Objectives

- Complete Section 2.2 – Specification-Based Structural Testing
- Cover section 3 of this paper (Fault-Based Adequacy Criteria)
- To give specific examples to prepare student for hands-on exercises

Lecture

1 Specification-Based Structural Testing

- Specification defines properties software must satisfy.
- Model based specification: state space (attributes) + behavior (operators)
- Property-based specification: axiomatic or algebraic specifications
- Specification plays two roles in testing
  - Oracle problem: defines correct output
  - Test Generation Algorithm

1.1 Decision Table based Testing

- Decision table is a specification
- Rule for generation of functional test cases
  - Based on feasible combinations of predicates
  - Result defined in terms of combinations of effects (actions)
- Same rule for evaluating adequacy of test set

1.2 Pre-/Post-Condition Specification

- See Sample A – Simple Account class
- Coverage – want post-conditions to be satisfied and feasible combinations of pre-conditions to be exercised
- Pre-Post conditions are a form a decision table (think about that)

1.3 Benefit of Specifications in Testing

- Formality – leads to precise statements (basis in mathematics)
- Formality provides basis for automation: generation and checking

1.4 Category-Partition Method for Specifying and Generating Functional Tests

• **READ THE PAPER.**
  • For each parameter, identify categories of test data
  • Subdivide categories into choices – at min \{INVALID, VALID\}
  • Subdivide choices
  • Example of domain analysis – divide-and-conquer approach.

### 1.5 Combining Specification-based and Program-Based Structural Testing

• Example: Decision table spec vs actual code
• Both provide a slightly different angle
• Best to combine them
• Recall the deficiency of white-box testing
• Recall the minimal test set of functional testing

### 2 Fault-Based Adequacy Criteria.

• How good is the test set at finding faults?
• Because the number of faults is always unknown, how can you estimate the fault-detection power of the test set?

### 1.6 Error Seeding

• Inject known errors
• Based on how many are found, project the fault-detecting power (efficiency) of the test set
• Efficiency = (#seeded faults found)/(#seeded faults)
• Limitation: Are seeded faults comparable to "natural" faults?

### 1.7 Mutation Testing

• A mutant is a program that differs from the original by a small amount
  • **Competent Programmer Assumption:** the mutation represents a small error made by a competent programmer, i.e., the mutant is close to being correct.
  • **Coupling Effect Assumption:** complex errors are related to simple errors. Complex errors are detectable by the detection of simple errors.
• Mutation Operators
  • **Read Paper:** Offut, Lee, Rothermel, et. al, "An Experimental Determination of Sufficient Mutation Operators" (handout)
  • Paper identifies a small number of operators to increase efficiency of mutation testing without sacrificing effectiveness.
• Mutation testing process
  • Create set of mutants
  • Kill as many as possible with the test set
  • Add test cases to kill additional mutants
  • Resulting test set is better

• Mutation Adequacy Score: \( S = \frac{\text{#dead mutants}}{[\text{#mutants} - (\text{#equivalent mutants})]} \)
• Efficiency of Mutation Testing
  • Undecidable problem: are two mutants equivalent
  • Large number of mutants
  • Mutation operators – can reduce number from 22 down to 5

• Test set must result in paths that contain a use of every definition
  Node = sequential statement terminated by branch
• Weak criterion
• See example A

1.8 Mutation Testing Applied to Specifications
• Consider mutations to a decision table specification
• Generate new test cases based on mutant decision table
• This test set reflects behavior that would have been implemented if the original decision table was misinterpreted by the programmer
• What operators make sense?

• Consider pre-/post-condition specifications also.
Example A – Pre-/Post-Condition Specification

Example Code A – Bank Account Class

SPECIFICATION for SIMPLE Bank Account Class

Purpose: The SimpleAcct class represents a single account holder at a bank. These user transactions are recognized: OpenAccount, Deposit, Withdraw(check), BalanceCheck, CloseAccount. A NSF exception transaction is triggered when there are non-sufficient funds (inadequate balance) when a Withdraw transaction is attempted. This class maintains the current account balance reflecting Deposits, Checks, and NSF charges; it also keeps a separate count and total of all Deposit, Check and NSF transactions.

An administrative transaction that occurs once per period (e.g., month) is PrintStatement: the counts and totals for Deposits, Withdraws and NSFs are written to an output file. The report "Bank Statement" also contains counts for the CheckBal transactions, average deposit and withdraw amounts, and the beginning and ending balances.

When the account is created, its account ID is initialized to 0000. When it is opened the account ID, the name of the account holder, and the initial balance, are all captured, and all counts and totals are assigned initial values zero.

Retained (private) data:

AcctID          -- 4 digit account code.
AcctHolder      -- 20-character string with name of account owner.
IsOpen          -- flag indicating account has been opened.
NSF_Fee         -- (constant $20.00) service charge for each NSF event
NSF_Event       -- flag indicating that an NSF situation is being handled.
CurBal          -- current balance (excluding pending transactions)
StartBal        -- beginning balance for period.
Deposit_Count   -- #deposits for billing period.
Withdraw_Count  -- #withdrawals for billing period.
BalCheck_Count  -- #balance checks for billing period.
NSF_Count       -- #NSFs for billing period.
Deposit_Total   -- Total amount of deposits for billing period.
Withdraw_Total  -- Total amount of withdrawals for billing period.
NSF_Total       -- Total charges for NSFs for billing period.
StmtPrinted     -- flag indicating the bank statement has been printed.

Operations (public), with semantics.

Open_Acct (accountnumber, acctowner, initbalance) -- open specified account for account owner with an initial balance.

precondition: IsOpen = FALSE
postcondition: (new CurBal) = initbalance
             (new StartBal) = initbalance
             (new AcctID) = accountnumber
             (new AcctHolder) = acctowner
             {{(new *_Count) = 0}
             {{(new *_Total) = 0}
             (new IsOpen) = true
             (new NSF_Event) = false
             (new StmtPrinted) = true

Close_Acct () -- close account.
Deposit (Amount) -- record deposit of given amount.
precondition: IsOpen = TRUE
postcondition: CurBalance = Amount + (old CurBalance)  
(new Deposit_Total) = (old Deposit_Total) + Amount  
(new Deposit_Count) = (old Deposit_Count) + 1

Withdraw (Amount) -- record withdrawal of given amount.
precondition: IsOpen = TRUE  
CurBal >= Amount
postcondition:  
precondition -->
[  
(new CurBal) = (old CurBal) - Amount  
(new Withdraw_Total) = (old Withdraw_Total) + Amount  
(new Withdraw_Count) = (old Withdraw_Count) + 1  
]
IsOpen & NOT CurBal >= Amount -->
[  
(new CurBal) = (old CurBal) - NSF_Fee  
(new NSF_Total) = (old NSF_Total) + NSF_Fee  
(new NSF_Count) = (old NSF_Count) + 1  
(new NSF_Event) = TRUE  
]

BalanceCheck (ostream) -- Display current balance in output stream.
precondition: IsOpen = TRUE
postcondition: unchanged (*)
(new BalCheck_Count) = (old BalCheck_Count) + 1

MonthlyStatement () -- displays all transactions for the billing period.
Statement is generated after all daily transactions have been posted.
precondition: IsOpen = TRUE
postcondition: (new StmtPrinted) = TRUE

EndOfMonth () -- Clears all totals and counts to zero, in order to start a
new month of transactions.
precondition: IsOpen = TRUE  
StmtPrinted = TRUE
postcondition: unchanged (CurBalance)
(new *_Count) = 0  
(new *_Total) = 0  
(new StmtPrinted) = FALSE  
(new NSF_Event) = FALSE

Private methods:

Charge_NSF (checkamount) -- post a non-sufficient funds
transaction against the account.
precondition: NSF_Event
postcondition: (new CurBal) = (old CurBal) - NSF_Fee
(new NSF_Count) = 1 + (old NSF_Count)
(new NSF_Total) = NSF_Fee + (old NSF_Total)

Log_NSF (checkamount) -- write the NSF transaction to
the master NSF log for the bank:

precondition: NSF_Event = TRUE
postcondition: (new NSF_Log) = (old NSF_Log) +
{(AcctID, checkamount, NSF_Fee)}
(new NSF_Event) = FALSE

=======================================================================