Testing

Insup Lee
Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
www.cis.upenn.edu/~lee/

Originally prepared by Eunkyoung Jee.

CIS 541, Spring 2010

What is Testing?

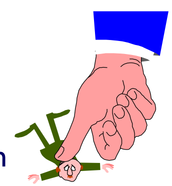
> *Software Testing is the process of executing a program or system with the intent of finding errors*
>
> G. Myers

- A successful test is one that finds an error
- Testing can
  - Identify discrepancies between actual results and expected behavior
  - Demonstrate functions are working according to specification
  - Provide an indication of correctness, reliability, safety, security, performance, fault tolerance, usability, ...

What Makes Testing So Difficult?

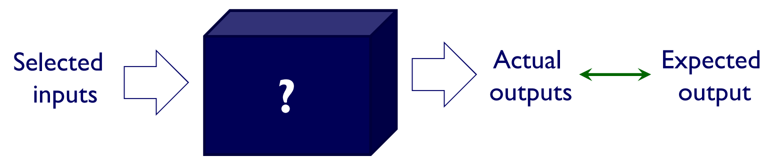
- Inherent complexity of software
- Constructing an operational environment for testing purpose
- Intractable and undecidable nature of testing
- Idiosyncrasy of software
  - Trivial clerical errors can have major consequences.
  - Errors manifest themselves in rare states, yet crucially important → *Murphy's Law*

Some Testing Principles



- A programmer should not test his/her own program
- One should test not only that the program does what it is supposed to do, but that it does not do what it is not supposed to
- The goal of testing is to find errors, not to show that the program is errorless
- No amount of testing can guarantee error-free program
- Parts of programs where a lot of errors have already been found are a good place to look for more errors
- The goal is not to humiliate the programmer!

Black-Box Testing (Functional Testing)



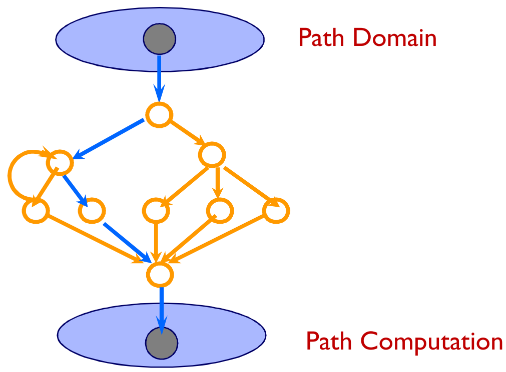
- Is not based on the structure of the program (which is unknown)
- Test cases are selected based on functional specification
- Question: Exhaustive input test?

Examples of Exhaustive Input Test

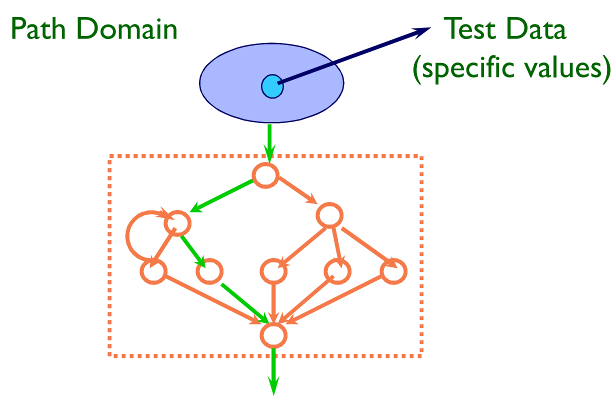
- Solution to $ax^2 + bx + c = 0$
- Input to a compiler
  - All possible valid and invalid programs
- Testing of OS, DBMS, reservation systems ….
  - All possible sequences of transactions

Tester's View of a Program

- Program = {Path Domain, Path Computation}

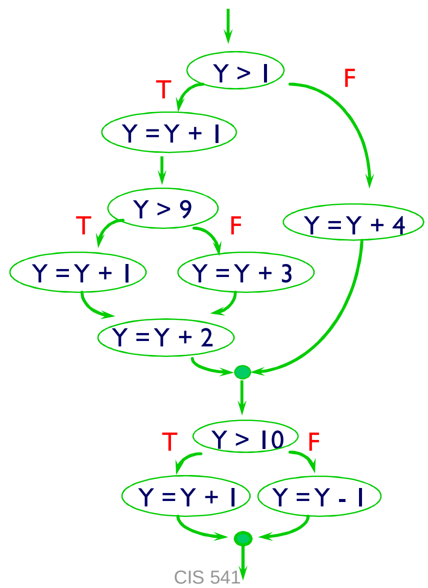


Test Data



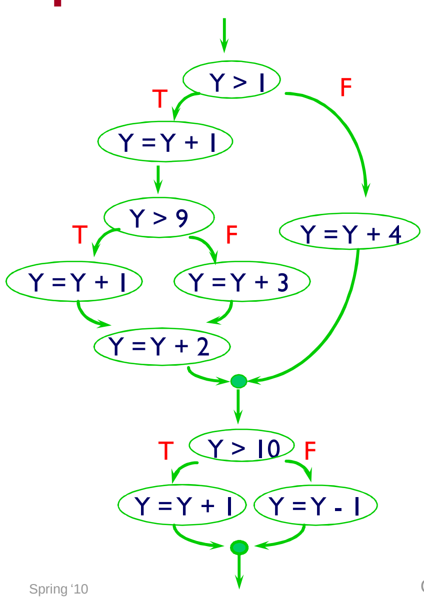
Execution Paths in a Program

```
IfY > 1 THEN
    Y =Y + 1
    IF Y > 9 THEN
        Y =Y + 1
    ELSE
        Y =Y + 3
    END
    Y =Y + 2
ELSE
    Y =Y + 4
END
IF Y > 10 THEN
    Y =Y + 1
ELSE
    Y =Y - 1
END
```



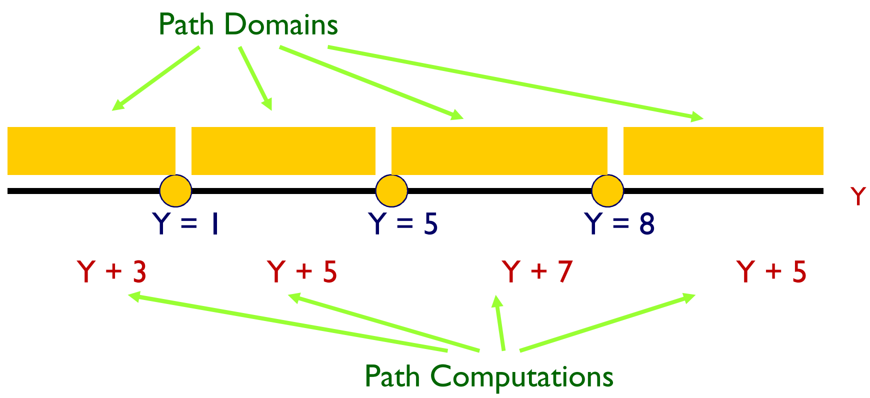
Program Path
T T T
T T F
T F T
T F F
F - T
F - F

Input Data for Executing Paths

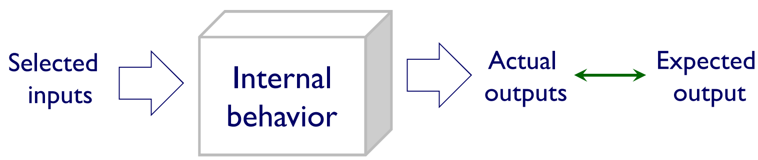


Program Path	Path Domain
T T T	Y > 8
T T F	Infeasible path
T F T	5 <= Y <= 8
T F F	1 < Y < 5
F - T	Infeasible path
F - F	Y <= 1

Path Domains and Computations

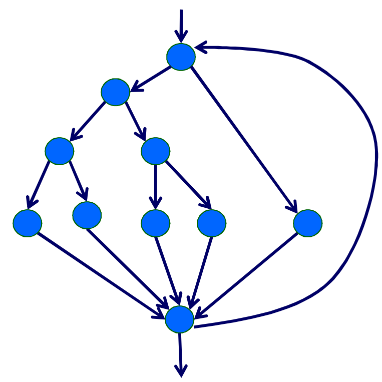


White-Box Testing (Structural Testing)



- Test cases are selected based on software structure/implementation
- There are several alternative criterions for checking "enough" paths in the program
- Question: Exhaustive path test?

Exhaustive Path Test



 # loops ≤ 6

#Paths: $5^1 + 5^2 + 5^3 + 5^4 + 5^5 + 5^6 \cong 20000$

Limitation of Testing

> *Testing can be a very effective way to show the presence of errors, but it is hopelessly inadequate for showing their absence.*

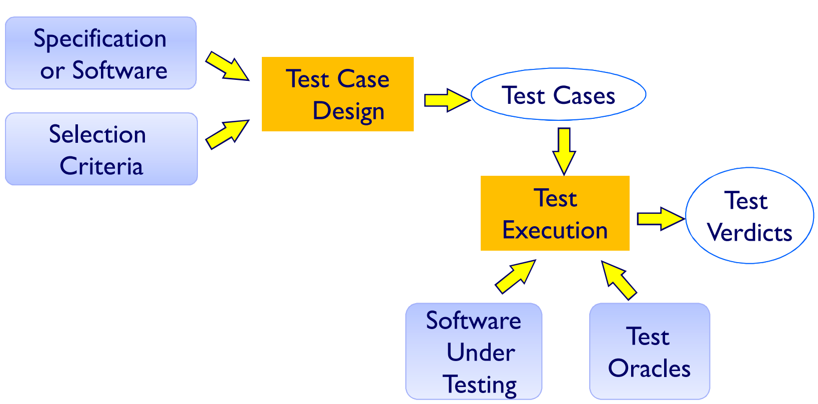
E. Dijkstra

- There are never sufficiently many test cases.
- Testing does not find all the errors.
- Testing is hard and takes a lot of time.
- Testing is still a largely informal task.

Strategic Approach to Testing

- An exhaustive testing is impractical
  - ➔ Relate the amount of testing to confidence about software
- A test of any program is necessarily incomplete
  - ➔ Test Coverage
- No way of knowing if the error detected is the last remaining error
  - ➔ Test Completion

Testing Process



Test Oracle

- A mechanism to produce the predicted outcomes to compare with the actual outcomes of the software under test
- "Any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object" -W. E. Howden
- As more software becomes standardized, more oracles will emerge as products and services.

Test Adequacy Criteria

- "A set of rules used to determine whether or not testing can be terminated" - Weyuker
- Represent a minimal standard for testing a program.
  - "The notion of adequacy is dependent on the method used for selecting the test set" - Zweben
    - Program-based criteria involve program's structure.
    - Specification-based criteria rely on specification.
    - Other criteria (e. g., Random testing) may ignore both specification and program.

TEST CASE DESIGN

Test Case Design

- Construct and organize tests to get the best testing effect with the least effort
  - Testing process is as good as its test cases
- A good test case is one that has a high probability of detecting as-yet undiscovered error

Typical Test Case Information

- Test Case ID - a unique identifier
- Purpose
  - a list of actions (functions, processes, services, etc.) that this test case will exercise
- Input
  - Preconditions
  - Inputs – a list of names and values for inputs to actions that this test case will exercise
- Output
  - Expected Outputs - a list of outputs that will result when this test case exercises actions
  - Post-conditions
- Execution History

Test Case Design for Black Box Testing

- Equivalence partition
- Boundary value analysis
- Cause-effect graphs

Equivalence Class

- A set of test cases such that any member of the class is as good a test case as any other
- For all input values in a particular equivalence class, the system shows the same kind of behavior
  - valid and invalid equivalent classes
  - input and also output domains

Equivalence Class Testing

- Divide the program's input space into domains such that all inputs within a domain are equivalent
- Identify test cases by using one element from each equivalence class
  - Testing with more inputs from the same class hardly increases the chance of finding defects
  - All inputs from the same equivalence class have an equal chance of finding a defect
- Most test techniques, functional or structural, fall under partition testing

Equivalence Partition

- Goals:
  - Find a small number of test cases
  - Cover as much possibilities as you can
- Try to group together inputs for which the program would likely to behave the same

Specification condition	Valid equivalence class	Invalid equivalence class

Example: A Legal Variable

- Begins with A-Z
- Contains [A-Z0-9]
- Has 1-6 characters

Specification condition	Valid equivalence class	Invalid equivalence class
Starting char	Starts A-Z 1	Starts other 2
Chars	[A-Z0-9] 3	Has others 4
Length	1-6 chars 5	0 chars, >6 chars 6 7

Equivalence Partition (cont.)

- Add a new test case until all valid equivalence classes have been covered
  - A test case can cover multiple such classes
- Add a new test case until all invalid equivalence class have been covered
  - Each test case can cover only one such class

Specification condition	Valid equivalence class	Invalid equivalence class

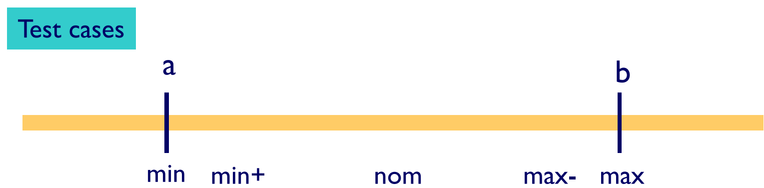
Example: A Legal Variable (cont.)

- AB36P (1,3,5)
- 1XY12 (2)
- A17#%X (4)
- (6)
- VERYLONG (7)

Specification condition	Valid equivalence class	Invalid equivalence class
Starting char	Starts A-Z 1	Starts other 2
Chars	[A-Z0-9] 3	Has others 4
Length	1-6 chars 5	0 chars, >6 chars 6 7

Boundary Value Analysis

- If an input condition specifies a range bounded by values a (min) and b (max), test cases should be designed with values at the minimum, just above the minimum, a nominal value, just below the maximum, and at the maximum

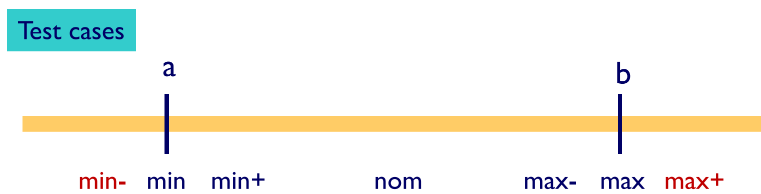


Example of Boundary Value Analysis

- If input is within range -1.0 ~ +1.0, select values -1.001, -1.0, -0.999, 0.999, 1.0, 1.001
- If needs to read N data elements, check with N-1, N, N+1. Also, check with N=0.

Robustness Testing

- A simple extension of boundary analysis
- It forces attention on exception handling.
  - worst case analysis



Test Case Design for White-Box Testing

- The main problem is to select a good coverage criterion
- Some options are:
  - Cover all paths of the program
  - Execute every statement at least once
  - Each decision has a true or false value at least once
  - Each condition is taking each truth value at least once
  - Check all possible combinations of conditions in each decision

How to Cover the Executions?

```
IF (A>1)&(B=0)
    THEN X=X/A;
    END;
IF (A=2)|(X>1)
    THEN X=X+1;
    END;
```

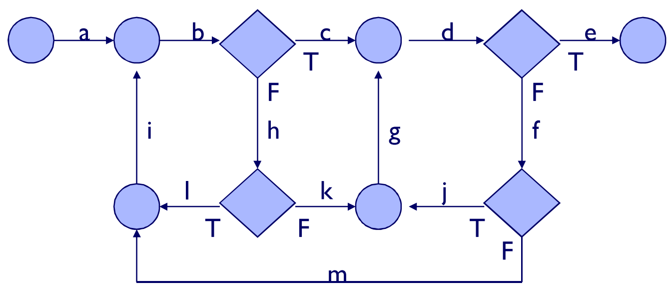
- Choose values for A, B, X
- Value of X may change, depending on A and B
- What do we want to cover? Paths? Statements? Conditions?

Control Flow Testing

- A family of test strategies based on selecting paths through the program's control structure
- If the set of paths is properly chosen, some measure of test thoroughness can be achieved
- Requires complete knowledge of the programs structure
- Most applicable to unit testing

An Example of Path Selection

- What is the fewest number of paths that will cover all statements and branches?



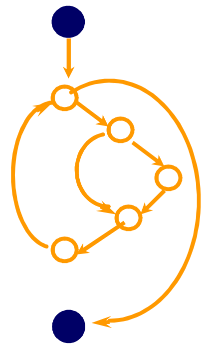
Statement Coverage

- Execute every statement at least once
- Minimum testing requirement in the IEEE unit test standard
- By choosing **A=2,B=0, and X=3**, each statement will be executed
- The case that the tests fail is not checked!

```
IF (A>1)&(B=0)
      THEN X=X/A;
      END;
IF (A=2)|(X>1)
      THEN X=X+1;
      END;
```

Statement Testing: An Example

```
    main()
    {
        int x, y, z;
1)      z = 1;
2)      while( y  != 0 ) {
3)          if ( y%2  != 0 )  z = z * x;
4)          y = y/2 ;
5)          x = x * x;
        }
6)      printf("x**y =%5d", z);
    }
```



Statement Testing: An Example

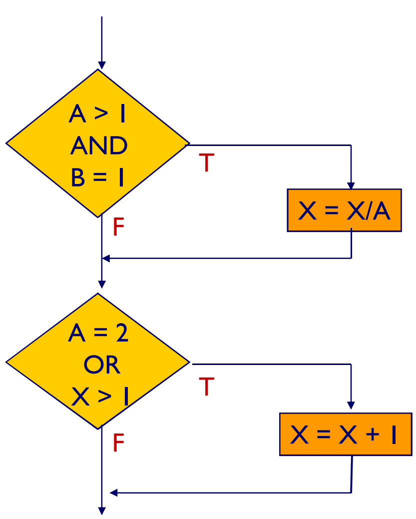
Inputs		Statement					
x	y	1	2	3	4	5	6
5	0	O	O	X	X	X	O
5	2	O	O	X	O	O	O
5	3	O	O	O	O	O	O

Branch (Decision) Coverage

- Each *branch*(*decision*) has a true and false outcome at least once
- Can be achieved using
  - **A=3,B=0,X=3**
  - **A=2,B=1,X=1**
- Problem: does not test individual conditions.
  - E.g., when **X>1** is erroneous in second decision

```
IF (A>1)&(B=0)
        THEN X=X/A;
        END;
IF (A=2)|(X>1)
        THEN X=X+1;
        END;
```


Branch Testing: An Example



Test Case 1 (Path TT)

T:A > 1 and B = 1
(A = 2, B = 1, any X)
T:A = 2 or X > 1 (true)

Test Case 2 (Path FF)

F:A > 1 and B = 1
(A = 1, X = 1)
F:A = 2 or X > 1 (false)

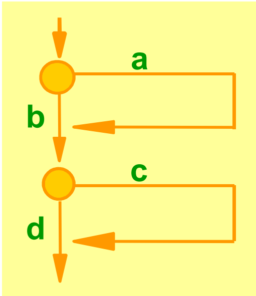
Test Case 3 (Path TF)

T:A > 1 and B = 1
(A = 2, B = 1, X = 1)
F:A = 2 or X > 1 (false)

Test Case 4 (Path FT)

F:A > 1 and B = 1
(A = 1, X = 2)
T:A = 2 or X > 1 (true)

Limitation of Branch Testing



Either paths bd & ac or paths ad & bc will cover all the decisions

Condition Coverage

- Each *condition* has a true and false value at least once
- For example:
  - **A=1,B=0,X=3**
  - **A=2,B=1,X=0**

  lets each condition be true and false once.
- Problem: covers only the path where the first test fails and the second succeeds

```
IF (A>1)&(B=0)
    THEN X=X/A;
    END;
IF (A=2)|(X>1)
    THEN X=X+1;
    END;
```

Multiple Condition Coverage

- Test all combinations of all conditions in each test
- Cases:
  - **A>1,B=0**
  - **A>1,B≠0**
  - **A<=1,B=0**
  - **A<=1,B≠0**
  - **A=2,X>1**
  - **A=2,X<=1**
  - **A≠2,X>1**
  - **A≠2,X<=1**

```
IF (A>1)&(B=0)
     THEN X=X/A;
     END;
IF (A=2)|(X>1)
     THEN X=X+1;
     END;
```


A Smaller Number of Cases:

- **Example:**
  - **A=2,B=0,X=4**
  - **A=2,B=1,X=1**
  - **A=1,B=0,X=2**
  - **A=1,B=1,X=1**
  - Note the X=4 in the first case: it is due to the fact that X changes before being used

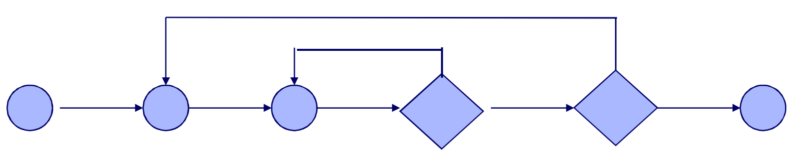
```
IF (A>1)&(B=0)
     THEN X=X/A;
     END;
IF (A=2)|(X>1)
     THEN X=X+1;
     END;
```

Loop Testing

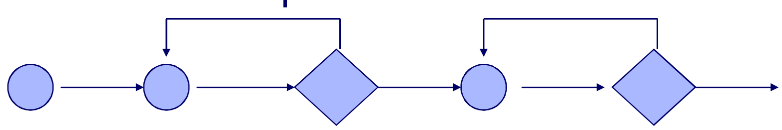
- Simple loops
- Nested loops
- Concatenated loops
- Unstructured loops
  - Redesign the loops

Loop Testing

- A nested loop



- A concatenated loop



Simple loop with maximum n number of passes

- Skip the loop entirely
- Only one pass through the loop
- m passes through the loop where m < n
- n passes through the loop
- If possible, try n – 1 and n + 1 passes through the loop

Nested loops

- Start at the inner most loop, set all other loops to minimum number of iterations
- Conduct simple loop tests for the innermost loop while holding outer loops at their minimum number of iterations
- Work outward, conducting tests for the next loop, keeping all outer loops at minimum iterations and keeping nested loops to "typical" values
- Continue until all loops have been tested

Concatenated loops

- If each of the loops is independent of the other, use tests for simple loops
- If loops are not independent, use tests for nested loops

Data Flow Testing

- A family of test strategies based on selecting paths through the program's control flow in order to explore sequence of events related to the status of data objects
- Select test paths of a program according to locations of definitions and uses of variables in the program

Data Usage

- d - defined, created, initialized
- k – killed, undefined, released
- u – used for something
  - Used in computation (c-use)
  - Used in a predicate (p-use)

Data-Flow Anomalies

- dd – probably harmless but suspicious
- dk – probably a bug
- du – a normal case
- kd – normal situation
- kk – harmless but probably buggy
- ku – a bug
- ud – usually not a bug
- uk – normal situation
- uu – normal situation

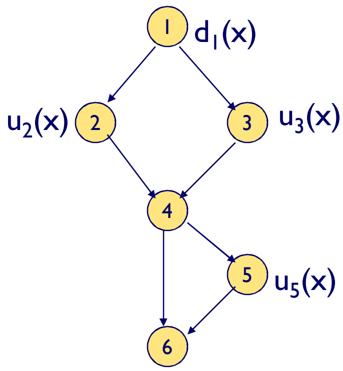
Definition-Use Chain

- DEF(S) = {X | statement S contains a definition of X}
- USE(S) = {X | statement S contains a use of X}
  - DU chain = [X, S, S']
  - $X \in DEF(S)$, $X \in USE(S')$

Data Flow Strategies

- All Definition-Use Paths
- All-Uses paths
- All-p-Uses/Some-c-Uses
- All-c-Uses/Some-p-Uses
- All Definitions
- All-Predicate Uses
- All Computation Uses

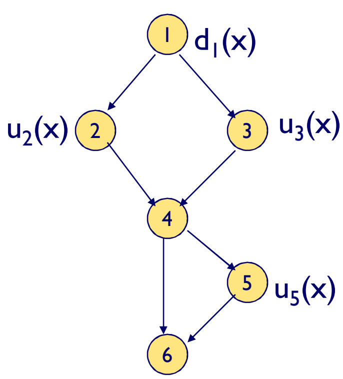
Data Flow Test: An Example (1)



All-Defs
- Requires:
  - $d_1(x)$ to a use
- Satisfactory Path:
  - 1, 2, 4, 6

All-Uses
- Requires:
  - $d_1(x)$ to $u_2(x)$
  - $d_1(x)$ to $u_3(x)$
  - $d_1(x)$ to $u_5(x)$
- Satisfactory Paths:
  - 1, 2, 4, 5, 6
  - 1, 3, 4, 6

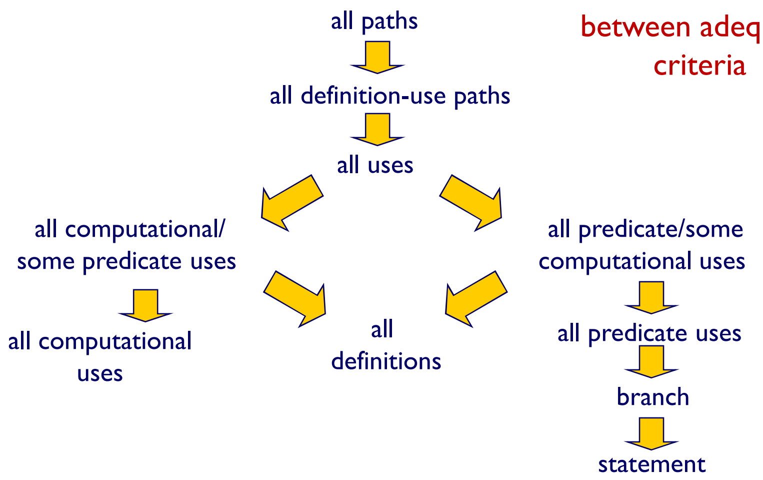
Data Flow Test: An Example (2)



All-Du-Paths
- Requires:
  - $d_1(x)$ to $u_2(x)$
  - $d_1(x)$ to $u_3(x)$
  - both paths for $d_1(x)$ to $u_5(x)$
- Satisfactory Paths:
  - 1, 2, 4, 5, 6
  - 1, 3, 4, 5, 6

Test Thoroughness

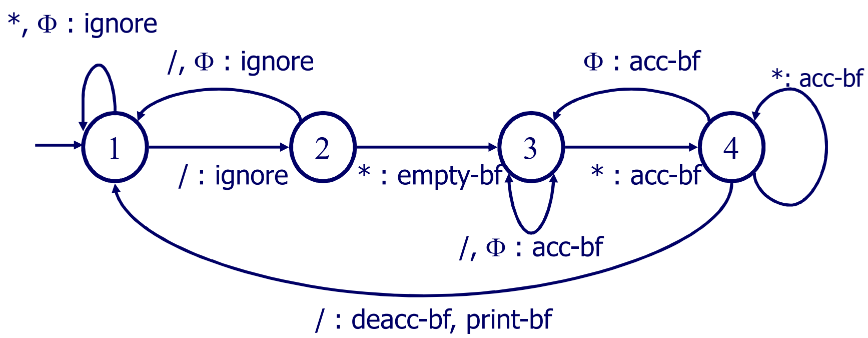
Subsume relation between adequacy criteria



State-Based Testing

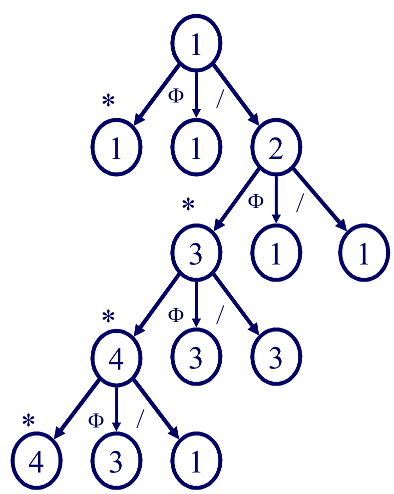
- Many computer systems are finite state machines
  - They can be in a limited number of different internal conditions (states) and the rules which determine when they change from one state to another are specified in terms of inputs to the system
- State-Based Testing
  - Verifies the relationships between events, actions, activities, states, and state transitions
  - Verifies that in response to input events the correct actions are taken and the system reaches the correct states

Finite State Machine of a Comment Printer

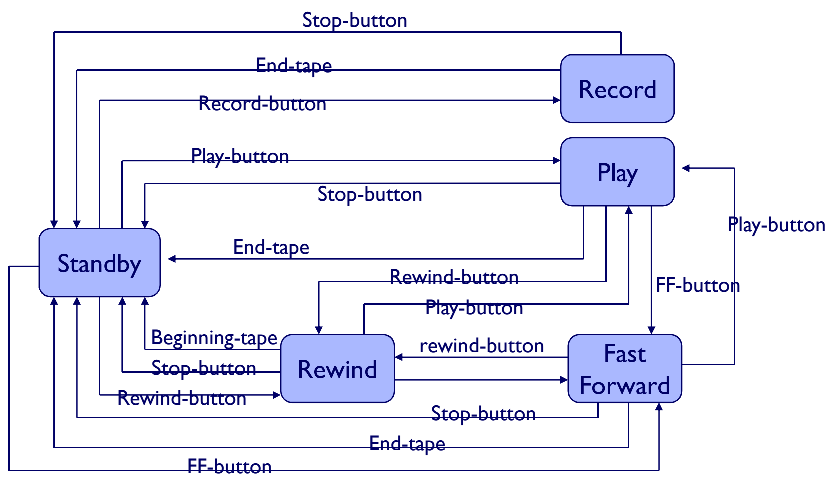


A Testing Tree

- From FSM, Testing tree is constructed to generate test sequence
- Each path from the root of the transition tree comprises a test



An example: Video Cassette Recorder (VCR)

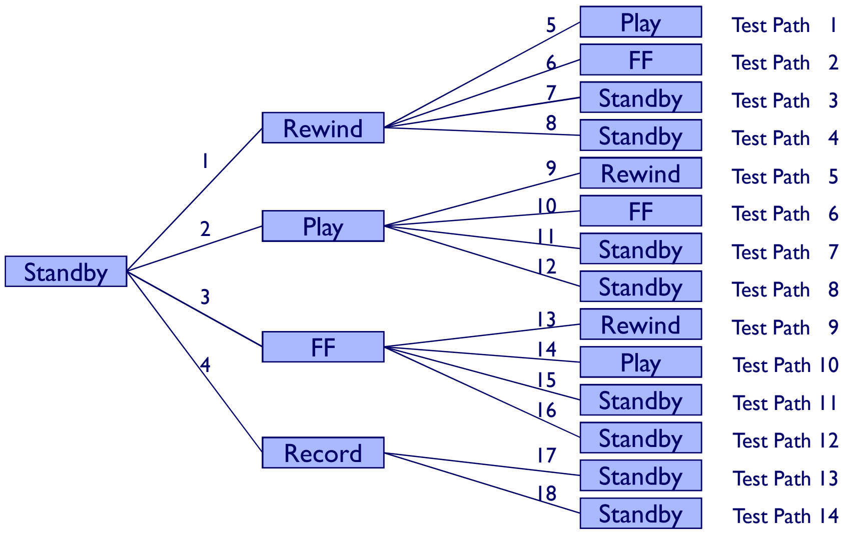


State-Event Table

- Is used in composing the transition tree.

Events\States	Standby	Rewind	Play	Fast Forward	Record
evRewindB	1-Rewind	*	9-Rewind	13-Rewind	*
evPlayB	2-Play	5-Play	*	14-Play	*
evFFb	3-FF	6-FF	10-FF	*	*
evRecordB	4-Record	*	*	*	*
evStopB	*	7-Standby	11-Standby	15-Standby	17-Standby
evEndtape	*	*	12-Standby	16-Standby	18-Standby
evBegintape	*	8-Standby	*	*	*

Transition Tree of a VCR



Mutation Testing

- A fault-based testing technique that helps the tester create a set of test cases to detect specific, predetermined types of faults
- The basic idea is to find a set of test cases that will reveal the faults that might be expected to be present in a given program
- Hypothesis
  - Competent programmers tend to write programs that are "close" to being correct. It is assumed that a fault is manifest as a small modification to the correct program code
  - A test data set that distinguishes all programs with simple faults is sensitive enough so that it will also distinguish programs with more complex faults

Mutants

- A copy of the program that contains a seeded fault.
- A mutant simulates a fault that programmers usually make.
- The effectiveness depends heavily on the types of faults.

Original Program

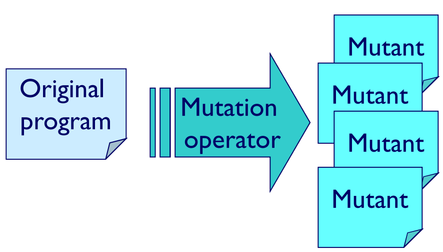
```
S3(int x)
{
    switch(x)
    {
        case 1:
            return x*2;
        case 2:
            return x;
    }
    return -1;
}
```

Mutated program

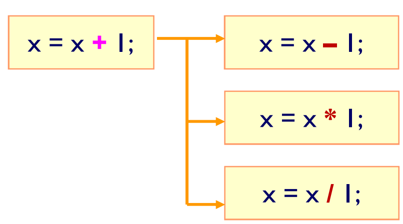
```
S3(int x)
{
    switch(x)
    {
        case 1:
            return x+2;    <- mutated
        case 2:
            return x;
    }
    return -1;
}
```


Mutation Operator

- A mutation operator simulates the fault generation rule by programmers.
- Design of the mutation operators is crucial for the effectiveness of mutation testing.

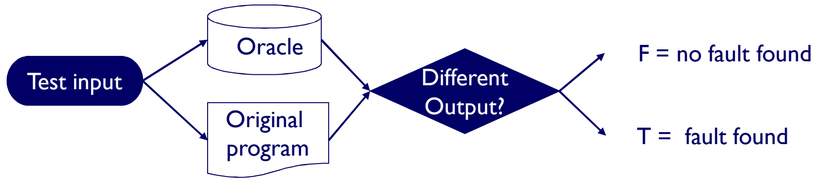


Example) AOR
(Arithmetic Operator Replace)

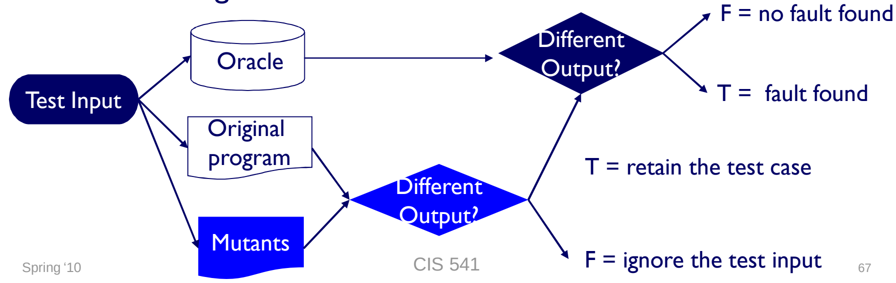


Mutation Testing Process

Conventional Testing Process



Mutation Testing Process



Model-Based Testing

- One way to generate test cases automatically is "model-based testing" where a model of the system is used for test case generation
- "Model-based testing is a testing technique where the runtime behavior of an implementation under test is checked against predictions made by a formal specification, or model." - Colin Campbell, MSR

Model in Model-Based Testing

- What is a model?
  - A model is a depiction of a system's behavior
  - Models are simpler than the systems they describe
  - Models help us understand and predict the system's behavior
- A model describes how a system should behave in response to an action
- Supply the action and see if the system responds as you expect

Using Models to Test

- Decide what type of model will be used
  - Finite State Machine, UML, Sets, Grammars, etc.
- Create the model
- Choose tests using the model
- Verify results

- (Some examples of model-based testing can be found in the tutorial slides of H. Robinson, "MBT Tutorial", StarWest 2006)

Benefits of Model-Based Testing

- Easy test case maintenance
- Reduced costs
- More test cases
- Early bug detection
- Increased bug count
- Time savings
- Time to address bigger test issues
- Improved tester job satisfaction

Obstacles to Model-Based Testing

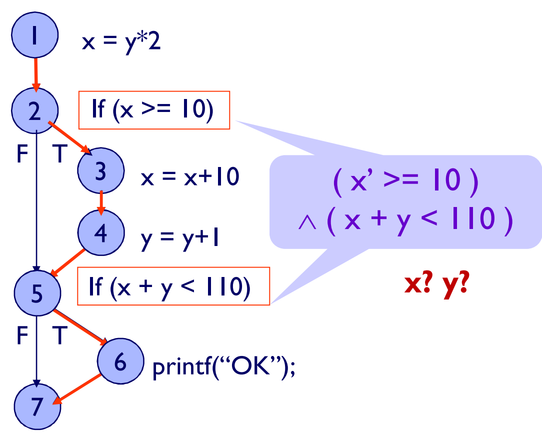
- Comfort factor
  - This is not traditional test automation
- Skill sets
  - Need testers who can design
- Expectations
  - Models can be a significant upfront investment
  - Will never catch all the bugs
- Metrics
  - Bad metrics: bug counts, number of test cases
  - Better metrics: spec coverage, code coverage

TEST DATA GENERATION

Test Data Generation

- Process of identifying program input which satisfy the selected testing criterion
- Given a program P and a path u, generate input $x \in S$ such that x traverses u
  - $P : S \rightarrow R$
  - S : the set of all possible inputs
    - the set of all vectors $x = (d_1, d_2, \ldots, d_n)$ such that $d_i \in Dx_i$, where Dx_i is the domain of input variable x_i
  - R : the set of all possible outputs
  1. Find the path predicate for the path
  2. Solve the path predicate in terms of input variables

Test Data Generation: An Example



Select a path for testing criterion

Extract the path condition

Solve the path condition to obtain test data

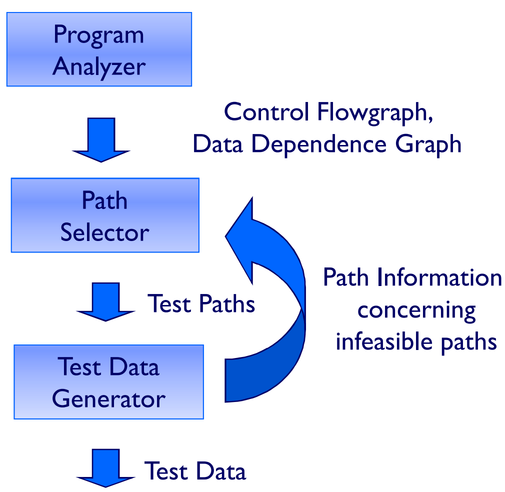
Complexity of Test Data Generation

- The problem of determining whether a solution exists to a system of inequalities is undecidable
- The path feasibility problem is undecidable

Problems in Test Data Generation

- Arrays, Pointers
  - Ambiguity
  - Complex-heap
- Objects
  - Dynamically allocated
  - Inheritance, Polymorphism
- Loops
  - Not having a constant number of iterations
- The only way of achieving an oracle is to supply extra information.
  - Requirement/design spec, assertion…

Architecture of Test Data Generator

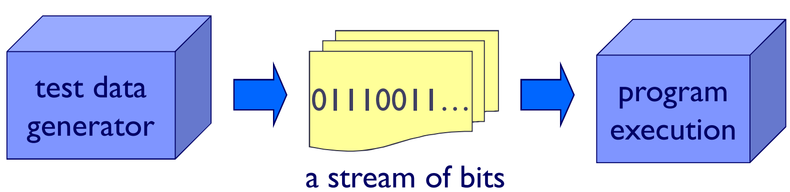


Automated Test Data Generation Techniques

- Approaches for test data generation
  - Random
  - Path-Oriented
  - Goal-Oriented
- Implementation methods
  - Static
  - Dynamic
  - Hybrid methods

Random Test Data Generation

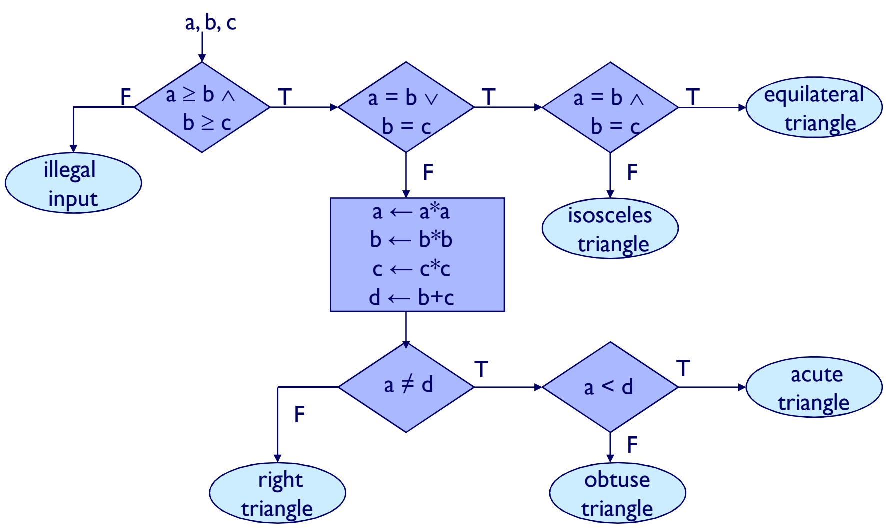
- Inputs are produced at random until a useful input is found.
  - Easy to implement
  - Frequently used as a benchmark since it is commonly reported in the literature
- The probability of selecting an input that discovers the semantically small faults is low



Specification of Triangle Classification Problem

- Input: three positive integers a, b, c, with $a \geq b \geq c$
- Output: indicate which of the following descriptions is satisfied by a, b, and c.
  1. They do not represent the sides of a triangle
  2. They are the sides of an equilateral triangle
  3. They are the sides of an isosceles, but not equilateral triangle
  4. They are the sides of a scalene right triangle
  5. They are the sides of a scalene obtuse triangle
  6. They are the sides of a scalene acute triangle

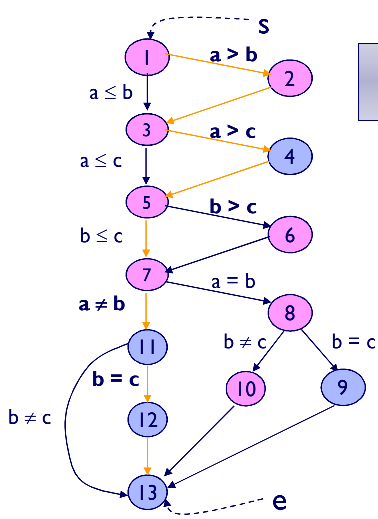
Flowchart for Triangle Classification Problem



Path-Oriented Test Data Generation

```
int triType(int a, int b, int c) {
       int type = PLAIN;
  1  if (a > b)
  2      swap(a,b);
  3  if (a > c)
  4      swap(a,c);
  5  if (b > c)
  6      swap(b,c);
  7  if (a==b) {
  8      if (b==c)
  9          type = EQUILATERAL;
         else
 10         type = ISOSCELES;
         }
 11  else if (b==c)
 12      type = ISOSCELES;
 13  return type;
}
```


Path-Oriented Test Data Generation



Path Condition
P = (a > b) ∧ (a > c) ∧ (b ≤ c) ∧ (a ≠ b) ∧ (b = c)

data dependency

Valid Path Condition
P = a > b = c

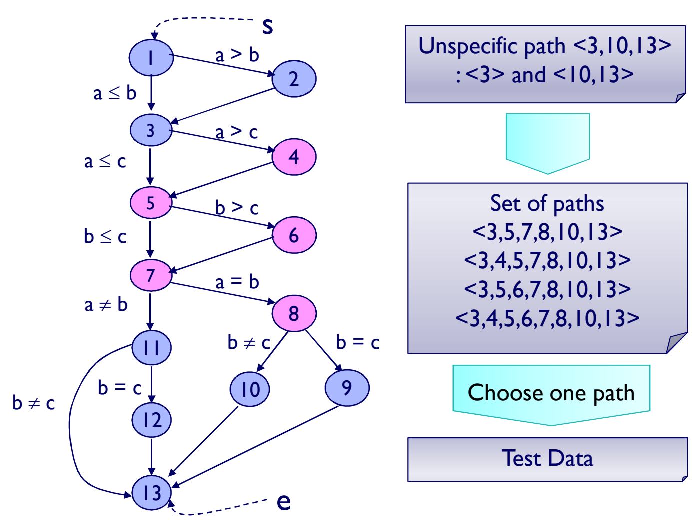
solve using ?
CLP, IRM, MILP…

Test Data: (a, b, c) = (5, 4, 4)

Goal-Oriented Test Data Generation

- Find-any-path concept
  - Hard to predict the coverage
  - More flexible to find test data
  - Alleviates the problem of selecting infeasible paths.
- Related works (Bogdan Korel)
  - Chaining approach (IEEE TSE, 1995)
    - Data dependence analysis
  - Assertion-oriented approach (IEEE TSE, 1996)
    - Assertions are inserted
    - Oracle is given in the code

Goal-Oriented Test Data Generation



Static Test Data Generation

- Generate test data using static information of the program
  - Symbolic execution is a representative techniques.
    - First proposed by J. C. King(1976)
- Difficulty with dynamic data structures, arrays, procedures, and loop conditions
- Overheads of repeated algebraic manipulation and simplification of variable and path expressions
- Not applicable to real-time software systems

Symbolic Evaluation

- Monitors manipulations performed on the input data
- Maintains the relationships between the input data and resulting values
- Represents a program's computations and domains by symbolic expressions
- Applications
  - Testing and debugging
  - Program verification
  - Program optimization and documentation
  - Test data generation

An Example Program

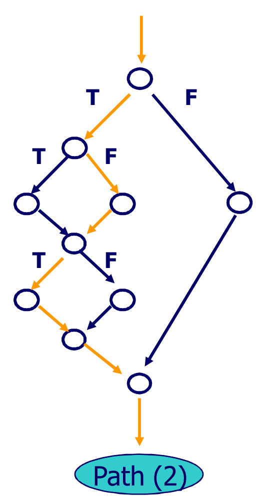
```
Procedure TRANSACT (DAYS: in integer; AMOUNT:in real;
                    BALANCE:in out real; INTEREST:out real;
                    BELOWMIN:out boolean; OVERDRAFT:out boolean)

  NEWBAL :real; -- new balance
  RATE :constant real:= 0.06; -- interest rate
  MINBAL :constant real:= 100.00 -- minimum balance
  BMCHARGE :constant real:= 0.10 -- below minimum charge
  ODCHARGE :constant real:= 4.00 -- overdraft charge

  S  begin
  1  OVERDRAFT := false;
  2  BELOWMIN := false;
  3  NEWBAL := BALANCE*(1 + RATE/365)**DAYS;
  4  INTEREST := NEWBAL – BALANCE;
  5  if AMOUNT > 0.0 then -- process deposit
  6      NEWBAL := NEWBAL + AMOUNT;
     endif;
```


An Example Program

```
7     if AMOUNT < 0.0 then -- process check
8         if - AMOUNT > NEWBAL then
9             OVERDRAFT := true;
10            NEWBAL := NEWBAL - ODCHARGE;
          else
11            NEWBAL := NEWBAL + AMOUNT;
          endif
12        if NEWBAL < MINBAL then
13            BELOWMIN := true;
14            NEWBAL := NEWBAL - BMCHARGE;
          endif
      endif
15    BALANCE := NEWBAL;
f     end TRANSACT;
```



Example of Path Condition

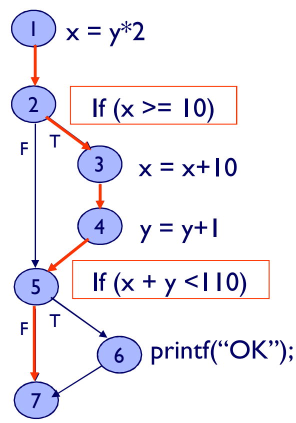
- Path Computation for Path P(2)
  - BALANCE = balance*(1+0.06/365)**days + amount - 0.1
  - INTEREST= balance*(1+0.06/365)**days – balance
  - BELOWMIN := true
  - OVERDRAFT := true
- Path Condition for Path P(2)
  - (amount <= 0.0) and
    (amount < 0.0) and
    (-amount <= balance*(1+0.06/365)**days) and
    (balance*(1+0.06/365)**days + amount < 100.0)

Symbolic Evaluation of Path (2)

Statement	Interpreted predicate	Interpreted assignments
s	true	DAYS = days, AMOUNT = amount, BALANCE = balance, INTEREST = ?, BELOWMIN = ?, OVERDRAFT = ?, NEWBAL = ?, RATE = 0.06, MINBAL = 100.0, BMCHARGE = 0.1, ODCHARGE = 4.0
1		OVERDRAFT = false
2		BELOWMIN = false
3		NEWBAL = balance*(1 + 0.06/365)**days
4		INTEREST = balance*(1 + 0.06/365)**days - balance
(5.7)	amount <= 0.0	
(7,8)	amount < 0.0	
(8,11)	– amount <= balance*(1 + 0.06/365)**days	
11		NEWBAL = balance*(1 + 0.06/365)**days + amount
(12,13)	balance*(1 + 0.06/365)**days + amount < 100.0	
13		BELOWMIN = true
14		NEWBAL = balance*(1 + 0.06/365)**days + amount – 0.1
15		BALANCE = balance*(1 + 0.06/365)**days + amount – 0.1

Symbolic Execution: An Example

input variable : y → Y



Node	Path condition	Path action
1		x = Y*2
2	Y*2 >= 10	
3		x = Y*2 + 10
4		y = Y + 1
5	(Y*2 + 10) + (Y + 1) >= 110	

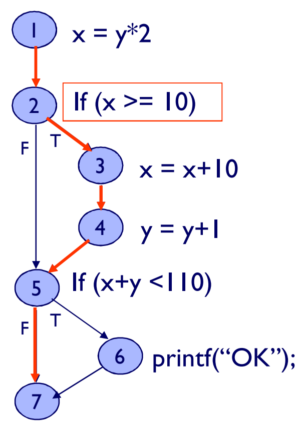
Y >= 5 ∧ Y >= 33 ➔ Y >= 33

Dynamic Test Data Generation

- Execution-based approach
  - The program is executed with some, possibly randomly selected input
  - If some desired test requirement is not satisfied, inputs are incrementally modified until one of them satisfies the test requirements
  - Function minimization search
    - Genetic algorithms
  - Dynamic data flow analysis [Korel]

Function Minimization Problem:

input variable : y



Find a value of y that minimizes F(y)

$F(y) = 10 - x_2(y)$ if $x_2(y) < 10$

0 otherwise

$x_2(y)$ is the value of x at the statement #2 when the program is executed the input y

Dynamic Test Data Generation

- Can handle dynamic data structures (pointer, array) and function calls
- Expensive; requires many iterations before a suitable input is found
- Inefficient in handling infeasible paths
- Monitoring can be done by instrumentation

EMBEDDED SOFTWARE TESTING

Spring '10 CIS 541 97

Embedded System

- A system whose prime function is not that of information processing, but which nevertheless requires information processing in order to carry out their prime function
- A system that is logically incorporated in a larger system whose primary function is not computation

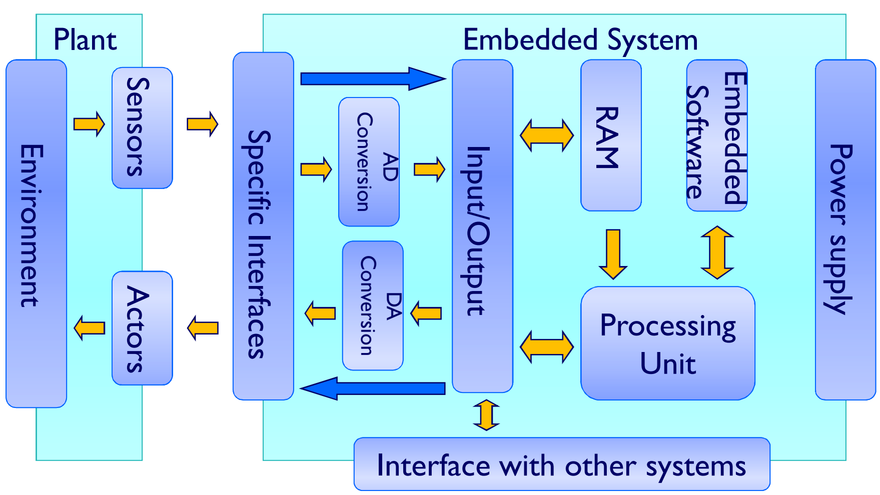
- Host Environment
  - The operating system or computer which the embedded software code is written on
- Target Environment
  - The operating system or device which the embedded software code will execute on

Spring '10 CIS 541 98

Examples of Embedded Systems

- Cruise control
- Set-top box
- Pacemaker
- NMR scanner
- Railroad signaling
- Telecom switching
- Missile defense systems

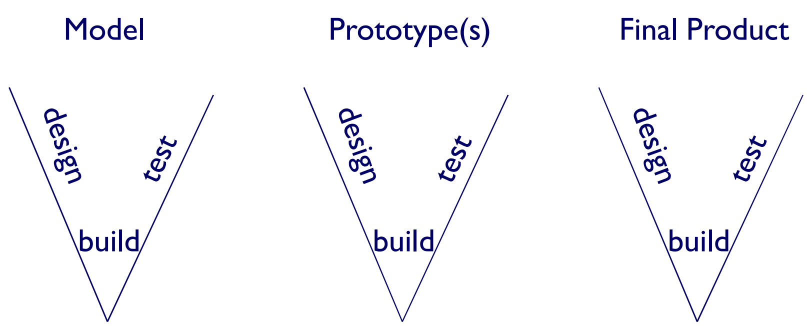
Generic Scheme of an Embedded System



Development of an Embedded Software System

- Model Building
  - A model of the system is built on PC
  - A model simulates the required system behavior
- Prototype Building
  - Code is generated from the model and embedded in a prototype
- Product Building
  - The experimental hardware of the prototypes is gradually replaced by the real hardware
  - The system is built in its final form and mass produced

Multiple V Development Lifecycle



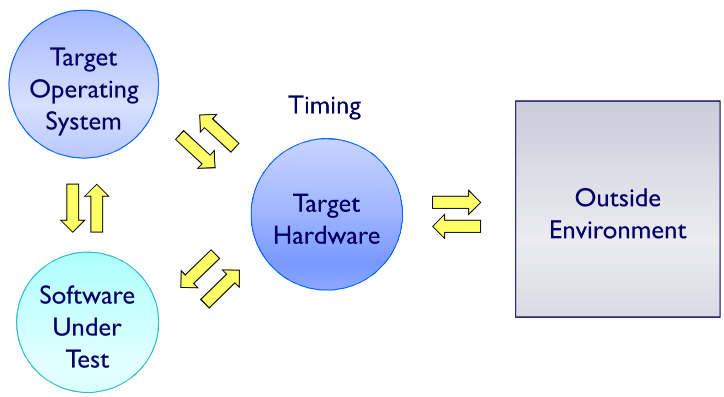
Problems in Testing Embedded Software

- Divergent environment
  - Testing happens too late and depends on hardware
  - Simulation-based testing
    - User interface, HW interface, Protocol, Time
- Reactive and Concurrent behavior
  - Tests are not repeatable and takes too much time and resources.
  - Static analysis vs. Run-time monitoring and checking
- Timing requirements
- Probe effects
  - Hardware monitoring
    - Monitoring the system bus activity for data and instructions
    - ROM monitors
    - In-circuit emulator

Testing Embedded Software

- Host (Development) Environment
  - Unix, Windows NT, …
- Target Environment
  - Product-specific
  - Specific for target system
  - Target OS
    - Communication, Scheduling, …
  - Target HW
    - I/O, interrupts, interfaces, …
  - Environment
    - Protocols, UI, …
  - Timing

Testing Embedded Software



Test Environment

- Hardware/Software/Network
- Test Databases
  - Test Data have to be stored
- Simulation and Measurement Equipment

Embedded Software Testing Techniques

- "White Box" or "Code-Based" Testing
  - Requires that the tester has a detailed knowledge of the software structure and its intended role.
  - Embedded software requires higher code coverage percentages due to the strict requirements for safety and reliability.
    - Many code-based testing tools on the host environment are introduced

Embedded Software Testing Techniques (cont.)

- "Black Box" or "Functional" Testing
  - The quality of the requirements will affect the resultant tests
  - An aspect of black box testing of embedded SW is to test to extremes
    - Functional testing should not only exercise how the software works, but how the software fails

Embedded Software Testing Phases

- Module Testing
- Integration Testing
- System Testing
- Hardware/Software Integration Testing

Spring '10 CIS 541 109

Embedded Software Testing Tools

- Memory Analysis Tools
  - Designed to address faults in allocation of dynamic memory
  - SW and HW based memory analyzers
- Performance Analysis Tools
  - Provide specific data about how and when execution time was spent
  - The majority of execution time is spent in a relatively small amount of code
- GUI Testers
  - Has the ability to record and playback operator actions

Spring '10 CIS 541 110

Embedded Software Testing Tools (cont.)

- Code Coverage Tools
  - Track the portion of the code that has been executed.
  - SW and HW based monitoring
- General Support Tools
  - Databases, Defect Tracking, Configuration Management

References

- Y. Kwon & H. Bae, SEP524 Software Quality Assurance
- D. Peled, Lecture2: Testing
- H. Robinson, Model-Based Testing Tutorial, StarWest 2006