Volume 7, Issue 4, April 2017 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Study of Dynamic Testing Techniques

Avinash H.Hedaoo*
Dept. of Computer Science, Dr. Ambedkar College,
RTM Nagpur University, Nagpur, Maharashtra, India

Abha Khandelwal
Dept. of Computer Science, Hislop College,
RTM Nagpur University, Nagpur, Maharashtra, India

Abstract: ***Dynamic execution based techniques focus on the range of ways that are used to ascertain software quality and validate the software through actual executions of the software under test. It is essential to test the software in controlled and expected conditions as a complex, non deterministic system might react with different behaviors to a same input, depending on the system state. Dynamic testing techniques are generally divided into the two broad categories block box testing and white box testing. We test the software with real or simulated inputs, both normal and abnormal, under controlled and expected conditions to check how a software system reacts to various input test data.***

Keywords: Dynamic testing, Software testing, Black box testing, White box testing, Structural testing

I. INTRODUCTION

Software testing constitutes a major part of software development lifecycle. Lack of testing has resulted in many software related problems in past, and have actually brought social problems and financial losses. Despite all the efforts people put in the quality of the software, effectiveness of testing remains lower than expectations. By one estimate USA incur approximately US$50B in losses from defective software each year [1].

Software Testing is the process of executing a program with the intent of finding errors[2]. Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. [3]. Software testing is commonly known as a test to evaluate and then guarantee the level of quality of software, or to verify if there are no risks. Software testing is also used to test the software for other software quality factors like reliability, usability, integrity, security, capability, efficiency, portability, maintainability, compatibility etc[4].

Existing software testing practices are divided into two main categories : **static testing** and **dynamic testing**. Accordingly we can divide software testing techniques into two main categories according to the criterion whether the technique requires actual execution or not : static testing and dynamic testing [5].

II. DYNAMIC TESTING TECHNIQUES

Dynamic execution based techniques focus on the range of ways that are used to ascertain software quality and validate the software through actual executions of the software under test. We test the software with real or simulated inputs, both normal and abnormal, under controlled and expected conditions to check how a software system reacts to various input test data. It is essential to test the software in controlled and expected conditions as a complex, non deterministic system might react with different behaviors to a same input, depending on the system state. The dynamic testing of a software product implies execution, as only by studying the result of this execution is it possible to decide whether or not (or to what extent) the quality level set for the dynamic aspect evaluated are met.

Dynamic testing techniques are generally divided into two broad categories depending on the criterion whether we require the knowledge of the source code or not for test case design : if it does not require the knowledge of source code, it is known as black box testing otherwise it is known as white box testing [6][7].

A. Black Box Testing

Black box testing is based on the requirements specifications and there is no need to examining the code. This is purely done based on customers view point only tester knows the set of inputs and predictable outputs. Black box testing is done on the completely finished product [8] [9].

Black box testing plays a significant role in software testing, it aids in overall functionality validation of the system. Black box testing is done based on customers' requirements-so any incomplete or unpredictable requirements can be easily identified and it can be addressed later. Black box testing is done based on end user perspective. The main importance of black box testing it handles both valid and invalid inputs from customer's perspective. Black box testing is done from beginning of the software project life cycle. All the testing team members need to be involved from beginning of the project. During black box testing testers need to be involved from customers' requirements gathering and analysis phase. In the design phase test data and test scenarios need to be prepared [10].

The main advantage of black box testing is that, testers no need to have knowledge on specific programming language, not only programming language but also knowledge on implementation. In black box testing both programmers and testers are independent of each other. Another advantage is that testing is done from user's point of view. The

significant advantage of black box testing is that it helps to expose any ambiguities or inconsistencies in the requirements specifications.

Black box testing techniques are [11]:

1) *Equivalence Class Partitioning*
2) *Boundary Value Analysis*
3) *Decision Tables*
4) *State Transition Diagrams*
5) *Orthogonal Arrays*
6) *All Pairs Technique*

1) Equivalence Class Partitioning : Equivalence class *Partitioning* is based upon the assumption that a program's input and output domains can be partitioned into a finite number of (valid and invalid) classes such that all cases in a single partition exercise the same functionality or exhibit the same behaviour [5] [11]. The partitioning is done such that the program behaves in a similar way to every input value belonging to an equivalence class. Test cases are designed to test the input or output domain partitions. Equivalence class is determined by examining and analysing the input data range. Only one test case from each partition is required, which reduces the number of test cases necessary to achieve functional coverage [5] [11]. The success of this approach depends upon the tester being able to identify partitions of the input and output spaces for which, in reality, cause distinct sequences of program source code to be executed.

a) Equivalence Class Partitioning- Test Cases [5]: The Steps for creating test cases are as follows **:**

- Define the equivalence classes .
- Write the initial test case that cover as many as valid equivalence classes as possible.
- Continue writing test cases until all of the valid equivalence classes have been included.
- Finally write one test case for each invalid class.

Example: Suppose a program computes the value of the function. $\sqrt{(X-1)*(X+2)}$ This function defines the following valid and invalid equivalence classes:

X < = -2	valid
-2 < X < 1	invalid
X >= 1	valid

b) Equivalence Class Partitioning- Advantages:

- It eradicates the need for exhaustive testing, which is not feasible.
- One of the advantages of equivalence class portioning is; it enables a tester to cover a large domain of inputs or outputs with a smaller subset selected from an equivalence class.
- It allows a tester to select a subset of test inputs with a high chance of identifying a defect.

c) Equivalence Class Partitioning- Limitations:

- One of the limitations of this technique is that it makes the assumption that the data in the same equivalence class is processed in the same way be the system [7][5].
- Equivalence partitioning is not a stand-alone method to determine test case. It has to be supplemented by *boundary value analysis* [7][5].

2) Boundary Value Analysis : Boundary value analysis is performed by creating tests that exercise the edges of the input and output classes identified in the specification. Test cases can be derived from the 'boundaries' of equivalence classes. Typically programming errors occur at the boundaries of equivalence classes are known as "Boundary Value Analysis". Generally some time programmers fail to check special processing required especially at boundaries of equivalence classes. A general example is programmers may improperly use < instead of <=. The choices of boundary values include above, below and on the boundary of the class [7] [5] [11].

If an input condition specifies a range (a, b): Example: (-3, 10),

test values: 0 (interior point); 0 is always good candidate!
test values: -3, 10 (extreme points)
test values: -2, 9 (boundary points)
test values: -4, 11 (off points)

d) Boundary Value Analysis- Limitations [5]:

- One of the limitations of boundary value analysis is it cannot be used for Boolean and logical variables.
- Cannot estimate boundary analysis for some cases such as countries.
- Not that useful for strongly-typed languages.

3) Decision Tables **:** Decision tables are human readable rules used to express the test experts or design experts knowledge in a compact form [12]. Decision Tables can be used when the outcome or the logic involved in the program is based on a set of decisions and rules which need to be followed. Decision table mainly consists of four areas called the condition stub, the condition entry, the action stub and finally action entry [2] [13] [11].

a) Decision Tables-Approach [11]: The Steps for using Decision Table testing are as given below:

Step1: Analyse the given test inputs or requirements and list out the various conditions in the decision table.

Step2: Calculate the number of possible combinations (Rules).

Step3: Fill Columns of the decision table with all possible combinations (Rules).
Step4: Find out Cases where the values assumed by a variable are immaterial for a given combination. Fill the same by "Don't care Symbol.
Step5: For each of the combination of values, find out the action or expected result.
Step6: Create at least one Test case for each rule. If the rules are binary, a single test for each combination is probably sufficient. Else if a condition is a range of values, consider testing at both the low and high end of range.
Example: Consider bank software responsible for debiting from an account. The relevant conditions and actions are:

C1: The account number is correct
C2: The Signature matches
C3: There is enough money in the account
A1: Give money
A2: Give Statement indicating insufficient funds
A3: Call vigilance to check for fraud!

For the above situation the decision table for bank software consists of:
Input:

C1: Account No: Correct, Incorrect
C2: Signature: Match, not match
C3: Enough money: Yes, No
Outputs:-
A1: Give Money
A2: Give Statement of insufficient funds
A3: Call vigilance
Now Rules are:
A1 when correct account no,
A2 when signature matched
A3 when sufficient money

Table1 Decision Table For Bank Account

Condition Entries	Account No.	Signature	Money	Actions
1	Correct	Match	Yes	A1
2	Correct	Match	No	A2
3	Correct	Not match	Yes	?
4	Correct	Not match	No	?
5	Incorrect	Match	Yes	A3
6	Incorrect	Match	No	?
7	Incorrect	Not match	Yes	?
8	Incorrect	Not match	No	?

? : There are no rules define for rest of the possibilities

4) State Transition Diagrams (or) State Graphs : State Graph is an excellent tool to capture certain types of system requirements and document internal system design. When a system must remember what happened before or when valid and invalid orders of operation exists, and then state transition testing could be used. This state graphs are used when system moves from one state to another state. State graphs are represented with symbols, circle is used to represent state, arrows are used to represent transition, and event is represented by label on the transition. Thus from the starting state to the end state the various transition and routes are represented in the form of a transition diagram as mentioned [14][15] [16].

Example: Here we have to model the starship Enterprise. It has three impulse drive settings: drive (d), neutral (n) and reverse (r). The ship itself has three possible states: such as moving forward (F), moving backward (B) and stopped (S). The combinations of which impulse thrusters are firing and how the ship moves create nine states:

.dF	.nF	.rF
.dS	.nS	.rS
.dB	.nB	.rB

The impulse driver thruster control requires that you go through neutral to get to drive or reverse. All thrusters are turned off in neutral: d< >n< >r. The possible inputs are: d>d, r>r, n>n, d>n, n>d, n>r, and r>n.

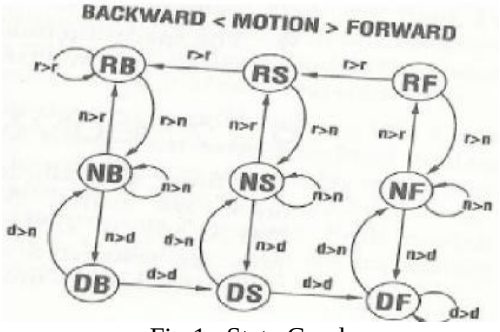

Fig.1 : State Graph

Table 2 State Table

Condition Entries	Account No.	Signature	Money	Actions
1	Correct	Match	Yes	A1
2	Correct	Match	No	A2
3	Correct	Not match	Yes	?
4	Correct	Not match	No	?
5	Incorrect	Match	Yes	A3
6	Incorrect	Match	No	?
7	Incorrect	Not match	Yes	?
8	Incorrect	Not match	No	?

5) Orthogonal Arrays : Orthogonal Array Testing Strategy (OATS) is a systematical, statistical way of testing pair-wise interactions by deriving suitable small set of test cases from a large number of scenarios. The testing strategy can be used to reduce the number of test combinations and provide maximum coverage with a minimum number of test cases. OATS utilizes an array of values representing variable factors that are combined pair-wise rather than representing all combinations of factors and levels.

Example for OATS: Here we have considered 3 parameters named as – A, B, and C and it has positive values as 1, 2 and 3. Testing all combinations of the 3 parameters would involve executing a total of 27 test cases. Generally while programming works start a fault will mostly occurs for two parameters, not for three. In this case the fault may occur for each of the 3 test cases as: A=1, B=1, C=1, A=1, B=1, C=2, and A=1, B=1, C=3. The use of OATS is no need to run all the 27 test cases, only 9 test cases are enough to test. The 9 scenarios outlined in Table3 address all possible pairs within the three parameters.

Table3 Pair-Wise Combination Of Parameters- Sample Array

	A	B	C
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

All possible pairwise combinations between parameters A and B, B and C, and C and A are displayed in Table 3 here nine scenarios provide the same coverage as executing all 27 scenarios. This same concept is applied to more complex scenarios where testing an application might require 10,000+ test cases and utilizing OATS, it can be reduced significantly in the number of test scenarios, such as down to 1,000 or less test cases to execute.

6) All Pairs Testing : This is an accepted technique for verifying a finite number of parameters with a finite number of values and keeping the number of test cases reasonable.

B. White Box Testing

White box testing mainly focus on internal logic and structure of the code. White-box is done when the programmer has full knowledge of the program structure. With this technique it is possible to test every branch and decision in the program.

When the internal structure is known it is interesting to look at different coverage criteria. One of the crucial one is decision coverage. The test is precise only if the tester recognizes what the program is supposed to do. The tester can then see if the program separates from its intended goal [17][18] [11].

White box testing is mainly used for detecting logical errors in the program code. It is used for debugging a code, finding random typographical errors, and uncovering incorrect programming assumptions [19].

White box testing is done at low level design and implementable code. It can be applied at all levels of system development especially Unit, system and integration testing. White box testing can be used for other development artefacts like requirements analysis, designing and test cases [20].

White box testing techniques are:

1) Static White Box Testing : Static white box testing which involves only the source code of the product and not the binaries or executable, static white box testing will be done before the code is executed or completed. For static white box testing only selected peoples are involved to find out the defects in the code. The main aim of the static testing is to

check whether the code is according to the Functional requirements, design, coding standards, all functionalities covered and error handling [18] [11]. Following are the types of static white box testing :

- *Desk checking:* Desk checking is the primary testing done on the code. Static checking will be done by programmers before compiled or executed, if any error is find it is going to check by author and he will correct the code, in this process the code is compared with requirements specification or design to see that the designed code is according to client adhoc requests [11].

Advantages of Desk Checking: In this process the authors who have knowledge in the programming language very well will be involved in desk checking testing. This can be done very quickly without much dependency on other developers or testers. The main advantages are defects detected in this stage are easily located and correct at same time.

- *Code walkthrough :-* This testing is also known as technical code walkthrough, in this testing process a group of technical people go through the code. This is one type of semi-formal review technique. In Code walkthrough process a high level employees involved such as technical leads, database administrators and one or more peers. The people who involved in this technical code walkthrough they raise questions on code to author, in this process author explains the logic and if there is any mistake in the logic, the code is corrected immediately .

Advantages of Code walkthrough: The main advantage of code walkthrough is that as a group of technical leads who have experience in programming look through the code, so the defects that are related to database or code can be easily identified. More over this process aid to ensure that program follows the proper coding standards [11].

- *Formal Inspections or Fagan Inspection process* : Inspection is a formal, efficient and economical method of finding errors in design and code [15]. It's a formal review and aimed at detecting all faults, violations and other side effects. According to M. E. Fagan "A defect is an instance in which a requirement is not satisfied" [16]. Fagan inspection process is a structured process of finding defects in the provided source code. The phases of Fagan inspection are as [21]:

Planning: In planning phase Moderator arrange the availability of the right participants and arrange suitable meeting place and time.

Overview: All inspection participants are given documentation of design where it includes overall view of design and even detailed design in specific areas like paths, logic of code and so on.

Preparation: Using design documentation we tried to understand the design and its logic. Depending on historical inspections and its ranking or errors we tried to increase the error detection, so that more fruitful areas can be concentrated.

Inspection: Inspection meeting involves the process in which the code is inspected and defects are found. Defects are noted down and handed to the author.

Rework: Rework is done to fix the defect. The code is corrected by the author.

Follow up: Follow up is done by the moderator to ensure that the defect has been fixed correctly.

*2) Structural White Box Testing :*Structural testing take into account the code, code structure, internal design and how they are coded. Commonly used techniques for structural testing are [22] [11]:

- *Control Flow/Coverage Testing :* It consists of Statement Coverage, Branch Coverage, Decision/Condition Coverage and Function Coverage as follows :

Statement Coverage :- In a Statement Testing each node or statements are traversed at least once, statement testing also known as node coverage [23] [24] [11].

The test cases are generated so that all the program statements are executed at least once.

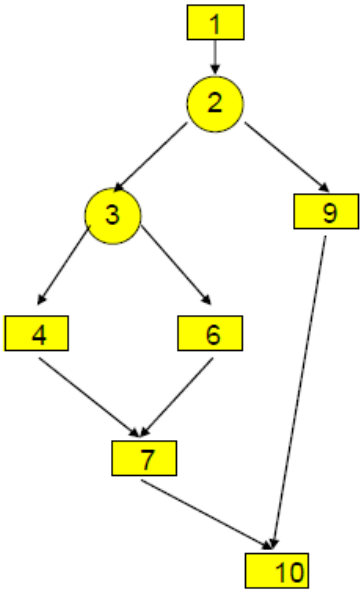

Fig 2 Statement Coverage

Branch Coverage:- In a branch testing each edge is traversed at least once. The outcome possibilities are at least true and false. The decision coverage or branch coverage is also known as Edge coverage [17]. Decision Coverage :- For

each decision, decision coverage measures the percentage of the total number of paths traversed through the decision point in the test. If each possible path has been traversed in a decision point, it achieves full coverage [17] [23] [24] [11].

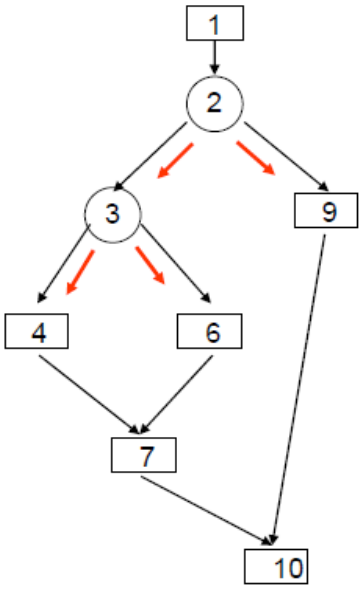

Fig 3 Branch Coverage

Decision/Condition Coverage : Here condition is to verify that all condition expression within each branch will be tested (the true and the false condition of each sub-expression within the decision branch must be tested at least one time) [17] [18] [11].

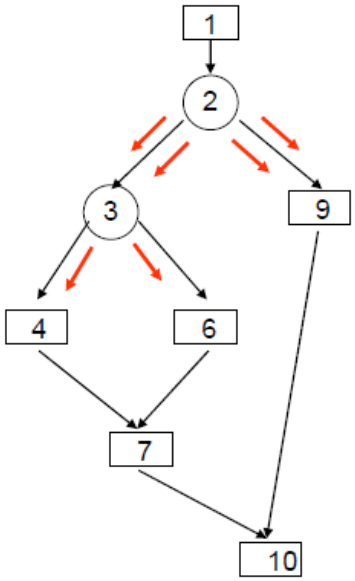

Fig 4 Condition Coverage

Condition: A condition is a Boolean valued expression that cannot be broken down into simpler Boolean expressions [2]. A decision is often composed of several Boolean conditions [17].

Function Coverage : In Function coverage, most programs are realized by calling a set of functions; in this, requirements of a product are mapped to functions during the design phase. Each function is the smallest logical unit that does a specific functionality; there could be functions for computing the average of 10 numbers, inserting a row into the database, calculating the premium etc. Tests are written to exercise each of the different functions in the code [25] [26].

- *Basic Path Testing :* It is as follows :

Flow Graph Notation : A Control Flow Graph (CFG) is a directed graph that consists of two types: node and control flow. (1) Node: expressed by a labeled circle, representing one or more statements, decision condition, procedures of program, or convergence of two or more nodes. (2) Control flow: expressed by an arc with arrow or line, can be called an edge, representing the program control flow. In a CFG, a node including condition is called a predicate node, and edges from the predicate node must converge at a certain node. Area defined by edges and nodes is referred to as region [8].

On a flow Graph: In flow graphs the symbol arrows called as Edges that represent the flow of control. Circles are called as nodes, which represent one or more actions. Areas bounded by edges and nodes called regions. A predicate node is a node containing a condition.

Any procedural design can be translated into a flow graph.

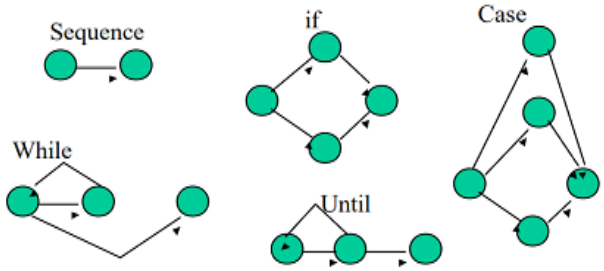

Fig 5 Different control Flow Graph Notations

Cyclomatic Complexity :The notion of cyclomatic complexity was presented by McCabe. Cyclomatic complexity is software metric that delivers a quantitative degree of the logical difficulty of a program. Cyclomatic Complexity (CYC) is derived as the number of edges of the program's control-flow graph minus the number of its nodes plus two times the number of its linked components. Cyclomatic complexity purely depend on the Control Flow Graph (CFG) of the program to be tested [27,9]. McCabe was also given calculation formula of complexity of a program structure.

$V(G) = e - n + 2$

An alternate way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = total number of non-overlapping bounded region + 1

i.e. V(G) =P+1

where P is the number of binary decision predicates.

Steps to arrive at Cyclomatic Complexity:

i. Draw a corresponding flow graph
ii. Determine Cyclomatic complexity
iii. Determine independent paths
iv. Prepare test cases

Deriving Test Cases : In this testing we have to use the design or code for drawing the corresponding control flow graphs, and have to determine the Cyclomatic complexity of the resultant flow graph V (G), after this we have to find the linearly independent paths. Finally prepare the test cases that will force execution of each path in the basis set [27][17][8][9] [11].

For example if we have six independent paths, then we should have six test cases. For each test case we need to define the input condition and expected output.

Graph Matrices : Graph matrices are used for derivation of flow graph and determination of a set of basis paths [11]. Software tools to do this can use a graph matrix.

Software tools to do this can use a graph matrix

Graph matrix:

a. Is square with #sides equal to #nodes
b. Rows and columns correspond to the nodes.
c. Entries correspond to the edges.
    1. Can associate a number with each edge entry.
    2. Use a value of 1 to calculate the Cyclomatic Complexity
a. For each row, sum column values and subtract 1
b. Sum these totals and add 1
    3. Interesting link weights are
a. Probability that a link (edge) will be executed
b. Processing time for traversal of a link
c. Memory required during traversal of a link
d. Resources required during traversal of a link

- *Loop Testing :* 1. Errors often occur near the beginnings and ends of the loop; in loop testing path has to cover at least once. a. Selects test paths according to the location of definitions and use of variables.

2. Test for loop (iterations) : a. Loop testing b. Loop fundamental to many algorithms. c. Can define loops as simple, concatenated, nested , and unstructured. Simple Loops [11]: Simple loops of size n: Skip loop entirely; Only one passes through loop; Two passes through loop; M passes through loop where, m<n. (n-1), n and (n+1) passes through the loop, Where n is the maximum number of allowable passes through the loop [11]. A typical Simple Testing loop is shown in figure 2. Nested Testing [11] : In this loop the number of probable tests increases as the number of levels of nesting grows. Start with inner loop. Set all other loops to minimum values. Conduct simple loop testing on inner loop. Work outwards. Continue until all loops are tested. A typical Nested Testing loop is shown in figure 2. Concatenated loop [11] : If independent loops, use simple loop testing. If dependent, treat as nested loops. A typical concatenated loop is shown in figure 2. Unstructured loops [11] : Don't test-redesign. A typical unstructured loop is shown in figure 2.

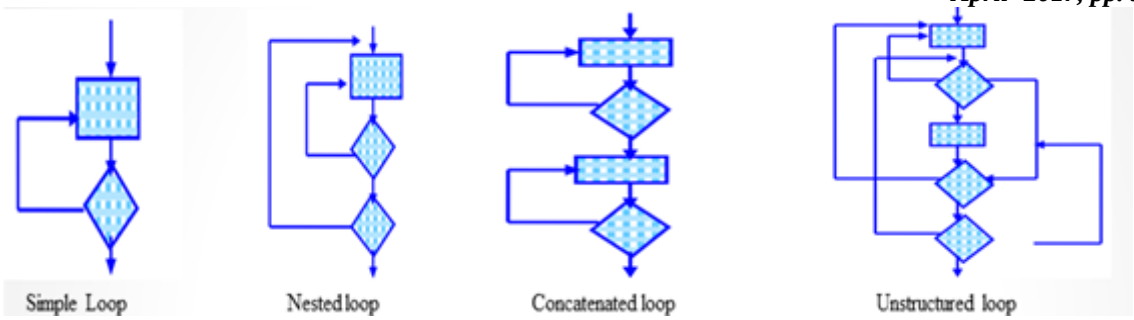

Fig 6 Different Loop Testing's

- *Data Flow Testing :* Data flow testing looks at the lifecycle of a particular piece of data (i.e. a variable) in an application [11]. Variables that contain data are created, used and killed (destroyed). Concerned with the flow of data in the program. By looking at patterns of data usage, risky areas of code can be found and more test cases can be applied. Dataflow testing uses control flow graphs to explore the unreasonable things that can happen to data. Data can be used in 2 ways- Defined and used.

Data Flow Testing_ Technique [11]: Data can be defined.

Example of defined data (Def)

```
int x;
x= a+b;
scanf(&x, &y);
X [i-1] = a+b;
```

Data can be used in a variable for performing some computations

Example of used data (Use)

```
A=X+2; (Data In X is being used for calculations)
Printf("value of x = ", x);
If(X<10)
```

Select paths through the program's control flow and test the status of data in each of these paths.

Pick enough paths to ensure that every data object has been initialized prior to use or all defined objects have been used for something.

All the def criteria (for definitions of all variables) must be exercised. All the use criteria of all variable definitions must be covered.

III. CONCLUSION

In this paper we proposed dynamic testing techniques. In this study we cover almost all testing techniques related to dynamic testing techniques. It is found that dynamic techniques are very useful if implemented in right manner in SDLC. To be effective we should use correct dynamic technique at correct place (which technique to use, where to use it in SDLC). We should have detail knowledge of static techniques (what is technique all about) to select appropriate technique and implement that properly. We should use both static techniques and testing (dynamic); Static techniques are about prevention while dynamic testing is about cure.

REFERENCES

[1] NIST-Final Report "The Economic Impacts of Inadequate Infrastructurefor Software Testing", Table 8-1, National Institute of Standards and Technology, May 2002.

[2] The art of software testing / Glenford J. Myers ; Revised and updated by Tom Badgett and Todd The Art of Software Testing, Second Edition Thomas, with Corey Sandler.—2nd ed. p. cm. ISBN 0-471-46912-2

[3] Exploratory Testing, Cem Kaner, Florida Institute of Technology, Quality Assurance Institute Worldwide Annual Software Testing Conference, Orlando, FL, November 2006

[4] Quadri, S.M.K and Farooq, SU, "Software Testing – Goals, Principles, and Limitations", International Journal of Computer Applications (0975 – 8887) Volume 6– No.9, September 2010

[5] Roper 1995

[6] Sommerville, 2007

[7] Beizer , 1995

[8] P. Mitra, S. Chatterjee, and N. Ali, "Graphical analysis of MC/DC using automated software testing," in Electronics Computer Technology (ICECT), 2011 3rd International Conference on 2011, vol. 3, pp. 145 – 149.

[9] T. Murnane and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique," in Software Engineering Conference, 2001. Proceedings. 2001 Australian, 2001, pp. 12 – 20

[10] T. Murnane and K. Reed, "On the effectiveness of mutation analysis as a black box testing technique," in Software Engineering Conference, 2001. Proceedings. 2001 Australian, 2001, pp. 12 –20.

[11] Weinberg, Gerald M., "The psychology of computer programming", Dorset House Pub.,1998, 292 pages

[12] Sommerville, I, "Software Engineering", Pearson, 2008, 864 pages, ISBN 978-81-317-2461-3

[13] Eldh 2011
[14] Kaner, Cem; Nguyen, Hung Q; Falk, Jack (1988). Testing Computer Software (Second ed.). Boston: Thomson Computer Press. ISBN 0-47135-846-0.
[15] Static Testing C++ Code: A utility to check library usability. http:/ / www. ddj. com/ cpp/ 205801074
[16] Fagan, Michael E: "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol. 15, No. 3, 1976; "Inspecting Software Designs and Code", Datamation, October 1977; "Advances In Software Inspections", IEEE Transactions in Software Engineering, Vol. 12, No. 7, July 1986
[17] P. Mitra, S. Chatterjee, and N. Ali, "Graphical analysis of MC/DC using automated software testing," in Electronics Computer Technology (ICECT), 2011 3rd International Conference on, 2011, vol. 3, pp. 145 –149.
[18] Glenford J. Myers, "The art of software testing" ; Revised and updated by Tom Badgett and Todd The Art of Software Testing, Second Edition Thomas, with Corey Sandler.—2nd ed. p. cm. ISBN 0-471-46912-2
[19] http://www.internetjournals.net/journals/tir/2009/January/Paper%2006.pdf
[20] F. Saglietti, N. Oster, and F. Pinte, "White and grey-box verification and validation approaches for safety-andsecurity-critical software systems," Information Security Technical Report, vol. 13, no. 1, pp. 10–16, 2008.
[21] A. Bertolino, "Knowledge Area Description of Software Testing",Chapter 5 of SWEBOK: The Guide to the Software Engineering Body of Knowledge. Joint IEEE-ACM Software Engineering Coordination Committee. 2001. http://www.swebok. org/.
[22] Koza, C. and Puschner, P., " Calculating the maximum execution time of real-time programs.", Real-Time Systems, 1:159- 176,1989.
[23] Graham, Dorothy, Van Veenendaal, Erik, Evans, Isabel and Black Rex, "Foundations of Software Testing: ISTQB Certification", Thomson, 2006, 258 pages.
[24] Sasidhar, K.N., "Static Techniques",http://www.scribd.com/doc/ 4548375/ Static-Testing-Techniques, Accessed on 8 Jul 2013.
[25] Park, C.Y., "Predicting program execution times by analyzing static and dynamic program paths.", Real-Time Systems, 5:31-62, 1993.
[26] IEEE Std . 1028-1997, "IEEE Standard for Software Reviews", clause 3.5
[27] H. Liu and H. B. Kuan Tan, "Covering code behavior on input validation in functional testing," Information and Software Technology, vol. 51, no. 2, pp. 546–553, Feb. 2009.

