Are We There Yet? Determining the Adequacy of Formalized Requirements and Test Suites *

Anitha Murugesan[1], Michael W. Whalen[1], Neha Rungta[2], Oksana Tkachuk[2], Suzette Person[3], Mats P.E. Heimdahl[1], Dongjiang You[1]

[1] Department of Computer Science and Engineering,
University of Minnesota, 200 Union Street, Minneapolis, MN 55455,USA
{anitha,whalen,heimdahl,djyou}@cs.umn.edu
[2] NASA Ames Research Center
{neha.s.rungta,oksana.tkachuk}@nasa.gov
[3] NASA Langley Research Center
suzette.person@nasa.gov

Abstract. Structural coverage metrics have traditionally categorized code as either covered or uncovered. Recent work presents a stronger notion of coverage, *checked coverage*, which counts only statements whose execution contributes to an outcome checked by an oracle. While this notion of coverage addresses the adequacy of the oracle, for Model-Based Development of safety critical systems, it is still not enough; we are also interested in how much of the oracle is covered, and whether the values of program variables are masked when the oracle is evaluated. Such information can help system engineers identify missing requirements as well as missing test cases. In this work, we combine results from checked coverage with results from requirements coverage to help provide insight to engineers as to whether the requirements or the test suite need to be improved. We implement a dynamic backward slicing technique and evaluate it on several systems developed in Simulink. The results of our preliminary study show that even for systems with comprehensive test suites and good sets of requirements, our approach can identify cases where more tests or more requirements are needed to improve coverage numbers.

1 Introduction

Model-Based Development (MBD) refers to the use of domain-specific modeling notations to create models of a desired system early in the development lifecycle. These models can be executed on the desktop, analyzed for desired behaviors, and then used to automatically generate code and test cases. Also known as correct-by-construction development, the emphasis in model-based development is on the engineering effort invested in the early lifecycle activities of modeling, simulation, and analysis. This reduces development costs by finding defects early in the lifecycle, avoiding rework that is necessary when errors are discovered during integration testing, and by automating the late life-cycle activities of coding and test case generation. In this way, Model-Based Development significantly reduces costs while also improving quality. There are

* This work has been partially supported by NSF grants CNS-0931931 and CNS-1035715.

several commercial MBD tools, including Simulink/Stateflow [19], SCADE [10], IBM Rhapsody [1] and IBM Rational Statemate [2].

An important part of MBD is automated test generation and execution. Tools such as Reactis [26], the MathWorks Verification and Validation plug-in for Simulink, and the IBM Rhapsody Automatic Test Generation add-on, as well as other tools, support automated test generation from models. These tools enable generation of structural coverage tests up to a high degree of rigor, e.g., tests satisfying the MC/DC coverage metric. In the domain of critical systems – particularly in avionics – demonstrating structural coverage is required for certification [27].

In principle, automated test generation represents a success for software engineering research: a mandatory -- and potentially arduous – engineering task has been automated. However, several studies have raised questions about the effectiveness of automated test generation towards a specific structural coverage metric (e.g., [12, 14, 31]), in some cases finding these tests less effective than randomly generated tests of the same length in terms of fault-finding capabilities. This often has to do with the *observability* capabilities of the test oracle, which determines whether the test passes or fails. In many cases, the code structure that was examined has no measurable effect on the test outcome.

In recent work, a metric proposed by Schuler and Zeller in [29, 30] addresses observability, but does so in a post-priori way: given a test suite and a set of requirements specified as assertions, it uses dynamic backward slicing from the requirements (assertions) to determine the set of program statements that affect the evaluation of the requirement. They call this metric *checked statement coverage*, because it only considers the statements that are checked (observed). They note that this metric judges the quality of the *test oracle* — a program with no assertions will have no coverage. Therefore, given any test suite, it is possible to increase coverage by adding additional oracles (requirements) to the suite. Our hypothesis is that this metric can be leveraged to better assess the quality of an automated testing process in MBD where formalized requirements serve as oracles for auto-generated tests [28].

In this work, we combine the results of checked coverage with the results of requirements coverage to determine for a given model whether its requirements and test suite are adequate. While the work in [30] focuses on whether or not the *oracles* (requirements) are adequate, we are interested in both the adequacy of the test suite and the requirements encoded as oracles: if checked coverage is low then either the requirements or the tests maybe incomplete. Specifically, we add to this notion of coverage by calculating checked coverage based on dynamic backward slicing *as well as* MC/DC masking information. Finally, we map the different forms of code coverage back to the model, and report the coverage of the requirements, in order to provide information to the system engineers about sources of incompleteness. Thus, the contributions of the paper are:

- An approach using checked, unchecked, and requirements coverage information to assess the adequacy of both test suites and requirements.
- An approach to calculate checked coverage based on backward dynamic slicing and MC/DC masking information, which leads to more precise checked coverage results than dynamic backward slicing alone.

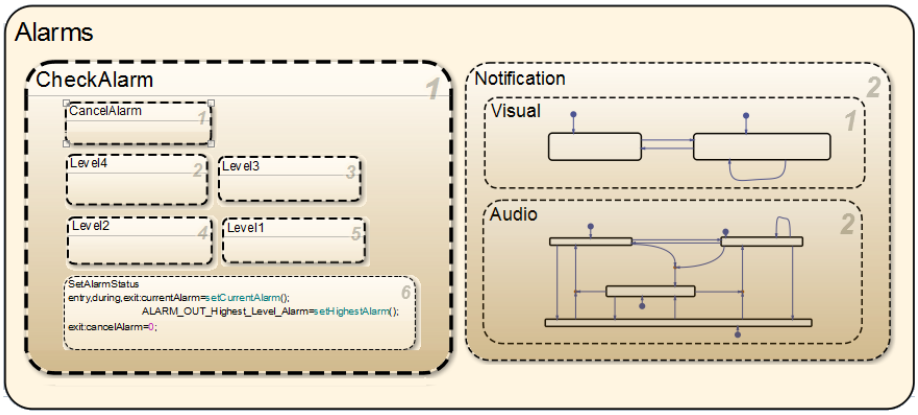

Fig. 1: Hierarchical state machine model of the ALARM subsystem.

– A preliminary evaluation of our technique on a set of examples that use Simulink as part of the MBD approach. In addition to computing coverage for the auto-generated code, we also map the results back to the models.

Our experience shows that even for case studies with comprehensive test suites and good sets of requirements, our approach can identify cases where more tests or more requirements are needed to improve the coverage numbers.

2 Motivation

Consider the control software for an infusion pump, a medical device that is typically used to infuse liquid drugs into a patient's body in a controlled fashion. An important subsystem of the controller is the ALARM subsystem shown in Fig. 1. The model for the system [22] was developed using MathWorks Simulink/Stateflow tool [19]. The "ALARM" subsystem is responsible for monitoring hazards (CheckAlarm state machine) with different levels of severity in the system, and alerting the clinicians (Audio and Visual state machines) to take the appropriate action when such conditions occur. We auto-generate the source code from the Simulink model, formalize the requirements as boolean expressions, and automatically generate the test cases from the model.

To motivate the utility of our proposed approach we use a snippet of auto-generated code from the Audio state machine in Fig. 1. The code is shown in Fig. 2. It raises an aural alert when a certain level of hazard is detected and the audio has not been disabled by the user. Assume the following oracle encodes a requirement of the system:

$$Hazard >= 3 \wedge Disable\_Audio = 0 \implies Audio\_Command = 1$$

Suppose we execute a test case, t, that covers program statements one to seven in Fig. 2 and the values of the variables used in the oracle are: $Hazard := 3$ and

```
1: if(localB->ALARM_OUT_Hazard >= 3){
2:   if(localB->Disable_Audio > 1){
3:     localB->ALARM_OUT_Audio_Command = 0;
4:     localB->ALARM_OUT_Audio_Disabled = 1;
5:     if(localDW->time_minutes > 3){
6:       localB->Disable_Audio = 0;
7:     }
8:   }
9: }else ...
```

Fig. 2: Code snippet from the ALARM system's audio notification functionality.

$Disable\_Audio := 2$. The corresponding checked coverage for the test does not contain the program statement at line 4 in Fig. 2; the $Audio\_Disabled$ variable defined at line 4 does not either directly or transitively impact the values used in the oracle. This example demonstrates that the *checked* coverage is lower than the set of *covered* statements.

The notion of *checked* coverage, however, does not take into account which parts of the oracle were covered and whether the values of certain program variables are masked when the oracle is evaluated. The values for variables $Hazard := 3$ and $Disable\_Audio := 2$ cause the antecedent in the requirement ($Hazard >= 3 \wedge Disable\_Audio = 0$) to be false; hence, the consequent of the requirement ($Audio\_Command = 1$) is not evaluated. Even though the program statement at line 3 in Fig. 2 writes to the variable $Audio\_Command$ used in the oracle, the test, t, does not evaluate $Audio\_Command$ in the oracle. We can leverage this information to define a more precise checked coverage measure by marking line 3 in Fig. 2 as unchecked. In the next section we present an overview of how we measure requirements coverage along with checked coverage to improve upon the checked coverage measure.

3 Methodology

There are three inputs to our technique: the model of the system being analyzed, a set of test cases (manual or auto-generated) that exercise the model, and a set of formalized requirements of the model as shown in Fig. 3. The requirements are transformed into assertions over program variables. We automatically generate the code from the model and execute the tests on the auto-generated code. The formalized requirements are used as a slicing criteria for program execution traces generated by the various tests as shown in Fig. 3. A dynamic backward slice is used to extract the set of program statements that operate on variables whose values are *checked* in the assertions. This is termed as checked coverage while all other executed statements are categorized as unchecked coverage. In addition to the code coverage we also measure the coverage of the requirements. Checked, unchecked, and uncovered code coverage are mapped back to the model to help the system engineers determine incompleteness in the requirements, tests, or the model.

We present an overview of the algorithm to partition coverage into *checked* coverage versus *unchecked* coverage in Fig. 4. The algorithm takes as input an auto-generated

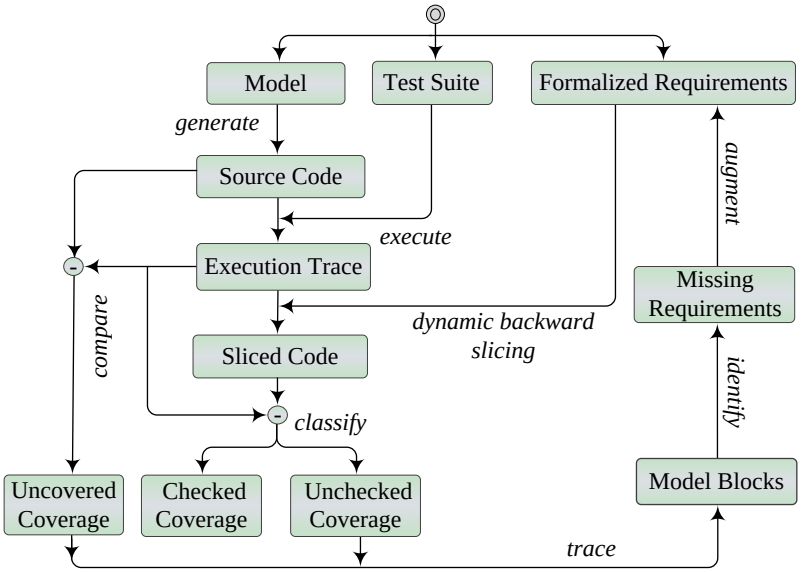

Fig. 3: Test Case Coverage Classification Approach Overview

program M, the test suite T for exercising the behaviors of the program, and the set of assertions that encode the formalized requirements. The sets $checked$ and $unchecked$ are initialized as empty. We run each test, t, in the test suite T on the program and generate the set of program statements $\langle l_0, \ldots, l_n \rangle$ executed by the test. Next, we generate a dynamic slice of the trace using each assertion a as the slicing criteria. In the case that a program statement l is in the dynamic slice then it is added to the $checked$ set; otherwise it is added to the $unchecked$ set.

Dynamic slicing is used to compute the basic form of checked coverage. A dynamic slice of an execution trace with respect to an assertion extracts the set of program statements in the trace that *may* impact the evaluation of the assertion. Standard flow analyses are used to generate the slice based on the assertion. Any program statements that read or write variables used in the assertion, as well as program statements computed by transitive closure of the reads and writes, are part of the dynamic slice. Suppose, boolean variables x and y are used in the assertion; all program statements that read and write program variables that *may* be used directly or transitively by x and y are added to the dynamic slice. This notion of *checked* coverage does not however take into account which parts of the assertion are covered and whether certain values are masked when the assertion is evaluated. In the rest of the section we first present how we measure the coverage of the assertions and then leverage the information to improve the precision of the checked coverage.

```
/ * checked := ∅, unchecked := ∅ * /
procedure initialize(M, T, A)
 1: for each t ∈ T ∧ a ∈ A do
 2:     ⟨l_0, ..., l_n⟩ := execute(P, t)
 3:     for each i ∈ [0, n] do
 4:         if l_i ∈ dynamicBackwardSlice(⟨l_0, ..., l_n⟩, a) then
 5:             checked := checked ∪ {l_i}
 6:         else
 7:             unchecked := unchecked ∪ {l_i}
 8:     unchecked := unchecked \ checked
```

Fig. 4: An algorithm to partition checked and unchecked coverage.

3.1 Coverage of Requirements

In this work we use the Modified Decision/Condition Coverage (MC/DC) metric to evaluate the assertion coverage for a given test suite. MC/DC is commonly used to evaluate the coverage of requirements in safety-critical systems. MC/DC coverage of a requirement encoded as an assertion requires that each condition in the assertion takes on all possible outcomes at least once and each condition is shown to *independently* affect the assertion's outcome. Note that a condition is a boolean expression that contains no boolean operators. We use the masking form of MC/DC to determine the independence of the conditions in the assertion. A condition is masked if changing its value does not affect the outcome of the assertion. For example, when evaluating `assert` x `and` y, in the case when x is `false`, the value of y is masked. We need to satisfy three possible coverage obligations:

1. $x \wedge y$
2. $x \wedge \neg y$
3. $\neg x \wedge y$

In order to check the MC/DC coverage of the assertion x `and` y, we replace the assertion in A with three new assertions synthesized from the expressions shown above. If there are test cases in T that can satisfy all three assertions, then we report 100% MC/DC coverage of the assertion. But if only one is satisfied by the test, then we report 33% coverage of the assertion. We believe that measuring the MC/DC coverage of the requirements for a given test suite enables us to better characterize the quality of the test suite with respect to a given set of requirements.

3.2 A More Precise Dynamic Backward Slice

We propose a more precise dynamic backward slice that takes into account which parts of the assertion are covered and whether certain values of program variables are not used when the assertion is evaluated. We leverage the *masking* information within an assertion for a given test to generate a more precise dynamic backward slice. As stated earlier, a condition is *masked* if changing its value cannot affect the outcome of a decision. So in the assertion, x `and` y, if the value for x is `false`, the value of y is masked.

In this more precise version of a dynamic slice we first extract the variables in the assertion that are not masked, then get all of the program statements in the execution trace that impact them. Therefore, instead of computing the slice based on both x and y, we generate a slice using x alone. Even though there are values of y being written to in the execution trace, since they are not being used in the evaluation of the assertion, they are not added to the $checked$ set. We believe this will reduce the size of the $checked$ set and provide a more precise characterization of parts of the program that are being checked in the assertions.

3.3 Mapping Back to the Model

In the final phase of our technique, for a given test suite, we report the following to the system engineers: (i) the precise checked coverage, (ii) the unchecked coverage, (iii) the uncovered coverage, and the (iv) coverage of the requirements. Note that we map the coverage of the code onto the model. We believe that these coverage measures help us bridge the gap between requirements, tests, and the model as discussed in [28]. The relationship between the various types of coverage can potentially help to determine the source of incompleteness in either tests, requirements, or the model. Low coverage of the requirements and high checked coverage could indicate missing functionality in the model. Low coverage of the requirements coupled with low checked coverage could be indicative of missing tests and/or missing requirements. Finally, high coverage of requirements along with low checked coverage could be indicative of missing requirements.

4 Evaluation

In this section we describe the evaluation of our approach on three systems. We first give a brief overview of the example systems, then we describe the experimental set up followed by the evaluation of the approach on the systems.

4.1 Case Examples

We consider three different systems: a medical device controller, an avionics system controller and a general appliance controller. Table 1 shows the specifics of the case examples considered. Following this section, we refer to each system and its test cases using the ID from the first column in Table 1. The second column gives the number of auto-generated source lines of code (LOC); column three presents the number of requirements available for each test suite; column 4 describes the source of the test suites. The last column shows the number of tests in each test suite.

The first system considered is the ALARM subsystem discussed in Section 2. The model of the ALARM subsystem was developed as a multi-level hierarchical state machine using the Mathworks Simulink/Stateflow tool. The source code of this model was automatically generated using MathWorks Simulink Coder [20]. The system has 18 formally verified [22] safety critical requirements. For testing the ALARM system, we

ID	System	# LOC	# Reqs	Test Suite : Source	# Tests
ALM_1	ALARM	1950	18	Set 1 : Manual	16
ALM_2	ALARM	1950	18	Set 2 : jKind	106
DCK_1	DOCKING	2240	3	Set 1 : Reactis	32
DCK_2	DOCKING	2240	3	Set 2 : SDV	69
MCR_1	MICROWAVE	537	11	Set 1 : Reactis	39
MCR_2	MICROWAVE	537	11	Set 2 : Reactis	23

Table 1: Case Example Artifacts Synopsis

ID	Statement	Condition	Requirements
ALM_1	43.65%	31.93%	65.71%
ALM_2	95.05%	95.80%	84.84%
DCK_1	39.43%	35.29%	26.66%
DCK_2	77.37%	78.89%	73.32%
MCR_1	79.07%	93.75%	60.86%
MCR_2	87.21%	100.00%	80.42%

Table 2: Case Example's Test Case Coverage Metrics

created manual test cases using the requirements as a reference and also generated a test suite with high structural coverage (MC/DC) using the jKind model checker [13].

The second example we consider is a docking approach system. This system specifies the mechanism for the docking of a space vehicle. This system was also developed using Mathworks Simulink/Stateflow tool and its source code was generated using Simulink Coder. A major issue with this system is that even though it is elaborately modeled, there are only a few requirements specified. Although we know that this system lacks a complete set of requirements, our goal was to analyze the adequacy of the sparse requirements for the test cases. For the Docking example, we generated a random test suite using the Reactis tool and another test suite with high structural coverage using MathWorks Simulink Design Verifier (SDV) [21] .

The third case example is a microwave's controller system used in our previous work [28], that was also modeled as hierarchical state machines using the MathWorks Stateflow notation. The microwave controller implements the usual functions of a regular microwave. We generated code for the microwave system using the Gryphon Tool Suite [34]. The advantage with the microwave model is that it has a comprehensive set of requirements. The test cases for microwave were generated using Reactis.

4.2 Tools and Experiment Set up

We use a combination of commercially available and free open source tools to implement our approach. As previously mentioned, the test suites and the source code are generated using various sources and tools in order to generate a variety of artifacts and determine the efficacy of the different test suites based on our metrics. However, assess-

ing the test suite generation techniques and tools is not the intent of this experiment. We used the gcov [17] tool to measure the statement and condition coverage of the test suites. In order to measure coverage of requirements we generate MC/DC obligations and replace the assertions with these obligations. The total number of obligations that are satisfied by the test suite are recorded and reported.

To generate dynamic backward slices, we use the Frama-C tool [7], an open source tool for analysis of C programs. Although Frama-C is primarily a static analysis tool, it provides the ability to construct dynamic backward slices by embedding the test vector into the program and using the `-slevel` slicing option. The Frama-C slicing plugin provides an implementation of dependence-based backward slicing. The Frama-C slicing plugin requires the slicing criterion to be expressed using ACSL [4], a formal specification language used for specifying behavioral properties of C source code. The ACSL notation allows C like syntax for specifying slicing criteria, which makes it straightforward to specify requirements as logical statements. For example, the slicing criteria for the ALARM's oracle described in Section 2 is translated into an expression for slicing as shown below:

```
//@slice pragma expr
 (!(Hazard >= 3 and Disable_Audio == 1) || (Audio_Command == 0));
```

The slice is obtained by executing each test case in the test suite and extracting the dynamic backward slice based on the slicing criterion (requirements). While executing the test, the execution trace is also obtained. Once all slices and execution traces are obtained, the slices are compared with the execution trace to identify the checked and unchecked covered lines of code. Similarly by comparing the source code and the execution trace, the uncovered lines of code are obtained.

4.3 Analysis of the Results

Table 2 shows the structural and requirements coverage metrics for the artifacts for a given test suite. The statement and condition coverage for ALM_1 and DCK_1 and the requirements coverage for DCK_1 is less than 50%. The rest of the coverage numbers are over 50%. The statement and condition coverage of ALM_2 is slightly above 95% and the requirements coverage is 84%. Similarly MCR_2 has statement and condition coverage of 87% and 100% respectively and requirements coverage of 80%. These are fairly reasonable values for traditional coverage metrics for this set of artifacts.

Table 3 shows the results obtained using the dynamic slicing based approaches. The first two columns show the checked and unchecked coverage values using the dynamic backward slicing technique as proposed by [29, 30], whereas the next two columns show the checked and unchecked coverage values using the more precise dynamic backward slicing approach presented in this paper. The results demonstrate that, overall, the checked coverage in Table 3 is lower compared to the set of covered statements shown in Table 2. Recall that the total number of checked statements plus the unchecked statements gives the covered statements. Table 3 shows that the unchecked coverage ranges from 2.37% for MCR_1 to 41.47% for DCK_2. Using the more precise dynamic slicing technique proposed in this work the checked coverage decreases even further while

	Slicing		Precise Slicing		
ID	**Checked**	**Unchecked**	**Checked**	**Unchecked**	**Uncovered**
ALM_1	36.50%	8.65%	20.01%	23.64%	56.35%
ALM_2	75.44%	19.61%	54.35%	38.70 %	4.95%
DCK_1	23.91%	15.52%	5.49%	33.96%	60.57%
DCK_2	35.63%	41.47%	16.06%	61.31%	22.63%
MCR_1	76.70%	2.37%	56.25%	22.82%	20.93%
MCR_2	73.86 %	13.35%	65.34%	21.87%	12.79%

Table 3: Coverage Metrics Partitioned based on Slicing

ID	Covered	Requirements	Checked	Improve, Add
ALM_1	43.65%	65.71%	20.01%	test cases, new reqs
ALM_2	95.05%	84.84%	54.35%	new reqs
DCK_1	39.43%	26.66%	5.49%	all
DCK_2	77.37%	73.32%	16.06%	new reqs
MCR_1	79.07%	60.86%	56.25%	test cases
MCR_2	87.21%	80.42%	65.34%	new reqs

Table 4: Data Summary

the unchecked coverage increases. The MCR_2 artifact has a reasonably high statement coverage of 87.21% as shown in Table 3. coverage In the MCR_1 example, the checked coverage using the slicing approach decreases from 76.70% in column one to 56.25% in column three of Table 3 when using precise slicing, because the tests are not able to exercise most variables in the requirements. The low requirements coverage of 60% as shown in Table 2 provides evidence for the same. In MCR_2, however, when more variables of the requirements are exercised by the test cases (indicated by requirements coverage of 80.42%) the decrease in the checked coverage is smaller—73.86% to 65.34%.

The results for the examples in this section provide evidence towards our hypothesis that taking into account the part of the requirements or oracle that are covered (not masked) by the tests can provide us with a stronger notion of structural coverage with respect to the requirements.

5 Discussion

We summarize the results of the empirical evaluation and provide some recommendations for improvement based on the data. Table 4 presents the three coverage metrics (i) covered, (ii) requirements, and (iii) checked, as well as the recommendations for which artifacts should be further augmented in order to improve the coverage of the code and the requirements. For example, ALM_1 has reasonable requirements coverage of 65.71% but fairly low covered program statements (43.65%) and even lower precise checked coverage (20.01%). Our recommendation is to first augment the test suite with

tests that exercise additional parts of the code, then try to identify missing requirements, and finally measure the requirements coverage with the augmented test cases. DCK_1 has fairly low coverage values for all metrics, suggesting that all artifacts need to be improved. This is not surprising since there are only three requirements for the model. The ALM_2, DCK_2, MCR_2 examples have reasonable statement and requirements coverage but low precise checked coverage. This suggests that the set of requirements may be incomplete. MCR_1 also has reasonable statement coverage but the coverage of existing requirements needs to be improved prior to identifying the missing requirements.

```
switch (ALARM_Functional_DW.is_IsIdleTimeExceeded)
.....
 case ALARM_Functional_IN_No:
   else if (ALARM_Functional_B.Current_System_Mode == 1)
    ALARM_Functional_DW.idletimer = 0;
    ALARM_Functional_DW.idletimer++;
 ...
```

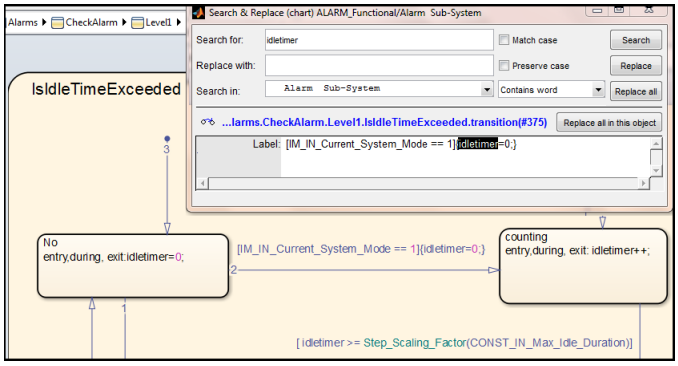

Fig. 5: Tracing unchecked lines of source code in the ALARM model

We demonstrate using an example of how the coverage information can be used by system engineers to detect potential causes of missing requirements. The ALARM system had 19.6% unchecked coverage (see Table 3). A snippet of code from the unchecked lines of code is shown in Figure 5. The variables used in these lines are then traced back to their source blocks in the model, as shown in Figure 5. Using this information, a system engineer might want to add a requirement that would check if the system has been IDLE for more than a certain amount of time.

This overall approach can be iteratively applied until we achieve the desired coverage metrics. Although achieving 100% for all the coverage criteria is ideal, it may not be practical. However, we believe that the metrics presented in the paper help identify the specific inadequacies in the test suite, that can be analyzed by the stakeholders to determine if and how they should be addressed. In future work, we would like to assess the fault finding capability improvement by improving these artifacts.

6 Related Work

Our work is built on the checked-coverage work of Schuler and Zeller [29, 30], which is in turn built upon dynamic slicing techniques [15] which follow from Weiser's original slicing work [32]. Checked coverage is in the category of *observability testing*, in which a metric tries to ensure that the code structure under test can be observed by the oracle. Often, the oracle is simply the outputs of the system under test. Observability testing has been a focus in testing of hardware logic circuits. The observability-based code coverage metric (OCCOM) attaches tags to internal states in a circuit and the propagation of tags is used to predict the actual propagation of errors (corrupted state) [9, 11]. A variable is tagged when there is a possible change in the value of the variable due to a fault. The observability coverage can be used to determine whether erroneous effects that are activated by the inputs can be observed at the outputs.

For software, dynamic taint analysis, or dynamic information flow analysis, marks and tracks data in a program at runtime in order to determine observability. This technique has been used in security as well as software testing and debugging [6, 18]. Taint propagation occurs in both explicit information flow (i.e., data dependencies) and implicit information flow (control dependencies). Although the way in which markings are combined varies based on the application, the default behavior is to union them [6]. Thus, dynamic taint analysis is conservative and does not consider masking. More accurate techniques for information flow modeling, such as [35], define path conditions to prove *non-interference*, that is, the non-observability of a variable or expression on a particular output. These information flow-based techniques have been used for testing in a metric called *Observable MC/DC* [33]; this work is very similar to checked coverage except that markings are *forward propagated* from observation points towards an oracle rather than (in checked coverage) *back-propagated* from the oracle towards observation points.

Mutation testing [3, 8, 23] is also concerned with quality of both *tests* and *oracles*. In mutation testing, one creates a set of programs that contain some small modification (*mutation*) of the original program and determines whether the discrepancy is detected (*killed*) by the test suite / oracle pair. Mutation testing suffers somewhat from the problem of *equivalent mutants*, which are program modifications that do not change the observable behavior of the program.

For requirements testing, much of the work has focused on requirements specified in temporal logic. In [24, 36], a coverage metric called *Unique First Cause Coverage* is defined by expanding the MC/DC test metric to formulas involving temporal logic operators. Similar work involves *vacuity checking* of temporal logic formulas [5, 16, 25]. Intuitively, a model M *vacuously satisfies* property f if a sub-formula 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:

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

For requirements specified as *synchronous observers*, the Simulink test generation tool Reactis and the Mathworks Verification and Validation plug-in for Simulink support MC/DC generation and coverage measurement over requirements.

7 Conclusion

There are a variety of mechanisms to generate test cases. The two main techniques for test case generation are (i) manual and (ii) automated test case generation techniques. In MBD, system engineers often write tests manually in order to cover the requirements as well as cover program statements. The system engineers study the requirements and try to determine the constraints on program inputs and their expected outputs on the model based on the statements in the requirements. This information is used to then create test inputs and a test oracle, using various techniques. Some operate on formalized requirements, some on the model, while others on the code auto-generated from the model. We can measure the structural coverage of the code when these tests are executed.

The challenge for automatically generated tests is that there is no oracle. Sometimes even in manually generated tests, defining a precise oracle for a given test is often a difficult endeavor. When present, system requirements that are either formalized or *can* be formalized serve as ideal candidates to be encoded as oracles. Even if the requirements are in a natural language such as English but describe the requirements in terms of the interface of the model, then we can convert these requirements into some formal notation.

Recent work presents a stronger notion of coverage of *checked* coverage, compared to traditional structural values of simply covered and uncovered [29, 30]. It uses dynamic backward slicing to count only statements whose execution contributes to an outcome checked by an oracle. In this work we add precision to the notion of checked coverage based on combining MC/DC masking information with dynamic backward slicing. We believe that this information can help system engineers identify missing requirements as well as missing test cases. The approach presented here allows us to connect the dots between test cases, requirements, and the model.

We demonstrated our approach using three case examples and also illustrated how the metrics can be actually used as a closed loop in identifying missing requirements and improving testing in a model-based approach. As part of future work, we would like to evaluate the proposed approach on the requirements and tests of the NASA's Lunar Atmosphere and Dust Environment Explorer (LADEE) mission.

8 Acknowledgments

This work was performed as part of an internship at NASA Ames Research Center funded by the Aviation Safety Program. We would like to thank Gregory Gay at University of Minnesota, for helping us measure requirements coverage of test cases.

References

1. IBM Rational Rhapsody. http://www.ibm.com/developerworks/rational/products/rhapsody/, 2014.
2. IBM Rational Statemate. http://www-03.ibm.com/software/products/en/ratistat, 2014.
3. Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, Washington, DC, USA, 2014. IEEE Computer Society.

4. Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto, and Inria Saclay Île-de france. ACSL: ANSI/ISO C specification language. 2008.
5. Ilan Beer, Shoham Ben-David, Cindy Eisner, and Yoav Rodeh. Efficient detection of vacuity in ACTL formulas. In *Formal Methods in System Design*, pages 141–162, 2001.
6. James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 Int'l Symposium on Software Testing and Analysis*, pages 196–206, 2007.
7. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.
8. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
9. Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. An observability-based code coverage metric for functional simulation. In *Proceedings of the 1996 IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 418–425, 1996.
10. Esterel-Technologies. SCADE Suite product description. http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html, 2004.
11. Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. OCCOM-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.
12. Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.
13. Andrew Gacek. JKind - a Java implementation of the KIND model checker. https://github.com/agacek.
14. Gregory Gay, Matt Staats, Michael W. Whalen, and Mats P. E. Heimdahl. Moving the goalposts: Coverage satisfaction is not enough. In *Proceedings of the 7th International Workshop on Search-Based Software Testing*, New York, NY, USA, 2014. ACM.
15. B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.
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. GNUGPL License. Gcov: Gnu coverage tool, https:gcc.gnu.org.
18. W. Masri, A. Podgurski, and D. Leon. Detecting and debugging insecure information flows. In *Proceedings of the 15th Int'l Symposium on Software Reliability Engineering*, pages 198–209, 2004.
19. MathWorks Inc. Simulink. http://www.mathworks.com/products/simulink.
20. MathWorks Inc. Simulink Coder. http://www.mathworks.com/products/simulink-coder/.
21. MathWorks Inc. Simulink Design Verifier. http://www.mathworks.com/products/sldesignverifier.
22. Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam, and Mats P.E. Heimdahl. Compositional verification of a medical device system. In *ACM Int'l Conf. on High Integrity Language Technology (HILT) 2013*. ACM, November 2013.
23. A. Jefferson Offutt and Ronald H. Untch. Mutation testing for the new century. chapter Mutation 2000: Uniting the Orthogonal, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.
24. Charles Pecheur, Franco Raimondi, and Guillaume Brat. A formal analysis of requirements-based testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 47–56. ACM, 2009.

25. M. Purandare and F. Somenzi. Vacuum cleaning CTL formulae. In *Proceedings of the 14th Conf. on Computer Aided Design*, pages 485–499. Springer-Verlag, 2002.
26. Reactive systems inc. http://www.reactive-systems.com/index.msp.
27. RTCA/DO-178C. Software considerations in airborne systems and equipment certification.
28. Neha Rungta, Oksana Tkachuk, Suzette Person, Jason Biatek, Michael W. Whalen, Joseph Castle, and Karen Gundy-Burlet. Helping system engineers bridge the peaks. In *Proceedings of the 4th International Workshop on Twin Peaks of Requirements and Architecture*, TwinPeaks 2014, pages 9–13, New York, NY, USA, 2014. ACM.
29. David Schuler and Andreas Zeller. Assessing oracle quality with checked coverage. In *Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, ICST '11, pages 90–99, Washington, DC, USA, 2011. IEEE Computer Society.
30. David Schuler and Andreas Zeller. Checked coverage: an indicator for oracle quality. *Software: Testing, Verification and Reliability*, 23(7):531–551, November 2013.
31. Matt Staats, Gregory Gay, Michael W Whalen, and Mats P.E. Heimdahl. On the danger of coverage directed test case generation. In *15th Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, April 2012.
32. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.
33. Michael Whalen, Gregory Gay, Dongjiang You, Mats PE Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *Proceedings of the 2013 Int'l Conf. on Software Engineering*. ACM, May 2013.
34. Michael W. Whalen, Darren D. Cofer, Steven P. Miller, Bruce H. Krogh, and Walter Storm. Integration of formal analysis into a model-based software development process. In Stefan Leue and Pedro Merino, editors, *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2007.
35. Michael W. Whalen, David A. Greve, and Lucas G. Wagner. *Model Checking Information Flow*. Springer-Verlag, Berlin Germany, March 2010.
36. Michael W. Whalen, Ajitha Rajan, and Mats P.E. Heimdahl. Coverage metrics for requirements-based testing. In *Proceedings of Int'l Symposium on Software Testing and Analysis*, pages 25–36. ACM, July 2006.