Introduction to Software Testing – A Logical Question and Answer Approach

Ning Chen, Ph.D., Professor,

Computer Science Department, California State University, Fullerton

Chapter 1 Introduction

Obviously, the first logical question is - what is Software Testing?

Maybe we can define the second term "Testing" first. If one searches the Internet, the following may pop up:

> *Dictionary Definition:*
>
> *testing*
>
> *adjective*
>
> *revealing a person's capabilities by putting them under strain; challenging "it's been quite a testing time for all of us"*
>
> *Powered by Oxford Dictionaries*

Well, if you want to use the above definition to define the term Software Testing. You may end up with

> *revealing software's capabilities by putting it under strain; challenging*

Most of us probably will say that this is a very reasonable way of explaining what Software Testing is.

If you want to go further, you may want to see the professional version of the definition:

> Software Testing Definition:
>
> According to IEEE Standard 829-2008:
>
> *Software testing: (1) An activity ① in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation ② is made of some aspect of the system or component. (2) To conduct an activity as in (1).*

Okay, not too bad! Most readers may feel comfortable at this point.

① An activity – This pretty much fits into the dictionary definition (*by putting it under strain; challenging - which is an activity*)

② An evaluation – This is the part that somehow is implicit in the dictionary definition. The professional definition spells it out explicitly.

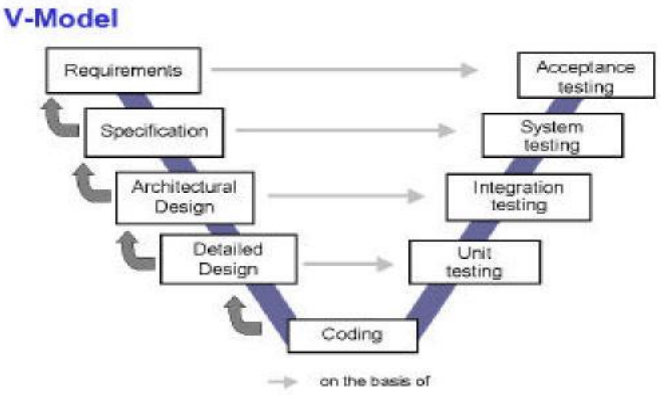
② Don't overlook the evaluation part. It may become much deeper than you expect. How to come up with a proper evaluation? How do you know your "proper evaluation" is proper? How do you *assure* that you do have a proper evaluation? You will encounter other terminologies when you dig in, for example, Verification and Validation, Software Quality Assurance, Software Quality Control, etc.

Now, you are ready to perform an activity in which a system or component is executed under specified conditions, the results are observed or recorded (let's ignore the evaluation part for the moment). The next logical question probably is a question that starts with when, where and how.

When① and Where② to perform a testing activity?

Wikipedia – V-Model (software development) gives the following:

> *In software development, the V-model represents a development process that may be considered an extension of the waterfall model, and is an example of the more general V-model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively.*



From http://sqa.fyicenter.com/FAQ/Software-Development-Models/Software_Development_Models_V_Model.html

①The V-Model does provide a sense of time line. For example, you perform unit testing when you work on that unit. When you have several units and you want to integrate them, you perform integration test.

② Does the V-Model also reveal a sense of place (where) you perform a testing activity? We think so. For example, the V-Model places unit testing at the bottom and integration testing. Above the unit testing, you may find system testing, acceptance testing. This implies you have different testing activities at different levels.

If you want to focus on *where* you conduct the testing activity, you may conclude that in terms of levels of testing①, we may have the following:

Unit Testing

Integration Testing

System Testing

Acceptance Testing

① Is this the only way we categorize different testing activities? No.

How to perform a testing activity?

We don't think there is a unique answer for this question. It all depends. In terms of whether or not you, as a tester, can access (or use) the source code, you may have:

White-Box Testing

Black-Box Testing

Wikipedia (White-box testing) gives the following:

> *White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing). In white-box testing an internal perspective of the system, as well as programming skills, are used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs.*

Wikipedia (Black-box testing) gives the following:

> *Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied to virtually every level of software testing: unit, integration, system and acceptance. It typically comprises most if not all higher level testing, but can also dominate unit testing as well.*

Do all people (researchers and practitioners) agree on those terms? Hardly! For example, still based on whether you want to involve program source code or not, you probably encounter the following:

Specification-Based Testing

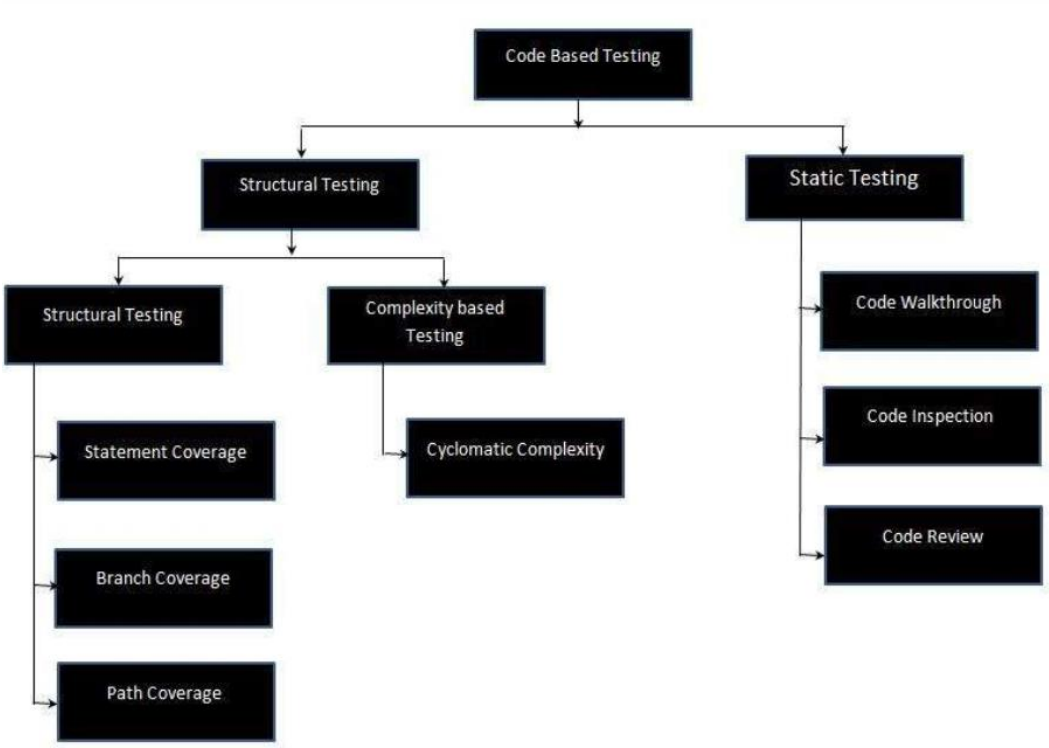
http://testingeducation.org/BBST/specbased/ gives the following:

> *Specification-based testing is a style of testing (a style is a collection of test-related activities and techniques) focused on*
>
> *discovering what claims are made in the specifications and*
>
> *testing the product against them.*

Code-Based Testing

http://www.tutorialspoint.com/software_testing_dictionary/code_based_testing.htm gives the following (including figure):

> Code-based testing corresponds to the testing that is carried out on code development, code inspection, unit testing① in software development process.
>
> The Code-based testing consists of following testing:
>
> Dynamic Testing - Statement coverage, Branch coverage, Path coverage
>
> Checking for Complexity of Code using techniques like Cyclomatic Complexit
>
> Static Testing - Code Inspection, Code Walkthrough, Code Review, Code Audit



① Some researchers may add that you can also apply code-based testing to integration testing.

Can we claim the following:

White-box testing is equivalent to Code-based testing and

Black-box testing is equivalent to Specification-based testing?

Well, we think so. Our experience is that most practitioner prefer to use the term "white-box, black-box" and researchers in academia prefer "code-based/specification-based."

The next logical question will be a pros/cons analysis on Specification-based (black-box) vs. Code-based (white-box).

TBD

Chapter 2 Examples of testing activities

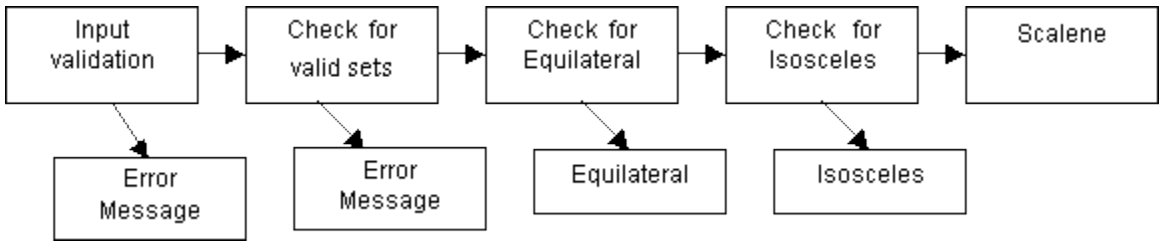
At this point a possible logical demand from you probably is to see a testing activity in action – a fair enough demand. There is a well-known test design example called Weinberg-Myers Triangle problem.

http://www.softwaretestpro.com/Item/4533/Nifty-Triangle-Test-Example/Software-Testing gives a simple definition of this problem:

> *The problem is simple - you have three inputs for sides of a triangle, and the computer will tell you if the values you entered are scalene, isosceles, equilateral, or not a triangle.*

Another website comes up with a nice flowchart that quickly guides us on how to proceed:

> We can make a basic and very rough flow diagram for the program:



> Input validation will ensure that any input other than a set of three integers will return some suitable error message. Next, the check for valid sets will verify that the relationships among the three elements of the input set are suitable for forming a triangle. Again, where these sets are not proper a suitable error message will be returned. Finally, given valid triangle sets these can be asses first for being equilateral and then isosceles. In each case the announcement of the type is returned. A valid set that is neither equilateral nor isosceles must be scalene, so that is the end result. Not included in the diagram is the implicit return path for another input set after any error message or triangle type determination is given.

For simplicity, we assume that this unit program has been written, but you are not given the source code. The question becomes how you test it.

> *The same website gives a nice approach below:*
>
> *Any element of an input set is negative*

Any element of an input set is zero

Any element of an input set is greater than 9③

Any element of an input set is an alphabetic letter

Any element of an input set is a character symbol

There are less than three elements of the input set

There are more than three elements of the input set

Any element of an input set is a combination of any of the invalid inputs

Next we want to test against the risk of sets of inputs of three numbers 1-9 that cannot form triangles. If any of these is input a suitable error message should result:

The sum of two numbers equals the third in any of three permutations a+b=c, a+c=b, b+c=a.

The sum of two numbers is less than the in any of three permutations a+b=c, a+c=b, b+c=a.

Next we want to ensure that given a proper set of inputs for a triangle the correct type is determined by the program:

If and only if all three digits of a valid input are the same the program displays that it has recognized an "equilateral" triangle

If and only if any two of the digits of a valid input are the same the program displays that it has recognized an "isosceles" triangle

If and only if each digit of a valid input is different the program displays that it has recognized a "scalene" triangle.

③ This is a condition just to simplify the problem. With it, a user is allowed to enter a number between 0 and 9 – not quite reasonable, but it does make the problem easier.

The above represents our good understanding of the testing challenge and based on common sense, one may come up with some good test cases. Indeed, the same website offers the following test cases:

Tests	*Inputs*	*Expected Results*
Any element of an input set is negative	*(-1,1,1), (1,-1,1), (1,1,-1)*	*Error Message*

Any element of an input set is zero (0,1,1), (1,0,1), (1,1,0), (1,0,0), (0,1,0), (0,0,1), (0,0,0) Error Message

Any element of an input set is greater than 9 (10,1,1), (1,11,1), (1,1,12) (15,15,15) Error Message

Any element of an input set is an alphabetic letter (T,1,1), (1,T,1), (1,5,T) Error Message

Any element of an input set is a character symbol (:,1,3), (5,., 8), (2,4,/) Error Message

There are less than three elements of the input set (1,2) Error Message

There are more than three elements of the input set (2,4,5,8) Error Message

Any element of an input set is a combination of any of the invalid inputs (1A, 3,4), (4,/W",8) (2,4,:) Error Message

The sum of two numbers equals the third in any of three permutations a+b=c, a+c=b, b+c=a (2,4,6), (4,9,5), (8,4,4) Error Message

The sum of two numbers is less than the third in any of three permutations a+b (3,3,8), (2,5,1), (7,3,3) Error Message

If all three digits of a valid input are the same the program displays that it has recognized an "equilateral" triangle (5,5,5) Equilateral !

If any two of the digits of a valid input are the same the program displays that it has recognized an "isosceles" triangle (3,3,4),(7,8,7),(5,6,6) Isosceles !

If each digit of a valid input is different the program displays that it has recognized a "scalene" triangle. (3,4,5), (3,5,4), (5,4,3) Scalene !

What is the knowledge gained from this well-known testing example? You may say, well, I understand the required features of the triangle program (that gives me some idea on the part of evaluation); I understand the strategy (flowchart) that allows me to test a unit software (probably a function); I can come with some test cases by which I can conduct a testing activity. Have I learned a lot? The answer probably is yes initially, then later on, after a deeper thinking, no. We do learn something here. Nevertheless, if you contemplate a bit, you may feel that the knowledge is kind of ad hoc. We don't have a systematic, generalized or holistic way of approach especially in terms of coming up test cases. BTW, it is not hard to conclude that this example, in terms of levels of testing, is a unit testing. In terms of white-box, black-box testing, it is a black-box testing. In terms of code-based, specification-based testing, it is a specification-based testing.

Chapter 3 Specification-Based, Unit, Boundary Value Testing

Do we have a way to come up with "enough" test cases for the triangular test example discussed in Chapter 2? If we do, can we claim that it is systematic, generalized, or holistic? Or, better, can it offer us some sense of confidence? Research done in the past claims that yes, we do have some ways. One of them is called Boundary Value Testing①.

①Strictly speaking, the full name of this testing method that applies to our triangle problem should be called specification-based, unit, and boundary value testing. Although no one wants to use this long name, it is essential to realize that in terms of use of source code, it is specification-based (instead of code-based) or, if you are a practitioner, black-box. In terms of levels, it is a unit testing for the given problem.

There are four forms of boundary value testing:

Normal boundary value testing

Robust normal boundary value testing

Worst-case value testing

Robust worst-case value testing

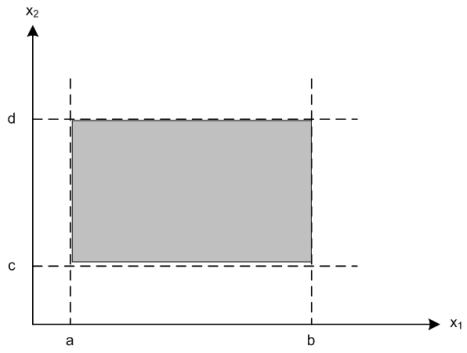
Logically, there are two questions arise. First, what is boundary value testing? Second, why do we have four different kinds of boundary value testing?

Wikipedia (Boundary testing) gives the following:

> *Boundary testing or boundary value analysis, is where test cases are generated using the extremes of the input domain, e.g. maximum, minimum, just inside/outside boundaries, typical values, and error values.*

Suppose you have two variables as the input domain, say, x1 and x2. We are also given that a <= x1 <= b and c <= x2 <=d. Graphically, "Software Testing – A Craftsman's Approach" by Paul C. Jorgensen gives the following

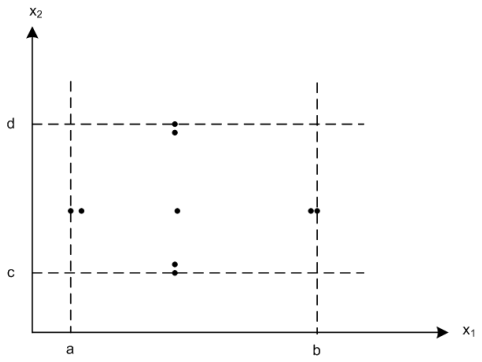
Input Domain of $F(x_1, x_2)$
where $a \le x_1 \le b$ and $c \le x_2 \le d$



If you pick maximum, minimum, just inside boundaries① and typical values②, you may end up with

(again, from "Software Testing – A Craftsman's Approach" by Paul C. Jorgensen)

Normal Boundary Value Test Cases



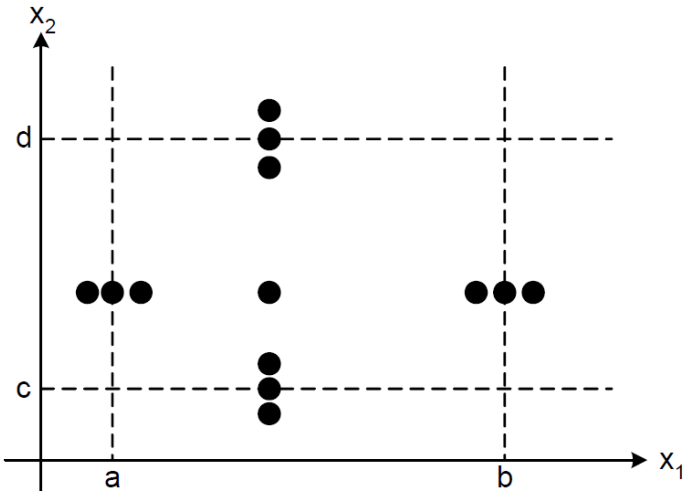
① If you notice, here, we do not include the outside boundaries. This is the reason that the selected dots on the above graph are falling inside the boundary.

② Selection of a typical value (or, you can call it a normal value) is easy. You select a value of x1 which is between a and b. Select a value of x2 which is between c and d.

The selection of other dots on the above graph requires a bit careful observation. If you hold x1 at its "normal" value③ and vary x2 to its at-the-boundary and just-inside-the-boundary, you end up with four more dots. If you repeat this step by holding x2 at its "normal" value and vary x1, you gain four more dots. Do we have a name that is used to label this type of selecting boundary values for testing? Yes. People call it Normal Boundary Value Testing.

③ Any justification why you want to hold one variable at its normal value and vary others? The reason is that the reliability theory claims that failures are rarely the result of the simultaneous occurrence of two or more faults. If you buy this theory, then you realize that when a failure occurs, mostly, all input values except one must be at the normal value. Not quite understand the above explanation? Let's go over a scenario play. Since the reliability theory says that most likely a failure is a result of the occurrence of one fault (that means something wrong with the processing of one input variable), we can bravely assume that there is one fault somewhere in the program and it is related to the input x1 and we want to hunt this fault down. How do you hunt this fault? The first test case is x1= typical value, x2= typical value. If no luck, your test case 2 is x1=upper inside-boundary value, x2=typical value. Your test case 3 is x1=upper at-boundary value, x2=typical value, test case 4 is x1=lower inside-boundary value, x2=typical value, and test case 5 is x1=lower at-boundary value, x2= typical. Okay, why do you vary x1 value? Well, the fault is associated with x1, so we vary x1 value to see whether we can catch it. How come you don't vary x2? We believe that the fault is associated with x1 only (single fault reliability theory) and x2 has nothing to do with it. Any x2 value is fine (we just pick a typical value). There is no need to vary x2 at all. What happens if your brave assumption that says the fault is associated with x1 is wrong? In this case varying x1 will not help in catching the fault. Easy, you simply repeat the above scenario starting with the assumption that the fault is associated with x2 (instead of x1).

How about the outside boundaries? If you want the outside boundaries, simply include them as shown below:

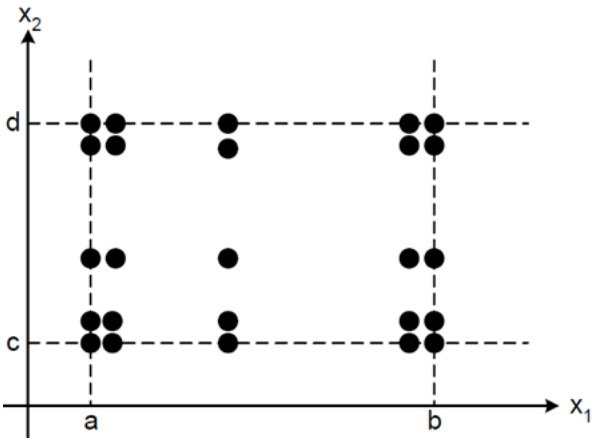


Including outside boundaries adds four more dots on the graph and deserves a new name and we call it Robust Normal Boundary Value Testing④.

④ Some literature calls it Robust Boundary Value Testing (omitting the term Normal). We don't like it. Shortening a name in some case is not that worthy. Without the term Normal in the

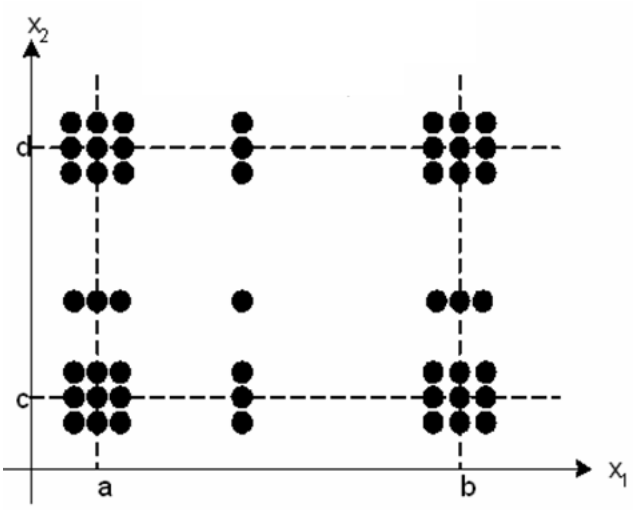
name, you may think this is a brand new approach. No. It is just an extension of the Normal Boundary Value Testing.

If you are not a believer of the reliability theory and worry about failures occurring at places when all variables having extreme values simultaneously, you end up with the following (for two variables):



This new consideration deserves a new name and we call it Worst-Cast Boundary Value Testing.

Wait a minute, how about the outside boundaries? Well, if you want to add that, here is the result:



What name should we use for this approach? The answer, not that surprising, is Robust Worst-Case Boundary Value Testing.

Revisit of the Triangle Problem

Let's assume three sides of a triangle is a, b and c. The valid range of a, b, and c is

0 <= a <=10

0 <= b <=10

0 <= c <=10

Can we come up with test cases for all four different tests (Normal boundary value testing, Robust normal boundary value testing, Worst-case boundary value testing, Robust worst-case boundary value testing)?

Examples:

TBD

TDB

Discussion: Have we missed anything?

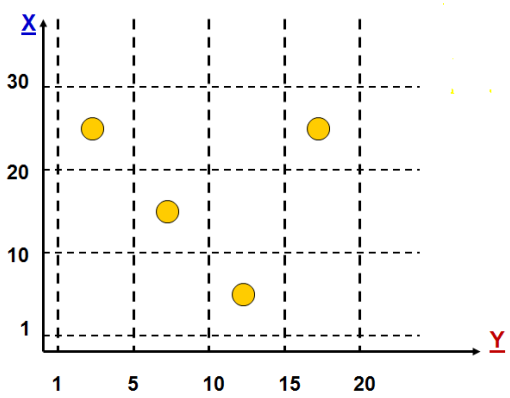
Chapter 4 Specification-Based, Unit, Equivalence Class Testing

Many people may oppose the use a long name. Nevertheless, spending a bit of time (at least the first time) to avoid some subtle confusion may have some value. This is the reason that we name this chapter as Specification-Based, Unit, Equivalence Class Testing. The use of the term Unit in the name generates a question. Can we use Equivalence Class Testing at other levels of testing (integration, system and acceptance testing)? The answer is yes. The Equivalence Class Testing can apply to all levels of testing, not just at the Unit level. The main reason (or shall we say, preference) is that we conduct all the explanations of this chapter at the Unit level. Maybe including the term Unit in the name keeps us from stretching ourselves to other claims (oh, btw, it applies to integration, system and acceptance testing) too fast and too soon.

We start this chapter by asking the why question. Why do we need another testing? Is the boundary value testing good enough? The boundary value testing is not bad. It offers us a systematic way of coming up with limited number of test cases. Nevertheless, we only test one normal value and the rest are around the boundaries. Will we have failures that occur not around the boundaries (say, maybe in between the normal value and the boundary)? Hard to say! Yes, we have covered one normal value and areas around the boundaries. Have we covered other "areas"? Should we?

Supposed we have a way to divide the input domain into several areas (or, better yet, some problems have an input area consisting several areas naturally), should we place one (or more) test case(s) in each area?

Let's see an example. Again, we assume that the input domain has two variables of x and y (so that it is easier to see graphically). X shows some kind of a natural separation of 1 to 10, 10 to 20, and 20 to 30 (3 separations). Y shows a separation of 1 to 5, 5 to 10, 5 to 15, and 15 to 20 (4 separations). If you plot the input domain, you will end up with a graph with 12 partitions as shown below:

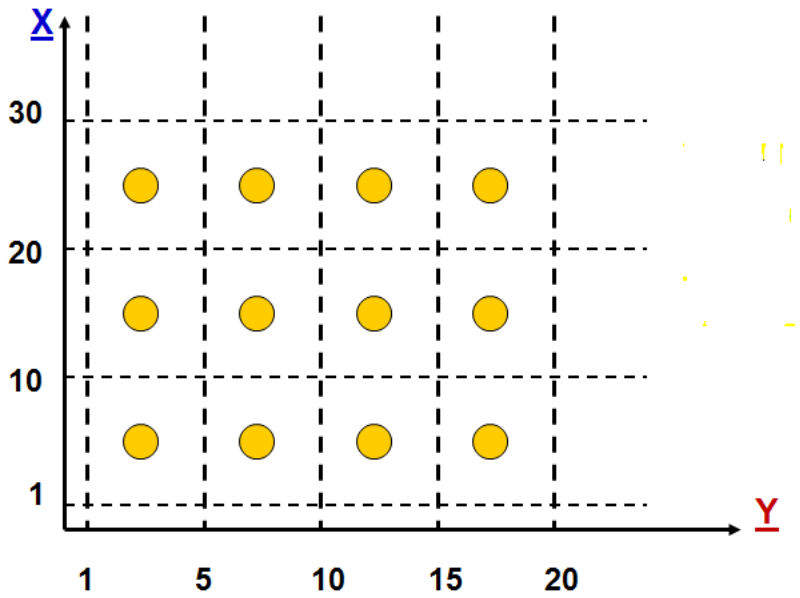


Now it is time to pick test cases. At this point, you may say, let's place one test case in each partition. We have 12 partitions, therefore, we should have 12 test cases (12 dots on the graph). Wait a minute! Someone only places 4 dots (not 12) on the graph. Why? Well, you have to realize that as a tester, we have conflicting goals. On one hand, we want to test all partitions to achieve a sense of "completeness." On the other hand, we also want to reduce the cost of testing (that means to use as few test cases as possible). If we can get away with fewer than 12 test cases, we should try. For those of you who believe in single fault assumption, it turns out, four test cases are sufficient (instead of 12). What do you mean "single fault assumption" anyway? This assumption states that most likely, a fault, when it occurs, can be traced back to things associated to a single input variable. For example, in our case, the input domain has two variables (x and y). When a fault occurs, we may say, well, this fault can be traced back to things (code, structure, whatever) related to the variable y, but not x. Now we want to do test to expose this fault. What should we do? Notice that the variable y has four partitions. Obviously, we want to have at least one test case in each of the y partition. If you check the partition line along the Y variable, indeed, the above given graph shows four dots. How about if we want to test a fault that is associated with the x variable? Do we have at least one test case in each partition of the x variable? Indeed, if you inspect the graph along the x variable, in each of its partition, you can find at least one dot①.

① Most readers find out that in the x variable partition between 20 and 30, we actually have two dots (two test cases). Yes. It is redundant here, but that extra one actually is there to cover the y variable test cases.

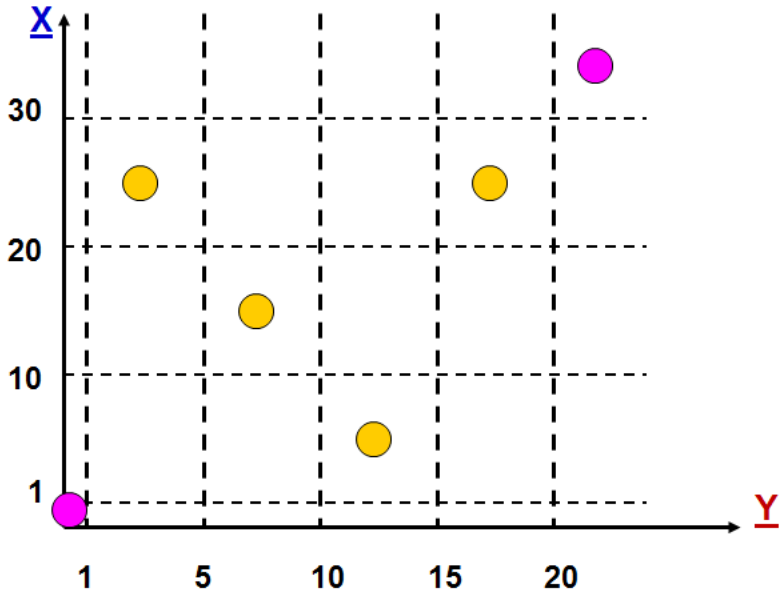
We need to come up with a name that describes what we have done here. The literature calls it Weak Normal Equivalence Class Testing. If we have a chance to name this approach ourselves (which we don't), we really prefer to call it Single-Fault Normal Equivalence Class Testing to remind ourselves that the single-fault assumption is the real "beef" here.

What if you don't buy the single fault assumption and believe that a fault may be associated with things that are a function of multiple variables? Well, easy, you simply place at least one test case in each possible partition combinations③ as shown below:



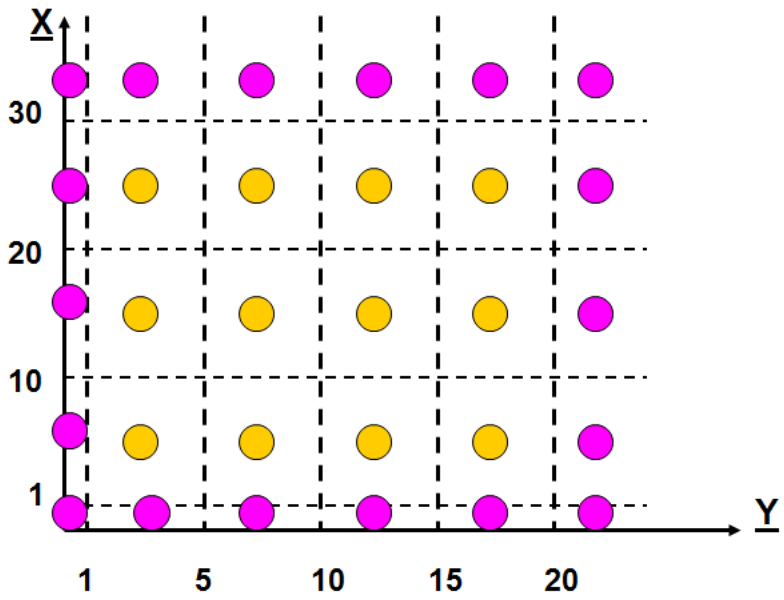
In literature this way of selecting test cases is called Strong Normal Equivalence Testing. Is this the best possible name. We don't thinks so. If we have our way (which we don't), we really prefer to call it Multiple Fault Normal Equivalence Testing.

Do we need to worry about areas outside the boundary of our partitions? It is up to you. If you do, we will have the following:



This approach is called Weak Robust Equivalence Testing. Again, if we have our way, we prefer to call it Single Fault Robust Equivalence Testing.

If we accept the multiple fault assumption, we will end up with



This approach, not surprising, is called Strong Robust Equivalence Class Testing. If we have our way, we prefer to call it Multiple Fault Robust Equivalence Class Testing.

Example

TBD

Chapter 5 Specification-Based Unit, Decision Table-Based Testing

Most readers probably hate our long name. Please bear with us. We put the term Unit in to remind ourselves that all the explanations are done at the Unit level. Can you apply the Decision Table-Based Testing to other levels? Yes, but at this point, let's not emphasize it.

Before we have any in-depth academic-style of discussion, we prefer to begin with an observation.

[http://reqtest.com/requirements-blog/a-guide-to-using-decision-tables/] starts the discussion with a simple scenario as below.

Let's take an example scenario for an ATM where a decision table would be of use.

A customer requests a cash withdrawal. One of the business rules for the ATM is that the ATM machine pays out the amount if the customer has sufficient funds in their account or if the customer has the credit granted. Already, this simple example of a business rule is quite complicated to describe in text. A decision table makes the same requirements clearer to understand:

Conditions	**R1**	**R2**	**R3**
Withdrawal Amount <= Balance	T	F	F
Credit granted	-	T	F
Actions			
Withdrawal granted	T	T	F

*In a decision table, **conditions are usually expressed as true (T) or false (F)**. Each column in the table corresponds to a rule in the business logic (and therefore the column's names, R1, R2…) that describes the unique combination of circumstances that will result in the actions. The table above contains three different business rules, and one of them is the "withdrawal is granted if the requested amount is covered by the balance①." It is normal to create at least one test case per column, which results in full coverage of all business rules.*

*One advantage of using decision tables is that they **make it possible to detect combinations of conditions that would otherwise not have been found** and therefore not tested or developed. The requirements become much clearer and you often realize that some requirements are illogical, something that is hard to see when the requirements are only expressed in text.*

*A disadvantage of the technique is that **a decision table is not equivalent to complete test cases** containing step-by-step instructions of what to do in what order. When this level of detail is required, the decision table has to be further detailed into test cases.*

① Actually, this business rule statement ends up with two rules. The first one is "withdrawal is granted if the requested is covered by the balance and the customer has the credit granted." The second rule is "withdrawal is granted if the requested is covered by the balance and the customer has the credit not granted." For convenience, since the second condition "the customer has the credit granted" is a "don't care" case, we can simple combine two rules into one "withdrawal is granted if the requested is covered by the balance." The don't care is represented by a dash line "-" in the table.

Observing another example may be beneficial. [www.compaid.com/caiinternet/ezine/rex-decisiontables.pdf] gives the following:

Consider an e-commerce application like the one found on our RBCS Web site, www.rbcs-us.com. At the user interface layer, we need to validate payment information, specifically credit card type, card number, card security code, expiration month, expiration year, and cardholder name. You can use boundary value analysis and equivalence partitioning to test the ability of the application to verify the payment information, as much as possible, before sending it to the server.
So, once that information goes to the credit card processing company for validation, how can we test that? Again, we could handle that with equivalence partitioning, but

there are actually a whole set of conditions that determine this processing:
Does the named person hold the credit card entered, and is the other information correct?
Is it still active or has it been cancelled?
Is the person within or over their limit?
Is the transaction coming from a normal or a suspicious location?
The decision table in Table 1 shows how these four conditions interact to determine which of the following three actions will occur:
Should we approve the transaction?
Should we call the cardholder (e.g., to warn them about a purchase from a strange place)?
Should we call the vendor (e.g., to ask them to seize the cancelled card)?
Take a minute to study the table to see how this works. The conditions are listed at the top left of the table, and the actions at the bottom left. Each column to the right of this left-most column contains a business rule. Each rule says, in essence, "Under this particular combination of conditions (shown at the top of the rule), carry out this particular combination of actions (shown at the bottom of the rule)."

Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Real account?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Active account?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Within limit?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Location okay?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Actions																
Approve?	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Call cardholder?	N	Y	Y	Y	N	Y	Y	Y	N	N	N	N	N	N	N	N
Call vendor?	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

Notice that the number of columns—i.e., the number of business rules—is equal to 2 (two) raised to the power of the number of conditions. In other words, 2 times 2 times 2 times 2, which is 16. When the conditions are strictly Boolean—true or false—and we're dealing with a full decision table (not a collapsed one), that will always be the case. Did you notice how I populated the conditions? The topmost condition changes most slowly. Half of the columns are Yes, then half No. The condition under the topmost changes more quickly but more slowly than all the others. The pattern is quarter Yes, then quarter No, then quarter Yes, then quarter No. Finally, for the bottommost condition, the alternation is Yes, No, Yes, No, Yes, etc. This pattern makes it easy to ensure you don't miss anything. If you start with the topmost condition, set the left half

of the rule columns to Yes and the right half of the rule columns to No, then following the pattern I showed, if you get to the bottom and the Yes, No, Yes, No, Yes, etc., pattern doesn't hold, you did something wrong.

Deriving test cases from this example is easy: Each column of the table produces a test case. When the time comes to run the tests, we'll create the conditions which are each test's inputs. We'll replace the "yes/no" conditions with actual input values for credit card number, security code, expiration date, and cardholder name, either during test design or perhaps even at test execution time. We'll verify the actions which are the test's expected results.

In some cases, we might generate more than one test case per column. I'll cover this possibility in more detail later, as we enlist our previous test techniques, equivalence partitioning and boundary value analysis, to extend decision table testing.

Collapsing Columns in the Table

Notice that, in this case, some of the test cases don't make much sense. For example, how can the account not be real but yet active? How can the account not be real but within limit? This kind of situation is a hint that maybe we don't need all the columns in our decision table.

We can sometimes collapse the decision table, combining columns, to achieve a more concise—and in some cases sensible—decision table. In any situation where the value of one or more particular conditions can't affect the actions for two or more combinations of conditions, we can collapse the decision table.

This involves combining two or more columns where, as I said, one or more of the conditions don't affect the actions. As a hint, combinable columns are often but not always next to each other. You can at least start by looking at columns next to each other.

To combine two or more columns, look for two or more columns that result in the same combination of actions. Note that the actions must be the same for all of the actions in the table, not just some of them. In these columns, some of the conditions will be the same, and some will be different. The ones that are different obviously don't affect the outcome. So, we can replace the conditions that are different in those columns with the dash character ("-"). The dash usually means either I don't care, it doesn't matter, or it can't happen, given the other conditions.

Now, repeat this process until the only further columns that share the same combination of actions for all the actions in the table are ones where you'd be combining a dash with Yes or No value and thus wiping out an important distinction for cause of action. What I mean by this will be clear in the example I present in a moment, if it's not clear already.

Another word of caution at this point: Be careful when dealing with a table where more

than one rule can apply at one single point in time. These tables have non-exclusive rules. We'll discuss that further later in this section.

Table 2 shows the same decision table as before, but collapsed to eliminate extraneous columns. Most notably, you can see that columns 9 through 16 in the original decision table have been collapsed into a single column.

Conditions	1	2	3	5	6	7	9
Real account?	Y	Y	Y	Y	Y	Y	N
Active account?	Y	Y	Y	N	N	N	-
Within limit?	Y	Y	N	Y	Y	N	-
Location okay?	Y	N	-	Y	N	-	-
Actions							
Approve?	Y	N	N	N	N	N	N
Call cardholder?	N	Y	Y	N	Y	Y	N
Call vendor?	N	N	N	Y	Y	Y	Y

Decision Table Example (Collapsed)

I've kept the original column numbers for ease of comparison. Again, take a minute to study the table to see how I did this. Look carefully at columns 1, 2, and 3. Notice that we can't collapse 2 and 3 because that would result in "dash" for both "within limit" and "location okay." If you study this table or the full one, you can see that one of these conditions must not be true for the cardholder to receive a call. The collapse of rule 4 into rule 3 says that, if the card is over limit, the cardholder will be called, regardless of location. The same logic applies to the collapse of rule 8 into rule 7.

Notice that the format is unchanged. The conditions are listed at the top left of the table, and the actions at the bottom left. Each column to the right of this left-most column contains a business rule. Each rules says, "Under this particular combination of conditions (shown at the top of the rule, some of which might not be applicable), carry out this particular combination of actions (shown at the bottom of the rule, all of which are fully specified)."

Notice that the number of columns is no longer equal to 2 raised to the power of the number of conditions. This makes sense, since otherwise no collapsing would have occurred. If you are concerned that you might miss something important, you can always start with the full decision table. In a full table, because of the way you generate it, it is guaranteed to have all the combinations of conditions. You can mathematically check if it does. Then, carefully collapse the table to reduce the number of test cases you

create. Also, notice that, when you collapse the table, that pleasant pattern of Yes and No columns present in the full table goes away. This is yet another reason to be very careful when collapsing the columns, because you can't count on the pattern or the mathematical formula to check your work.

Chapter 6 Code-Based, Unit, Path Testing (Or, Structure-Based, Unit, Control Flow Testing)

Again, code-based path testing does apply beyond the unit testing. We prefer to include the name "unit" to stress the fact that all the explanation we have in this Chapter uses unit testing as example. The reason we also include a second title (Control Flow Testing) is that unfortunately, there is no consensus on the proper name. In general, structure-based means code-based. Path testing deals with the flow of control, which obviously flows through the paths of the program. It is perfectly acceptable to name it Structure-Based, Unit, Control Flow Testing.

The desire to involve source/binary code in testing is not that surprising. Although the specification-based testing methods (boundary value testing, equivalence class testing, decision table testing) do offer ways of exposing faults. Our confidence level may not be high and we may want (or need) to do more.

To increase the confidence level one needs to look at the code (statement) of the program. Obviously, looking at each individual statement some of the time is fine, but looking at each individual statement all the time will be very tedious and may not be productive. We need some way to reduce the complexity (or to achieve abstraction). How to abstract a given program is not a new challenge. Important research has been done since 1970s and this chapter's main goal is to "regurgitate" the research results to the readers. There are two main ways of "regurgitating knowledge." One is bottom-up and the other is top-down. In a bottom-up approach, the readers are given parts (definitions, theories, proofs …) first, then some conclusions/results later. Nothing wrong is the bottom-up approach, but, quite often, by the time you finished the parts without seeing any big pictures, you are so tired and may lose interest. We prefer to use the top-down approach in our way of "regurgitating" code-based testing methods to our readers. The top-down approach starts with motivation, followed by observation, and then some common-sense solution. For sure, the common-sense solution is never rigorous (but, it does offer an intuitive, intimate and holistic understanding of the issue). If the readers want to make sure that the common-sense solution stands in all situations, they may want to dig out the original (first hand) research results instead of the regurgitated second-hand information offered by a textbook.

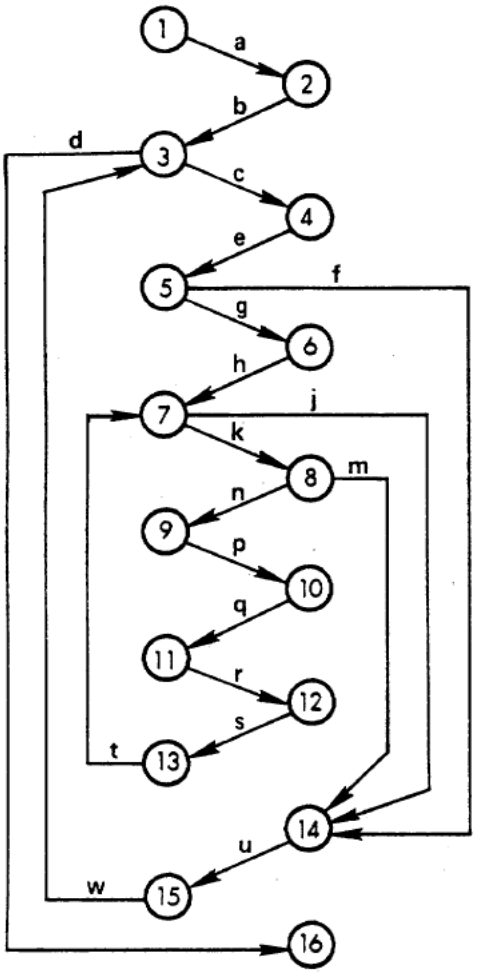
The top-down approach starts with abstraction. Imaging that you are a user of google maps, you want to use the zoom in-and-out feature. Seeing things at the street level is good, but, from time to time, you really want to see things from high above. When we look at a given source code, we also want to have the zoom in-and-out capability (it is called abstraction).

```
 1   SUBROUTINE BUBBLE (A,N)
 2      BEGIN
 3      FOR I = 2 STEPS 1 UNTIL N DO
 4          BEGIN
 5          IF A(I) GE A(I-1) THEN GOTO NEXT
 6          J = I
 7      LOOP:  IF J LE 1 THEN GOTO NEXT
 8         IF A(J) GE A(J-1) THEN GOTO NEXT
 9         TEMP = A(J)
10         A(J) = A(J-1)
11         A(J-1) = TEMP
12         J = J - 1
13         GOTO LOOP
14      NEXT: NULL
15         END
16      END
```

We start our observation with an example from an old① research result [Paige, 1977].

①You can tell it is old – goto statements are used all over the place.

If we consider each program statement as a node, we may end up with the following "Program Graph."



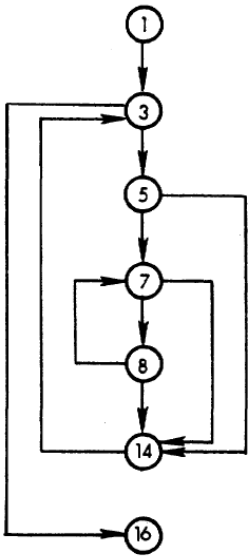
Label	Edge	Label	Edge
a	(1,2)	m	(8,14)
b	(2,3)	n	(8,9)
c	(3,4)	p	(9,10)
d	(3,16)	q	(10,11)
e	(4,5)	r	(11,12)
f	(5,14)	s	(12,13)
g	(5,6)	t	(13,7)
h	(6,7)	u	(14,15)
j	(7,14)	w	(15,3)
k	(7,8)		

How do we reduce the complexity of the given program graph? One intuitive idea is [Paige, 1977]:

> *A segment is a block of consecutively executable statements/ nodes with one entry and one exit; any statement within a particular segment is executed if and only if all statements*

within that segment are executed and each statement is found in one and only one segment.

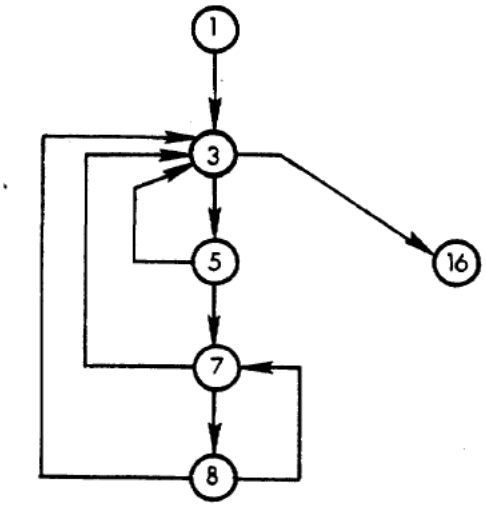
If we apply the above idea to our program graph, we end up with a "segment graph" as shown below:



An easy observation reveals that we hide some nodes from the program graph to construct the segment graph. For example, we keep nodes 1 and 3, but hide node 2. Do you delete node 2? No. We can't (if you delete node 2, you change the original program). In a sense, we simple "merge" node 2 to the edge. In our case, the edge between nodes 1 and 3 actually contains two edges now (edge a and edge b).

If you examine the segment graph, you may observe that some nodes are decision nodes that are related to decision making. For example, node 3 offers two outward edges – one goes to node 5 and the other goes to node 16. By inspection we have the following decision nodes: nodes 3, 5, 7 and 8. If you consider the entry node and exit node are decision node (although they don't have multiple outgoing edges, they do represent entry decision or exit decision in some way), our decision nodes now are: node 1, 3, 5, 7, 8 and 16. The only non-decision node is node 14.

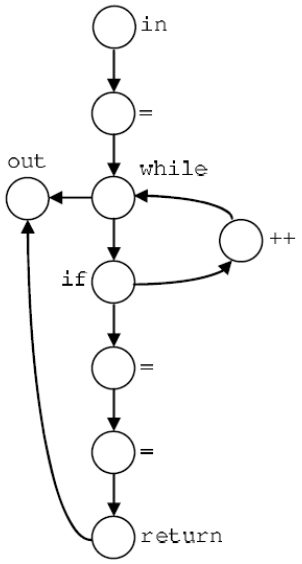
If we hide the non-decision nodes in the segment graph we discussed, we will end up with another level of abstracted graph, called Decision Graph (or Decision Path Graph, DD-path Graph) as follows:



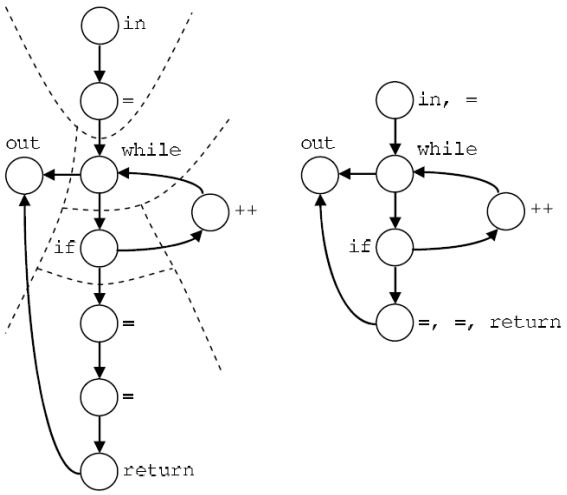
Note that we hide node 14 and merge its property to the edge of nodes 5, 7 and 8. For example, in the segment graph, node 5 connects to node 14 and then goes to node 3 eventually. Once you hide node 14, then node 5 connects to node 3 directly. Sure enough, you also need to come up with a definition of the new edge that connects node 5 to node 3 (it is a combination of the old edge from node 5 to node 14 and the edge from node 14 to node 3).

Since in our top-down approach we totally rely on our observation and intuition, seeing another example maybe is helpful. We found another example from [Robert Gold, 2010] as follows:

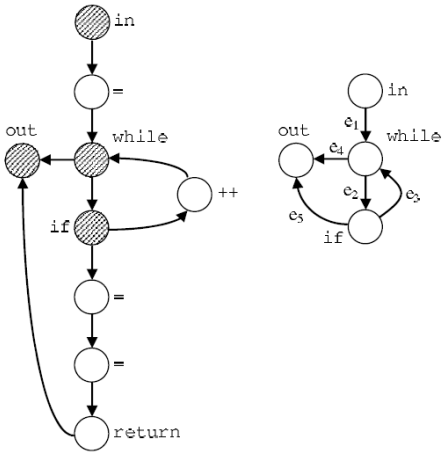
```
void Search(int arr[], int key,
            int *found, int *index)
{
  int i = 0;
  int b;

  *found = 0;

  while (i < N)
  {
    if (b = isabsequal(arr[i], key))
    {
      *found = b;
      *index = i;
      return;
    }

    i++;
  }
}
```



The program graph above can be abstracted into a segment graph as below:

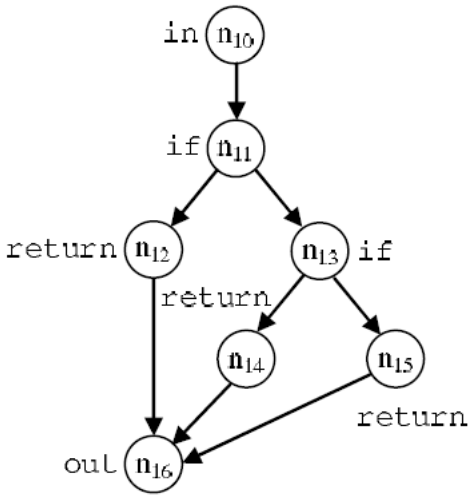


The next step is to abstract further and obtain a decision graph as below:

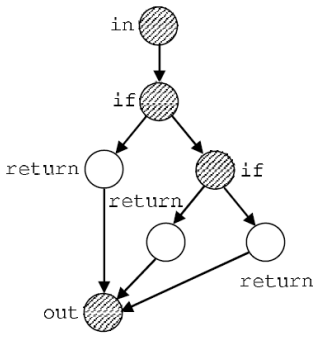


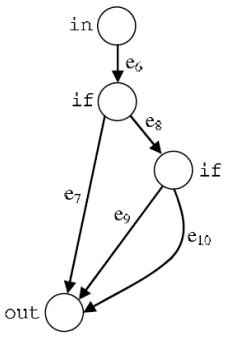
The Search function comes with a subroutine. We may want to analyze it as follows:

```
int isabsequal(int x, int y)
{
  if (x == y)
    return 1;
  else if (x == -y)
    return -1;
  else
    return 0;
}
```



Its segment graph and decision graph are:





After seeing those two examples, do we know how to reduce a program graph to a segment graph and then, to a decision graph? Well, sort of, but not exactly. We really are not so sure our simple common sense observation applies to all possible cases. At this point, our urgent job is not trying to prove something (in fact, someone has done it already). Our urgent task actually is to understand what benefits the abstracted graphs (segment graph and decision graph) can bring to us.

An example from [Robert Gold, 2010] gives us some good clues.

Let's say we need to test the program *p* that consists of the functions Search and isabsequal.

We have the following test cases:
Test case #1:

input arr == {1, 2, 3, 4},
key == -3

We will cover the following decision graph edges:

*e*1 *e*2 *e*3 *e*2 *e*3 *e*2 *e*5
*e*6 *e*8 *e*10
*e*6 *e*8 *e*9

Test case #2
input arr == {1, 2, 3, 4},
key == 3

Test case #2 passes a new edge:

*e*6 *e*7

Test case #3

arr == {1, 2, 3, 4},
key == 0,

a new path *e*1 *e*2 *e*3 *e*2 *e*3 *e*2 *e*3 *e*2 *e*3 *e*4 is executed.

At this point, all edges of the decision graphs are covered.

If your test cases cover all the edges of a decision graph, what have you achieved? Well, if you realize that an edge of a decision node represents a branch of the program. If your test cases cover all the edges of the decision graph, you have tested all the branches of your program.

One subtle and tricky (but important) question is that if you cover all the nodes of the decision graph (NOT all the edges of the decision graph), can you claim that you cover all the statements of a given program?

The above test cases actually reveal the answer (the answer is no). If you only use Test case #1 and Test case #3 (do not use Test case #2), you still cover (or traverse) all the nodes of the decision graph. It is obviously that without using Test case #2, you will never cover the node 12 of the program graph.

As for the program graph, we may conclude the following:

1. Given a program, you can find its program graph.
2. When you run a test case through the program graph, you know what nodes of the program graph have been traversed.
3. If your test cases that collectively traverse all nodes of the program graph, you can say that your test cases produce 100% statement coverage.

As for the decision graph, we may conclude the following:

1. Given a program graph you can find its decision graph.
2. When you run a test case through the program's decision graph, you know what edges of the decision graph have been traversed.
3. If your test cases that collectively traverse all edges of the decision graph, you can say that your test cases produce 100% branch coverage.

Basis Path Testing

100 % statement coverage may not be as good as you think.
http://www.onjava.com/pub/a/onjava/2007/03/02/statement-branch-and-path-coverage-testing-in-java.html?page=2 gives the following example:

The following example demonstrates this. Here, the `returnInput()` method is made up of seven statements and has a simple requirement: its output should equal its input.

```
 * Copyright (c) 2004-2006 Codign Software, LLC.
 *
 * All rights reserved. This program and the accompanying materials are made
 * available under the terms of the Eclipse Public License v1.0 which
 * accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 ******************************************************************************/


package com.codign.sample.pathexample;

public class PathExample {

    public int returnInput(int x, boolean condition1,
                                  boolean condition2,
                                  boolean condition3) {
        if (condition1) {
            x++;
        }
        if (condition2) {
            x--;
        }
        if (condition3) {
            x=x;
        }
        return x;
    }
}
```

Figure 1. Code sample

Next, you can create one JUnit test case that satisfies the requirement and gets 100 percent statement coverage.

```
/**
 Test method for 'com.codign.sample.pathexample.PathExample.returnInput(int,
                                                  boolean, boolean, boolean)'
 @CoViewTest (coview_methodundertest=com.codign.sample.pathexample.
               PathExample#returnInput(int,boolean,boolean,boolean))*/
public void testReturnInputIntBooleanBooleanBooleanTTT() {
    PathExample constructorInstance = new PathExample();

    // Method under test
    int methodReturn = constructorInstance.returnInput(0, true, true, true);
    assertEquals(0, methodReturn);

}
```

Figure 2. Statement coverage

There's an obvious bug in `returnInput()`. If the first or second decision evaluates true and the other evaluates false, the return value will not equal the method's input. An astute software developer will notice this right away, but the statement coverage report shows 100 percent coverage. If a manager sees 100 percent coverage, he or she may get a false sense of security, decide that testing is complete, and release the buggy code into production.

Recognizing that statement coverage may not fit the bill, the developer decides to move on to a better testing technique: branch coverage.

Branch Coverage

A branch is the outcome of a decision, so branch coverage simply measures which decision outcomes have been tested. This sounds great because it takes a more in-depth view of the source code than simple statement coverage, but branch coverage can also leave you wanting more.

Determining the number of branches in a method is easy. Boolean decisions obviously have two outcomes, true and false, whereas switches have one outcome for each case—and don't forget the default case! The total number of decision outcomes in a method is therefore equal to the number of branches that need to be covered plus the entry branch in the method (after all, even methods with straight line code have one branch).

In the example above, `returnInput()` has seven branches—three true, three false, and one invisible branch for the method entry. You can cover the six true and false branches with two test cases:

```
/**
 Test method for 'com.codign.sample.pathexample.PathExample.returnInput(int,
                                                    boolean, boolean, boolean)'
 @CoViewTest (coview_methodundertest=com.codign.sample.pathexample.
                PathExample#returnInput(int,boolean,boolean,boolean))*/
public void testReturnInputIntBooleanBooleanBooleanTTT() {
    PathExample constructorInstance = new PathExample();

    // Method under test
    int methodReturn = constructorInstance.returnInput(0, true, true, true);
    assertEquals(0, methodReturn);

}

/**
 Test method for 'com.codign.sample.pathexample.PathExample.returnInput(int,
                                                    boolean, boolean, boolean)'
 @CoViewTest (coview_methodundertest=com.codign.sample.pathexample.
                PathExample#returnInput(int,boolean,boolean,boolean))*/
public void testReturnInputIntBooleanBooleanBooleanFFF() {
    PathExample constructorInstance = new PathExample();

    // Method under test
    int methodReturn = constructorInstance.returnInput(0, false, false, false);
    assertEquals(0, methodReturn);
}
```

Figure 3. Branch coverage

> *Both tests verify the requirement (output equals input) and they generate 100 percent branch coverage.* But, even with 100 percent branch coverage, the tests missed finding the bug. *And again, the manager may believe that testing is complete and that this method is ready for production.*
>
> *A savvy developer recognizes that you're missing some of the possible paths through the method under test. The example above hasn't tested the* TRUE-FALSE-TRUE *or* FALSE-TRUE-TRUE *paths, and you can check those by adding two more tests.*
>
> *There are only three decisions in this method, so testing all eight possible paths is easy. For methods that contain more decisions, though, the number of possible paths increases exponentially. For example, a method with only ten Boolean decisions has 1024 possible paths. Good luck with that one!*
>
> *So, achieving 100 percent statement and 100 percent branch coverage may not be adequate, and testing every possible path exhaustively is probably not feasible for a complex method either. What's the alternative? Enter basis path coverage.*

Before we try to understand the basis path, maybe we want to try our intuition first. The given example has 3 decision statements in a row. Each decision gives us two possible branches. This

gives us 8 possible paths that go through all possible combinations of branches①. In order to catch the bug, you may decide to have test cases that cover all 8 possible paths.

①If you go through the true branch of the first decision, then the false branch of the second decision and finally the true branch of the third decision, can we call this combination of three branches as a new branch? Well, we don't want to call it this way. It is just confusion. We use the term path to describe this combination of branch.

Sure enough with test cases that cover all 8 paths, you found the bug. But, if you have 10 decisions, what happens? Your test cases need to cover 1024 possible paths. That is just too much!

Is there a way to catch the bug without covering all 8 paths (or 1024 paths if we have 10 decisions in the program)? For our example, if you inspect the code carefully, you may find the path (True, False, True) catches the bug. Nevertheless, this is an ad hoc approach and requires a careful analysis of the code.

A field of research starting by [McCabe, 1982] promotes the idea that some paths are more special/important/significant than others. This line of research claims that if you treat each path as a vector, you may discover that a group of vectors can be used to construct all other vectors. In math, we call the vectors in this special group as independent vectors. For our case, this means that you may discover a group of paths (independent paths) that can be used to construct all the rest paths. Since all the rest of paths are mathematically dependent of the independent paths, we may further claim that testing of all independent paths implies testing of all possible paths. This is a big claim and you may object. Well, maybe we can reduce the claim a bit lighter as follows:

Testing of all independent paths may give us a sense of completeness of testing all possible paths, since all the rest of paths are mathematically dependent of the independent paths.

To some degree, it is not that hard for us to accept this idea. If you recall, we pretty much follow the same line of thinking in our treatment of the equivalence class testing.

Do we have a name for this group of independent paths? Yes, we call them Basis Paths (not a bad name!). As a result, this particular testing method is called Basis Path Testing which is a sub-class of Path Testing.

The next logical question is how do you determine how many paths are basis paths and how do find them?

This first question is not as hard as you may think. If you don't have any decision in your program, you end up with only one path (just from start to exit). In this case, the number of the basis paths is 1①. If you have one two-way decision in your program, your total number of paths is 2 and the number of independent paths is 2②. If you have two two-way decisions in your

program, your total number possible paths is 4. Nevertheless, the number of independent paths (basis paths) is 3③. If you have three two-way decisions in your program, your total number of paths is 8 and the number of independent paths (basis paths) is 4④. If you are not tired, let's do that one more time. If you have four two-way decisions in your program, your total number of paths is 16 and the number of independent paths (basis paths) is 5.

① You only have 1 path in total and for sure, that path is the independent path.

② You only have 2 paths in total. Is the second path independent of the first path? The answer is yes, since the second path is the result of "flipping" the decision from "true" to "false" (or from "false" to "true") at the decision point. Mathematically, two vectors (remember that we view paths as vectors) are "orthogonal" (a fancy way of saying independent).

③ Now you have two two-way decision in your program. You will end up with 4 possible paths in total and they are TT (stands for True, True), TF, FT, and FF. We pick TT as our first independent path. By flipping the logic at the first decision only, we end up with FT. We claim that FT is independent of TT. Indeed, you really can't construct FT using TT. Now we flip the logic at the second decision only, we end up with TF. We claim that you really can't construct TF from TT or from FT or from a linear combination of TT and FT. Therefore, we say TF is an independent path. How about FF? Is FF an independent path? Well, mathematically TT is expressed by a vector of (1,1) and FF is expressed by a vector of (0,0). The vector (0,0) can be considered as (1,1) – (1,1). Therefore, it is not an independent path. Wow! What do you mean negating TT minus TT? Do you have any intuition on the notion of negating a path? Well, unfortunately, this line of research does not offer good interpretation but simply claims the dependency in the mathematical sense. The end result is that the number of basis paths is 3.

④ Now, it becomes a bit tedious. For readers with keen observations, they probably get it already and don't want to go over one more run. Please bear with us here. We just feel that sometimes leaving a subject too soon and too fast may not be beneficial, at least, to some readers who may need more time to contemplate. In the case of 3 decisions, the total number of possible paths is 8 (2^3=8) and they are TTT, TTF, TFT, TFF, FTT, FTF, FFT, and FFF. We pick TTT as our first independent path. The second independent path is obtained by flipping the logic at the first decision only and we end up with FTT. The third independent path is obtained by, still starting from TTT, flipping the logic at the second decision only and we end up with TFT. The fourth independent path is obtained, starting from TTT, by flipping the logic at the third decision only and we end up with TTF. Now, we have four independent paths (TTT, FTT, TFT and TTF) as our basis paths. Therefore, the number of basis paths is 4. How come, for example, FTF is not an independent path? If you try a bit harder, you soon realize that FTF when expressed as a vector is (0,1,0) can be expressed by (1,1,1) – (1,0,1), a linear combination of two independent paths. Or, if you insist, FTF=TTT – TFT. Wow! A path can be a result of one path minus the other! What does that mean? Again, this line of research does not offer good physical intuition.

Nevertheless, mathematically we know (or feel) that the basis paths are special. By testing the basis paths we may avoid testing all possible paths (which is economically not feasible in most cases).

By now, the readers should have an intuitive way of calculating the number of basis paths for a given program (in literature, it is called Cyclomatic Complexity) is simply, number of 2-way decisions + 1.

Cyclomatic Complexity = Number of 2-Way Decisions + 1

Or,

Cyclomatic Complexity = e – n + 2p

Where

e = number of edges for a given program graph (or called control graph)

n = number of nodes

p= number of connected regions (for most cases, usually is 1)

It is time to bring back the example given by
http://www.onjava.com/pub/a/onjava/2007/03/02/statement-branch-and-path-coverage-testing-in-java.html?page=2

To achieve 100 percent basis path coverage, you need to define your basis set. The cyclomatic complexity of this method is four (one plus the number of decisions), so you need to define four linearly independent paths. To do this, you pick an arbitrary first path as a baseline, and then flip decisions one at a time until you have your basis set.

***Path 1**: Any path will do for your baseline, so pick true for the decisions' outcomes (represented as* TTT*). This is the first path in your basis set.*

***Path 2**: To find the next basis path, flip the first decision (only) in your baseline, giving you* FTT *for your desired decision outcomes.*

***Path 3**: You flip the second decision in your baseline path, giving you* TFT *for your third basis path. In this case, the first baseline decision remains fixed with the true outcome.*

***Path 4** : Finally, you flip the third decision in your baseline path, giving you* TTF *for your fourth basis path. In this case, the first baseline decision remains fixed with the true outcome.*

So, your four basis paths are TTT, FTT, TFT, *and* TTF. *Now, make up your tests and see what happens.*

In the attached code, you can see that `testReturnInputIntBooleanBooleanBooleanTFT()` *and* `testReturnInputIntBooleanBooleanBooleanFTT()` *found the bug that was missed by your statement and branch coverage efforts. Further, the number of basis paths grows linearly with the number of decisions, not exponentially, keeping the number of required tests on par with the number required to achieve full branch coverage. If fact, because basis path testing covers all statements and branches in a method, it effectively subsumes branch and statement coverage.*

Chapter 7 Code-Based Data Flow Testing

As an opposite way of Control Flow Testing, we may pay attention to the data flowing through the structure of the program (instead of the flowing of the control). This type of testing methods is under the category of Code-Based Data Flow Testing (also called Structure-Based Data Flow Testing).

Why do we want to pay attention to flowing of data? Hunting down bugs is not an easy task. In a real-world scenario, the program under test is just too big or too complicated. Instead of viewing the given program as a whole, maybe we can apply the divide-and-conquer strategy. By following one portion or part of the input data set that flows through the program, we try to divide the whole program into several smaller sub-programs. Since each sub-program is smaller, it is intuitively easier for us to figure how to test.

Past research work based on the above ideas (flow of data and divide-and-conquer) achieved the following two sub-approaches:

Define/Use Testing (or DU testing)

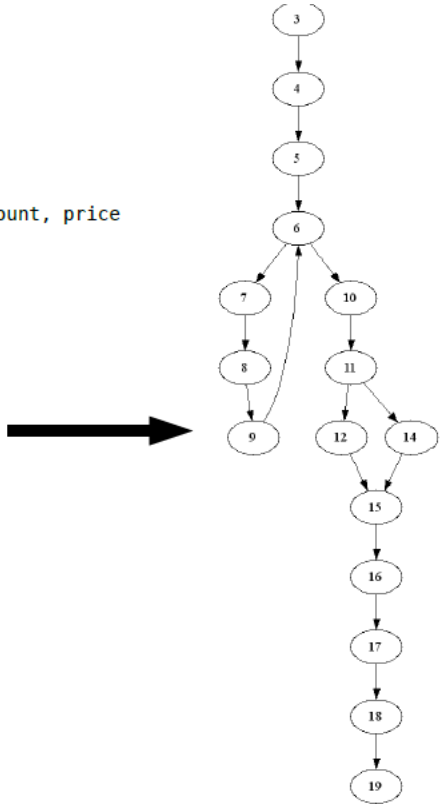
Slice-Based Testing

Define/Use Testing (or DU testing)

We start our discussion with Define/Use Testing. Again, we ask a logical question first. How do you follow or monitor the flow of data? We prefer to build some intuition by observation before we touch the formal definitions and theories.

https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwiE9uTf_t_KAhUCWCYKHYS_BCMQFggtMAE&url=http%3A%2F%2Fwww.cs.swan.ac.uk%2F~csmarkus%2FCS339%2Fpresentations%2F20061202_New_Data_Flow_Testing.pdf&usg=AFQjCNGwmJe1sBfIgO0B3aU14q5-f22-4A gives the following example:

```
1  program Example()

2  var staffDiscount, totalPrice, finalPrice, discount, price

3  staffDiscount = 0.1
4  totalPrice = 0

5  input(price)
6  while(price != -1) do
7   totalPrice = totalPrice + price
8   input(price)
9  od

10 print("Total price: " + totalPrice)

11 if(totalPrice > 15.00) then
12  discount = (staffDiscount * totalPrice) + 0.50
13 else
14  discount = staffDiscount * totalPrice
15 fi

16 print("Discount: " + discount)
17 finalPrice = totalPrice - discount
18 print("Final price: " + finalPrice)

19 endprogram
```



If you pay attention to the data (variable) "price" at node 5, you find that the variable *price* is defined there. Node 6 shows a use of this variable (price != -1). Node 7 shows another use of this variable (totalPrice = totalPrice + price). At node 8, the program defines the variable *price* again (input (price)). You may conclude that if you focus on the variable *price*, you end up with:

2 defined nodes: node 5 and node 8 (or DEF (price, 5), DEF (price,8))

2 use nodes: node 6 and node 7 (or USE (price, 6), USE (price, 7)

For those four nodes (2 defined, 2 use), you may form four pairs (called DU pairs):

Pair (DEF (price, 5), USE (price, 6))

Pair (DEF (price, 5), USE (price, 7))

Pair (DEF (price, 8), USE (price, 6))

Pair (DEF (price, 8), USE (price, 7))

A DU pair reveals the beginning and the ending of the data flow associated with a given variable. What happens to the data flow in between the beginning and the end? To answer that question,

we need to find path(s) that links the beginning node to the ending node of a given DU pair. This kind of path is call DU path.

Now, given a variable, can we find all its DU pairs? Yes, we just did it for the given variable price in our example. Can we find all the DU paths for all the DU pairs? Yes, for sure. You just find all the DU paths associated with a DU pair. Then, repeat the process for all other pairs. For our example, all the DU paths for a given variable price are:

<5, 6>

<5, 6, 7>

<8, 9, 6>

<8, 9, 6, 7>

What can you do with the DU paths?

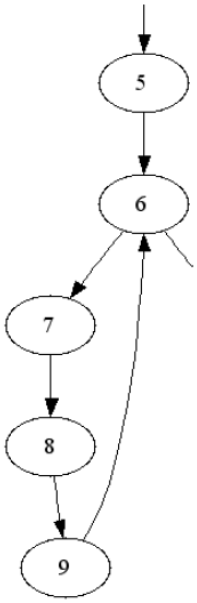
DU paths of a given variable show you all the possible paths of that variable flowing through the program. As a tester, you may want to design test cases to traverse all the DU paths. If all things check out, you do have some confidence to claim that your program handles the given variable correctly. What happens to other variables (say, the variable totalPrice in our example)? Easy, you simply repeat the same process to find all the DU paths associated with the next variable.

- For price variable in example

2 define nodes
DEF(price, 5)
DEF(price, 8)

2 use nodes
USE(price, 6)
USE(price, 7)

Du-paths:
<5, 6>
<5, 6, 7>
<8, 9, 6>
<8, 9, 6, 7>
All are definition-clear.



After we have established our intuition, it is time to make things more rigorous.

Set DEF(n) contains variables that are defined at node n (i.e., they are written)

Set USE(n) contains variables that are used at node n (i.e., they are written)

C-use (Computation use)
Example: x = 2*y; /* y has been used to compute a value of x. */

P-use (Predicate use)
Example: if (y > 100) { …} /* y has been used in a condition. */

Global c-use: A c-use of a variable x in node i is said to be a global c-use if x has been defined before in a node other than node i.

Global definition: A node i has a global definition of variable x if node i has a definition of x and there is a def-clear path w.r.t. x from node i to some node containing a global c-use, or edge containing a p-use of variable x.

Definition clear path: A path (i – n1 – … nm – j), $m \geq 0$, is called a definition clear path (def-clear path) with respect to variable x

A **def-use pair** (DU) for variable x is a pair of nodes (n1,n2) such that x is in DEF(n1). The definition of x at n1 reaches n2 and x is in USE(n2)

All-DU-paths: For each DU pair (n1,n2) for x, exercise **all possible paths** n1, n2 that are clear of a definition of x

All-uses: for each DU pair (n1,n2) for x, exercise at least one path n1 n2 that is clear of definitions of x

All-definitions: for each definition, cover at least one DU pair for that definition

All-c-uses

For each variable x and each node i, such that x has a global definition in node i, select complete paths which include def-clear paths from node i to all nodes j such that there is a global c-use of x in j.

All-p-uses

For each variable x and each node i, such that x has a global definition in node i, select complete paths which include def-clear paths from node i to all edges (j, k) such that there is a p-use of x on (j, k).

All-p-uses/some-c-uses

This criterion is identical to the all-p-uses criterion except when a variable x has no p-use. If x has no p-use, then this criterion reduces to the some-c-uses criterion.

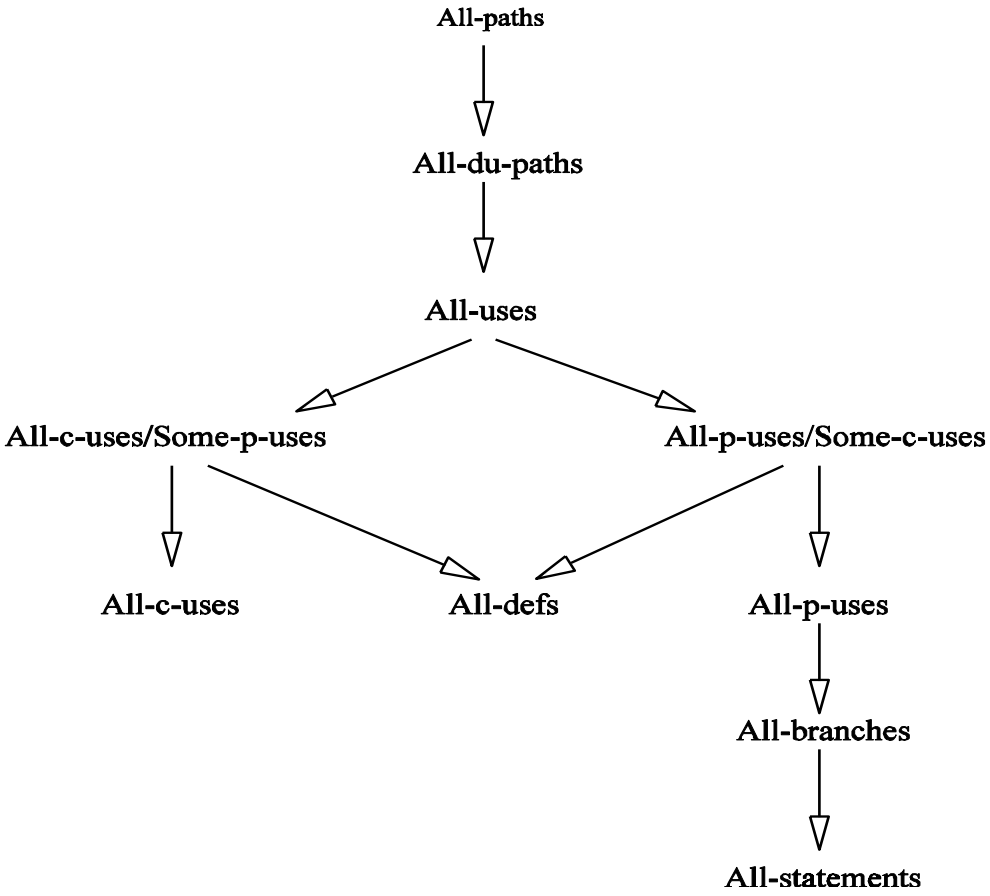
> Some-c-uses: For each variable x and each node i, such that x has a global definition in node i, select complete paths which include def-clear paths from node i to some nodes j such that there is a global c-use of x in j.

All-c-uses/some-p-uses

This criterion is identical to the all-c-uses criterion except when a variable x has no c-use. If x has no global c-use, then this criterion reduces to the some-p-uses criterion.

> Some-p-uses: For each variable x and each node i, such that x has a global definition in node i, select complete paths which include def-clear paths from node i to some edges (j, k) such that there is a p-use of x on (j, k).

"Software Testing and Quality Assurance: Theory and Practice Kshirasagar Naik, Priyadarshi Tripathy" give the following graph that shows the relationship of different criteria.



Clearly, all-definitions is subsumed by all-uses which is subsumed by all-DU-paths

Slice-Based Testing

Another way of tracking data that flows through the program is called slice-based testing. The first logical question is - what is the slice-based testing? Let's begin with an observation. [A Survey of Program Slicing Techniques by Frank Tip] gives the following program example:

```
(1)   read(n);
(2)   i := 1;
(3)   sum := 0;
(4)   product := 1;
(5)   while i <= n do
      begin
(6)     sum := sum + i;
(7)     product := product * i;
(8)     i := i + 1
      end;
(9)   write(sum);
(10)  write(product)
```

(a)

```
read(n);
i := 1;

product := 1;
while i <= n do
begin

  product := product * i;
  i := i + 1
end;

write(product)
```

(b)

(a) is just a typical program under test. (b) is a subset (or called slice) of program (a) if we only want to analyze the flow of the variable *product* all the way from the beginning to line 10. Our first observation is that this slice is obtained by "slicing away" all computations not relevant to what happens to the variable *product* at line 10. The second observation you may have is that program (b) is a reduced, executable program obtained from program (a). Why do we want to do this? The original proposer [Weiser 1984] who advocated the concept of program slicing claims that it is much easier to debug a program if you view it as a collection of multiple slices. Each slice is smaller and simpler than the original program. It is not that hard to accept this claim. In a way, it is just another form of "divide-and-conquer."

After a brief explanation of what and why, the next logical question is how. You may say, by inspection, we have no problem to get program (b) from program (a) given a criterion of (variable *product* and line 10). For simplicity, we want to write this criterion as S (V, n)① where V represents a specific variable we are interested in, or a set of variables if we really want to handle multiple variables simultaneously②, n stands for the program line location③ where we start our investigation.

① We use S to represent a criterion since a given criterion will produce a slice (or a collection of slices if we want to handle more than one variable at a time).

② Although it is possible to handle multiple variables simultaneously, in a typical scenario, we really prefer to handle one variable at a time.

③ Once you determine the line location, you start examine all the statements starting from that line location *backward* to the beginning of the program. This is the reason why in some literature people call it backward④ slicing.

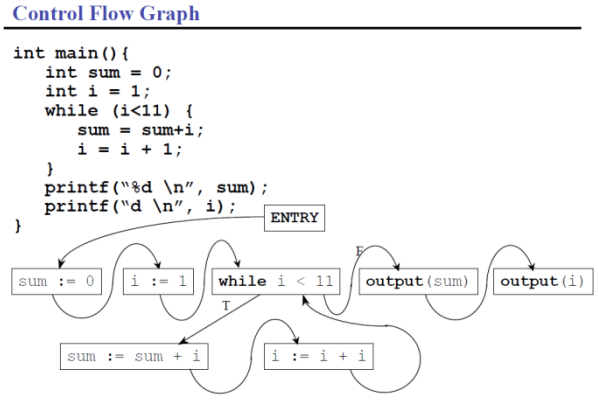
④ Since we have backward slicing, do we have forward slicing? The answer is yes. Another line of research actually focused on the program statements that are affected by the variable V only after the line location n. This means you examine the program forward starting line location n all the way to the end of the program. It is called forward slicing.

Notice that in the above discussion we didn't consider any pre-defined input values. To highlight this fact, we may call our slicing as static slicing. Can we add some pre-defined input values in our criterion? The answer is yes and one more research line actually does that. In this case, we call our slicing as dynamic slicing.

A method that is based solely on inspection is not scalable. Are we going to inspect a program with more than thousand lines? Our question on "how" needs to focus on a systematic way of turning a given program into some desirable slices. Once we found a systematic way, for sure, we can produce an automatic software tool to do the chores.

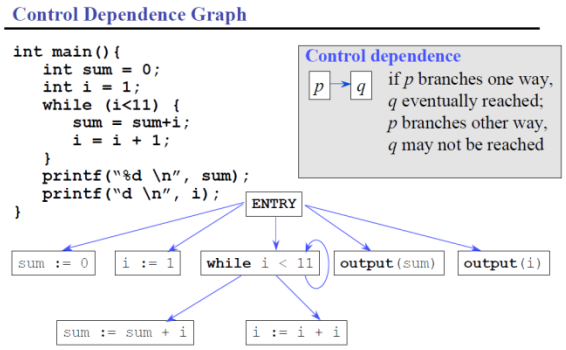
A slicing example explained in [www.cis.upenn.edu/~cis570/.../lecture14.pdf] as follows provide us a very good starting point toward this goal.

The idea starts from forming a Program Dependence Graph (PDG) in which nodes are statements and edges represent either control dependence, or data dependence. To illustrate this new graph, we use a single function① and find its corresponding Control Flow Graph (or called Program Graph).



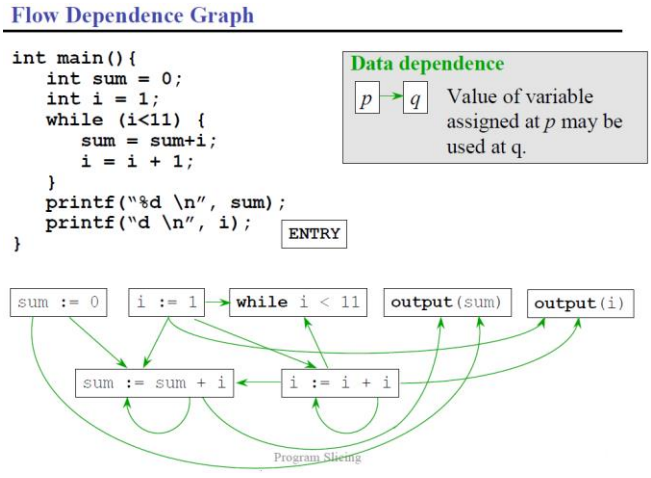
① Some literatures use a fancy name for this single function scenario as *Intraprocedural Slicing*. Due to the fact that this scenario deals with things happened with one function/procedure, therefore, the fancy term "Intra."

From the Control Flow Graph one can come up with the Control Dependence Graph as follows:



We use the following rules to form the Control Dependence Graph. Notice that here we call a node as a vertex. A Program Dependence Graph of a program P contains a Control Dependence edge from vertex v1 to vertex if and only if one of the following holds:

> *Vertex v1 is the entry vertex and v2 represents a component of program P that is not nested within a loop or conditional;*
>
> *Vertex v1 represents a control dependence and v2 represents a component of P immediately nested within a loop or a conditional statement whose predicate is represented by v1. If v1 is a test predicate of a loop structure the edge is labeled True.*

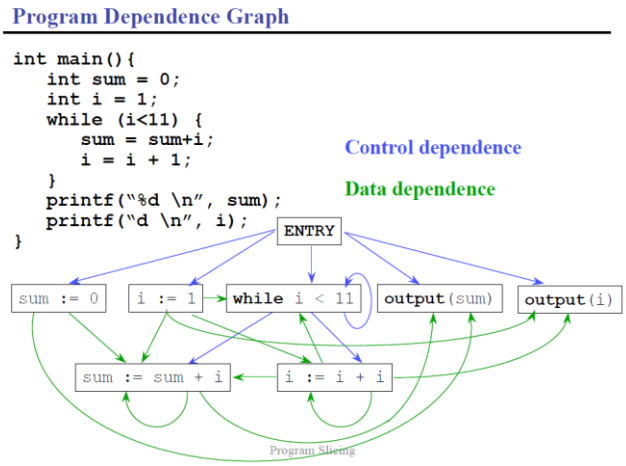


We also obtain the Flow Dependence Graph (also called Data Dependence Graph) by the following mechanism:

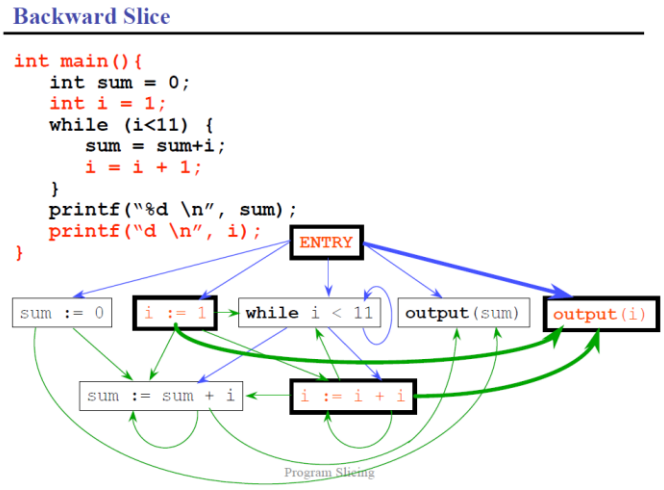
A Program Dependence Graph of a program P contains a Data Dependence edge from vertex v1 to vertex v2 if and only if all of the following holds:

Vertex v1 is a vertex that defines variable x
Vertex v2 is a vertex that uses variable x (or x's aliases)
Control can reach v2 after v1, via an execution path along which there is no other intervening definition of x. That is there is a path in the program's CFG by which definition of x at v1 reaches the use of x at v2.

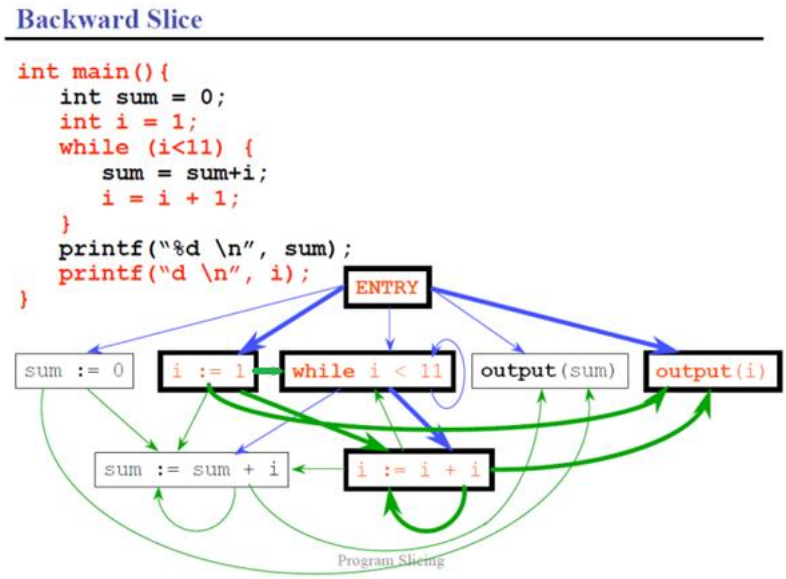
It is possible to plot both Control Dependence Graph and Data Dependence Graph together as shown below:



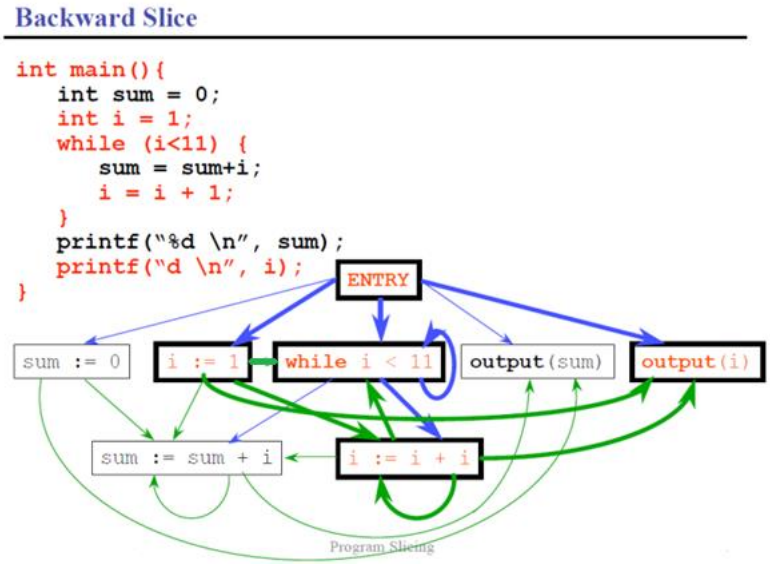
Once we have obtained both graphs, we may begin our backward slicing. For our example, our slicing criterion is the *variable i* and at the line *printf("d \n", i)*. Once this criterion is determined, we work backward starting from the line location *printf("d\d", i)* or for convenience, we label this line as *Output (i)*.



As we work backward in the first step, we notice that there are several vertexes (nodes) that reach the vertex/node Output(i) with just one incoming edge either as control dependence or data dependence edge. In the above graph, those edges are highlighted. As the second step, we now shift our attention to the new vertexes connected by the highlighted edges (and they are Entry, i :=1, i :=i+1). Again, we highlight the incoming edges to those three vertexes/nodes. This resulting graph is shown below.



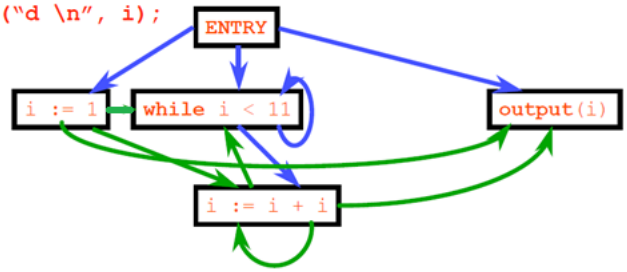
As the third step in our example, the new vertex/node in this step is *while i < 11*. Note that after highlighting its incoming edges, we reach no new vertexes/nodes.



At this point, we stop and take out all untouched vertexes and edges. We end up with the following graph and therefore, the slice of the program based on the criterion (*variable i, at line printf("d \n", i)*) as shown below.

Slice Extraction

```
int main(){

   int i = 1;
   while (i<11) {

      i = i + 1;
   }

   printf("d \n", i);
}
```



The above analysis is restricted within one single function/procedure and is called Intraprocedural Slicing. If more than one procedure is involved, we need to perform Interprocedural Slicing.

The same source gives the following example:

First, since more than one procedure is involved, we prefer to change the name of Program Dependence Graph to System Dependence Graph (SDG) that represents dependences within entire system (i.e., more than one procedure). The way to handle this new scenario is best explained by an example.

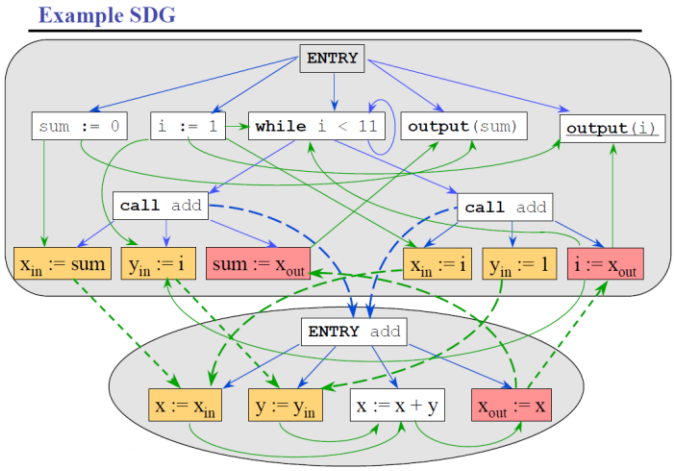
Interprocedural Slice

```
int main(){                    int add(int x, y){
   int sum = 0;                   return x + y;
   int i = 1;                  }
   while (i<11) {
      add(sum,i);
      add(i, 1);
   }
   printf("%d \n", sum);
   printf("d \n", i);
}
```

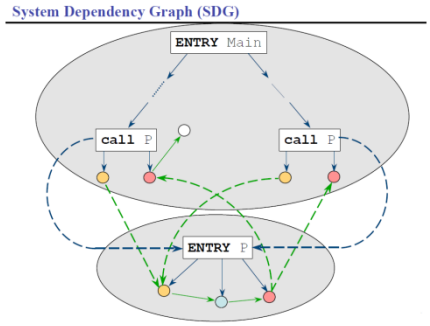
Should we include **add(sum,i)**?

In this example, we have two procedures. One is the main procedure and the other is the procedure add (sum, i). Also, we still use the same slicing criterion of (*variable i, printf("d \n", i)*).

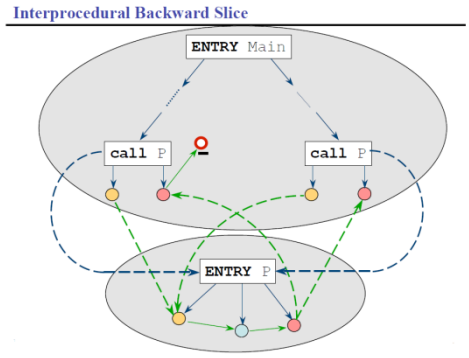
If you practice the mechanism of finding the control dependence and data dependence edges (not shown here), you will end up with the following graph.

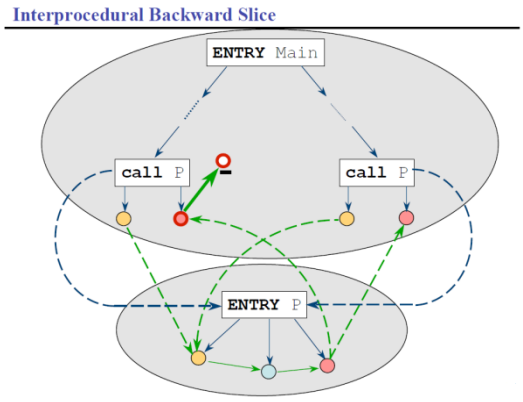


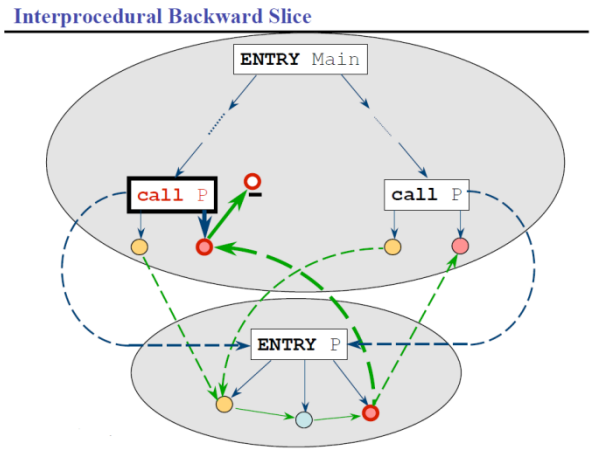
Its "zoom-out" or "abstracted" version is shown below.

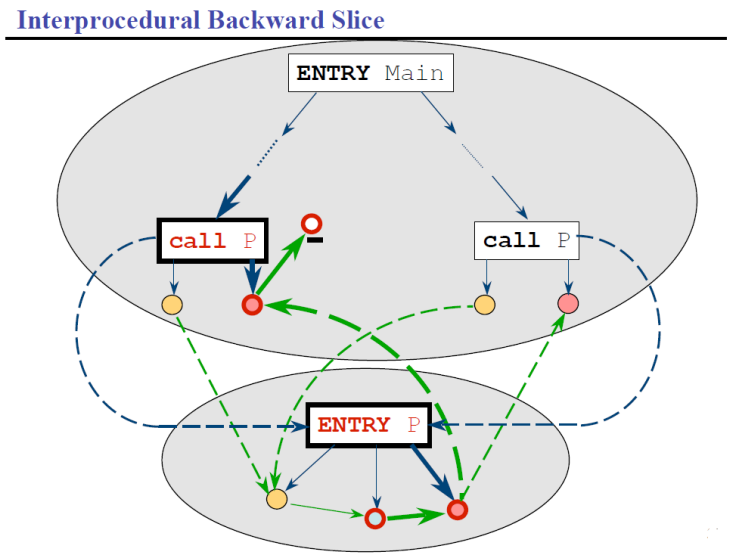


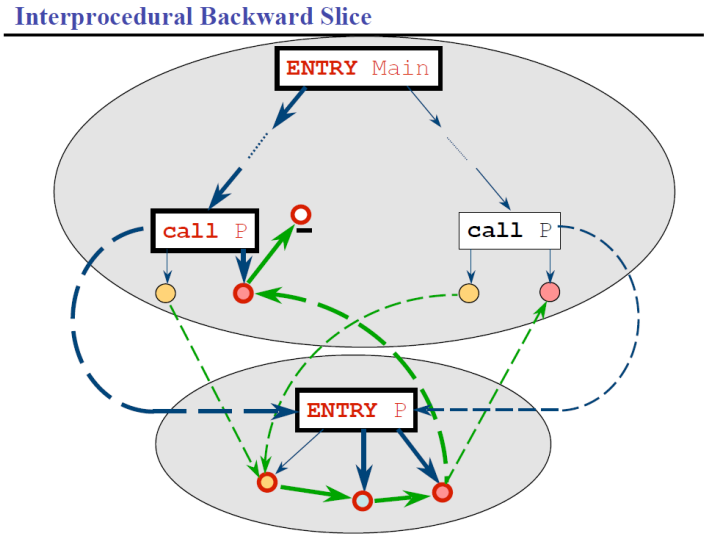
Again, as the first step, we start the targeted line *print ("d \n", i)* also called Output (i) in our graph and we work our way backward.



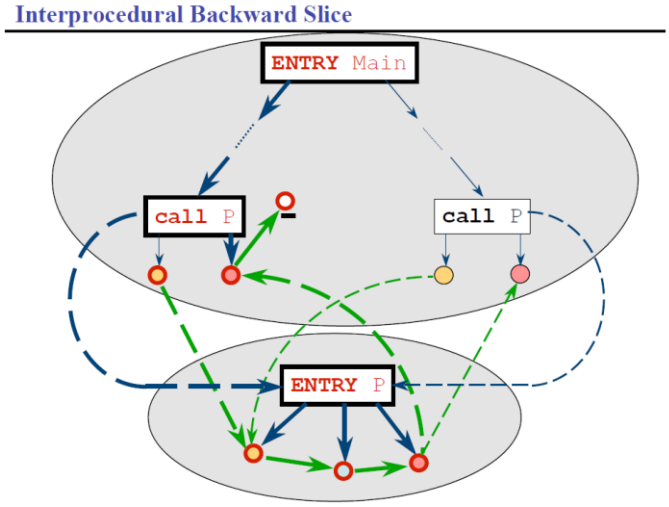

Interprocedural Backward Slice
ENTRY Main
call P
call P
ENTRY P

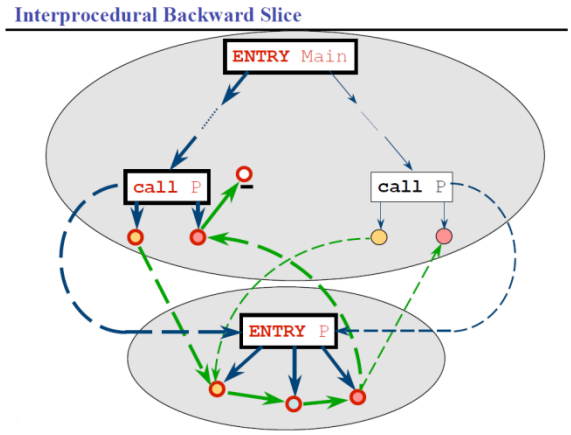

Interprocedural Backward Slice
ENTRY Main
call P
call P
ENTRY P

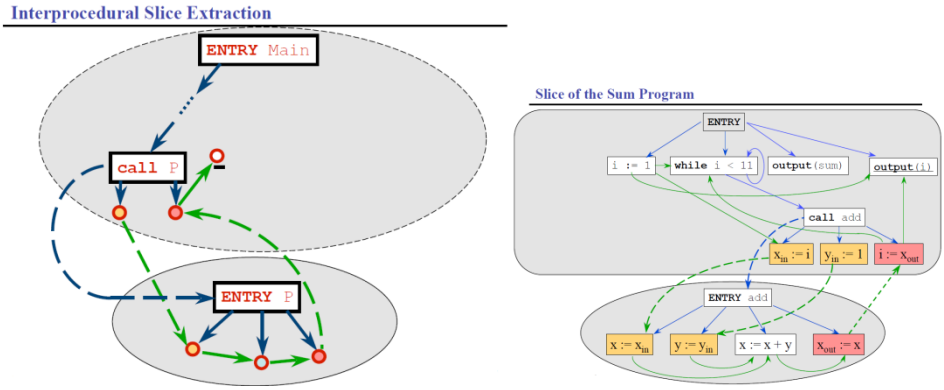

Interprocedural Backward Slice
ENTRY Main
call P
call P
ENTRY P



It is important to note the extra rule we create here. The ENTRY P actually can be tracked back to two "call P" boxes. Nevertheless, we don't want to highlight both of the dashed incoming edges that link two "call P" boxes to the "ENTRY P" box. The reason is that only one of the "call P" boxes actually calls the "ENTRY P" box. Which "call P" box is the one that actually calls the "ENTRY P" box? The answer is the highlighted one. Therefore, we should only highlight the edge from an already highlighted "call P" box to the "ENTRY P" box.







From the final graph, we now have our slice as follows:

```
int main() {
   int i=1;
while (i<11) {
add (I, 1);
printf("%d \n", i);
}
int add(int x, y) {
Return x+y;
}
```

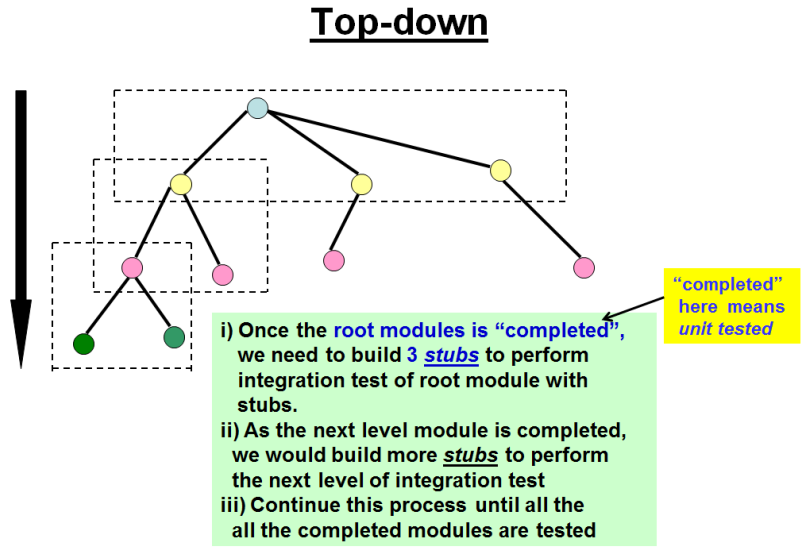
Chapter 8 Integration Testing

Once we are done with unit testing of one or more units, we may want to start the integration testing. To make testing more efficient and to detect problems as early as possible, there is no need to wait for the finish of all units. Let's consider a scenario. As a tester, you know that there are several units of software are under unit testing now and in the next two weeks, you will have some units on hand for the integration testing. Most testers at this point probably are eager to study the interface requirements of those units and try to figure out test cases for them. Well, it turns out that this is not the top priority. The important task at this moment actually is to figure out (or estimate) the arrival time of those units and plan your integration testing schedule (or "itinerary" if you don't mind to use that word). In our scenario, the root (top) module that interacts with three second-level modules arrives first. The second-level modules, four of them, arrive one or two days later. Another two days pass and the third-level modules arrive. Once you realize that the arrival timing of those units actually is an important factor that may affect your testing time and budge, you may ask a logical question on the *what* question (i.e., what should I do?).

Top-Down Structural Decomposition-Based Integration Testing

If the units arrive in a top-down fashion, then, sure enough, we will find a way to deal with it and we call our way as Top-Down Structural Decomposition-Based Integration Testing. Yes, it is an awkward long name. But, in an effort to avoid subtle confusion, let's just at least use this long name once.

[Xu] provides a good example that explains the testing mechanism as below:

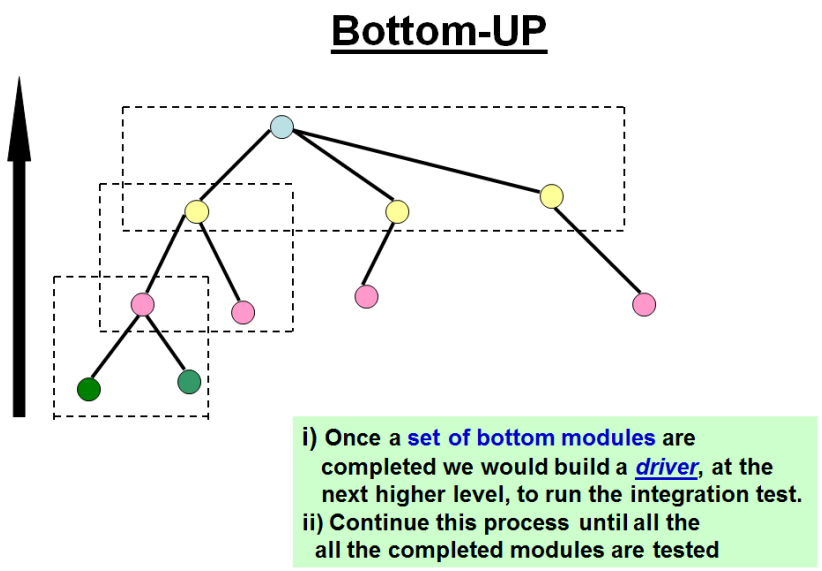


Once the root unit (or called module, if you like) arrives, you may want to construct three test stubs① so that you can test the root unit immediately.

> ①Test stubs are programs that simulate the behaviors of software components (or modules) that a module undergoing tests depends on.

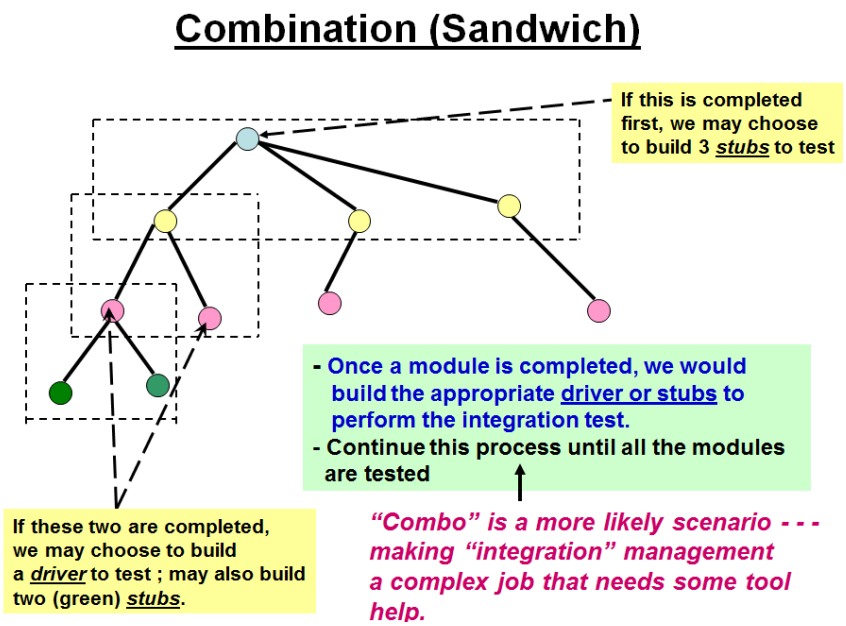
It is important to understand that the edge in the above Structural Graph may or MAY NOT represent calling relationship. We use an edge to link one unit to another unit may simply because the second unit logically falls under the hierarchy of the first unit (for example, the second unit resides in a sub-directory of the first unit). The first unit may not have any direct calls to the second unit.

Bottom-Up Structural Decomposition-Based Integration Testing

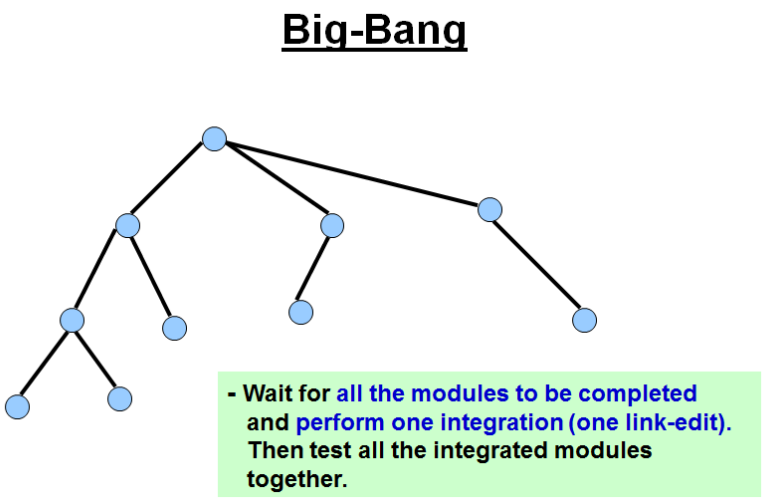


Obviously, the same idea can apply to the bottom-up situation.

Combination/Sandwicch Structural Decomposition-Based Integration Testing

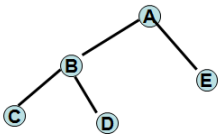


Big-Bang Structural Decomposition-Based Integration Testing



Some Metrics for Integration Test based on Structural Decomposition

- For Top-down approach with *n nodes*, there is a potential need to construct as much as *(n-1) stubs*.
- For Bottom-up approach with *n nodes and v leaves*, there is a potential need to construct as much as *(n - v) drivers*.
- For both cases there may be as much as *(n – v + edges) number of test sessions (e.g. cases or scenarios)*



- There are 5 nodes and 3 leaves and 4 edges:

1. n- 1 = 5 -1 = *4 stubs*
2. n - v = 5 - 3 = *2 drivers*
3. n - v+ edges = 5 – 3 + 4 = *6 test sessions*

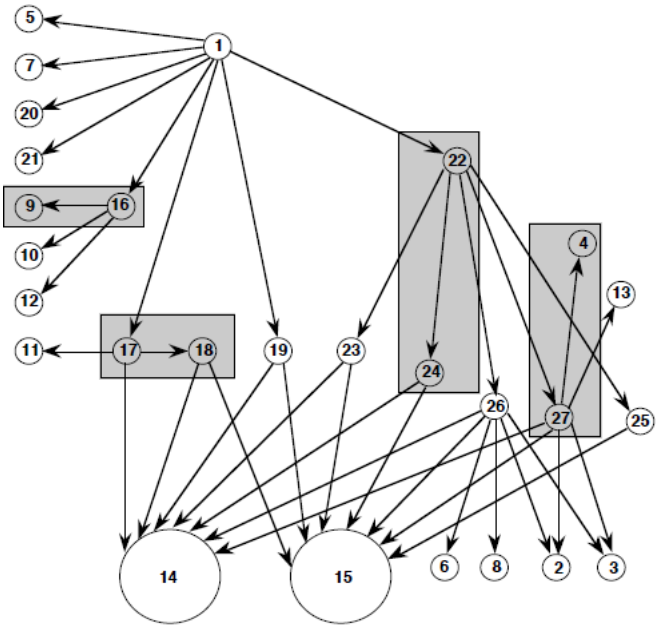
Number of test sessions = 6:

(Top-down)	(Bottom-up)
1. A is complete: test with 2 stubs	E is complete: test with driver A
2. B is complete: test with 2 stubs	D is complete: test with driver B
3. B is complete: test with A	C is complete: test with driver B
4. C is complete: test with B	B is complete: test with C and D
5. D is complete: test with B	B is complete: test with driver A
6. E is complete: test all modules	A is complete: test all modules

Call Graph

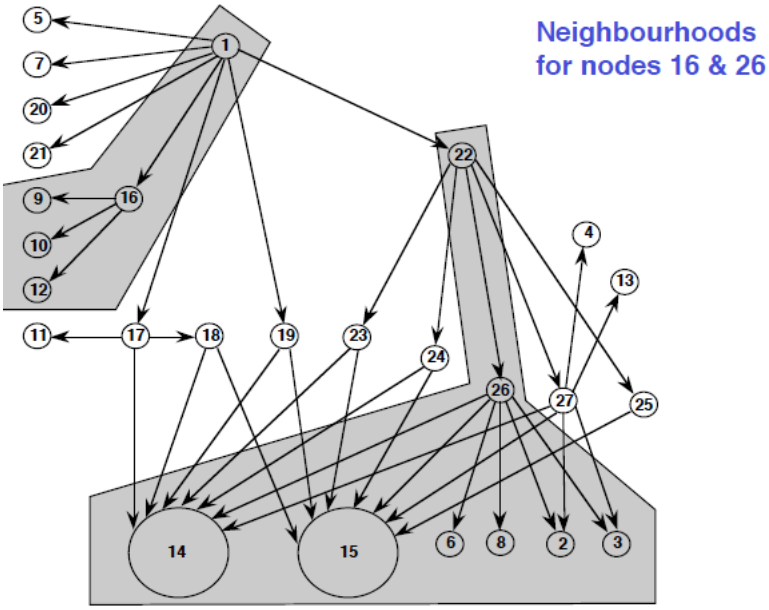
A Call Graph is a directed graph in which vertices are program units (or methods) and a directed edge joins calling vertex to the called vertex. Note that a call graph is different from a structural graph. An edge in a call graph indeed represents a calling relationship that links one unit to another.

Pair-Wise Call Graph-Based Integration



The above call graph shows calling relationship among 15 units of program. You only perform the integration testing on a pair of units which has a calling edge from one to another, for example, an integration testing on the pair (unit 9 and unit 16). The motivation is that since unit 9 is available, you don't need to come up with a test driver stub during the test of unit 16.

Neighborhood Call Graph-Based Integration Testing



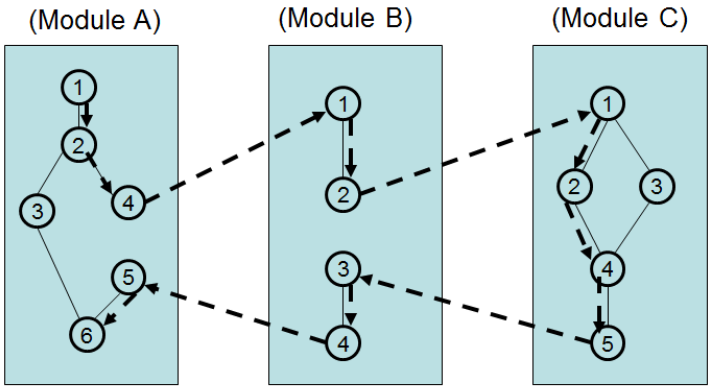
Another variant of call graph testing is the use of neighborhood concept. The above call graph highlights a neighborhood of unit/node 16 and a neighborhood of unit/node 26. A neighborhood of a given unit/node is the collection of all the units/nodes one edge away from it and itself. You may want to conduct one integration testing on each neighborhood at a time.

Pros and Cons of Call-Graph Integration

TBD

Conducting integration testing by following paths

A call graph shows us a static calling relationship among given units. For example, we use an edge in a call graph to say unit 1 has a source code statement that calls unit 2. At run time, will unit 1 actually call unit 2? Well, it may not, if the call is under a decision statement (e.g., if false, it will not call). This observation opens up the idea of Path-Based Integration Testing. Let's take a look at a scenario.

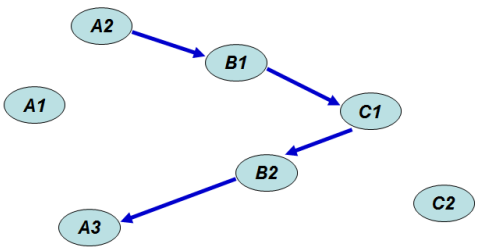


Module Execution Paths: ***A1***= <1,2,3,6>; ***A2***=<1,2,4> ; ***A3*** = <5,6>
B1= <1,2> ; ***B2*** = <3,4>
C1= <1,2,4,5> ; ***C2***= <1,3,4,5>

The above graph shows three program units or called modules here (A, B and C). The nodes in each unit are program statements. In this example, we assume that in one scenario, the execution starts in Module A (from Node 1 to Node 4), then, calls Module B(from Node 1 to Node 2), then, calls Module C (from Node 1 to Node 5). The return path starts from Module C (Node 5), passing Module B, Node 4, finally ends in Module A Node 6. Why should we pay attention to this long possible execution path? Well, for a tester who conducts an integration test on Modules A , B and C, this path may represent a major feature of the software and for sure, you may want your test case that "walk/pass" through this path. Is this the only path scenario? Certainly not. For example, the execution may start in Module A through Node 1, 2, 3 and 6, which never calls Module B. We name this kind of path that links modules along their messaging (or invocation/calling) mechanism as MM-Path (Module-Message-Path) [Jorgensen].

If we also label all other possible ordinary paths① in each module, we may end up with the following paths:
A1= a path that goes through Node 1, 2, 3, 6
A2= a path that goes through Node 1, 2, 4
A3= a path that goes through Node 5, 6
B1= a path that goes through Node 1,2
B2= a path that goes through Node 3, 4
C1= a path that goes through Node 1, 2, 4, 5
C2= a path that goes through Node 1, 3, 4, 5

Using the above possible path notations, we may redraw our original scenario as:



We observe that the MM-Path we discussed before, composes of paths of A2, B1, C1, B2 and A3. Notice that paths A1 and C2 are not touched. In some sense, the MM-Path we identified can be viewed as a *slice* of all possible paths in the combined modules. Now, you probably can see the key idea (or claim) – if we can find and test all possible MM-Paths, we will be in good shape in terms of integration testing. We prefer to call this approach as **MM-Path Path-Based Integration Testing** – an awkward and long name, but, at least, we should use it one time.

> ① You may wonder, for instance, why we don't have a path that goes through Node 1, 2, 4, 5, 6 in Module A? One "on the face" answer is that well, there is no edge that links Node 4 to 5 in our graph. Nevertheless, we soon realize that Node 5 is the next statement of Node 4. In a typical program graph, we should draw an edge from Node 4 to Node 5. Well, in our discussion, we know that Node 4 is a calling statement (it calls Node 1 in Module B). Due to this detour, we really can't say that we have a direct path from Node 4 to Node 5 in Module A.

Pros and cons of mm-path path-based integration

TBD

MM-path integration testing compared to other methods

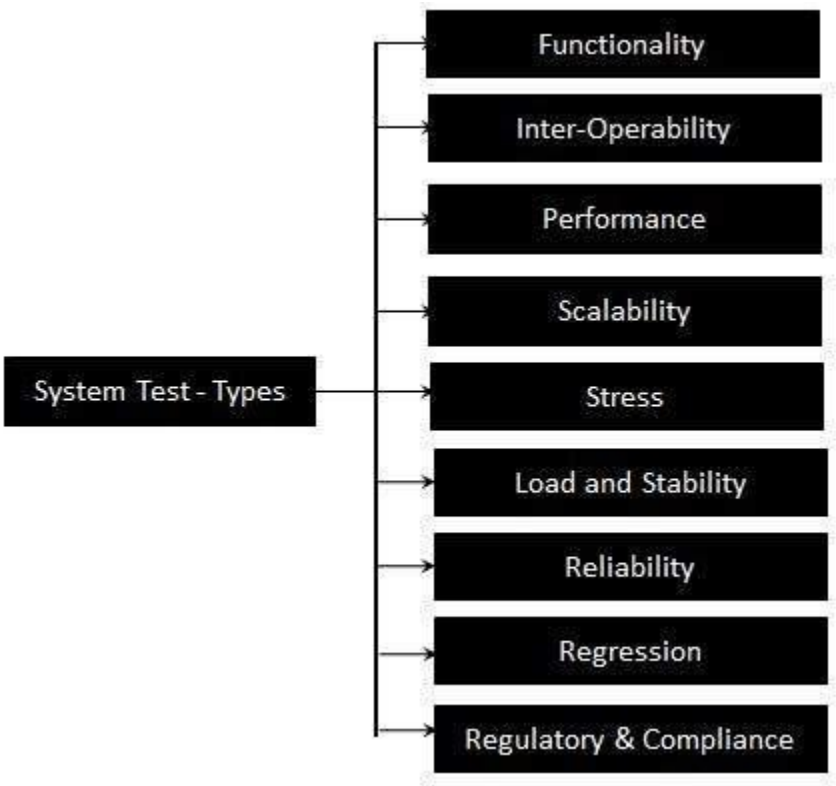
Strategy	Ability to test interfaces	Ability to test co-functionality	Fault isolation resolution
Functional decomposition	Acceptable, can be deceptive	Limited to pairs of units	Good to faulty unit
Call-graph	Acceptable	Limited to pairs of units	Good to faulty unit
MM-path	Excellent	Complete	Excellent to unit path level

Chapter 9 System Testing

There are many different versions of definition on the term system testing. We don't think a precise definition is important nor necessary. One source [http://www.tutorialspoint.com/software_testing_dictionary/system_testing.htm] gives a good enough explanation on this term:

> *System Testing is a black box testing technique performed to evaluate the complete system the system's compliance against specified requirements. In System testing, the functionalities of the system are tested from an end-to-end perspective. System Testing is usually carried out by a team that is independent of the development team in order to measure the quality of the system unbiased. It includes both functional and Non-Functional testing.*

If you want to zoom in further, you may find that there are many categories (or types) under the system testing. The same source gives the following:



[https://en.wikipedia.org/wiki/System_testing] gives another possible types of system testing:

Types of tests to include in system testing

The following examples are different types of testing that should be considered during System testing:

- Graphical user interface testing
- Usability testing
- Software performance testing
- Compatibility testing
- Exception handling
- Load testing
- Volume testing
- Stress testing
- Security testing
- Scalability testing
- Sanity testing
- Smoke testing
- Exploratory testing
- Ad hoc testing
- Regression testing
- Installation testing
- Maintenance testing[clarification needed]
- Recovery testing and failover testing.
- Accessibility testing, including compliance with:
  - Americans with Disabilities Act of 1990
  - Section 508 Amendment to the Rehabilitation Act of 1973
  - Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C)

Does the above two sources cover all the possible types of system testing? Hardly! Do we, as a practitioner, need to know all possible types of system testing? Probably not! In the Internet age, "stocking" knowledge, in some cases, is same as stocking inventory. Same as inventory, some knowledge may become obsolete before you can recoup your investment. Similar to the idea of Just-In-Time-Inventory, Just-In-Time-Learning may deserve some thoughts too.

In this chapter, rather than trying to cover all types, we would like to focus on some common system testing types.

Graphical user interface testing
[https://en.wikipedia.org/wiki/Graphical_user_interface_testing] gives the following:

In software engineering, graphical user interface testing is the process of testing a product's graphical user interface to ensure it meets its specifications. This is normally done through the use of a variety of test cases.

Usability testing

https://en.wikipedia.org/wiki/Usability_testing

Usability testing is a technique used in user-centered interaction design to evaluate a product by testing it on users. This can be seen as an irreplaceable usability practice, since it gives direct input on how real users use the system. This is in contrast with usability inspection methods where experts use different methods to evaluate a user interface without involving users.

Usability testing focuses on measuring a human-made product's capacity to meet its intended purpose. Examples of products that commonly benefit from usability testing are foods, consumer products, web sites or web applications, computer interfaces, documents, and devices. Usability testing measures the usability, or ease of use, of a specific object or set of objects, whereas general human-computer interaction studies attempt to formulate universal principles.

Simply gathering opinions on an object or document is market research or qualitative research rather than usability testing. Usability testing usually involves systematic observation under controlled conditions to determine how well people can use the product. However, often both qualitative and usability testing are used in combination, to better understand users' motivations/perceptions, in addition to their actions.

Rather than showing users a rough draft and asking, "Do you understand this?", usability testing involves watching people trying to use something for its intended purpose. For example, when testing instructions for assembling a toy, the test subjects should be given the instructions and a box of parts and, rather than being asked to comment on the parts and materials, they are asked to put the toy together. Instruction phrasing, illustration quality, and the toy's design all affect the assembly process.

Stress testing
https://en.wikipedia.org/wiki/Stress_testing#Software

In software testing, a system stress test refers to tests that put a greater emphasis on robustness, availability, and error handling under a heavy load, rather than on what would be considered correct behavior under normal circumstances. In particular, the goals of such tests may be to ensure the software does not crash in conditions of insufficient computational resources (such as memory or disk space), unusually high concurrency, or denial of service attacks.

Examples:

A web server may be stress tested using scripts, bots, and various denial of service tools to observe the performance of a web site during peak loads. These attacks generally are under an hour long, or until a limit in the amount of data that the web server can tolerate is found.

Stress testing may be contrasted with load testing:

Load testing examines the entire environment and database, while measuring the response time, whereas stress testing focuses on identified transactions, pushing to a level so as to break transactions or systems.

During stress testing, if transactions are selectively stressed, the database may not experience much load, but the transactions are heavily stressed. On the other hand, during load testing the database experiences a heavy load, while some transactions may not be stressed.

System stress testing, also known as stress testing, is loading the concurrent users over and beyond the level that the system can handle, so it breaks at the weakest link within the entire system.

Scalability testing

https://en.wikipedia.org/wiki/Scalability_testing
Scalability Testing, is the testing of a software application to measure its capability to scale up or scale out in terms of any of its non-functional capability.

Performance, scalability and reliability testing are usually grouped together by software quality analysts.

The main goals of scalability testing are to determine the user limit for the web application and ensure end user experience, under a high load, is not compromised. For example, can a web page be accessed in a timely fashion with limited delay in response. Another goal is to check the server can cope i.e. Will the server crash if it is under a heavy load?

Dependent on the application that is being tested, different parameters are tested. If a webpage is being tested, the highest possible number of simultaneous users would be tested. [2] Also dependent on the application being tested is the attributes that are tested - these can include CPU usage, network usage or user experience.

Successful testing will project most of the issues which could be related to the network, database or hardware/software.

Security testing
https://en.wikipedia.org/wiki/Security_testing
Security testing is a process intended to reveal flaws in the security mechanisms of an information system that protect data and maintain functionality as intended. Due to the logical limitations of security testing, passing security testing is not an indication that no flaws exist or that the system adequately satisfies the security requirements.

Typical security requirements may include specific elements of confidentiality, integrity, authentication, availability, authorization and non-repudiation. Actual security requirements tested depend on the security requirements implemented by the system. Security testing as a term has a number of different meanings and can be completed in a number of different ways. As such a Security Taxonomy helps us to understand these different approaches and meanings by providing a base level to work from.

Confidentiality

A security measure which protects against the disclosure of information to parties other than the intended recipient is by no means the only way of ensuring the security.

Integrity

Integrity of information refers to protecting information from being modified by unauthorized parties

A measure intended to allow the receiver to determine that the information provided by a system is correct.

Integrity schemes often use some of the same underlying technologies as confidentiality schemes, but they usually involve adding information to a communication, to form the basis of an algorithmic check, rather than the encoding all of the communication.

To check if the correct information is transferred from one application to other

Authentication

This might involve confirming the identity of a person, tracing the origins of an artifact, ensuring that a product is what its packaging and labeling claims to be, or assuring that a computer program is a trusted one.

Authorization

The process of determining that a requester is allowed to receive a service or perform an operation.

Access control is an example of authorization.

Availability

Assuring information and communications services will be ready for use when expected.

Information must be kept available to authorized persons when they need it.

Non-repudiation

In reference to digital security, nonrepudiation means to ensure that a transferred message has been sent and received by the parties claiming to have sent and received the message. Nonrepudiation is a way to guarantee that the sender of a message cannot later deny having sent the message and that the recipient cannot deny having received the message.

Security Testing Taxonomy

Common terms used for the delivery of security testing:

Discovery - The purpose of this stage is to identify systems within scope and the services in use. It is not intended to discover vulnerabilities, but version detection may highlight deprecated versions of software / firmware and thus indicate potential vulnerabilities.

Vulnerability Scan - Following the discovery stage this looks for known security issues by using automated tools to match conditions with known vulnerabilities. The reported risk level is set automatically by the tool with no manual verification or interpretation by the test vendor. This can be supplemented with credential based scanning that looks to remove some common false positives by using supplied credentials to authenticate with a service (such as local windows accounts).

Vulnerability Assessment - This uses discovery and vulnerability scanning to identify security vulnerabilities and places the findings into the context of the environment under test. An example would be removing common false positives from the report and deciding risk levels that should be applied to each report finding to improve business understanding and context.

Security Assessment - Builds upon Vulnerability Assessment by adding manual verification to confirm exposure, but does not include the exploitation of vulnerabilities to gain further access. Verification could be in the form of authorised access to a system to confirm system settings and involve examining logs, system responses, error messages, codes, etc. A Security Assessment is looking to gain a broad coverage of the systems under test but not the depth of exposure that a specific vulnerability could lead to.

Penetration Test - Penetration test simulates an attack by a malicious party. Building on the previous stages and involves exploitation of found vulnerabilities to gain further access. Using this approach will result in an understanding of the ability of an attacker to gain access to confidential information, affect data integrity or availability of a service and the respective impact. Each test is approached using a consistent and complete methodology in a way that allows the tester to use their problem solving abilities, the output from a range of tools and their own knowledge of networking and systems to find vulnerabilities that would/ could not be identified by automated tools. This approach looks at the depth of attack as compared to the Security Assessment approach that looks at the broader coverage.

Security Audit - Driven by an Audit / Risk function to look at a specific control or compliance issue. Characterised by a narrow scope, this type of engagement could make use of any of the earlier approaches discussed (vulnerability assessment, security assessment, penetration test).

Security Review - Verification that industry or internal security standards have been applied to system components or product. This is typically completed through gap analysis and utilises build / code reviews or by reviewing design documents and architecture diagrams. This activity does not utilise any of the earlier approaches (Vulnerability Assessment, Security Assessment, Penetration Test, Security Audit)

Load Testing

[https://en.wikipedia.org/wiki/Load_testing] gives the following:

Load testing is the process of putting demand on a software system or computing device and measuring its response. Load testing is performed to determine a system's behavior under both normal and anticipated peak load conditions.

Regression Testing

[https://en.wikipedia.org/wiki/Regression_testing] gives the following:
Regression testing is a type of software testing that verifies that software that was previously developed and tested still performs correctly after it was changed or interfaced with other software. Changes may include software enhancements, patches, configuration changes, etc.

Exploratory testing
https://en.wikipedia.org/wiki/Exploratory_testing

Exploratory testing is an approach to software testing that is concisely described as simultaneous learning, test design and test execution. Cem Kaner, who coined the term in 1984, defines exploratory testing as "a style of software testing that emphasizes the personal freedom and responsibility of the individual tester to continually optimize the quality of his/her work by treating test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel throughout the project."

While the software is being tested, the tester learns things that together with experience and creativity generates new good tests to run. Exploratory testing is often thought of as a black box testing technique. Instead, those who have studied it consider it a test approach that can be applied to any test technique, at any stage in the development process. The key is not the test technique nor the item being tested or reviewed; the key is the cognitive engagement of the tester, and the tester's responsibility for managing his or her time.

Smoke testing
https://en.wikipedia.org/wiki/Smoke_testing
It is testing that tries the major functions of software before carrying out formal testing