International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 2 Issue: 2 249 – 259

Dynamic Slice of Aspect Oriented Program: A Comparative Study

Sk. Riazur Raheman, Amiya Kumar Rath, Hima Bindu M

Dept. of CSE, Raajdhani Engineering College, Bhubaneswar, Odisha, 751017, India
Professor in Dept. of CSE & Principal DRIEMS, Cuttack, Odisha, 754022, India
Professor & Head Dept. of Computer Science and Application, NOU, Odisha, 757003, India

skriazur79@gmail.com, amiyaamiya@rediffmail.com, mhimabindu@yahoo.com

Abstract-- Aspect Oriented Programming (AOP) is a budding latest technology for separating crosscutting concerns. It is very difficult to achieve crosscutting concerns in object-oriented programming (OOP). AOP is generally suitable for the area where code scattering and code tangling arises. Due to the specific features of AOP language such as joinpoint, point-cut, advice and introduction, it is difficult to apply existing slicing algorithms of procedural or object-oriented programming directly to AOP.

This paper addresses different types of program slicing approaches for AOP by considering a very simple example. Also this paper addresses a new approach to calculate the dynamic slice of AOP. The complexity of this algorithm is better as compared to some existing algorithms.

Keywords *– Program Slicing, Aspect, AOP, OOP, SDG, Data dependence, Control dependence, Static Slice, Dynamic Slice*

*****

1. Introduction

The goal of AOP is to separate concerns in software. It is possible to encapsulate the crosscutting concerns as module unit *aspect* that is easier to develop, maintain and reuse [7, 40, 42, 8]. Aspects separated from an object-oriented program are composed by *Aspect Weaver* to construct the program with the crosscutting concerns. *AspectJ is* an aspect weaver for Java. AspectJ is developed by Xerox Parc. It works as a precompiler and you can see the generated code. It also gives the advantage that the end users don't need to install anything special to run the programs except the virtual machine.

There are number of differences between procedural or object-oriented programming languages and aspect-oriented programming languages. [45]. The concepts of aspect oriented programming like, aspects, join points, advice, and their associated constructs are different from procedural or object-oriented programming languages [44]. All these concepts of aspect-oriented programming should be handled appropriately, because it plays a measure role on the calculation of program slices for aspect-oriented program [6, 28].

2. Aspects

AOP complements OOP by providing a different way of thinking about program structure. AOP defines an innovative program construct, called an *aspect.* The key unit of modularity in OOP is the class, whereas in AOP the unit of modularity is the *aspect*. Like classes in object oriented programs aspects in aspect oriented Programs gathers all the functionality inside of it. The application classes keep their well-defined responsibilities. Consider the example of an aspect given below.

```
Aspect ExampleAspect
{

}
```

This example declares a new aspect with the name "ExampleAspect" [20, 33, 41].

An AspectJ program is divided into two parts. **Base code or non-aspect code,** which includes classes, interfaces and other standard Java constructs. **Aspect code**, which implements the cross-cutting concerns in the program. Given below (figure-1) is an aspect program to test a number is prime or not.

3. Features of AspectJ

AspectJ adds some new features to Java [8].

These features include

- join points
- point-cut
- advice
- introduction
- point-cut designator

3.1 Join Points

These are the points through which we can link the aspect and non aspect code. Examples of some join points are, method call (a point where method is called), method execution (a point where method is invoked) and method reception join points (a point where a method received a call, but this method is not executed yet) [10, 33, 39].

Non aspect code	Aspect code

Import java.util.*;
public class prime{
private static int n;
1. public static void main(String args[]){
2. n=Integer.parseInt(args[0]);
3. if(isprime(n))
4. System.out.println("IS PRIME");
else
5. System.out.println("IS NOT PRIME");
}
6. public static boolean isprime(int n){
7. for(int i=2; i<=n/2; i++){
8. if(n%i == 0){
9. return false;
} }
10. return true;
} }
	11. public aspect PrimeAspect{
	12. public pointcut primeoperation(int n): call (boolean prime.isprime(int) && args(n);
	
	13. before (int n): primeoperation(n){
	14. System.out.println("Testing the prime number for " +n); }
	
	15. after(int n) returning (boolean result): promeoperation(n){
	16. system.out.println(showing the prime status for" + n);
	}
	}

Figure-1 (Aspect program to test a number is prime or not)

3.2 Point-cut

This is a predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut. AspectJ defines different types of point-cut designators that can identify all types of join points [10]. For example in figure-1 the point-cut primeoperation at statement 12 picks out each call to the method isprime() of an instance of the class prime, where an int is passed as an argument and

it makes the value to be available to the point-cut and advice [33, 34, 38].

3.3 Advice

Actions taken by an aspect at a particular join point. It is a method-like construct which is used to define cross-cutting behaviour at join points [37]. Using advice we can define certain code to be executed when a point-cut is reached [10, 33].
There are three types of advice in AspectJ:
after

- *before*
- *around*

After advice on a particular join point runs after the program proceeds with that join point. For example in figure-1, after advice at statement 15 runs after each join point picked out by the point-cut primeoperation and before the control is return to the calling function.

Before advice runs as a join point is reached, before the program proceeds with the join point. For example in figure-1 the before advice at statement 13 runs before the join point picked out by the point-cut primeoperation.

Around advice on a join point runs as the join point is reached, and has explicit control over whether the program proceeds with the join point.

3.4 Introduction

It allows an aspect to add methods, fields or interfaces to existing classes. It can be *public* or *private* [10]. An introduction declared as *private* can be referred or accessed *only* by the code in the aspect that declared it. An introduction declared as *public* can be accessed by *any* code [39, 41].

3.5 Point-cut Designator

A point-cut designator identifies all types of join points. A point-cut designator simply matches certain join points at runtime. For example in figure-1 the point-cut designator *Call (boolean prime.isprime(int),* at statement 12 matches all the method calls to factorial from an instance of the class *prime* [41].

4. Comparison of OOP and AOP

OOP	AOP
Class: Code unit that encapsulates methods and attributes.	*Aspect:* Code unit that encapsulates pointcuts, advice, and attributes.
Method signatures: Define the entry points for the execution of method bodies.	*Pointcut:* Define the set of entry points in which advice is executed.
Method bodies: Implementations of the primary concerns.	*Advice:* Implementations of the crosscutting concerns.
Compiler: Converts source code into object code.	*Weaver:* Instruments code (source or object) with advice.

5. Some of the existing slicing techniques of AOP

The first attempt towards the development of aspect-oriented system dependence graph (ASDG) to represent aspect oriented programs was made by Zhao [18]. To construct the ASDG of aspect code, Zhao first constructed the SDG of non aspect code then aspect dependence graph (ADG) for aspect code. He combined the SDG and ADG using some extra dependence arcs to construct ASDG [10, 4]. Zhao used the two-phase slicing algorithm proposed by Larsen and Harrold to compute the static slice of aspect oriented programs [15].

Let's implement the Zhao approach to calculate the static slice of the program given in figure-1.

To represent the aspect oriented program, aspect-oriented system dependence graph (ASDG) is constructed by combining SDG and ADG using some additional arcs (figure-2).

By applying the two-phase slicing algorithm in figure-2 the static slice comes as: 1 , 2 , 3 , 6 , 7 , 8 , 9 , 10 , 13 , 14 , 15 , 16.

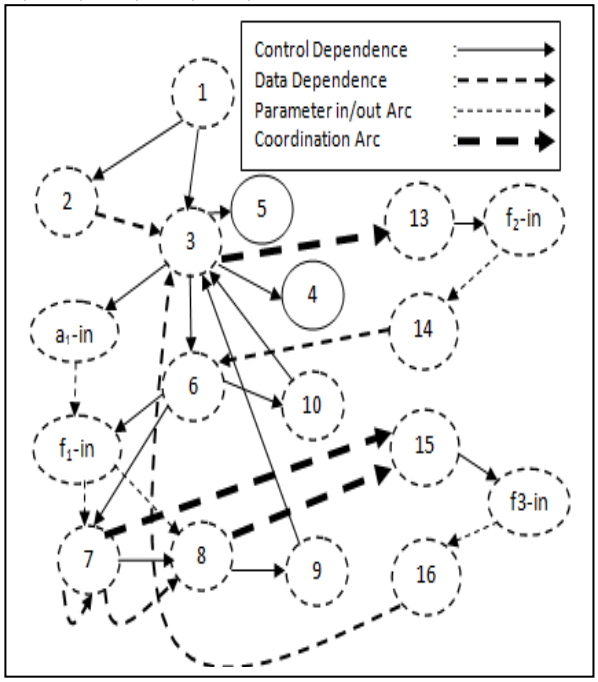

Figure-2 (ASDG of the program given in figure-1)

D P Mohapatra et. al., [10] has proposed an algorithm to calculate the dynamic slice of aspect-oriented programs. They have used Dynamic Aspect-Oriented Dependence Graph (DADG) to represent the aspect oriented program. Initially they have executed the aspect program for any particular input and the execution history of the program is traced using a trace file.

Let's implement the D P Mohapatra et al. approach to calculate the dynamic slice of the program given in figure-1. First we have to execute the program for a given input. Lets execute the program for the value of n=7. The execution trace of the program is as follows:

1(1) : Public static void main(String args[])
2(1) : n=Integer.parseInt(args[0])
3(1) : if(isprime(n))
12(1) : public pointcut primeoperation(int n): call (boolean prime.isprime(int) && args(n)
13(1) : before (int n): primeoperation(n)
14(1) : system.out.println("Testing the prime number for " +n)
6(1) : public static boolean isprime(int n)
7(1) : for(int i=2; i<=n/2; i++)
8(1) : if(n%i == 0)
7(2) : for(int i=2; i<=n/2; i++)
8(2) : if(n%i == 0)
7(3) : for(int i=2; i<=n/2; i++)
15(1) : after(int n) returning (boolean result): promeoperation(n)
16(1) : system.out.println(showing the prime status for" + n)
10(1) : return true
4(1) : system.out.println("IS PRIME")

Now construct the Dynamic Aspect-Oriented Dependence Graph (DADG) for the execution trace of the program [figure-3]. To compute the dynamic slice for the slicing criterion < 10, n >, apply either the breadth first search or depth first search algorithm on the DADG [12]. The dynamic slice comes as : 1 , 2 , 3 , 6 , 7 , 10 , 13 , 14 , 15 , 16.

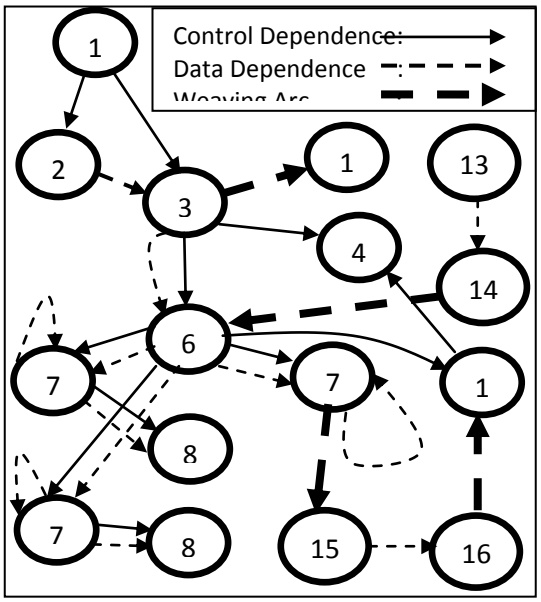

Figure-3 (DADG for the execution trace of the program given in Fig - 1)

This paper proposes a graph called *Aspect-Oriented System Dependence Graph (AOSDG)* to represent aspect program. This AOSDG represents all the features of an aspect program as well as it also represents the weaving arc. The construction of AOSDG can be carried out in two phases, first construction of system dependence graph for non aspect code then construction of system dependence graph for aspect code and finally linking the system dependence graph of aspect and non aspect code using call and weaving arc.

6. Proposed Algorithm

We discussed two existing techniques to calculate the static and dynamic slice os aspect oriented program. The existing techniques discussed can be improved on certain points. In the first technique proposed by Zhao, the point-cuts are not handled properly and also the weaving process is not represented correctly. In the second technique proposed by D. P. Mohapatra et. al., If a statement will execute for n number of times it will create n vertices for each iteration of the statement, which will be a difficult task. To overcome all these we have proposed a new approach to compute the dynamic slice of aspect oriented program.

1. Construction of System Dependence Graph (SDG) for non-aspect code.
2. Construction of System Dependence Graph (SDG) for aspect code.
3. Construction of Aspect-Oriented System Dependence Graph (AOSDG).
4. To compute the dynamic slice traverse the AOSDG using breadth first search or depth first search taking a vertex as starting point of traversal.

7. Construction of System Dependence Graph (SDG) for aspect and non-aspect code

This section describes the construction of SDG for aspect and non-aspect code [7, 9, 11, 15, 19, 24, 26, 31]. In SDG of an aspect program, following types of dependence arcs may exist.

- control dependence arc
- data dependence arc
- call arc

Control dependence represents the control flow relationship of a program i.e, the predicates on which a statement or an expression depends during execution [2]. Consider statements s_1 and s_2 in a source program p. When all of the following conditions are satisfied, we say that control dependence (CD), from statement s_1 to statement s_2 exists [16, 20, 21, 22, 29, 32, 36]:

1. s_1 is a conditional predicate, and
2. the result of s_1 determines whether s_2 is executed or not.

For example, in the Figure-1, there is a control dependency between statement 3 and 4, because statement 3 is a conditional predicate and the result of 3 determines whether 4 will be executed or not.

Data dependences represent the relevant data flow relationship of a program i.e., the flow of data between statements and expressions. When the following conditions are satisfied, we say that data dependence (DD), from statement s_1 to statement s_2 by a variable v, exists [17, 20, 21, 22, 32]:

1. s_1 defines v, and
2. s_2 refers to v, and
3. at least one execution path exists from s_1 to s_2 without redefining v.

For example, in the Figure-1, there is a data dependency between statement 7 and 8, because statement 7 is defining the value of i and statement 8 is using the value of i defined by statement 7 and in between statement 7 and 8 there is no redefinition of value of i.

Call arc represents the function call in a program. For example in figure-1, there is a call arc from statement 3 to statement 6. Because function isprime() is called at statement 3 and the function isprime() is defined at statement 6.

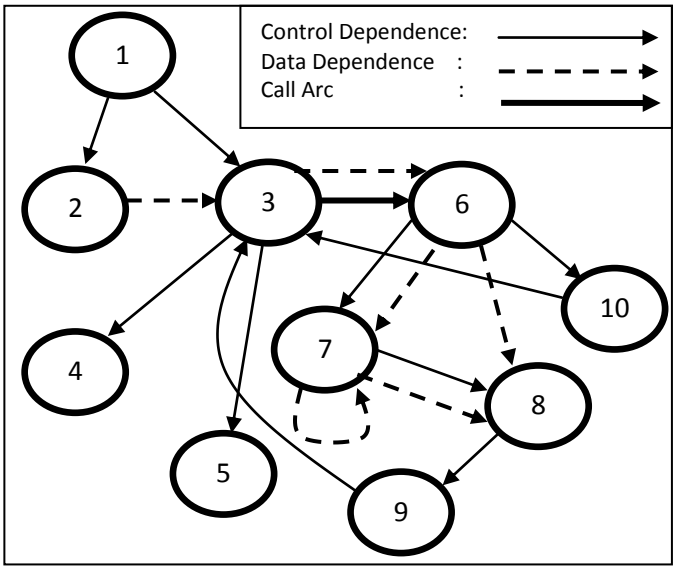

Figure-4 (SDG for non-aspect code of the program given in figure-1)

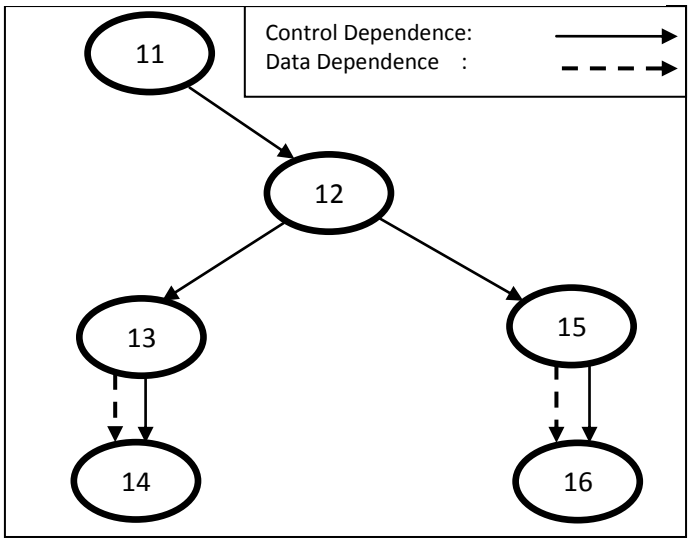

Figure-5 (SDG for aspect code of the program given in figure-1)

8. Construction of Aspect-Oriented System Dependence Graph (AOSDG)

This section describes the construction of AOSDG for aspect program [43, 45]. The AOSDG is a *graph* (*V, A*), where *V* represents the set of vertices and A represents the set arcs to connect the two vertices. Each statement and predicates of the aspect-oriented program is represented as one vertex.

 ISSN: 2321-8169
249 – 259

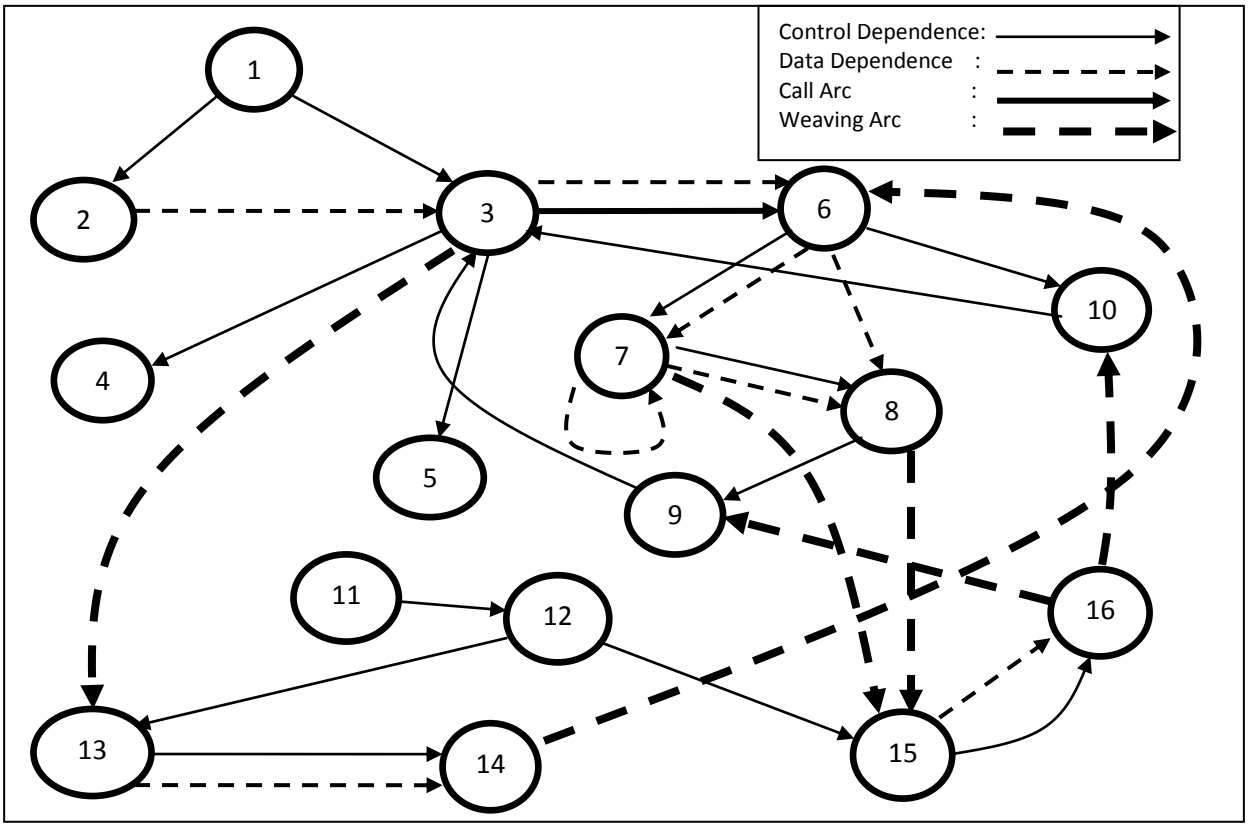

Figure-6 (AOSDG for the aspect program given in figure-1)

AOSDG of an aspect program consists of following types arcs.

- control dependence arc
- data dependence arc
- call arc
- weaving arc

Weaving arc reflects the joining of aspect code and non-aspect code at appropriate join points. For example in figure-1, there is a weaving arc from statement 3 to statement 13. Because at statement 3 there is function call isprime() and at statement 13 a before advice is defined related to function call.

Statement 13 represents a *before* advice. This means that the *advice* is executed before control flows to the corresponding function i.e., to the function *isprime()*. So, we add a *weaving arc* from vertex 14 to vertex 6. Similarly, statement 15 represents an *after* advice. This means that the *advice* is executed after the function *isprime()* has been executed and before control flows to the calling function i.e., the function *main()*. That's why we add a weaving arc from vertex 8 to vertex 15. After the execution of *after* advice at statement 16, control transfers to statement 9 or 10 where it returns a value to the calling function *main()*. So, a weaving arc is added from vertex 16 to vertex 9 and 10 [45].

To compute the dynamic slice we can perform either the breadth first search or depth first search algorithm on the AOSDG [1, 3, 12, 5, 23] in figure - 6. For the slicing criterion < 10, n >, the dynamic slice comes as 1 , 2 , 3, 6 , 7 , 8 , 10 , 11, 12 , 13 , 14 , 15 , 16.

9. Complexity analysis

The proposed technique is having some advantages in terms of time and space complexity as compared to previous techniques.

9.1 Space complexity:

Let P be an aspect-oriented program with N number of statements. Each statement of the program will be represented by a single vertex in the AOSDG. Thus, there are N numbers of vertices in the AOSDG corresponding to all the statements of program P. So, the space complexity is O(N).

9.2 Time complexity:

Let *P* be an aspect-oriented program with N number of statements. The total time complexity is for finding the dynamic slice of aspect oriented program is:

1. Time required for constructing the AOSDG with respect to the N number of statements in the program which is $O(N)$.
2. Time required to traverse the AOSDG either using Breadth First Search or Depth First Search, which is $O(N^2)$.

So, the time complexity is $O(N^2)$.

The proposed algorithm is better than the D. P. Mohapatra et. al. approach in terms of space and time complexity. In D. P. Mohaptra et. al. approach S represents the length of execution of the program. Thus S may be same or more than the number of statements in the program. Where as in proposed approach N represents the number of statements in the program, hence number of nodes created in proposed approach may be same or less than the D. P. Mohaptra et. al. approach.

While calculating the dynamic slice of the aspect program given in figure-1 using D.P.Mohapatra et. al. technique, we are creating three vertices for statement 7 and two vertices for statement 8 as given in figure-3. This is a repeating work which will take more space.

10. Comparison of existing slicing algorithm with proposed algorithm

Zhao Approach	1. **Time Complexity:** $O(S^2)$, Where S represents the number of statements in the program. 2. **Space Complexity:** O(S), Where S represents the number of statements in the program. 3. **Drawbacks:** *point-cuts* are not handled properly and also the *weaving process* is not represented correctly.
D P Mohapatra et. al. Approach	1. **Time Complexity:** $O(S^2)$, Where S represents the length of execution of the program. 2. **Space Complexity:** O(S), Where S represents the length of execution of the program. 3. **Drawbacks:** If a statement will execute for n number of times in the execution trace, it will create n vertices for each iteration of the statement.
Proposed Approach	1. **Time Complexity:** $O(N^2)$, Where N represents the number of statements in the program. 2. **Space Complexity:** O(N), Where N represents the number of statements in the program. 3. **Advantage:** Its time and space complexity is less as compared to D. P. Mohapatra et. al, approach and it handles the point-cut and weaving process efficiently as compared to Zhao approach.

11. Conclusion

This paper discussed the features of aspect oriented programs. Also we have discussed about various types of existing slicing approaches for Aspect Oriented Program (AOP) with example. Also this paper proposed an approach to slicing aspect oriented programs using an Aspect-Oriented System Dependence Graph (AOSDG),

which extends previous system dependence graphs, to represent Aspect Oriented Program. The complexity of the proposed algorithm is also calculated, it comes better than some existing algorithm.

12. References

[1] B. Korel et. al., Dynamic program slicing, *Information Processing Letters*, 29(3):155–163, 1988.

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

[3] Frank Tip, A Survey of Program Slicing Techniques, Journal of Programming Languages, 1995.

[4] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. SIGSOFT Softw. Eng. Notes, 30(2):1-36, March 2005.

[5] Binkley D.W. and Gallagher K.B, program slicing, Frontiers of Software Mentenance, 58-67, 2008.

[6] M. Weiser, *Program slices, IEEE Transactions on Software Engineering, VOL. SE-10, NO. 4, Pages 352-357, JULY 1984.*

[7] Zhao J, Dynamic Slicing of Object-Oriented Programs, Uhan University Journal of Natural Sciences, Vol 6, No 1-2, 391-397, 2001.

[8] Abhishek Ray et. al., An Approach for Computing Dynamic Slice of Concurrent Aspect-Oriented Programs, International Journal of Software Engineering and Its Applications, Vol. 7, No. 1, January, 2013.

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

[10] Mohapatra D. P. et. al., Dynamic Slicing of Aspect-Oriented Programs, informatica 32, 261-274, 2008.

[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] 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.

[14] 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.

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

[16] S. Horwitz and T. Reps, Efficient comparison of program slices, *Acta Informatica*, 28, Volume 28 Issue 9, pages 713–732, 1991.

[17] G B Mund and R mall, An efficient dynamic program slicing technique, Information and Software Technology, Volume 44, Number 2, pp. 123-132(10), 15 February 2002.

[18] Zhao J., Slicing Aspect-Oriented Software, In *Proceedings of 10th International Workshop on Program Comprehension*, pages 251–260, June 2002.

[19] M. Sahu and D. P. Mohapatra, A Node-Marking Technique for Slicing Concurrent Object-Oriented Programs, International Journal of Recent Trends in Engineering, Vol 1, No. 1, May 2009.

[20] 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), page 3, 2003.

[21] G. B. Mund et. al., Computation of intraprocedural dynamic program slices, Information and Software Technology 45, 499–512, 2003.

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

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

[24] Durga Prasad Mohapatra, Rajib Mall and Rajeev Kumar, An Overview of Slicing Techniques for Object-Oriented Programs, Informatica 30, 253–277, 2006.

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

[26] Mohapatra D. P., Mall R., and Kumar R. An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, vol -1 , pp 60-65, Sept. 2004.

[27] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, 13:187–195, 1990.

[28] Mrs. Sonam Jain et. al., A New approach of program slicing: Mixed S-D (static & dynamic) slicing, *International Journal of Advanced Research in Computer and Communication Engineering Vol. 2, Issue 5, May 2013.*

[29] Zhao J. and Rinard M. System Dependence Graph Construction for Aspect-Oriented Programs. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, USA, March 2003.

[30] On bytecode slicing and AspectJ interferences, Antonio Castaldo D'Ursi, Luca Cavallaro, Mattia Monga, Proceedings of the 6th workshop on Foundations of aspect-oriented languages, Pages 35-43, 2007.

[31] S. R. Mohanty et. al., A Simplified Approach to Compute Dynamic Slices Of Procedural Programs, International Journal of Research in Engineering, IT and Social Sciences, Volume 2, Issue 11, November, 2012.

[32] Application of Aspect-Oriented Programming to Calculation of Program Slice Takashi Ishio, Shinji Kusumoto, Katsuro Inoue. Technical Report, Submitted to ICSE2003,

[33] Aspect Oriented Programming, By Gustav Evertsson, 2002

[34] Aspect-Oriented Programming, Jyri Laukkanen , seminar paper, UNIVERSITY OF ELSINKI , 2008.

[35] Keith Brian Gallagher, Using program slicing for program maintenance, PhD thesis, University of Maryland, College Park, Maryland, 1990.

[36] S. Gupta et. al., An Effective Methodology for Slicing C++ Programs, *International Journal of Advances in Engineering Research(IJAER), Vol. No. 1, Issue No. VI, JUNE 2011.*

[37] Ishio , Shinji Kusumoto, Katsuro Inoue Debugging Support for Aspect-Oriented Program Based on Program Slicing and Call Graph, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004.

[38] Gary Pollice, A look at aspect-oriented programming, Worcester Polytechnic Institute, from The Rational Edge, ibm.com/developerWork, 2004.

[39] Madhusmita Sahu, Durga Prasad Mohapatra, A Node-Marking Technique for Dynamic Slicing of

Aspect Oriented Programs, published in the proceedings of 10th International Conference on Information Technology(ICIT), pages 155-160, 2007.

[40] David Robinson, "An Introduction to Aspect Oriented Programming in *e*", Rel_1.0: 21-JUNE-2006.

[41] Gregor Kiczales et. al., An Overview of AspectJ, published in proceedings of the 15th European Conference on Object Oriented Programming, pages 327-353, 2001.

[42] Amogh Katti et. al., Application of Program Slicing for Aspect Mining and Extraction – A Discussion , *International Journal of Computer Applications (0975 – 8887) Volume 38– No.4, January 2012.*

[43] SHI Liang et. al., System Dependence Graph construction for Aspect Oriented C++, Wuhan University Journal of Natural Sciences, Vol-11, No-3, Pages 555-560, 2006.

[44] P. Sikka et. al., Program Slicing Techniques and their Need in Aspect Oriented Programming, *International Journal of Computer Applications(0975 – 8887) Volume 70– No.3, May 2013.*

[45] Sk. Riazur Raheman et. al., Dynamic Slicing of Aspect-Oriented Programs using AODG, (IJCSIS) International Journal of Computer Science and Information Security,Vol. 9, No. 4, April 2011.