics are provided as variant re-formulations of the neutral semantics. However, they can also be described as syntactic transformations of neutral formulas that can then be checked using the neutral semantics. We define *weak*$[F]$ to be the weakening of a formula F and *strong*$[F]$ to be the strengthening of formula F. The transformations weak and strong are defined in Table 6. We refer the reader to [10] for a full description of the three semantics and their effect on provability within the defined logic.

Given these transformations, we can re-formulate the necessary UFC paths in LTL. The idea is that we want a prefix of a satisfying path that conclusively demonstrates that a particular condition affects the outcome of the formula. To create such a prefix, we want a *neutral* formula up to the state that demonstrates the atomic condition and a weak formula thereafter. The modified formulas defining UFC over finite prefixes are shown in Table 7.

The only formulas that are changed in Table 7 from the original formulation in Section 3.1.2 are $G(A)^+$, $F(A)^-$, one branch of $(A\ \mathrm{U}\ B)^+$, and one branch of $(A\ \mathrm{U}\ B)^-$. These are the formulas that have additional obligations to match a prefix of an accepting path after showing how the focus condition affects the path.

3.1.4 Discussion

It is possible to take two views when measuring the coverage of requirements from a given test suite. The first perspective states that each test case must have sufficient evidence to demonstrate that the formula of interest is true and that a condition of interest affects the outcome of the formula. This perspective can be achieved using the *neutral* finite LTL rules and our original property formulation.

The second perspective states that each test case is a *prefix* of an accepting path for the formula of interest and that the condition of interest affects the outcome of the formula. This perspective can be achieved using the weakened UFC obligations shown in Table 7.

Making this discussion concrete, given our example formula:

$$f = ((a \vee b)\ \mathrm{U}\ c)$$

the UFC obligations for the original and modified rules are shown in Table 5. In the original formulation, the first two obligations are not satisfied by the test suite in Table 4 because c never becomes true in the first test case. In the weakened formulation, however, the requirement is covered because the first test case is a potential prefix of an accepting path.

[1]The *strong* semantics are too strong – any property containing a G-operator will be disproved.

1. $weak\ [true] = true$	12. $strong\ [true] = true$
2. $weak\ [p] = p$	13. $strong\ [p] = p$
3. $weak\ [\neg A] = \neg strong\ [A]$	14. $strong\ [\neg A] = \neg\ weak\ [A]$
4. $weak\ [A \wedge B] = weak\ [A] \wedge weak\ [B]$	15. $strong\ [A \wedge B] = strong\ [A] \wedge strong\ [B]$
5. $weak\ [A \vee B] = weak\ [A] \vee weak\ [B]$	16. $strong\ [A \vee B] = strong\ [A] \vee strong\ [B]$
6. $weak\ [X!(A)] = X(weak\ [A])$	17. $strong\ [X!(A)] = X!(strong\ [A])$
7. $weak\ [X(A)] = X(weak\ [A])$	18. $strong\ [X(A)] = X!(strong\ [A])$
8. $weak\ [G(A)] = G(weak\ [A])$	19. $strong\ [G(A)] = false$
9. $weak\ [F(A)] = true$	20. $strong\ [F(A)] = F(strong\ [A])$
10. $weak\ [A\ \mathrm{U}\ B] = weak\ [A]\ W^2 weak\ [B]$	21. $strong\ [A\ U\ B] = strong\ [A]\ \mathrm{U}\ strong\ [B]$
11. $weak\ [A\ \mathrm{W}\ B] = weak\ [A]\ W weak\ [B]$	22. $strong\ [A\ W\ B] = strong\ [A]\ \mathrm{U}\ strong\ [B]$

Table 6: Definitions of *weak* and *strong* LTL transformations

$$\begin{aligned}
G(A)^+ &= \{A\ \mathrm{U}\ (a \wedge weak[G(A)]) \mid a \in A^+\}\\
G(A)^- &= \{A\ \mathrm{U}\ a \mid a \in A^-\}\\
F(A)^+ &= \{\neg A\ \mathrm{U}\ a \mid a \in A^+\}\\
F(A)^- &= \{\neg A\ \mathrm{U}\ (a \wedge weak[G(\neg A)]) \mid a \in A^-\}\\
(A\ \mathrm{U}\ B)^+ &= \{(A\ \wedge\ \neg B)\ \mathrm{U}\ ((a \wedge \neg B) \wedge weak[A\ \mathrm{U}\ B]) \mid a \in A^+\} \cup\\
&\qquad \{(A \wedge \neg B)\ \mathrm{U}\ b \mid b \in B^+\}\\
(A\ \mathrm{U}\ B)^- &= \{(A \wedge \neg B)\ \mathrm{U}\ ((a \wedge \neg B) \mid a \in A^-\} \cup\\
&\qquad \{(A \wedge \neg B)\ \mathrm{U}\ (b \wedge (\neg weak[A\ \mathrm{U}\ B])) \mid b \in B^-\}\\
X(A)^+ &= \{X(a) \mid a \in A^+\}\\
X(A)^- &= \{X(a) \mid a \in A^-\}
\end{aligned}$$

Table 7: Weakened UFC LTL Formulas describing Accepting Prefixes

Original Fomulation:

$\{\ ((a \vee b) \wedge \neg c)\ \mathrm{U}\ (a \wedge \neg c \wedge ((a \vee b)\ \mathrm{U}\ c)),$
$((a \vee b)\ \wedge \neg c)\ \mathrm{U}\ (b \wedge \neg c \wedge ((a \vee b)\ \mathrm{U}\ c)),$
$((a \vee b) \wedge \neg c)\ \mathrm{U}\ c\ \}$

Weakened Formulation:

$\{\ ((a \vee b) \wedge \neg c)\ \mathrm{U}\ (a \wedge \neg c \wedge ((a \vee b)\ \mathrm{W}\ c)),$
$((a \vee b)\ \wedge \neg c)\ \mathrm{U}\ (b \wedge \neg c \wedge ((a \vee b)\ \mathrm{W}\ c)),$
$((a \vee b) \wedge \neg c)\ \mathrm{U}\ c\ \}$

Table 5: UFC obligations for $((a \vee b)$ U $c)$

3.2 Automatically Generating Requirements-Based Tests

Several research efforts have developed techniques for automatic generation of tests from formal models using model checkers as test case generation tools [22, 23, 20, 11]. Model checkers build a finite state transition system and exhaustively explore the reachable state space searching for violations of the properties under investigation [9]. Should a property violation be detected, the model checker will produce a counterexample illustrating how this violation can take place. In short, a counterexample is a sequence of inputs that will take the finite state model from its initial state to a state where the violation occurs.

One way to use a model checker to find test cases is by formulating a test criterion as a verification condition for the model checker. In the previous section, we described UFC over paths and defined sets of LTL formulas that were sufficient to show that a particular atomic condition affects the outcome of the property. Given this set and a formal model of the software system, we can now challenge the model checker to find a way of generating a path to satisfying one of these formulas by asserting that there is no such path (i.e., negating the formula). We call such a formula a trap formula or trap property [11]. The model checker will now search for a counterexample demonstrating that this trap property is, in fact, satisfiable; such a counterexample constitutes a test case that will show the UFC obligation of interest over the model. By repeating this process for all formulas within the set derived from a property, we can derive UFC coverage of the property over the model. By performing this process on all requirements properties of interest, we can derive a test suite that generates the UFC set of requirements. We illustrate this process in the case study in the following section.

4. CASE STUDY

To assess the feasibility of auto-generating tests from requirements on a moderately-sized example, we generated a set of requirements-based tests from the FGS case example described in Section 2. The test cases were generated in three ways. First, we simply negated the formal requirements (LTL properties) and provided them as trap properties to the model checker; the resulting test suite provides one test case for each requirement. Second, we generated trap properties for what we call *requirements antecedent coverage*. Requirements antecedent coverage ensures that in requirements of the form $G(A \rightarrow B)$ the antecedent becomes true at least once along a satisfying path so that the requirement is not vacuously satisfied (antecedent is false throughout). Third, we generated trap properties for the positive UFC (Unique First Cause) obligations discussed in Section 3.1 and used the model checker to generate UFC-adequate tests over the requirements. In this initial experiment we were interested in determining (1) the feasibility of generating such tests with a model checker, (2) the number

[2] W is the Weak Until operator, defined in LTL as $p\ \mathrm{W}\ q \equiv (p\ \mathrm{U}\ q) \vee G(p)$

of test cases needed to provide requirements UFC coverage for a substantial and realistic example, and (3) what coverage of the model these test sets would provide.

To provide realistic results, we conducted the case study using the requirements and model of the close to production model of a flight guidance system we introduced earlier in the paper. This example consists of 293 informal requirements formalized as LTL properties as well as a formal model captured in our research notation RSML^{-e} [30]. All the properties in the FGS system are safety properties, there are no liveness properties.

4.1 Setup

The case study followed 3 major steps:

Trap Property Generation: We started with the 293 requirements of the FGS expressed formally as LTL properties. We generated three sets of trap properties from these formal LTL properties.

First, we generated tests to provide *requirements coverage*; one test case per requirement illustrating one way in which this requirement is met. We obtained these test cases by simply negating each requirement captured as an LTL property and challenged the model checker to find a test case.

Second, we generated tests to provide *requirements antecedent coverage.* Consider the requirement

$$G(A \rightarrow B)$$

Informally, it is always the case that when A holds B will hold. A test case providing requirements antecedent coverage over such a requirement will ensure that the antecedent A becomes true at least once along the satisfying path. That is, the test case would satisfy the obligation

$$G(A \rightarrow B) \wedge F(A)$$

We implemented a transformation pass over the LTL specifications so that for requirements of the form $G(A \rightarrow B)$ we would generate trap properties requiring A to hold somewhere along the path.

Third, we generated tests to provide *requirements UFC coverage* over the syntax (or structure) of the required LTL properties. The rules for performing UFC over temporal operators were explained in Section 3.1. Using these rules, we implemented a transformation pass over the LTL specifications to generate trap properties for both the neutral and weakened UFC notions discussed in section 3.1 (we used the same implementation to generate tests for requirements coverage, and requirements antecedent coverage mentioned above). However, both neutral and weakened notions of UFC result in the same test suite for this case example, since the LTL property set for the FGS system has no 'future' and 'until' temporal operators. We only generated the positive UFC set over the temporal properties since each of the properties is known to hold of the model.

Although the properties were already known to hold of the model, generating tests is still a useful excercise. For a developer, it provides a rich source of requirements-derived tests that can be used to test the behavior of the object code. For the purposes of this paper, it provides a straightforward way to test the fault finding capability and completeness of our metrics.

Test suite Generation: To generate the test cases we leveraged a test case generation environment that was built in a previous project [15]. This environment uses the bounded model checker in NuSMV for test case generation. We automatically translate the FGS model to the input language of NuSMV, generate all needed trap properties, and transform the NuSMV counterexamples to input scripts for the NIMBUS RSML^{-e} simulator so that the tests can be run on the model under investigation. Previously, we had enhanced the NIMBUS framework with a tool to measure different kinds of coverage over RSML^{-e} models [14]. Therefore, we could run the test cases on the RSML^{-e} models and measure the resulting coverage.

Coverage Measurement: We measured coverage over the FGS model in RSML^{-e}. We ran all three test suites (requirements coverage, requirements antecedent coverage, and requirements UFC coverage) over the model and recorded the model coverage obtained by each. We measured State coverage, Transition coverage, Decision coverage, and MC/DC coverage.

- **State Coverage**: (Often referred to as variable domain coverage.) Requires that the test set has test cases that enable each control variable (Boolean or enumerated variable) defined in the model to take on all possible values in its domain at least once.
- **Transition Coverage**: Analogous to the notion of branch coverage in code and requires that the test set has test cases that exercise every transition definition in the model at least once.
- **Decision Coverage**: Each decision occurring in the model evaluates to true at some point in some test case and evaluates to false at some point in some other test case.
- **Modified Condition and Decision Coverage (MC/DC)**: Every condition within the decision has taken on all possible outcomes at least once and every condition has been shown to independently affect the decision's outcome.

4.2 Results and Analysis

We used our tools to automatically generate and run three test suites for the FGS; one suite for requirements coverage, one for requirements antecedent coverage and another one for requirements UFC coverage. The results from our experiment are summarized in Tables 8 and 9.

	Requirements coverage	Antecedent coverage	UFC coverage
Trap Properties	293	293	887
Test Cases	293	293	715
Time Expended	3 min	14 min	35 min

Table 8: Summary of the test case generation results.

Table 8 shows the number of test cases in each test suite and the time it took to generate them. It is evident from the table that the UFC coverage test suite is three times larger than the requirements coverage and requirements antecedent coverage test suites and can therefore be expected to provide better coverage of the model than the other two test suites. Also, the time expended in generating the UFC coverage test suite was significantly higher than the time necessary to generate the other two test suites.

We observe that our algorithm generating UFC over the syntax of the requirements generated 887 trap properties.

Nevertheless, only 715 of them generated counterexamples[3]. For the remaining 172 properties, the UFC trap property is *valid*, which means that the condition that it is designed to test *does not* uniquely affect the outcome of the property. In each of these cases the original formula was *vacuous* [3], that is, the atomic condition was not required to prove the original formula. We discuss the issue of vacuity checking further in Section 5.

Although it was possible to explain many of the vacuous conditions through implementation choices that satisfied stronger claims than the original properties required, the number of vacuous conditions was startling and pointed out several previously unknown weaknesses in our original property set. Rather than correcting the incorrectly vacuous formulas in our property set before proceeding with our experiment, we decided to use the property set "as-is" as a representative set of requirements that might be provided before a model is constructed. If test cases were manually constructed from this set of requirements, we postulate that many of these weaknesses would be found when trying to construct test cases for the vacuous conditions.

For all three coverage metrics mentioned, we did not minimize the size of the test suites generated. In previous work we found that test suite reduction while maintaining desired coverage can adversely affect fault finding [14]. However, the work in [14] was based on test suites generated using *model coverage criteria*. We plan to explore the effect of test suite reduction techniques on test suites generated using *requirements coverage criteria* in our future work.

Model Coverage Metric	Requirements coverage	Antecedent coverage	UFC coverage
State	37.19%	98.78%	99.12%
Transition	31.97%	89.53%	99.42%
Decision	46.42%	85.75%	83.02%
MC/DC	0.32%	23.87%	53.53%

Table 9: Summary of the model coverage obtained by running the requirements based tests.

In Table 9 we show the coverage of the formal model achieved when running the test cases providing requirements coverage, requirements antecedent coverage and requirements UFC coverage respectively. We measured four different model coverage criteria as mentioned in Section 4.1.

The results in Table 9 show that the test suite generated to provide requirements coverage (one test case per requirement) gives very low state, transition, and decision coverage, and almost no MC/DC coverage. This is in part due to the structure of the properties. Most of the requirements in the FGS system are of the form

$$G(a \rightarrow Xb)$$

The test cases found for such properties are generally those in which the model goes from the initial state to a state where a is false, thus trivially satisfying the requirement. Such test cases exercise a very small portion of the model and the resultant poor model coverage is not at all surprising.

The requirements antecedent coverage is a stronger metric than the requirements coverage measure. It gives high state, transition and decision coverage over the model. However, the MC/DC coverage generated over the model is low. As mentioned earlier, many of the requirements in the FGS system are of the form

$$G(a \rightarrow Xb)$$

Requirements antecedent coverage will ensure that the test cases found for such properties will exercise the antecedent a (i.e., make a true). Therefore, the requirement is not trivially satisfied and we get longer test cases and, thus, better model coverage than the requirements coverage test suite.

On the other hand, the test suite generated for UFC over the syntax of the properties provides high state, transition, and decision coverage. Nevertheless, the decision coverage provided by the UFC test suite is in this experiment lower than that provided by the requirements antecedent coverage test suite. When we looked more closely at both test suites we found that for variables specified in the property (we call these *variables of interest*), both test suites had the same values. In other words, for the variables of interest, the UFC test suite is a superset of the requirements antecedent coverage test suite. However, for variables not mentioned in the properties (we call these *free variables*), the model checker has the freedom to choose any suitable value for that variable. We found that the values for these free variables differed between the two test suites. We believe that this is the reason for the antecedent coverage test suite generating a higher decision coverage over the model than the UFC test suite; the model checking algorithm in NuSMV simply picked values for the free variables that happened to give the requirements antecedent test suite better coverage. If we could control the selection of the free variables the UFC test suite would yield the same or higher decision coverage than the requirements antecedent coverage test suite. Clearly, the test case generation method plays an important role in the types of test cases generated. We plan to explore the effect of different test case generation methods in our future work.

The UFC test suite generated low MC/DC coverage over the model, although not as low as the other two test suites. After some thought, it became clear that this is due in part to the structure of the requirements defined for the FGS. Consider the requirement

"When the FGS is in independent mode, it shall be active"

This was formalized as a property as follows:

$$\mathrm{G}(m\_Independent\_Mode\_Condition.result \rightarrow \mathrm{X}(Is\_This\_Side\_Active = 1))$$

Informally, it is always the case (G) that if the condition for being in independent mode is true, in the next state the FGS will always be active. Note here that the condition determining if the FGS is to be in independent mode is abstracted to a macro (Boolean function) returning a result (the .result on the left hand side of the implication). Many requirements for the FGS were of that general structure.

$$\mathrm{G}(Macro\_name.result \rightarrow \mathrm{X}\ b)$$

[3]We initially used the NuSMV bounded model checker with depth 5, that is, we only looked for test cases with a length of 5 steps or shorter. If the bounded model checker did not find such a test case we provided the trap property to the symbolic model checker in NuSMV and found that in all cases it verified the property as being true; that is, there was no test case for this particular UFC obligation.

Therefore, when we perform UFC over this property structure, we do not perform UFC over the—potentially very complex—condition making up the definition of the macro since this condition has been abstracted away in the property definition.

The macro *Independent_Mode_Condition* is defined in RSML^{-e} as:

```
MACRO Independent_Mode_Condition():
   TABLE
      Is_LAPPR_Active           : T *;
      Is_VAPPR_Active           : T *;
      Is_Offside_LAPPR_Active   : T *;
      Is_Offside_VAPPR_Active   : T *;
      Is_VGA_Active             : * T;
      Is_Offside_VGA_Active     : * T;
   END TABLE
END MACRO
```

Since the structure of *Independent_Mode_Condition* is not captured in the required property, the test cases generated to cover the property will not be required to exercise the structure of the macro and we will most likely only cover one of the MC/DC cases needed to adequately cover the macro.

Note that this problem is not related to the method we have presented in this paper; rather, the problem lies with the original definition of the properties. Properties should not be stated using internal variables, functions, or macros of the model under investigation; to avoid a level of circular reasoning (using concepts defined in the model to state properties of the model) the properties should be defined completely in terms of the input variables to the model. If a property must be stated using an internal variable (or function) then additional requirements (properties) are required to define the behavior of the internal variable in terms of inputs to the system. In this example, a collection of additional requirements defining the proper values of all macro definitions should be captured. These additional requirements would necessitate the generation of more test cases to achieve requirements UFC coverage and we would presumably get significantly better coverage of the model.

To get a better idea of how many additional test cases UFC coverage would necessitate when we add requirements defining macros, we considered the sample requirement property mentioned earlier:

$$\mathrm{G}(m\_Independent\_Mode\_Condition.result \rightarrow \mathrm{X}(Is\_This\_Side\_Active = 1))$$

We constructed a property defining the *Independent_Mode_Condition* macro. When we performed UFC over this additional macro defining requirement we got 13 additional test cases. This result shows that adding requirements that define all the macros would make a substantial difference to the test suite size and presumably the model coverage.

5. RELATED WORK

The work in this paper is closely related to work assessing the completeness and correctness of formulae in temporal logics. The most similar work involves *vacuity checking* of temporal logic formulas [3, 16, 21]. Intuitively, a model M *vacuously satisfies* property f if a subformula ϕ of f is not necessary to prove whether or not f is true. Formally, a formula is vacuous if we can replace ϕ by any arbitrary formula ψ in f without affecting the validity of f:

$$M \models f \quad \equiv \quad M \models f[\phi \leftarrow \psi]$$

Beer et al. [3] shows that it is possible to detect whether a formula contains vacuity by checking whether each of its atomic subformulas can be replaced by 'true' or 'false' without affecting the validity of the original formula (the choice depends on the structure of the formula and whether it is satisfied or not). To place it in our terms, this check determines whether each atomic condition independently affects the formula in question. This approach can be used to generate witness counterexamples for each atomic condition, similar to the trap properties for UFC that are described in Section 4. Nevertheless, the goal of this work is quite different than ours. The purpose of performing vacuity detection on a formula over a model is to see whether or not a valid formula can be replaced by a stronger valid formula. This stronger formula may indicate problems within either the formula or the model. Our work is concerned with adequately testing requirements, potentially in the absence of a model. The complete vacuity check defined in [3] is one possible metric for assessing the adequacy of a test set. It is simpler and more rigorous metric than our UFC metric when used for test generation. In future work, we plan to reformulate the metric in [3] to support "partial" weakening (as described in Section 3.1.3) and investigate its effectiveness.

Tan et al. [28] use the vacuity check presented in [3] to define a property coverage metric over LTL formulas. They present a model-checking assisted approach that generates a test suite based on the property coverage criteria that is finite in size and in length. Their main motivation is to enable the testing of linear temporal properties on the implementation by generating a black-box or white-box test suite that satisfies the defined property coverage metric. The goal of our work is to define a structural coverage metric based on requirements that allows us to measure the adequacy of black-box test suites. Also, Tan et al. [28] do not have any notion of test weakening and their notion of acceptable black box tests, while rigorous, is not very practical. The reason is that to test a finite prefix of a lasso shaped test, they repeat the loop part of the lasso n times where n is the number of system states. Many of the systems used in practice have very large n, thus making their methodology of generating acceptable black-box tests highly impractical. Also, they do not address the efficacy of their proposed metric.

In our case study we examined how well coverage of requirements mapped to coverage of an implementation model. This is similar to recent research assessing the completeness of LTL properties over models. In [8], Chockler et al. propose coverage metrics for formal verification based on metrics used in hardware simulation. Mutations, capturing the different metrics of coverage, are applied to a given design and the resultant mutant designs are examined with respect to a given specification. Two coverage checks were performed on the mutant design: *falsity coverage* (does the mutant still satisfy the specification?), and *vacuity coverage* (if the mutant design still satisfies the specification, does it satisfy it vacuously?). Symbolic algorithms to compute the different types of coverage are proposed. In [7, 6], Chockler et al. propose additional metrics to determine whether all parts of a model are covered by requirements. Our goal is to create a coverage metric that when provided a robust set of formal requirements and a test suite satisfying the metric will yield a high level of coverage of an implementation using standard coverage metrics. Chockler's work provides a more direct (and potentially accurate) assessment of the adequacy of the requirements. The process of building the set of mutant specifications and re-checking the properties is very expensive, however, and may not be feasible in practice.

6. DISCUSSION

In this paper, we explore the idea of requirements coverage and define three potential metrics that could be used to assess requirements coverage. To our knowledge, the notion of requirements coverage as a test adequacy criterion has not been previously addressed in any systematic way. There are several potential benefits of defining requirements adequacy criteria, including:

- a *direct measure* of how well a black-box test suite addresses a set of requirements,
- an *implementation independent* assessment of the adequacy of a suite of black-box tests,
- a means for measuring the *adequacy of requirements* on a given implementation,
- a formal framework that, given a model, allows for *autogeneration* of tests that are immediately traceable to requirements

Our hypothesis is that given a complete set of requirements and a rigorous testing metric, we should achieve a high level of coverage of an implementation of the requirements. If this hypothesis is true, then a requirements-based test suite can be used to help determine the completeness of a set of requirements with respect to an implementation, and a test suite which yields high requirements coverage and low model coverage illustrates one of three problems:

1. The model is incorrect. The model allows behaviors not specified by the requirements. Hence a test suite that provides a high level of requirements coverage will not cover these incorrect behaviors, thus resulting in poor model coverage.
2. There are missing requirements. Here, the model under investigation may be correct and more restrictive than the behavior defined in the original requirement; the original requirements are simply incomplete and allow behaviors that should not be there. Hence we need additional requirements to restrict these behaviors. These additional requirements necessitate the creation of more test cases to achieve requirements coverage and will, presumably, lead to better coverage of the model.
3. The criteria chosen for requirements coverage is too weak.

Given a robust and complete set of requirements, we still do not necessarily anticipate 100% MC/DC coverage of a model given 100% UFC coverage of the requirements. There are usually several designs that may satisfy a "good" set of requirements and these designs will introduce details that may not be covered by requirements-based tests.

This report only provides the start of the rigorous exploration of metrics for requirements-based testing; we have merely defined the notion and explored the feasibility of the approach. There are several topics that require further study:

Requirements formalization: Since formalizing high-level requirements is a rather new concept not generally practiced, there is little experience with how to best capture the informal requirements as formal properties. Finding a formalism and notation that is acceptable to practicing developers, requirements engineers, and domain experts is necessary. In our work we have used CTL and LTL, but we are convinced that there are notations better suited to the task at hand.

Requirements coverage criteria: To our knowledge, there has been little other work on defining coverage criteria for high-level software requirements. Therefore, we do not know what coverage criteria will be useful in practice. We must find coverage criteria that (1) help us assess test suites as to their effectiveness in finding problems in implementations derived from the requirements and (2) do not require test suites of unreasonable size.

Requirements versus model coverage: We must explore the relationship between requirements-based structural coverage and model or code-based structural coverage. Given a "good" set of requirements properties and a test suite that provides a high level of structural coverage of the requirements, is it possible to achieve a high level of structural coverage of the formal model and of the generated code? That is, does structural coverage at the requirements level translate into structural coverage at the code level?

Test case generation method: We plan to evaluate the effect that different test case generation methods have on the fault finding capability and model coverage achieved by test suites that provide a high level of structural coverage of the requirements. We plan to investigate a variety of model checkers and their strategies in this regard.

In sum, we believe that the notion of coverage metrics for requirements-based testing holds promise and we look forward to exploring each of these topics in future investigations.

7. REFERENCES

[1] P. E. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of the Fourth IEEE International Symposium on High-Assurance Systems Engineering.* IEEE Computer Society, Nov. 1999.

[2] R. Armoni, D. Bustan, O. Kupferman, and M. Y. Vardi. Aborts vs. resets in linear temporal logic. In *TACAS*, pages 65–80, November 2003.

[3] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *Formal Methods in System Design*, pages 141–162, 2001.

[4] B. Bezier. *Software Testing Techniques, 2nd Edition.* Van Nostrand Reinhold, New York, 1990.

[5] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[6] H. Chockler, O. Kupferman, R. P. Kurshan, and M. Y. Vardi. A practical approach to coverage in model checking. In *Proceedings of the International Conference on Computer Aided Verification (CAV01), Lecture Notes in Computer Science 2102*, pages 66–78. Springer-Verlag, July 2001.

[7] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for temporal logic model checking. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 2031*, pages 528–542. Springer-Verlag, April 2001.

[8] H. Chockler, O. Kupferman, and M. Y. Vardi. Coverage metrics for formal verification. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods, volume 2860 of Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, October 2003.

[9] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[10] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of Computer Aided Verification (CAV)*, pages 27–39, 2003.

[11] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.

[12] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[13] K. Hayhurst, D. Veerhusen, and L. Rierson. A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA, 2001.

[14] M. P. Heimdahl and G. Devaraj. Test-suite reduction for model based tests: Effects on test quality and implications for testing. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE)*, Linz, Austria, September 2004.

[15] M. P. Heimdahl, S. Rayadurgam, and W. Visser. Specification centered testing. In *Second International Workshop on Analysis, Testing and Verification*, May 2001.

[16] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. *Journal on Software Tools for Technology Transfer*, 4(2), February 2003.

[17] Z. Manna and A. Pnueli. Temporal verification of reactive systems: Safety. Technical report, Springer-Verlag, New York, 1995.

[18] S. Miller, A. Tribble, T. Carlson, and E. J. Danielson. Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA, June 2003.

[19] S. P. Miller, M. P. Heimdahl, and A. Tribble. Proving the shalls. In *Proceedings of FM 2003: the 12th International FME Symposium*, September 2003.

[20] A. J. Offutt, Y. Xiong, and S. Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.

[21] M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Proceedings of the 14th Conference on Computer Aided Design*, pages 485–499. Springer-Verlag, 2002.

[22] S. Rayadurgam. *Automatic Test-case Generation from Formal Models of Software*. PhD thesis, University of Minnesota, November 2003.

[23] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.

[24] S. Rayadurgam and M. P. Heimdahl. Generating MC/DC adequate test sequences through model checking. In *Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW-03*, Greenbelt, Maryland, December 2003.

[25] Reactive Systems Inc. Reactis product description. http://www.reactive-systems.com/index.msp.

[26] D. J. Richardson, S. L. Aha, and T. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. Springer, May 1992.

[27] RTCA. *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.

[28] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. In *IEEE Int. Conf. on Information Reuse and Integration (IEEE IRI-2004)*, November 2004.

[29] E. Technologies. Scade suite product description. http://www.esterel-technologies.com/v2/ scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.

[30] M. W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.