

# Review Tutorial

ECE453/SE465/CS447

April 2009

Zarrin Langari

# Race Condition

Data race: two threads simultaneously update a data structure.

Symptoms:

- The most common symptom of a race condition is unpredictable values of variables that are shared between multiple threads.

Solution: locking ensures only one thread at a time has access.

- Check that all shared memory accesses follow a consistent locking discipline.

# Race Condition

## Example:

- Thread 1

Total = Total + val1

- Thread 2

Total = Total - val2

# Assembly Example

## Thread 1:

1. `mov eax,dword ptr ds:[031B49DCh]`
2. `add eax,edi`
- 3-5. ...
6. `mov dword ptr ds:[031B49DCh],eax`

## Thread 2:

1. `mov eax,dword ptr ds:[031B49DCh]`
2. `sub eax,edi`
- 3-5. ...
6. `mov dword ptr ds:[031B49DCh],eax`

# Deadlock

- A *deadlock* occurs when two threads each lock a different variable at the same time and then try to lock the variable that the other thread already locked.
- As a result, each thread stops executing and waits for the other thread to release the variable.
- Because each thread is holding the variable that the other thread wants, nothing occurs, and the threads remain deadlocked.
- Deadlock behaves almost like an infinite loop.

## Reason:

Excessive synchronization and inconsistent locks.

## Symptoms:

The program or group of threads stops responding. This is also known as a *hang*.

Solution: One way is to have an ordering among thread execution, or prioritize acquiring locks by processes.

```

public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s has bowed to me!\n",
                this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s has bowed back to me!\n",
                this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}

```

# Syntax-based testing

- Terminal Symbol Coverage (TSC) : TR contains each terminal symbol  $t$  in the grammar  $G$ .
- Production Coverage (PC) : TR contains each production  $p$  in the grammar  $G$ .
- Derivation Coverage (DC) : TR contains every possible string that can be derived from the grammar  $G$ .

# Syntax-based testing

- The number of TSC tests is bound by the number of terminal symbols
- The number of PC tests is bound by the number of productions
- The number of DC tests depends on the details of the grammar

# Mutation Testing

- A mutant is a variation of a valid string
  - Mutants may be valid or invalid strings
- Mutation is based on “mutation operators” and “ground strings”
- **Mutation Coverage (MC)** : For each  $m \in M$ , TR contains exactly one requirement, to kill  $m$ .
  - What does the killing mean? Why should we kill mutants?
- **Mutation Operator Coverage (MOC)** : For each mutation operator, TR contains exactly one requirement, to create a mutated string  $m$  that is derived using the mutation operator.
- **Mutation Production Coverage (MPC)** : For each mutation operator, TR contains several requirements, to create one mutated string  $m$  that includes every production that can be mutated by that operator.

# Example: Stream Grammar

**Stream** ::= action\*

*Start symbol*

**action** ::= actG | actB

*Non-terminals*

**actG** ::= "G" s n

**actB** ::= "B" t n

*Production rule*

**s** ::= digit<sup>1-3</sup>

**t** ::= digit<sup>1-3</sup>

**n** ::= digit<sup>2</sup> "." digit<sup>2</sup> "." digit<sup>2</sup>

*Terminals*

**digit** ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" |  
"7" | "8" | "9"

# Example: Stream Grammar

**Stream ::= action action \***  
**::= actG action\***  
**::= G s n action\***  
**::= G digit<sup>1-3</sup> digit<sup>2</sup> . digit<sup>2</sup> . digit<sup>2</sup> action\***  
**::= G digitdigit digitdigit.digitdigit.digitdigit action\***  
**::= G 18 08.01.90 action\***  
  
...

## Valid Mutants

### Ground Strings

*G 18 08.01.90*

*B 14 06.27.94*

### Mutants

*B 18 08.01.90*

*B 45 06.27.94*

## Invalid Mutants

*13 18 08.01.90*

*B 134 06.27*

**Stream ::= action action \***  
**::= actG action\***  
**::= G s n action\***  
**::= G digit<sup>1-3</sup> digit<sup>2</sup> . digit<sup>2</sup> . digit<sup>2</sup> action\***  
**::= G digitdigit digitdigit.digitdigit.digitdigit action\***  
**::= G 18 08.01.90 action\***  
 ...

Ground String

*G 18 08.01.90*

*B 14 06.27.94*

Mutants using MOC

*B 18 08.01.90*

*B 19 06.27.94*

Mutation Operators

- *Exchange actG and actB*
- *Replace digits with other digits*

Mutants using MPC

*B 18 08.01.90      G 14 06.27.94*

*G 28 08.01.90      B 11 06.27.94*

*G 38 08.01.90      B 13 06.27.94*

*G 48 08.01.90      B 15 06.27.94*

*G 58 08.01.90      B 16 06.27.94*

...

...

# Mutation Testing

- Mutation testing :
  - most commonly used for unit testing and integration testing of classes.
- The original and most widely known application of syntax-based testing is to modify programs
- Operators modify a ground string (program under test) to create mutant programs
- Mutant programs must compile correctly (valid strings)
- Mutants are not tests, but used to find tests
- Once mutants are defined, tests must be found to cause mutants to fail when executed
- This is called “killing mutants”

# Mutation Testing

- What do we conclude if a mutant can not be killed?
  - If a mutation was introduced without the behavior (generally output) of the program being affected, then:
    - either the code that had been mutated was never executed (redundant code)
    - or that the testing suite was unable to locate the injected fault.

# Grammar-Based Testing

## Program-based

## Integration

## Model-Based

## Input-Based

String mutation

String mutation

String mutation

String mutation

Grammar

- Program mutation
- Valid strings
- Mutants are not tests
- Must kill mutants

- FSMs
- Model checking
- Valid strings
- Traces are tests

- Input validation testing
- XML and others
- Invalid strings
- No ground strings
- Mutants are tests

- Compiler testing
- Valid and invalid strings

- Test how classes interact
- Valid strings
- Mutants are not tests
- Must kill mutants
- Includes OO

Grammar

- Input validation testing
- XML and others
- Valid strings

Δ2

Reachability: True

Infection:  $(B < A) \neq (B > A) \equiv A \neq B$

Propagation: True

Δ 3

Reachability: True

Infection:  $A \neq \text{minVal} \equiv \text{False} \rightarrow \text{equivalent}$

Propagation: N/A

Δ 4

Reachability :  $B < A$

Infection: True

Propagation : True

Δ 5

Reachability :  $B < A$

Infection :  $A \neq B$

Propagation: True

Δ 6

Reachability :  $B < A$

Infection :  $B \neq 0$

Propagation: True

## Section 5.2 -- Q2

```
//Effects: If numbers null throw NullPointerException
// else return LAST occurrence of val in numbers[]
// If val not in numbers[] return -1

public static int findVal(int numbers[], int
val)
{
    int findVal = -1;

    for (int i=0; i<numbers.length; i++)
        // for (int i=(0+1); i<numbers.length; i++)
        if (numbers [i] == val)
            findVal = i;
    return (findVal);
}
```

```
//Effects: If x null throw NullPointerException
// else return the sum of the values in x

public static int sum(int[] x)
{
    int s = 0;
    for (int i=0; i < x.length; i++)
    {
        s = s + x[i];
        // s = s - x[i];
    }
    return s;
}
```

## Section 5.2 -- Q2

(a)

**findVal:** The mutant is always reached, even if  $x = \text{null}$ .

**sum:** If  $x$  is null or the empty array, ie  $x = \text{null}$  or  $[],$  then the mutant is never reached.

(b)

**findVal:** Infection always occurs, even if  $\text{num} = \text{null},$  because  $i$  always has the wrong value after initialization in the loop.

**sum:** Any input with all zeroes will reach but not infect. Examples are:  $x = [0]$  or  $[0, 0].$

(c)

**findVal:** As long as the last occurrence of  $\text{val}$  isn't at  $\text{numbers}[0],$  the correct output is returned. Examples are:  $(\text{numbers}, \text{val}) = ([1, 1], 1)$  or  $([-1, 1], 1)$  or  $(\text{null}, 0).$

**sum:** Any input with nonzero entries, but with a sum of zero, is fine. Examples are:  $x = [1, -1]$  or  $[1, -3, 2].$

(d)

**findVal:** Any input with  $\text{val}$  only in  $\text{numbers}[0]$  works. An example is:  $(\text{numbers}, \text{val}) = ([1, 0], 1)$

**sum:** Any input with a nonzero sum works. An example is:  $x = [1, 2, 3]$

```

1. public static int cal (int month1, int day1, int month2, int day2, int year)
2. {
3. //*****
4. // Calculate the number of Days between the two given days in
5. // the same year.
6. // preconditions : day1 and day2 must be in same year
7. //     1 <= month1, month2 <= 12
8. //     1 <= day1, day2 <= 31
9. //     month1 <= month2
10. //     The range for year: 1 ... 10000
11. //*****
12.     int numDays;
13.
14.     if (month2 == month1) // in the same month //mutant 6
15.         numDays = day2 - day1; //mutant 1 - 4
16.     else
17.     {
18.         // Skip month 0.
19.         int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
20.         // Are we in a leap year?
21.         int m4 = year % 4;
22.         int m100 = year % 100;
23.         int m400 = year % 400;
24.         if ((m4 != 0) || ((m100 == 0) && (m400 != 0))) //mutant 5, 7 - 9
25.             daysIn[2] = 28;
26.         else
27.             daysIn[2] = 29; //mutant 12
28.         // start with days in the two months
29.         numDays = day2 + (daysIn[month1] - day1); //mutant 10, 11
30.         // add the days in the intervening months
31.         for (int i = month1 + 1; i <= month2-1; i++)
32.             numDays = daysIn[i] + numDays;
33.     }
34.     return (numDays);
35. }

```

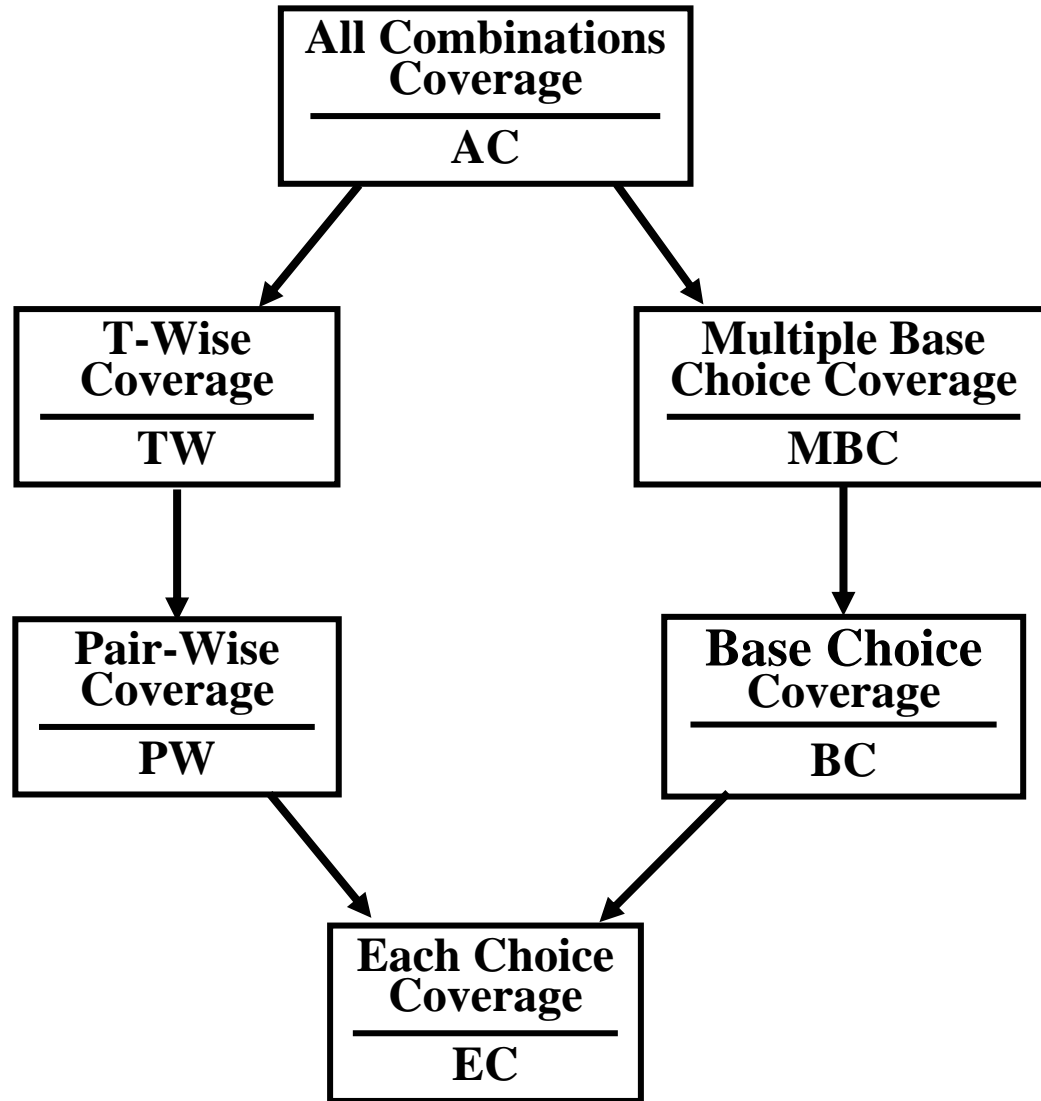
```
1 numDays = abs (day2 - day1); // ABS operator
2 numDays = day2 - -abs (day1); // ABS operator (negAbs)
3 numDays = failOnZero (day2 - day1); // ABS operator
  (failOnZero)
4 numDays = day2 * day1; // AOR operator
5 if ((m4 > 0) || ((m100 ==0) && (m400 != 0))) // ROR operator
6 if (month2 >= month1) // ROR operator
7 if (false) // ROR operator (falseOp)
8 if ((m4 != 0) || ((m100 ==0) || (m400 != 0))) // COR operator
9 if ((m4 != 0)) // COR operator (leftOp)
10 numDays = day2 + -(daysIn[month1] - day1); // UOI operator
11 numDays = day1 + (daysIn[month1] - day1); // SVR operator
12 Replace "daysIn[2] = 29;" with "Bomb();" // BSR operator
```

# Input Space Partitioning - Summary

- Application to testing
  - Find characteristics in the inputs : parameters, semantic descriptions, ...
  - Partition each characteristics
  - Choose tests by combining values from characteristics
- The partition must satisfy two properties :
  1. blocks must be pairwise disjoint (no overlap)
  2. together the blocks cover the domain  $D$  (complete)
- Two Approaches to Input Domain Modeling
  1. Interface-based approach
    - Develops characteristics directly from individual input parameters
  2. Functionality-based approach
    - Develops characteristics from a behavioral view of the program under test

# Input Space Partitioning - Summary

- **All Combinations (ACoC)** : All combinations of blocks from all characteristics must be used.
- **Each Choice (EC)** : One value from each block for each characteristic must be used in at least one test case.
- **Pair-Wise (PW)** : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.
- **t-Wise (TW)** : A value from each block for each group of t characteristics must be combined.
- **Base Choice (BC)** : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.
- **Multiple Base Choice (MBC)** : One or more base choice blocks ...



(a) list = [2, 4, 2]; e = 2 or list = [2]; e = 2

(b) list = [2, 3, 4]; e = 6

(c) separate the characteristics into separate partitions:

- Whether e is first in the list: true, false
- Whether e is last in the list: true, false

May add:

Whether e is in the list: true, false

(a,b,c)

### Characteristic: Stack state

- Whether the stack is empty.
  - true (Value stack = [])
  - false (Values stack = ["cat"], ["cat", "hat"])
- The size of the stack.
  - 0 (Value stack = [])
  - 1 (Possible values stack = ["cat"], [null])
  - more than 1 (Possible values stack = ["cat", "hat"], ["cat", null], ["cat", "hat", "ox"])
- Whether the stack contains null entries
  - true (Possible values stack = [null], [null, "cat", null])
  - false (Possible values stack = ["cat", "hat"], ["cat", "hat", "ox"])

### Characteristic for Object x is

- Whether x is null.
  - true (Value x = null)
  - false (Possible values x = "cat", "hat", "")

### Characteristic : combination of Object x and the stack state

- Does Object x appear in the stack?
  - true (Possible values: (null, [null, "cat", null]), ("cat", ["cat", "hat"]))
  - false (Possible values: (null, ["cat"]), ("cat", ["hat", "ox"]))

- a) Yes, because the union of all blocks cover the whole domain for sets.
- b) Yes, because a set is either empty, or it's pointer does not refer to anything, or it has an element.
- c) No, it does not include sets with some common elements:  $s1=\{1,2\}$   $s2=\{1,3\}$
- d) No, The first three blocks are not disjoint for  $s1=\{1,2\}$   $s2=\{1,2\}$
- e)  $1 + (3-1)$ //for first partition +  $(4-1)$ //for second partition = 6

## Characteristics

- Whether queue is empty.
- Whether queue is full.
- Size of queue.
- Value of cap.
- Value of capacity. (Note that this may differ from cap)
- Whether x is null.

# Testing - Summary

- Most testing is actually regression testing
- Test criteria make regression testing much easier to **automate**
- OOP has changed the way in which we integrate and test software components
- To be successful, testing has to be integrated throughout the **process**
- Identifying correct outputs is almost as hard as writing the program
  - four general methods for checking outputs:
    - Direct verification
    - Redundant computation
    - Consistency checks
    - Data redundancy

# Types of Test Plans

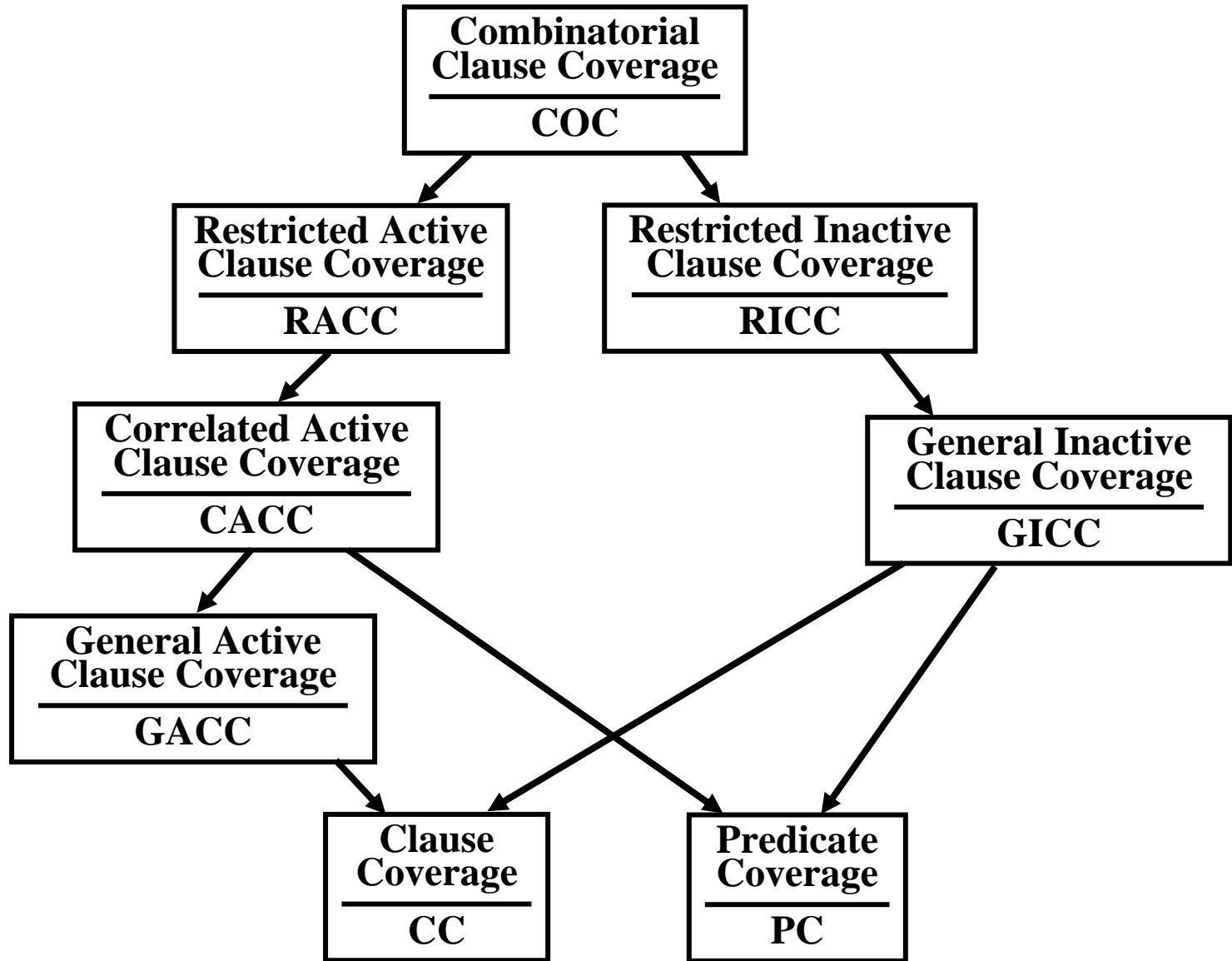
## Test Plan

- A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.
- Mission plan – tells “why”
  - Usually one mission plan per organization or group
  - Least detailed type of test plan
- Strategic plan – tells “what” and “when”
  - Usually one per organization, or perhaps for each type of project
  - General requirements for coverage criteria to use
- Tactical plan – tells “how” and “who”
  - One per product
  - More detailed
  - Living document, containing test requirements, tools, results and issues such as integration order

# Test Plan Contents – Tactical Testing

- Purpose
- Outline
- Test-plan ID
- Introduction
- Test reference items
- Features that will be tested
- Features that will not be tested
- Approach to testing (criteria)
- Criteria for pass / fail
- Criteria for suspending testing
- Criteria for restarting testing
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing & training needs
- Schedule
- Risks and contingencies
- Approvals

# Logic Coverage Criteria Subsumption



# References

- <http://java.sun.com/docs/books/tutorial/essential/concurrency/>
- <http://cs.gmu.edu/~offutt/softwaretest/>
- <http://msdn.microsoft.com/en-us/magazine/cc163744.aspx>
- Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.