A State Coverage Tool for JUnit

Ken Koster
Agitar Software Laboratories
450 National Avenue
Mountain View, California 94043
kenk@agitar.com

ABSTRACT

We present a JUnit test runner that informs users of missing behavior checks in their tests. The tool tracks variable updates and definitions over the course of a test execution and determines which variables influence which assertions via dynamic taint analysis. The program statements that set outputs which do not influence the outcome of any test assertions are reported as state coverage inadequacies. With traditional code coverage tools, users can ensure that tests execute all program statements; with this tool, they can additionally ensure that program output is checked, in one way or another, by a test.

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, state coverage, structural testing, taint analysis, test adequacy criteria, unit testing

1. INTRODUCTION

State coverage is a test adequacy criterion that requires tests to check programs' output variables [3]. All variables still defined when executing in test scope (even those which are not visible, such as private fields of objects) are considered by state coverage. Each program statement that sets an output variable (referred to as an *output-defining statement*, or ODS) which is not subsequently checked by a test is a state coverage inadequacy. Each inadequacy points to a test weakness; since the test does not specify the behavior of an uncovered ODS, the ODS might be faulty or might become faulty in the future despite its execution by the test.

In the example JUnit[1] test below, the test of the `min` method, `testMinWithoutCheck`, does not fail despite its execution of a fault.

Without a programmatic assertion on the return value of `min`, such as `assertEquals(1, x)`, human inspection is required to discover its erroneous behavior. A state coverage report, however, would flag the executed `return` statement of `min` as an inadequacy of `testMinWithoutCheck`.

We have developed a tool that runs JUnit tests and produces a state coverage report, identifying all unchecked ODS.

```
int min(int a, int b) {
  if (a < b)  // check should be: a > b
    return b;
  else
    return a;
}

public void testMinWithoutCheck() {
  int x = min(3, 1);
  System.out.println("min(1,3):" + x);
}
```

Programmers can improve their tests by adding assertions in a directed fashion to inadequate tests. Even if programmers are not inclined to revise their tests, state coverage can be used by software development managers as a metric to assess the thoroughness of test suites.

2. STATE COVERAGE REPORTS

A state coverage report lists the ODS that were executed for each test run. For each ODS, it indicates whether it was covered, and if so, by which assertions. Figure 1 is a representation of a state coverage report of a Java class, `HumanName`, and its JUnit test.

The `HumanName` class has seven potential ODS. Statements that set variables that will still be defined from a test's scope are potential ODS. Java's potential ODS are returns from non-private, non-void methods; field assignments (both static and instance fields); and constructor returns (although Java constructors do not have explicit return statements, they implicitly return the object being constructed). The `HumanName` constructors contribute three ODS each, two from field assignments and one from an implicit constructor return. The method `isCelebrity` has one return statement, and thus one ODS.

The test method `testCelebrity` executes four ODS and checks three of them. By checking `isCelebrity`'s return value, it directly checks the executed `isCelebrity` output-defining return statement of line 17. Because the return value of `isCelebrity` is derived from the value of `first`, the executed assignment of that field on line 13 is indirectly checked. Since the `assertTrue` statement dereferences the object returned from the constructor, it indirectly checks the return ODS of that object's constructor (by acting as an implicit `assertNotNull` check of the variable `pele`). However,

[1]http://junit.org

ICSE'08, May 10–18, 2008, Leipzig, Germany.
ACM 978-1-60558-079-1/08/05.

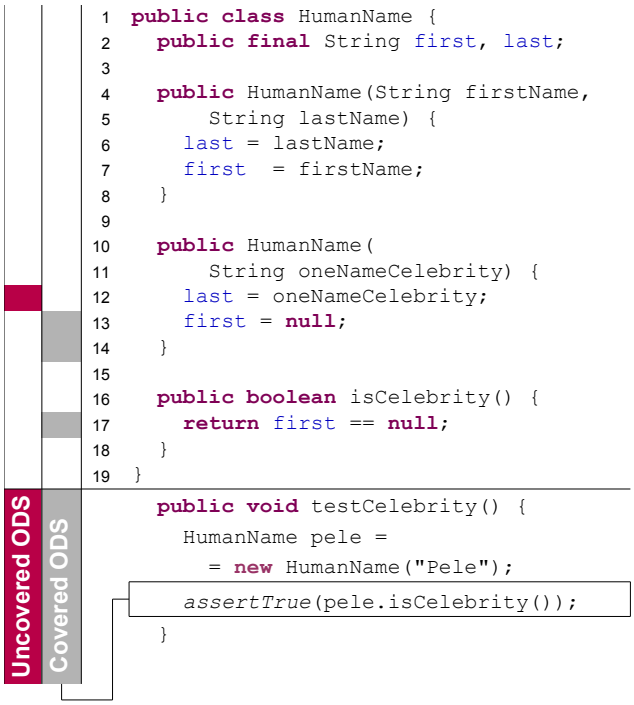

Figure 1: Example State Coverage Report

the assignment of the field `last` goes unchecked by the test, and is flagged by the tool as an uncovered ODS.

One interpretation of the uncovered `HumanName` ODS is that the test does not specify the behavior of the constructor with respect to the field `last`. The pessimistic interpretation of this uncovered ODS is that there are potential faults in the assignment of `last`. In either case, users of the test runner receive feedback about unchecked program output.

The tool's reports are based on *dynamic* state coverage. Although `HumanName` has seven potential ODS, only the four that were executed by tests figure in the report; the three ODS from the unexecuted two-argument constructor are not included in the state coverage calculation.

3. IMPLEMENTATION

State coverage was originally defined in terms of program slices [3, 6]. Given the challenge of implementing a highly performant and scalable dynamic slicing algorithm, and the relative success with which dynamic taint analysis has been applied to large programs, we chose to implement Stacoru via a lightweight Java virtual machine in Java with hooks for dynamic taint analysis [1, 2, 7].

Data taint analysis – and more generally information flow – has been a focus of study by the security community [5], but has also seen use in other activities, such as testing and debugging. A taint analysis marks data values coming from *sources* of interest [4]. Whenever a tainted value is used to compute another value, taints from the tainted value propagate to the new value. This propagation can be based on data-flow (when a new value derives directly from a tainted value) or on control-flow (when tainted values are used in conditions). Ultimately, tainted values are traced to *sinks*,

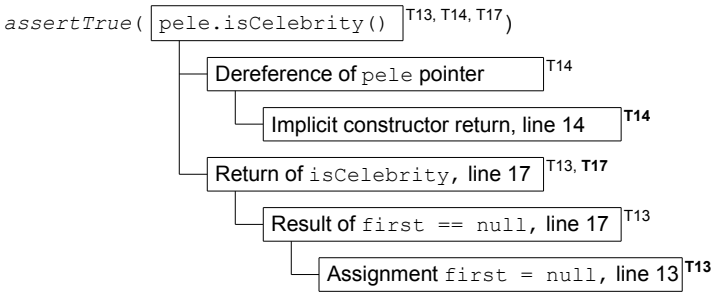

Figure 2: Taint Propagation in testCelebrity

where the impact of using tainted values is evaluated. A canonical example comes from web application security: web forms for user data entry are taint sources; values entered on those forms are tainted; those taints are tracked to sinks such as database and file system operations.

The state coverage test runner uses ODS as taint sources and assertions as taint sinks. The taints processed by sinks lead to covered ODS, while those that do not end up in any sinks correspond to uncovered ODS. The tool propagates taints through both control-flow and data-flow.

Figure 2 depicts the taint propagation pathway for the `testCelebrity` example, from their sources on lines 13, 14, and 17, to their processing by the `assertTrue` sink. In the figure, taint names are formed with the letter "T" and the line number of the ODS taint source (e.g. "T13" for a taint from line 13), appear as superscripts, and are in bold typeface at their source.

The test runner currently runs tests approximately 70 times slower than the standard JUnit test runner. Of this slowdown, more than half is due to running in an interpreter that is written in Java; the other half is due to the taint analysis. In practice, this slowdown is not prohibitive for interactively running the small scope, unit-level tests for which state coverage is designed.

4. REFERENCES

[1] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, 2007.

[2] V. Haldar, D. Chandra, and M. Franz. Dynamic taint propagation for java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311, 2005.

[3] K. Koster and D. C. Kao. State coverage: A structural test adequacy criterion for behavior checking. In *ESEC/FSE*, 2007.

[4] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium*, pages 18–18, 2005.

[5] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 2003.

[6] M. Weiser. Program slicing. In *ICSE*, 1981.

[7] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In *Conference on Programming Language Design and Implementation*, 2006.