Impact of Dependency Graph in Software Testing

Pardeep Kaur[1], Er. Rupinder Singh[2]
[1]*Computer Science Department, Chandigarh University, Gharuan, Punjab*
[2]*Assistant Professor, Computer Science Department, Chandigarh University, Gharuan, Punjab*

Abstract - A Software development environment is conceived an interactive system which support program development. The internal program representation selection is vital role for software development environment according to their nature. For this purpose, we have an intermediate graph, a dependency graph that represent the data and control flow dependencies between nodes, statements. It must also play a role in software testing. Testability in software is important quality characteristics for checking the maintenance effort and provides help for finding the post release failures. We also find the exact feature location by using Dependency graph, where the user wants actual changes in software. It must also helpful for regression testing that based on code or software design. We also have a system dependence graph that supports object oriented features like class, objects, inheritance etc.

Keywords - Software testing[1], dependency graph, software complexity metrics, program slice, regression testing, program dependency graph, control flow dependency graph, data flow dependency graph, Feature location, system dependence graph.

I. INTRODUCTION

The software engineering terminology is basically "architecture" as the organizational structure of a System or component. The scope of software Testing includes examination of code as well as execution of that code [4] in the scenario of software development; a testing organization is different from the development team. In the context of software testing process, is a long process, which can be done by manually or automatically by following such a long process. So here we define an intermediate solution for testing process is dependency graph. A dependency graph is basically a collection of nodes and edges. A node represents the statements, procedures and edges represent the control and data flow between those statements. Here we represent the need of dependence graph in s/w testing. It represents the scenario when one component of our software is dependent on other, called dependent component [3]. In the software development environment, dependency graph may used in software testing for debugging by making slices of program, test case generation, test automation, maintenance, code optimization, software re-engineering and for depicting various software complexity metrics. In this paper, we also present a new approach to define metrics for software dependencies.

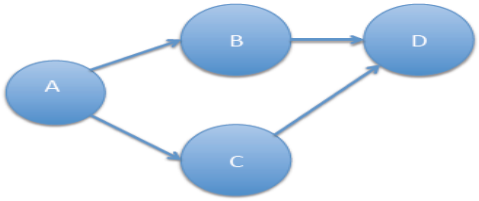

Fig.1: Dependency graph[2]

- A directed graph is a pair G =(V,E)of:
- A set V of elements called vertices or nodes
- A set E of ordered pair vertices, called edges (arrows) and in figure1 we have 4 nodes (A,B,C,D) and 4 edges (AB,BD,AC,CD)

A. *Important Terms*

- *Dependency:* Dependency of component an on component B exists if component A requires component B to compile or function correctly. Dependencies define a relation between components, i.e. between two components there is at most one direct dependency in each direction.
- *Data Dependencies:* when statements compute data that are used by other statements.
- *Control Dependencies:* are those which arise from the ordered flow of control in a program.

[1] Software testing mainly provides the information about the quality of product. It also provides an objective, independent view of software and check that particular software meets the user requirements or not. It is also helpful for ensuring the verification and validation of software development phases.

[2] . A dependency graph is a directed graph representing dependencies of several objects towards each other.

- *Component* is a static building block of a system which can be a module, a class or interface, a package, or a subsystem.
- *Strength:* The strength of a dependency increases with the number of relationships that cause the dependency.
- *Structure of dependency graph:* The structure of a system is defined by its components and the dependencies between the components and can be represented as a directed dependency graph.
- *Feature location:* Concept location is a process that maps domain concepts to the software components, Feature or concepts Location is relatively easy in small systems, which the Programmer fully understands the task. For large and complex systems, it can be a risky task.
- *Data Dependency graph:* A data dependence graph contains nodes represent program statements and edges represent data dependencies between statements. A data dependency graph **G= (I, E)** consists of set of Instructions **I** and a Set of transitive relation **R= I x I,** with **(a, b)** in **R** if the instruction **a** must be evaluated before **b**. In other words, there is an edge from a to b: If a must be evaluated before b.

Fig.2: Data dependency between a & b

- *Control flow graph (CFG):* CFG is language-independent and machine independent representation of control flow in programs used in high-level and low-level code optimizers. It represents the control information between nodes of edges.

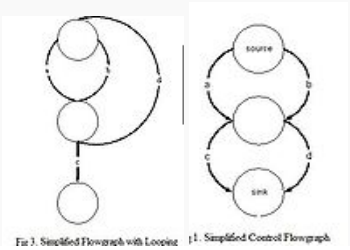

Fig.3. Control flow graphs

In these CFG's, control information must pass from one node to another.

- *Program Dependence Graph:* A PDG is a mapping of dependencies, useful in optimizing transformations for utilizing multiple cores and parallelism**.**

- *Program Dependence Graph (PDG)* consists of

1. Set of nodes, as in the CFG
2. Control dependence edges
3. Data dependence edges

II. DEPENDENCIES AND TESTABILITY

For testing scenario, we check Direct and indirect dependencies of a component.

- Find the dependee components[3] and dependent components of a graph.
- The multiple effects based on the time and effort need to test a graph, are the following ones:
- The dependee components have to be considered during test design.
- More components have to be compiled before test execution which increases the time to rebuild the system after component changes.
- The dependee components have to be instantiated during test setup.
- The components involved in the cycle have to be tested at once.

A. Software dependencies

The dependencies[4] between the components must be represented in the hierarchy of a graph. Edges represent the dependencies between nodes; the leaves of the tree provide lower level services that higher components depend on. Number of dependencies between components represents degree of coupling. A software dependency is a relationship between two pieces of code, such as a data dependency or call dependency. A relation (A, B) between binaries A and B signifies that A makes a call on B. Here two Entities A and B, A call to B from many different Call sites. The count of a dependence (A, B) is either 0 or 1, based on whether A contains a call to B (1) or not (0). The frequency of dependence (A, B) is the (total) number of calls from A to B.[4] All information is analyzed and based on this information we collect a set of eight dependency measures described below for each binary.

- Same Component Count
- Same Component Frequency
- Different Component Count
- Different Component Frequency

[3] A component that depends on another component is called a dependent component; a component that is required by some other component is called a depended component.

[4] A software dependency is a relationship between two pieces of code, such as a data dependency (component A uses a variable defined by component B) or call dependency (component A calls a function defined by component B).[4]

- Same Area Count
- Same Area Frequency
- Different Area Count
- Different area Frequency

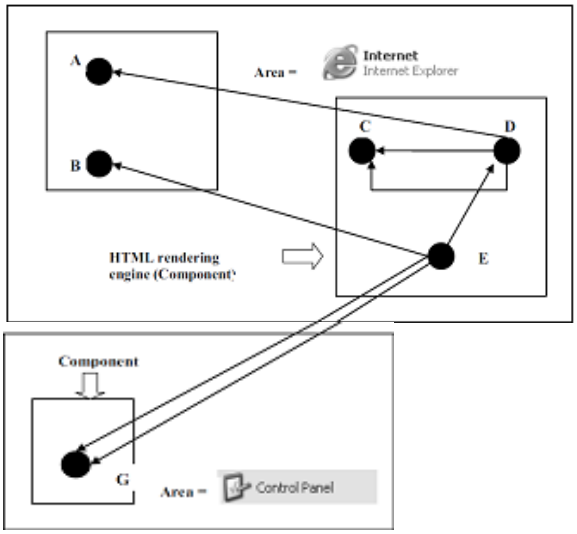

Fig.4: Metric description

Binary Name	Same Component Count	Same Component Frequency	Different Component Count	Different Component Frequency	Same Area Count	Same Area Frequency	Different Area Count	Different Area Frequency
D	1 (D→C)	2 (D→C) (D→C)	1 (D→A)	1 (D→A)	2 (D→C) (D→A)	3 (D→C), (D→C), (D→A)	0	0
E	1 (E→D)	1 (E→D)	2 (E→B) (E→G)	3 (E→B), (E→D), (E→G)	2 (E→B) (E→D)	2 (E→B) (E→D)	1 (E→G)	2 (E→G) (E→G)

Fig.5: Software dependency measurement description

In these both figures we Consider the dependence frequencies/counts for the binary (D) in Internet Explorer area. The binary (D) has three outgoing dependencies. Two of these are within the component (HTML rendering engine), directed from binary D to binary C. So the same component dependence frequency is two and the same component dependence count is one (i.e. D→C). There exists one dependence between the binary D and binary A in different component. The different Component frequency and the different component count is thus one (D→A) There is no dependence from binary D (in the Internet Explorer area) to the Control Panel area. This is the cross area dependency. The different area count and frequency are hence zero.

B. *Software churn*

Code churn is a measure of the amount of code Change with respect to time[5]. This measure is used to compute the overall change in Lines of code by added, deleted, and modified [4]. Suppose component A has many dependencies on component B. If the code of component B changes, component A will need to change the amount of churn to keep synch with component B.

III. DEFINING METRICS FOR DEPENDENCIES

Metrics is the unit of measurement; define local characteristics and global characteristics of an entire system. First is, reduction metrics check the impact of a particular Dependency [3] a reduction metric r_m describes the degree in which quality metric m is reduced if a dependency d is removed.

A. *Program Complexity Metrics*

In the process of software development, measurements provide important information to programmer.[1] Measurements such as McCabe's cyclomatic complexity metric and Hoffman's reach ability metric for control flow graph, Halstead metric are based on count for operators and operands. Other complexity metric is information flow metric defined by Henry, describe data values flow in program.

B. *Testability metric*

For each activity and sub-activity we identify design attributes that have an impact on testability [3] each attribute is then decomposed into lower-level attributes. Testability for unit testing is concerned with low-level design artifacts, whereas testability

For integration testing is concerned with high-level design artifacts.

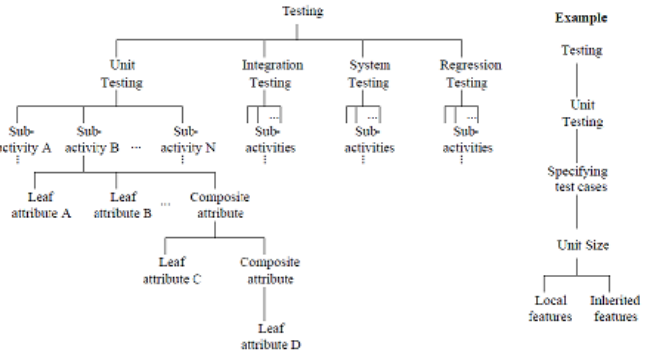

Fig.6: Testability attributes classification

We first provide an overview of the structure of our framework and establishing their relationship to testability.

[5] Software fault-proneness is defined as the Probability of the presence of faults in the software.

- *Unit testing:* A unit represents a class, a cluster of classes or a subsystem.
- *Integration testing:* A set of units is being integrated, preferably in a stepwise manner, and testing focuses on making interfaces between units.
- *System testing:* The system is tested as a whole. Though drivers are needed, stubs are only required for external devices and systems.

For testing sub-activities:

- *Specifying test cases:* this activity consists of all tasks that aim at defining the specification of test cases based on software artifacts such as specifications, design, or code.
- *Developing drivers:* this activity consists of writing the required code to execute Test cases.
- *Developing stubs:* This activity consists of developing stubs emulating the behavior of components.

Our important observation is that an attribute can impact the testability of several testing activities. Moreover, attributes (e.g., size) can be decomposed in different ways according to the (sub-) activities they impact. It results that the measure of a given testability attribute may vary according to the activity being considered. For instance, "Unit coupling" is a testability attribute that has an impact on both "Unit Testing" and "Integration Testing".

1. Goal-Question-Metric

Other meaningful metrics to measure testability, we apply the goal question-metric approach. This means that we 1)define goals related to testability improvement, 2) describe questions that help to evaluate the degree to which goals have been achieved, and 3) define metrics that allow to Answer the questions.

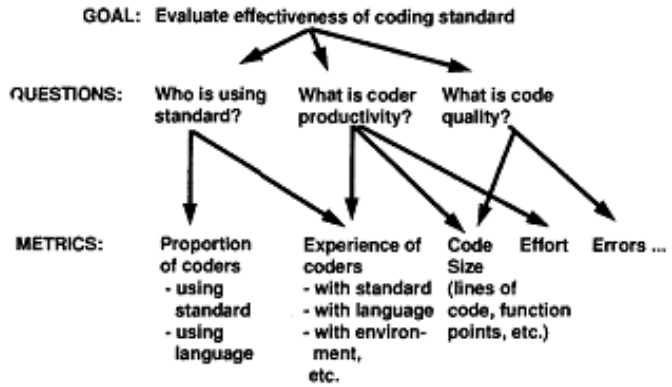

Fig.7: Goal-Question-Metric

IV. PROGRAM SLICING IN TESTING

Programmer use slicing, while debugging of program. In the software testing[6] environment some program may be small or some may be complex. It based on the LOC (Lines of code). For large or complex programs, program has to be sliced for easily understandable the code. Slicing must be based on some value or variable that must be relevant for all variables in a code [1] and call automatically for further development of code.

V. TEST SCENARIO DEPENDENCIES AND TRACEABILITY IN REGRESSION TESTING

Regression testing is important quality Assurance technique, it based on code or software design. Regression testing check the system working, after changes being made there, changes like requirement change or any technology change, etc. It mainly verifies integrality and correctness of the modified System. Two strategies for selecting Regression test cases: retest-all and Selective-retest[5] Most of the existing regression test selection techniques are code-based using program slicing, program dependence graphs, data flow and control flow analysis. Dependence analysis provides the basis for regression testing and ripple effect analysis. The following lists the kinds of dependence:

- *Functional Dependence:* A functional Requirement test the system on basis of normal input, and exception handling. Thus functional dependence identifies how a set of test scenarios is related to each other.
- *Input Dependence:* Input dependence identifies the common inputs shared by a set of test Scenarios, including input data, actions and triggering events.
- *Output Dependence:* Similar to input Dependence, test scenarios may also share Common output, either data or messages.
- *Input/output Dependence:* An output of a test Scenario may be an input of another test scenario, and any change to one of the test scenarios may affect the other one.
- *Execution Dependence:* Execution dependence Captures component and interaction relationships between individual execution paths of test Scenarios.
- *Ripple Effect Analysis* (REA) is used to analyze and eliminate negative effects due to changes and to ensure consistency and integrity after changes are made to software.
- *Traceability analysis in Regression testing:* By using traceability information, we can find affected components

[6] **Unit testing** works on smaller or individual components whereas **integration testing** integrate those components and test them and **system testing** test whole or single system that is collection of those smaller components.

and their associated test scenarios and test cases for regression testing.

VI. FEATURE LOCATION FOR TESTING USING DEPENDENCY GRAPH

In the software development and maintenance process, change request arises by customer to add or modify some concept or feature in particular area. Before any changes can be made to the system, software programmers must locate the concept location.[2] Here input is a change request send by customer, expressed in natural language and output is set of those components, where we implement the concept or feature.

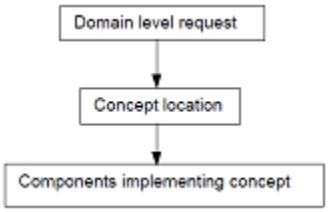

Fig.8 concept location process

In this Figure, in the first level user send domain request for make changes in particular area in software and in the next level with help of dependency graph find the concept location area and last changes made by development team in exact component. Feature or concept location[7] is easy in small systems, where the programmer fully understands it. For large and complex systems, it can be a considerable task. Because in this process, here is translation from input level to implementation level, for this task extensive knowledge is required, including domain knowledge, Programming knowledge, knowledge of algorithms and data structures, knowledge of the software components and their interactions, etc., the programmer who has this knowledge must participate in the location process. So, in each step of the search, one component is chosen for visit. All visited components and their neighbors constitute a search graph. At the starting, the search graph contains only the one component. Each visit to a component expands the search graph, by explores the source code, dependence graph, and documentation, and the process continues until all the components implementing the feature or concept are located.

[7] Concept location is a process that maps domain Concepts to the software components.

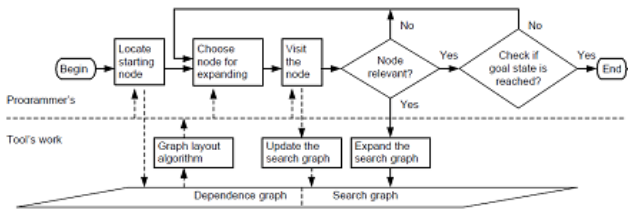

Fig.9 Feature Location using Search graph

In this figure, it represents the feature location concept with help of search graph; the programmer has to do the following:

- *Locate starting component:* At the beginning, little is known about the system. The starting point is often the top component, i.e. function main (), because the top component summarizes all the requirements of the entire system.
- *Choose a component for visit:* In every step, one Component is selected for a visit and expansion of the search graph. The programmer explores the source code, dependence graph, and documentation.
- *Check if goal is reached:* The programmer checks whether all components dealing with the feature have been found.
- *Extract dependence graph of the system:* Dependence graph is extracted from the source code by the program analyzer.
- *Update the search graph:* After the programmer: Visited a component, the tool will add it to the search Graph. Based on the search graph, the programmer can backtrack, undo, or redo some of his/her previous Operations. If a step is not a backtracking one, the selected component will be investigated and the search graph will expand.

There are several strategies of search graph expansion:

- *Top-down strategy* expands search graph by called functions. The scenario starts with a function main () that summarizes the requirements for the whole program.
- *Bottom-up strategy* is the opposite of top-down Strategy and expands the search graph by calling Functions.
- *Backward data flow strategy* is employed when functionality of the system depends on specific values in specific variables.
- *Forward data flow strategy* is the opposite and the Programmer is searching for the destination of the Values.

So, feature location is a sequence of Search graphs S1, S2, Sn, starts with a single component search graph and ends with the feature located. At the beginning of the search, S1 =

{s<t>}. Formally S is a search graph if and only if there exists d Î comp(S)
Such that s<d> Î S (d is selected component) or d is starting component.

VII. PREDICTING DEFECTS WITH PROGRAM DEPENDENCIES

A program dependency is a direct relationship between two Pieces of code (variables, expressions, methods). There exist dependencies are: a data dependency between the definition and use of Values and call dependencies between the declarations of functions and function call [7] Software development is a complex and error-prone task. An important factor in the development of complex systems is the understanding of the dependencies that exist between two pieces of the code. Dependencies must be helpful for predict post-release defects[8] in software program. Dependencies are decided during design or early in the implementation phase, and prediction models can be used to estimate the risk of failure. Code churn and dependencies can be used as efficient indicators of post-release defects. We can also make predictions about the presence of defects in system software. We address two problems:
1. **Classification.** Can we predict which binaries (executable files (COM, EXE, etc.) and dynamic-link files (DLL) will have defects?
2. **Ranking.** Can we predict which binaries will have the most defects?

A. Detection of code clone by using Program dependence graph

In the software development for enhancing the code quality, we have a technique of code clone or code reuse. PDG-based detection is suitable to Detect non-contiguous code clones whereas other detection techniques are string-based or token-based are not suited to detect them.[6] The advantage of PDG based detection is that it is suitable to detect non-contiguous code clones. Non-contiguous code clones are ones whose elements are not consecutively located on the source code. Major categories should be line-based, token-based, metrics-based, AST based, and PDG-based. We introduce special dependency, execution dependency to Program Dependency Graphs. ED is the same as control flow of control flow Graphs, there is an ED between two nodes if the program element represented by one node may be executed just after the program element represented by the other node.

B. Identifying Similar Code with Program Dependence Graphs

In the testing process, Duplicity code is common problem in all kind of software systems. If we perform code reusability in our software system, in coding phase then problem can occur if code was not in correct format, there should be logical errors and computation error and they must pass into another level of coding, if we use code reuse. So, we have to find duplicate code at exact time. Similar code based on finding similar sub graphs of directed graph. Program dependence graph (PDG) is a directed graph whose vertices represent the statements and control predicates that occur in a program [8]. Some of the vertices are entry vertices, which represent the entry of procedures. The edges represent the dependences between the components of the program. They have two attributes: the first is separating the edges into control and data dependence edges. The approach must follow fine-grained program dependence graphs (PDGs) which represent the structure of a program and the data flow within it.

VIII. SYSTEM DEPENDENCE GRAPH CONSTRUCTION FOR ASPECT-ORIENTED PROGRAMS

We have other dependence based representation, is system dependence graph used for aspect mining[9]. It construct module dependence graph for each introduction, method we use in aspect and classes [9]. It also support various object oriented features like classes and objects, inheritance, polymorphism, encapsulation. It connects various MDG' to form SDG and for this purpose it uses various vertices of graph like call vertex, formal-in, formal-out vertex. These formal parameter vertices used to passing parameter between methods. Other we have actual-in vertex used for actual parameter and actual-out parameter for identifying the parameters must be modified by called method.

IX. CONCLUSION

In SDLC Software testing is one of the major concern areas that are time consuming, high effort is needed for performing the testing manually. To solve this issue after reviewing the literature of software testing is has been found that there is need of an intermediate graph. Intermediate or dependency graph is one of the area in software testing that are using for debugging by making slices of program, test case generation, test automation, maintenance, code optimization, software re-engineering and for depicting various software complexity metrics. Dependency graph must also helpful for finding the concept location, where user wants actual changes in existing software. Dependency graph also helps for finding the post

[8] Defect-proneness is the probability that a particular software element has a defect that will lead to a failure. Post release defects leading to failures that occurred in the field within six months after release of software.

[9] Aspect oriented programming is new language paradigm proposed for modularizing the cross cutting structure of concerns like exception handling, synchronization and resource sharing.

release failures into existing software. In this study, we have discuss the impact of dependency graph in software testing after analyzing various dependency graphs like SDG, PDG, CDG, DDG etc. The propose of the research to discuss the issues surrounding the use of dependency graph in software environment, and to find the single super dependency graph which will overcome the issues of others graphs as well as can be implemented in any testing strategy as intermediate graph.

X. REFERENCES

[1] Ottenstein, K. J., & Ottenstein, L. M. (1984, April). The program dependence graph in a software development environment. In ACM Sigplan Notices (Vol. 19, No. 5, pp. 177-184). ACM.

[2] Chen, K., & Rajlich, V. (2000, June). Case study of feature location using dependence graph. In International Conference on Program Comprehension (pp. 241-241). IEEE Computer Society.

[3] Jungmayr, S. (2002, October). Testability measurement and software dependencies. In Proceedings of the 12th International Workshop of (pp179- 202).

[4] Nagappan, N., & Ball, T. (2007, September). Using software dependencies and churn metrics to predict field failures: An empirical case study. In Empirical Software Engineering and Measurement, 2007. ESEM 2007.First International Symposium on (pp. 364-373). IEEE.

[5] Tsai, W. T., Bai, X., Paul, R., & Yu, L. (2001). Scenario- based functional regression testing. In Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International (pp. 496-501). IEEE.

[6] Higo, Y., & Kusumoto, S. (2009, October). Enhancing quality of code clone detection with program dependency graph. In Reverse Engineering, 2009. WCRE'09.16th Working Conference on (pp. 315-316). IEEE

[7] Zimmermann, T., & Nagappan, N. (2009, October).Predicting defects with program dependencies. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (pp. 435-438). IEEE Computer Society.

[8] Johnson, R. & Pintail, K. (1993, August).Identifying similar code with program dependence Graph. In ACM Sig Plan Notices (Vol. 28, No. 6, pp. 78-89). ACM.

[9] Zhao, J., & Reynard, M. (2003). System dependence graph construction for aspect-oriented programs.