**Selecting Software Test Data Using Data Flow Information**

**By Sandra Rapps and Elaine J. Weyuker (1985)**

**Introduction**

Testing is one of the most important ways through we check if a program works correctly. The general idea is to take inputs from the program’s domain, run the program, and then compare the actual outputs with the expected outputs.

But it is not possible to test all inputs because the input space is usually infinite or extremely large. So, testers usually choose a subset of inputs that represent the whole domain. Here, the problem is, how do we select this subset so that it is able to detect the maximum possible errors.

Traditionally, test data selection has been done using control flow-based coverage criteria, like

* Statement coverage: every statement must be executed at least once.
* Branch coverage: every decision branch (true/false) must execute at least once.
* Other path coverage measures.

However, these criteria are not always effective. For instance, even if all paths are executed, some errors may remain hidden because certain definitions of variables are never actually used in meaningful computations. Also, loops can create infinitely many paths, leading to impractical coverage.

To solve this, the authors propose using data flow information for test data selection. Instead of just looking at control structures, they focus on how variables are defined, assigned values, and used in computations or conditions. By ensuring that every variable definition is properly exercised and its effects are observed, testing becomes more reliable.

**Programming Language Model**

The authors define a simple, theoretical programming language for their analysis. The language contains:

* Start, stop statements.
* Input (read a), output (print b) statements
* Assignments (y=f(x1,…,xn))
* Conditional and unconditional jumps.

Every program is a finite sequence of such statements, starting with start and ending with either stop or a jump to stop. Programs can be divided into blocks, each of which executes sequentially without interruption, except at its end.

**Flow Graph Concepts**

Programs are represented using flow graphs:

* Each block is a node.
* Edges connect nodes based on control flow (successors, predecessors).
* Start node has no predecessors, exit node has no successors.

A path is a sequence of nodes connected by edges. A complete path goes from the start to the exit. Loops in the program mean infinitely many possible paths, but certain loop-free subpaths are more useful for testing.

**Definition-Use (DU) Graph**

This paper introduces the definition/use graph, which extend the flow graph with data flow information.

* Definition: a variable is assigned a value, DEF(u, v)
* C-use (computation use): variable used in calculation or output.
* P-use (predicate use): variable used in a conditional expression.

Each occurrence of a variable is marked as either DEF, c-use, or p-use.

basically, they help distinguish between global uses, i.e, values coming from outside the current block, and local uses, values defined and used within the same block.

From this, they define def-clear paths, which are paths where no redefinition of a variable occurs between its definition and its use.

Therefore, a du-path connects a variable’s definition to its uses (either c-use or p-use), without any redefinition in between.

**Path Selection Criteria (Proposed Family)**

The paper has defined a family of test adequacy criteria based on these def/use associations:

1. All-nodes: every node in the program graph must be executed (same as statement coverage).
2. All-edges: every edge must be executed (branch coverage).
3. All-defs: for every variable definition, there must be at least one path to some use of that definition.
4. All-p-uses: every definition must be connected to all predicate-uses.
5. All-c-uses/some-p-uses: for every definition, all c-uses must be exercised; if no c-use exists, then at least one p-use must be tested.
6. All-p-uses/some-c-uses: dual of the above, all p-uses must be exercised, but if none exist, at least one c-use must be tested.
7. All-uses: every definition must reach all its c-uses and p-uses.
8. All-du-paths: every definition must reach every possible use along every def-clear path.
9. All-paths: all complete paths in the graph must be tested (impractical if loops exist).

**Relationships Between Criteria**

The criteria form a hierarchy of strength:

* All-paths is the strongest but infeasible due to loops.
* All-du-paths is slightly weaker but still very strong.
* All-uses is strong and practical in many cases.
* All-p-uses/some-c-uses and All-c-uses/some-p-uses are intermediate.
* All-defs is weaker.
* All-edges and All-nodes are the weakest but most widely used.

The authors here also show that some criteria are incomparable, i.e. neither stronger nor weaker than the other. For example, *all-defs* and *all-edges* may cover different cases.

**Example**

The paper presents a small program with an intentional error: a misordered statement.

* If we apply all-edges coverage, the error goes undetected because the definition of a variable will never actually be used.
* If we apply all-defs or all-uses, the error is detected because these criteria force paths where the faulty definition is used.

This demonstrates the practical advantage of data flow–based criteria.

**Conclusions and Future Work**

This paper concludes that:

* Data flow criteria provide a balance between the very weak node/edge coverage and the infeasible all-paths coverage.
* They ensure that definitions of variables are meaningfully tested by connecting them to their uses.
* These criteria can be systematically applied and even checked mechanically.

They have suggested building a software tool that takes a program, a test set, and a chosen criterion, and reports which def-use pairs or paths were not exercised by the tests. While some paths may be unexecutable (which is generally undecidable), the tool would still help highlight neglected areas of the code.