State Coverage: A Structural Test Adequacy Criterion for Behavior Checking

Ken Koster
kenk@agitar.com

David C. Kao
dvk@agitar.com

Agitar Software Laboratories
450 National Avenue
Mountain View, California 94043

ABSTRACT

We propose a new language-independent, structural test adequacy criterion called state coverage. State coverage measures whether unit-level tests check the outputs and side effects of a program.

State coverage differs in several respects from existing test adequacy criteria, such as code coverage and mutation adequacy. Unlike other coverage-based criteria, state coverage measures the extent of checks of program behavior. And unlike existing fault-based criteria such as mutation adequacy, state coverage has been designed to be readily automated and to present users with easily understood test inadequacy reports.

An experiment showed strong positive correlations between the number of behavior checks and both state coverage and mutation adequacy.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging–*testing tools*; D.2.8 [Software Engineering]:Metrics

General Terms: Measurement, Reliability

Keywords: Coverage, fault-based, mutation testing, state coverage, structural testing, test adequacy criteria, unit testing

1. INTRODUCTION

How thoroughly tested is a program? This is the main question that test adequacy criteria attempt to answer. Each criterion answers this question differently according to how it measures program "size" and how much of that size is evaluated by a test. For instance, branch coverage measures a program by the number of branches it contains and informs users which of those branches were not taken during test execution. *State coverage*, the test adequacy criterion we propose in this paper, evaluates tests according to behavior checking; it measures a program by the number of statements that define output and side effect variables and judges tests according to whether they check those variables.

The example JUnit[1] test below, `testAbsWithoutCheck` does not check program behavior automatically. Human intervention is required to examine the output of the test to verify the correctness of the behavior of the program. The implementation of `abs` could be replaced by any compiling code that terminates normally without causing the test to fail. Adding the assertion statement `assertEquals(4, x)` is one possible way to have `testAbsWithoutCheck` check the return value – and thus the behavior – of `abs`.

```
int abs(int n) {
  if (n < 0)
    return n * -1;
  else
    return n;
}

public void testAbsWithoutCheck() {
  int x = abs(-4);
  System.out.println("abs(-4):" + x);
}
```

State coverage detects such missing checks of behavior, reporting which program statements set unchecked outputs or side effects. (In the `abs` example, the `return` statements of `abs` require checks.) By reporting test inadequacies in terms of statements, statement coverage reports have been designed to be as easy to understand and as intuitive a metric as statement/branch coverage.

2. EXISTING ADEQUACY CRITERIA

Most of the various test adequacy criteria studied by the research community are impractical. Recent surveys have found that few development projects – on the order of 25% – use even the most basic of structural test criteria, statement/branch code coverage; the use of other criteria is close to non-existent [8, 12, 14]. Although some cite cost as the major factor preventing adoption [16], the difficulty of understanding and using adequacy criteria is likely of comparable importance.

Zhu et al. categorize test adequacy criteria into three main categories: structural, error-based, and fault-based [16]. The criteria of these categories measure program size along distinct dimensions: structural criteria measure a program's size by its number of control-flow or data-flow structures [11]; error-based criteria measure the number of functionally equivalent subdomains of a program's input space; fault-based criteria (of which mutation testing is the most widely

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
ACM 978-1-59593-811-4/07/0009.

[1]http://junit.org

studied and used) measure the number of "close" variants of the program under test [4, 6].

Of these three categories of adequacy criteria, only fault-based criteria consider whether tests check behavior; structural and error-based criteria relate only what portion of a program's structures or input spaces have been tested. Structural data-flow criteria determine which variable definitions end up being used, and output-influencing-All-du (OI-All-du) in particular determines which variable definitions do not contribute to the output of a program; still, this is different from checking whether tests check outputs [11, 5].

Compared to structural criteria, mutation testing is difficult to automate and relatively difficult to use. The determination of whether a mutant is equivalent to the original program is an undecidable problem requiring human intervention. A live mutant also requires much more thought to understand than an uncovered statement or an unevaluated expression; each live mutant is a different program the user must analyze to understand why a test did not fail. Mutants may not halt, further complicating automation.

Not surprisingly, these difficulties stymie the adoption of fault-based adequacy criteria. For the small portion of projects that use test adequacy criteria, the consequent reliance on structural and error-based criteria opens a behavior checking test adequacy gap. State coverage is designed to fill this gap. It aims to report inadequacies similar to those of existing fault-based criteria, but for the low cost and effort of structural criteria.

3. STATE COVERAGE DEFINITION

3.1 Matching Checks to Definitions

A program, at its most basic, takes in inputs, processes them, and either terminates with some output or runs indefinitely. A test runs a program and checks that the program's outputs are correct if/when the program terminates.

State coverage determines which of a program's outputs are checked by a test and which are not. (This is different from checking whether the tests' checks are correct, which is known as the *oracle* problem.) State coverage considers all outputs which are available to the test as variables; it considers the state of memory at the time that checks are evaluated, but it does not consider outputs to files, network sockets, or other types of I/O.

Clearly, a faulty variable definition would never be detected if it was never used. This is part of the justification for data-flow based adequacy criteria. However, we are concerned not only with whether the value of the assignment is used by the program, but whether it is checked by a test.

We first define what we mean by a program output, then define what it means to cover the statement that defined that output. We assume the following test structure:

1. The test code initializes state to satisfy preconditions of the test case and the code under test (CUT).

2. The CUT is executed by the test code, "stimulating" the test.

3. Outputs are checked by the test code.

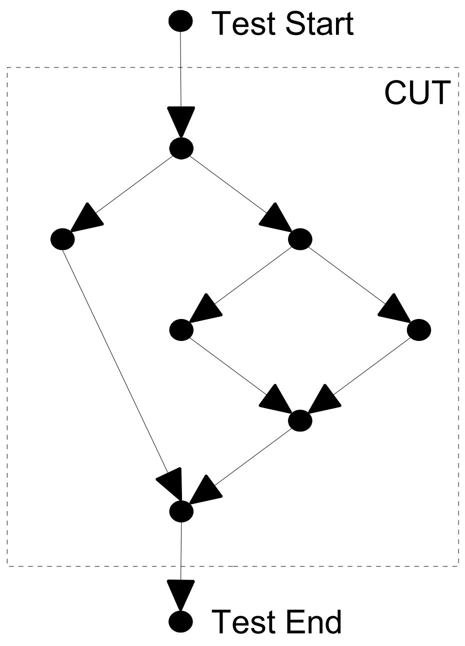

Figure 1: Control-Flow Graph of Test and CUT

The popular xUnit frameworks formalize these phases of the test life-cycle, referring to them as Fixture initialization, Test Case execution, and Checks [2]. These stages correspond to the three elements of a Hoare triple $\{P\}S\{Q\}$, where P is a set of preconditions, S is a set of statements, and Q is a set of predicates that should be true after S has executed subject to P [7].

At this point, it will be useful to refer to the program control flow graph (CFG) of the test and CUT, in which each statement or control predicate corresponds to a node, and each possible flow of control from one node to another is represented by an edge. Since the test executes the CUT, the test and CUT together form a single CFG, with the CUT CFG a subgraph of the whole CFG (see Figure 1). With respect to a CFG, $DEF(i)$ is the set of variables defined at node i, and $REF(i)$ is the set of variables node i references.

DEFINITION 1. *Let T be a test of code under test CUT. Let n_t be the first node executed in T after the last node executed in CUT. Let $n_d \in CUT$ and $x \in DEF(n_d)$. If test T executes a path free of definitions of x from n_d to n_t, and x is still a defined variable at n_t, then x is said to be an* ***output*** *of $< T, CUT >$ and n_d is said to be an* ***output-defining node*** *of x.*

The set of variables X that are outputs of CUT subject to T are the outputs that state coverage will require T to check. If T checks all of the outputs X, then T will provide full state coverage. The requirement that x be a defined variable at n_t excludes local variables of functions, de-allocated memory, and all other temporary state from the output set. It does not, however, exclude variables that might not be visible from test scope, such as the private members of an object.

State coverage presumes the use of a standard construct to check the value of an output, such as a built-in language assertion feature or constructs defined by an xUnit framework. State coverage is, however, construct-independent; it assumes the use of some construct for checking outputs but does not mandate the construct itself. We shall adopt the xUnit term for such constructs and refer to them as checks.

There are several possible ways that an output x might be checked by a check c. The simplest way is for c to use x directly, as in the example check of `testAbsDirectly` (where c is the `assertEquals` call and x is the return value of `abs`).

```
public void testAbsDirectly() {
  assertEquals(5, abs(-5));
}
```

The variable might also be checked indirectly. There might exist, for instance, a variable y that is somehow derived from x. Since state coverage is concerned with detecting outputs that go completely unchecked, state coverage shall consider a check of y to also be a check of x, as in the example `testAbsIndirectly`.

```
public void testAbsIndirectly() {
  boolean y = abs(-5) > 0;
  assertTrue(y);
}
```

We shall consider the statement that defines a variable x to be covered if the value of x or any variable derived from x is used by a check. The state coverage percentage of a test is the number of covered output-defining nodes divided by the total number of output-defining nodes. State coverage is not defined for programs without outputs.

3.2 Calculating State Coverage

We can determine which statements – and by implication, which variable definitions – affected the value used by a check in a particular test via *program slicing* [15]. A program slice is the subset of a program that has affected the value of a variable at some point in program execution. This point is known as a *slicing criterion*, and consists of variables, statements, and possibly program inputs of interest.

Program slicing was originally defined by Weiser to aid in program debugging and comprehension [15]. Among test adequacy criteria, slicing is used by OI-All-du to determine which evaluated D-U pairs also influence output [5].

The numerous variations on Weiser's original definition are split between those that rely on compile-time information (*static program slicing*) to extract slices and those that also incorporate run-time information (*dynamic program slicing*) [13]. While the slicing criterion used for *static state coverage* is the tuple <variable(s), statement>, a dynamic slicing criterion also includes the input used for execution and consists of the triple <variable(s), statement, input(s)>. Since tests specify inputs during the Fixture initialization stage, state coverage can be based on either dynamic or static slicing.

State coverage based on dynamic slicing (*dynamic state coverage*) will be more accurate than static slicing for particular program executions, especially for code that uses pointers, arrays, or dynamic dispatch. Because static slicing does not consider program input, *static state coverage* would not be able to report the adequacy of test checks with respect to the inputs specified by the test. Two tests with the same syntactic structure, differing only in the inputs provided to the program, would always have the same static state coverage even though their dynamic state coverage could totally diverge. However, static state coverage does offer one advantage uncommon among test adequacy criteria: it does not require tests to be executed.

We offer the following definition of dynamic state coverage. The definition of static state coverage differs only in that the slicing criterion does not consider the inputs I.

Definition 2. *Let n_d be the output-defining node of variable x of CUT and T. Let c be a check of T, let I be the inputs to CUT specified by T, and let $X = REF(c)$. Let S be the program slice of CUT with slicing criterion $< X, c, I >$. The variable x and its output-defining node n_d are covered if and only if $n_d \in S$.*

According to the state coverage definition, it is not necessary for $x \in X$ for n_d to be covered (i.e. the variable x may be used indirectly by the check c). However, it follows as a corollary that if $x \in X$, then $n_d \in S$ and n_d is covered.

3.3 Test Suite State Coverage

As defined above, state coverage measures the extent of behavior checking of a single test. The naive strategy to combine state coverage percentage from multiple tests in a suite calculates a weighted mean between all tests' state coverage percentages. This would result in a *pessimistic state coverage* metric, since it would require every test to cover every one of its output-defining nodes.

Definition 3. *Let N_i be the set of output-defining nodes of CUT subject to test t_i, and V_i be the set of covered output-defining nodes of CUT subject to test t_i. The **pessimistic state coverage** of test suite T is*

$$\frac{\sum |V_i|}{\sum |N_i|}$$

Especially among test-driven developers, it is considered good practice to write unit tests with small to medium scope [3]. Some even advocate writing tests to have exactly one check each [1]. Pessimistic state coverage percentages for suites composed of many small-granularity tests would be artificially low.

A more optimistic state coverage calculation would require each output-defining node to be covered by at least one test. Considering output-defining nodes per suite rather than per test has the advantage of rendering the state coverage calculation insensitive to test suite style.

Definition 4. *Let N_i be the set of output-defining nodes of CUT subject to test t_i, and V_i be the set of covered output-defining nodes of CUT subject to test t_i. The **optimistic state coverage** of test suite T is*

$$\frac{|\cup V_i|}{|\cup N_i|}$$

4. PRELIMINARY EVALUATION

We performed an experiment comparing mutation adequacy to state coverage statistics. We hypothesized that since both mutation testing and state coverage measure the extent of behavior checking, the two metrics would positively correlate to the number of checks in a test suite.

4.1 Experimental Design

We used version 1.0 of the open source Apache Jakarta Commons Lang[2] library as a test subject. This library of helper utilities for core Java classes consists of 26 top-level classes and 1718 lines of code. Its 309 test methods provide 74.5% statement/condition code coverage and 1607 checks.

We reduced the number of checks in the library's test suite at each stage of the experiment by randomly and cumulatively deactivating 40 to 50 of the suite's checks. After each round, we re-calculated the suite's optimistic state coverage statistic and the suite's mutation adequacy.

The Indus Java Program Slicer [10], a static slicer, was used to calculate the slices using the remaining active checks as slicing criteria. The line numbers of these slices were intersected with the line numbers of output-defining nodes to determine the state coverage score.

Mutation adequacy was calculated using muJava [9]. Except for interfaces and abstract classes, all classes were mutated with the full set of class-level and method-level mutator operations.

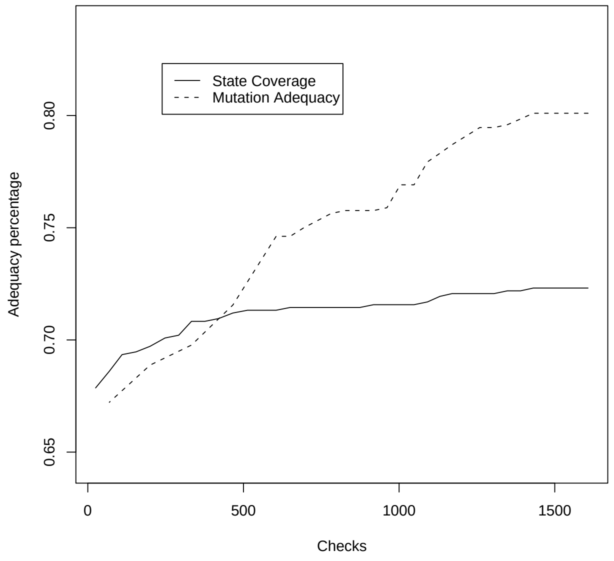

Figure 2: Commons Lang State Coverage and Mutation Adequacy at Varying Numbers of Checks

4.2 Experiment Results

Figure 2 shows the observed relationship between state coverage, mutation adequacy, and the number of checks in the Commons Lang test suite. As hypothesized, both state coverage and mutation adequacy were strongly positively correlated with the number of checks. For 25 to 1607 checks, the correlation coefficient (ρ) between the number of checks and state coverage was 0.88; between the number of checks and mutation adequacy it was 0.97. Sensitivity to checks was more consistent for mutation testing across the whole range of checks. State coverage responded more consistently to lower numbers of checks, with a 0.96 correlation coefficient to the number of checks for stages with less than 500 checks. This can probably be attributed to the use of static slicing. Numerous tests in the suite have duplicate call sequences differing only in input values; static state coverage is affected only by removing the last of those tests, a more likely scenario in the later stages of check deactivation.

[2]http://jakarta.apache.org/commons/lang

5. CONCLUSIONS AND FUTURE WORK

State coverage holds promise to provide software projects with a practical adequacy criterion for checks of behavior. Our experiences running the experiment of Section 4 confirmed our beliefs that state coverage would require less human intervention than mutation testing, yet still provide comparable feedback on the quality of test behavior checks. Although state coverage was less sensitive to behavior checks than mutation testing, state coverage did not report false positives. We believe that dynamic state coverage should improve on the results of static state coverage, approaching the behavior checking capabilities of mutation testing. More work is required to explore dynamic state coverage, as well as to validate state coverage against larger, more representative projects.

6. ACKNOWLEDGMENTS

Venkatesh Prasad Ranganath provided invaluable help with the Indus program slicer. We also thank the numerous Agitar Software colleagues who provided feedback on this work.

7. REFERENCES

[1] D. Astels. One assertion per test. http://www.artima.com/weblogs/viewpost.jsp?thread=35578.

[2] K. Beck. *Kent Beck's Guide to Better Smalltalk*. Cambridge University Press, 1998.

[3] T. Burns. Effective unit testing. *ACM Ubiquity*, 1(42), 2001.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4), April 1978.

[5] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Second Irvine Software Symposium*, 1992.

[6] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4), July 1977.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[8] D. Hyland-Wood and D. Carrington. 2006 software engineering practices survey summary of results. Technical report, The University of Queensland, 2007. http://www.itee.uq.edu.au/~dwood/2006SEPSurvey/2006SEPSurveyResults.html.

[9] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava : An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[10] V. P. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri, 2005. http://people.cis.ksu.edu/~rvprasad/publications/sttt05-submission.pdf

[11] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, 1985.

[12] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[13] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[14] R. Torkar and S. Mankefors. A survey on testing and reuse. In *Proceedings of the 2003 IEEE International Conference on Software-Science, Technology & Engineering*, 2003.

[15] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[16] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.