Code Coverage Metrics

...And How to Use (and Misuse) Them

```c
#include <stdio.h>
main()
{
    int i, n, pp, p, c;
    printf("Enter element count: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("No %d series!\n", n);
        n = -1;
    } else {
        printf("{ ");
        pp = 0;
        p = 1;
        for (i=0; i < n; i++) {
            c = pp + p;
            printf("%d ", c);
            pp = p;
            p = c;
        }
        printf("} \n");
    }
    exit(n);
}
```
Code Coverage Metrics

- Code coverage metrics are tools for measuring the extent of white box test coverage.
- What types of white box coverage apply?
  - Statement, branch, condition, multicondition, multicondition decision, and loop coverage
  - McCabe basis paths
  - Set-use pair (dataflow) coverage
  - Checklist-based test requirements
- How does white box testing compare to other types of testing?
- Can tools help with the tedious bits?
- What bugs does white box testing miss?
Code Coverage and White Box Testing

- Dynamic test (compiled, executable)
  - Execute the item under test
  - Create test conditions (e.g., submit inputs)
  - Compare expected and actual results
- Pure requirements-based tests can miss 80% or more of the code: Do you know which 80% you are not testing?
- Test design based on how the system works
  - Figure out the control flows
  - Figure out data flows
  - Check against standard checklists or bug taxonomies
- Code coverage tools monitor the completeness of white box tests and handle the tedious details for you
- Let’s look at these in more detail
Code-Based Testing

Creating tests to achieve levels of coverage

- **Statement coverage**: every statement executed
- **Branch coverage**: every branch (decision) taken each way, true and false
- **Condition coverage**: every condition evaluated once true and once false
- **Multicondition coverage**: for compound conditions, every possible combination of constituent conditions covered
- **Multicondition decision coverage**: for compound conditions, every possible combination of constituent conditions that can affect the decision covered
- **Loop coverage**: Ideally all possible loop paths taken, but usually loop paths taken zero, once, and multiple (ideally, maximum) times
**Code Coverage Example**

What test values for n do we need to cover all the statements?
- n < 0, n > 0

Does that get us branch coverage?
- No, need to add n = 0

Does that get us condition coverage?
- Yes: no compound conditions

How about loop coverage?
- Need to cover n = 1 and n = max

```c
main()
{
    int i, n, pp, p, c;
    printf("Enter element count: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("No %d series!\n", n);
        n = -1;
    } else {
        printf("{ ");
        pp = 0;
        p = 1;
        for (i=0; i < n; i++) {
            c = pp + p;
            printf("%d ", c);
            pp = p;
            p = c;
        }
        printf("} \n");
    }
    exit(n);
}
```
Variations on Condition Coverage

- So, when do condition coverage, multicondition coverage, and multicondition decision coverage vary?
- How many tests for each type of coverage for lines 1-5?
- How about for lines 7-11?
- Does multicondition coverage even make sense for lines 7-11?
- Let’s compare the testing for lines 1-5 with lines 13-17

```c
1 if ((a>0) && (b>a)) {
2   printf("Success.\n");
3 } else {
4   printf("Failure.\n");
5 }
6
7 if ((f!=NULL) && (*f==0)) {
8   printf("Found.\n");
9 } else {
10  printf("Not found.\n");
11 }
12
13 if ((a<=0) || (b<=a)) {
14   printf("Failure.\n");
15 } else {
16   printf("Success.\n");
17 }
```
McCabe Cyclomatic Complexity

- McCabe’s Cyclomatic Complexity measures control flow complexity
  - Measured by drawing a directed graph
  - Nodes represent entries, exits, decisions
  - Edges represent non-branching statements
- It has some useful testing implications
  - High-complexity modules are inherently buggy and regression-prone
  - The number of basis paths through the graph is equal to the number of basis tests to cover the graph
- Let’s see how…
Cyclomatic Complexity for Fibonacci

Cyclomatic Complexity

\[ C = \#\text{Regions} + 1 \]
\[ = 2 + 1 = 3 \]

\[ C = \#\text{Edges} - \#\text{Nodes} + 2 \]
\[ = 5 - 4 + 2 = 3 \]

```
main()
{
    int i, n, pp, p, c;
    printf("Enter element count: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("No %d series!\n", n);
        n = -1;
    } else {
        printf("{ ");
        pp = 0;
        p = 1;
        for (i=0; i < n; i++) {
            c = pp + p;
            printf("%d ", c);
            pp = p;
            p = c;
        }
        printf("} \n");
    }
    exit(n);
}
```
Basis Paths and Tests

Basis Paths
1. 124
2. 1234
3. 12334

Basis Tests
1. n = -1
2. n = 0
3. n = 1

main()
{
    int i, n, pp, p, c;
    printf("Enter element count: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("No \ series!\n", n);
        n = -1;
    } else {
        printf("\n");
        pp = 0;
        p = 1;
        for (i=0; i < n; i++) {
            c = pp + p;
            printf("%d \n", c);
            pp = p;
            p = c;
        }
        printf("\n");
    } exit(n);
Analyzing Data Flow Set-Use Pairs

- So far, we’ve focused on testing control flows
- How about testing data flows?
- We can analyze data flows using set-use pair coverage
- Do we cover every variable set-and-subsequent-use (assigning the variable a value, then using that variable, but before assigning the variable again)?
Data Flow Coverage Example

- i: (13,13)
  - n=0 covers assign-use
  - n=1 covers increment-use

- n: (5,6), (5,13), (5,21), (8,21)
  - n=-1 covers (5,6), (8,21)
  - n=0 adds (5,13), (5,21)

- pp: (11,14), (16,14)
  - n=1 covers (11,14)
  - n=max adds (16,14)

- p: (12,14), (12,16), (17,14)
  - n=1 covers (12,14)
  - n=max adds (12,16), (17,14)

- c: (14,15), (14,17)
  - n=1 covers (14,15)
  - n=max adds (14,17)

- Basis cover != data-flow cover
  - For c into p into pp into c, iterate loop three times

```
1 main()
2 {
3   int i, n, pp, p, c;
4   printf("Enter element count: ");
5   scanf("%d", &n);
6   if (n < 0) {
7     printf("No %d series!\n", n);
8     n = -1;
9   } else {
10     printf("{ ");
11     pp = 0;
12     p = 1;
13     for (i=0; i < n; i++) {
14       c = pp + p;
15       printf("%d ", c);
16       pp = p;
17       p = c;
18     }
19     printf("} \n");
20   }
21   exit(n);
22 }
```
Checklist-based Test Requirements

Tests can be designed based on a catalog of test requirements, a bug taxonomy or other checklists of important aspects to be tested.

Frequent program operations combined with typical buggy aspects of those operations make a good catalog of test requirements.

Select test requirements; cover with tests.

Let’s test Fibonacci with Brian Marick’s catalog, from The Craft of Software Testing.

Click here to see the test requirements catalog, from Marick’s book

Click here to see the tests developed using the test requirements catalog
A Basic Catalog-Based Test Design Process

- Review requirements, design, code, and data structures
  - Compare each element to checklist or catalog
  - Accumulate clues and test requirements
- Design tests to cover the test requirements
  - Avoid covering multiple error-related test requirements in one test where possible
  - Seek as much test variety as possible
- Run tests
## Comparing Techniques

<table>
<thead>
<tr>
<th></th>
<th>Static</th>
<th>White box</th>
<th>Black box</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>What</strong></td>
<td>Test without running</td>
<td>Test how system works</td>
<td>Test what system does</td>
</tr>
<tr>
<td><strong>Why</strong></td>
<td>Identify bugs before they’re built</td>
<td>Identify bugs in functions or interfaces</td>
<td>Identify bugs in system results and behavior</td>
</tr>
<tr>
<td><strong>Who</strong></td>
<td>All stakeholders</td>
<td>Typically developers</td>
<td>Typically testers</td>
</tr>
<tr>
<td><strong>Phase</strong></td>
<td>During specification and development</td>
<td>Usually unit, component, integration</td>
<td>Usually integration, system, and acceptance</td>
</tr>
<tr>
<td><strong>Tools</strong></td>
<td>Simulators, code analyzers, spell- and grammar checkers, graphic/diagramming, spreadsheets</td>
<td>Profilers, coverage analyzers, harnesses, debuggers, data generators, test case generators</td>
<td>Intrusive and non-intrusive GUI tools, load generators, performance tools, data generators</td>
</tr>
</tbody>
</table>

Tools, data, and cases can be shared: encourage cross-pollination of techniques
What a Tedious Chore!

- Trying to measure metrics like control-flow coverage and Cyclomatic Complexity is not a fun pastime
- Fortunately, there are tools that can help
- It’s harder to find data-flow coverage tools
- Anyone have a test catalog coverage tool?
What Can’t White Box Tests Find?

- As essential as it is to have well-tested units and builds, white box testing can’t find every kind of bug
- Bugs of omission
  - Missing requirements, design elements, code
  - Use requirements, design and code reviews
- Unmaintainable code
  - Inscrutable variable names or constructs, etc. (though McCabe complexity helps)
  - Use code reviews and static analysis tools
- Systemic problems
  - Performance, security, usability, interoperability, etc.
  - Use code reviews, black-box system testing, beta/pilot tests, usability studies, prototyping, etc.
Conclusions

● White box testing is…
  ▪ dynamic testing (running item under test)…
  ▪ based on knowledge of how it works

● There are four main theoretical techniques
  ▪ Statement, branch, condition, and loop coverage
  ▪ Basis paths
  ▪ Set-use pair (dataflow) coverage
  ▪ Test requirements catalogs

● Tools are very helpful in measuring the coverage of white box tests

● White box testing is complementary to…
  ▪ black box testing
  ▪ static testing

● Use all three to form a complete testing strategy
For over a dozen years, RBCS has delivered services in consulting, outsourcing and training for software and hardware testing. Employing the industry’s most experienced and recognized consultants, RBCS conducts product testing, builds and improves testing groups and hires testing staff for hundreds of clients worldwide. Ranging from Fortune 20 companies to start-ups, RBCS clients save time and money through improved product development, decreased tech support calls, improved corporate reputation and more. To learn more about RBCS, visit www.rbcus.com.

Address: RBCS, Inc.
31520 Beck Road
Bulverde, TX 78163-3911
USA

Phone: +1 (830) 438-4830
Fax: +1 (830) 438-4831
E-mail: info@rbcu.com
Web: www.rbcus.com