Data Flow Coverage 2

Stuart Anderson
Coverage: the point, revisited

- We are attempting to decide what makes a good test. i.e judge the adequacy of our test suite.
- Surely an adequate test suite will show our software is correct? Impossible. Same as proving the software is correct.
- So can we say some test suites are better than others? Yes, if we can define effective, testable adequacy criteria. Such as?
  - Statement coverage = 1
    But if our test doesn’t exercise all statements, surely it’s no good?
  - Branch coverage = 1
    But if our test doesn’t exercise all branches, surely it’s no good?
  - Path coverage = 1
    But if our test doesn’t exercise all paths, surely it’s no good? (!)
- So they are actually really inadequacy criteria
Subsumption

- So really, no tests are as good as we’d want. But some are provably worse than others, e.g. branch coverage necessarily includes statement coverage.

- **Definition:** *test coverage criterion A subsumes test coverage criterion B if and only if, for every program P, every test set satisfying A with respect to P also satisfies B with respect to P.*

- If you have branch coverage, you also always have statement coverage — Branch coverage subsumes statement coverage.

- If criterion A subsumes criterion B, and a test suite satisfying B is guaranteed to find a fault, then a suite satisfying A will also find that fault.
  - But these criteria provide no guarantees.
  - And with no guarantee that B will find a fault, we have no guarantee for A either.
Adequacy review 1

- **Statement adequacy:** all statements have been executed by at least one test case.

- **Branch adequacy:** all branches have been executed by at least one test case.

- **Basic condition adequacy:** each basic condition evaluates to true in at least one test case, and to false in at least one test case.

- **Compound condition adequacy (simplistic definition):** each combination of truth values of basic conditions must be visited by at least one test case.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>(X&amp;Y)</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Good definitions are important: basic condition

- \((X=Y=Z=F)\); \((X=Y=Z=T)\) appears to achieve B.C.A., but condition \(Y\) is never evaluated in the first case, nor \(Z\) in the second.

- Need, e.g. \((X=F, Y=?, Z=T)\); \((X=T, Y=Z=F)\); \((X=Y=T, Z=?)\) \((?=\text{don't care, because it's never evaluated})\).
Exercise 5

Test suite adequacy 1

- $T_0 = \{ "", "test", "test+case\%1Dadequacy" \}$
- $T_1 = \{ "adequate+test\%0Dexecution" \}$
- $T_2 = \{ "\%3D", "\%A", "a+b", "test" \}$
- $T_3 = \{ "\", "+%0D+%4J" \}$
- $T_4 = \{ "first+test\%9Ktest\%K9" \}$

<table>
<thead>
<tr>
<th>Coverage Criterion</th>
<th>T0</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic Condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compound Condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[P&Y p.213-214, Figures 12.1 & 12.2]
Test suite adequacy 1

- T2 uncovers a bug in the program. What bug?
- Branch coverage appears the same as statement coverage here. Suggest a code construct which would show branch coverage to be superior to statement coverage.
- Basic condition coverage clearly doesn’t subsume branch coverage.
- While T4 technically satisfies basic condition coverage, you can argue that it doesn’t. How?
- You can also argue that compound condition coverage is impossible for this code fragment, for a similar reason. This might lead us to modify our definitions of basic and compound condition coverage, to make them more practical. How?
- Can you suggest enhancements to each test in order to achieve compound condition coverage?
Adequacy review 2

- Test suite $T$ satisfies the path adequacy criterion for program $P$ iff for each path $p$ of $P$ there exists at least one test case in $T$ that causes the execution of $p$.

- **Loop boundary adequacy criterion:** test cases exist such that each loop is executed zero times, exactly once, and many times.

Some common sense necessary in application: Some loops have a fixed number of iterations. How many is ‘many’?
Test suite adequacy 2

- This routine loops through elements 0 to n-1 of array A (stopping if it finds an element that’s greater than or equal to X). As it does so, it replaces any negative entries in A with their absolute (positive) value.

- Generate a test suite (in the form of some suggested values for array A, e.g. [1, 2], [3, 4]) which satisfies the path adequacy criterion for this program. Assume $n = |A|$.

- Generate a test suite which satisfies the loop boundary adequacy criterion.
Test suite adequacy 2

• Path adequacy is impossible, even for this trivial example!

• Consider the below code fragment. On the surface there are four paths through it, but a little attention makes it clear that no test suite could ever exercise one of those paths:

```java
if(a < 0)
    a = 0;
if(a > 10)
    a = 10;
```

• So, realistically, we must settle for less than 100% coverage.
Adequacy review 3: data flow basics

- Data flow criteria are concerned with **definition-clear paths** from definition to use of individual variables.
- Context is a graph representation of the program, with vertices being basic blocks.
- A **definition-use pair (DU pair)** is a pairing of definition and use of a variable, with at least one def-clear path between them (there could be many).
- \(dcu(x, v)\) is the set of vertices \(v'\) which use variable \(x\) in **computations**, and could be directly affected by a definition of \(x\) at \(v\) (i.e. there is a def-clear path from \(v\) to \(v'\)).
- \(dpu(x, v)\) is the set of edges \((v', v'')\) which use variable \(x\) in their predicates (conditions/branches), and could be directly affected by a definition of \(x\) at \(v\) (i.e. there is a def-clear path from \(v\) to \(v'\)).
Exercise

Data flow basics

- Identify DU pairs for $c$ (your answer will be a list of pairs of line numbers).
- Identify DU pairs for digit_high.
- Identify the def-predicate uses in your answers.
- Identify the def-computation uses in your answers.
- What is $\text{dcu(ok,34)}$?
- What is $\text{dpu(ok,20)}$?
- What is $\text{dpu(digit\_high, 30)}$?

```c
-17: int cgi_decode(char *encoded, char *decoded) {
-18:     char *eptr = encoded;
-19:     char *dptr = decoded;
*20:     int ok=0;
*21:     while (*eptr) {
*22:         char c;
*23:         c = *eptr;
-24:         /* Case 1: '+' maps to blank */
*25:         if (c == '+') {
*26:             *dptr = ' ';  
*27:         } else if (c == '%') {
*28:             /* Case 2: '%xx' is hex for character xx */
*29:             int digit_high = Hex_Values[*++eptr];
*30:             int digit_low = Hex_Values[*++eptr];
*31:             if ( digit_high == -1 || digit_low == -1 ) {
*32:                 /* *dptr='?'; */
*33:                 ok=1; /* Bad return code */
*34:             } else {
*35:                 *dptr = 16* digit_high + digit_low;
*36:             }
*37:         } else {*38:             *dptr = *eptr;
*39:         }
*40:         ++dptr;
*41:         ++eptr;
-42:     }/* Case 3: All other chars map to themselves */
*43:     *dptr = *eptr;
*44: }
*45: }++dptr;
*46: *dptr = '\0'; /* Null terminator for string */
*47: return ok;
*48: }
```
Adequacy review 4: data flow criteria

- **All-defs** requires that test T exercises each definition in program P at least once. This means not just executing the definition, but using its result in at least one computation or predicate.

- **All-p-uses** requires exercise of all DU pairs culminating in predicates. Note pairs, not paths: only one def-clear path needed per DU pair.

- **All-c-uses** requires exercise of all DU pairs culminating in computations. Note pairs, not paths.

- **All-p-uses/some-c-uses** and **all-c-uses/some-p-uses** expand the above two by requiring that all-defs hold as well.

- **All-uses** requires that both all-p-uses and all-c-uses hold.

- **All-du-paths** expands on all-uses by requiring that all def-clear paths between each DU pair are exercised, modulo loops.
Data flow criteria

- Suggest a set of path(s) which satisfy all-defs.

- Suggest a set of path(s) which satisfy all-c-uses.

- Suggest a set of path(s) which satisfy all-du-paths.
All-Defs Coverage Criterion

- We require to use all definitions.
- Here we assume we only use the variable x.
- We require to use each def.
- So the path A,B,D,F is OK.
- Suppose we defined a variable y in C and used it in E what would be a suitable test set?

```
defs(A) = \{x, \ldots\}
c-use(C) = \{x, \ldots\}
c-use(B) = \{x, \ldots\}
c-use(E) = \{x, \ldots\}
```
All-Uses Coverage Criterion

- We need to ensure we exercise every use.

- So we need the set of test paths to include:
  - A to B
  - A to C
  - A to E

- So a satisfactory test set is:
  - A, B, D, F
  - A, C, D, E, F
All DU-paths Coverage Criterion

- Here we need to consider all loop-free paths between A and vertices that use x.

- So we need to include:
  - A,B
  - A,C
  - A,B,D,E
  - A,C,D,E

- So the following test set satisfies the coverage criterion:
  - A,B,D,E,F
  - A,C,D,E,F
More Complex Data Flow Criteria

- Ntafos proposed a generalisation of the original data-flow criteria to allow iteration of definition/use chains

- Foundation:
  - Chains of alternating definitions and uses linked by definition-clear subpaths (k-dr interactions)
  - $i^{th}$ definition reaches $i^{th}$ use,
  - which defines $i^{th}+1$ definition
  - $k$ is number of iterations
k-dr Interactions

\[ x_1 = \ldots \]
\[ x_2 = \ldots x_1 \ldots \]
\[ x_3 = \ldots x_2 \ldots \]
\[ x_k = \ldots x_{k-1} \ldots \]

Def-clear for \( x_1 \)

Def-clear for \( x_2 \)

Def-clear for \( x_{k-1} \)
Wrapping up

• So we can argue that certain criteria are less bad than others. Where does this get us?

• Not terribly far unfortunately: most of the theoretical research seems to indicate you cannot conclude much about test effectiveness from your adequacy criteria.

• But there is empirical evidence that at very high levels of coverage, stronger criteria are worth pursuing.

• It does not seem surprising though that writing ten times as many tests in order to satisfy a stronger criterion gives you better results. The question then is whether these extra criterion-driven tests are better than extra random ones.

• Research now seems to be heading in this more empirical direction, rather than focusing on theoretical adequacy comparisons.
Readings

Required Readings

- **Textbook (Pezzè and Young):** Chapter 9, Test Case Selection and Adequacy