Chapter 9

Test Case Selection and Adequacy

A key problem in software testing is selecting and evaluating test cases. This chapter introduces basic approaches to test case selection and corresponding adequacy criteria. This chapter serves as a general introduction to the problem and provides a conceptual framework for functional and structural approaches described in subsequent chapters.

Required Background

- Chapter 2

  The fundamental problems and limitations of test case selection are a consequence of the undecidability of program properties. A grasp of the basic problem is useful in understanding Section 9.3.

9.1 Overview

Experience suggests that software that has passed a thorough set of systematic tests is likely to be more dependable than software that has been only superficially or haphazardly tested. Surely we should require that each software module or subsystem undergo thorough, systematic testing before being incorporated into the main product. But what do we mean by thorough testing? What is the criterion by which we can judge the adequacy of a suite of tests that a software artifact has passed?

Ideally we should like an “adequate” test suite to be one that ensures correctness of the product. Unfortunately, that goal is not attainable. The difficulty of proving that some set of test cases is adequate in this sense is equivalent to the difficulty of proving that the program is correct. In other words, we could have “adequate” testing in this sense only if we could establish correctness without any testing at all.

In practice we settle for criteria that identify inadequacies in test suites. For example, if the specification describes different treatment in two cases, but the test suite
Test Case Selection and Adequacy

does not check that the two cases are in fact treated differently, then we may conclude that the test suite is inadequate to guard against faults in the program logic. If no test in the test suite executes a particular program statement, we might similarly conclude that the test suite is inadequate to guard against faults in that statement. We may use a whole set of (in)adequacy criteria, each of which draws on some source of information about the program and imposes a set of obligations that an adequate set of test cases ought to satisfy. If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite. If a set of test cases satisfies all the obligations by all the criteria, we still do not know definitively that it is a well-designed and effective test suite, but we have at least some evidence of its thoroughness.

9.2 Test Specifications and Cases

A test case includes not only input data but also any relevant execution conditions and procedures, and a way of determining whether the program has passed or failed the test on a particular execution. The term “input” is used in a very broad sense, which may include all kinds of stimuli that contribute to determining program behavior. For example, an interrupt is as much an input as is a file. The pass/fail criterion might be given in the form of expected output, but could also be some other way of determining whether a particular program execution is correct.

A test case specification is a requirement to be satisfied by one or more actual test cases. The distinction between a test case specification and a test case is similar to the distinction between a program specification and a program. A test case specification might be met by several different test cases, and vice versa. Suppose, for example, we are testing a program that sorts a sequence of words. “The input is two or more words” would be a test specification, while test cases with the input values “alpha beta” and “Milano Paris London” would be two among many test cases satisfying the test specification. A test case with input “alpha beta,” on the other hand, would satisfy both the test specification “the input is two or more words” and the test specification “the input is eight or fewer words.”

Characteristics of the input are not the only thing that might be mentioned in a test case specification. A complete test case specification includes pass/fail criteria for judging test execution and may include requirements, drawn from any of several sources of information, such as system, program, and module interface specifications; source code or detailed design of the program itself; and records of faults encountered in other software systems.

Test specifications drawn from system, program, and module interface specifications often describe program inputs, but they can just as well specify any observable behavior that could appear in specifications. For example, the specification of a database system might require certain kinds of robust failure recovery in case of power loss, and test specifications might therefore require removing system power at certain critical points in processing. If a specification describes inputs and outputs, a test specification could prescribe aspects of the input, the output, or both. If the specification is modeled as an extended finite-state machine, it might require executions corresponding to
### Testing Terms

While the informal meanings of words like “test” may be adequate for everyday conversation, in this context we must try to use terms in a more precise and consistent manner. Unfortunately, the terms we will need are not always used consistently in the literature, despite the existence of an IEEE standard that defines several of them. The terms we will use are defined below.

**Test case:** A test case is a set of inputs, execution conditions, and a pass/fail criterion. (This usage follows the IEEE standard.)

**Test case specification:** A test case specification is a requirement to be satisfied by one or more actual test cases. (This usage follows the IEEE standard.)

**Test obligation:** A test obligation is a partial test case specification, requiring some property deemed important to thorough testing. We use the term “obligation” to distinguish the requirements imposed by a test adequacy criterion from more complete test case specifications.

**Test suite:** A test suite is a set of test cases. Typically, a method for functional testing is concerned with creating a test suite. A test suite for a program, system, or individual unit may be made up of several test suites for individual modules, subsystems, or features. (This usage follows the IEEE standard.)

**Test or test execution:** We use the term test or test execution to refer to the activity of executing test cases and evaluating their results. When we refer to “a test,” we mean execution of a single test case, except where context makes it clear that the reference is to execution of a whole test suite. (The IEEE standard allows this and other definitions.)

**Adequacy criterion:** A test adequacy criterion is a predicate that is true (satisfied) or false (not satisfied) of a ⟨program, test suite⟩ pair. Usually a test adequacy criterion is expressed in the form of a rule for deriving a set of test obligations from another artifact, such as a program or specification. The adequacy criterion is then satisfied if every test obligation is satisfied by at least one test case in the suite.
particular transitions or paths in the state-machine model. The general term for such test specifications is “functional testing,” although the term “black-box testing” and more specific terms like “specification-based testing” and “model-based testing” are also used.

Test specifications drawn from program source code require coverage of particular elements in the source code or some model derived from it. For example, we might require a test case that traverses a loop one or more times. The general term for testing based on program structure is “structural testing,” although the term “white-box testing” or “glass-box testing” is sometimes used.

Previously encountered faults can be an important source of information regarding useful test cases. For example, if previous products have encountered failures or security breaches due to buffer overflows, we may formulate test requirements specifically to check handling of inputs that are too large to fit in provided buffers. These fault-based test specifications usually draw also from interface specifications, design models, or source code, but add test requirements that might not have been otherwise considered. A common form of fault-based testing is fault-seeding, purposely inserting faults in source code and then measuring the effectiveness of a test suite in finding the seeded faults, on the theory that a test suite that finds seeded faults is likely also to find other faults.

Test specifications need not fall cleanly into just one of the categories. For example, test specifications drawn from a model of a program might be considered specification-based if the model is produced during program design, or structural if it is derived from the program source code.

Consider the Java method of Figure 9.1. We might apply a general rule that requires using an empty sequence wherever a sequence appears as an input; we would thus create a test case specification (a test obligation) that requires the empty string as input.\(^1\) If we are selecting test cases structurally, we might create a test obligation that requires the first clause of the `if` statement on line 16 to evaluate to true and the second clause to evaluate to false, and another test obligation on which it is the second clause that must evaluate to true and the first that must evaluate to false.

### 9.3 Adequacy Criteria

We have already noted that adequacy criteria are really imperfect but useful indicators of inadequacies, so we may not always wish to use them directly to generate test specifications from which actual test cases are drawn. We will use the term “test obligation” for test specifications imposed by adequacy criteria, to distinguish them from test specifications that are actually used to derive test cases. Thus, the usual situation will be that a set of test cases (a test suite) is created using a set of test specifications, but then the adequacy of that test suite is measured using a different set of test obligations.

We say a test suite satisfies an adequacy criterion if all the tests succeed and if every test obligation in the criterion is satisfied by at least one of the test cases in the test suite. For example, the statement coverage adequacy criterion is satisfied by a

---

1Constructing and using catalogs of general rules like this is described in Chapter 10.
/**
 * Remove/collapse multiple spaces.
 * @param String string to remove multiple spaces from.
 * @return String
 */

public static String collapseSpaces(String argStr)
{
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0; cIdx < argStr.length(); cIdx++)
    {
        char ch = argStr.charAt(cIdx);
        if (ch != ' ' || last != ' ')
        {
            argBuf.append(ch);
            last = ch;
        }
    }

    return argBuf.toString();
}

Figure 9.1: A Java method for collapsing sequences of blanks, excerpted from the
StringUtils class of Velocity version 1.3.1, an Apache Jakarta project. (c) Apache
Group, used by permission.
particular test suite for a particular program if each executable statement in the program (i.e., excluding comments and declarations) is executed by at least one test case in the test suite. A fault-based adequacy criterion that seeds a certain set of faults would be satisfied if, for each of the seeded faults, there is a test case that passes for the original program but fails for the program with (only) that seeded fault.

It is quite possible that no test suite will satisfy a particular test adequacy criterion for a particular program. For example, if the program contains statements that can never be executed (perhaps because it is part of a sanity check that can be executed only if some other part of the program is faulty), then no test suite can satisfy the statement coverage criterion. Analogous situations arise regardless of the sources of information used in devising test adequacy criteria. For example, a specification-based criterion may require combinations of conditions drawn from different parts of the specification, but not all combinations may be possible.

One approach to overcoming the problem of unsatisfiable test obligations is to simply exclude any unsatisfiable obligation from a criterion. For example, the statement coverage criterion can be modified to require execution only of statements that can be executed. The question of whether a particular statement or program path is executable, or whether a particular combination of clauses in a specification is satisfiable, or whether a program with a seeded error actually behaves differently from the original program, are all provably undecidable in the general case. Thus, while tools may be some help in distinguishing feasible from infeasible test obligations, in at least some cases the distinction will be left to fallible human judgment.

If the number of infeasible test obligations is modest, it can be practical to identify each of them, and to ameliorate human fallibility through peer review. If the number of infeasible test obligations is large, it becomes impractical to carefully reason about each to avoid excusing an obligation that really is feasible, though difficult to satisfy. A common practice is to measure the extent to which a test suite approaches an adequacy criterion. For example, if an adequacy criterion based on control flow paths in a program unit induced 100 distinct test obligations, and a test suite satisfied 85 of those obligations, then we would say that we had reached 85% coverage of the test obligations.

Quantitative measures of test coverage are widely used in industry. They are simple and cheap to calculate, provide some indication of progress toward thorough testing, and project an aura of objectivity. In managing software development, anything that produces a number can be seductive. One must never forget that coverage is a rough proxy measure for the thoroughness and effectiveness of test suites. The danger, as with any proxy measure of some underlying goal, is the temptation to improve the proxy measure in a way that does not actually contribute to the goal. If, for example, 80% coverage of some adequacy criterion is required to declare a work assignment complete, developers under time pressure will almost certainly yield to the temptation to design tests specifically to that criterion, choosing the simplest test cases that achieve the required coverage level. One cannot entirely avoid such distortions, but to the extent possible one should guard against them by ensuring that the ultimate measure of performance is preventing faults from surviving to later stages of development or deployment.
9.4 Comparing Criteria

It would be useful to know whether one test adequacy criterion was more effective than another in helping find program faults, and whether its extra effectiveness was worthwhile with respect to the extra effort expended to satisfy it. One can imagine two kinds of answers to such a question, empirical and analytical. An empirical answer would be based on extensive studies of the effectiveness of different approaches to testing in industrial practice, including controlled studies to determine whether the relative effectiveness of different testing methods depends on the kind of software being tested, the kind of organization in which the software is developed and tested, and a myriad of other potential confounding factors. The empirical evidence available falls short of providing such clear-cut answers. An analytical answer to questions of relative effectiveness would describe conditions under which one adequacy criterion is guaranteed to be more effective than another, or describe in statistical terms their relative effectiveness.

Analytic comparisons of the strength of test coverage depends on a precise definition of what it means for one criterion to be “stronger” or “more effective” than another. Let us first consider single test suites. In absence of specific information, we cannot exclude the possibility that any test case can reveal a failure. A test suite $T_A$ that does not include all the test cases of another test suite $T_B$ may fail revealing the potential failure exposed by the test cases that are in $T_B$ but not in $T_A$. Thus, if we consider only the guarantees that a test suite provides, the only way for one test suite $T_A$ to be stronger than another suite $T_B$ is to include all test cases of $T_B$ plus additional ones.

Many different test suites might satisfy the same coverage criterion. To compare criteria, then, we consider all the possible ways of satisfying the criteria. If every test suite that satisfies some criterion $A$ is a superset of some test suite that satisfies criterion $B$, or equivalently, every suite that satisfies $A$ also satisfies $B$, then we can say that $A$ “subsumes” $B$.

Test coverage criterion $A$ subsumes test coverage criterion $B$ iff, for every program $P$, every test set satisfying $A$ with respect to $P$ also satisfies $B$ with respect to $P$.

In this case, if we satisfy criterion $C_1$, there is no point in measuring adequacy with respect to $C_2$. For example, a structural criterion that requires exploring all outcomes of conditional branches subsumes statement coverage. Likewise, a specification-based criterion that requires use of a set of possible values for attribute $A$ and, independently, for attribute $B$, will be subsumed by a criterion that requires all combinations of those values.

Consider again the example of Figure 9.1. Suppose we apply an adequacy criterion that imposes an obligation to execute each statement in the method. This criterion can be met by a test suite containing a single test case, with the input value (value of `argStr`) being “doesn’tEvenHaveSpaces.” Requiring both the true and false branches of each test to be taken subsumes the previous criterion, and forces us to at least provide an input with a space that is not copied to the output, but it can still be satisfied by a suite with just one test case. We might add a requirement that the loop be iterated zero times, once, and several times, thus requiring a test suite with at least three test cases. The obligation to execute the loop body zero times would force us to add a test case...
with the empty string as input, and like the specification-based obligation to consider an empty sequence, this would reveal a fault in the code.

Should we consider a more demanding adequacy criterion, as indicated by the subsumes relation among criteria, to be a better criterion? The answer would be “yes” if we were comparing the guarantees provided by test adequacy criteria: If criterion $A$ subsumes criterion $B$, and if any test suite satisfying $B$ in some program is guaranteed to find a particular fault, then any test suite satisfying $A$ is guaranteed to find the same fault in the program. This is not as good as it sounds, though. Twice nothing is nothing. Adequacy criteria do not provide useful guarantees for fault detection, so comparing guarantees is not a useful way to compare criteria.

A better statistical measure of test effectiveness is whether the probability of finding at least one program fault is greater when using one test coverage criterion than another. Of course, such statistical measures can be misleading if some test coverage criteria require much larger numbers of test cases than others. It is hardly surprising if a criterion that requires at least 300 test cases for program $P$ is more effective, on average, than a criterion that requires at least 50 test cases for the same program. It would be better to know, if we have 50 test cases that satisfy criterion $B$, is there any value in finding 250 test cases to finish satisfying the “stronger” criterion $A$, or would it be just as profitable to choose the additional 250 test cases at random?

Although theory does not provide much guidance, empirical studies of particular test adequacy criteria do suggest that there is value in pursuing stronger criteria, particularly when the level of coverage attained is very high. Whether the extra value of pursuing a stronger adequacy criterion is commensurate with the cost almost certainly depends on a plethora of particulars, and can only be determined by monitoring results in individual organizations.

**Open research issues**

There has been a good deal of theoretical research on what one can conclude about test effectiveness from test adequacy criteria. Most of the results are negative: In general, one cannot be certain that a test suite that meets any practical test adequacy criterion ensures correctness, or even that it is more effective at finding faults than another test suite that does not meet the criterion. While theoretical characterization of test adequacy criteria and their properties was once an active research area, interest has waned, and it is likely that future theoretical progress must begin with a quite different conception of the fundamental goals of a theory of test adequacy.

The trend in research is toward empirical, rather than theoretical, comparison of the effectiveness of particular test selection techniques and test adequacy criteria. Empirical approaches to measuring and comparing effectiveness are still at an early stage. A major open problem is to determine when, and to what extent, the results of an empirical assessment can be expected to generalize beyond the particular programs and test suites used in the investigation. While empirical studies have to a large extent displaced theoretical investigation of test effectiveness, in the longer term useful empirical investigation will require its own theoretical framework.
Further Reading

Goodenough and Gerhart made the original attempt to formulate a theory of “adequate” testing [GG75]; Weyuker and Ostrand extended this theory to consider when a set of test obligations is adequate to ensure that a program fault is revealed [WO80]. Gourlay’s exposition of a mathematical framework for adequacy criteria is among the most lucid developments of purely analytic characterizations [Gou83]. Hamlet and Taylor show that, if one takes statistical confidence in (absolute) program correctness as the goal, none of the standard coverage testing techniques improve on random testing [HT90], from which an appropriate conclusion is that confidence in absolute correctness is not a reasonable goal of systematic testing. Frankl and Iakounenko’s study of test effectiveness [FI98] is a good example of the development of empirical methods for assessing the practical effectiveness of test adequacy criteria.

Related Topics

Test adequacy criteria and test selection techniques can be categorized by the sources of information they draw from. Functional testing draws from program and system specifications, and is described in Chapters 10, 11, and 14. Structural testing draws from the structure of the program or system, and is described in Chapters 12 and 13. The techniques for testing object-oriented software described in Chapter 15 draw on both functional and structural approaches. Selection and adequacy criteria based on consideration of hypothetical program faults are described in Chapter 16.