A Novel Technique for Call Graph Reduction for Bug Localization

Prabhdeep Singh
Student (ME), Department of CSE, Thapar University, Patiala, India

Shalini Batra
Assistant Professor, Department of CSE, Thapar University, Patiala, India

ABSTRACT
In today's era, software industries are competing for software quality which depends upon the sound software testing phase. To deliver the quality product the most challenging job for software industry is to localize bugs automatically and fix them before release. One of the techniques for automated bug localization is usage of call graph. Since size of the call graph generated is quite large, various call reduction approaches have been proposed. In this paper a novel approach for call graph reduction has been proposed where the size of the call graph is reduced without changing the basic structure and no major loss of the information is incurred. The output generated using the proposed methodology shows promising results.

1. INTRODUCTION
Software is rarely free from defects and debugging is the process of identifying the root cause of an error and correcting it. Manual debugging can be extremely expensive and localizing defects is the most time consuming and difficult activity in this context.

Numerous software testing techniques are applied to maintain quality of large software systems [2,10,11]. Since localization of bugs is the most time-consuming part of debugging, automated methods for bug localizations are required.

Various techniques have been developed for locating bugs[15]. One direction of research is static analysis, where properties of the source code or the version history are analyzed. Another direction is dynamic analysis, which requires the execution of the program [11] .

1.1 Call Graph
A call graph is a binary relation over selected entities in a program, such as methods, classes, subsystem, modules, files, etc., which represents invocations between those entiti]. Call graphs are either static or dynamic. A static call graph can be obtained from the source code. It represents all methods of a program as nodes and et al possible method invocations as edges. A dynamic call graph is the invocation relation that represents a specific set of runtime executions of a program. Dynamic call graph extraction is a typical application of dynamic analysis to aid compiler optimization, performance analysis, program understanding, etc [12]. Dynamic call graphs represent an execution of a particular program and reflect the actual invocation structure of the execution. Without any further treatment, a call graph is a rooted ordered tree. The main method of a program usually is the root, and the methods invoked directly are its children[13].

1.2 Call-Graph Representations
A call graph is a directed graph whose nodes represent the functions of program and directed edges symbolize function calls. Nodes can represent either one of the following two types of functions:

Local functions, implemented by the program designer.

External functions: system and library calls.

Local functions are the most frequently occurring functions in any program. They are written by the programmer of the binary executable. External functions, as system and library calls, are stored in a library as part of an operating system. Contrary to local functions, external functions never invoke local functions. Call graphs are formally defined as follows:

Definition (Call Graph): A call graph is a directed graph G with vertex set V=V(G), representing the functions, and edge set E=E(G), where $E(G) \subseteq V(G) \times V(G)$, in correspondence with the function calls[4].

2. LITERATURE REVIEW
Call graph are representations of program executions. Raw call graphs typically become much too large for graph-mining algorithms, as program might be executed for a long period and frequently call other parts of the program, which adds information to the graph. Therefore, it is essential to compress the graphs by a process called reduction[1,7,8]. It is usually done by a lossy compression technique. This involves the trade-off between keeping as much information as possible and a strong compression. The researchers have proposed a number of different call-graph representations[3,4], standing for different degrees of reduction and different types and amounts of information encoded in the graphs.

There are two approaches of reducing software call graphs

Total Reduction (Liu et al.[14])

Zero-one-many reduction (DiFatta et al.[15])

Total reduction is proposed by Liu et al In totally reduced graphs, every function is represented by a node. A direct edge is connected with the corresponding nodes when one function has called another function. Total reduction technique shortens the size of source call graph [8]. This technique has been introduced by Liu et al. In this technique, every method occurs just once within the graph. The major shortcoming of this technique is that it changes the structure of the graph. On the other side, much information about the program execution is lost, e.g., frequencies of the execution of methods and information on different structural patterns within the graphs. So it is very difficult to retrieve required information from this reduced graph.

The below fig 1 is shown the call graph of source graph

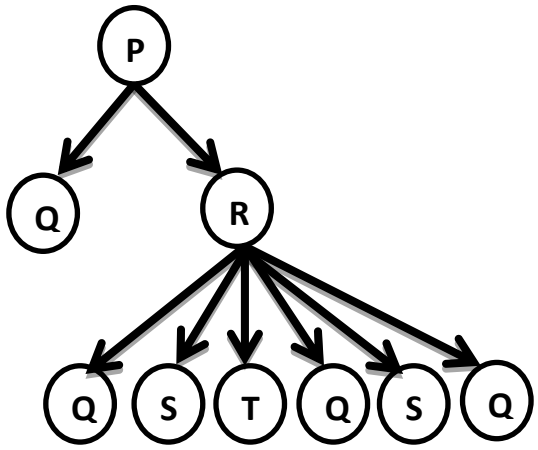

Figure 1

Figure 1 is derived from the source code, called source call graph[9]. As seen in figure the graph has 3 levels where P is root node and Q and R are its children. In next level R is considered as Q, S and T's parent. Q is called 3 times so 3 direct edges are connected from R to Q and S is called 2 times so 2 edges are connected with R to S. Function T is called single time so edge is connected from R to T singly. After applying Liu et al. approach reduced graph is shown in Figure 2. Using this technique the 2nd level children are reduced from two to one. In source call graph P is connected with Q at 2nd level but after reducing it is directly connected with Q in 3rd level of graph. This reduced call graph doesn't show the call frequency of nodes. Its structure has also been changed.

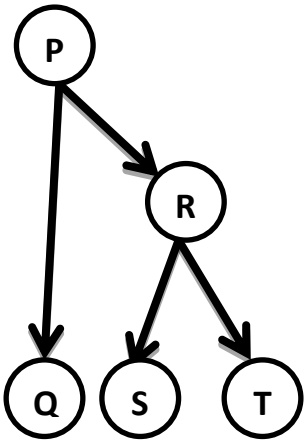

Figure 2

The other approach given by DiFatta et al. covers the drawback of Liu et al. approach as it does not change the structure but the reduction is not properly done. The improper reduction increases its complexity and it is difficult to find frequent sub structure from graph. Reduced graph can provide near information about call frequency but exact information is not known. Call frequencies are important for detecting certain groups of bugs.

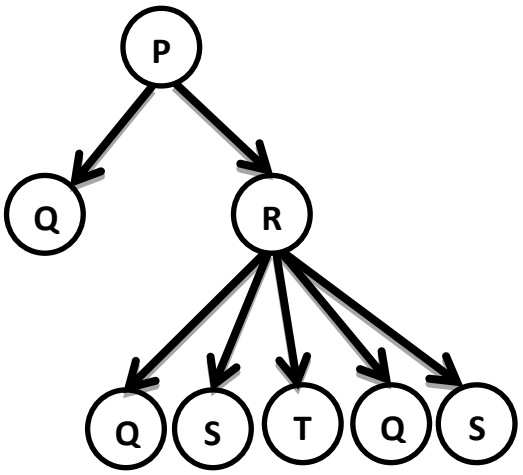

Figure 3

In the figure 3 reduced callgraph by approach of DiFatta et al. has been shown. Frequency of nodes has been changed but the structure of call graph remains same. In this approach of reduced call graph node is shown two times if it is shown more than two times in source code. The exact frequency of nodes is not known by using this approach.

3. PROPOSED APPROACH

To overcome the drawbacks of both techniques a new approach is proposed. In this approach the reduced call graph shows the call frequency of each node without changing the structure of source call graph. First of all, all functions of source code are labeled so that it can easily be interpreted. Then a call graph is made using these labeled functions. The main task is to save the node into computer memory with its parent's information which is not possible with adjacency list or adjacency matrix. Therefore the parent of each child is stored in the matrix. Rows represent the levels of call graph as 1st row represent 1st level's nodes, 2nd row represent 2nd level's nodes and so on. Every node also contains the information about its parent. The proposed algorithm to save the node with its parent's information in the computer memory and efficiently reduce the graph is as follows:

Algorithm: Reducing call graph

Input: Matrix of structure containing children, label, parent

Output: Reduced call graph

Set j=Getstr[100][]

foreach aa←1 to 10 **do**

Set j[aa - 1] = **Getstr**[200]

end for

Set count= **GetArray**(level)

foreach i←0 to levels-1 **do**

```
    print "Enter no. of children at level 0"
    Input(children)
    count[i]=children
    foreach x←0 to children -1 do
        print "Enter the label of (x) children"
        j[i][x].label = Input(label)
        print "Enter the parent of (x) children"
        j[i][x].parent = Input(parent)
        j[i][x].count = 1
    end for
end for
foreach k←levels down to 1 do
    foreach l←0 to count[levels-1] -1 do
        foreach m←l+1 to count[levels -1] -1 do
            if j[k-1][l].label=j[k-1][m].label AND j[k-1][l].parent=j[k-1][m].parent AND j[k-1][l].parent ! = -1 then
                j[k-1][m].parent = -1
                j[k-1][l].count++
            end if
        end for
    end for
end for
```

Algo1: Reducing Call Graph

4. IMPLEMENTATION DETAILS

$$\begin{bmatrix} A_{-1} & 0 & 0 & 0 & 0 & 0 \\ B_0 & C_0 & 0 & 0 & 0 & 0 \\ D_0 & C_1 & 0 & 0 & 0 & 0 \\ D_0 & D_1 & D_1 & D_1 & D_1 & E_1 \end{bmatrix}$$

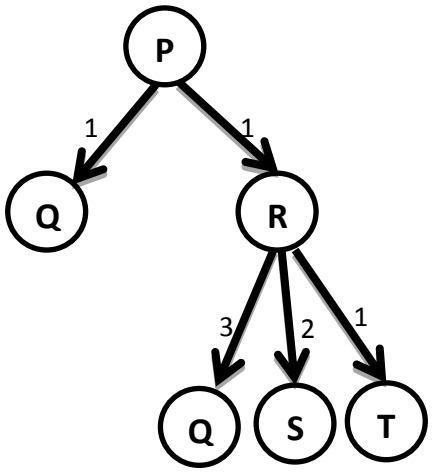

Figure 4

The above algorithm has three inputs from the source graph as children, label and parent at each level and stored in the matrix which is implemented from line number 6 to 16. Similarly the reduction of graph is implemented from line no 17 to 25 where it merges the children of same label of same parent in each level.

5. CONCLUSION

New static and dynamic approaches for bug localization have been developed, the diffusion into other disciplines has proceeded at a rapid pace, and knowledge of all aspects of the field has grown even more profound. At the same time, one of the most striking trends in graph theory is constantly increasing emphasis on the interdisciplinary nature of the field. Graph mining today is basic research tool in all areas of engineering, medicine, and the sciences. The bug localization techniques based on graph mining are successfully applied in a wide range of practical problems arising in software industry.

In this paper a novel algorithm for call graph reduction has been proposed In order to use the respective call graphs for bug localization, the developed technique stores the parent information in the matrix and reduced at each level drastically. Information about each node is retained by using the call frequency by annotating each edge with a numerical weight. Similarly the algorithm used to reduced call graph has various advantages over traditional techniques. It takes various parameters for consideration such as information of nodes, basic structure of graphs and call frequency. Here the detailed study of call graph reduction in graph mining made the study of various other techniques in bug localization very easy

6. FUTURE SCOPE

The proposed algorithm works only when there are same types of nodes at a particular level in a call graph.

In future this work can be extended to multiple levels of call graph will make the graph mining algorithm efficiently.

Secondly the storage of graph can be upgraded with any new storage technique where it would require lesser storage space as well as lesser access time leading to further optimize reduction of call graph.

7. REFERENCES

[1] W. Sadiq, M. E. Orlowska, "Applying Graph Reduction Techniques for Identifying Structural Conflicts in Process Models Distributed Systems Technology", Proceeding of the 11th International Conference on Advanced Information Systems Engineering, 1999, pp.195–209.

[2] I. R. Katz, J. R. Anderson, "Debugging: an analysis of bug-location strategies", Human Computer Interaction, 1987, pp.351–399.

[3] R.Hastings,B.Joyce, "Purify: Fast detection of memory leaks and access errors", Proceedings of the Winter USENIX Conference, 1992, pp.125.

[4] O.Lhotak, "Comparing Call Graphs", Proceedings of the 7th ACM Sigplan-sigsoft Workshop on Program Analysis for Software Tools and Engineering, PASTE 07, San Diego, California, USA, 2007, pp. 37-42.

[5] L. Dietz, V. Dallmeier, A. Zeller, T. Scheffer "Localizing Bugs in Program Executions with Graphical Models", Advances in Neural Information Processing Systems 22,Proceedings of the Conference, Vancouver, Canada, 2009, pp. 468-477.

[6] W. Ren, R. Beard, E.Atkins, "Information consensus in multivehicle cooperative control" Control Systems, IEEE, Vol.27, No. 2, 2007, pp.71 -82.

[7] G. Shi, Weiwei "A Graph Reduction Approach to Symbolic Circuit Analysis" Proceeding of Design Automation Conference, 2007.

[8]S. K. Lukins, N. A. Kraft, L. H. Etzkorn, "Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation, "Proceedings of the 15th Working Conference on Reverse Engineering, 2008, pp.155-164.

[9] D. Binkley, "Source Code Analysis: A Road Map" Proceedings of the 29th International Conference on Software Engineering, 2007.

[10] Boris Beizer, "Software Testing Techniques", Van Nostrand Reinhold Co., 2nd Ed., 1990.

[11] L. Dietz, V. Dallmeier, A. Zeller, T. Scheffer "Localizing Bugs in Program Executions with Graphical Models", Advances in Neural Information Processing Systems 22,Proceedings of the Conference, Vancouver, Canada, 2009, pp. 468-477.

[12] B. Ryder, "Constructing the call graph of a program", Software Engineering, IEEE Transactions on,vol. SE-5, no. 3, 1979, pp. 216 – 226.

[13] X. Hu, T. Chiueh, K. G.Shin, "Large-scale malware indexing using function-call graphs", ACM Conference on Computer and Communications Security, E.AlShaer, S. Jha, A. FD.Keromytis,Eds,2009, pp. 611–620.

[14] C. Liu, X Yan, H Yu, J Han,., P.S. Yu, "Mining Behavior Graphs for \Backtrace" of Noncrashing Bugs". In: Proc. of the 5th Int. Conf. on Data Mining, 2005

[15] Di Fatta, G. Leue, S.Stegantova,"Discriminative Pattern Mining in Software Fault Detection". In: Proc. of the 3rd Int. Workshop on Software Quality Assurance,2006.