Indian Journal of Science and Technology, *Vol 9(40), DOI: 10.17485/ijst/2016/v9i40/79423, October 2016*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Competent Dependence Graph Acclimatize as Intermediate Representation to Effectuate the Best Slicing Technique for Object Oriented Programs

Preeti Sikka[1*] and Kulwant Kaur[2]

[1]Punjab Technical University, Jalandhar, Punjab, India; preetisikka@gmail.com
[2]Apeejay Institute of Management Technical Campus, Punjab Technical University, Jalandhar, Punjab, India; kulwantkaur@apjimtc.org

Abstract

Objective: The main objective of this paper is to propose a Program Slicing Technique that includes more number of features than available in any other slicing technique. **Methods:** Numerous slicing techniques have been originated for enhancing the outcropping of Program Slicing. Simultaneously varied intermediate representation for representing the important features and dependencies of statements of a program on one another has been created for the easiness of creating the slicing technique. Unvaried part in varied techniques while concentrating on one good feature to be interpolated was overlooking some other important features. The flawed areas of the Available techniques have been found and based on the solution to overcome them, the technique is proposed that will consist of all important features with taking care of their flawed areas. **Findings:** Intermediate representation of a program accompanied by an algorithm is proposed to surpass the existing graphs by covering their flawed areas and by including more number of features than available in any other existing representation or technique. **Conclusion:** From the case studies it is concluded that the important features like Fault localization, Change Impact Analysis, Better Visualization etc can be achieved with a single technique. Added advantage is the complexity is reduced, accuracy and efficiency in terms of space and time is improved.

Keywords: Change Impact Analysis, Dependence Graph, Fault Localization, Program Slicing, Slicing Technique

1. Introduction

Diversified representations have been used to represent the dependencies among the statement of a program. Based on representation slice of a program is directly obtained generally by linear time backward walk from some point in the graph, visiting all its predecessors[1]. Starting from the Program Dependence graph various other graphs have been developed to obtain slice of a program till date by adding their special feature in the previous one or by overcoming the limitation of previous one[1]. While concentrating on one side, the other side of the coin has been ignored for e.g. dynamic slice has reduced the size of static slice but has increase in demand of memory space, and several work done for space efficiency has not met either precision or has increase in its runtime overhead etc.

The purpose of the paper is to create the best technique to slice a program with proper concern given to all important features like efficiency in space and time, precision etc. The added features like fault localization, change impact analysis, Good visualization are attached to it in same technique.

The better the representation is, the more ease it provides in creating the slice of a program. Following this methodology first an intermediate representation has been created a slicing technique is implemented that enhances the basic purpose of program slicing with numerous features all together in single technique and hence making the slicing technique competent.

*Author for correspondence

The paper specifies the existing graphs with their deficiencies in Section 2. The scope to overcome the deficiencies is mentioned together. While overcoming the existing graphs and techniques, the other important features that are consolidated in the representation and slicing technique are overviewed in Section 3.Taking a sample program, Section 4 shows the actual construction of graph and algorithm imbedded with the discussed features. Related work in brief is described in Section 5. Finally section 6 concludes the paper with the direction of continuation of work to be covered in future.

2. Comparative Study of Existing Dependence Graphs

The graphs available are summarized in a Table 1 with the scope mentioned that is achieved in this paper.

3. Our Approach

After analyzing the deficiencies of different graphs in section 2, their extended scope is achieved by upping the basic feature of program slicing that is to identify the set of statements in a program that may be affected by a given change in a program, with the following mentioned features step by step in single representation.

Features to be added are-

3.1 Space and Time Efficient

Criterion of slicing a program is taken as of dynamic slice i.e. statement number with the input value given at the time of getting the erroneous value. Its limitation of requirement of large amount of memory space is handled in the paper. Several ways are adopted to find out the number of edges and vertices that can be removed with-

Table 1. Existing Dependence Graphs with scope to extend

Sr. no.	Name of the Graph	Purpose	Extended scope after analyzing the deficiency
1.	Program Dependence Graph(PDG)[1]	Representation of Data Dependency and Control Dependency among the statements of a program	Addition of control edge at every node of PDG makes it restrictive. Input value is added in the slice criteria that limit the control edges by reducing the successor of the node in graph to one
2.	System Dependence Graph(SDG)[2]	It is extension of PDGto represent the inter-procedural programs	The Reachability algorithm of SDG was imprecise. Precision is added by ignoring only irrelevant statements and including only relevant statements. Also the Visualization is improved than of SDG by lessen down by the number of vertices using fractal value[3].
3.	Class Dependence Graph(CLDG)[4]	Representing the dependencies of distinct classes.	Other than representing the dependency of distinct classes, the proposed graph is capable to handle object oriented features like inheritance as well.
4.	Dynamic Dependence Graph(DDG)[5] and DODG[6]	DDGadds precision while computing dynamic slices. In addition to it DODG represents the dynamic dependency between instances for a particular execution of an object oriented program	Achieving Precision,DDG and DODG faces the problem of space especially in case of loop
5.	Weighted System dependence graph(WSDG)[7]	Itguides the developer about the locations that highly correlates to failure.	Better way is given in the paper to find the locations that correlates to failure by differentiating the statements in 'Must set' and 'May set' so as to not ignore any of the statement whose absence may affect the accuracy.

out affecting the precision. Also Object oriented features included here, have the capability to reduce the size and time for running or for testing.

3.2 Prioritizing

In the obtained slice all the statements are not equally responsible for the produced error[8]. Statements are distinguished into three priorities: Statements that are giving correct output only, Statements responsible for wrong outputs, in case of more than one output- statements that are giving one correct output and another wrong output. The set of statements producing errors for sure is termed as 'Must set', 'May set' comprise of statements that may or may not produce the error, priority 1 is given to the statements that are producing correct values for sure.

3.3 Precision

Précised slice care is taken to include only those statements that actually affect the slicing criterion for the given execution[9]. While creating slice care is being taken not to miss any statement that can affect the criteria and also not to include any statement that will not affect the criteria.

3.4 Fault Localization

Program Dependence graph helps in locating the fault[10]. Once the priorities are set and statements are known of their behavior, removal of statements with priority 1 will make the task of locating the fault easy and quick.

3.5 Change Impact Analysis

Other than finding the faulty statement, the proposed slicing technique is helpful in finding the statements which can be affected if any change is occurred to the program[11]

3.6 Good Visualization

Vertex near the criterion is connected to many vertices which may not be required, using the concept of fractal value[3] those vertices are filtered out resulting in the reduced readable size of visualized slice and for precision it can also be elaborated if required

4. Implementation

Steps discussed in Section 3 are here reformed as an algorithm with the help of sample program shown in Figure 1 which is being represented in a graphical form first shown in Figure 2. Where the circle represents the statements of a program and edges joining the vertices 'a' to 'b' represents the dependencies of vertex 'a' on vertex 'b'. Figure 3 and Figure 4 is the addition of 'Inheritance' feature in Figure 1 and Figure 2 respectively.

```
#include <iostream>
using namespace std;
1.  class base
    {
2.  protected:
3.  int side;
4.  public:
5.  void calc(int l,int b)
    {
6.  side=setside(l,b);
7.  float r=side/2;
8.  float ar_cr=3.14*r*r;
9.  float ar_sq=side*side;
10. cout<<"Maximum circle you can
    create of  "<<ar_cr<<"cm";
11. cout<<"\nMaximum square you can
    create of  "<<ar_sq<<"cm"; }
12. int setside(int l, int b)
    {
13. if(l<b)
14. return l;
15. else
16. return b;
    }
17. };   //End of Class
18. int main()
    {
19. base ob;
20. int len;
21. int bre;
22. cout<<"Enter the length and breadth of
    your land"<<endl;
23. cin>>len>>bre;
24. ob.calc(len,bre);
25. return 0;   }
```

Figure 1. Program to calculate the area of maximum square and maximum circle from the land whose length and breadth is given by the user.

```
#include <iostream>
using namespace std;
1   class base
    {
2   protected:
3     int side;
4   public:
5     void setside(int
    l, int b)
      {
6     if(l<b)
7     side=l;
8     else
9     side=b;
      }};
10  class derived :public base
    {
11    int r;
12  public:
13    void calc()
      {
14    r=side/2;
15    float ar_cr=3.14*r*r;
16    float ar_sq=side*side;
17    cout<<"Maximum circle
    you can create of
    "<<ar_cr<<"cm";
18    cout<<"\nMaximum
    square you can create of
    "<<ar_sq<<"cm;}};
19  int main()
    {
20  derived ob;
21  int len;
22  int bre;
23  cout<<"Enter the
    length and
    breadth of your
    land"<<endl;
24  cin>>len>>bre;
25  ob.setside(len,br
    e);
26  ob.calc();
27  return 0;
    }
```

Figure 2. Program with inheritance.

In Sequential statements the control will automatically flow to next statement so dependency is only Data Dependency, while in case of Selection statements control can be passed through any of the two cases so as to represent the flow of control an extra vertex has been taken where both the cases will reach. This extra vertex can be seen in Figure 2 after statement 16 and 18. In Iterative statements, an edge will be drawn from the last statement of the block to the first statement of the block to represent that and first to next statement after block. This way the Limitation of PDG of Restrictive nature is taken care of by removing the control edges.

In Figure 3 Calc is a call site node attached with the edges carrying actual in vertices and copying that to 5

as formal in vertices similarly carrying actual in vertices from 6 to 12 and taking formal out vertices from x to 6.

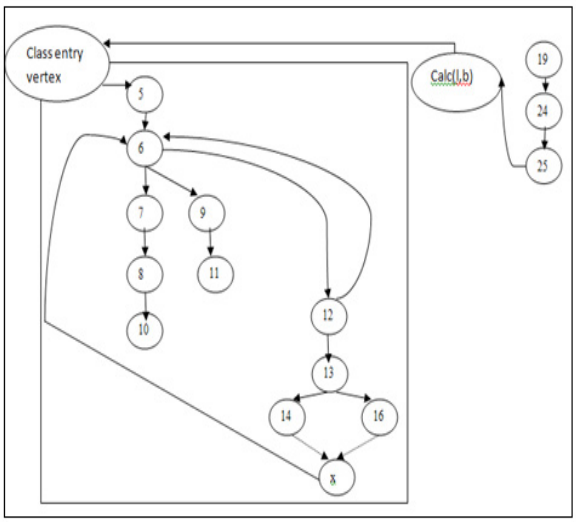

Figure 3. Dependence graph of Figure 1.

In Figure 4 again Calc is a call site node attached with the edges carrying actual in vertices and copying that to 13 that is the node in a derived class that demands a value of side from base class and base class is attached with a call site node 'setside' carrying actual in vertices len and bre and copying it to 5.

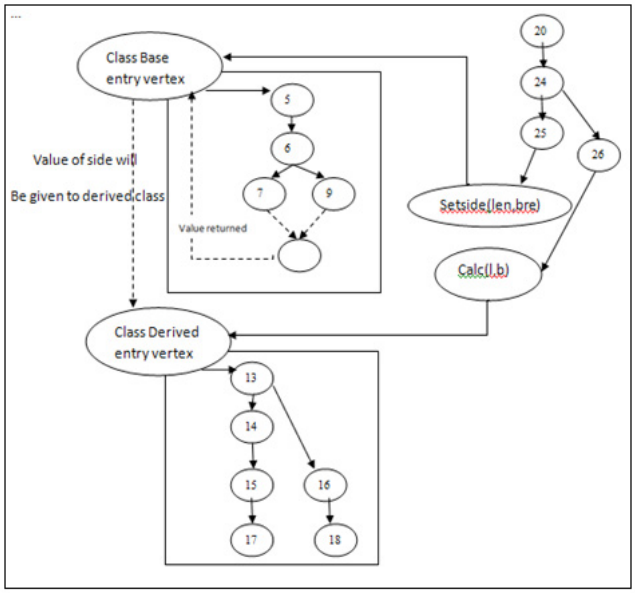

Figure 4. Dependence graph of Figure 2.

Inheritance is indicated by the class dependence edge which passes betweeen base class and derived class, inherited data members and methods can simply be computed by traversing up the class dependence edge and along the class membership/data member edges of base class[12].

Graphical representation is created by taking 'Main' as an initial statement. Representation of flow of data within and across is shown in previous work[13]. Taking that representation as step 1, the other steps of algorithm is mentioned here.

4.1 Algorithm of Graphical Representation

Step 1: Get the execution trace: Execute the program once and produce its execution trace (procedure wise) to know the dependencies among the statements of a program. Term the set of edges representing the dependencies as 'E'.

Step 2: Lessens down the number of vertices and Edges: Vertices are not added for the explanatory statements. For reduction in number of edges, copy the set of edges from 'E' to 'F' one by one. While inserting edge to 'F" checks the following conditions-

- If the same edge is already available (i.e. its successor and predecessor is same) then don't insert it again.
- In case of one time run of loop, u is dependent on v where v is coming later than u. then no edge between u and v is drawn. Example u: x=y+2; v: y=y*2; changed value of 'y' in 'v' won't affect 'x' in 'u' because of non repetition of loop.
- In case of input Values inside the loops, i.e while (i<=n){read(x)}…if(x%2==0)…. Else…}. Add that input variable i.e. 'x' here in dynamic slicing criteria so as to automatically ignore the statements coming in else part of the condition and
- If the edge is already created for one true block, don't create the same edge again i.e. all even values given to x will give same isomorphic graphs.
- In case of same code repeating more than once, edges will not be created again.

Step 3: Graphical Representation of Program: Bind the different processed together to create representation of complete program by putting an entry node start and connecting it to the starting node of every procedure graph created, similarly insert a unique exit node 'stop' connecting to the end and Join the different created PDG's using call site nodes where the edge is going from the calling

function's vertices to the entry vertices of the method/ procedure by call edges and actual vertices with formal vertices with parameter edges

The representation is shown by taking the sample program in Figure 1 who's PDG has been shown in Figure 2, Figure 3 and Figure 4 shows the same program with its PDG respectively after adding inheritance to it.

4.2 Algorithm after Creating Graphical Representation

Step 5: Slice generation: In case of errors take statement giving output as criteria node and in case of any change occurred to the program take the statement of change as criteria node. Follow the arrow from that criteria node in a backward direction and put all reachable vertices till the starting point in set of backward slice similarly going forward from the criterion node till the ending node and put all the reachable nodes in set of forward slice.

- In case of errors, required slice is backward slice set for'*S[output variable]*'. For example s[ar_cr]=are 8, 7, 6, 12, 13, 14,16, 5, 25, 24, 19. S[ar_sq]= 9,6, 12, 13, 14, 16, 5, 25, 24, 19. Add input value to the criteria to get reduced slice for example suppose ar_cr is coming as correct but ar_sq giving wrong value, the new criteria will be [ar_sq, l=4, b=3] and S[ar_sq,l,b]= 9, 6, 12, 13, 16, 5, 25, 24, 19.(14 is not included now)
- In case of any change occurred in a program there is a need to find and check the statements that are now affected, nodes of forward slice set will be the statement to be only analyzed.

Step 6: Precision: Create Must set as [ar_sq]-S[ar_cr] (node 9 in above example) that will contain the nodes that are surely producing the error as the nodes in S[ar_cr] are giving the correct outputs and hence cannot be erroneous. May set constitutes of other nodes that are not even a part of the set giving correct values and may or may not produce the error.

The two important steps to attain precision is –

- Not to ignore any relevant statement: Only Must set is ignored
- Not to include any irrelevant statement-After ignoring statements from must set, path is to be followed by backtracking from the criterion node in the representation that helps to include the only nodes that is for sure relevant to the criteria.

Hence precision is attained.

Step 7: Adding Efficiency in Space and Time:

- For different input values instead of creating different slice sets, the set will be made as union of all sets i.e. the nodes already in the set will not be inserted again, they will be just pointed by the new slice set. Showing with an example in Figure 4, the new slice set S2 is 1, 2, 4, 10, 11. Only node 4 has to be created, instead of creating the all the nodes again, the nodes already present in slice set s1 are just pointed by slice set S2. Whenever the nodes of specified criteria are required, the pointed node numbers are highlighted in the main graph. This way the approach overcomes the limitation of dynamic slicing of huge space requirement and become space efficient without any compromise in its quality and precision.

Also the reduced edges lead to reduced amount of time taken.

- While connecting the two representations of procedures for the final graph, if the representation of same code exists is in the two parts i.e. their graphical representation is same, merge the second exact representation with the first one by avoiding its representation and connecting it with the previous one, it is also in the case of representing the another class derived from it.
- Special care is given in case of arrays or loops by performing Static Slicing first- Suppose the criteria to create a slice is <x, 10>, first create a static slice based on x, so that whenever an input is given especially in case of loops or arrays, slice is to be created from less number of statements i.e. static slice already created, resulting in less time.
- Object oriented features themselves have the capability to reduce the size and time for running or for testing.

Step 8: Assign weight to each node: Assign the weight 1 to the node representing the statement of the criteria. The weight of other nodes is given by considering parent child relationship. Directional edge going from 'a' to 'b' defines 'a' as parent node and 'b' is its child node. Taking the node of criteria as a root node, its immediate child has the same weight till the node has two children. At the time of two children, the weight of both children nodes will be 0.5 and similarly thee weight gets divided with other children.

Step 9: Better Visualization: Reduce the number of nodes as per their Fractal value where fractal values is calculated by assigning the weight 1 to the node representing the node of criteria, fractal value of other node is same as its parent node and in case the vertex has multiple parent vertices, the *FRACTAL VALUE* of a vertex is the maximum of weights from the weights it is getting from its different parent vertices[3].

From the point where the fractal value of a child is coming to be higher than of the parent value, remove all the nodes from that point and add one node labeled as 'more'. It will lead to visualize the important nodes only but can be expanded if needed.

The criteria chosen in the paper is same as of dynamic slicing which enjoys the importance over static slicing in its reduced size, its ease in handling arrays, pointers etc[13] and also suitable for OOP that contains dynamically bound for which static slicing is not suitable, It has capability of handling different thread[4] allowing Parallelism in threaded programs etc. With its advantageous features, its main drawback is its huge requirement of space in memory that has been worked by different researchers, it had been tried by removing some kind of information from the graph but that led to imprecise slice[5]. Precision can't be compromised, realizing this another time it was tried to retrieve the data and control dependencies directly from a program trace[14], but that technique was very time consuming in performing the sequential search of the trace. Besides its efficiency, its major limitation is its compressed graphs have to be fitted entirely in main memory which may be not possible in case of long-running or complex software. These limitations are worked out in this paper.

6. Conclusions and Future Work

Graphical Representation of a program and slicing technique based on that has been created which is indulged with several important features that have not been all together in any representation. Dynamic slicing is better than static slicing; enjoying its feature, its main disadvantage of space is overcome here without compromising in quality and precision. Approach of connecting the duplicated codes allow to get executed only once helps in reducing the complexity in case of errors which also gets duplicated with the part of the code has to get executed under several circumstances. Other features like locating faults, analyzing after change is also an added feature of the technique. Matching with the needs of today's software organization, graph is capable to handle the classes, inheritance and when different classes inherit the same base code, the basic representations will be reused without making the size of graph increased.

With the features included in the graph and slicing technique, Work will be extended further to make the representation as of lightweight analysis, to reduce overhead, to increase speed, these features will be achieved with parallelism that can handle dynamic threads generation in iterative statements with precision.

7. References

1. Ottenstein KJ, Ottenstein LM. The program dependence graph in a software development environment. Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments; 1984 May. p. 177-84. Doi:10.1145/800020.808263
2. Horwitz S, Reps T, Binkley D. Inter procedural slicing using dependence. PLDI'88 Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation; 1988. p. 35-46. Doi: 10.1145/53990.53994
3. Kashima Y, Ishio T, Etsuda S, Inoue K. Variable data-flow graph for lightweight program slicing and visualization. IEICE Trans INF and Syst. 2015 Jun; E98(6):1194-205
4. Larsen L, Harrold MJ. Slicing object-oriented software. Proceedings of 18th International Conference on Software Engineering; Berlin. 1996. p. 495–505.
5. Agarwal H, Horgam JR. Dynamic program slicing. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation; 1990. p. 246-56. Doi: 10.1145/93542.93576, 246-256
6. Zhao J. Dynamic slicing of object-oriented programs. Wuhan University Journal of Natural Sciences. 2001; 6(1-2):391-7.
7. Deng F, Jones JA. Weighted system dependence graph. IEEE 5th International Conference on Software Testing, Verification, Validation; Montreal, QC. 2012. p. 380-9.
8. Zhang X, Gupta N, Gupta R. Pruning dynamic slices with confidence. Proceedings of ACM Conference on Programming Language Design and Implication; 2006 Jun. p. 169-80.
9. Mund GB, Mall R, Sarkar S. An efficient dynamic program slicing technique. Department of Computer Science and Engineering, IIT Kharagpur. 2002; 44(2):123-32.

10. Baah GK, Podgurski A, Harrold MJ. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans Softw Eng. 2010 Jul; 36(4):528-45.
11. Bouteraa I, Bounour N. Towards The use of program slicing in the change impact analysis of aspect oriented programs. Proceedings International Arab Conference on Information Technology (ACIT); Arabia Saudita. 2011.
12. Walkinshaw N, Strathclyde U, Glasgow UK, Roper M, Wood M. Java system dependence graph. Proceedings of 3rd IEEE International Workshop; Amsterdam, Netherlands. 2003 26-27 Sept. p. 55-64. Doi: 10.1109/SCAM.2003.1238031
13. Sikka P, Kaur K. Program slicing techniques and their need in aspect oriented programming. International Journal of Computer Applications. 2013; 70(3):11-4. Doi: 10.5120/11941-7735
14. Zhang X, Gupta R, Zhang Y. Precise dynamic slicing algorithms. 25th International Conference on Software Engineering (ICSE'03); Portland. 2003 May. p. 319-29.