Code Coverage Testing & Tool Support

W. Eric Wong
Department of Computer Science
The University of Texas at Dallas
ewong@utdallas.edu
http://www.utdallas.edu/~ewong

Speaker Biographical Sketch

• Professor & Director of International Outreach
  Department of Computer Science
  University of Texas at Dallas
• Guest Researcher
  Computer Security Division
  National Institute of Standards and Technology (NIST)
• Vice President, IEEE Reliability Society
• Secretary, ACM SIGAPP (Special Interest Group on Applied Computing)
• Principal Investigator, NSF TUES (Transforming Undergraduate Education in Science, Technology, Engineering and Mathematics) Project
  – Incorporating Software Testing into Multiple Computer Science and Software Engineering Undergraduate Courses
• Founder & Steering Committee co-Chair for the SERE conference
  (IEEE International Conference on Software Security and Reliability)
  (http://paris.utdallas.edu/sere13)
Our Focus

- We focus on testing programs
  - subsystems or complete systems
  - written in a formal language
  - a large collection of techniques and tools

Testing for Correctness?

- Identify the input domain of $P$
  - Input domain of a program $P$ is the set of all valid inputs that $P$ can expect
  - The size of an input domain is the number of elements in it
  - An input domain could be finite or infinite
  - Finite input domains might still be very large!

- Execute $P$ against each element of the input domain

- For each execution of $P$, check if $P$ generates the correct output as per its specification $S$
  - This form of testing is also known as exhaustive testing as we execute $P$ on all elements of the input domain.
Testing for Correctness? Sorry!

- For most programs exhaustive testing is not feasible
  - It will take several light years to execute a program on all inputs on the most powerful computers of today!
- What is the alternative?

Confidence in Your Program

- Confidence is a measure of one's belief in the correctness of the program.
- It is not measured in binary terms: a correct or an incorrect program.
- Instead, it is measured as the probability of correct operation of a program when used in various scenarios.
- It can be measured, for example, by test completeness
  - The extent to which a program has been tested and errors found have been removed.
How and why does testing improve our confidence in program correctness?

Example: Increase in Confidence

- We consider a non-programming example to illustrate what is meant by “increase in confidence.”
- Example: A rectangular field has been prepared with respect to certain specifications.
  - One item in the specifications is “There should be no stones remaining in the field.”
**Rectangular Field**

- Search for stones inside a rectangular field

![Diagram of a rectangular field with coordinates (0,0) and (L,0) on the x-axis, and (0,W) on the y-axis.](image)

**Testing the Rectangular Field**

- The field has been prepared and our task is to test it to make sure that it has no stones.

- *How should we organize our search?*
Partitioning the Field

- We divide the entire field into *smaller search rectangles*.
- The length and breadth of each search rectangle is *one half* that of the *smallest* stone.
**Input Domain**

- *Input domain* is the set of all possible inputs to the search process.
- In our example this is the set of all points in the field. Thus, the input domain is *infinite*!
- To reduce the size of the input domain we *partition the field into finite size rectangles*.

**Rectangle Size**

- The length and breadth of each search rectangle is one half that of the smallest stone.
- This ensures that each stone covers at least one rectangle.
- *Is this always true?*
**Constraints**

- Testing must be completed in less than $H$ hours
- Any stone found during testing is removed
- Upon completion of testing the probability of finding a stone must be less than $P$

**Number of Search Rectangles**

- Let
  - $L$: length of the field
  - $W$: width of the field
  - $\alpha$: length of the smallest stone
  - $\beta$: width of the smallest stone
- Size of each rectangle: $(\alpha/2) * (\beta/2)$
- Number of rectangles: $N = (L/\alpha)*(W/\beta)*4$
- Assume that $L/\alpha$ and $W/\beta$ are integers.
**Time to Test**

- Let \( t \) be the time to look inside one rectangle. Assume that *no rectangle is examined more than once*.
- Let \( o \) be the overhead in moving from one rectangle to another.
- Total time to search \( T = N \cdot t + (N - 1) \cdot o \)
- Testing with \( N \) rectangles is feasible only if \( T < H \)

**Partitioning the Input Domain**

- This set consists of all rectangles \( (N) \).
- Number of partitions of the input domain is finite \( (N) \).
- However, if \( T > H \) then the number of partitions is too large and scanning each rectangle once is infeasible.
- What should we do in such a situation?
**Option 1: Do a Limited Search**

- Of the \( N \) rectangles we examine only \( n \) where \( n \) is such that 
  \[ t \cdot n + o*(n - 1) < H. \]
- This limited search will satisfy the time constraint.
- Will it satisfy the probability constraint?

**Distribution of Stones**

- To satisfy the probability constraint we must scan enough rectangles so that the probability of finding a stone, after testing, is less than \( \varphi \).
- Let us assume that
  - there are \( s_i \) stones remaining after \( i \) test cycles.
  - There are \( N_i \) rectangles remaining after \( i \) test cycles.
  - Stones are distributed uniformly over the field
  - An estimate of the probability of finding a stone in a randomly selected remaining search rectangle is 
    \[ p_i = s_i / N_i. \]
Probability Constraint

- We will stop looking into rectangles if \( p_i \leq P \)
- Can we really apply this test method in practice?

Why Not

- Number of stones in the field is not known in advance.
- Hence we cannot compute the probability of finding a stone after a certain number of rectangles have been examined.
- The best we can do is to scan as many rectangles as we can and remove the stones found.
Coverage

- After a rectangle has been scanned for a stone, we say that the rectangle has been *covered*.
- Suppose that \( n \) rectangles have been scanned from a total of \( N \). Then we say that the coverage is \( n / N \).

Coverage and Confidence

- What happens when coverage increases?
  - *As coverage increases so does our confidence in a "stone-free" field*
- In this particular example, when the coverage reaches 100%, all stones have been found and removed.
- *Can you think of a situation when this might not be true?*
Option 2: Reduce Number of Partitions

- If the number of rectangles to scan is too large, we can increase the size of a rectangle.
  - This reduces the number of rectangles.

- Increasing the size of a rectangle also implies that there might be more than one stone within a rectangle.
  - Is it good for a tester?
  - It also implies . . . . . . . . . . . . . . . . . .

Rectangle Size

- As a stone may now be smaller than a rectangle, detecting a stone inside a rectangle (by examining only one point) is not guaranteed.

- Despite this fact our confidence in a “stone-free” field still increases with coverage.

- However, when the coverage reaches 100% we cannot guarantee a “stone-free” field.
**Coverage versus Confidence**

Does not imply that the field is “stone-free”.

**Rectangle Size**

\[ p = \text{Probability of detecting a stone inside a rectangle, given that the stone is there} \]

\[ t = \text{time to complete the testing} \]
**Analogy**

- Field: Program
- Stone: Error
- Scan a rectangle: Test program on one input
- Remove stone: Remove error
- Partition: Subset of input domain
- Size of stone: Size of an error
- Rectangle size: Size of a partition (wrt “Program”)

**Confidence and Probability**

- Increase in coverage increases our confidence in a “stone-free” field.
- It might not increase the probability that the field is “stone-free.”
Review Questions

• What is the effect of reducing the partition size on probability of finding errors?

• How does coverage affect our confidence in program correctness?

• Does 100% coverage imply that a program is fault-free?

• Indicate whether the following statements are true or false
  – The objective of software testing is to prove the correctness of the program being tested
  – The reliability of a program will always increase as your confidence of the program being correct increases

What is coverage and what role does it play in testing?
**Coverage Principle**

- The basic idea of coverage testing is that testing is complete when a well-defined set of tests is complete.
  - Example
    - Pilots use pre-flight check lists
    - Shoppers use grocery lists
to assure the correct completion of their tasks
  - In the same way testers can count the completed elements of a test plan
    - Example
      - Requirements
      - Functionalities
      - Blocks, Decisions (control-flow based)
      - C-uses, P-uses and All-Uses (dataflow-based)

---

**The Role of Coverage in Testing**

- It provides a way of monitoring and measuring the progress of testing against explicit quantitative completion criteria
  - Gives a clear measure of the completion of the testing task
  - Example, for requirements testing
    - How many of the requirements have been tested?
    - How many tests have run per requirement?
Topics

- Code Coverage testing and code inspection
- Code Coverage testing and functional testing
- Controlflow-based testing
- Dataflow-based testing

What is Code Coverage Testing

- It is “White Box Testing”
- Takes into account the structure of the software being tested
- Measures how thoroughly the code has been tested with respect to certain metrics
Code Coverage Testing versus Code Inspection

- Code inspection is a technique whereby the source code is inspected for possible errors
- Code coverage testing is a *dynamic* method, whereas code inspection is a *static* method
- Code coverage testing is a form of code inspection
  - Code that is executed successfully is disregarded for visual inspection
  - Code that is not executed is inspected
  - One is not likely to replace testing by code inspection

Code Coverage Testing versus Functional Testing

- When test inputs are generated using *program specifications*, we say that we are doing functional testing
  - Functional testing tests how well a program meets the *functionality requirements*
- These two types of testing are complementary
  - Basic functionalities should always be tested
  - The set of tests generated from functional testing provides a good basis for code coverage testing
History of Code Coverage Testing

- Using profiling tools to assess the amount of code coverage during testing (1960’s)
- Using tcov to give statement coverage data for C and Fortran programs (1970’s)
- Two groups of test criteria
  - Controlflow-based testing (block & decision)
  - Datallow-based testing (c-use, p-use and all-uses)

Basic Block

- A basic block is a sequence of consecutive statements or expressions, containing no branches except at the end, such that if one element of the sequence is executed all are.

A program, its control flowgraph, basic blocks, and decision
**Decision**

- A decision is a boolean predicate with two possible values, *true* and *false*.
**Importance of Code Coverage Testing**

- In general, a piece of code must be executed before a fault in it can be exposed.

- **Helps early fault detection**
  - Are system testers finding faults that should have been found and fixed by developers?
  - Relative cost of fixing a software fault

![Graph showing the importance of code coverage testing](image)

**State of Practice**

- A published study (ICSE'92)
  - Coverage above 60-70% in system testing is very difficult

- Don Knuth’s system testing of TeX (23,000 LOC)
  - 85% block and 72% decision coverage (1992)

- Brian Kernigan’s testing of AWK
  - 70% block and 59% decision coverage (1991)
**Efficient Coverage Testing (1)**

- How much code is currently tested?
  - Which statements were exercised?
  - Which paths were traversed?
  - Which def-use associations were exercised?
  - Which functions got invoked from where?
- Need help in creating tests?
  - Which statement should I try to cover next?

Analyzing the control flow graph of the program to find the dominant blocks, decisions, and def-use pairs.

For example, when a test covers highly dominant blocks it will cover many other blocks.

---

**Efficient Coverage Testing (2)**

<table>
<thead>
<tr>
<th>File</th>
<th>Blocks</th>
<th>Functions</th>
<th>DefUse</th>
</tr>
</thead>
<tbody>
<tr>
<td>file1</td>
<td>100</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>file2</td>
<td>150</td>
<td>75</td>
<td>37.5</td>
</tr>
</tbody>
</table>

Summary by file:

- file1: 100 blocks, 50 functions, 25 def-use pairs
- file2: 150 blocks, 75 functions, 37.5 def-use pairs

Summary by test case:

- Test case A: 50 blocks, 25 functions, 12.5 def-use pairs
- Test case B: 75 blocks, 37.5 functions, 18.75 def-use pairs

Block coverage summary by function over all files:

- Function 1: 50 of 50 blocks, 25 of 25 functions
- Function 2: 75 of 75 blocks, 37.5 of 37.5 functions

Block coverage summary by file:

- file1: 50 of 50 blocks, 25 of 25 functions
- file2: 75 of 75 blocks, 37.5 of 37.5 functions

---

Code Coverage Testing & Tool Support (© 2012 Professor W. Eric Wong, The University of Texas at Dallas)
Efficient Coverage Testing (3)

> Use prioritization and visualization to provide hot spots that give the most value in coverage.
> Each color represents a different weight determined by a control flow analysis using the concept of superblocks and dominators.

Code in white has already been covered by a test case and covering it again will not add new coverage.

Covers this red block guarantees the execution of at least 8 additional blocks.

Efficient Coverage Testing (4)

Covers either true or false branch guarantees the execution of at least another 8 branches.
**Dominator & Super Block (1)**

- A super block consists of one or more basic blocks that if one block in the super block is executed all are
  - If any statement in a super block is executed, then all statements in it must be executed, provided the execution terminates on that input
  - A super block needs not be contiguous
- Block \( u \) dominates block \( v \) if every path from entry to end, via \( v \), contains \( u \)
  - \( u \) dominates \( v \) if covering \( v \) implies the coverage of \( u \)
  - Test execution cannot reach \( v \) without going through \( u \)
- Given a program, identify a subset of super blocks whose coverage implies that of all super blocks and, in turn, that of all basic blocks

---

**Dominator & Super Block (2)**

An example C program

```
1;  
while (e2) {  
  switch (e3) {  
    case 1: e4;  
    case 2: e5;  
    default:  
      if (e6) {  
        e7;  
        continue;  
        do e10; while (e11);  
      }  
      e13;  
  }  
  e14;  
}
```

Control Flowgraph
**Dominator & Super Block (3)**

- Quiz: Does node 4 or node 12 predominate node 13? Why?
- Quiz: Does node 9 postdominate node 8? Why?

**Dominator & Super Block (4)**

- A strongly connected component of a basic block dominator graph has the property that every node in the component dominates all other nodes in that component.
**Dominator & Super Block (5)**

- Obtained by removing the composite edges in the right Figure on the previous slide.
- An edge $e$ from a node $u$ to a node $v$ is said to be a composite edge if $v$ is also reachable from $u$ without going through $e$.

**Dominator & Super Block (6)**

- At most four test cases need to be developed to cover all 14 basic blocks.
- The order in which the targeted basic blocks are covered and the corresponding cumulative coverages achieved.
- An alternative order is 10, 7, 4, and 9.
**Dominator & Super Block (7)**

- Experimental results

<table>
<thead>
<tr>
<th>program</th>
<th>basic blocks</th>
<th>blocks that need to be covered</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>455</td>
<td>138</td>
<td>30%</td>
</tr>
<tr>
<td>spiff</td>
<td>1266</td>
<td>361</td>
<td>29%</td>
</tr>
<tr>
<td>mgr</td>
<td>3848</td>
<td>1043</td>
<td>27%</td>
</tr>
<tr>
<td>ion</td>
<td>4886</td>
<td>1280</td>
<td>26%</td>
</tr>
<tr>
<td>atac</td>
<td>8737</td>
<td>2574</td>
<td>29%</td>
</tr>
<tr>
<td>odin</td>
<td>9870</td>
<td>2344</td>
<td>24%</td>
</tr>
<tr>
<td>xlib</td>
<td>15580</td>
<td>5111</td>
<td>33%</td>
</tr>
<tr>
<td>tvo</td>
<td>17680</td>
<td>6267</td>
<td>35%</td>
</tr>
</tbody>
</table>

---

**Weight Re-Computation (1)**

- The weight of a given node is the number of nodes that have not been covered but will be if that node is covered.

- Why is it important to take a “conservative” approach?
  - Will node 6 be covered by covering node 18?

- To arrive at node 18 requires the execution also go through nodes 1, 2, 4, 7, 12 and 13.
- Node 18 is dominated by nodes 1, 2, 4, 7, 12 and 13.
- These nodes will be covered (if they haven’t been) by a test execution if that execution covers node 18.
- Assuming none of the nodes is covered so far, we say that node 18 has a weight of 7 because covering it will increase the coverage by at least 7 additional nodes.
**Weight Re-Computation (2)**

* Arriving at node 6 requires the execution only goes through nodes 1, and 2
* Assuming none of the nodes is covered so far, we say that node 6 has a weight of 3

**Weight Re-Computation (3)**

* The execution of certain tests may change the weights of nodes that are not covered by these tests.

* After a test is executed to cover node 18, the weight (in terms of increasing the coverage) of node 6 is reduced from 3 to 1.
The **χSuds Tool Suite**

- Telcordia Technologies (formerly Bellcore or Bell Communications Research)
  - **χSuds** *(Software understanding and diagnosis systems)*: a set of software testing, analysis, and understanding tools for C and C++ programs
    - χATAC
    - χSlice
    - χRegress
    - χVue
    - χProf

**χSuds Home Page**

http://xsuds.argreenhouse.com
**KATAC Demo: Coverage Testing of C Code**

Compile code with KATAC

Initial display

KShds User's Manual

Source display after executing wordcount 1

Source display after executing wordcount 2

100% block coverage after executing wordcount 3

Source display after executing wordcount 6

100% block & decision coverage after executing wordcount 9

---

**Coverage Testing Tools for Java Code**

- eXVantage (eXtreme Visual-aid novel testing and generation)
  - A tool suite for code coverage prioritization, test generation, test execution, debugging, and performance profiling of Java, C, and C++ programs
  - Based on the JBT (Java Bytecode Testing) tool suite developed at UTD since 2002
- Clover
- Cobertura
- etc.
The End

Code Coverage Testing & Tool Support (© 2012 Professor W. Eric Wong, The University of Texas at Dallas)