# Testing Methodology of Algorithm in Slicing of Program Elements

Sushree Sujata[1], Kunwar Babar Ali[2]
[1]*P.G student,* [2]*Assistant professor,*
[12]*department of computer science, Noida, International university, Noida, India*

***Abstract-*** Even after thorough testing of a program, usually a few bugs still remain. These residual bugs are usually uniformly distributed throughout the code. It is observed that bugs in some parts of a program can cause more frequent and more severe failures compared to those in other parts. It should, then be possible to prioritize the statements, methods and classes of an object-oriented program according to their potential to cause failures. Once the program elements have been prioritized, the testing effort can be apportioned so that the elements causing most frequent failure are tested more. Based on this idea, in this paper we propose a program metric called the influence of program elements. Influence of a class indicates the potential of class to cause failures. In this approach, we have used the intermediate graph representation of a program. The influence of a class is determined through a forward slicing of the graph. Our proposed program metric can be useful in applications such as coding, debugging, test case design and maintenance etc.

***Key Words-*** Prioritization of Program Elements, Slicing, Intermediate representation, Program testing, Object-oriented programming.

## I. INTRODUCTION

Software solutions are increasingly permeating our everyday life. Software industries are in immense pressure to provide very reliable products where tolerance to bugs is very less. Usually testing of the software products is carried out in various levels to identify all defects existing in the software product. However, for most practical systems, even after satisfactorily carrying out the testing process, it is not possible to guarantee that a software product is error free. This situation is caused by the fact that input data domain of most software products is very large. Also, each software product development project is constrained by time and cost. As a result, it is not practical to test a software product exhaustively using each value that the input data may assume. At present, testing takes on an average 50% of the total development cost and time. Thus, possibility of increasing the testing effort any further appears bleak. In traditional testing techniques, each element of the software product is tested with equal thoroughness. This causes usually a uniform distribution of bugs in the software product. But presence of bugs in some parts cause more severe and frequent failures than other parts. For example, if a statement produces crucial data that is useful for many other statements, then an error in this statement would affect many other statements. So our aim is to identify those more critical parts of a program, for which more exhaustive testing has to be carried out. We define influence of an element as the measure of criticality and severity of that element. We proposed a metric to compute the influence of a statement and influence of a method. With the help of these two metrics we can calculate the influence of a class. The characterization of code can help in designing, coding, testing and maintenance phases of software development cycle. We use Extended System Dependent Graph for intermediate code representation.

## II. CONCEPT

**Slicing**

A program slice is a part of the code that contributes in computation of certain variable at a program point of interest. More formally a slice can be defined as follows:

1. **Program Slice**
2. In computer programming, **program slicing** is the computation of the set of program statements, the **program slice** that may affect the values at some point of interest, referred to as a **slicing criterion**. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control.
3. Slicing techniques have been seeing a rapid development since the original definition by Mark Weiser. At first, slicing was only static, i.e., applied on the source code with no other information than the source code. BogdanKorelandJanusz Laski introduced *dynamic slicing*, which works on a specific execution of the program (for a given execution trace). Other forms of slicing exist, for instance path slicing.
4. . For statement s and variable v, the slice of a program P with respect to the slicing criterion < s, v > includes only those statements of P needed to capture the behaviour of v at s.

5. Static slicing

Based on the original definition of Weiser, informally, a static program slice S consists of all statements in program P that

may affect the value of variable v in a statement x. The slice is defined for a slicing criterion C=(x,v) where x is a statement in program P and v is variable in x. A static slice includes all the statements that can affect the value of variable v at statement x for any possible input. Static slices are computed by backtracking dependencies between statements. More specifically, to compute the static slice for (x,v), we first find all statements that can a directly affect the value of v before statement x is encountered. Recursively, for each statement y which can affect the value of v in statement x, we compute the slices for all variables z in y that affect the value of v. The union of all those slices is the static slice for (x,v).

D**ynamic slicing:**

Makes use of information about a particular execution of a program. A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program.

- Example to clarify the difference between static and dynamic slicing. .consider a small piece of programunit , in which there is an iteration block containing an if- else-block . There are few statement in both the if and else block that an effect the variable. In case of static slicing since the whole program unit is locked at irrespective of particular execution of the program , the affect statement in both block would be included in the slice..
- Dynamic slicing makes use of the information about a particular execution of a program. The execution of a program is monitored and the dynamic slices are computed with respect to execution history. A dynamic slice with respect to a slicing criterion < s, v >, for a particular execution, contains those statements that actually affect the slicing criterion in the particular execution. Therefore, dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing.

## III. PROGRAM REPRESENTATION

In the following, we present a few basic concepts associated with intermediate representation of program that are used in later sections.

**Control Flow Graph**

- The control flow graph (CFG) is an intermediate representation for programs that is useful for data flow analysis and for many optimization code transformations such common sub expression elimination, copy propagation, and loop invariant code motion
- In a control flow graph each node in the graph represents a basic block , i.e. a straight-line piece of code without any jumps or jump targets ; jump targets start a block, and jumps end a block.
- . Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the *entry block*, through which control enters into the flow graph, and the *exit block*, through which all control flow leaves.[[3]
- The CFG can thus be obtained, at least conceptually, by starting from the program's (full) flow graph—i.e. the graph in which every node represents an individual instruction—and performing an edge contraction for every edge that falsifies the predicate above, i.e. contracting every edge whose source has a single exit and whose destination has a single entry. This contraction-based algorithm is of no practical importance, except as a visualization aid for understanding the CFG construction, because the CFG can be more efficiently constructed directly from the program by scanning it for basic blocks

## IV. PROGRAM DEPENDENCE GRAPH

**Program Dependence Graph** (**PDG**) in computer science is a representation usinggraph notation that makes data dependencies and control dependencies explicit. These dependencies are used during dependence analysis in optimizing compilers to make transformations so that multiple cores are used, and parallelism is improved

- The program dependence graph *G* of a program *P* is the graph *G* = (*N*, *E*), where each node *n* 2 *N* represents a statement of the program *P*. The graph contains two kinds of directed edges: control dependence edges and data dependence edges. A control(or data) dependence
- An edge (m, n) indicates that *n* is control (or data) dependent on *m*. Note that the PDG of a program *P* is the union of a pair of graphs: Data dependence graph and control flow graph of *P*.

**System Dependence Graph**

- The PDG can't handle procedure calls. Horwitzetal . Introduced the System Dependence Graph (SDG) representation which models the main program together with all associated procedures. The SDG is very similar to the PDG. Indeed, a PDG of the main program is a sub graph of the SDG. In other words, for a program without procedure calls, the PDG and SDG are identical. The technique for constructing an SDG consists of first constructing a PDG for every procedure, including the main procedure, and then adding dependence edges which link the various sub-graphs together.
- An SDG includes several types of nodes to model procedure calls and parameter passing:

• Call-site nodes represent the procedure call statements in the program.

• Actual-in and actual-out nodes represent the input and output parameters at call site. They are control dependent on the call-site nodes.

• Formal-in and formal-out nodes represent the input and output parameters at called procedures. They are control dependent on procedure's entry node.

• Control dependence edges and data dependence edges are used to link the individual PDGs in an SDG. The additional edges that are used to link the PDGs are as follows:

• Call edges link the call-site nodes with the procedure entry nodes.

• Parameter-in edges link the actual-in nodes with the formal-in nodes.

## V. METHOD

An object-oriented program comprises of a set of classes. We assume that each class consists of variables and methods. Influence of a class is sum of influence of all it's elements. So we calculate influence of each statement and if a statement involves a method call then influence of method will also be calculated. Our approach is based on static analysis of the code and it does not consider the value of variables. So it can't deal with recursive function calls and loops effectively. Sum of influence of all statements and all relevant methods is the influence of class. This approach statically computes the influence of a class. Execution of program is not necessary. First, we construct the intermediate representation (SDG/ESDG) of the program. Then, we calculate the influence of desired element using the proposed algorithms. We first discuss computation of influence of a statement, then subsequently influence of method and influence of class are discussed.

**Influence statement:**In a program the result of one statement may depend on the result computed by other statements. If the influence is more, then the statement is more critical. The influence of the statement is defined by the number of other statements of the given program which use that variable directly or indirectly. We give a metric to calculate influence considering no call vertex. If a statement is call vertex then its influence will be calculated separately using the influence of method metric and will be added to get total influence of the desired statement. Influence of the statement expressed as a percentage is given by:

$$\frac{\text{Total number of nodes marked influenced}}{\text{Total number of nodes in graph}} * 100$$

Let us say *In f luence*(*u*, *stmt*) denote the node *u* and statement '*stmt*0, where *stmt*can be any variable or 'if' or 'while' or 'printf' etc. Let (*x*1, *u*1), (*x*2, *u*2), ...(*xk*,

*uk*) be all there outgoing data flow edges of *u* in the PDG of that program. Where *x*1, *x*2, ...,*xk*are dependency edges and *u*1, *u*2, ..., *uk* are nodes.

So influence of a statement corresponding to node *u* is given by:

*Influence*(*u*, *stmt*) = {*u*1, *u*2, ..., *uk*}[{*Influence*(*u*1, *stmt*1)[*Influence*(*u*2, *stmt*2)[. . .[*Influence*(*uk*, *stmtk*)}

**Algorithm**

**Input**: Program code and the *statement*.

**Output**: *In f luence*of given statement.

***StmtInfluence(statement)***{

*1. Construct ESDG of the program statically.*

*2. For statement traverse it's all dependency edges and mark them.*

*3. For each marked node repeat step 2 until no dependency edges are found.*

*4. If any marked node is a call vertex then calculate its influence using* ***Method Influence(callvertex)****.*

*5. Count marked nodes and calculate Influence using expression (1).*

*6. Stop.*

}

## VI. INFLUENCE OF A METHOD

The result computed by a method of a program affects the other methods and statements. A method may influence one or more methods and other statements of the program. If the influence of the method is more, then method is more critical. We have designed a program metric called Influence of a method for object-oriented programs. The influence of a method is defined by the number of other statements and other methods of the given program, which uses the results computed by the method directly or indirectly.

If other methods are called by the given method for which we want to find the influence, then the overall influence of the method will be influence of the method itself and the influence of other called methods. We first represent the input program in ESDG as intermediate representation and after that we apply our proposed algorithm on resulting ESDG. Then we count the number of nodes influenced from that method's formal parameter out nodes as well as other called method's formal out parameters and we count the total no nodes in graph.

• The influence of a method expressed as a percentage is given by:

$$\frac{\text{Total number of nodes influenced}}{\text{Total number of nodes in the graph}} \times 100$$

***Algorithm***

***Input****: A program and name of the method of that program.*

***Output****: Influence of the method.*

- *Method Influence(call vertex){*

*1. Construct ESDG of the program.*

*2. For the method entry vertex of the method traverse all edges and mark them visited.*

*3. For each visited node traverse through it's all edges marking it's corresponding node as visited and if it is not a call-vertex node then mark it as influenced if not marked already.*

*4. Check each visited node and if it is a call vertex, traverse through it's call edge and*:

*(a) If next node is polymorphic call vertex then traverse through each polymorphic edge and in*

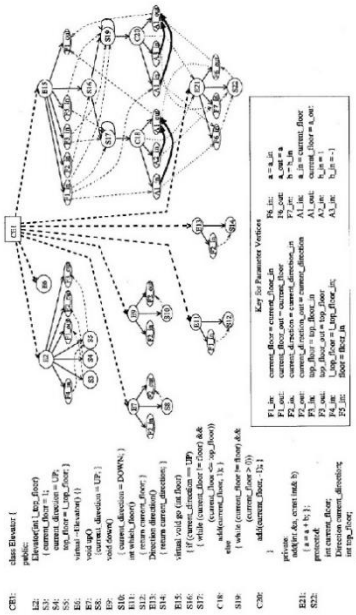

Figure 5.1: A C++ Elevator class and it's System Dependence Graph

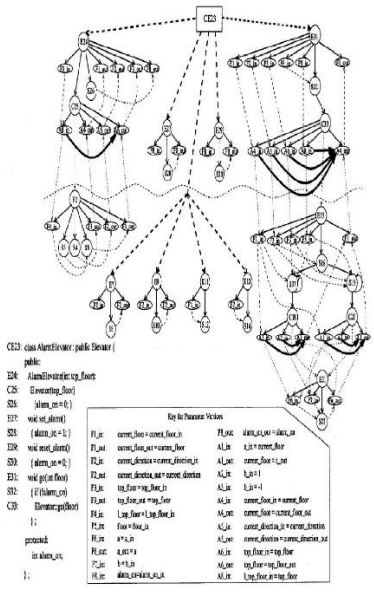

Figure 5.2: A C++ AlarmElevator class and its System Dependence Graph

## VII. EXPERIMENTAL RESULTS

We have taken test cases based on operational profile of the case study in each test suite. In the traditional testing method, we seeded bugs in each class in random fashion and tested the first copy of the case study using both structural and functional testing method loges . The numbers of seeded bugs for each class are selected using the operation profile of the case study. For prioritized testing method, we tested second copy of the case study with the same number of test cases and with the same testing methodologies as in first copy but, the number of test cases for each class are taken according to it's influence. In this case, we seeded the bugs in each method of the class according to it's influence. Hence, in the prioritized testing method the elements with higher influence are tested with more number of test cases. From the above table it is clear that as we gave more efforts in testing the more influenced elements we caught some more bugs. Although the number of extra caught bugs in each sample program were not many in number but the number of failures of the programs were reduced greatly. This shows that if we reduce the number of bugs from more critical elements by testing them more exhaustively the rate of failure is reduced. In each phase of software development cycle we can use the results and can give extra efforts to develop the more critical elements.

## VIII. CONCLUSION

We have purposed a program metric which called the influence of program elements. The influence shows that which elements affect more than others in a program. So the elements with higher influence are more critical and presence of bugs in them will increase the probability of failure of software. So, the purposed metrics greatly help in finding out the more critical elements and guides to take utmost care in developing the elements with higher influence during software development cycle. This also suggests that elements with least priority can be tested with least number of test cases rather

than giving similar efforts as more critical elements and hence saving the very important time for testing the more critical elements.

1. It is based on static analysis of a program.
2. Useful in test case design and test case prioritization.
3. Useful to characterize the influence of various components of the program. So one can have more reliable components to be tested thoroughly.

## IX. REFERENCES

[1]. Horwitz S., Reps T., and Binkley D. Inter-procedural slicing using dependence graphs. ACM Transactions on Programing Languagees and Systems 12, 1(1990), 26-61.

[2]. Zhang X., Gupta R., and Zhang Y. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In International conference of Software Engineering(2004).

[3]. Agrawal H., DeMillo R, A., and Spafford E. H. Debugging with dynamic slicing and backtracking. Software Practice and Experience 23, 6(1993), 589-616.

[4]. Dhamdhare D.M., Gururaja K., and Ganu P. G.A compact education history for dynamic slicing.Information Processing Letters 85(2003), 145-152.

[5]. Korel B., and Rilling J. Dynamic Program Slicing Methods. Information and Software Technology 40(1998), 155-163.

[6]. Xu B., Qian J., Zhang X., Wu Z., and Chen L. A Brief Survey of program slicing. ACM SIGSOFT Software Engineering Notes 30, 2(2005), 1-36.

[7]. WeiserM. Programmers use slices when debugging. Communication of ACM25, 7(1982), 446-452.

[8]. Ball T. The Use of Control Flow and Control Dependence in Software Tools.PhD thesis, Computer Science Department, University of Wisconsin-Madison, 1993.

[9]. Song Y., and Huynh D. Forward Dynamic Object-Oriented Slicing. Application Specific Systems and Software Engineering and Technology(ASSET'99). IEEE CS Press, 1999.

[10]. Ferrante J., Ottenstein K., and Warren J. The program dependence graph and it's use in optimization. ACM Transactions on Programming Languages and Systems 9, 3(1987), 319-349.

[11].http://linuxgazette.net/100/vinayak.html