Volume 3, Issue 11, November 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering

Research Paper

Available online at: www.ijarcsse.com

An Overview of Program Slicing and its Different Approaches

Sk. Riazur Raheman[1]
Dept. of CSE
Raajdhani Engineering College,
Odisha, India, 751017

Amiya Kumar Rath[2]
Professor in CSE & Principal
Bhubaneswar DRIEMS, Cuttack
Odisha, India, 754022

M Hima Bindu[3]
Professor & Head Dept. of CA
NOU, Baripada
Odisha, India, 753003

Abstract- The most difficult task a programmer faces is how to decompose a problem into a set of smaller and simpler sub-problems. Program slicing is a well-known program analytical technique that extracts the elements of a program related to particular computation. The conventional notion of a program slice is the set of all statements that might affect the value of a variable. Program slicing has been used for efficient program debugging activities, testing, program comprehension, restructuring, downsizing, and parallelizing. In this paper we address different types of program slicing techniques by considering a very simple example. Program slice is computed by analyzing dependence relations between program statements. To compute program slices we have constructed intermediate structures of a program such as program dependence graph. Also this paper addresses the comparison between different types of slices and applications of program slice in different fields.

Keywords- Program Slicing, PDG, Data dependence, Control dependence, Static Slice, Dynamic Slice.

I. INTRODUCTION

Program slicing was first introduced by Weiser in 1979 [3]. It is a decomposition technique that extracts the relevant statements from program for a particular computation [2]. Weiser [3] defined a program slice S as a reduced, executable program obtained from a program P by removing statements, such that S replicates part of the behaviour of P. A program slice consists of the parts of a program that affect the values computed at some point of interest [3]. Such a point of interest is referred to as a slicing criterion, and typically consists of a pair C = <S, V>, where S is a program statement and V is a program variable [2]. The parts of a program that have an effect on the values computed at a slicing criterion C represent the program slice with respect to criterion C. The task of computing program slices is called program slicing.

II. SLICING APPROACHES

There are two major approaches in program slicing. The first approach is based on iteration of dataflow equations given by Weiser's [3]. According to Weiser's slices are computed in an iterative process, by computing successive sets of appropriate variables for each node in the CFG. The algorithm first computes directly appropriate statements for each node in the CFG, and then indirectly relevant statements are gradually added to the slice. The process stops when no more relevant statements are found. The second approach is slicing via graph reachability [1]. In this approach, slicing is divided into two steps. In the first step, dependence graph of the program is constructed, and then the graph will be traversed based on algorithm to produce slices. A dependence graph is a directed graph where vertices to represent program statements and edges to represent dependences among the statements. Reachablity analysis on the graph can be done by traversing edges on the graph from a node representing the slicing criteria [3]. In this paper we are using graph reachability approach to find the program slice. So let us discuss graph reachability approach in detail before calculation of slices.

III. PROGRAM DEPENDENCE GRAPH

Program Dependence Graph (PDG) of a program P is the graph G = (N, E), where N represents a statement of the program P and E represents inter-statement data or control dependency edges in program P [11, 7, 15, 22]. The program dependence graph contains two kinds of directed edges: data dependence edges and control dependence edges.

Data Dependence: let G be the Program Dependence graph of program P and x and y be two nodes in G [6, 31, 33]. Node x is said to be data dependent on a node y if there exist a variable var of the program P such that the following hold:

1. The node x defines var
2. The node y uses var, and
3. There exists a directed path from x to y along which the var is not redefined.

Consider the program given in Fig. 1 and its program dependence graph given in Fig. 2. The node 5 has data dependence on node 3 and 4. It is because node 3 and 4 define var i and n respectively and node 5 uses var i and n without redefinition. Like wise node 7 is data dependant on node 6. It is because node 6 defines var j and node 7 uses var j without redefinition. Like this, we have other types of data dependence in the graph.

Control Dependence: let G be the Program Dependence graph of program P and x and y be two nodes in G [6, 33]. Node y is control dependent on a node x if the following hold:

1. x is a conditional predicate, and
2. The result of x determines whether y will be executed or not.

Consider the program given in Fig. 1 and its program dependence graph in Fig. 2. The nodes 8, 9 and 10 are control dependent on node 7. It is because node 7 is a conditional predicate and execution of nodes 8, 9 and 10 depends on the value evaluated at node 7. Like that nodes 11, 12 and 13 are also control dependent on node 7 and so on.

```
1.      int main()
        {
2.              int A,B,C,i,j,n;
3.              i = 1;
4.              scanf("%d",&n);
5.              while (i<n) {
6.                      scanf("%d",&j);
7.                      if(j<0){
8.                              A=n-j;
9.                              B=A+i;
10.                             C=A+B;
                        }
                        else{
11.                             A=n+j;
12.                             B=A-i;
13.                             C=A+B-i;
                        }
14.                     i = i + 1;
                }
15.             printf("%d \n", A);
16.             printf("%d \n", B);
17.             printf("%d \n", C);
        }
```

Fig. 1: A Sample Program P

IV. TYPES OF PROGRAM SLICING

Depending upon the run-time environment slicing can be

1. Static slicing
2. Dynamic slicing

Depending upon graph traversal slicing can be

1. Backward slicing
2. Forward slicing

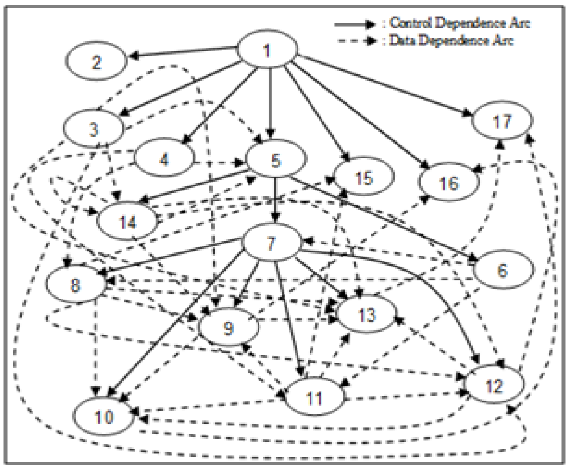

Fig. 2: Program Dependence Graph of Program in Fig. 1

1. **STATIC SLICING**

Static slices are computed by computing consecutive sets of relevant statements, according to data flow and control flow dependences [6]. This is called static slice, because only statically available information is used for computing slices. A static slice contains all statements that may affect the value of a variable at a program point for every possible input [35]. A static slice is always same or greater than the dynamic slice. To calculate static slice, first construct the Program Dependence Graph (PDG) of the given program. We calculate static slice with respect to a slicing criterion (P,V) where P is a statement and V is a variable used. In order to get a static slice for slicing criterion (P,V), PDG nodes are traversed in backward reachabilty using the concept of either Breadth First Search (BFS) or Depth First Search (DFS) starting from node P. Fig.3 shows a static slice of slicing criterion (15, A) for the program given in Fig.1. The static slice with respect to criterion (15, A) consists of the nodes { 1 , 3 , 4 , 5 , 6 , 7 , 8 , 11 , 14 , 15 }.

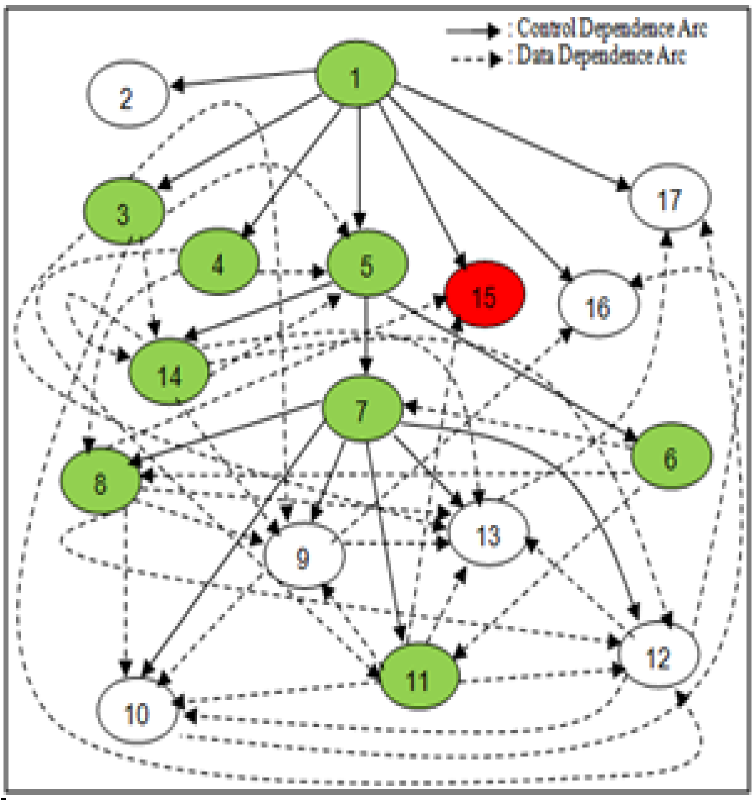

Fig. 3: Static Slice W.R.T Criterion (15, A)

2. **DYNAMIC SLICE**

Dynamic slice was first introduced by Korel and Laski [4]. Dynamic slicing is based on particular execution of a program. The dynamic slices are computed with respect to the execution history. A dynamic slice with respect to a slicing criterion < P, V >, for a particular execution, contains those statements that actually affect the slicing criterion in the particular execution [6, 12, 35]. Dynamic slices are usually smaller than static slices, thus allowing an easier localization of the bugs and are more useful in program testing. We can say that dynamic slicing techniques compute precise slices. In calculation of a dynamic slice, we analyze dependence from an execution history. This history records the execution of statements as the program executes. Dynamic slice is calculated from PDG based on the dependence analysis of the execution history. Thus, statements that have not executed are removed from slice [12].

We calculate dynamic slice as follows [12]:

1. All nodes in the PDG are drawn dotted in the beginning.
2. As statements are executed, corresponding nodes in the graph are made solid.
3. In order to get a dynamic slice PDG is traversed only for solid nodes in backward reachabilty using the concept of either BFS or DFS starting from node P. All nodes reached during the traversal are made bold. The set of all bold nodes gives the desired slice.
4. Consider a particular execution of the program in Fig. 1 for the input value n=3, j=1 and j=-3 and its corresponding dynamic slice in Fig. 4. The input value n=3, j=1 and j=-3 yields the execution history < 1 , 2, 3, 4 , 51 , 6 , 7 , 11 , 12 , 13 , 14 , 52 , 62 , 72 , 8 , 9 , 10 , 142 , 53 , 15 , 16 , 17 >. Now we construct the PDG of the program in Fig. 1. Initially all the nodes in PDG are drawn dotted. After getting the execution history corresponding nodes in the PDG are made solid. To calculate the dynamic slice PDG is traversed based on the slicing criterion [12]. The dynamic slice with respect to criterion (n = 3 , j = (1 , -3) , 15 , A) consists of the nodes { 1 , 3 , 4 , 5 , 6 , 7 , 8 , 11 , 14 , 15 }.

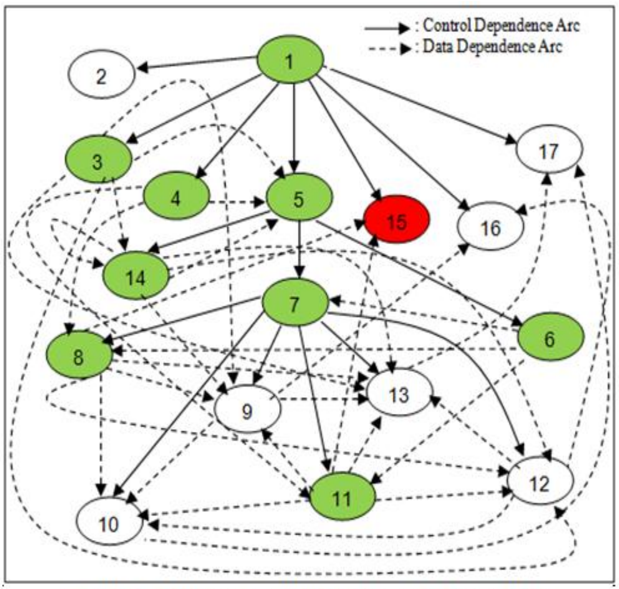

Fig. 4: Dynamic Slice in Approach 1 W.R.T Criterion (n=3, j= (1,-3), 15, A)

Consider the program in Fig. 1. In the statement 15 we display the value of A. This value of A is updated in statement 8 and 11. When we display the final value of A in statement 15 either it is updated by statement 8 or 11. As statement 15 is data dependent on statement 8 and 11 both the statements 8 and 11 are coming under the slice while calculating dynamic slice. Similarly, statement 5 is data dependent on statement 3 and 14. In the first iteration statement 5 depends on the value of i defined in statement 3, but in further execution value of i is updated by statement 14. Hence in next iteration statement 5 is no more depending on statement 3 for value of i, rather it is depending on statement 14. As statement 5 is data dependent on statement 3 and 14, both statements 3 and 14 are coming under dynamic slice. Hence dynamic slice calculated in the above method is not precise.

We calculate dynamic slice in approach 2 as [12, 19, 28]:

1. Mark the edges of the Program Dependence Graph by solid lines as the corresponding dependencies arise during the program execution.
2. Unmark the solid edges of the program dependence graph by dotted lines when the dependency goes off.
3. Then traverse the graph using BFS or DFS only along solid edges to find the slice.

Consider a particular execution of the program in Fig. 1 for the input value n=3, j=1 and j= -3 and its corresponding dynamic slice in Fig. 5 using approach 2. The dynamic slice with respect to criterion (n = 3 , j = (1 , -3) , 15 , A) consists of the nodes { 1 , 4 , 5 , 6 , 7 , 8 , 14 , 15 }.

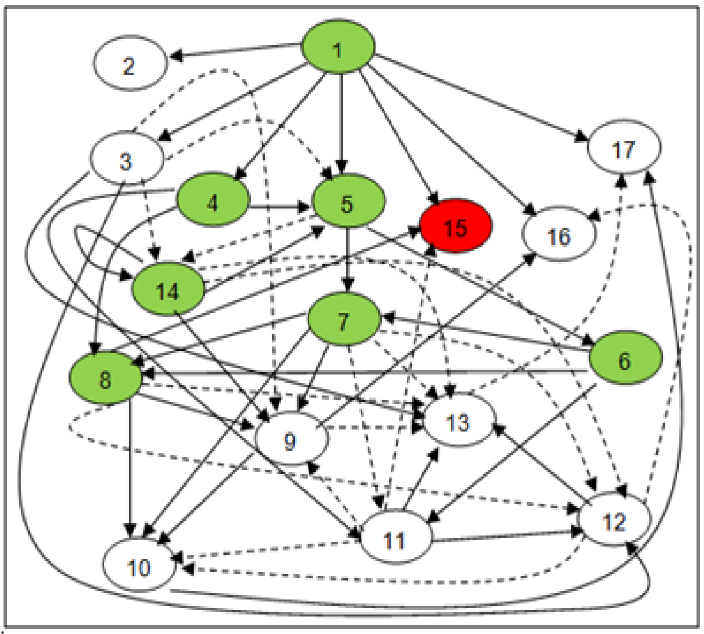

Fig. 5: Dynamic Slice in Approach 2 W.R.T Criterion (n=3, j= (1,-3), 15, A)

3. BACKWARD SLICE

A program can be traversed forward or backward from the slicing criterion. When we traverse it backwards it gives backward slicing. A backward slice of a program with respect to a program point p and set of program variables V consists of all statements and predicates in the program that may affect the value of variables in V at p. Backward slices contain all parts of the program that might have influenced the variable at the statement under consideration. The main applications of backward slicing are debugging and testing [20].

To compute a backward slice from point p, compute backward reachability in the PDG from node p using BFS or DFS. Consider the program given in Fig. 1 and its corresponding backward slice in Fig. 6 with respect to criterion (15, A). Backward slice consists of the nodes { 1 , 3 , 4 , 5 , 6 , 7 , 8 , 11 , 14 , 15 }.

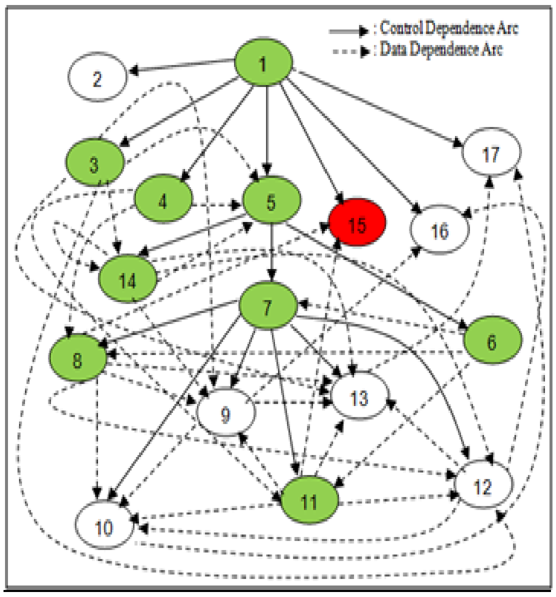

Fig. 6: Backward Slice W.R.T Criterion (15, A)

4. FORWARD SLICE

When we traverse the graph forwards, it gives forward slicing. Forward slicing determines how a modification in a part of the program will affect other parts of the program. A forward slice of a program with respect to a program point p and set of program variables V consists of all statements and predicates in the program that may be affected by the value of variables in V at p. Forward slices contain all parts of the program that might be influenced by the variable. The main applications of forward slicing are Program debugging, program comprehension, program analysis, software maintenance and testing [20, 30].

To compute a forward slice from point p, compute forward reachability in the PDG from node p to other nodes. Consider the program given in Fig. 1 and its corresponding forward slice in Fig. 7 with respect to criterion (4, n). Forward slice consists of the nodes { 4 , 5 , 8 , 11 }.

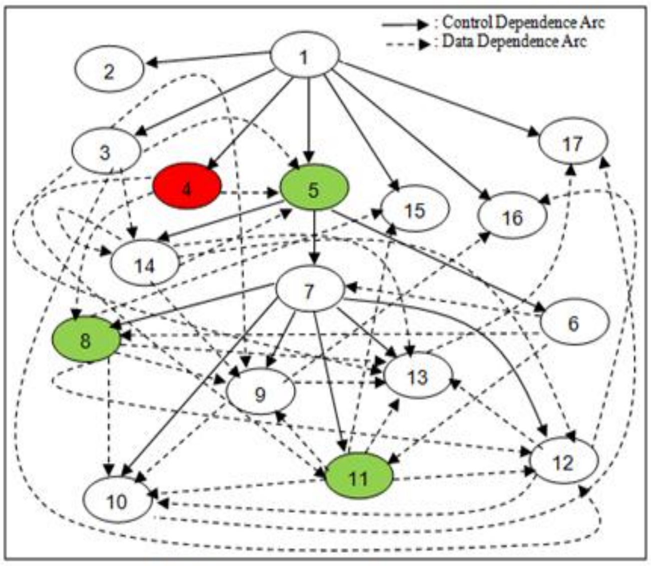

Fig. 7: Forward Slice W.R.T Criterion (4, n)

V. COMPARISON OF STATIC AND DYNAMIC SLICES

Static Slice	Dynamic Slice
1. Only the dependences that occur in a specific execution of the program are taken into account.	1. Slicing criterion is a triple (input, occurrence of a statement, variable). It specifies the input, and distinguishes between different occurrences of a statement in the execution history.
2. Does not make assumptions regarding the input.	2. Assumes fixed input for a program.
3. Slices will typically be larger, but will cater for every possible execution of the original program.	3. Slices will typically be much smaller, but they will only cater for a single input.
5. Contains all statements that may affect a variable for every possible	5. Contains all statements that actually affect the value of a variable
6. Not suitable for debugging as compared to dynamic slice	6. Are very attractive as an aid to debugging, and in particular, that they are superior to static slices for this application.

VI. APPLICATIONS OF SLICING

Slicing is helpful in many aspects of the software development life cycle, including program debugging, software testing, software measurement, program comprehension, software maintenance, program parallelization and so on. Also it can be been applied to many other problem areas. The following section lists some of the applications of slicing in different fields.

1. DEBUGGING

Debugging is a generalized term which essentially means to step through a process in order to systematically eliminate errors. In programming, debugging is done when undesirable program execution or termination occurs. Debugging is a necessary process in almost any new software or hardware development process, whether a commercial product or an enterprise or personal application program. Finding bugs in a program is always difficult [35].

The main application that Weiser had in mind for slicing was debugging [3], if a program computes an erroneous value for some variable x at some program point, the bug is likely to be found in the slice with respect to x at that point. This is achieved by setting the slicing criterion to be the variable for which an incorrect value is observed. Dynamic slicing in particular is appropriate when applying program slicing to debugging, as it produces smaller slices and makes available the inputs that caused the fault.

Slicing cannot be used to identify bugs such as `missing initialisation of variable'. If the original program does not contain a line of code then the slice will not contain it either. Although slicing cannot identify errors of omission, it has been argued that slicing can be used to aid the detection of such errors.

2. REGRESSION TESTING

Regression Testing plays an important role in any scenario where a change has been made to a previously tested software code. Regression means retesting the unchanged parts of the application. Test cases are re-executed in order to check whether previous functionality of application is working fine and new changes have not introduced any new bugs. This testing is done to make sure that new code changes should not have side effects on the existing functionalities [21, 35].

Program slicing can be used to partition tests cases into those that need to be rerun, as they may have been affected by a change, and those that can be ignored, as their behavior can be guaranteed to be unaffected by the change.

3. SOFTWARE MAINTENANCE

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. The main challenges in effective software maintenance are to understand various dependencies in existing software and to make changes to the existing software without introducing new bugs [27].

4. REVERSE ENGINEERING

Reverse engineering is the process of analyzing a technology specifically to ascertain how it was designed or how it operates. Reverse engineering concerns the problem of comprehending the current design of a program and the way this design differs from the original design. Reverse engineering can be viewed as the process of analyzing a system to create representations of the system in another form or a higher level of abstraction [35].

5. SOFTWARE QUALITY ASSURANCE

Software quality assurance consists of a means of monitoring the software engineering processes and methods used to ensure quality. It provides a level of confidence that software is free from vulnerabilities, either intentionally designed into the software or inserted at anytime during its lifecycle and that the software functions in the intended manner [35].

The original motivation for program slicing was to aid the location of faults during debugging activities. This is achieved by setting the slicing criterion to be the variable for which an incorrect value is observed.

VII. CONCLUSION

This paper discussed about various types of slicing techniques like the static slicing, dynamic slicing, forward and backward slicing. Each category of slicing is explained in detail with program portion which was developed in C language. The application of slicing in various areas like debugging, maintenance and re-engineering and testing are highlighted in this paper.

REFERENCES

[1] Baowen Xu et al., A Brief Survey Of Program Slicing Department of Computer Science and Engineering, Southeast University, Nanjing 210096, China , 2005.

[2] Binkley D.W. and Gallagher K.B program slicing. Advances in computer, 43, 1996.

[3] M. Weiser. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[4] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[5] Sebastian Danicic et al., A unifying theory of control dependence and its application to arbitrary program structures, Theoretical Computer Science 412 6809–6842, 2011.

[6] Frank Tip, A Survey of Program Slicing Techniques, *CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, 2004.*

[7] Raju Halder and Agostino Cortesi, Abstract program slicing on dependence condition graphs, Science of Computer Programming, 2012.

[8] Torben Amtoft and Anindya Banerjee, A logic for information flow analysis with an application to forward slicing of simple imperative programs, Science of Computer Programming 64, 3–28, 2007.

[9] Xiangyu Zhang et al., A study of effectiveness of dynamic slicing in locating real faults, Springer Science, 2006

[10] Wes Masri and Andy Podgurski, Algorithms and tool support for dynamic information flow analysis, Information and Software Technology 51, 385–404, 2009.

[11] Liang D. and Harrold M. J. Slicing Objects Using System Dependence Graph. In *Proceedings of the International Conference on Software Maintenance, IEEE,* pages 358–367, November 1998.

[12] Agrawal H. and Horgan J. R. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation, SIGPLAN Notices, Analysis and Verification*, volume 25, pages 246–256, 1990.

[13] Zhao J., Dynamic Slicing of Object-Oriented Programs. Technical report, Information Processing Society of Japan, May 1998.

[14] Larsen L. and Harrold M. J., Slicing Object-Oriented Software. In *Proceedings of 18th International Conference on Software Engineering*, pages 495–505, March 1996.

[15] Horwitz S. et al., Inter-Procedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[16] Chen Z. and Xu B. Slicing Object-Oriented Java Programs. *ACM SIGPLAN Notices*, 36(4), April 2001.

[17] Varun Gupta, Jitender Kumar Chhabra, Dynamic cohesion measures for object-oriented software, Journal of Systems Architecture 57, 452–462, 2011.

[18] Tao Jiang et al., Locating dependence structures using search-based slicing, Information and Software Technology 50, 1189–1209, 2008.

[19] Mohapatra D. P. et al., A Node-Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *Proceedings of Conference on Software Design and Architecture (SODA'04)*, 2004.

[20] G.A.Venkatesh., The semantic approach to program slicing. In *Proceedings of the ACMSIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 107–119, 1991. *SIGPLAN Notices* 26(6).

[21] Swarnendu Biswas and Rajib Mall, Regression Test Selection Techniques: A Survey, Informatica 35, 289–321, 2011.

[22] A. Lakhotia, Graph theoretic foundations of program slicing and integration. ReportCACS TR-91-5-5, University of South-western Louisiana, 1991.

[23] S. Horwitz and T. Reps, The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, 1992.

[24] S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Informatica*, 28:713–732, 1991.

[25] Andreas Lochbihler and Gregor Snelting, On temporal path conditions in dependence graphs, Autom Softw Eng, Springer Science, 16: 263–290, 2009.

[26] Dennis Giffhorn and Christian Hammer, Precise slicing of concurrent programs - An Evaluation of static slicing algorithms for concurrent programs, Autom Softw Eng, Springer Science, 16: 197–234, 2009.

[27] Keith Brian Gallagher and James R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering, 17(8), pp. 751-761, 1991.

[28] G B Mund and R mall, An efficient dynamic program slicing technique, Information and Software Technology, 2002.

[29] Tonella et al., Flow insensitive C++ pointers and polymorphism analysis and its application to slicing. In Proceedings of 19th International Conference on Software Engineering, pp. 433-443, 1997.

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

[31] Madhusmita Sahu and Durga Prasad Mohapatra, A Node-Marking Technique for Slicing Concurrent Object-Oriented Programs, International Journal of Recent Trends in Engineering, Vol 1, No. 1, May 2009.

[32] Takashi Ishio et al., Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique, Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03).

[33] G.B. Mund, R. Mall*, S. Sarkar, Computation of intraprocedural dynamic program slices, Information and Software Technology 45, 499–512, 2003

[34] G.B. Mund, Rajib Mall, An efficient interprocedural dynamic slicing method, The Journal of Systems and Software 79, 791–806, 2006.

[35] N.Sasirekha et al., Program Slicing Techniques And Its Applications , International Journal of Software Engineering & Applications (IJSEA), Vol.2, No.3, July 2011.

[36] Durga Prasad Mohapatra et al., An Overview of Slicing Techniques for Object-Oriented Programs, Informatica 30, 253–277, 2006.

[37] Jeff Russell, Program Slicing Literature Survey, December 2001.