Mathematical Statistician and Engineering Applications
ISSN: 2094-0343
2326-9865

Putting in Order the Components of the Program Based on the Test Requirements

Sachin Sharma[1], Digvijay Singh[2], Rahul Bhatt[3]

[1, 2, 3] Assistant Professor, Computer Science & Engineering, School of Computer Science & Engineering, Dev Bhoomi Uttarakhand University, Chakrata Road, Manduwala, Naugaon, Uttarakhand 248007

[1]socse.sachin@dbuu.ac.in, [2]socse.digvijaysingh@dbuu.ac.in, [3]socse.rahul@dbuu.ac.in

Article Info
Page Number: 4123-4130
Publication Issue:
Vol. 71 No. 4 (2022)

Article History
Article Received: 25 March 2022
Revised: 30 April 2022
Accepted: 15 June 2022
Publication: 19 August 2022

ABSTRACT
The purpose of the testing process is to identify any problems that may exist inside a software product. However, even after successfully completing the testing stage for the majority of actual systems, it is difficult to guarantee that the program is error-free. This is because it is impossible to test every conceivable combination of inputs and outputs. This is because the bulk of software programs have a very broad input data area to choose from. Testing the program in every potential configuration that the input data may take is not a practical option. The relevance of the testing procedure should not be minimized, even taking into account the fact that it is constrained by the actual world. It is essential to keep in mind that testing will, in fact, uncover a great deal of fault in a software application. Testing is thus an effective way for reducing system defects and increasing users' trust in a constructed system. Even after considerable testing has been performed on software, it is not uncommon for a few defects to remain. In most cases, these residual errors are scattered throughout the code. It has been observed that problems in some components of a program might result in failures that are both more common and more severe than those caused by problems in other components of the program. Therefore, the statements, methods, and classes that make up an object-oriented program should be able to be organized in accordance with the probability that they may result in mistakes. After the individual components of the program have been organized, the time spent testing may be divided up in such a way that the components with the highest failure rate get greater attention. Taking use of a program's intermediate graph representation is what this approach is all about. When estimating the impact of a class, a forward slice of the graph is often used. Coding, debugging, the creation and management of test cases, and a host of other activities are examples of applications for our proposed program measure.

Keywords: Prioritization of Program Elements, Slicing, Intermediate Representation, Program Testing, and Object-Oriented Programming are some of the key terms that are discussed in this article.

1. INTRODUCTION

Many different strategies for prioritizing and choosing test cases and affecting nodes have been developed, and each of these strategies has been experimentally explored. Examine the [1][2] most recent information pertaining to these fields. According to the results, the majority of the currently available techniques are applicable. Requirements for either the structural or functional coverage of the source code that is is executed when the test cases are

being performed. In this particular respect, our thoughts are distinct from those presented in the aforementioned body of work.

Even while testing software is a costly process in and of itself, the cost of releasing software without testing it might be substantially greater. This is particularly true for systems that are concerned with the safety of humans. Software testing may be thought of as any activity with the goal of evaluating a feature or capability of a program or system and determining whether or not it accomplishes the intended goals. In spite of the fact that understanding software principles is vital to the quality of software testing and that software principles are routinely employed by programmers and testers, software testing is still regarded as n art owing to a lack of knowledge of software principles. The complexity of the software being tested is at the heart of the issue; we are unable to do exhaustive testing on a program with just a moderate level of complexity. There are many facets to testing, and debugging is only one of them. There are a few different purposes that testing may serve, including dependability, quality assurance, verification, and validation. Testing may also be used as a stand-in for other types of metrics. Testing for correctness and testing for reliability are two important subfields within testing. Testing software requires making compromises in terms of cost, duration, and overall quality.

When errors are discovered and repaired early on in the software life cycle, a lower total cost is incurred as a result of these actions. When errors are discovered and repaired early on in the software life cycle, a lower total cost is incurred as a result of these actions. Software applications are becoming more integrated into the fabric of our everyday life. Software companies are under a huge amount of pressure to provide solutions that are incredibly dependable and have very little room for mistakes Testing is often performed on several levels for software products in order to uncover any and all bugs. the application running on the machine. It is difficult to guarantee that a software product is devoid of errors for the vast majority of real systems, even after the testing procedure has been completed successfully and passed with flying colors. This issue is brought about by the fact that the majority of software products have a relatively broad input data domain to work with.

In addition, any endeavor to produce a software product is subject to limitations placed on both its timeline and its financial resources. As a direct consequence of this, it is not feasible to do exhaustive testing on a software product by utilizing each and every conceivable value for the input data. At the present, testing consumes an average of fifty percent of both the overall cost and the amount of time that is spent developing [11].

As a result, it is quite doubtful that the amount of work put into testing will be raised any more. Traditional testing procedures are used to do in-depth evaluations on each individual component of the software product. This results in a bug population that is spread out evenly across the software product. However, the existence of flaws in some components results in failures that are more severe and more frequent than those in other components. For instance, if a statement generates essential data that is helpful for a great number of other statements, then a mistake in this statement would have an effect on a great number of other claims. As a result, our objective is to determine which aspects of a program are the most important so that we may do more thorough testing on those aspects. The degree to which an element is important and threatening is what we mean when we refer to it as having an effect. We suggested a metric that might be used to quantify the effect of a technique in addition to the influence of a statement. We are able to compute a class's level of impact with the assistance of these two measures. The steps of the software development life cycle that include planning, writing, testing, and maintaining the program may all benefit from the

characterization of the code. When it comes to the representation of intermediate code, we make use of the Extended System Dependent Graph. As a result, there is a slim chance that the amount of testing that is done might be increased even higher. Because testing is a sample, it is always necessary to make judgments on what to test and what not to test, as well as whether more or less of anything should be done. The majority of systematic testing approaches, such as white box testing and black box testing techniques such as equivalence partitioning, boundary value analysis, and cause-effect graphing, all have the same fundamental flaw: they produce an excessively large number of test cases. [11]

2. MOTIVATION FOR OUR WORK

Because computers and software are so regularly used in essential applications, even the smallest error might have very negative repercussions. It's possible that bugs may cause enormous losses. Errors in critical computer systems have resulted in the destruction of airplanes, the failure of the systems that operate the space shuttle, and the suspension of stock market transactions. A bug may kill. Bugs have been known to trigger catastrophes. In a culture in which everything is computerized, the reliability and quality of software are of the utmost importance. This may be accomplished only via the administration of exhaustive tests.

3. OBJECTIVE OF OUR WORK

There is a wide range in the degree to which different aspects of a program contribute to the overall dependability of the program. Therefore, the effect of the various components must be explained, and the components with the highest degree of trustworthiness must be subjected to comprehensive testing. to locate previously unknown flaws by using the standards as a guide. Before distributing the product or making it available to the public, check to see that it is free of any problems. "Your Satisfaction Is 100% Guaranteed."

The fundamental objective of our study is to develop efficient algorithms for analyzing the ways in which a statement, a method, and a class influence the behavior of an object-oriented program. Finding and fixing software bugs as early as possible in the process of developing software is one of the primary focuses of our work. This is done to ensure that the product in question does not experience frequent or severe failures. Our work is focused on locating and isolating software flaws at the earliest feasible stages of the software development cycle in order to reduce the frequency and severity of software failures. This is the overarching aim of our efforts. Our goal is to cut down on the number of times a system fails tests while maintaining the existing testing budget. There are two components of this that are included in the test plan. (1) The most important aspects of the program should be checked for functionality first. (2) Perform exhaustive testing on the areas of the code where the existence of even a single bug raises the probability of an unsuccessful outcome. The first one may be figured out by having a look at the function's visibility, the use frequency, and the possible failure costs. Regarding the second one, we have devised a method to locate the essential components of the source code [2], which uses an algorithm.

4. Work That Is Related

[14, 15] are only two of the many publications that have been published on the topic of prioritizing test cases. However, not much has been published about the work that was done to prioritize the code in order to determine which aspects of the program are the most significant.

Very little study has been done on the subject of enhancing testing prior to the construction of test cases. One of these areas of study is the creation of software with testability in mind. This study makes an effort to provide advice on how to design software that will be simple to test, with the goal of lowering the cost of testing.

In the work that is being shown here, the preconditions, postconditions, and assertions for each module are determined upon at the design phase of the development life cycle. The third component of the pretesting process is the prioritization of the code to be tested. I suggested a technique for prioritization calculation that ranks the essential components of the code in order of importance and draws attention to those components so that code coverage may be rapidly improved. The term "code coverage" refers to a measure that indicates what percentage of an application's source code gets executed whenever the application's unit tests are executed. In principle, the functional quality of the code will be better if the proportion of lines of code that are being tested is greater. However, in fact, having code coverage of one hundred percent does not ensure that a program is bug-free. The number of other objects in the given preprograms employing that object either directly or indirectly is taken into account when calculating an object's level of influence in that prog program direction in which other objects in a program are directed to execute is sometimes determined by the value that is returned by the method of an object. As a consequence of this, the impact of an item is computed based on the number of other objects in the given preprogram are both control- and data-dependent on that object, either directly or indirectly. This may be done in a variety of ways. After doing an examination of the source code, the first step in the process is to create an intermediate representation known as a control dependency graph. After that, we put the software together using the data set that was provided. In Section 4.1, we will discuss the technique that we have suggested. This algorithm can determine the influence value of any given object as well as extract the dynamic slice from an object at any time in the execution process. Our work includes testing on a prioritized basis, with the goal of ensuring that the quality of the software produced by the testing process is commensurate with the cost of the testing. In light of this, the focus of this section will be squarely on the research findings that were published in the context of prioritization approaches, during the process of test case selection in test suites, and prior to the development of test cases. [2]

5. CONTROL FLOW

The control flow graph (CFG) is a representation of a program that may be utilized as a step in many different optimization code transformations. These transformations include the elimination of similar subexpressions, the propagation of copies, and the transfer of loop invariant code.

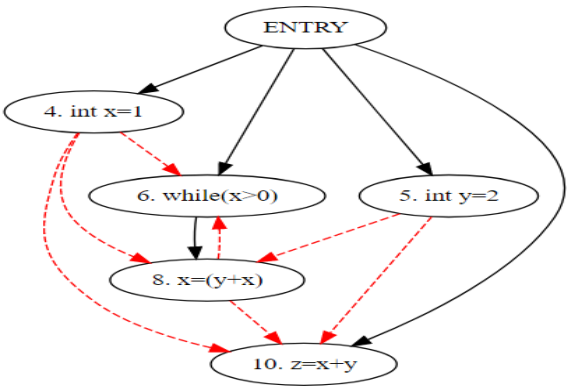

Figure (1): Program Control Dependence Graph

6. PROFILING

When we profile a program, we are able to learn where that program spent its time and which functions it called when those functions were operational. These statistics have the potential to show the parts of the program that are running slower than expected and might perhaps benefit from having their code rewritten in order to speed up execution. In addition, it may show you whether features are being utilized much more or significantly less often than you had planned. It's possible that you'll be able to see some issues that you would have missed otherwise.

7. THE SUGGESTED PROCEDURES

Ranking of program components and their priorities. In this part, we will provide our methodology for the ordering of program components in accordance to the level of testing that should be performed on each of those elements. To begin, we will present an outline of our methodology. After that, we will discuss the methods that we use to compute the effect of the statement, the influence of the method, and the influence of the class, in that order. Classes in object-oriented software are made up of code in their entirety. It is a given that variables and methods are present in each and every class. The influence wielded by a class is the sum total of the effects wielded by all of its individual components. As a consequence of this, we evaluate the consequences of each statement, and if any of those statements call a method, we also evaluate the consequences of that method. Our methodology disregards the values of variables and relies instead on a static examination of the source code. As a consequence of this, it has difficulty managing loops and calling recursive functions. The effect of class is the same as the aggregate of the effects of all relevant assertions and methods. The impact that a class has is decided using this strategy in a static manner. First, we go through how to figure out the effect of a comment, then we talk about the influence of a method, and finally, we talk about the influence of a class.It is possible for the results of one statement in a program to be contingent on the results of another statement. If the impact is greater, then the remark should be taken more seriously. The effect of a given statement is determined by the number of other statements in the provided program that make use of the variable, either directly or indirectly. We provide a measure for calculating influence in light of the fact that there is no call vertex. If a statement is identified as a vertex, its influence will be calculated separately using the method metric's impact. This influence will then be added to the other statements' influences in order to get the desired statement's total influence. The proportion of the statement's effect is determined by taking into account the following factors:

The sum of the nodes that have been designated as affected is more than or equal to 100.

The total amount of nodes present in the network

Algorithm

Input: the statement as well as the program code.

The output is the influence of the statement that was provided.

1. Create the edge of the program in a static manner using the statement function of the statement.
2. Have the for statement go along all of its dependence edges, and then mark them.

3. Repeat step 2 for each of the indicated nodes until there are no more dependence edges detected.
4. Determine the effect of any indicated node by using the method influence function if that node is a call vertex (call vertex).
5. Count the nodes that have been tagged and determine the influence by using an expression (1).
6. Stop.

}

8. Effects of utilizing a strategy

The outcome of the computation performed by one method in a program has an impact on the other methods and statements. It's possible for a single method to have an effect on several other methods and statements across the program. The significance of the approach increases in proportion to the degree to which it is able to exert its impact. We have developed a programming statistic for object-oriented programs that is termed the impact of a method.

The number of other statements and methods in a particular program that utilize the results calculated by the method either directly or indirectly is one way to measure the method's level of impact inside the program.

If the supplied method which we want to detect the effect calls other methods, then the overall influence of the method will be the combination of the influence of the method itself plus the impact of any other methods that are called by the method. The following factors contribute to the effect that a technique has when represented as a percentage:

The total number of nodes that were impacted by _____ was less than 100. Number of nodes comprising the whole graph as a whole

Algorithm

A program as well as the name of a method inside that program should be input.

The effect that the procedure has is the output.

Method influence(call vertex){

1. Develop an edge for the program.

2. Beginning at the method's entrance vertex, go along all of the method's edges and mark those edges as having been visited.

3. For each node that has been visited, go over all of its edges and label the node that corresponds to it as visited. If the node in question is not a call-vertex node, you should mark it as influenced if you haven't done so previously.

4. Check each previously visited node, and if it is a call vertex, traverse through its call edge. If the next node is a polymorphic call vertex, then traverse through each polymorphic edge and insert the corresponding node in a queue q. 4.a. If the next node is a polymorphic call vertex, then traverse through each polymorphic edge and insert the corresponding node in a queue q. 4.b. If

(b) if not, place the node at the q position.

5. Remove any nodes that are in q. Identify the node that is being impacted, then repeat steps 2–4 for that node.

6. Continue with step 5 until the contents of q are gone.

7. For each node that has been tagged as affected, traverse all of the edges connected to it, and label each of those connections as influenced if they have not previously been marked.

8. Determine the method's impact by using an equation to calculate (2).

9. Stop.

}

The aggregate of the influences exerted by all of the other components of the provided program that make use, either directly or indirectly, of the class's output is what is referred to as the influence of the class. We make a tally of the number of nodes that are impacted. The metamethod influence vertex) metric will be used to determine the effect of nodes that include function calls, whilst the statement influence (statement) metric will be used to calculate the influence of any other statements.

The following are examples of how classes exert their influence:

The total number of nodes that were impacted by _____ was less than 100.

Number of nodes comprising the whole graph as a whole

Algorithm

Input: a sample of the program, as well as the class's name.

Product: the impact that the class had.

Classinfluence(classname) \s{

1. Use a static construction method to build the esdg of the program.

2. Beginning from the class entrance vertex, go to each individual member of the class, and mark them as having been visited.

3. For each node that has been visited, go over all of its edges and label the node that corresponds to it as visited. If the node in question is not a call-vertex node, you should mark it as influenced if you haven't done so previously.

4. Determine whether or not each visited node is a call vertex, and if it is, determines its effect by using the metamethod influence ion (call vertex For each node that has been tagged as being affected, traverse all of the edges, and mark each one as being influenced if it has not already been marked.

6. Determine the amount of influence possessed by the specified class by using the phrase (3).

7. Stop.

}

Experimental Studies

Total number of nodes influenced	Total nodes in program	% of influence
96	134	71.64
22	134	16.41
8	134	5.9
7	134	5.2
1	134	.7

- We have identified the node with the greatest amount of influence, and because this node has an effect on the highest node, we are required to test it first.
- We need to determine the statement, method, and class that is impacted the most, and then test appropriately.

Conclusion

One of the program metrics that we established is based on the effect of different program aspects. The influence allows one to determine which aspects of the program are considerably more effected most affected. Therefore, the elements that have more influence are more significant, and the presence of these factors will raise the risk that the program will fail to function properly. As a result, the planned metrics are a huge help in identifying the most critical features, and they urge us to exercise extra care while developing the components that have the most effect on the software development cycle. This demonstrates that testing the components that are less significant may be completed with a smaller number of test cases than testing the parts that are more important, so sparing the essential time for testing the elements that are more important. A program is analyzed statically in order to do this. This is helpful in the following ways:

- It may be used to construct test cases and prioritize them.
- It can be used to characterize the effect that different parts of the program have. As a direct consequence of this, we now have more dependable components that need to be rigorously verified.

REFERENCES

1. Prioritization of Program Elements Based on Their Testing Requirements , Computer Science and Engineering , Kanhaiya Lal Kumawat, National Institute of Technology Rourkela(2009)
2. Reliability Improvement Based on Prioritization of Source Code , Mitrabinda Ray and Durga Prasad Mohapatra , Department of Computer Science and Engineering , National Institute of Technology Rourkela(2010)
3. Danjun Zhu, Gangtian Liu, "Deep Neural Network Model-Assisted Reconstruction and Optimization of Chinese Characters in Product Packaging Graphic Patterns and Visual Styling Design", Scientific Programming, vol. 2022, Article ID 1219802, 12 pages, 2022. https://doi.org/10.1155/2022/1219802