Coverage Metrics for Requirements-Based Testing: Evaluation of Effectiveness[*]

Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl
University of Minnesota
[staats,whalen,heimdahl]@cs.umn.edu

Ajitha Rajan
Laboratoire d'Informatique de Grenoble
arajan@cs.umn.edu

Abstract

In black-box testing, the tester creates a set of tests to exercise a system under test without regard to the internal structure of the system. Generally, no objective metric is used to measure the adequacy of black-box tests. In recent work, we have proposed three *requirements coverage metrics*, allowing testers to objectively measure the adequacy of a black-box test suite with respect to a set of requirements formalized as Linear Temporal Logic (LTL) properties. In this report, we evaluate the effectiveness of these coverage metrics with respect to fault finding. Specifically, we conduct an empirical study to investigate two questions: (1) *do test suites satisfying a requirements coverage metric provide better fault finding than randomly generated test suites of approximately the same size?*, and (2) *do test suites satisfying a more rigorous requirements coverage metric provide better fault finding than test suites satisfying a less rigorous requirements coverage metric?*

Our results indicate (1) only one coverage metric proposed—Unique First Cause (UFC) coverage—is sufficiently rigorous to ensure test suites satisfying the metric outperform randomly generated test suites of similar size and (2) that test suites satisfying more rigorous coverage metrics provide better fault finding than test suites satisfying less rigorous coverage metrics.

1 Introduction

When validating a system under test (SUT), the creation of a black box test suite—a set of tests that exercise the behavior of the model without regard to the internal structure of the model under test—is often desirable. Generally, no objective standard is used for determining the adequacy of test suite with respect to a set of requirements. Instead, the adequacy of such suites is inferred by examining different coverage metrics on the executable artifact defining the SUT, such as source code. However, given formalized software requirements it is possible to define meaningful coverage metrics *directly on the structure of the requirements*. Of interest here is previous work in which we adopted structural code coverage metrics to define three increasingly rigorous requirements coverage metrics over Linear Temporal Logic (LTL) requirements: *requirements coverage*, *antecedent coverage*, and *Unique First Cause (UFC) coverage* [19]. The relationship between the criteria forms a *linear subsumption hierarchy*, as test suites satisfying more rigorous coverage metrics are guaranteed to satisfy less rigorous coverage metrics.

In this work we empirically evaluate the fault finding ability of test suites satisfying these requirements coverage metrics. Specifically, we investigate if (1) in general a test suite satisfying a requirements coverage metric provides greater fault finding than a randomly generated test suite of the same size and (2) the linear subsumption hierarchy between metrics reflects the relative fault finding of test suites satisfying these requirements coverage metrics. Our study is conducted using four commercial examples drawn from the civil avionics domain. For each case example, we generate 600 mutants, a test suite

[*]This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

satisfying each requirements coverage metric, and 1,000 random tests. We resample these sets to create 25 sets of 200 mutants and 25 reduced test suites for each coverage metric. For each reduced test suite, we create a random test suite of approximately the same size by resampling from the set of 1,000 random tests. We explore fault finding using each set of mutants and each test suite. In total, we examine the fault finding of 15,000 combinations of case examples, test suites, and mutant sets.

From our results we draw the following observations. First, the relative rigor of the requirements coverage metrics reflects the fault finding ability of test suites satisfying them. Second, the structure of requirements significantly impacts the fault finding of test suites satisfying the coverage metrics, an observation previously made in relation to other coverage metrics [16]. Finally, test suites satisfying the UFC coverage criterion often provide greater fault finding than randomly generated test suites of equal size, while test suites satisfying the other coverage metrics do not. This indicates that of the three coverage metrics evaluated, only UFC coverage is a consistent measure of the adequacy of test suites—the intuition behind the other metrics may be useful, but they are not rigorous enough to be used as an objective measure of test suite adequacy. Our results highlight how this lack of rigor creates the potential for test suites to "cheat" a coverage criterion by *technically* satisfying the criterion, but doing so in an uninteresting or non-representative manner. This is particularly a problem when using automatic test generation tools, as these tools generally have no concept of realistic tests and tend to generate the simplest possible tests satisfying a coverage criterion.

2 Related Work

In [19], we formally defined the requirements coverage metrics studied in this paper and outlined our rationale for each. As detailed in [19], these metrics are related to work by Beer et al., Kupferman et al., and Chockler et al. on *vacuity checking* and *coverage metrics* for temporal logics [2, 4, 10]. More recently, we examined the fault finding effectiveness of test suites satisfying UFC coverage of the requirements on a system and MC/DC (Modified Condition/Decision Coverage) over the system itself [16]. We determined that test suites satisfying MC/DC generally outperform test suites satisfying UFC coverage (though fault finding was often close). This study, however, did not investigate the fault finding effectiveness of requirements and antecedent coverage as compared to UFC.

Fraser and Gargantini [5] and Fraser and Wotawa [6] explored automatic test generation using NuSMV for a number of coverage criteria, including UFC and mutation coverage. Neither of these works compare the effectiveness of test suites satisfying UFC coverage to randomly generated tests suites of equal size or perform any analysis that considering the size of the test suite.

3 Requirements Coverage Metrics

Previously, we defined three requirements coverage metrics: *requirements coverage*, *antecedent coverage*, and *Unique First Cause (UFC)* coverage [19]. Each requirements coverage metric is defined over formalized requirements expressed as Linear Temporal Logic (LTL) properties [15]. These coverage metrics provide different levels of rigor for measuring test adequacy and form a linear subsumption hierarchy, with requirements coverage as the weakest and UFC coverage as the strongest.

Requirements Coverage: To satisfy *requirements coverage*, a test suite must contain at least one test per requirement that—when executed—causes the requirement to be met. Consider the following natural language requirement: "*The onside Flight Director cues shall be displayed when the Auto-Pilot is engaged.*" A test derived from this requirement might examine the following scenario: (1) Engage the Auto-Pilot, and (2) Verify that the Onside Flight Director comes on. Alternatively, the test might simply leave the Auto-Pilot turned off. Technically, this test meets the requirement, albeit in a way that is not particularly illuminating.

Antecedent Coverage: Requirements, such as the previous example, are often implications, formally $G(A \rightarrow B)$. A on the left hand side of $\rightarrow$ is referred to as the *antecedent*, and B on the right hand side as the *consequent*. To satisfy antecedent coverage, a test suite must contain at least one test case per requirement that, when executed, causes the requirement to be met and causes the antecedent to evaluate to *true*.

Unique First Cause (UFC) Coverage: Requirements are often more complex than the simple example given previously. For such complex requirements (and often simple ones), it is desirable to have a rigorous coverage metric that requires tests to demonstrate the effect of each atomic condition in the requirement; this ensures that every atomic condition is necessary and can affect the outcome of the property. Requirements and antecedent coverage cannot do this. Therefore, we defined a coverage metric called *Unique First Cause (UFC)* coverage over LTL requirements. It is adapted from the MC/DC criterion [3, 9], a structural coverage metric designed to demonstrate the independent effect of basic Boolean conditions (i.e., subexpressions with no logical operators) on each Boolean decision (expression) within source code.

A test suite satisfies UFC coverage over a set of LTL requirements if executing the test cases in the test suite will guarantee that (1) every basic condition in each formula has taken on all possible outcomes at least once and (2) each basic condition in each formula has been shown to independently affect the formula's outcome.

4 Study

Based on the subsumption hierarchy between these metrics outlined in the previous section we expect that test suites satisfying UFC coverage will provide better fault finding than test suites satisfying the other requirements coverage metrics, and we expect that test suites satisfying antecedent coverage will provide better fault finding than test suites satisfying requirements coverage. Furthermore, we expect that a test suite satisfying a coverage metric will provide better fault finding than a randomly generated test suite of equal size since a test suite satisfying a requirements coverage metric is designed to systematically exercise the system.

We conducted an experiment to evaluate the following hypotheses:

Hypothesis 1 (H_1): *A test suite satisfying a requirements coverage metric provides greater fault finding than a randomly generated test suite of approximately equal size.*
Hypothesis 2 (H_2): *A test suite satisfying a requirements coverage metric provides greater fault finding than a test suite satisfying a less rigorous requirements coverage metric.*

We used four industrial systems in our experiment. For each case example, we performed the following steps:

1. **Generated a test suite for each coverage metric:** We generated three suites providing UFC, antecedent, and requirements coverage. (Section 4.2.)
2. **Generated random tests:** We generated random inputs for 1,000 random tests. (Section 4.3.)
3. **Generated mutants:** We generated 600 single-fault mutants (Section 4.4.)
4. **Ran test suites with mutants and case example:** We ran each mutant and the original case example using the three generated test suites and the random test suite.
5. **Generated reduced requirements coverage test suites:** Each requirements coverage test suite was naïvely generated and thus highly redundant. We generated 25 reduced test suites that maintain the original coverage from each full test suite. (Section 4.2.)
6. **Generated random test suites:** For each reduced requirements coverage test suite, we randomly generated a random test suite approximately the same size as the reduced requirements coverage test suite by sampling the 1,000 random tests previously generated. (Section 4.3.)

7. **Generated mutants sets:** We randomly generated 25 sets of 200 mutants by resampling from the 600 mutants. (Section 4.4.)
8. **Assessed fault finding ability of each reduced test suite:** For each test suite and each mutant set, we determined how many mutants were detected by the test suite.

In the remainder of this section, we describe in detail our experimental approach.

4.1 Case Examples

In our experiment, we use four industrial avionics applications from displays and flight guidance systems. All four systems were modeled using the Simulink notation from Mathworks Inc. [12] and were translated to the Lustre synchronous programming language [8] to take advantage of existing automation [13]. For more information on these systems, see [16].

4.2 Requirement Test Suite Generation and Reduction

We have used an automated approach to generate test cases from a model of the system behavior. This model represents the knowledge of the desired system behavior a domain expert might possess. Using this technique, we can generate the large number of tests required for our experiments. Furthermore, the approach tends to generate the shortest, simplest tests that satisfy obligations, and, thus, the tests generated are unlikely to be unusually effective (and may in fact be unusually poor), a phenomenon previous observed in [16].

Several research efforts have developed automatic test generation techniques based on formal models and model checkers [7, 18]. The technique we use is outlined in [18] and operates by generating NuSMV counterexamples through trap properties. Using this technique, we can create test suites achieving the maximum achievable coverage for a specified coverage metric.

A test suite generated using this approach will be highly redundant, as a single test case will often satisfy several obligations. Such a test suite is not representative of the coverage metric, as it may contain far more tests than are needed and may therefore bias the test suite's fault finding ability. We therefore reduce each test suite using a greedy algorithm. First, we randomly select a test case from the generated test suite, determine which obligations are satisfied by the test and add it to a reduced test suite. We continue by randomly choosing another test case from the generated test suite, determining if any obligations not satisfied by the reduced test suite are satisfied by the test case selected, and if so adding the test case to the reduced test suite. This process yields a randomly reduced test suite achieving the same requirements coverage as the the original test suite. We generated 25 reduced test suites for each coverage metric and each case example to avoid threats to validity due to a small sample size.

4.3 Random Test Suite Generation and Reduction

We generated a single set of 1,000 random tests for each case example. Each individual test contains between 2-10 steps with the number of tests of each test length distributed evenly in each set of tests. For each reduced requirements coverage test suite we create a random test suite with the same number of steps (or slightly more) by randomly sampling from the set of 1,000 random tests generated. As a result, each set of reduced requirements coverage test suites satisfying a coverage metric has a corresponding set of random test suites of approximately the same size. The number of steps was used as a measurement of size rather than the number of tests as this avoids creating a random test suite with significantly longer or shorter tests on average than the corresponding reduced requirements coverage test suite.

4.4 Mutant Generation

We created 600 *mutants* (faulty implementations) for each case example by introducing a single fault into the correct implementation. Each fault was introduced by either inserting a new operator into the system or by replacing an operator or variable with a different operator or variable. The faults seeded include

	RC	RC-RAN	AC	AC-RAN	UFC	UFC-RAN
DWM_1	77.8%	78.7%	78.6%	78.7%	88.0%	81.6%
DWM_2	0.0%	1.60%	0.0%	1.29%	5.41%	9.42%
Latctl_Batch	17.1%	49.8%	59.0%	62.1%	85.2%	82.7%
Vertmax_Batch	22.4%	26.6%	46.9%	42.5%	81.5%	72.2%

	AC:RC Imp	UFC:RC Imp	UFC:AC Imp	RC:RC-RAN Imp	AC:AC-RAN Imp	UFC:UFC-RAN Imp
DWM_1	1.01%	13.1%	12.0%	-1.0%	-0.1%	7.8%
DWM_2	0.0%	∞	∞	-100%	-100%	-42.0%
Latctl_Batch	244.1%	396.9%	44.4%	-65.0%	-5.1%	3.0%
Vertmax_Batch	108.9%	262.4%	73.5%	-15.0%	10.3%	12.9%

Table 1: Average and Relative Improvement in Fault Finding
Column Header ($X : Y$) Denotes Relative Fault Finding Imp. Using Test Suite X over Test Suite Y
RC = Requirements Coverage, AC = Antecedent Coverage
X-RAN = Random Test Suite w/ Size $\approx$ Size of a Reduced Test Suite Satisfying X Coverage

	UFC $\leq$ AC	UFC $\leq$ RC	AC $\leq$ RC	UFC $\leq$ UFC-RAN	AC $\leq$ AC-RAN	RC $\leq$ RC-RAN
DWM_1	<0.001	<0.001	<0.001	<0.001	0.62	1.0
DWM_2	<0.001	<0.001	1.0	1.0	1.0	1.0
Latctl_Batch	<0.001	<0.001	<0.001	<0.001	1.0	1.0
Vertmax_Batch	<0.001	<0.001	<0.001	<0.001	1.0	1.0

Table 2: Statistical Analysis for $H0_1$ and $H0_2$
RC = Requirements Coverage, AC = Antecedent Coverage
X-RAN = Random Test Suite w/ Size $\approx$ Size of a Reduced Test Suite Satisfying X Coverage

arithmetic, relational, boolean, negation, delay introduction, constant replacement, variable replacement and parameter replacement and are described in more detail in [16].

We generated 600 mutants for each case example. From these 600 mutants, we generated 25 sets of 200 mutants. We generated multiple sets of mutants to avoid threats to validity due to a small sample size. Note that we did not check that generated mutants are semantically different from the original implementation. This weakness in mutant generation does not affect our results since we are interested in the relative fault finding ability between test suites.

5 Results

To determine the fault finding of a test suite T and a mutant set M for a case example we simply compare the output values produced by the original case example against every mutant $m \in M$ using every test case $t \in T$. The fault finding effectiveness of the test suite for the mutant set is computed as the number of mutants killed divided by the number of mutants in the set. We perform this analysis for each test suite and mutant set for every case example yielding 7,500 fault finding measurements per case example. We use the information produced by this analysis to test our hypotheses and infer the relationship between coverage criteria and fault finding.

The complete analysis is too large to include in this report[1]. For each case example, we present in Table 1 the average and relative improvement in fault finding for each coverage metric and for the random test suites corresponding to each coverage metric. We limit the discussion in this section to statistical analysis, a discussion of these results and their implications follows in Section 6.

5.1 Statistical Analysis

To evaluate our hypotheses (from Section 4), we first evaluate each hypothesis for each combination of a case example and requirements coverage metric (for H_1) or pairing of requirements coverage metrics (for H_2). Using these results, we determine what conclusions can be generalized across all systems.

[1]Our data can be retrieved at `http://crisys.cs.umn.edu/public_datasets.html` and is available to the research community for additional analysis.

UFC $>$ AC	**UFC $>$ RC**	**AC $>$ RC**	**UFC $>$ UFC-RAN**	**AC $>$ AC-RAN**	**RC $>$ RC-RAN**
Supported	Supported	Unsupported	Unsupported	Unsupported	Unsupported

Table 3: Conclusions for Hypotheses Across All 4 Case Examples
RC = Requirements Coverage, AC = Antecedent Coverage
X-RAN = Random Test Suite w/ Size $\approx$ Size of a Reduced Test Suite Satisfying X Coverage
Each column ($X > Y$) Denotes Hypothesis of "Test Suites Satisfying X Have Greater Fault Finding Than Test Suites Satisfying Y"

Each of the individual hypotheses states that one set of fault finding measurements should be in general higher than another set of fault finding measurements. For example, we hypothesize that fault finding measurements for reduced test suites satisfying UFC coverage are better than fault finding measurements for the set of comparably sized random test suites (for each oracle and case example). To evaluate our hypotheses, we use a bootstrapped permutation test, a non-parametric statistical test that determines the probability that two sets of data belong to the same population [11], and explore 1,000,000 permutations when calculating each p-value. From a practical standpoint, we are only interested in scenarios where test suites satisfying a requirements coverage metric outperform random test suites and where test suites satisfying a more rigorous coverage metric outperform a less rigorous coverage metric. Consequently, we formulate our null hypotheses as *one-tailed* tests, restating H_1 and H_2 as the null hypotheses $H0_1$ and $H0_2$, respectively:

H0*$_1$*: *For case example CE, the data points for percentage of mutants caught by test suites satisfying coverage metric C are less than or equal to the data points for percentage of mutants caught by random test suites of approximately the same size.*

H0*$_2$*: *For case example CE, the data points for percentage of mutants caught by test suites satisfying coverage metric C_1 are less than or equal to the data points for percentage of mutants caught by test suites satisfying a less rigorous coverage metric C_2.*

We evaluate $H0_1$ and $H0_2$ using 4 case examples and 3 coverage metrics or 3 pairings of coverage metrics, respectively. We therefore produce 12 p-values each for $H0_1$ and $H0_2$, listed in Table 2. These p-values are used to generalize across our results.

We use the Bonferonni correction, a conservative method for guarding against erroneously rejecting the null hypothesis when generalizing results. The Bonferonni correction sets the alpha required for each null hypothesis at $1/n$ times the alpha desired for the entire set, where n is the number of results generalized; in this case, the alpha is set at $1/4 * 0.05 = 0.0125$ for each hypothesis. We present the results in Table 3.

Given these results, we can evaluate our original hypotheses:

Hypothesis 1 (H_1): *A test suite satisfying a requirements coverage metric provides greater fault finding than a randomly generated test suite of approximately equal size.*
Hypothesis 2 (H_2): *A test suite satisfying a requirements coverage metric provides greater fault finding than a test suite satisfying a less rigorous requirements coverage metric.*

The majority of test suites satisfying a coverage metric provide worse fault finding than random test suites of similar size, and thus H_1 is not statistically supported. Furthermore, test suites satisfying antecedent and requirements coverage provide similar levels of fault finding for the *DWM_2* case example, and thus H_2 is not statistically supported.

However, we note here several interesting results from Tables 2 and 3, and explore them in Section 6. First, H_2 fails only when test suites satisfying antecedent and requirements coverage find no faults. Second, test suites satisfying UFC coverage outperform random test suites 75% of the time (3 of 4 case examples). Finally, random test suites always outperform test suites satisfying requirements and antecedent coverage *with statistical significance* (derivation of this significance is not shown), indicating that the random tests are not simply equivalent to the tests generated to satisfy a coverage metric, but preferable. These results and our recommendations are explored in Section 6.

5.2 Threats to Validity

External Validity: We only used four synchronous reactive critical systems in the study. We believe, however, that these systems are representative of the critical systems in which we are interested and that our results can therefore be generalized to other related systems.

As our implementation language we used the synchronous programming language Lustre [8] rather than a more common language such as C or C++. Systems written in Lustre are similar in style to traditional imperative code produced by code generators. Therefore, testing Lustre code is sufficiently similar to testing reactive systems written in C or C++ to generalize the results to such systems.

We have used automatically seeded faults to evaluate the fault finding ability of tests suites. It is possible the faults seeded are not representative. Nevertheless, previous work indicates fault seeding methods similar to our own are representative of real faults encountered in software development [1].

We only report results using test oracles based on output variables. In pilot studies, we observed that test oracles considering internal variables in addition to output variables produced similar results.

Finally, we used automatic test generation rather than a domain expert when creating test suites. Clearly, the tests produced by an automated tool differ from tests likely to be produced by a domain expert, particularly in the case of requirements coverage. Nevertheless, we are interested in evaluating the *coverage metrics*, not domain experts, and thus view the worst-case test generation provided by the tools to be preferable—these tools tend to highlight deficiencies in coverage metrics, and thus our evaluation is not influenced by the skill of a domain expert.

Conclusion Validity: For each case example, we have performed resampling using a random test suite containing 1,000 tests and a set of 600 mutants. These values are chosen to yield a reasonable cost for the study. It is possible that sampling from larger random test suites may yield different results, or that the number of mutants is too low. Based on past experience, however, we have found results using 200 mutants to be representative [17, 16], and thus believe 600 mutants to sufficient.

For each case example, we have generated 25 reduced test suites for each coverage metric, a corresponding set of 25 random test suites for each coverage metric, and 25 sets of 200 mutants. These numbers were chosen to keep the cost of resampling reasonable. It is possible that the fault finding measurements that result do not accurately represent the set of possible results. Nevertheless, the low variance in fault finding measurements observed in the sets of test suites and mutants—coupled with the consistency of our results across case examples—indicates our conclusions are accurate.

6 Discussion

In Section 5, we showed that neither hypothesis was statistically supported. Nevertheless, the results lead to several worthwhile observations. First, despite the lack of statistical significance, the subsumption hierarchy between coverage metrics generally reflects the relative fault finding of test suites satisfying a coverage metric; the more rigorous the coverage metric, the more effective the corresponding test suite will be. This indicates a useful tradeoff in test suite rigor and test suite size exists.

This observation, however, is rendered largely moot by the effectiveness of random testing. For both requirements and antecedent coverage, random test suites provide greater fault finding than test suites satisfying these coverage metrics, often with statistical significance. Furthermore, for the *DWM_2* system, test suites satisfying UFC coverage do not outperform randomly generated test suites. The former result highlights problems with using automatic test generation tools to generate tests satisfying a coverage criterion. The latter result highlights how the structure of requirements can affect the usefulness of a coverage metric.

In the remainder of this section, we will discuss the implications of these results and recommendations concerning requirements coverage metrics.

6.1 Pitfalls of Automatic Test Generation

As mentioned in Section 4.2, automatic test generation tends to produce tests that satisfy coverage obligations using minimal effort. Tests generated tend to frequently use the default inputs (e.g., all signals set to false), and may in that way technically satisfy obligations while not exercising useful or representative scenarios. Consequently, a test suite generated to meet a coverage criterion might do mostly nothing, while a random test suite of equal size will generally do something (albeit something arbitrary). This accounts for the relatively good fault finding achieved by random test suites.

The degree to which this is a problem depends on the complexity of the coverage obligations. For example, the coverage obligations produced using requirements coverage for the *DWM_2* system are very simple, and consequently every test generated is exactly the same; therefore, one test satisfies every obligation. Conversely, the coverage obligations produced using UFC coverage for the *Vertmax_Batch* system are quite complex, requiring over 10 times as many tests as needed to satisfy requirements or antecedent coverage for the same system. When using complex obligations, automatic test generation must generate tests meeting a wide variety of constraints, including specific sequences of events, and consequently the test cases produced tend to be more realistic and more effective.

6.2 Pitfalls of Requirements Structure

For the *DWM_2* system, the randomly generated test suites outperform the test suites satisfying UFC coverage. We have observed similar issues in previous work [16] where we noted the requirements for this particular system are structured such that the UFC criterion is rendered ineffective as compared to our other case examples[2].

```
 LTLSPEC G(var_a > (
    case
        foo : 0 ;
        bar : 1 ;
    esac +
    case
        baz : 2 ;
        bpr : 3 ;
    esac
));
```

Figure 1: Original LTL Requirement

```
LTLSPEC G(var_a > (
    case
        foo & baz : 0 + 2 ;
        foo & bpr : 0 + 3 ;
        bar & baz : 1 + 2 ;
        bar & bpr : 1 + 3 ;
    esac
));
```

Figure 2: Restructured LTL Requirement

We note that many of the requirements for the *DWM_2* system were of the form (formalized as SMV [14]) in Figure 1. Although this idiom may seem unfamiliar, this use of case expressions is not uncommon when specifying properties in NuSMV. Informally, the sample requirement states that *var_a* is always greater than the sum of the outcomes of the two case expressions. Generating UFC obligations for the above requirement yields very simple test obligations since there are no complex decisions – we simply need to find tests where the top-level relational expression and each atomic condition has taken on the values true and false.

This requirement can be restructured without changing the meaning as shown in Figure 2. Achieving UFC coverage over this restructured requirement will require more obligations than before since the boolean conditions in the case expression are more complex and we must demonstrate independence of each condition in the more complex decisions. Thus, the structure of the requirements has a significant impact on the number and complexity of UFC obligations required. For this experiment, we did not restructure any requirements. Consequently, the UFC obligations for the *DWM_2* do not have as complex

[2]This discussion is partially adopted from [16] as we encountered the same phenomenon in that investigation.

of a structure as those for the other three systems, and automatic test generation (as with requirements and antecedent coverage) produces poor tests.

6.3 Recommendations

The goal of using a coverage criterion is to infer the efficacy of a test suite. By verifying that a test suite satisfies a coverage criterion, one hopes to provide evidence that the test suite is “good” and is likely to uncover faults. While no objective standard for “good” fault finding exists, at a minimum a test suite satisfying a useful coverage criterion should provide better fault finding than a randomly generated test suite of similar size; otherwise, little confidence in the quality of a test suite is gained by showing it satisfies the coverage criterion.

The test suites generated by automatic test generation to satisfy requirements and antecedent coverage are in some sense “worst-case” test cases, and it is likely a domain expert using these metrics as a guide would produce far better tests. We are, however, not evaluating our coverage metrics as guidelines for developing tests, but as objective measure of black-box test suite adequacy, and our results clearly demonstrate that it is easy to satisfy requirements or antecedent coverage using inadequate test suites. We therefore conclude that *requirements and antecedent coverage are not sufficiently rigorous* to ensure a test suite satisfying one of these metrics is better than a random test suite of equal size—the intuition behind the metrics may be useful, but merely satisfying the metric does not provide convincing evidence.

Conversely, the test suites generated to provide UFC coverage, despite also being “worst-case” test cases, generally outperform random test suites of approximately the same size. The caveat to this is requirements structure—by breaking up complex requirements into several less complex requirements, the benefits of UFC coverage can be diminished. Nevertheless, our results indicate that the adequacy of black-box tests can be inferred using UFC coverage. We therefore conclude that UFC coverage is sufficiently rigorous to ensure a test suite is better than a random test suite of equal size provided the requirements are not structured to mask complexity. Admittedly, this is a rather weak conclusion and more research is needed to identify coverage criteria that are more effective and more robust with respect to the structure of the requirements.

Thus, in practice, testers who wish to objectively infer the adequacy of a black-box test suite with respect to a set of requirements formalized as LTL properties should use UFC coverage while noting the pitfalls outlined above.

7 Conclusion

In [19], we defined coverage metrics over the structure of requirements formalized as Linear Temporal Logic (LTL) properties. Such metrics are desirable because they provide objective, model-independent measures of the adequacy of black-box testing activities. In this paper, we empirically demonstrated that of the three coverage metrics explored, only the UFC coverage criterion is rigorous enough to be a useful measurement of test suite adequacy. We also noted that the usefulness of the UFC coverage criterion is influenced by the structure of the requirements. Consequently, we conclude that testers who wish to measure the adequacy of a test suite with respect to a set of LTL requirements should use a coverage criterion at least as strong as UFC coverage, but should be mindful of the structure of their requirements.

Furthermore, our work highlights the difference between using coverage metrics as guidelines for developing test suites, and using coverage metrics as objective measurements of test suite adequacy. We demonstrate how using an insufficiently rigorous coverage criterion for inferring test suite adequacy can lead to incorrectly concluding a test suite that technically meets the criterion is adequate, when the test suite is no better than a randomly generated test suite of approximately the same size. We believe this has implications in domains such as avionics and critical systems, where test coverage metrics are used by regulatory agencies to infer the efficacy of a testing process and thus the quality of a software system. We hope to investigate this problem in depth in future work.

References

[1] J.H. Andrews, L.C. Briand, and Y. Labiche. Is Mutation an Appropriate Tool for Testing Experiments? *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 402–411, 2005.

[2] 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.

[3] J. J. Chilenski and S. P. Miller. Applicability of Modified Condition/Decision Coverage to Software Testing. *Software Engineering Journal*, pages 193–200, September 1994.

[4] 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.

[5] G. Fraser, A. Gargantini, and V. Marconi. An evaluation of model checkers for specification based test case generation. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 41–50. IEEE Computer Society Washington, DC, USA, 2009.

[6] G. Fraser and F. Wotawa. Complementary criteria for testing temporal logic properties. In *Proceedings of the 3rd International Conference on Tests and Proofs*, page 73. Springer, 2009.

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

[8] 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.

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

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

[11] P.H. Kvam and B. Vidakovic. *Nonparametric Statistics with Applications to Science and Engineering*. Wiley-Interscience, 2007.

[12] Mathworks Inc. Simulink product web site. http://www.mathworks.com/products/simulink.

[13] S. Miller, E. Anderson, L. Wagner, M. Whalen, and M. Heimdahl. Formal verification of flight critical software. In *Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit*, August 2005.

[14] The NuSMV Toolset, 2005. Available at
http://nusmv.irst.itc.it/.

[15] A. Pnueli. Applications of temporal logic to specification and verification of reactive systems: A survey of current trends. *Lecture Notes in Computer Science Number 224*, pages 510–584, 1986.

[16] A. Rajan, M. Whalen, M. Staats, and M.P. Heimdahl. Requirements coverage as an adequacy measure for conformance testing. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, pages 86–104. Springer, 2008.

[17] A. Rajan, M.W. Whalen, and M.P.E. Heimdahl. The effect of program and model structure on MC/DC test adequacy coverage. In *Proc. of the 30th Int'l Conference on Software engineering*, pages 161–170. ACM New York, NY, USA, 2008.

[18] Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. of the 8th IEEE Int'l. Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, April 2001.

[19] M.W Whalen, A. Rajan, and M.P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of International Symposium on Software Testing and Analysis*, pages 25–36, July 2006.