Software Testing

TEST CASE SELECTION AND ADEQUACY TEST EXECUTION

Overview, Test specification and cases, Adequacy criteria, comparing criteria, Overview of test execution, From test case specification to test cases; Scaffolding; Generic versus specific scaffolding; Test Oracles; Self Checks as oracles; Capture and replay
7.1 Learning Objectives

- Understand the purpose of defining test adequacy criteria, and their limitations
- Understand basic terminology of test selection and adequacy
- Know some sources of information commonly used to define adequacy criteria
- Understand how test selection and adequacy criteria are used
- Appreciate the purpose of test automation
  - Factoring repetitive, mechanical tasks from creative, human design tasks in testing
- Recognize main kinds and components of test scaffolding
- Understand some key dimensions in test automation design
  - Design for testability: Controllability and observability
  - Degrees of generality in drivers and stubs
  - Comparison-based oracles and self-checks
7.2 TEST CASE SELECTION AND ADEQUACY CRITERIA

Overview, Test specification and cases, Adequacy criteria, comparing criteria,

OVERVIEW

This Chapter provides an overview of test case selection and adequacy criteria. The key problem in software testing is selecting and evaluating test cases.

Though experience suggests that software that has undergone a systematic testing is more dependable than the software that has been only superficially tested, determining the criteria for adequacy of testing or when to stop testing is a critical issue for successful software testing this objective of this chapter is to

• Understand the purpose of defining test adequacy criteria, and their limitations
• Understand basic terminology of test selection and adequacy
• Know some sources of information commonly used to define adequacy criteria
• Understand how test selection and adequacy criteria are used

7.3 TEST SPECIFICATION AND CASES

Definition OF Terms

• Test case: a set of inputs, execution conditions, and a pass/fail criterion.
• Test case specification: a requirement to be satisfied by one or more test cases.
• Test obligation: a partial test case specification, requiring some property deemed important to thorough testing.
• Test suite: a set of test cases.
• Test or test execution: the activity of executing test cases and evaluating their results.
• Adequacy criterion: a predicate that is true (satisfied) or false (not satisfied) of a \(\langle\text{program, test suite}\rangle\) pair.

The design of software testing can be a challenging process. However software engineers often see testing as an after thought, producing test cases that feel right but have little assurance that they are complete. The objective of testing is to have the highest likelihood of finding the most errors with a minimum amount of timing and effort. A large number of test case design methods have been developed that offer the developer with a systematic
approach to testing. Methods offer an approach that can ensure the completeness of tests and offer the highest likelihood for uncovering errors in software.

A Test Case Includes input, the expected output, pass/fail criteria and the environment in which the test is being conducted. Here the term input means all the data that is required to make a test case. A Test Case specification is a requirement to be satisfied by one or more test cases. Specification-based testing uses the specification of the program as the point of reference for test input data selection and adequacy. A test specification can be drawn from system, interface or program. The distinction between a test case and test specification is similar to the distinction between program specification and program.

Software products can be tested in two ways:
(1) Knowing the specified functions that the product has been designed to perform, tests can be performed that show that each function is fully operational
(2) knowing the internal workings of a product, tests can be performed to see if they jell.

The first test approach is known as a black box testing and the second white box testing.

Software Test cases derived from test specifications based on system requirements are generally termed as functional or black box testing.

Software Test cases derived from specifications of interface and programs are generally termed as glass box or white box testing.

Test cases should uncover errors like:

1. Comparison of different data types
2. Incorrect logical operators are precedence
3. Expectation of equality when precision error makes equality unlikely
4. Incorrect comparison or variables
5. Improper or non-existent loop termination.
6. Failure to exit when divergent iteration is encountered
7. Improperly modified loop variables.
Good design dictates that error conditions be anticipated and error handling paths set up to reroute or cleanly terminate processing when an error does occur.

7.4 TEST ADEQUACY

- Adequacy criterion = set of test obligations
- A test suite satisfies an adequacy criterion if
  - all the tests succeed (pass)
  - Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.
  - Example: the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”.
- Sometimes no test suite can satisfy a criterion for a given program
  - Example: Defensive programming style includes “can’t happen” sanity checks
    
    ```java
    if (z < 0) {
        throw new LogicError(
            “z must be positive here!”
        )
    }
    ```

No test suite can satisfy statement coverage for this program (if it’s correct)

Test adequacy criteria is useful to

- Test selection approaches
  - Guidance in devising a thorough test suite
    - Example: A specification-based criterion may suggest test cases covering representative combinations of values
- Revealing missing tests
  - Post hoc analysis: What might I have missed with this test suite?
- Often in combination
  - Example: Design test suite from specifications, then use structural criterion (e.g., coverage of all branches) to highlight missed logic
• Adequacy criteria provide a way to define a notion of “thoroughness” in a test suite
  - But they don’t offer guarantees; more like design rules to highlight inadequacy
• Defined in terms of “covering” some information
  - Derived from many sources: Specs, code, models, ...
• May be used for selection as well as measurement
  - With caution! An aid to thoughtful test design, not a substitute

7.5 TEST COVERAGE
• Measuring coverage (% of satisfied test obligations) can be a useful indicator of progress toward a thorough test suite, of trouble spots requiring more attention. However, care must be taken so that Coverage will not be a proxy for thoroughness or adequacy
• It’s easy to improve coverage without improving a test suite (much easier than designing good test cases)
• The only measure that really matters is (cost-)effectiveness

7.6 COMPARING CRITERIA
• How to distinguish stronger from weaker adequacy criteria?
  - Empirical approach: Study the effectiveness of different approaches to testing in industrial practice
    • Depends on the setting; may not generalize from one organization or project to another
  - Analytical approach: Describe conditions under which one adequacy criterion is provably stronger than another
    • Stronger = gives stronger guarantees
    • One piece of the overall “effectiveness” question

THE SUBSUMES RELATION
Test adequacy criterion A subsumes test adequacy criterion B if, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P.
• Example:


- Exercising all program branches (branch coverage) *subsumes* exercising all program statements

- A common analytical comparison of closely related criteria
  - Useful for working from easier to harder levels of coverage, but not a direct indication of quality.

### 7.7 TEST EXECUTION

**Automating Test Execution**

Designing test cases and test suites is creative - Like any design activity: A demanding intellectual activity, requiring human judgment

Executing test cases should be automatic - Design once, execute many times

Test automation separates the creative human process from the mechanical process of test execution.

### 7.8 From test case Specification to Test Cases

- Test design often yields test case specifications, rather than concrete data
  - Ex: “a large positive number”, not 420023
  - Ex: “a sorted sequence, length > 2”, not “Alpha, Beta, Chi, Omega”

- Other details for execution may be omitted

- Generation creates concrete, executable test cases from test case specifications

Example Tool Chain for Test Case Generation & Execution:

- We could combine ... A combinatorial test case generation (like genpairs.py) to create test data
  - Optional: Constraint-based data generator to “concretize” individual values, e.g., from “positive integer” to 42
  - DDSteps to convert from spreadsheet data to JUnit test cases
  - JUnit to execute concrete test cases

- Many other tool chains are possible ...
7.9 Scaffolding

Code produced to support development activities (especially testing), Not part of the “product” as seen by the end user May be temporary (like scaffolding in construction of buildings) Includes Test harnesses, drivers, and stubs.

- Test driver
  - A “main” program for running a test
    - May be produced before a “real” main program
    - Provides more control than the “real” main program
      - To driver program under test through test cases

- Test stubs
  - Substitute for called functions/methods/objects

- Test harness
  - Substitutes for other parts of the deployed environment
    - Ex: Software simulation of a hardware device

7.10 Controllability & Observability

Example: We want automated tests, but interactive input provides limited control and graphical output provides limited observability

```
GUI input (MVC "Controller")

Program Functionality

Graphical output (MVC "View")
```

A design for automated test includes provides interfaces for control (API) and observation (wrapper on output).
It should be generic or specific?

- How general should scaffolding be? To answer
  - We could build a driver and stubs for each test case or at least factor out some common code of the driver and test management (e.g., JUnit)
  - ... or further factor out some common support code, to drive a large number of test cases from data (as in DDSteps)
  - ... or further, generate the data automatically from a more abstract model (e.g., network traffic model)

- A question of costs and re-use
  - Just as for other kinds of software

7.11 Test Oracles

- Did this test case succeed, or fail? No use running 10,000 test cases automatically if the results must be checked by hand!

- Range of specific to general, again, ex. JUnit: Specific oracle (“assert”) coded by hand in each test case

- Typical approach: “comparison-based” oracle with predicted output value, Not the only approach!

7.12 Comparison-based oracle
With a comparison-based oracle, we need predicted output for each input

- Oracle compares actual to predicted output, and reports failure if they differ

- Fine for a small number of hand-generated test cases

  - E.g., for hand-written JUnit test cases

**Self-Checking Code as Oracle**

- An oracle can also be written as *self-checks*. Often possible to judge correctness without predicting results

- Advantages and limits: Usable with large, automatically generated test suites, but often only a *partial* check. E.g., structural invariants of data structures, recognize *many* or *most* failures, but not all

### 7.13 Capture and Replay

- Sometimes there is no alternative to human input and observation: Even if we separate testing program functionality from GUI, some testing of the GUI is required
- We can at least cut *repetition* of human testing

- *Capture* a manually run test case, *replay* it automatically

  - with a comparison-based test oracle: behavior same as previously accepted behavior

    - reusable only until a program change invalidates it

    - lifetime depends on abstraction level of input and output