Software testing

Structural testing

Marco Aurélio Graciotto Silva[1], Ellen Francine Barbosa[2], Márcio Eduardo Delamaro[2], Auri Marcelo Rizzo Vincenzi[3], José Carlos Maldonado[2]

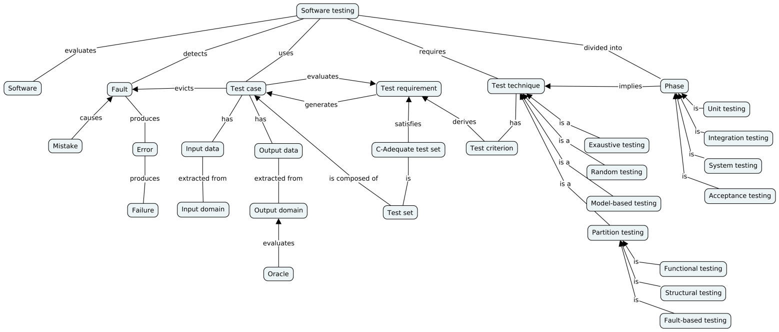
[1]Federal University of Technology – Paraná (UTFPR)
Campo Mourão, PR, Brazil

[2]University of São Paulo (USP)
São Carlos, SP, Brazil

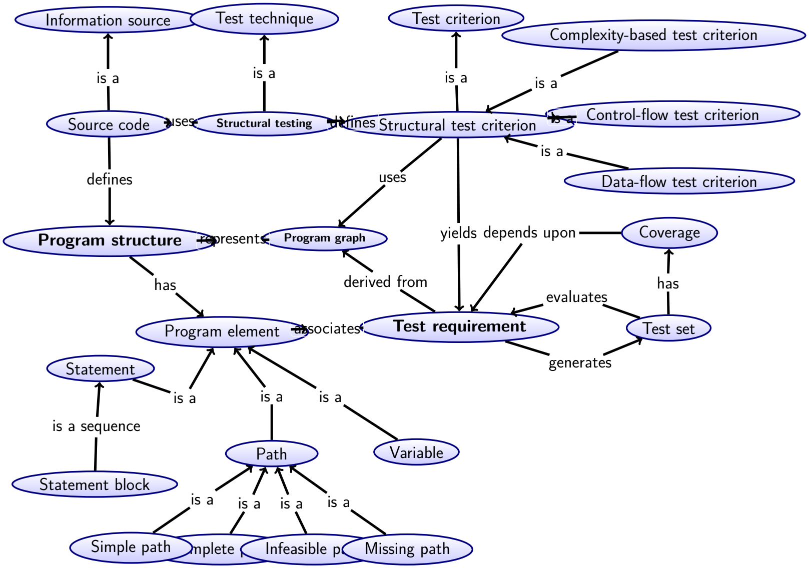
[3]Federal University of São Carlos (UFSCar)
São Carlos, SP, Brazil

2017

Software Testing



Structural testing



Structural testing

Definition

Structural testing is a technique in which testing is based on the internal paths, structure, and implementation of the software under test.

White-box testing

As structural testing must see the inner details of the software, it is also known as white box testing.

Why is structural testing important?

- Structural testing has efficacy on determine logical or programming faults in the program under testing, specially at the unit level.

Structural testing

Limitations

- Structural testing requires detailed programming skills.
  - Structural testing requires tester intervention in order to determine infeasible paths.
- The number of execution paths may be so large that they cannot all be tested.
- The test cases chosen may not detect data sensitivity errors.
- Structural testing assumes that control flow is correct (or very close to correct). Since the tests are based on the existing paths, nonexistent paths cannot be usually discovered through structural testing.

Structural testing

When can I use structural testing?

- Structural testing can be applied at unit, integration, and system testing phases.

Structural testing and test phases

- Structural testing, when applied at the unit testing phase, involves paths that are within a module.
- Structural testing, when applied at the integration testing phase, involves paths that are between modules within subsystems and paths between subsystems within systems.
- Structural testing, when applied at the system testing phase, involves paths that are between entire systems.

Structural testing

Test activities

1. The program under testing implementation is analyzed.
2. Paths through the program under testing are identified.
3. Inputs are chosen to cause the program under testing to execute selected paths. This is called path sensibilization.
4. Expected outputs for those inputs are determined.
5. The tests are run.
6. Actual outputs are compared with the expected outputs, verifying if the the actual output is correct.

Definition-use graph

Definition

The definition-use graph (DUG) is a program graph suitable to represent the different status of a variable.

Definition-use graph and control flow graph

- The definition-use graph is an extension of the control flow graph.
- The definition-use graph includes information about variable definitions, uses and destructions on each node.
    - It contains all definition-use pairs for a program.

Definition-use graph
Graph elements

Graph elements

- The nodes of a program graph are indivisible code blocks.
- The edges of a program graph represent the possible transference of execution between code blocks.
- There is just one input node, which corresponds to the code block with the first statement of the program unit.
- It is possible to have many output nodes, i.e., nodes without a succeeding node.

Example: Program graph   Example: Control-flow graph for Identifier

Definition-use graph
Code block

Definition

A code block of a program is a sequence of consecutive statements with a single entry point and a single exit point.

Control flow and code blocks

- Control always enter a code block at its entry point and exits from its exit point.
- There is no possibility of exit or a halt at any point inside a code block except at its exit point.

Code block

Control flow and code blocks

- The entry and exit points of a code block coincide when the code block contains only one statement.
- Function calls are often treated as code blocks of their own because they cause the control to be transfered away from the currently executing function and hence raise the possibility of abnormal termination of the program.

Definition-use graph
Decision

Definition

A decision is a statement that causes a deviation in the flow of a program.

Decision statements

- The transference of execution between code blocks is a consequence of decision statements.
- Most high-level languages provides statements, such as `if`, `while` and `switch`, to serve as context for decisions.

Definition-use graph
Variable definition

Definition

A variable definition is the assignment of a value before a variable be used.

- A variable definition occurs when a value is stored in a memory position.

Definition-use graph
Variable use

Definition

A variable use is when the reference to such a variable does not define it.

Types of variable use

- There are two kinds of variable use: computational use and predicative use.
    - A variable computational use (c-use) directly affects the computation being performed or allows the result from a previous variable definition to be observed.
    - A variable predicative use (p-use) directly affects the control flow of the product implementation.

DUG construction (1/2)

1. Consider a control-flow graph $G = (N, E)$ of a program P, where N is the set of nodes and E the set of edges. Each node in G corresponds to a code block in P: those blocks are denoted as $b_1, b_2, ..., b_k$, assuming that P contains $k > 0$ code blocks.
2. Let def_i denote the set of variables defined in block i.
3. Let $c - use_i$ denote the set of variables that have a computational use in block i.
4. Let $p - use_i$ denote the set of variables that have a predicative use in block i.

Example: Definition-use graph for Identifier

DUG construction (2/2)

5. Compute def_i, $c - use_i$, and $p - use_i$ for each code block i in P.
6. Associate each node i in N with def_i, $c - use_i$, and $p - use_i$.
7. For each node i that as a nonempty predicative use sets and ends in condition C, associate edges (i, j) and (i, k) with C and $!C$, respectively, given that edge (i, j) is taken when the condition is true and (i, k) is taken when the condition is false.

Example: Definition-use graph for Identifier

Path

Informal definition

A path is a sequence of statements.

Definition

A path is a finite sequence of nodes $(n_1, n_2, ..., nk)$, $k \geqslant 2$, so that there is an edge from n_i to $n_i + 1$ for $i = 1, 2, ..., k - 1$.

Path
Executable and infeasible path

Definition

An executable path is a path for which it exists an input data that can execute it.

Definition

An infeasible path is a path that, for any input value, cannot be executed.

Limitations and implications

- It is impossible to determine, automatically, infeasible paths.
- Any complete path that includes an infeasible path is an infeasible path.

Example: Infeasible path example for Identifier

Path
Definition-clear path

Informal definition

A definition-clear path is a path which no other variable definition is made.

Definition

A definition-clear path with respect to a variable x is a path between two nodes A and B, being x defined in A, with an use in B and with no other definition of x in the other nodes present in the path between A and B.

Example: Definition-clear path for Identifier

Structural test criterion

Definition

Structural test criterion identifies the execution paths inside a program unit and then creates and executes test cases to cover those paths, alongside to further test requirements established by specific structural testing criteria.

Types of structural test criteria

- There are three types of structural test criteria: complexity-based test criterion, control flow test criterion, data flow testing criterion.

Structural testing and JaBUTi

- JaBUTi supports structural software testing using control-flow and data-flow test criteria.

Control flow test criterion

Definition

- Control flow test criterion employs only characteristics related to the control structure of the program to determine the set of test requirements.
- Control flow test criterion identifies the execution paths inside a module and then creates and executes test cases to cover those paths.

Control flow test criteria

- All-Nodes.
- All-Edges.
- All-Paths.

Control flow test criteria and JaBUTi

- Regarding control-flow, JaBUTi supports the following test criteria: All-Nodes and All-Edges.

Control flow test criterion

Limitations

- In control flow test criterion, the number of test requirements can be huge and thus untestable within a reasonable amount of time:
  - Every decision doubles the number of paths; every loop multiplies the paths by the number of iterations through the loop.
- Paths called for in the specification may simply be missing in the module.
- Defects may exist in processing statements within the module even through the control flow itself is correct.
- The module may execute correctly for almost all data values but fail for a few.

Control flow test criterion
All-Nodes

Definition

All-Nodes requires a test set that exercises at least once each node of the control flow graph, which is equivalent to executing all code blocks of a program at least once.

Example: All-nodes example

Control flow test criterion
All-Nodes

Limitations

- Even though the All-Nodes criterion is the simplest structural test criterion, it may be difficult to satisfy in practice:
    - Often programs have code that is executed only in exceptional circumstances-low memory, full disk, unreadable files, lost connections, etc.
    - Testers may find it difficult or even impossible to simulate these circumstances and thus code that deals with these problems will remain untested.

Control flow test criterion
All-Edges

Definition

All-Edges requires a test set which traverses at least once each edge of the control flow graph, i.e., the test set must ensure that each conditional statement assumes true and false values at least once.

Control flow testing criterion
All-Paths

Definition

All-Paths requires a test set that executes all possible paths of the control flow graph.

Limitations

- For program units without loops, the test requirements for the All-Paths criterion is generally small enough so that a test case can actually be constructed for each path.
- For program units with loops, the test requirements for the All-Paths criterion can be enormous and thus pose an intractable test problem.

Complexity-based test criterion

Definition

Complexity-based test criterion uses information about program complexity in order to derive test requirements.

Complexity-based test criteria

- A well known complexity-based test criterion is the McCabe's criterion.

Complexity-based criterion
McCabe's criterion

Definition

The McCabe's criterion requires a set of linearly independent complete paths from the control flow graph to be traversed by the execution of the test set.

- The McCabe's criterion uses the cyclomatic complexity to derive the set of testing requirements.
- Satisfying McCabe's criterion automatically guarantees both decision coverage (All-edges) and statement coverage (All-nodes).

Complexity-based criterion
McCabe's criterion

1. Derive the CFG from the software module.
2. Compute the graph's Cyclomatic Complexity (C).
3. Select a set of C linearly independent paths.
   - 3.1 Pick a baseline path (it must be a complete path).
     - 3.1.1 This path should be a reasonably typical path of execution rather than an exception processing path.
     - 3.1.2 The best choice would be the most important path from the tester's viewpoint.
   - 3.2 To choose the next path, change the outcome of the first decision along the baseline path while keeping the maximum number of other decisions the same as the baseline path.
   - 3.3 Generate the remaining paths by varying the remaining decisions, one by one.
4. Create a test case for each basis path.

Example: McCabe's criterion example

Data flow test criterion

Definition

Data flow test criteria explore the interaction involving definitions of variables and further references (uses) to such definitions to establish the test requirements.

- Data flow test criteria aims to detect faults related with the definition and use of variables in a program, i.e., its target is the flow of data instead of the flow of control of a program.
  - Data flow testing is a powerful approach to detect improper use of data values due to coding mistake.
- Data flow test criteria complements the control flow test criteria.

Data flow testing criterion
All-Defs

Definition

The All-Defs requires that a data flow association for each variable definition to be exercised, at least once, by a definition-clear path with respect to a c-use or p-use.

Data flow testing criterion
All-Uses

Definition

The All-Uses requires that all data flow associations between a variable definition and all its subsequence uses (c-uses and p-uses) to be exercised by at least one definition-clear path.

Example: All-Uses test requirements for Identifier

Data flow testing criterion
All-P-Uses

Definition

The All-P-Uses requires that a data flow association for each predicative use of a variable to be exercised at least once.

Data flow testing criterion
All-C-Uses

Definition

The All-C-Uses requires that a data flow association for each computational use of a variable to be exercised at least once.

Data flow testing criterion
All-Pot-Uses

Definition

The All-Pot-Uses requires for each node i containing a definition of a variable x that to all node and edge that can be reached from i by a definition-clear path with respect to x to be exercised.

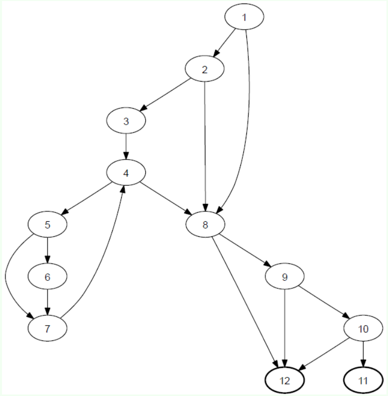
Example: All-Potential-Uses test requirements for Identifier

References

Credits

- Reviewers:
  - Fabiano Cutigi Ferrari
  - Márcio Eduardo Delamaro
  - Otávio Augusto Lazzarini Lemos

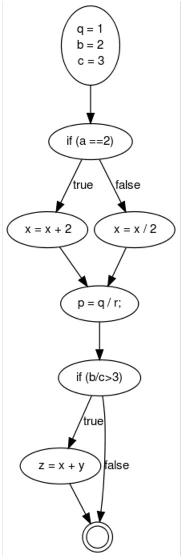
Identifier control-flow graph



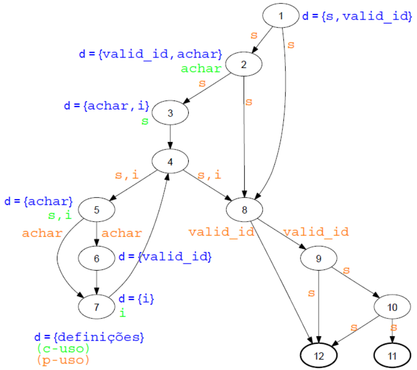
Program graph

```
q = 1;
b = 2;
c = 3;
if (a ==2) {
    x = x + 2;
} else {
    x = x / 2;
}
p = q / r;
if (b/c>3) {
    z = x + y;
}
```

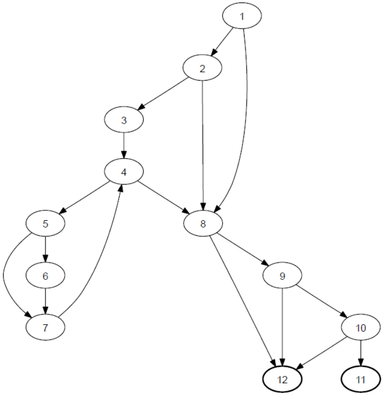
Program graph



Identifier definition-use graph

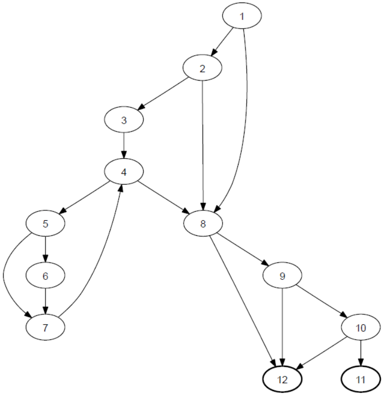


The following path is infeasible: (5, 6, 7, 4, 8, 9, 10, 11)



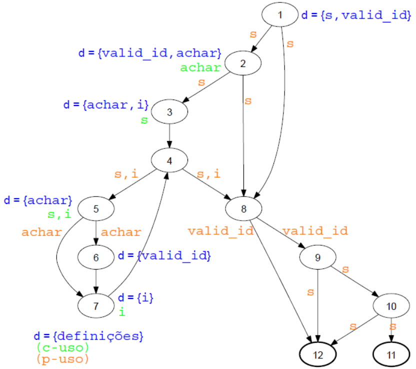
Identifier
Complete path

The following path is complete: (1, 2, 3, 4, 5, 6, 4, 8, 12)



Definition-clear paths example

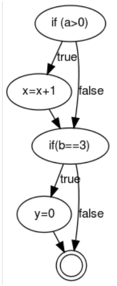
Consider the following graph:



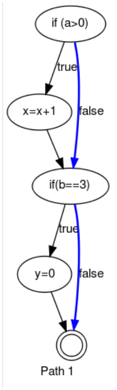
Definition-clear paths example

- Path (1,8,12) is a definition-clear path with respect to `valid_id` defined at node 1.
- Path (1,2,8,12) is not a definition-clear path with respect to `valid_id` defined at node 1, because `valid_id` is redefined at node 2.

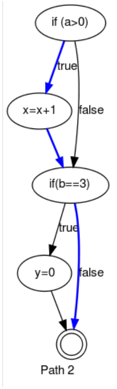
All-nodes



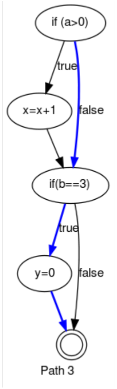
All-nodes
Path 1



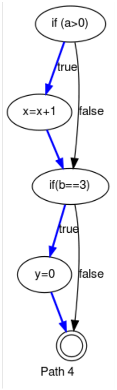
All-nodes
Path 2



All-nodes
Path 3



All-nodes
Path 4



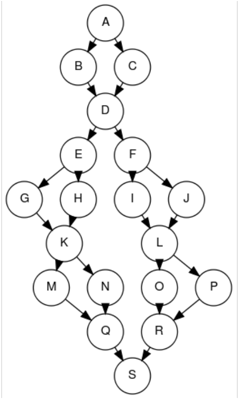
All-paths infeasibility example

The block below executes `doSomethingWith()` one billion times (1000 x 1000 x 1000).

```
for (i =1; i <=1000; i ++)
    for (j =1; j <=1000; j ++)
        for ( k =1; k <=1000; k++)
            doSomethingWith(i, j, k);
```

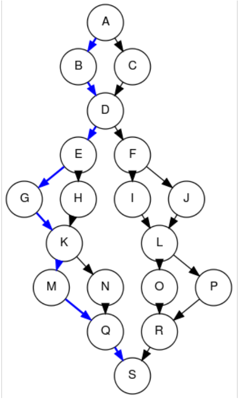
McCabe example

Consider the graph below:

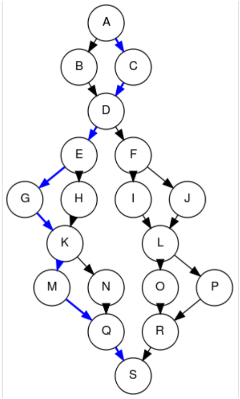


The cyclomatic complexity of the graph is 7. So, seven test requirements, thus seven complete paths, must be devised for the graph.

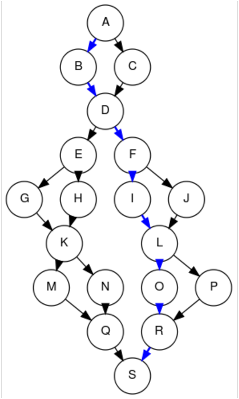
McCabe example
Path 1



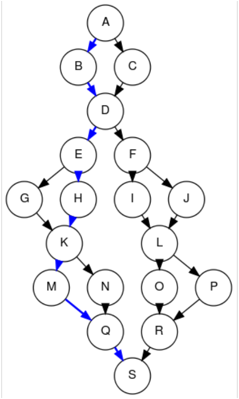
McCabe example
Path 2



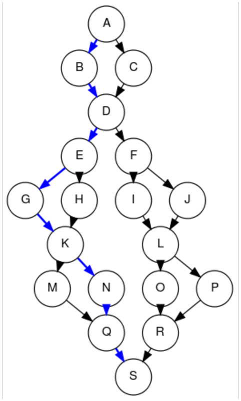
McCabe example
Path 3



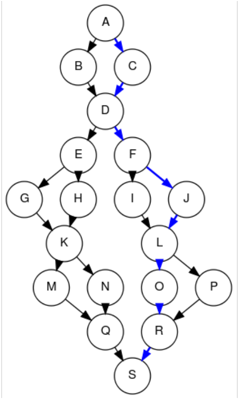
McCabe example
Path 4



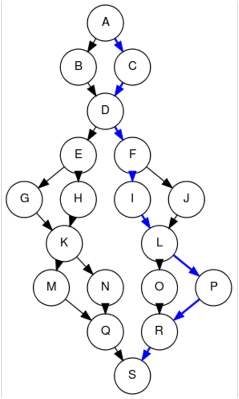
McCabe example
Path 5



McCabe example
Path 6



McCabe example
Path 7

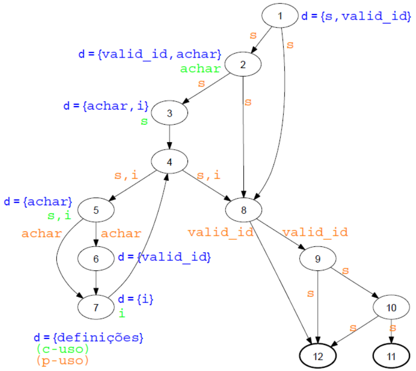


All-Uses for Identifier

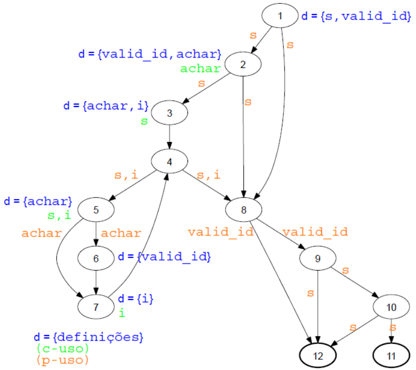
Some test requirements for Identifier considering the All-uses criterion:

- (length, 1, 2)
- (achar , 1, 3)
- (valid id, 1, (1, 3))

All-Uses for Identifier



All-Potential-Uses for Identifier



All-Potential-Uses for Identifier

Some test requirements for Identifier considering the All-Pot-uses criterion:

- (length, 1, 2)
- (achar , 1, 3)
- (valid id, 1, (1, 3))
- (length, 2, (8, 10))
- (achar , 3, (8, 10))