Lec 6 Validation and Verification

Objectives

- [ ] White-box testing techniques
    - [ ] Branch conditions testing
    - [ ] Definition usage testing
- [ ] Fault-based testing
    - [ ] Fuzz testing
    - [ ] Mutation testing

Branch condition Testing (White-box testing)

- Test coverage items
    - “Possible Boolean values (i.e. true or false) of the conditions within decisions and the decision outcomes from each decision”

Exercise

How many tests are required for a 100% branch condition coverage?

```
#include <stdbool.h>
typedef enum direction_e { GO_LEFT, GO_RIGHT, STOP} direction_t;


void update_direction(bool sensor_left, bool sensor_right,direction_t * direction) {
  if(*direction==STOP || (sensor_left && sensor_right))
    *direction = STOP;
  else {
    if(sensor_left)
      *direction = GO_RIGHT;
    else if(sensor_right)
      *direction = GO_LEFT;
    else if(*direction == GO_LEFT)
      *direction = GO_RIGHT;
    else if(*direction == GO_RIGHT)
      *direction = GO_LEFT;
  }
}
```

- if(*direction==STOP || (sensor_left && sensor_right)
    - If A OR (B AND C)
    - A= True, B = whatever, C =whatever → True
    - A = False, B = True, C = False → False
    - A = False, B = True, C = True → True
    - A= False, B = False, C = True → False
    - **4 Test cases**
- if(sensor_left)
    - **2 Test cases**
- else if (sensor_right)
    - **2 Test cases**
- else if (* direction == GO_LEFT)
    - **2 Test cases**
- else if (*direction == GO_RIGHT)
    - **2 Test cases**

100% Branch condition coverage = 4 + 2 + 2 + 2 + 2 = 12

Data Flow Testing
Test model: definitions/use of variables

- **“Definitions”** → where a variable is possibly given a new value
- A “use” is an occurrence of a variable in which the variable is not given a new value

All-use (All def-use) Testing

- **Test coverage items:** A set of control flow sub-paths from each variable definition to every use of that definition (with no intervening definitions)

1. Identify a control flow sub-path from a variable definition to a subsequent use of that definition that has not yet been executed during testing
2. Determine the test inputs that will cause the control flow sub-path from the identified definition to be exercised
3. Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis
4. Repeat 1. to 2. Until the required level of test coverage is achieved

Exercise 1

```
#include <stdbool.h>
typedef enum direction_e { GO_LEFT, GO_RIGHT, STOP} direction_t;

1 void update_direction(bool sensor_left, bool sensor_right,direction_t * direction) {
2   if(*direction==STOP || (sensor_left && sensor_right))
3     *direction = STOP;
    else {
4     if(sensor_left)
5       *direction = GO_RIGHT;
6     else if(sensor_right)
7       *direction = GO_LEFT;
8     else if(*direction == GO_LEFT)
9       *direction = GO_RIGHT;
10    else if(*direction == GO_RIGHT)
11      *direction = GO_LEFT;
    }
}
```

It is assumed that path 1 is when the parameters are defined.

Variable	**Def-Use Pairs**
*direction	(1,2), (1,8), (1,10)
sensor_left	(1,2), (1,4)
sensor_right	(1,2), (1,6)

Test	**Def-Use Pair Coverage (def, use, variable)**
*direction=STOP, sensor_left=0, sensor_right =1	(1,2,*direction)
*direction=GO_LEFT, sensor_left=0, sensor_right =1	(1,8,*direction)
*direction=GO_RIGHT, sensor_left=0, sensor_right =1	(1,10,*direction)
*direction=GO_RIGHT, sensor_left=1, sensor_right =1	(1,2, sensor_left), (1,2,sensor_right)
*direction=GO_RIGHT, sensor_left=1, sensor_right =0	(1,4, sensor_left)

*direction=GO_RIGHT, sensor_left=0, sensor_right =1	(1,6, sensor_right)

Total test cases for 100% All-use = 6 (Confirmed with Lecturer that the correct answer is 6 not 7)

Exercise 2

```
1.  // gcd() method, returns the GCD of a and b
2.  static int gcd(int a, int b)
3.  {
4.    // stores minimum(a, b)
5.    int i;
6.    if (a < b)
7.      i = a;
8.    else
9.      i = b;
10.
11.   // take a loop iterating through smaller number to 1
12.   for (i = i; i > 1; i--) {
13.
14.     // check if the current value of i divides both
15.     // numbers with remainder 0 if yes, then i is
16.     // the GCD of a and b
17.     if (a % i == 0 && b % i == 0)
18.       return i;
19.   }
20.
21.   // if there are no common factors for a and b other
22.   // than 1, then GCD of a and b is 1
23.   return 1;
24. }
```

Variable	**Def-Use Pairs**
int a	(1,6), (1,7), (1,17)
int b	(1,6), (1,9), (1,17)
int i	(7,12), (9,12), (12, 17), (12, 18), (12, 12)

Test	**Def-Use Pair Coverage (def, use, variable)**
a=5, b=28	(1,6,a), (1,7,a), (1,17,b), (7,12,i), (12,17,i), (12,12,i), (12,18,i)
a=28, b=5	(1,6,b), (1,9,b), (1,17,b), (9,12,i), (12,17,i), (12,12,i), (12,18,i)

Total Test cases for 100% All-use = 2

All-du-paths testing

- Test Coverage Items: **The set of all loop-free** control flow sub-paths from each variable definition to every use of that definition (**with no intervening definitions**)

1) Identify a control flow sub-path from a variable definition to a subsequent p-use or c-use of that definition that has not yet been executed during testing.

    - p-use → def-use pairs that covers conditional statements (e.g. if, else)
    - c-use → used in expression statement → value computed or manipulated
2) Determine the test inputs that will cause the control flow sub-path from the identified definition to the subsequent p-use or c-use to be exercised
3) Determine the expected result from exercising the control flow sub-path by applying the test inputs to the test basis
4) Repeat steps 1) to 3) until the required level of test coverage is achieved.

https://www.youtube.com/watch?v=0-YzbA_9Ogs
https://www.inf.ed.ac.uk/teaching/courses/st/2017-18/tutorial3-solutions.html

---

Faults-based testing → push software to failure
Checking Software Robustness:
- Boundary value analysis
- Fuzz Testing

Checking faults caught by test suite
- Mutation Testing

Fuzz testing
- **Objective** → improve robustness of software product
    - Push product to the limits and observer behaviour
    - Uncover unexpected behaviour
- **Method** → feed the software with unexpected/malformed/invalid inputs

- **Challenges** → produce lots of non-significant bugs, resource/analysis intensive

Mutation testing
- **Objective** → Assess test suite quality
- **Method** →injects faults in the source code, test
- **Metric:**
    - Each mutated version is called a *mutant* and tests detect and reject mutants by causing the behaviour of the original version to differ from the mutant. This is called *killing* the mutant
    - Killed mutants/ (killed mutants + surviving mutants)

- **Challenges**
    - Huge “fault injection space”
    - Requires tooling
    - Requires criteria to decide if new test cases must be added or not

---

V&V certification
Testing certification
- International Software Testing Qualifications Board (ISTQB)
    - 3 levels: Foundation, Expert, ADvanceds
    - Teaches testing techniques, from foundations to very advanced topics

Warnings:
- Standards/certification do not imply quality
- Standards should not simply be used by themselves