SENG3320 – Notes

Week Three

Bryce Tuppurainen

Basis Path Testing – Path Coverage Protocol for White-Box Testing

Testing mechanism proposed by McCabe and is about performing independent passes through a program execution based on a provided test case input. It’s a mixture of Branch and Path coverage for white box analysis of a program that has been converted into a CFG (Control Flow Graph)

An Independent Path – Useful for getting around having to test loops

Is a path that is independent from all other paths, only if it includes at least some edge or vertices that are not included in the other paths

Cyclomatic Complexity

A programs complexity can be measured by the number of independent paths for a CFG. The function to determine this metric from Graph Theory is:

Max number of cases required to guarantee basis path coverage

Where e is the number of edges, n is the number of nodes, and p is the number of connected components (for a single component p = 1).

A Simplified Cyclomatic Complexity Calculation

Max number of cases required to guarantee basis path coverage

Where d is the number of predicate notes, i.e. loops, if-statements, switch-statements that exist in the graph, really anything that has more than two edges connected to it. Each of the connected notes counts for one decision point. Remember that it is not the point of the decision, it’s the number of possible extra outcomes from that note.

McCabe’s Basis Path Testing

Four stage test which is less effective than a full path coverage test case set, but more effective than simply branch testing. Firstly, you **generate a CFG**, then use one of the functions above which are equivalent to **determine the cyclomatic complexity** and then finally **design a basis set of linearly independent paths** and **generate test cases to achieve all these paths**.

Cyclomatic Complexity provides the UPPER BOUND for the number of test cases required to provide full basis path coverage. It is often possible to create a test case that covers a more extensive path to use less test cases to achieve full basis path coverage.

Another Way to Think About BASIS PATH COVERAGE

All that you’re doing is just iterating through all the if statements as if they were an individual path for each outcome they run down, so in many situations you’re just counting up in binary. REMEMBER to note that in a switch case if you had three arguments that is the equivalent of adding two predicate nodes, if you had four, then three predicate nodes. i.e., for three if statements that are independent of one another (don’t worry about the test cases required to do this and whether they exist) you can have each be either true, or false, therefore 2 x 2 x 2 - 2 = 2 ^ (3-1) = 4 is the cyclomatic complexity of this program.

Decision Coverage (also known as Branch Decision Coverage)

By contrast to branch coverage, we don’t care about how many inputs there are into the if-statement when designing our tests, for an AND, OR, XOR and their inverses, we just look at whether or not the if statement itself is true. i.e. we don’t care about internalised predicates, only the predicate as a whole.

Condition Coverage – Will always be greater than zero

Compliments decision coverage, in this case we do care about the conditions in the expression. We just want to make sure that every single predicate that exists in the ENTIRE program has both its true and false situations covered by the test cases.

C/DC Coverage

C/DC or C&DC Literally just means condition and decision coverage, we want to find the minimum number of test cases required to provide the highest possible Condition Coverage and Decision Coverage for the program’s execution.

A little trick for Boolean Logic – The minimum number of cases required to provide Conditional Coverage is always the all predicates true, and then the all predicates false, similarly for Decision, you do the same thing but for the predicates as a whole instead.

MCC or Multiple Condition Coverage

Even stronger than C/DC it is literally every single possible combination of true-false for every single condition. So it’s just 2c where c is the number of conditions. So if you’ve got three conditions you get 8 possible combinations:

False, False, False - 000

False, False, True - 001

False, True, False - 010

False, True, True - 011

True, False, False - 100

True, False, True - 101

True, True, False - 110

True, True, True - 111

This is the denominator when determining how well covered as a percentage a set of test cases are.