Abstract—Software quality is the major concern in modern industries. Software is never released bug free if tested extensively. The most challenging job in software industries is to locate bugs automatically and fix them before release. One of the dynamic techniques to automated bug localization is the usage of call graph. Call graph generated from source code is quite large which is usually complex. Various call graph reduction techniques exist which has some limitations and strengths. In this paper, a novel technique has been proposed, by which call graph is reduced without any loss of information and changing its basic structure. The resulted reduced call graph gives better structure with more information and earlier bug localization.

Keywords—Introduction, Bug localization, Call graph, Call graph reduction.

I. INTRODUCTION

Today’s software industries are competing for software quality which depends upon sound software testing phase. No software is released bug free if tested extensively. Bug localization is the process of identify set of statements that causes the failure of a program. Nowadays, software reliability is becoming a top concern in modern industries. Software bugs cost the U.S. economy an estimated 59.5 billion dollars annually, or approximately 0.6% of the GDP, according to a report from the National Institute of Standards and Technology (NIST). Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a computer program or art of determining the location of a bug in a program. It is hard to locate the bugs in complex programs. Bug localization can be performed by analysis of call graph. One analysis technique is Static analysis technique which requires large database or history of bugs for analysis. Another is Dynamic analysis technique which requires the execution of the program. [6]

i. Call Graph:

Call graph is a directed graph which represents the calling relationship between different methods of a program. Call graphs are program analysis result that helps to further analyses and for human understanding of invocations of different values between procedures. Call graph can be either Static or Dynamic. Static call graph describe every possible runs of a program. Static call graph can be generated from source code. Dynamic call graph describe the record of an execution of a program.

It considers only one run of a program. 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 [10].

ii. Call Graph Reduction:

Call graph generated from source code is quite large by which graph mining process becomes a difficult task. Mining process on raw call graphs has the difficulty of algorithm scaling. Therefore reduction algorithms are developed and applied first. Various call graph reduction techniques have been proposed which have some limitations and benefits. Call graph reduction means to reduce multiple numbers of edges in such a manner that there is no loss of important information. Compressing or reduction of call graph involves trade-off between loss of information and size of resulting graphs [4]. While reducing a call graph the important information is call frequency of all methods. 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 call graph by a process called reduction [5, 8, 9].

II. LITERATURE REVIEW

A Call-Graph is a useful tool in researching problems in source code. By seeing a visual reference of the order calls are executed, it allows the individual to trace back to a piece of code quicker. This allows them to test data output in various points faster by reducing the number of files that will have to be opened and searched to track this data manually. There are two reduction techniques that are to be considered:

Total Reduction (Liu. et al (2005)) [1].
Zero-one-many Reduction (Di Fatta et al (2006)) [3].

Total reduction technique is proposed by Liu et al. in 2005. Total reduction technique projects every node representing same method in the call graph to a single node. Total reduction shortens the size of the source call graph [8]. This reduction technique limits the size of the call graph to number of methods in a program. The major disadvantage of this technique is that it changes the structure of the source call graph totally. Lot of information regarding frequencies of execution of methods and information of different structural patterns within the graph is lost. So it is very difficult to retrieve the required information from this reduced graph. Here below the input source call graph created from source code in this research.
Figure 1 is derived from source code called source call graph. Above given source call graph has four levels where 'a' is root node and b, c and e are its children at 2nd level. At the 3rd level, b has three children and both e node has a sub tree with two nodes in which d and e is called two times so there is two directed edges from e to d and from e to c. f is called four times so there is four directed edges from d to f. After applying Liu et al. approach, in the reduced call graph, at the 2nd level instead of three there are two child remains left and only one subtree of e node exists and at the third level d is directly connected to f singly. The reduced call graph is shown below:

III. PROPOSED TECHNIQUE

To overcome the drawbacks of conventional approaches a new novel technique has been proposed. By this reduction technique, the reduced call graph provides better structure with more information. Initially, all the nodes in source call graph is assigned the frequency of 1 and all the methods are labelled as nodes which are connected to each other through edges while generating call graph. The source call graph has depth of 4 through which the graph is traced level by level. The The call graph generating algorithm is as follows:

Algorithm: Generating Call graph.
1. First of all remove comments from program.
2. Take name of function as A,B,C,D
3. Put the different functions in a string array and find their children.
4. Child at previous level = functions children [O]. Split('_'); node no.=2
5. Repeat for I from 1 to depth.
6. Let the children at previous level=0
7. Repeat for j=0 to the child at previous level.
8. Find the node such that n.children.contains(node no);
9. If (n node no!= parent),then, parent=n=node no.
Node.chil
Node.children= ";
Kmax=functions.children [(int) node.nodename = 97] split ('-').count();
Repeat for k=0 to kmax
Node.children=node.children+nodechildren.
Nodechildren++
childrenatnextlevel++
End loop
New tree. Add(node);
Nodeeno.=nodeeno.+1;
Child=child+1;
End loop
Child at previous level=childrenatnextlevel
End loop

After creating call graph from source code the main task is to reduce the call graph in top-down manner which provides the
bug localization at earlier stage. It considers the nodes at the same level to compare the node name at the same level for reduction. If the name of parent is same as child’s name then it creates recursion which is denoted by using self-loop arc and the frequency of parent node and child node is to be added. The proposed algorithm efficiently reduces the call graph as is follows:

Algorithm: Reduced call graph
1. Starting from root node towards its children
   a) Find the no of similar nodes and count the number of similar nodes which will be multiplication factor.
   b) For the first existence of similar node call Remove nodes(node, false, multiplication factor)
   c) For rest of occurrences call function Remove nodes(node, true, 0)

Algorithm Removenodes
Input: Node node, bool remove, int mulfac
Output: Remove node and it’s subtree if remove is true else multiply node and its subtree with mulfac
1. Child = node.children
2. While (! child.equals (""))
3. Children = child.split ("-")
4. tempchild = ""
5. Repeat for i from 0 to children.count ()
6. Find node n such that n.nodeno==children[i]
7. removenodes.Add(n)
8. Endloop
9. Endloop
10. Repeat for each node n1 in removenodes
11. if (remove) then
12. newtree.Remove(n)
13. Otherwise.

IV IMPLEMENTATION
The proposed technique has inputs of 15 nodes and 14 edges and produced 6 nodes and 5 edges. The proposed algorithm considers same node at same level to reduce call graph. The reduced call graph by using proposed technique is as:

V CONCLUSION
Fault localization for software system is becoming a challenging job for software organization. Various new static and dynamic approaches for bug localization have been developed. DiFatta use the frequent pattern mining algorithm to enhance fault localization for software system. This method is applicable to software system for which a large set of test cases as well as test oracle are available. To increase scalability, the question to which extent other program analysis and abstraction methods, such as program slicing, can be used in combination with this method. Liu use novel method to classify the structured traces of program executions using software behaviour graphs. It investigates the capability for computers to classify correct and incorrect executions based on observation of program behaviours. By this technique, it is not clear that whether the used method can be effective at tracing large software programs with the existence of multiple bugs in different program modules, how to further develop used software behaviour graph method to make the traces deeper with finer granularity.

VI FUTURE SCOPE
The proposed algorithm considers self-recursion or direct recursion. In future work, the indirect recursion concept can be discussed with some constraints and approximatively techniques. The proposed technique considers same nodes at the same level. In future work, while reducing call graph same nodes can be consider at multiple levels.

REFERENCES