**Formal Method**

Myths of formal methods

1. can guarantee that software is perfect.
2. are all about program proving.
3. are only useful for safety-critical systems.
4. require highly trained mathematicians.
5. increase the cost of development.
6. are unacceptable to users.
7. are not used on real, large-scale software.

used in

• architecture (AADL)

* Architectural Analysis and Design Language
* Used to model the software and hardware

architecture of an embedded, real-time system

• large-scale systems (CICS)

- Customer Information Control System

- IBM’s CICS used Z and:

2.5 times fewer customer-reported errors;

- 9% saving in the total development costs of the release

• non-life-critical systems (CICS)

**GUI testing**

**Security testing**

**Predicate Analysis (test generation)**

Definitions

Disjunctive Normal Form – sum of product terms, can covert Boolean between

Conjunctive Normal Form- product of sums, can covert Boolean between

Infeasible constraint – no value can satisfy predicate

Singular – each variable only occurs once in Boolean expression

Mutually Singular – expressions that don’t share variables

BOR – singular predicates

BRO – singular predicates

MI test gen – nonsingular DNF predicates

BOR-MI test gen – smaller testes, more powerful than MI

**Finite State Machines (test generation)**

Finite state machines

• Minimality

- number of states in M is less than or equal to any other

FSM equivalent to M.

• fully specified

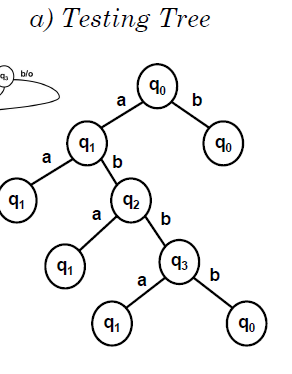
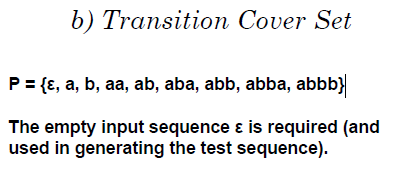
* If from each state in M there exists a transition for each input symbol.

• strongly connected

Characterization sets (W)

* finite set of input sequences that
  + distinguish the behavior of any pair of states in M.
  + each input sequence in W is of finite length

Testing trees

 - > ->

Transition cover sets

Definitions

State- description of status of system

Transition- set of actions to be executed when a condition is fulfilled.

**Predicate Analysis (test generation)**

**Control Flow (Test Adequacy Assessment and Enhancement)**

Statement, block, decision, condition coverage

Coupled conditions

Short circuit evaluation

Condition/decision (branch condition) coverage

Multiple condition coverage

LCSAJ coverage

MC/DC coverage

**Data Flow (Test Adequacy Assessment and Enhancement)**

Definitions and usages

Computational uses (c-uses), predicate uses (p-uses)

Data flow graphs

Def clear paths

Def-use pairs (dcu, dpu)

All-uses coverage

Subsumes relationship

**Mutation (Test Adequacy Assessment and Enhancement)**

Mutants, equivalent mutants

Distinguishing / killing mutants

Strong vs weak mutations

Distinguishing a mutant

- reachability, state infection, state propagation

Competent programmer hypothesis

Coupling effect

Traditional mutation operators (Fortran: 22)

**Regression Testing (Test Phases)**

Test Selection

Minimization

Prioritization

Test selection for regression testing can be done

using any of the following methods:

1. only the modification traversing tests
   1. based on CFGs
2. tests using execution slices
   1. based on execution traces
3. tests using dynamic slices
   1. based on execution traces and dynamic slices
4. tests using code coverage
   1. based on the coverage of testable entities
5. tests using a combination of code coverage and human judgment
   1. based on amount of the coverage of testable entities

**Unit Testing (Test Phases)**

**Integration Testing (Test Phases)**

**Problem Types**

**Predicate Based Test Generation**

Cross Product

Let A = {t, =, >} and B = {f, <}

A x B = {(t, f), (t, <), (=, f), (=, <), (>,f), (>,<)}

Onto product

Let A = {t, =, >} and B = {f, <}

A ⨷ B = {(t, f), (=,<), (>,<)}

Conventions

1. order {(t), (f)}, {(<), (=), (>)}, {(-ε), (=), (+ ε)} in initial sets
2. match corresponding ONTO terms until reaching the end of the shorter set; then continue matching with the last item in the shorter set
3. pick the first item for a {tx} or {fx}

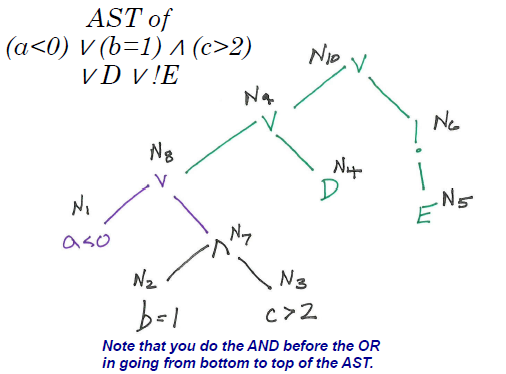
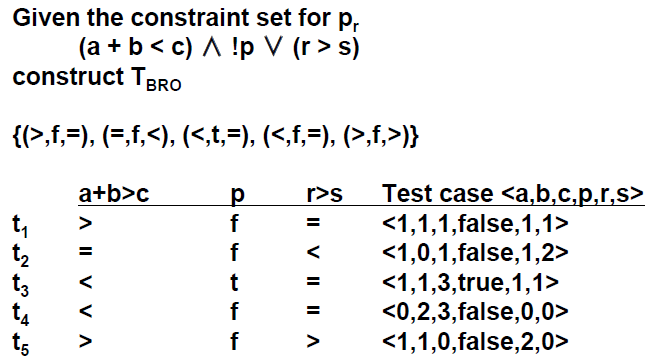
Node Type

1. AND
   1. True (Onto product of true conditions from each node)
      1. SN3t = SN1t ⨷ SN2t = {(t)} ⨷ {(t)} = {(t,t)}
   2. False
      1. ( Cross product for one true and one false, Union with mirror false and true cross product)
      2. SN3f = (SN1f x {t2}) U ({t1} x SN2f)
2. OR
   1. True
   2. False (Onto product of false conditions from each node)

BOR – Boolean operator

1. Label each leaf node N of AST(pr) with its constraint set SN. For each SN = {t, f}.
2. Visit each non-leaf node in AST(pr) in a bottom-up manner.
   * Let N1 and N2 denote the direct descendants of node N, if N is an AND-node or OR-node.
   * If N is a NOT-node, then N1 is its direct descendant.
   * SN1 and SN2 are the BOR-constraint sets fornodes N1 and N2 respectively.
   * For each non-leaf node N, compute SN asfollows.
3. The constraint set for the root of AST(pr) is the desired BOR-constraint set for pr.
4. End of procedure BOR-CSET

BRO – Boolean and Relational Operator

1. Draw AST
2. Label nodes, Sn1t and Sn1f, put the operators that correspond to the state in brackets (ie Sn1t = {<} and Sn1f = {=, >})
3. Traverse tree and computer constraints set for each internal node using AND / OR rules
4. Compute root node and construct TBRO with Test Cases
5. 

MI

BOR-MI

1. Express in DNF form (sum of products, means distribute as much as possible)
   * a(bc + !bd)
   * E = abc + a!bd
   * E = e1 + e2
2. Step 1.
   * Construct true constratiants for each ei
   * Te1 = {(t,t,t,t), (t,t,t,f)}
   * Te2 = {(t,f,t,t), (t,f,f,t)}
3. Step 2.
   * + - From each Tei , remove the constraints that are in any other Tej
       - Note
       - There are no common constraints between Te1and Te2 in our example
4. Step 3.
   * Construct SEt by selecting one elementfrom each TSei
   * Note there are four possible choices from Te1 Te2
   * SEt = {(t,t,t,f), (t,f,f,t)}
5. Step 4.
   * For each term in E, obtain terms by complementing each literal, one at a time.
   * e1 1 = !abc e1 2 = a!bc e1 3 = ab!c
   * e2 1 = !a!bd e2 2 = abd e2 3 = a!b!d
   * From each term e above, derive constraints Fe that make e true
   * Fe1 1 = {(f,t,t,t), (f,t,t,f)} !abc
   * Fe1 2 = {(t,f,t,t), (t,f,t,f)} a!bc
   * Fe1 3 = {(t,t,f,t), (t,t,f,f)} ab!c
   * Fe2 1 = {(f,f,t,t), (f,f,f,t)} !a!bd
   * Fe2 2 = {(t,t,t,t), (t,t,f,t)} abd
   * Fe2 3 = {(t,f,t,f), (t,f,f,f)} a!b!d
6. Step 5.
   * Construct FSe by removing from Fe any constraint that appeared in any of the two sets Te constructed earlier
   * FSe1 1 = Fe1 1 = {(f,t,t,t), (f,t,t,f)}
   * FSe1 2 = {(t,f,t,f)} {(t,f,t,t), (t,f,t,f)}
   * FSe1 3 = Fe1 3 = {(t,t,f,t), (t,t,f,f)}
   * FSe2 1 = Fe2 1 = {(f,f,t,t), (f,f,f,t)}
   * FSe2 2 = {(t,t,f,t)} {(t,t,t,t), (t,t,f,t)}
   * FSe2 3 = Fe2 3 = {(t,f,t,f), (t,f,f,f)}
   * Note, compare Step 1 to Step 4
   * Remove anything from Step 4 that was in Step 1
7. Step 6.
   * Construct SEf by selecting one constraint from each FSe
   * Note
   * If you choose a constraint that is duplicated, only one is needed
8. Step 7.
   * Union all your choices
9. End of procedure MI-CSET

BRE – Boolean and Relational Expression

**FSM**

Testing tree

Transition Cover Set

**Control Flow**

statement coverage

block coverage

BT = |Bc| / (|Be| – |Bi|)

|Bc| is the number of blocks covered

|Be| is the total number of blocks in the program

|Bi| is the number of unreachable blocks

decision coverage

If/ while = 1 decision

Compound if = 1 decision

Might not cover both sides

A decision can have three possible outcomes, true, false, and undefined.

Dt = |Dc| / (|De| – |Di|)

|Dc| is the number of decisions covered

|De| is the total number of decisions in the program

|Di| is the number of infeasible decisions

condition coverage

Both true and false side covered

Ct = |Cc| / (|Ce| – |Ci|)

|Cc| is the number of simple conditions covered

|Ce| is the total number of simple conditions in the program

|Ci| is the number of infeasible simple conditions

LCSAJ, Linear Code Sequence and Jump

Execution of sequential programs that contain at least one condition proceeds in pairs where:

1. the first element of the pair is a sequence of statements (a block) executed one after the other
2. terminated by a jump to the next such pair(another block)

**Data flow**

Graph

dcu/dpu table