Introduction to Software Testing
(*2nd edition*)
Chapter 5

Criteria-Based Test Design

Paul Ammann & Jeff Offutt

http://www.cs.gmu.edu/~offutt/softwaretest/

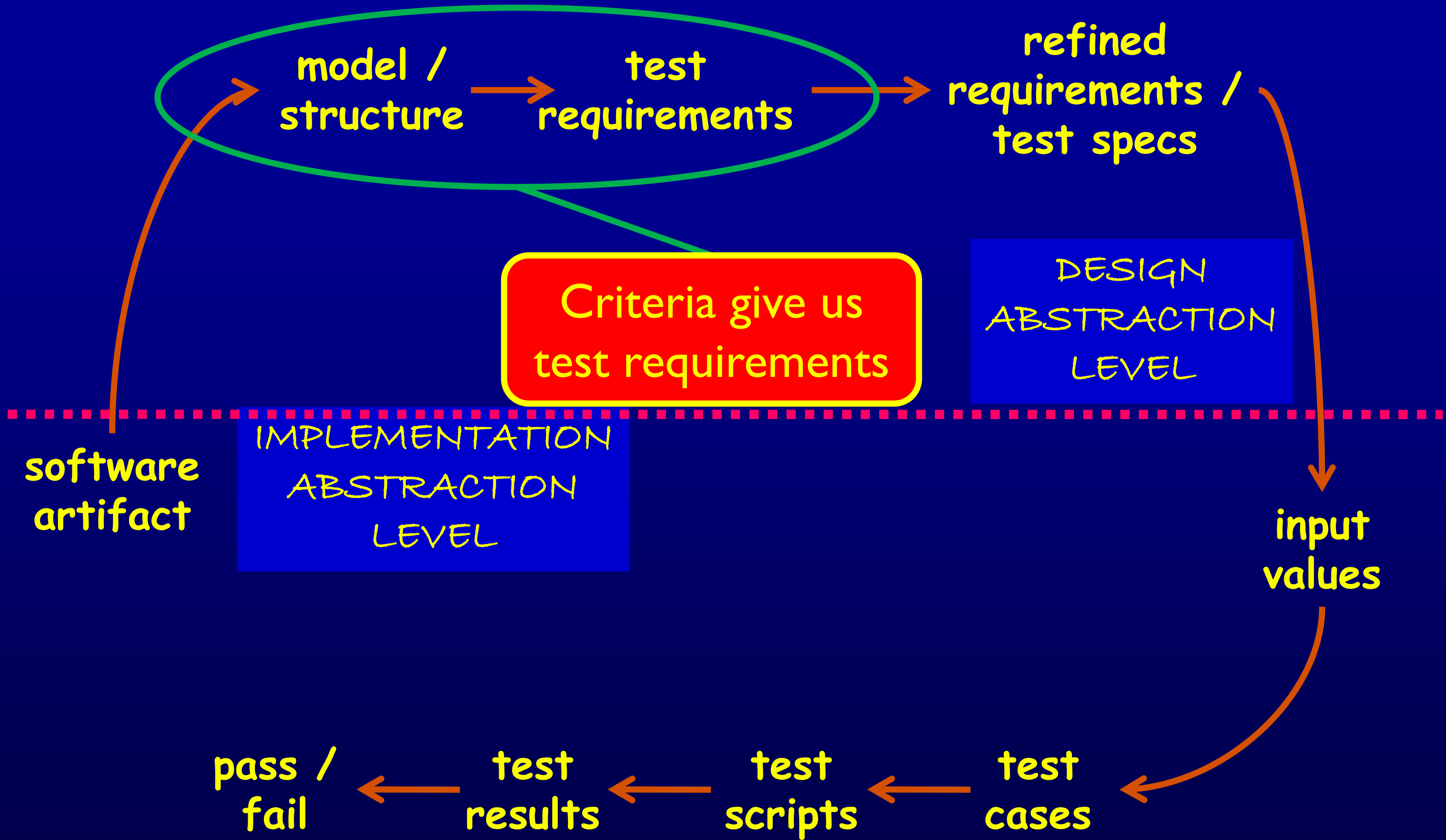
20 September 2015

Changing Notions of Testing

- Old view focused on testing at each software development phase as being very different from other phases
  – Unit, module, integration, system …

- New view is in terms of structures and criteria
  – input space, graphs, logical expressions, syntax
- Test design is largely the same at each phase
  – Creating the model is different
  – Choosing values and automating the tests is different

Model-Driven Test Design



New : Test Coverage Criteria

A tester's job is simple : Define a model of the software, then find ways to cover it

- Test Requirements : A specific element of a software artifact that a test case must satisfy or cover

- Coverage Criterion : A rule or collection of rules that impose test requirements on a test set

Example : Jelly Bean Coverage

Flavors :

1. Lemon
2. Pistachio
3. Cantaloupe
4. Pear
5. Tangerine
6. Apricot

Colors :

1. Yellow (Lemon, Apricot)
2. Green (Pistachio)
3. Orange (Cantaloupe, Tangerine)
4. White (Pear)

- Possible coverage criteria :
  1. Taste one jelly bean of each flavor
     - Deciding if yellow jelly bean is Lemon or Apricot is a controllability problem
  2. Taste one jelly bean of each color

Test Coverage Criteria

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures …

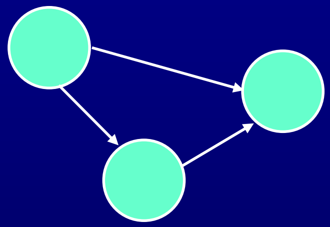
Criteria Based on Structures

Structures : Four ways to model software

1. Input Domain Characterization (sets)

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}

2. Graphs



3. Logical Expressions

(not X or not Y) and A and B

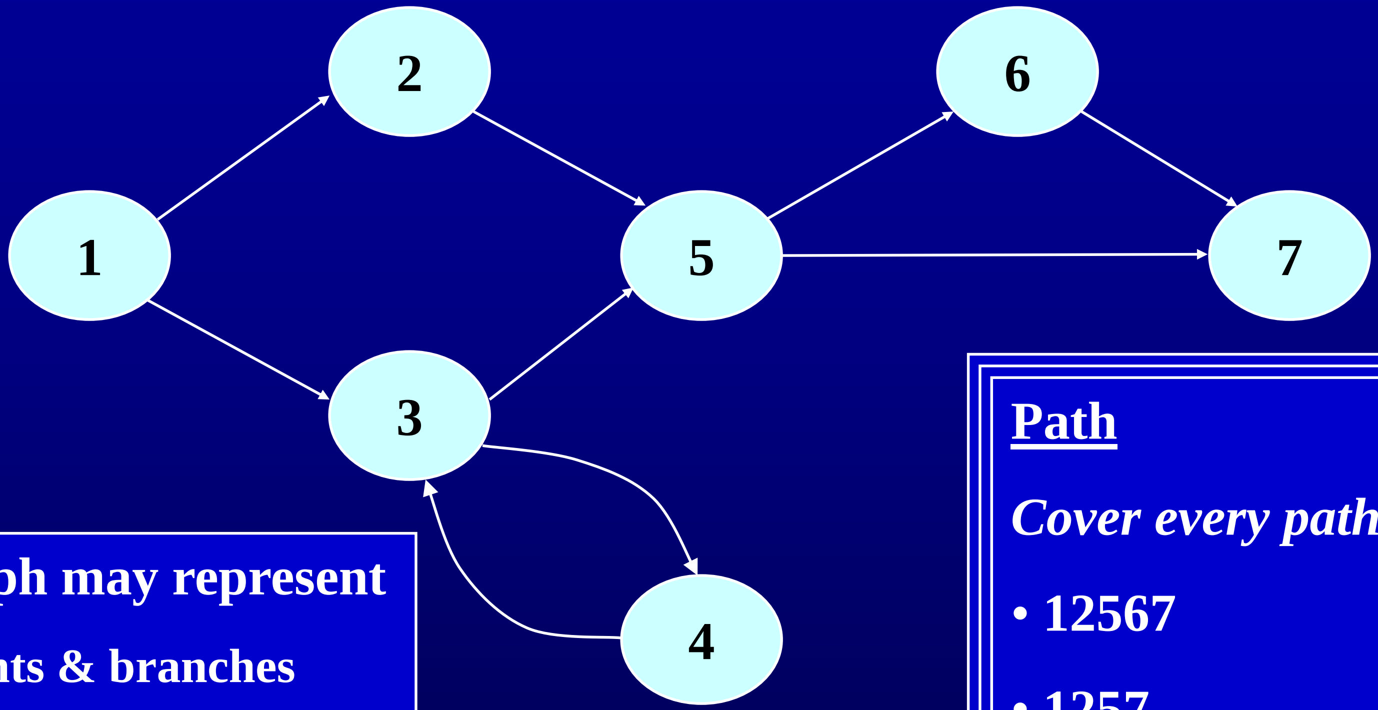
4. Syntactic Structures (grammars)

```
if (x > y)
   z = x - y;
else
  z = 2 * x;
```

1. Input Domain Characterization

- Describe the input domain of the software
  - Identify input parametrs
  - Partition each input into finite sets of representative values
  - Choose combinations of values
- Example
  - Parameters *F (int X, int Y)*
  - Possible values X: { *<0, 0, 1, 2, >2* }, *Y* : { *10, 20, 30* }
  - Tests
    - *F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)*

2. Graph Coverage



This graph may represent

- **statements & branches**
- **methods & calls**
- **states and transitions**

Path

Cover every path

- **12567**
- **1257**
- **13567**
- **1357**
- **1343567**
- **134357 …**

3. Logical Expressions

((a > b) or G) and (x < y)

Transitions

Program Decision Statements

Software Specifications

Logical Expressions

3. Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$

- **Example:**
  - **Predicate Coverage : Each predicate must be true and false**
    - $((a>b) \text{ or } G) \text{ and } (x < y)$ = True, False
  - **Clause Coverage : Each clause must be true and false**
    - $(a > b)$ = True, False
    - G = True, False
    - $(x < y)$ = True, False

4. Syntactic Structures-I

- Based on a grammar, or other syntactic definition
- Primary example is mutation testing
  1. Induce small changes to the program: mutants
  2. Find tests that cause the mutant programs to fail: killing mutants
  3. **Killing mutants** is defined as different output from the original program
- Original program and a mutant:

```
if (x > y)
    z = x - y;
else
    z = 2 * x;
```

```
if (x > y)
Δif (x >= y)
    z = x - y;
else
    z = 2 * x;
```

- Find a test case to kill the mutant
  – x = y != 0 , For example: (2,2)

4. Syntactic Structures-II

- Two overall phases of mutation testing:

1. Creating a test suite *T* using syntactic structures
   - Creating mutants
   - Generating test cases to kill them
2. Testing the original program with T to find faults
   - Comparing expected outputs with actual results

- The fundamental premise of mutation testing:
  - In practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault
- Mutation testing is difficult to apply by hand
  - An automated system should create mutants and generate killing test cases

4. Syntactic Structures-III

- The idea of "killing" a mutant is not as obvious as "reaching" a node, "traversing" a path, or "satisfying" a set of truth assignments
- However, it is clear however, <u>that the software is tested well, or the test cases do not kill mutants</u>
- Mutation testing aims to evaluate the fault detection capability of **<u>a test suite</u>**
  – Testers change specific components of an application's source code to ensure a software test suite will be able to detect the changes (i.e., intentional defects)
  – Changes are intended to cause errors in the program
  – Mutation testing is directed to ensure the quality of a software testing suite

4. Syntactic Structures-IV

- If the testing suite has failed to detect the mutations
  - It should then be worked on to be more effective
- The software test suite can be scored by using the mutation score
  - The percentage of killed mutants divided by the total number of mutants
- A test set that kills all mutants is said to be *adequate*
- A mutation score of 1.00 is usually impractical
- The tester defines a "threshold" value
  - a minimum acceptable mutation score
- Until the threshold mutation score is reached, new test cases are generated to target live mutants

Coverage

Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

- Infeasible test requirements : test requirements that cannot be satisfied
  - No test case values exist that meet the test requirements
  - Example: Dead code
  - Detection of infeasible test requirements is formally undecidable for most test criteria
- Thus, 100% coverage is impossible in practice

More Jelly Beans

T1 = { three Lemons, one Pistachio, two Cantaloupes, one Pear, one Tangerine, four Apricots }

- Does test set T1 satisfy the flavor criterion ?

T2 = { One Lemon, two Pistachios, one Pear, three Tangerines }

- Does test set T2 satisfy the flavor criterion ?
- Does test set T2 satisfy the color criterion ?

Coverage Level

The ratio of the number of test requirements satisfied by *T* to the size of *TR*

- T2 on the previous slide satisfies 4 of 6 test requirements

Comparing Criteria with Subsumption (5.2)

- Criteria Subsumption : A test criterion *C1* subsumes *C2* if and only if every set of test cases that satisfies criterion *C1* also satisfies *C2*
- Must be true for every set of test cases
- *Examples* :
  - The flavor criterion on jelly beans subsumes the color criterion … if we taste every flavor we taste one of every color
  - If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement

Advantages of Criteria-Based Test Design (5.3)

- Criteria maximize the "bang for the buck"
  - Fewer tests that are more effective at finding faults
- Comprehensive test set with minimal overlap
- Traceability from software artifacts to tests
  - The "why" for each test is answered
  - Built-in support for regression testing
- A "stopping rule" for testing—advance knowledge of how many tests are needed
- Natural to automate

Criteria Summary

- Many companies still use "monkey testing"
  - A human sits at the keyboard, wiggles the mouse and bangs the keyboard
  - **No automation**
  - Minimal training required
- Some companies **automate** human-designed tests
- But companies that use both automation and criteria-based testing

Save money

Find more faults

Build better software

Structures for Criteria-Based Testing