Cpt S 422: Software Engineering Principles II

Black-box testing – Part 4

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

Logic Functions Testing

Definitions

- A *predicate* is an expression that evaluates to a boolean value

- Predicates may contain boolean variables, non-boolean variables that are compared with the comparator operators {>, <, =, …}, and function calls (return boolean value)

- The internal predicate structure is created by *logical operators* {not, and, or, …}

- A *clause* is a predicate that does not contain any of the logical operators, e.g., (a<b)

- Predicates may be written in different, logically equivalent ways (Boolean algebra)

Definitions (cont.)

- A logic function maps from *n* boolean input variables (clauses) to 1 boolean output variable

- To make expressions easier to read we will use adjacency for the *and* operator, + for the *or* operator, and a ~ for the negation operator.

- Example: Enable or disable the ignition of a boiler based on four input variables
  - NormalPressure (A): pressure within safe operating limit?
  - CallForHeat (B): ambient temperature below set point?
  - DamperShut (C): exhaust duct is closed?
  - ManualMode (D): manual operation selected?

- Logic Function: Z = A(B~C+D)

Boiler Truth Table I

$Z = A(B\sim C + D)$

Input Vector Number	**Normal Pressure**	**CallForHeat**	**DamperShut**	**ManualMode**	**Ignition**
	A	**B**	**C**	**D**	**Z**
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0

Boiler Truth Table II

$Z = A(B{\sim}C+D)$

Input Vector Number	**Normal Pressure**	**CallForHeat**	**DamperShut**	**ManualMode**	**Ignition**
	A	**B**	**C**	**D**	**Z**
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	1

Total of 16 variants!

Elements of Boolean Expressions

- *Boolean space:* The n-dimensional space formed by the input variables
- *Product term or conjunctive clause*: String of clauses related by the *and* operator
- *Sum-of-products or disjunctive normal form (DNF)*: Product terms related by the *or* operator
- *Implicant*: Each term of a sum-of-products expression – sufficient condition to fulfill for *True* output of that expression
- *Prime implicants*: An implicant such that no subset (proper subterm) is also an implicant
- *Logic minimization*: Deriving compact (irredundant) but equivalent boolean expressions, using boolean algebra

Boiler Example

- Logic function: Z=A(B~C+D)

- Sum-of-Product Form (DNF): Z=A(B~C+D) = AB~C+AD

- Implicants: AB~C, AD

- Prime implicant:
  - AB~C = TTFx = {TTFT, TTFF},
  - AD=TxxT={TFFT, TFTT, TTFT, TTTT}
  - Both terms are prime implicants

From Graph to Logic Function

- Once a cause-effect graph is reviewed and considered correct, we want to derive a logic function for the purpose of deriving test requirements (in the form of a decision table)

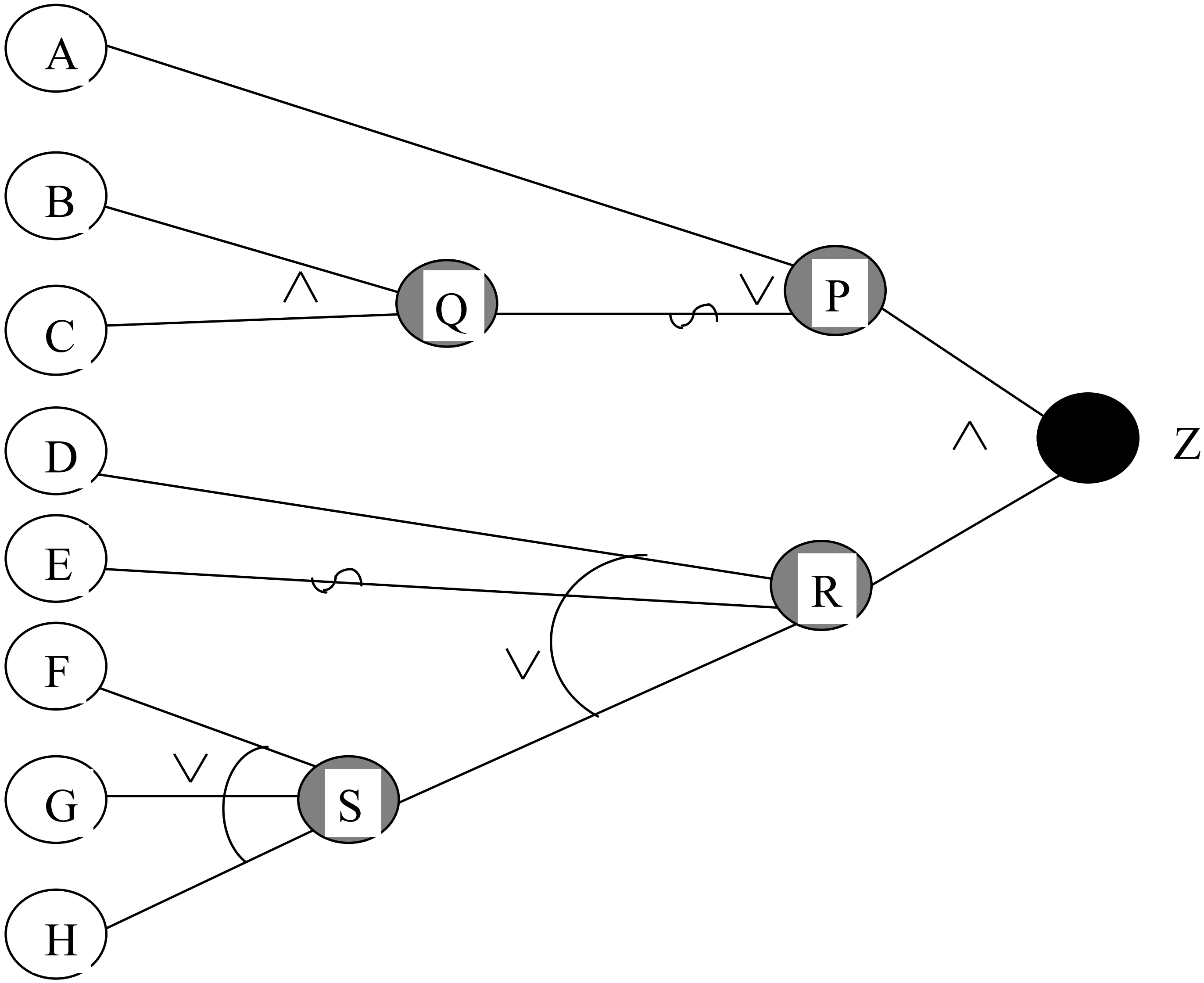
- One function (predicate, truth table) exists for each effect (output variable)

- If several effects are present, then the resulting decision table is a composite of several truth tables that happen to share decision / input variables and actions/effects

- Easier to derive a function for each effect separately

- Derive a boolean function from the graph in a systematic way

Example



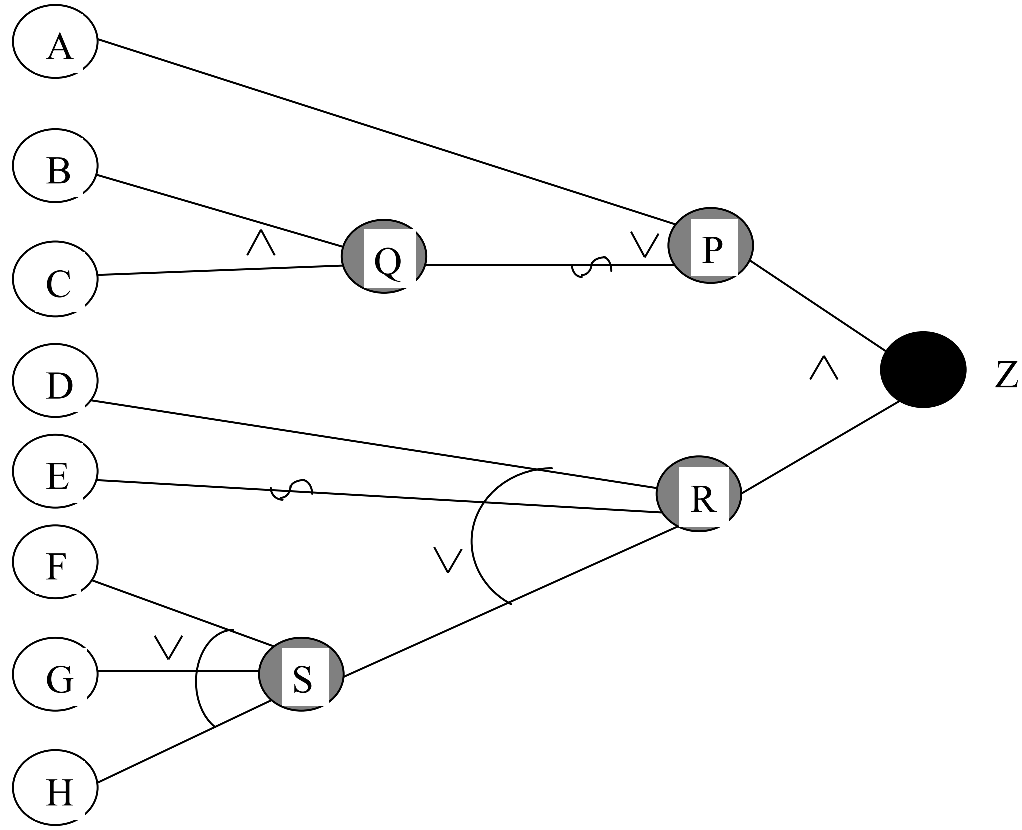
Generate a Logic Function

- Generate an initial function
  - Start from effect node
  - Backtrack through the graph
  - Substitute higher level clauses with lower level clauses and boolean expressions, until you reach cause nodes
- Transform into minimal, DNF form
  - Use boolean algebra laws to reduce boolean expressions
  - Re-express in sum-of-products form (disjunctive normal form)
  - There exist algorithms to do that automatically

Reminder: Laws of Boolean Algebra

- Associative
  - (A+B)+C = A+(B+C), (AB)C = A(BC)
- Distributive
  - A+(BC) = (A+B)(A+C), A(B+C) = AB+AC
- De Morgan's laws
  - ~(A+B)=~A~B, ~(AB)=~A+~B
- Absorption
  - A + AB = A
  - A(A+B) = A
  - A+(~AB) = A+B, A(~A+B) = AB
  - AB+AC+B~C = AC+B~C

Example



- Z = PR (effect)
- P = A + ~Q (intermediate)
- Q = BC (intermediate)
- R = D + ~E + S (intermediate)
- S = F + G + H (intermediate)
- Z = (A + ~(BC)) (D + ~E + (F+G+H)) (substitution)
- Z=(A+~B+~C)(D+~E+F+G+H) (De Morgan's law)
- Z = AD + A~E + AF + AG + AH + ~BD + ~B~E + ~BF + ~BG + ~BH + ~CD + ~C~E + ~CF + ~CG + ~CH (Distributive law is used to obtain sum-of-products)

Fault Model for Logic-based Testing

- Expression Negation Fault (ENF): The logic function is implemented as its negation
- Clause Negation Fault (CNF): A clause in a particular term is replaced by its negation
- Term Omission Fault (TOF): A particular term in the logic function is omitted.
- Operator Reference Fault (ORF): A binary operator *or* in the logic function is implemented as *and* or vice-versa
- Clause Omission Fault (COF): A clause in a particular term of the logic function is omitted
- Clause Insertion Fault (CIF): A clause not appearing in a particular term of a logic function is inserted in that term
- Clause Reference Fault (CRF): A clause in a particular term of a logic function is replaced by another clause not appearing in the term

Basic Test Criteria

- The goal is to test an implementation and make sure it is consistent with its specification, as modeled by the logic function (or graph)

- There exist a number of test coverage criteria that do not assume a disjunctive normal form for predicates:

  - Predicate coverage
  - Clause coverage
  - Combinatorial coverage
  - (in)active clause coverage

- Notation: P is set of predicates, C is set of clauses in P, Cp is the set of clauses in predicate p

Predicate Coverage (PC)

- Predicate Coverage: *For each p $\in$ P, we have two test requirements (TR): p evaluates to true, and p evaluates to false.*

- For p=A(B~C+D) two test that satisfy Predicate Coverage are (1) (A=true, B=false, C=true, D=true) where p evaluates to true, (2) (A=false, B=false, C=true, D=true) where p evaluates to false

- Problem: Individual clauses are not exercised

Clause Coverage (CC)

- Clause Coverage: *For each c $\in$ C, we have two test requirements: c evaluates to true, and c evaluates to false.*

- For p=(A+B)C, two tests that satisfy Clause Coverage: (1) (A=true, B=true, C=false), (2) (A=false, B=false, C=true)

- Note: Clause coverage does not <u>subsume</u> predicate coverage or vice-versa (in the previous example p evaluates to false in both cases)

- <u>Definition</u>: A test criterion C1 subsumes C2 if and only if every test set that satisfies C1 will also satisfy C2.

Example

- Z = A+B
- t1 = (A = true; B = true) => Z
- t2 = (A = true; B = false) => Z
- t3 = (A = false; B = true) => Z
- t4 = (A = false; B = false) => ~Z

- T1 = {t1; t2}: it satisfies neither Clause Coverage (A is never false) nor Predicate Coverage (Z is never false).
- T2 = {t2; t3}: satisfies Clause Coverage, but not Predicate Coverage (Z is never false).
- T3 = {t2; t4}: satisfies Predicate Coverage, but not Clause Coverage (because B is never true).
- T4 = {t1; t4}: is the only pair that satisfies both Clause Coverage and Predicate Coverage.

Combinatorial Coverage (CoC)

- We would certainly like a coverage criterion that tests individual clauses and that also tests the predicate
- **Combinatorial coverage**: *For each $p \in P$, we have test requirements for clauses in Cp to evaluate each possible combination of truth values*
- Subsumes predicate coverage
- There are $2^{|Cp|}$ possible assignments of truth values
- Problem: Impractical for predicates with more than a few clauses

Masking Effects

- When we introduce tests at the clause level, we want to have an effect on the predicate
- Logical expressions (clauses) can mask each others
- In the predicate AB, if B = *false*, **B can be said to mask A**, because no matter what value A has, AB will still be *false*.
- We need to consider circumstances under which a clause affects the value of a predicate, to detect possible implementation failures

Determination

- Determination: *Given a clause c_i in predicate p,* **called the major clause**, *we say that c_i* determines *p if the remaining minor clauses $c_j \in p$, $j <> i$ have values so that changing the truth value of c_i changes the truth value of p.*

- We would like to test each clause under circumstances where it determines the predicate

- Test set T4={t1,t4} in previous slide satisfied both predicate and clause coverage but it neither tests A nor B effectively.
  - Z = A+B
  - t1 = (A = true; B = true) => Z
  - t4 = (A = false; B = false) => ~Z

Active Clause Coverage (ACC)

- Active Clause Coverage: *For each* $p \in P$ *and* ***each major clause*** $\boldsymbol{c_i} \in Cp$*, choose minor clauses* c_j*,* $j <> i$ *so that* c_i *determines p. We have two test requirements for each* c_i*:* $\boldsymbol{c_i}$ ***evaluates to true and*** $\boldsymbol{c_i}$ ***evaluates to false****.*

- For example, for Z=A+B we have:
  - For clause A, A determines Z if and only if B is false: {(A = *true; B =false*), (A = *false; B = false*)}
  - For clause B, B determines Z if and only if A is false: {(A = *false; B = true*), (A = *false; B = false*)}
  - We end up with a total of three tests: {(A = *true; B =false*), (A = *false; B = false*), (A = *false; B = true*)}

- The most important questions are whether (1) ACC should subsume PC, (2) the minor clauses c_j need to have the same values when the major clause c_i is true as when c_i is false.

Correlated ACC (CACC)

- *For each $p \in P$ and* ***each major clause c_i*** *$\in Cp$,* ***choose minor clauses c_j****, $j <> i$ so that c_i determines p. There are two test requirements for each c_i:* ***c_i evaluates to true and c_i evaluates to false****. The values chosen for the* ***minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other****, that is, it is required that* ***$p(c_i = true) <> p(c_i = false)$****.*

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

CACC - Example

- P = A(B+C)
- What does it take so that A determines the value of P?

A	B	C	A(B+C)
T	T	T	T
T	**T**	**F**	**T**
T	F	T	T
F	T	T	F
F	T	F	F
F	**F**	**T**	**F**

- We satisfy CACC for A with for example:
  - T1 = {(A = true, B = true, C = false), (A = false, B = false, C = true)}

Restricted ACC (RACC)

- For each $p \in P$ and **each major clause c_i** $\in Cp$, choose minor clauses c_j, j <> i so that c_i determines p. There are two test requirements for each c_i : **c_i evaluates to true and c_i evaluates to false**. The values chosen for the **minor clauses c_j must be the same when c_i is true as when c_i is false**, that is, it is required that $c_j(c_i = true) = c_j(c_i = false)$ for all c_j .

- RACC makes it easier than CACC to determine the cause of the problem, if one is detected: major clause

- But is it common in specification to have constraints between clauses, making RACC impossible to achieve in some cases.

RACC - Example

- P=A(B+C)
- We satisfy RACC for A with for example:

A	B	C	A(B+C)
T	T	T	T
F	T	T	F

- But also with:

A	B	C	A(B+C)
T	T	F	T
F	T	F	F

- But also with:

A	B	C	A(B+C)
T	F	T	T
F	F	T	F