SENG3320 – Notes

Week Four

Bryce Tuppurainen

Complexity of MCC (Multiple Condition Coverage)

The time and space complexity, i.e. number of required unique test cases, of MCC is the same in both time and space, and is 2^N; where N is the number of conditions. Similarly, this applies to BDC but where N is the number of total decisions.

Modified Condition/Decision Coverage (MC/DC)

Better time complexity coverage consideration, we just look for “***Important Combinations***”. i.e. An Important Combination is just when each basic condition indepently affects the outcome of the decision. Complexity is (2^N – M); Where N is the number of conditions, and M is the number of combinations that are not important, there are **four** rules to **MEMORISE** for this:

1. **Each entry and exit point is invoked** – if there are multiple returns out of a function or an object has multiple methods that call the function (**Note, this means that we may not be dealing with just pure functions**)
2. **Each decision takes every possible outcome** – Decision coverage must be 100%
3. **Each condition in a decision takes every possible outcome** – Condition coverage must be 100%
4. **Each condition in a decision is shown to independently affect the outcome of the decision** – For example, in an AND, as soon as some

Condition Coverage Comparison

**Statement <= Decision <= Condition <= CDC <= MC/DC <= MCC**

Data Flow Coverage

Variable Definition

A program variable is **defined** whenever its **value is modified** (i.e. L-Values), whenever it is **changed in an input (user/file/IO input) statement** or **used as a reference** in a subroutine

Variable Use

R-Values, if you don’t reassign the value of the variable, or pass it as a reference to a subroutine, then it is being *used* NOT *defined*

* This can be a **P-Use** where it is used in a **comparison** or predicate (x <= 65) ? 1 : 1
* Or a **C-Use** where it is being used for **computation**, for example y += x or print(x), as well as in the return statement

Note that is is possible to do both at the same time, such as the L-Value in an increment

DU-Pair or Definition-Use Pair

In DU-Pair we’re concerned as to whether, there’s any possible changes on an execution path between the original definition of the variable and when it is used. So there can be many du-pair in a subroutine, start with the shortest from the beginning and then work your way down (this is one effective method to do so). Note this is always a pair, not a path, a DU-Pair can have many paths consumed by that pair.

* Typically, this is written as (1, 5) – Where 1 and 5 are the definition and use nodes/statements in the CFG, note that the du-pair should be specified with the path/s (there can be multiple def-clear paths) where the value is not modified if it is def-clear (not modified/defined before use). This is for **C-Use (,)**
* Otherwise, you specify the possible branches in < > where the defined value is used for a decision. Such as (1, <5, 6>) and (1, <5, 8>) where 5 is the number of the node where the defined variable is used as a **P-Use (,<>)**
* REMEMBER TO START FROM EVERYWHERE THAT EACH VARIABLE CAN BE MODIFIED / DEFINED / REDEFINED
  + Unless you’re looking at All-Def/s coverage, then it is only one def-clear path from each definition

All-Def/s Coverage

* For every variable definition in the program for every variable, use at least one pair (C-Use or P-Use) for every variable definition mapped in this list.

Subsumed Paths

Look at the pairs that occur in the original path, you just need to ensure that the whole path exists sequentially in the super-path to be subsumed, NOT just that the nodes exist in the other path.