Cpt S 422: Software Engineering Principles II

Black-box testing – Part 5

Dr. Venera Arnaoudova

WASHINGTON STATE UNIVERSITY

Black-box testing methods

- ✓ Equivalence Class Partitioning
- ✓ Boundary-Value Analysis
- ✓ Category-Partition
- ✓ Decision tables
- ✓ Cause-Effect Graphs
- ❑ Logic Functions (cont.)

Inactive Clause Coverage (ICC)

- The Active Clause Coverage Criteria focus on making sure that the major clauses do affect their predicates. A complementary criterion to Active Clause Coverage ensures that changing a major clause that should *not* affect the predicate does not, in fact, affect the predicate.

- **Inactive Clause Coverage:** *For each $p \in P$ and each major clause $c_i \in Cp$,* ***choose minor clauses c_j*** *, j <> i so that* ***c_i does not determine p****. There are four test requirements for* ***c_i under these circumstances: (1) c_i evaluates to true with p true, (2) c_i evaluates to false with p true, (3) c_i evaluates to true with p false, and (4) c_i evaluates to false with p false****.*

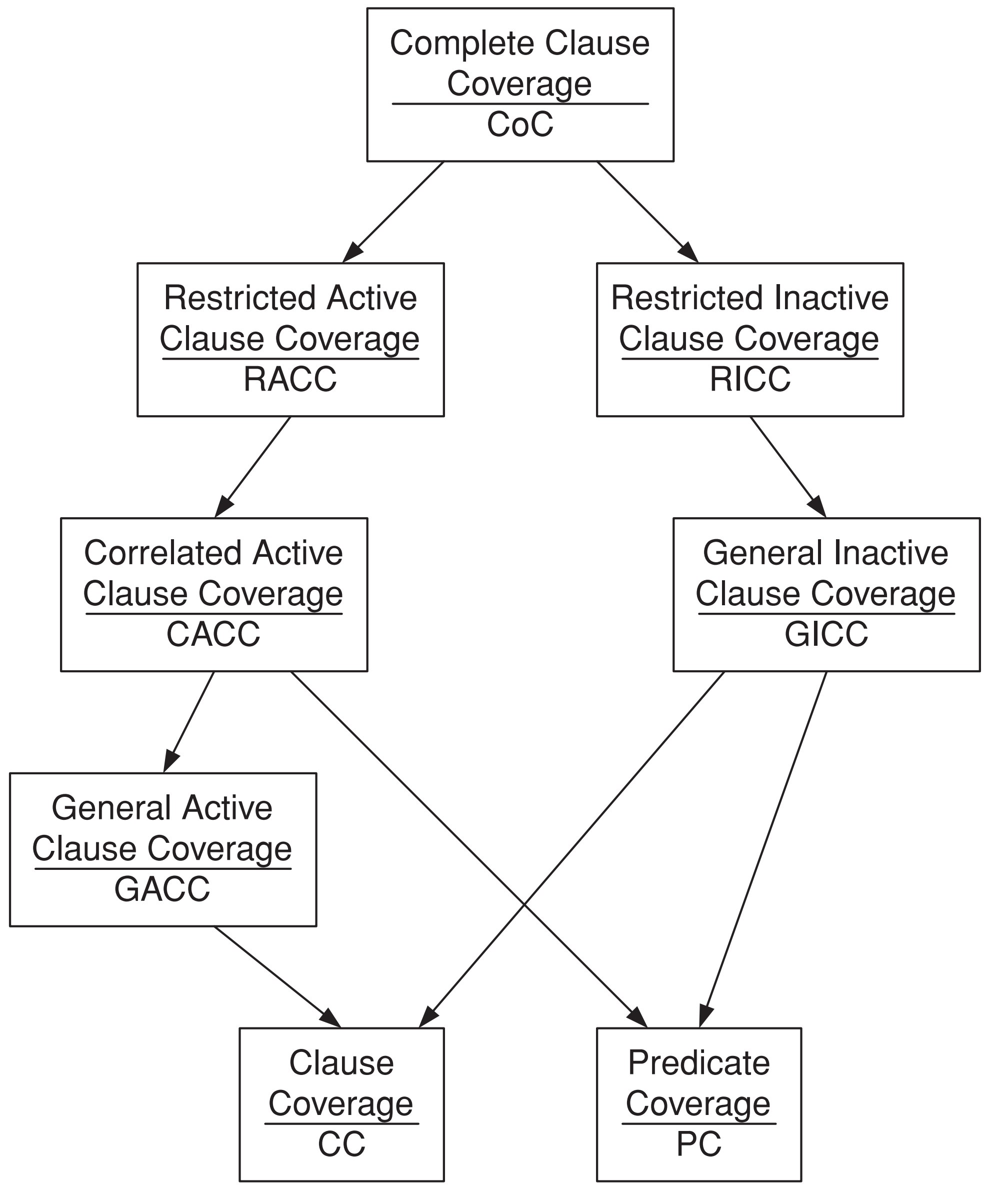
- ICC is subsumed by combinatorial clause coverage and subsumes clause/predicate coverage

General ICC (GICC) and Restricted ICC (RICC)

- **General Inactive Clause Coverage (GICC):** For each $p \in P$ and each major clause $ci \in Cp$, choose minor clauses cj, j <> i so that ci does not determine p. **The values chosen for the minor clauses cj do not need to be the same** when ci is true as when ci is false, that is, cj(ci = true) = cj(ci = false) OR cj(ci = true) <> cj(ci = false) $\forall$ cj.

- **Restricted Inactive Clause Coverage (RICC):** For each $p \in P$ and each major clause $ci \in Cp$, choose minor clauses cj, j <> i so that ci does not determine p. **The values chosen for the minor clauses cj must be the same** when ci is true as when ci is false, that is, it is required that cj(ci = true) = cj(ci = false).

Subsumption among logic coverage criteria



Disjunctive Normal Form (DNF) Coverage Criteria

- Here criteria assume the predicates have been re-expressed in a disjunctive normal form (DNF).

- Criteria:
  - Implicant coverage
  - Prime implicant coverage
  - Variable negation strategy

- Those test strategies allow to reduce the number of test cases, while hopefully preserving most of the test effectiveness

Implicant Coverage (IC)

- **Implicant Coverage**: Given DNF representations of a predicate p and its negation ~p, for each implicant, a test requirement is that **the implicant evaluates to true**.

- This tests different situations in which an action should (not) be taken (e.g., a boiler turned on)

- Steps:
  - Identify the implicants of p
  - Negate p and add its implicants to the set of implicants in the previous step
  - Identify a test set of values that satisfy the criterion

- IC subsumes predicate coverage, but not necessarily Active Clause Criteria.

IC - Example

- p: AB+B~C
- ~p (one representation): ~B+~AC
- Four implicants: {AB, B~C, ~B, ~AC}
- Many test sets can satisfy this criterion, e.g., for ABC, respectively, we can use {TTF, FFT}

Problems with IC

- A problem with IC is that tests might be chosen so that a single test satisfies multiple implicants (as in the previous example).
- Although this lets testers minimize the size of test suites, it is a bad thing from the perspective of testing the unique contributions that each implicant might bring to a predicate.
- Thus we introduce a method to force a kind of “independence" of the implicants.

Prime Implicants

- The first step is to obtain a DNF form where each implicant can be satisfied without satisfying any other implicant.
- Fortunately, standard approaches already exist that can be used. A *proper subterm* of an implicant is the implicant with one or more clauses omitted.
- A *prime implicant* is an implicant such that no proper subterm of the implicant is also an implicant.
- Example: ABC+AB~C+B~C
  - ABC is not a prime implicant because a proper subterm (AB) is also an implicant

Prime Implicant Coverage (PIC)

- Assume that our DNF predicate only contains prime implicants

- PIC: *<u>Given nonredundant, prime-implicant DNF</u> representations of a predicate p and its negation ~p, **for each implicant**, a test requirement is that **the implicant evaluates to true, while all other implicants evaluate to false.***

- An implicant is **redundant** if it can be omitted without changing the value of the predicate.

- In AB+AC+B~C, is AB redundant?

PIC Example & Discussion

- p: AB+B~C
- ~p: ~B+~AC
- Both are nonredundant, prime implicant representations
- Find a test set that satisfies PIC
  - Ex: T = {TTT, FTF, FFF, FTT}
- PIC is a powerful coverage criteria: none of the clause coverage criteria subsume PIC
- Though up to 2^{n-1} prime implicants, many predicates generate a modest number of tests for PIC
- It is an open question whether PIC subsumes any of the clause coverage criteria.

Variable Negation Strategy

- ***Unique true points***: with respect to the *ith* implicant is an assignment of truth values such that the *ith* implicant is true and all other implicants are false.
  - If the set of implicants is {AB~C,AD} then for TTFF: AB~C is true, AD is false
- ***Near false points***: *near false point* for *p* with respect to clause *c* in implicant *i* is an assignment of truth values such that *p* is false, but if *c* is negated and all other clauses are left as is, *i* (and hence *f*) evaluates to true.
  - E.g., (TTTF) for AB~C where if ~C is negated the implicant evaluates to true
- Unique true points or near false points may NOT exist
- Such variants constitute Test Candidate Sets (TCS)
- Generate TCS for each product term in logic function
- The test suite is formed by selecting the smallest suite that covers all unique true points and near false points

Discussion

- If one product term implementation does not evaluate to true when it should - implying that at least one clause in that product term does not evaluate to *true* when expected - test cases from the TCS (***unique true points***) corresponding to the term will be able to detect it, without masking effect from other clauses or terms

- If one product term implementation does not evaluate to false when it should, that is the negation of (at least) a clause has not the effect expected on the logic function (false), test cases from the TCS (***near false points***) corresponding to the negated clause will be able to detect it, without masking effect from other clauses or terms

Variable Negation Strategy versus Faults

- Expression Negation Fault (ENF): Any point in the Boolean space
- Clause Negation Fault (CNF): Any unique true point or near false point for the faulty term and clause negated
- Term Omission Fault (TOF): Any unique true point for the faulty term
- Operator Reference Fault (ORF):
  - *or* implemented as *and*: Any unique true point of one of the two terms
  - *and* implemented as *or*: any near false point of one of the two terms
- Clause Omission Fault (COF): Any near false point for the faulty term and clause omitted
- Clause Insertion Fault (CIF): *All* near false points and unique true points for the faulty term
- Clause Reference Fault (CRF): *All* near false points and unique true points for the faulty term

Weyuker et al. Study

- TCASII, aircraft collision avoidance system
- 20 predicates/logic functions formed the specifications
- Roughly *6 percent* of the All-Variant test suite (2^n) is needed to meet the *variable negation criteria*
- On average 10 distinct clauses per expression
- Five mutation operators, defined for boolean expressions, we used to seed faults in the specifications
- Random selection of test cases (same size) leads to an average mutation score of 42.7%
- The variable negation strategy is therefore doing much better with an average of 97.9%

E. Weyuker, T. Goradia, and A. Singh, "Automatically generating test data from a Boolean specification," in *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 353-363, May 1994.

Summary of Functional Testing

- All techniques see a program as a mathematical function that maps inputs onto its outputs
- By order of sophistication: (1) boundary value analysis, (2) equivalence class testing, (3) Category-partition (4) Cause-effect graphs
- (1) Mechanical, (2) devise equivalence classes, (3) partitions, categories, and logical dependencies (4) logical dependencies between causes themselves, and causes and effects
- Less test cases with (3) or (4)
- Trade-off between test identification and test execution effort

Black-box testing methods

- ✓ Equivalence Class Partitioning
- ✓ Boundary-Value Analysis
- ✓ Category-Partition
- ✓ Decision tables
- ✓ Cause-Effect Graphs
- ✓ Logic Functions