Blackbox Testing

Arnav Gupta

April 5, 2024

Contents

1 Blackbox Testing

Testing without having insight into code, based on a system's specification rather than structure. Can have notion of coverage applied.

Rigorous specifications help functional testing, test case generation, and test oracles.

Advantages:

- no need for source code
- wide applicability

Disadvantages:

- does not test hidden functions

1.1 Error Guessing

Ad-hoc approach based on experience, developing and maintaning own error models.

Approach:

1. Make list of possible errors or error-prone situations
2. Yields an error model
3. Design test cases to cover error model

1.2 Test Case Derivation from Functional Requirements

Involves clarification and restatement of the requirements such that requirements are testable. Requires point form formulation that enumerates single requirements and groups related requirements.

For each requirement, provide

- one test case showing requirement functioning
- one test case trying to falsify something
- test boundaries and constraints when possible

1.3 Test Case Derivation from Use Cases

For each use case:

1. develop a scenario graph
2. determine all possible scenarios
3. analyze and rank scenarios
4. generate test cases from scenarios to meet coverage goal
5. execute test cases

Scenario graphs are generated from a use case:

- a node corresponds to a point where we wait for the next event to occur
- one starting node, end of use case is finish node
- edge corresponds to event occurrence
- full scenario is a path from start to finish

If too many scenarios, rank based on criticality and frequency, always including main scenario, then generate test cases.

2 Equivalence Partitioning

Motivation: want a sense of complete functional testing and hope to avoid redundancy

Equivalence classes (ECs): a partition of the input set according to specification

- entire set is covered: completeness
- disjoint classes: avoid redundancy

ECs must be chosen wisely using heuristics.

Weak equivalence class testing (WECT): choose one variable value from each equivalence class

Strong equivalence class testing (SECT): based on Cartesian product of partition subsets

If error conditions are a high priority, should extend SECT to include both valid and invalid inputs. Equivalence partitioning is appropriate when input data is defined in terms of range and sets of discrete values. SECT makes the assumption that the variables are independent. Heuristics are required in the case of too many test cases.

Advantages of Equivalence Partitioning:

- small number of test cases needed
- probability of uncovering defects with the selected test cases higher than that with a randomly chosen test suite of the same size

Limitations of Equivalence Partitioning:

- strongly typed languages eliminate need for consideration of some invalid inputs
- specification does not always define expected output for invalid test cases
- test selection approach does not work well when input variables are not independent
- brute force definition of test case for every combo of input ECs is impractical

2.1 Heuristics for Identifying EC

For each external input:

1. if input is a range of valid values, define:
    - one valid EC (within the range)
    - two invalid EC (one outside each end of the range)
2. if input is a number N of valid values, define
    - one valid EC
    - two invalid EC (none and more than N)
3. if input is an element from a set of valid values, define:
    - one valid EC (within set)
    - one invalid EC (outside set)
4. if input is a must be situation (condition), define:

- one valid EC (satisfies condition)
- one invalid EC (does not satisfy condition)

5. if there is a reason to believe that elements in an EC are not handled in an identical manner by the program:
   - subdivide EC into smaller ECs
6. consider creating equivalence partitions that handle the default, empty, blank, null, zero, or none conditions

2.2 Test Selection Approach

1. until all valid ECs have been covered by test cases, write a new test case that covers as many of the uncovered valid ECs as possible
2. until all invalid ECs have been covered by test cases, write a test case that covers only one of the uncovered invalid ECs (allows tester to observer response to single invalid input)

3 Boundary Value Analysis

Some typical programming errors tend to occur near extreme values, focus of boundary value testing. Simpler but complementary to previous techniques.

Data types with boundary conditions: numeric, character, position, quantity, speed, location, size, etc.

3.1 Guidelines

Use input variable values (within a class) at min, just above min, a nominal value, just below max, and at max.

Hold all other values at nominal values and include each boundary condition in at least one valid test case.

3.2 Limitations

Test cases are 4 times the number of variables. This works well with variables that represent bounded physical quantities, since there is no consideration of the nature of the function and the meaning of the variables.

For more robust testing, include value just beyond boundary in at least one invalid test case.

3.3 Worst Case Testing

For programs with more than one variable with an extreme value, take the Cartesian product of the min, min+, nom, max-, and max.

Much more effort: 5^n test cases

4 Category Partitioning

Systematic, spec-based methodology that uses an informal functional spec to produce a formal test spec.

Integrated EC testing and boundary value analysis and adds refinements.

Consists of:

- identification of categories
- partitioning each category into choices
- creating test frames as selection of choices
- creating test cases

Steps

1. decompose the fucntional spec into functional units that can be independently tested
2. examine each functional unit
   (a) identify params
   (b) identify environmental conditions
3. find categories for each param and environmental condition (major property or characteristic)
4. subdivide categories further into choices in the same way as equivalence partitioning is applied
5. determine constraints among choices: how they interact and affect each other
6. create test frames

7. create test cases

4.1 Criteria for Using Chocies

All Combinations (AC): typically done when using category partitioning, one value for every choice of every param must be used with one value of every possible choice of every other category

Number of test cases for AC is sum of number of choices in all categories.

Each Choice (EC): weaker criterion, one value from each choice for each category must be used at least in one test case

Number of test cases for EC is maximum number of choices in a category.

Base Choice (BC): a compromise

- a base choice is chosen for each category and a first base test is formed by using the base choice for each category
- subsequent tests chosen by holding all but one base choice constant and forming choice combos by covering all non-base choices of the selected category
- this procedure is repeatd for all categories

Number of test cases for BC is sum of the number of choices in all categories minus the number of categories.

BC can be the simplest, smallest, first in some ordering, or the most likely to be used.

5 Decision Tables

Precise and compact way to model complex logic, by associating conditions with actions to perform.

5.1 Terminology

Limited entry table: condition entries restricted to binary values

Extended entry table: condition entries have more than 2 values

5.1.1 Don't Care

Reduces number of variants, not the same as unknown.

Corresponds to the cases where:

- inputs are necessary but have no effect
- inputs may be omitted
- mutually exclusive cases

When conditions are equivalent classes, inputs are mutually exclusive.

5.2 Software Testing

Condition entries in a decision table are interpreted by a computer program as input and equivalence class of inputs.

Action entries in a decision table are interpreted as output and major functional processing portions.

Rules then become test cases.

Limited entry tables have exponential growth from conditions to rules, though this can be reduced with don't cares.

Less rules than combination rule count indicates missing rules, and more rules than combination count indicates either redundant rules or inconsistent table.

5.3 Applicability

Given spec can be converted to a decision table:

- the order in which predicates or rules are evaluated does not affect rule interpretation or resulting actions
- once a rule is satisfied and the action is selected, no other rule needs to be exaned
- the order of executing actions in a satisfied rule does not matter

5.4 Issues

Before deriving test cases, ensure:

- rules are complete, that every combo of predicate truth values is explicit in the decision table
- rules are consistent, that every combo of predicate truth values is only one action or set of actions

5.5 Guidelines and Observations

Decision table is best for programs with:

- lots of decision making
- important logical relationships among input variables
- calculations involving subsets of input variables
- cause and effect between input and output
- complex computational logic

Decision tables do not scale well and must be iteratively refined.

Important to look for redundant, inconsistent, and missing rules.