Principles of Software Testing

Spring, 2023

Yi Xiang xiangyi@scut.edu.cn

Contents

- Static and Dynamic Verification
- Black and White Box Testing
- Test Artefacts

1. Static and Dynamic Verification

- Static Verification does not require the execution of the software code, while Dynamic Verification does.

- Static Verification (or Static Analysis) can be as straightforward as having someone of training and experience reading through the code to search for faults.

- Dynamic Verification (or Software Testing) confirms the operation of a program by executing it.

Static Verification

- It could also take a mathematical approach consisting of symbolic execution of the program

- It could be a formal approach consisting of symbolic verification of the translation between the specification and the source code

```
1   int twice (int v) {
2           return 2*v;
3   }
4
5   void testme (int x, int y) {
6           z = twice (y);
7           if (z == x) {
8                   if (x > y+10)
9                           ERROR;
10                  }
11          }
12  }
13
14  /* simple driver exercising testme() with sym inputs */
15  int main() {
16          x = sym_input();
17          y = sym_input();
18          testme(x, y);
19          return 0;
20  }
```

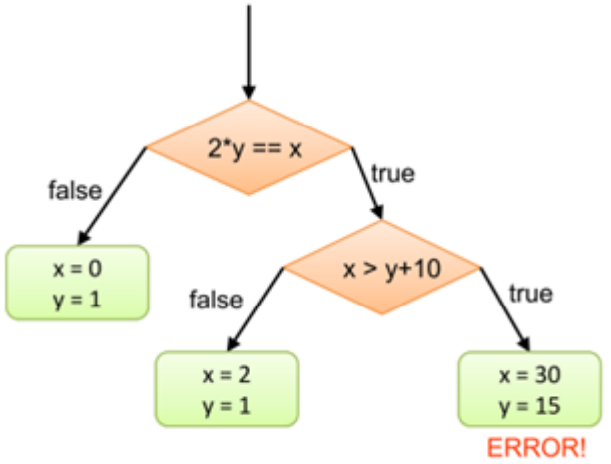


Figure 2. Execution tree for the example in Figure 1.

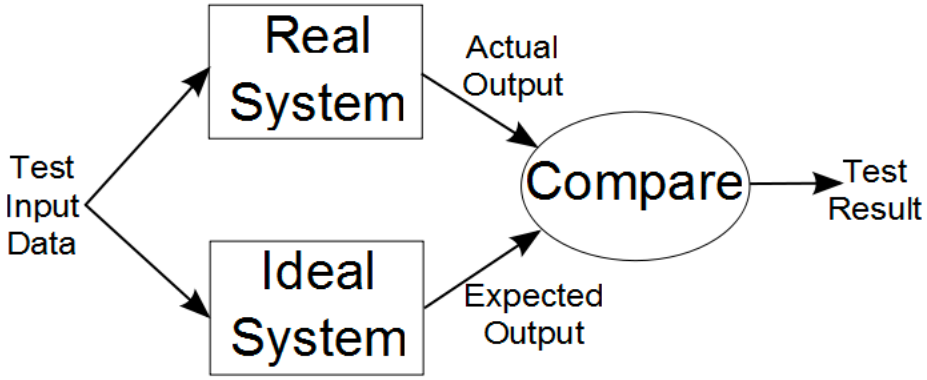
KLEE工具：https://github.com/klee/klee

Dynamic Verification

- Test Cases are created that guide the selection of suitable Test Data (consisting of Input values and Expected Output values

- The *Input values* are provided as inputs to the program during execution

- The *Actual Outputs* are collected from the program, and then they are compared with the *Expected Outputs*.



- The Ideal system is represented by the specification while the Real system is the actual code.
- For a test to be successful a pass result is not required. A failed test will also impart some new knowledge about the system

2. Black and White Box Testing

- Black Box testing is based entirely on the program specification and aims to verify that the program meets the specified requirements

- White box testing uses the implementation of the software to derive the tests. The tests are designed to exercise some aspect of the program code

Compare Black and White Box Testing

Black-Box Testing	White-Box Testing
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated to fix a fault, or additional functionality is added.	Tests are generally invalidated by any changes to the code.
Tests only require the specification to be developed.	Tests require that the code is written prior to developing the tests.
Tests do not ensure that all the code components have been exercised however, and thus are liable to missing "errors of commission" where extra functionality has been added to the code that is not in the specification.	Tests ensure that all the (selected) components have been exercised, however they do not ensure that the specification has been fully covered. They are thus liable to missing "errors of omission" where required functionality has not been implemented.
In general, it is difficult to automate measuring the effectiveness of Black-Box tests (i.e. how much of the specification has been covered).	In general, it is relatively easy to automate measurement of the effectiveness of White-Box tests (i.e. how much of the implementation has been covered).

Coverage Interpretation

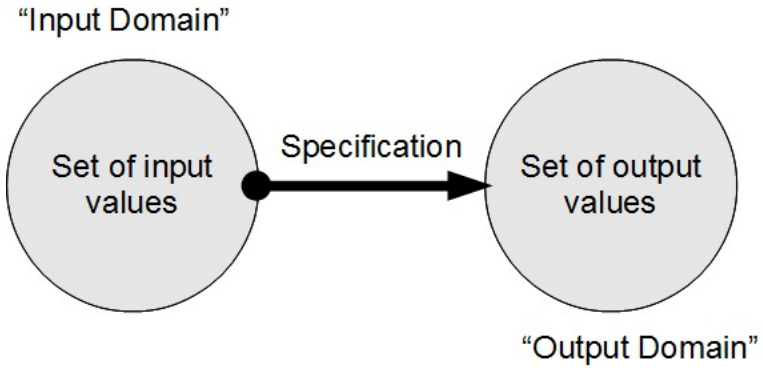
Black-Box testing provides for coverage of the specification, but not full coverage of the implementation. That is, there may be code in the implementation that produces results not stated in the specification.

White-Box testing provides for coverage of the implementation, but not of the specification. That is there may be behaviour stated in the specification for which there is no code in the implementation.

Black-Box Testing

The basic principle of Black-Box Testing can be expressed in a number of different ways:

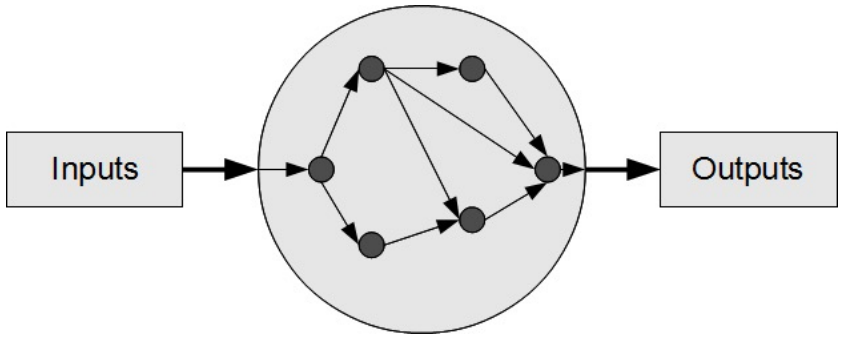
1. Test against the specification.
2. Use test coverage criteria based on the specification.
3. Develop test cases derived from the specification.
4. “Exercise” the specification.



White-Box Testing

The basic principle of White-Box Testing can be expressed:

1. Test against the implementation
2. Use test coverage criteria based on the implementation
3. Develop test cases derived from the implementation
4. “Exercise” implementation



Fault Insertion

- The most common technique is referred to as Mutation Testing where faults (or "mutations") are inserted into the source code, and the code checked to see if the mutant produces a different output.

- In Strong Mutation Testing this check is carried out by executing the code.

Offut's 5 sufficient mutations

- ABS: -ABS(), ABS(), 0
- AOR: + - * / %
- LCR: (a && b) or (a ll b) ->

  (a other-op b), (a), (b)

  (a op true), (a op false), (true op b), (false op b)
- ROR: (a cmp b) ->

  a [< <= > >= != ==] b, true, false
- UOI: insert "! - ++ --" before expression

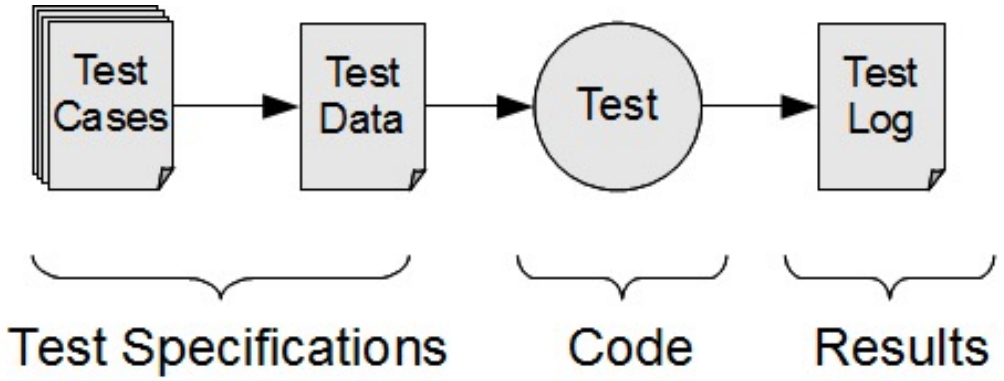
• AOR (arithmetic operator replacement) Replace one of the arithmetic operators by one of the others

• LCR (logical connector replacement) Replace one of the logical operators by one of the others

• ROR (Relational operator replacement) Replace one of the relational operators by one of the others

• UOI (unary operator insertion) Insert a unary operator before the expression

3. Test Artefacts



ID	Test Cases Covered	Inputs	Expected Output
		(parameter names here)	(parameter names here)
(ID) etc	(List here)	(Values here)	(Values here)