Introduction to Software Testing

Elements and Concepts

Daniel “bodom_lx” Graziotin, http://bd-things.net

Summary

Introduction

This document contains some basic concepts and definitions about software testing. It has been written for studying a part of the Software Engineering Project course at my University. It is composed by a summary of the intersection of more than 10 different sources, all of which are cited. If you feel that some contents of this publication belong to your intellectual property and it is not cited, please contact the author who is willing to correct any mistake.

The first part of the paper focuses on the definition of the most important key aspects of software testing. Then some information about input partitioning are given. What follows is a research about code coverage and two useful and famous tools, Control-flow coverage and Data-flow analysis. A complete example on using those tools is then given. The second half of the document also contains the definition of the most important software testing practices.

The goal of this tiny document is to clarify key terms and therefore become a base start for the reader to go in deep with the interested topics. Another goal is to give a simple but clear example about data flow analysis, as I realized that not all the people understand the examples around the Net.

Software Testing

Software Testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate. Software Testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs. It can also be stated as the process of validating and verifying that a software program/application/product meets the business and technical requirements that guided its design and development, so that it works as expected and can be implemented with the same characteristics.[1]

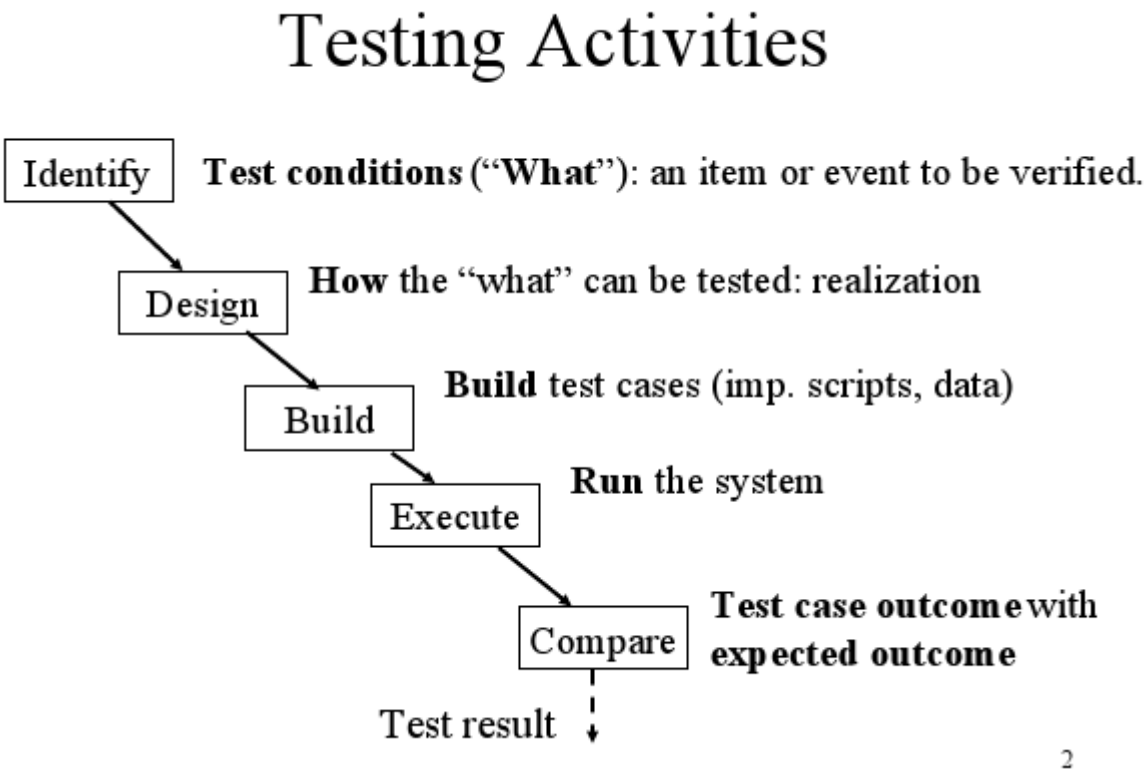
Why Software Testing is Important

- To Improve quality
- For Verification and Validation
- For Reliability estimation

Functionality (exterior quality)	Engineering (interior quality)	Adaptability (future quality)
Correctness	Efficiency	Flexibility
Reliability	Testability	Reusability
Usability	Documentation	Maintainability
Integrity	Structure	

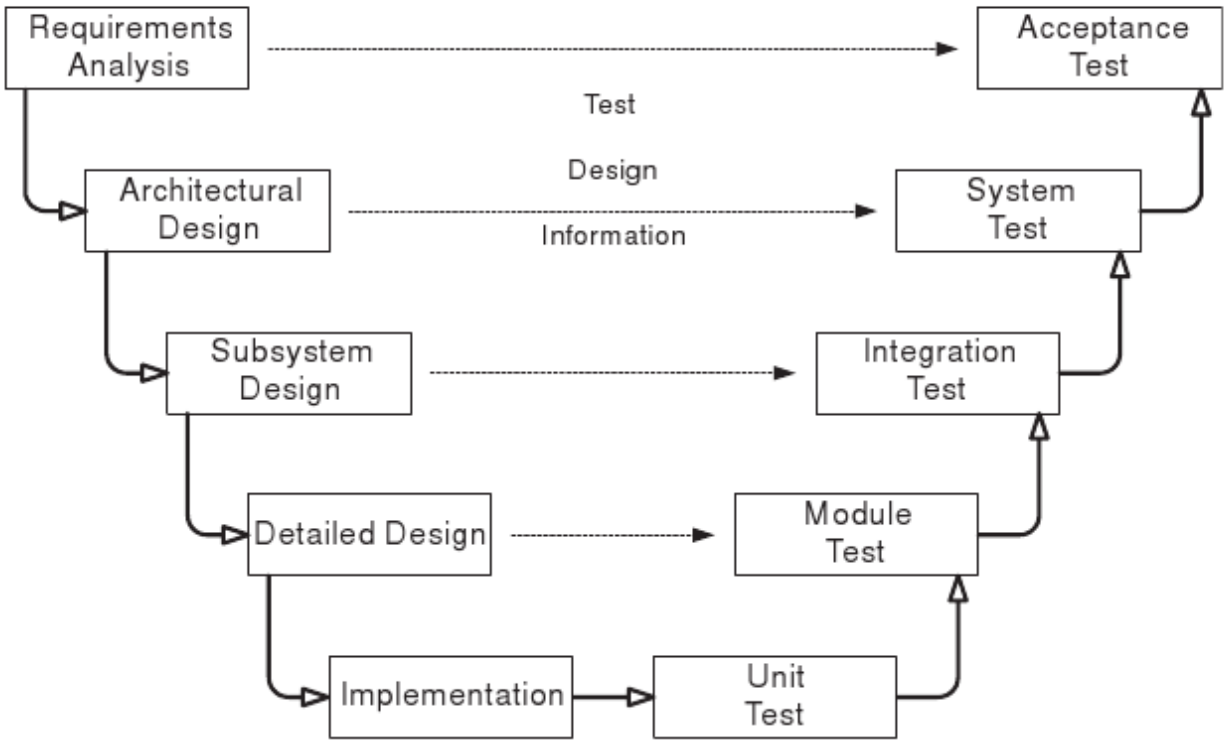
taken from http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/#introduction

Testing Activities



1 http://en.wikipedia.org/wiki/Software_testing

Testing Processes with respect to Software Activities



Black-Box Testing

Black box testing takes an external perspective of the test object to derive test cases. These tests can be functional or non-functional, though usually functional. The test designer selects valid and invalid inputs and determines the correct output. There is no knowledge of the test object's internal structure.[2]

- You don't know anything about how the code is written
- Therefore, the possible flows of computation of the program is unknown
- You play with compiled (pieces of) software, modifying things like function parameters

White-Box Testing

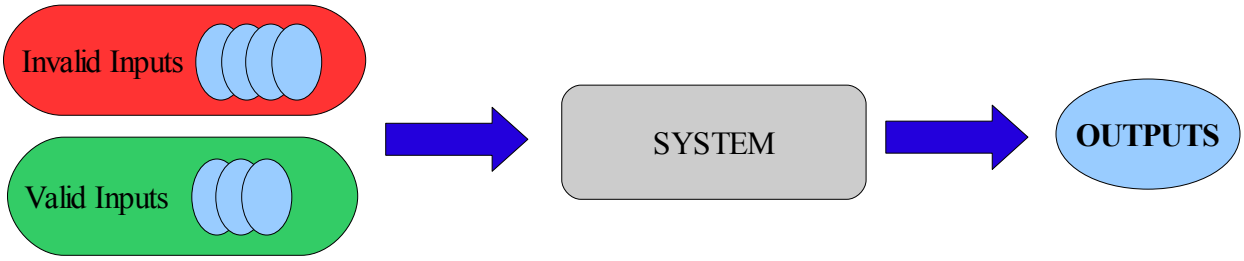
White box testing (a.k.a. clear box testing, glass box testing, transparent box testing, translucent box testing or structural testing) uses an internal perspective of the system to design test cases based on internal structure. It requires programming skills to identify all paths through the software. The tester chooses test case inputs to exercise paths through the code and determines the appropriate outputs.[3]

- You have complete access on the source code of the program
- The flow of computation can be observed
- Problem of code coverage of the tests

2 http://en.wikipedia.org/wiki/Black-box_testing
3 http://en.wikipedia.org/wiki/White_box_testing

Equivalence Partitioning

Equivalence partitioning is a software testing technique that divides the input data of a software unit into partition of data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once. This technique tries to define test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.[4]



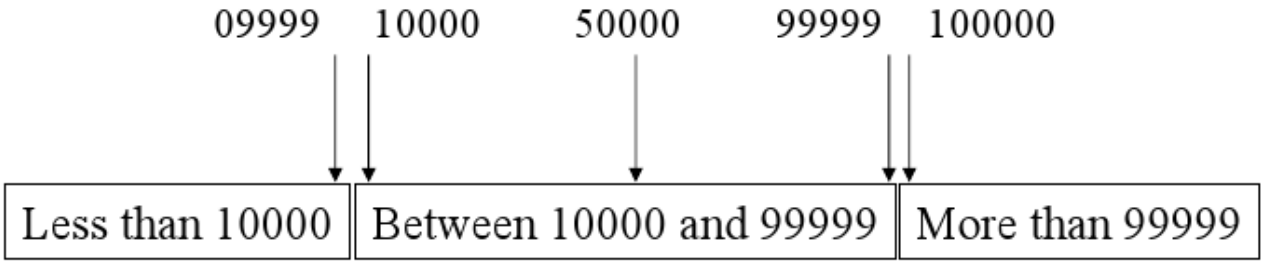
Strategy for Input Partition in General

Look at the boundaries of the input.

If input is a 5-digit between 10000 and 99999, equivalence partitions are:

1. < 10000
2. 10000 – 99999
3. > 99999

Choose test cases at the boundaries of the partitions:



Every procedure has pre-conditions and post-conditions, example:

Binary Search (element)

pre: array with at least one element → A[first] <= A[last]

post: -element is found and referenced by l → l = A[element] OR
-element is not found → l = false;

Use

1. inputs which conform the pre-conditions
2. inputs where a pre-condition does not hold
3. inputs derived from the post-conditions.

4 http://en.wikipedia.org/wiki/Equivalence_partitioning

Strategy for Input Partition when dealing with arrays

Test using:

1. Arrays with a single value
2. Arrays of zero length
3. Arrays with different sizes in different tests
4. Implement tests such that first, last and a middle element of the array are taken in consideration

Array	Element
Single value	In array
Single value	Not in array
More than 1 value	First element in array
More than 1 value	Last element in array
More than 1 value	Middle element in array
More than 1 value	Not in array
No value	Not in array

Continuing with the example, a good test for binary search will include:

- Pre-conditions satisfied, key element in array
- Pre-conditions satisfied, key element not in array
- Pre-conditions unsatisfied, key element in array
- Pre-conditions unsatisfied, key element not in array
- Input array has a single value
- Input array has an even number of values
- Input array has an odd number of values

Code Coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing[5]

- **Function coverage** - Has each function in the program been called?
- **Statement coverage** - Has each line of the source code been executed?
- **Decision coverage** (also known as Branch coverage) - Has each control structure (such as an if statement) evaluated both to true and false?
- **Condition coverage (or Predicate coverage)** - Has each boolean sub-expression evaluated both to true and false (this does not necessarily imply decision coverage)?
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?

We will just focus on path coverage.

Path coverage testing based on Cyclomatic Complexity

Path coverage focuses on executing distinct parts rather than just edges or or statements. It can be feasible or infeasible depending on the conditions of the program. It explodes very quickly depending on the size of the program. There are several path coverage strategies but we are interested on path testing based on Cyclomatic Complexity.

Cyclomatic complexity (or conditional complexity) is a software metric (measurement). It was developed by Thomas J. McCabe Sr. in 1976 and is used to measure the complexity of a program. It directly measures the number of linearly independent paths through a program's source code[6]

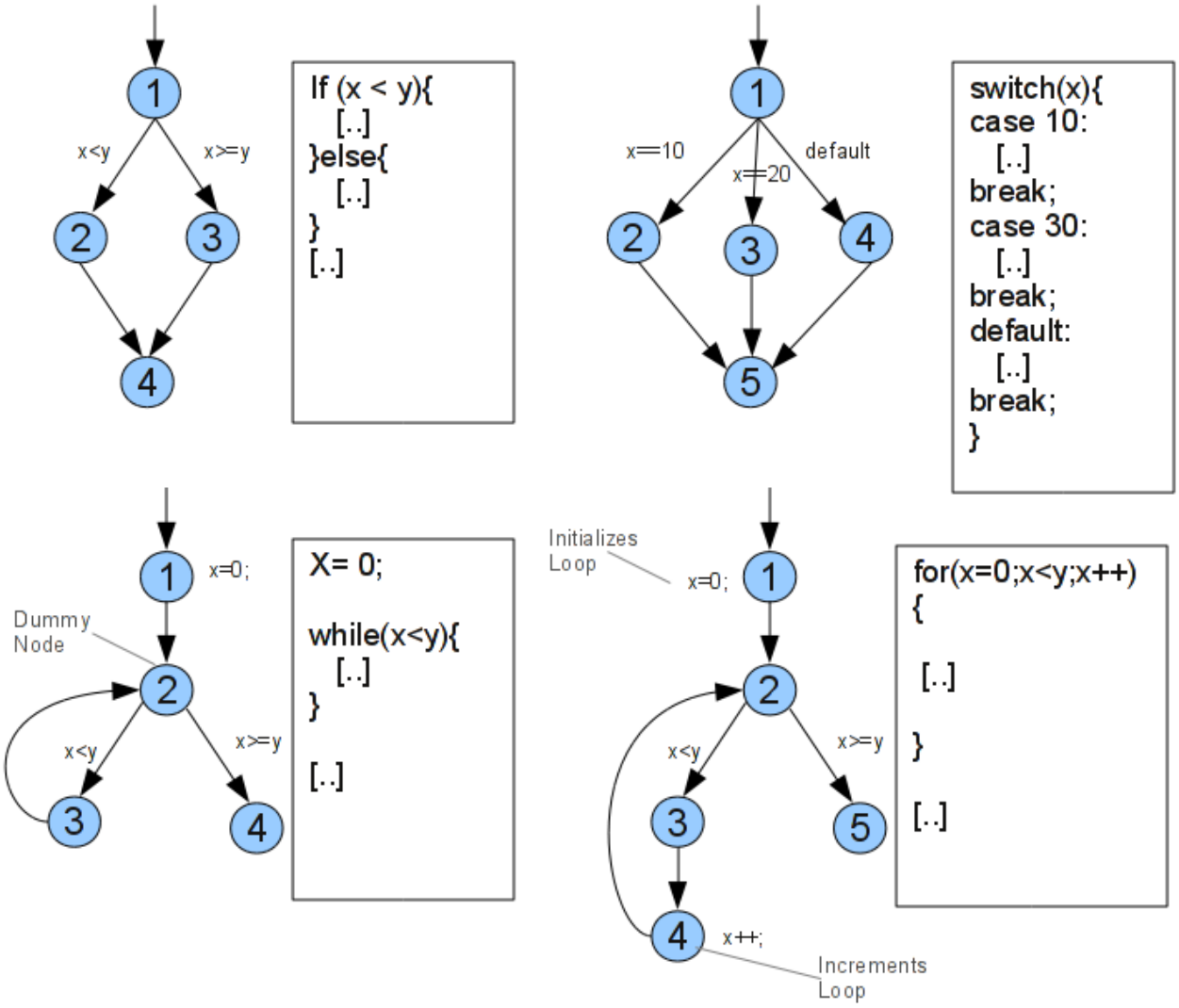
Its Aim is to derive a logical complexity measure of a procedural design and use this as a guide for defining a basic set of execution paths. The number of test cases is proportional to the size of the program. More precisely, the value of Cyclomatic Complexity can be seen as an upper bound for the number of test cases needed for path coverage, as it is a measure of the number of the independent paths.[7]

5 http://en.wikipedia.org/wiki/Code_coverage
6 http://en.wikipedia.org/wiki/Cyclomatic_complexity
7 Practical software testing, p. 110

Cyclomatic complexity is computed using control flow graphs, so we will introduce them first.

A control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution.[8]

Basic structure:



On a flow graph:

- **Arrows** called ***edges*** represent flow of control
- **Circles** called ***nodes*** represent one or more actions.
- **Areas** bounded by edges and nodes called ***regions***.
- A ***predicate*** *node* is a node containing a condition[9]
- succ[n] is the set containing the successors of a node n
- pred[n] is the set containing the predecessors of a node n

In the first CFG, succ[1] = {2,3}, succ[2] = succ[3] = {4}, pred[4] = {2,3}, pred[2] = pred[3] = {1}

Computing **Cyclomatic Complexity V(G)**. Choose one between:

1. number of Edges - number of Nodes + 2 * (number of connected components [usually 1])
2. number Predicate Nodes + 1
3. Number of regions of flow graph. (area surrounded by nodes/edges)

8 http://en.wikipedia.org/wiki/Control_flow_graph
9 http://users.csc.calpoly.edu/~jdalbey/206/Lectures/BasisPathTutorial/index.html

Data Flow Testing

The next few testing criteria are based on the assumption that to test a program adequately, we should focus on the flows of data values. Specifically, we should try to ensure that the values created at one point in the program are created and used correctly. [10] Data Flow Testing (or Data Flow Analysis) is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.[11]

There are several versions of data flow testing, we will focus on the easiest one.

Elements:

- Definition (**def**) : A location where a value for a variable is stored into memory
- **Use** : A location where a variable's value is accessed
- **def (n)** or **def (e)** : The set of variables that are defined by node n or edge e
- **use (n)** or **use (e)** : The set of variables that are used by node n or edge e
- **DU** pair : A pair of locations (l_i, l_k) such that a variable v is defined at l_i and used at l_k.
- **Reach** : If there is a def-clear path from l_i to l_k with respect to v, the def of v at l_i reaches the use at l_k.
- **du-path** : A simple subpath that is def-clear with respect to v from a def of v to a use of v
- **du (n_i, n_k, v)** – the set of du-paths from n_i to n_k
- **du-chain(X,S,S1)** – X is in def(S) and in use(S1), and def(X) at S is live at S1
- **du (n_i, v)** – the set of du-paths that start at n_i
- **live** – a variable is live at a particular point if its value at that point will be used in the future
- **live-out** – a variable is live out at a particular point if it is live on any out-edges
- **live-in** – a variable is live in at a particular point if it is live on any in-edges
- **dead** – a variable which is not live.
- **mindef(i)** – set of variable that have a definition at node I
- **defclear(i)** – all variables that have def-clear paths (never used)
- **Visit** : A test path tp visits node n if n is in tp

  A test path tp visits edge e if e is in tp
- **Tour** : A test path tp tours subpath sb if sb is a subpath of tp
- **Test Path** : A path that starts at an initial node and ends at a final node, represent execution of test cases
- A test path tp **du-tours** subpath sb with respect to v if tp tours db and the subpath taken is def-clear with respect to v.

10 Introduction to Software Testing, p. 44 and http://cs.gmu.edu/~offutt/softwaretest/powerpoint/Ch2-1-2-overviewGraphCoverage.ppt

11 http://en.wikipedia.org/wiki/Data-flow_analysis

There are also several data-flow based tests criteria. The most simple are:

1. Make sure that every def reaches a use
   All-defs coverage (ADC) : For each set of du-paths S = *du (n, v)*, test requirement TR contains at least one path d in S.
2. Make sure that every def reaches all possible use
   All-uses coverage (AUC) : For each set of du-paths to uses S = du (ni, nj, v), test requirement TR contains at least one path d in S.
3. Cover all paths between defs and uses
   All-du-paths coverage (ADUPC) : For each set S = du (ni, nj, v), test requirement TR contains every path d in S.

Data-flow information can be collected by setting up and solving systems of equations that relate information at various points in the program. General equations are:

$$\text{out}[n] = gen[n] U (\text{in}[n] - kill[n])$$

$$\text{in}[n] = gen[n] U (\text{out}[n] - kill[n])$$

Definitions of in, out, gen, kill depend on the type of the analysis we want to perform.

The data-flow equations for **live-variable analysis** are:

$$\text{in}[n] = use[n] U (out[n] - \text{def}[n])$$

$$out[n] = \underset{s \in succ[n]}{U} \text{in}[s]$$

Application of live variable analysis: the value of dead variables need not to be saved at the end of a basic block

General rules about **liveness** of a variable:

1. Generate liveness:
   If a variable is in use[n], it is live-in at node n
2. Push liveness across edges:
   If a variable is live-in at a node n then it is live-out at all nodes in pred[n]
3. Push liveness across nodes:
   If a variable is live-out at node n and not in def[n] then the variable is also live-in at n

The data-flow equations for **uninitialized-variable analysis** are:

$$mindef(n) = \cap(mindef[pred(n)] \cup \text{def}[n])$$ mindef(first node) = {all variables}

$$defclear(n) = \cap(defclear[pred(n)] \cup \text{def}[n] - use[n])$$ defclear(first node) = {}

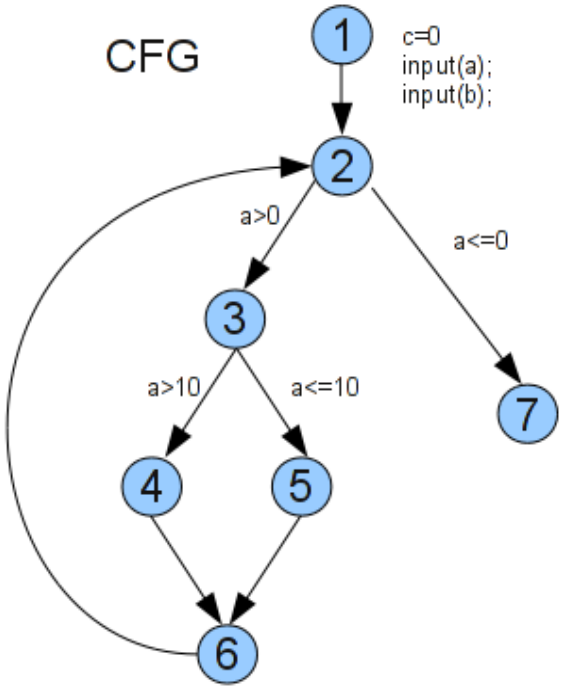
General rules about uninitialized-variable analysis:

1. d- : defclear(last node) != {}
2. dd: defclear(n) ∩ def(n) != {}
3. -u: use(n) - mindef(n) != {}

Example

Control-flow Graph

```
c = 0;
input(a);
input(b);
while (a>0) {
   if (a>10)
      a = a – b;
   else
      b = b + 1;
   }
print a,b
```

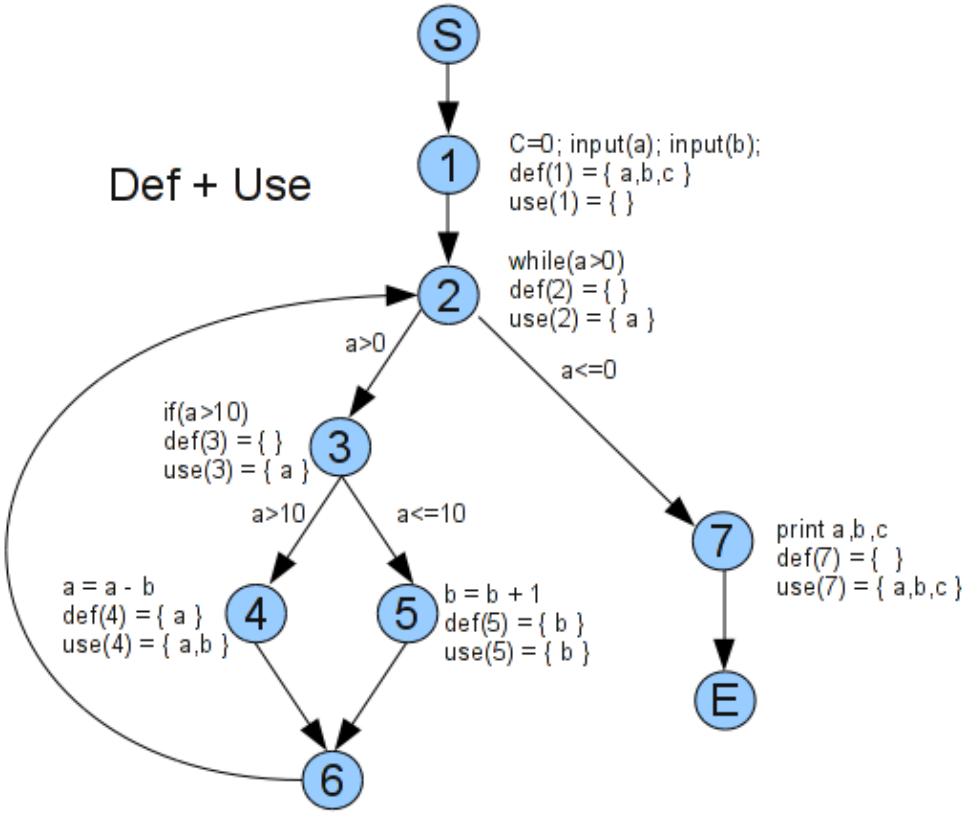


number of Edges e: 8
number of Nodes n: 7

=> Cyclomatic Complexity V(G): 8 – 7 + 2(1) = 3
The independent paths are 1-2-7, 1-2-3-4-6-2-7 and 1-2-3-5-6-2-7.

Data-flow Analysis: def & use

ADUPC: to satisfy the all def-use criterion, the tester must identify and classify occurrences of all the variables in the software under test. A tabular summary could also be useful. The following is the graph with all def and use.



The tester then generates test data to exercise all of these def-use pairs.

Test data set 1: a = -1, b = -1 covers pairs 1,2 for a; 1 for b and 1 for c
=> the flow is S-1-2-7-E
Test data set 2: a = 11, b = 12 covers pairs 1,3,4,5,6 for a; 1,2 for b and 1 for c
=> the flow is S-1-2-3-4-6-2-7-E
Test data set 3: a = 11, b = 9 covers pairs 1,3,4,5 for a; 2,5 for b
=> the flow is S-1-2-3-4-6-2-3-5-6-2-3-5-6-{2-3-5-6}+
Test data set 4: a = 11, b = 0 covers pairs 1,3,4,5,7 for a; 2 for b
=> the flow is S-1-2-3-4-6-{2-3-4-6}+
Test data set 5: a = 11, b = -1 covers pairs 1,3,4,5,7 for a; 2 for b
=> the flow is S-1-2-3-4-6-{2-3-4-6}+
Test data set 6: a = 9, b = 1 covers pairs 1,3,4,5,7, 5 for b
=> the flow is S-1-2-3-5-6-{2-3-5-6}+

The first information that we derive from the tests is that there are many values for a and b that cause the software to fail (infinite loop). Another information is about path coverage: it seems that there are no possible values to cover pairs 3 and 4 for b. Therefore, there are also paths not accessible during the execution of the program.
The last useful and important information is that the independent path 1-2-3-5-6-2-7 is never used.

Data-flow Analysis: live variable analysis

Liveness analysis for a and b (not considering c)

Variable a:
is dead in every path containing 2-4-6, because we assign a new value for a in 4
=> is live in 1-2-7 and in 1-2-3-5-6-2-7 paths
=> is dead in 1-2-3-4-6-2-7
Variable b:
is dead in every path containing 2-4-5, because we assign a new value for b in 5
=> is live in 1-2-7 and in 1-2-3-4-6-2-7 paths
=> is dead in 1-2-3-5-6-2-7 (contains 2-4-6)

More precisely, and considering all variables:

$$\text{in}[n] = use[n] \, U \, (out[n] - \text{def}[n])$$

$$out[n] = \bigcup_{s \in succ[n]} \text{in}[s]$$

live-in [1] = {} U (live-out[1] – {a,b,c})
live-out[1] = live-in[2]

live-in [2] = {a} U (live-out[2] – {})
live-out[2] = live-in[3] U live-in[7]

live-in[7] = {a,b,c} U ({} - {}) = {a,b,c}
live-out[7] = {} // end of program.

live-in [3] = {a} U (live-out[3] – {})
live-out[3] = live-in[4] U live-in[5]

live-in [4] = {a,b} U (live-out[4] – {a})
live-out[4] = live-in[6]

live-in [5] = {b} U (live-out[5] – {b})
live-out[5] = live-in[6]

=> **live-in[6]** = {} U (live-out[6] – {}) = live-out[6], anyway
live-out[6] = live-in[2] = {a} U (live-in[3] U live-in[7] – {})
= {a} U (live-in[3] U {a,b,c} - {}) = {a,b,c}, anyway

=> **live-out[5]** = live-in[6] = live-out[6] = {a,b,c};
live-in[5] = {b} U (live-out[6] – {b}) = {b}U {a,b,c} – {b} = {a,b,c}
=> **live-out[4]** = live-in[6] = live-out[6] = {a,b,c};
live-in[4] = {a,b} U (live-out[6] – {a}) = {a,b} U {a,b,c} – {a} = {a,b,c}

=> **live-out[3]** = {a,b,c} U {a,b,c} = {a,b,c}
=> **live-in[3]** = {a} U ({a,b,c} - {}) = {a,b,c}

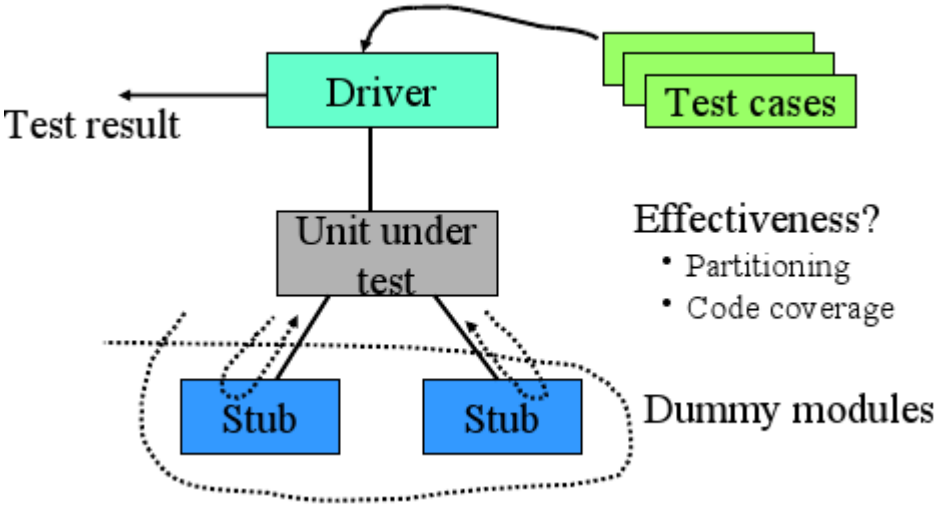
=> **live-out[2]** = live-in[3] U live-in[7] = {a,b,c} U {a,b,c} = {a,b,c};
live-in[2] = {a} U ({a,b,c} - {}) = {a,b,c}

=>**live-out[1]** = live-in[2] = {a,b,c}
live-in [1] = {} U (live-out[1] – {a,b,c}) = {a,b,c} – {a,b,c} = {} // beginning of the program.

Unit Testing

A software verification and validation method where the programmer gains confidence that individual units of source code are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual program, function, procedure, etc., while in object-oriented programming, the smallest unit is a class, which may belong to a base/super class, abstract class or derived/child class.

Ideally, each test case is independent from the others: substitutes like method stubs, mock objects, fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.[12]



Integration Testing

Integration testing is a logical extension of unit testing. In its simplest form, two units that have already been tested are combined into a component and the interface between them is tested. A component, in this sense, refers to an integrated aggregate of more than one unit. In a realistic scenario, many units are combined into components, which are in turn aggregated into even larger parts of the program. The idea is to test combinations of pieces and eventually expand the process to test your modules with those of other groups. Eventually all the modules making up a process are tested together. Beyond that, if the program is composed of more than one process, they should be tested in pairs rather than all at once.[13]

System Testing

System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic.

As a rule, system testing takes, as its input, all of the "integrated" software components that have successfully passed integration testing and also the software system itself integrated with any applicable hardware system(s). The purpose of integration testing is to detect any inconsistencies between the software units that are integrated together (called assemblages) or between any of the assemblages and the hardware. System testing is a more limiting type of testing; it seeks to detect defects both within the "inter-assemblages" and also within the system as a whole. [14]

12 http://en.wikipedia.org/wiki/Unit_testing
13 http://msdn.microsoft.com/en-us/library/aa292128%28VS.71%29.aspx

14 http://en.wikipedia.org/wiki/System_testing

The purpose of system Testing methodology is

• To define how system testing projects are executed and

• To serve as a basis for different phases in a system testing project[15]

Keyword	Definition
Bug Tracking	Bug Tracking process covers the defect lifecycle from identification of a defect to its resolution / closure.
Functional Testing	Process to determine that the features / functionality of the application / product is as per the requirements
Load testing	Testing the application behavior under varying acceptable loads
Performance Testing	Testing conducted to evaluate the compliance of a system or component with specified performance requirements
Regression Testing	• Process to ensure that the earlier applications/ products still work with the new changes. • Tests ensure that changes do not introduce unintended behavior or additional errors.
Smoke testing	Smoke testing to ensure the build version is ready for undertaking testing
System Testing	Process to determine that the system functions as intended or documented.

Test Stopping Criteria

• Meet deadline, exhaust budget, … management

• Achieved desired coverage

• Achieved desired level failure intensity

Acceptance Testing

An acceptance test is a test that the user defines, to tell whether the system as a whole works the way the user expects. Ideally, the acceptance tests are defined before the code that implements the feature. [16]

Regression Testing

- Whenever a system is modified (fixing a bug, adding functionality, etc.), the entire test suite needs to be rerun – Make sure that features that already worked are not affected by the change
- Automatic re-testing before checking in changes into a code repository
- Incremental testing strategies for big systems

15 http://www.scribd.com/doc/7703029/system-testing-methodology
16 http://xp123.com/xplor/xp0105/index.shtml

Further Reading and Sources

Introduction to Software Testing, Paul Ammann and Jeff Offutt, Cambridge Press
Practical software testing, Ilene Burnstein, Springer
Testing Object-oriented Systems Models, Patterns, and Tools, Robert V. Binder, Addison-Wesley.
Ian Sommerville slides (http://www.cs.st-andrews.ac.uk/~ifs/Books/SE8/Syllabuses/INTRO-SLIDES/)
All Wikipedia Pages cited in the publication
All other sources cited in the publication
This very useful article: http://www.softwaretestinghelp.com/practical-software-testing-tips-to-test-any-application/