ISSN:0975-9646

Slicing Aspect-oriented program Hierarchically

S. R. Mohanty
Dept. of CS
RIMS, Rourkela
Odisha, India Pin 769012

P. K. Behera
Dept. of CSA
Utkal University, Vani Vihar
Odisha, India

D. P. Mohapatra
Dept. of CSE NIT,
Rourkela Odisha, India

***Abstract*—While developing a software system, the complexity in describing a problem should be reduced. This can be done by separating the concern in a clean and explicit way. Each of the concern can be addressed by partitioning a software system into modules. Concerns are clearly identifiable with a special linguistic construct called *Aspects*, which has been introduced by a new programming paradigm known as aspect-oriented programming. No doubt aspect-oriented programming brings lots of opportunities for the software developer. On the other hand, it is very difficult for analysing those programs for different software engineering activities. In such scenario, *slicing* plays a vital role. This paper proposes an approach to compute the dynamic slices of aspect-oriented programs. In our approach, we have introduced different level dependence graphs, such as *aspect-oriented statement level dependence graph*, *aspect-oriented method level dependence graph*, *aspect-oriented AC weaving level dependence graph*, *aspect-oriented package level dependence graph* to represent an aspect-oriented program under consideration. Then, we apply *aspect-oriented reverse hierarchical dynamic slicing algorithm* on the intermediate program representation to compute the dynamic slices. Our algorithm traverses the dependence edges starting from the slicing node in a reverse hierarchical manner to list the reached node, which constitute the dynamic slice of the aspect-oriented program under consideration. This approach is advantageous as it constructs the graph and computes the dynamic slices level wise. At a particular instance the level dependence graph is quite manageable and quickly traversed. It can also generate intermediate slices moving from the statement level to package level. The space complexity of our algorithm is $O(n^2)$, where n is the total number of nodes in the graph and time complexity is $O(n^2)$, Where n is the total number of executed statements in the execution trace of the program.**

I. Introduction

While developing a software system, the complexity in describing a problem required to be reduced. This can be done by separating *concern* in a clean and explicit way. Each of the concern can be addressed by partitioning a software system into modules. Modularization, on the other hand, increases the comprehensibility of the software system.

Concerns are clearly identifiable with a special linguistic construct called aspect. These aspects, are the new features introduced by a new programming paradigm known as *aspect-oriented programming*. Object-oriented programming has increased its popularity among the software developers. The aspect-oriented programming is not a replacement of it. In object-oriented programming languages, the concerns are distributed among the objects. Whereas aspect-oriented programming paradigm introduces a new module, encapsulating the concern named as *aspect*.

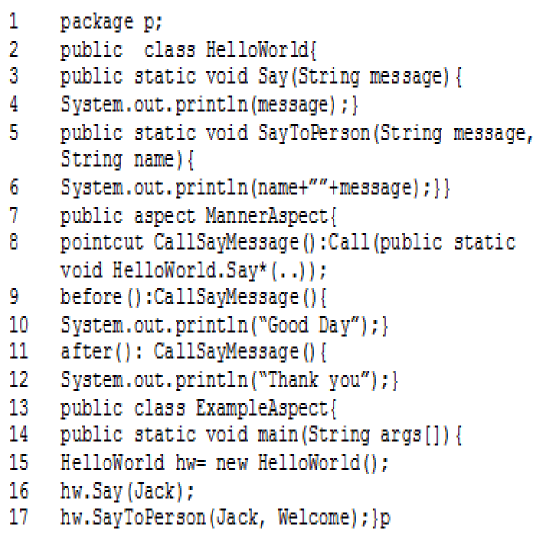
Such parts are developed with isolation and then they assembled together to produce the whole system. Aspect-oriented programming brings lots of opportunities for the software developer. On the other hand, it is very difficult for analyzing those programs through a technique like program slicing. Because aspect-oriented programming introduces some new features like *join point*, *pointcut*, *advices*, *introduction* and *aspect*. Which needs to be handled properly while slicing those programs. This paper focuses on dynamic slicing of aspect-oriented programs.

A. Motivation

Aspect-oriented programming paradigm is introduced with a new feature called *aspect*. Aspect helps to reduce the complexity while developing a software. Aspect can be developed with isolation. AOP brings lots of opportunities for the software developer with its new features like *join point*, *pointcut*, *advices*, *introduction*, and *aspect*. But on the other hand, it is difficult to analyze the technique called program slicing. In this paper, we tried to handle all these features of AOP through suitable intermediate program representation. And to develop a technique to get dynamic slice using it.

B. Objectives

This paper introduced with the objectives of developing a suitable intermediate representation for an aspect-oriented program. We also focused on developing an algorithm to compute the dynamic slices of AOP.
The rest of the paper is organized as follows: Some basic concepts and definitions are discussed in section II. Section III presents the proposed algorithm to compute the dynamic slices of an aspect-oriented program and discusses the working of the algorithm along with the correctness and complexity analysis. Section IV presents the proposed tool architecture. Section V contains a comparative study of the proposed work with some of the existing ideas related to our work. Section VI concludes the paper and presents the future work.



```
package p;
public  class HelloWorld{
public static void Say(String message){
System.out.println(message);}
public static void SayToPerson(String message,
String name){
System.out.println(name+""+message);}}
public aspect MannerAspect{
pointcut CallSayMessage():Call(public static
void HelloWorld.Say*(..));
before():CallSayMessage(){
System.out.println("Good Day");}
after(): CallSayMessage(){
System.out.println("Thank you");}
public class ExampleAspect{
public static void main(String args[]){
HelloWorld hw= new HelloWorld();
hw.Say(Jack);
hw.SayToPerson(Jack, Welcome);}p
```

Fig. 1. An aspect-oriented program example-1

II. Background

This section explains the proposed intermediate representation of the aspect-oriented program as well as various terms used in the proposed algorithm. But before that this paper presents an overview of the aspect-oriented program.

A. Overview of aspect-oriented programs

An aspect-oriented programming paradigm is a bit different concept than that of the object-oriented programming paradigm, we have taken an aspect-oriented program and explained it features like join point, pointcut, advice and aspect. Let us consider an aspect-oriented example program in Fig.1.

In the example program in Fig.1, a class with name HelloWorld is defined with two different member functions Say() and SayToPerson(). Say() method takes one argument message of string type and displays it. SayToPerson() method accept two arguments name and message. Both are of string type and display them on the screen.

Then, the example program declares an aspect, whose declaration is same as a class declaration. But it contains a pointcut named as CallSayMessage(), which declare a join point. Join point tells that when a method of class HelloWorld started with a string "Say" is called with any number of parameters, two pieces of advice will be executed. The aspect define two pieces of advice one is before(), another one is after(). Before() advice will print "Good day" and after advice will print "Thankyou". Before() advice will run and produce the output before the specified method call in the join point. And after() advice will run and produce its output after the specified method call in the join point.

The example program declares the main class named as ExampleAspect. We called it main class because it contain the main() method. In the main() method object "hw" of the class HelloWorld is declared. With the object "hw" two distinct methods Say() and SayToPerson() is being called. The methods called in main() satisfied the criteria specified in the join point declared in the aspect MannerAspect.

The two classes HelloWorld, ExampleAspect along with the aspect MannerAspect are grouped into package "P". When this example program will be compiled by AspectJ compiler the advice i. e. before() and after() are weaved with the join point i. e. Call(public static void HelloWorld.Say*(..)). So When the program starts its execution, the object "hw" is created. At first before() advice is executed then the Say() method is executed. After the execution of Say() method after() advice is executed. Same thing happens when the method SayToPerson() called with the object "hw".

The example program in Fig 1 explains the features of an aspect-oriented program. In the example program, we can see how the concerns are separated from the core module and introduced in a new unit called MannerAspect. Although those concerns are kept isolated they assemble with the main module during the compilation and produce the desired output when the program is executed.

B. Basic concepts and definitions

This section presents some basic concepts and definitions used in the proposed algorithms.

1) AOSL_Dslice: This is the aspect-oriented statement level dynamic slice. Our approach computes dynamic slice of an aspect-oriented program level wise. We broadly defined four levels, they are statement level, method level, class level and package level.
AOSL_Dslice is the slice computed in the statement level. It contains all the statements that affect the slicing node.

2) AOML_Dslice: This is the slice computed at the method level. We called it as the aspect-oriented method level dynamic slice. It contains all the method entry nodes that affect the slicing node.

3) AOCL_Dslice: This is the dynamic slice of an aspect-oriented program in the class level. It contains the classes and aspects that may affect the slicing node. It contains all the classes and aspects that affect the slicing node.

4) AOPL_Dslice: This is the dynamic slice of an aspect-oriented program at the package level. It contains all the packages that affect the slicing node.

5) Slicing node: Before the dynamic slices of the different level are computed, our approach first constructs the different level graphs of an aspect-oriented program under consideration. This graph contains nodes and edges. Nodes represent the statements, methods, class or packages. And the edges represent the dependency among the nodes. Here slicing node represents a node in the graph on which the dynamic slice has to be computed.

6) AO_Dslice: All the different level slices are integrated to form the AO_Dslice. This is the aspect-oriented dynamic slice of an aspect-oriented program under consideration. While eliminating the concepts of different levels, it contains all the statements of the aspect-oriented program that affect the slicing node

C. Intermediate Program Representation

A systematic approach of computing dynamic slice of an aspect-oriented program followed two distinct steps:

- Construction of a suitable intermediate program representation
- Applying the dynamic slicing algorithm on the intermediate program representation.

Our approach is based on constructing the intermediate program representation based on the *execution trace*. We first run the program with the required input values to get the execution trace. Then, we construct the intermediate program representation.

In this section, we present an intermediate program representation for an aspect-oriented program. We have proposed to construct the intermediate program representation of an aspect-oriented program in different levels. We introduced four different level graphs. They are aspect-oriented statement level dependence graph, aspect-oriented method level dependence graph, aspect-oriented AC weaving level dependence graph and aspect-oriented package level dependence graph.

We have introduced these level graphs for computing the dynamic slices of aspect-oriented programs. But we introduced an extra node to represent the weaving of an advice with a join point.

Let us consider an example program given in Fig. 2. We are quoting this example program from the research work done by Zhao [6]. In this section, we construct and explain the graphs for different level i.e. aspect-oriented statement level, aspect-oriented method level, aspect-oriented AC weaving level and aspect-oriented package level of the example program .

The example program contains two classes named as *Point* and *Shadow*. It contain one aspect named as *PointShadowProtocol*. All these are grouped into an package called *P*. Class Point have two data members *x* and *y*, both are of integer

```
1  package P;
2  public class Point{
3  protected int x,y;
4  public Point(int _x,int _y){
5  x=_x;
6  y=_y;}
7  public int getX()
8  return x;}
9  public int getY()
10 return y;}
11 public void setX(int _x){
12 x=_x;}
13 public void setY(int _y){
14 y=_y;}
15 public void printPosition(){
16 System.out.println("point at("+x+","+y")");
17 public static void main(String [ ] args ){
18 Point p=new Point(1,1);
19 p.setX(2);
20 p.setY(2);}}
21 class Shadow{
22 public static final int offset=10;
23 public int x,y;
24 Shadow(int x, int y){
25 this.x=x;
26 this.y=y;}
27 public void printPosition(){
28 System.out.println("point at("+x+","+y")");}}
29 aspect PointShadowProtocol{
30 priate int shadowCount=0;
31 public static int getShadowCount(){
32 return PointShadowount.aspectOf().shadowCount;}
33 private Shadow Point.shadow;
34 public static void associate(Point p, Shadow s){
35 p.shadow=s;}
36 public static Shadow getShadow(Point p){
37 return p.shadow;}
38 Pointcut setting(int x, int y, Point p):args(x,y)&&Call (Point
   .new(int,int));
39 Pointcut settingX(Point p):target(p)&&Call(void Point.setX(int));
40 Pointcut settingY(Point p):target(p)&&Call(void Point.setY(int));
41 after (int x, int y,Point p)returning :setting(x,y,p){
42 Shadow s=new Shadow(x,y);
43 associate(p,s);
44 shadowCount++;}
45 after(Point p):settingX(p){
46 Shadow s=new getShadow(p);
47 s.x=p.getX()+shadow.offset;
48 p.printPosition();
49 s.printPosition();}
50 after(Point p):settingY(p){
51 Shadow s=new getShadow(p);
52 s.y=p.getY()+shadow.offset;
```

Fig. 2. An aspect-oriented program example-2

types.The *Point* class contains one constructor and six methods. Class *Shadow* contains a static final data member *offset* initialized with *10*, two integer *x* and *y*, one constructor and one method. Aspect *PointShadowProtocol* contains a private data member *shadowCount*, which is an integer initialized with *0*, a data member *Point.shadow*. Three *Pointcuts settings, settingX, settingY*. And three *after advices*. These three *after advices* are weaved after the calling of constructor *Point()*, *method setX()*, *method setY()* respectively.

Now, we will explain our proposed level dependence graphs for aspect-oriented program with respect to the example program given in Fig.2.

1) Aspect-oriented statement level dependence graph: Aspect-oriented statement level graph is an arc classified graph and can be defined as follows:

$Gh_{AOSL} = \{N1_d, E1_d\}$

Where $N1_d$ is the set of nodes in the aspect-oriented statement level graph and $E1_d$ is the set of edges depicting various types

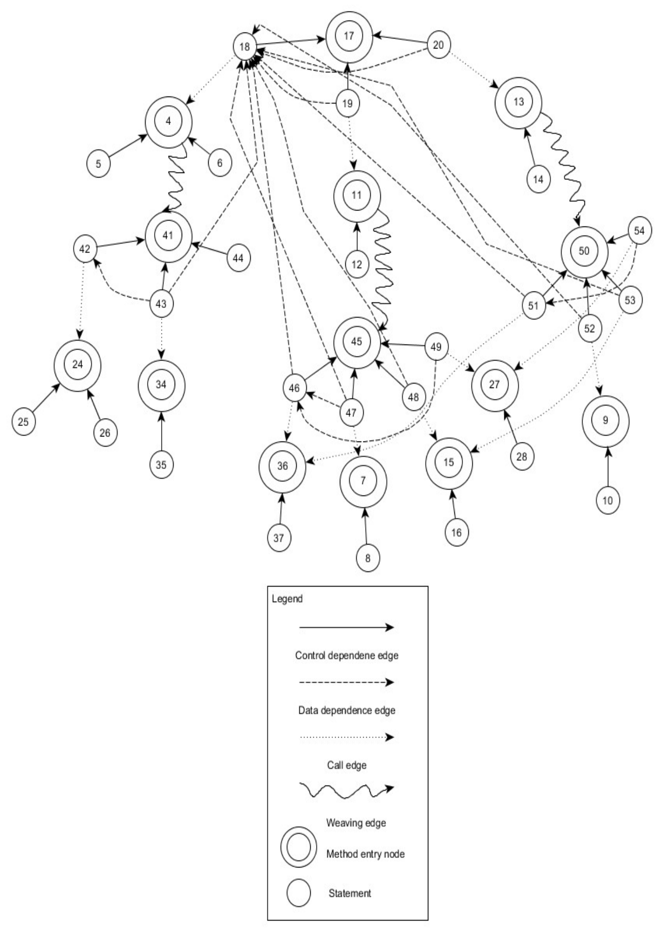


Fig. 3. Aspect-oriented statement level dependence graph of example program given in Fig. 2

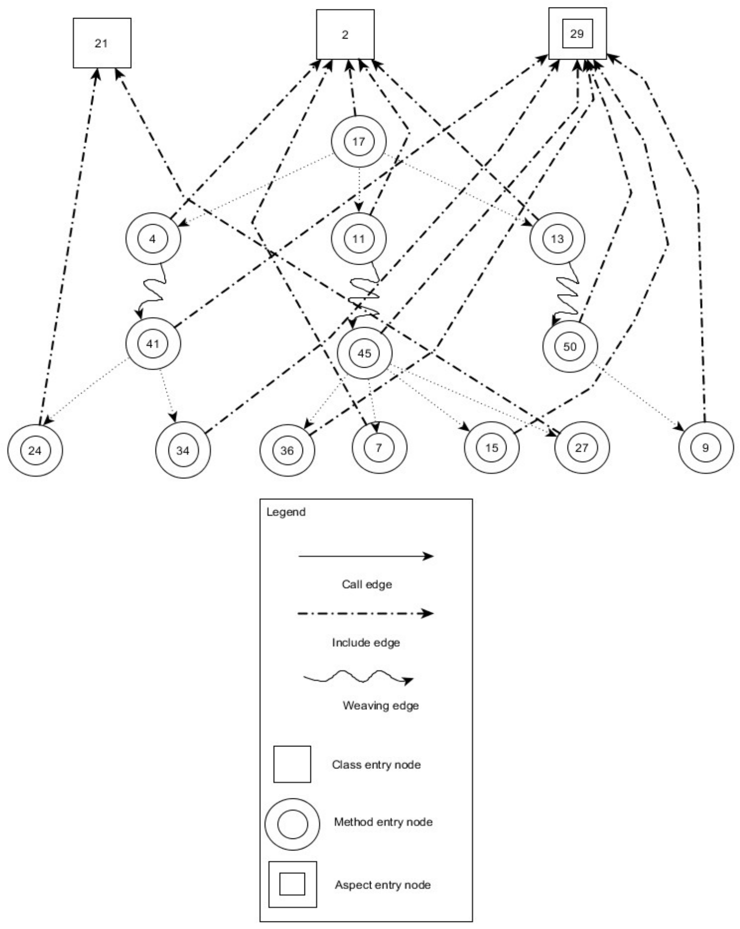


Fig. 4. Aspect-oriented method level dependence graph of example program given in Fig. 2

of dependencies between the nodes. The above two sets can be defined as follows:

$N1_d = \{n \mid n$ is the statement of the program $P_{ro}\}$.

$E1_d = \{e \mid e$ represents the dependency between n_1 and n_2, such that $n_1, n_2 \in N1_d\}$.

Aspect-oriented statement level graph contains all the executed statements, method entry nodes. It also contains edges to represents the data dependencies, control dependencies, call dependencies, weaving dependencies etc. AOSLDG of the example program in Fig.2 is given in Fig.3.

2) Aspect-oriented method level dependence graph: Aspect-oriented method level graph is an arc classified graph and can be defined as follows:

$Gh_{AOML} = \{N2_d, E2_d\}$

Where $N2_d$ is the set of nodes in the aspect-oriented method level graph and $E2_d$ is the set of edges depicting various types of dependencies between the nodes. The above two sets can be defined as follows:

$N2_d = \{n \mid n$ is the statement of the program $P_{ro}\}$.

$E2_d = \{e \mid e$ represents the dependency between n_1 and n_2, such that $n_1, n_2 \in N2_d\}$.

Aspect-oriented method level dependence graph contains method entry nodes, class entry nodes, and aspect-entry nodes. Besides these three nodes, it also represents the method to a class, method to aspect and method to method interdependencies.

3) Aspect Entry Node: Entry to an aspect is represented by Aspect entry node. We have proposed double square to distinguish the aspect entry node and class entry node. Class entry node is represented by a square.

4) Weaving edge: In our proposed graph, we have introduced a new edge called *weaving edge* to represent the weaving of an advice with the join point. We represent the weaving edge with a waved line.

AOMLDG of the example program in Fig.2 is given in Fig.4.

5) Aspect-oriented AC(Aspect-Class) weaving level dependence graph: Aspect-oriented AC weaving level dependence graph is an arc classified graph and can be defined as follows:

$Gh_{AOACWL} = \{N3_d, E3_d\}$

Where $N3_d$ is the set of nodes in the aspect-oriented method level graph and $E3_d$ is the set of edges depicting various types

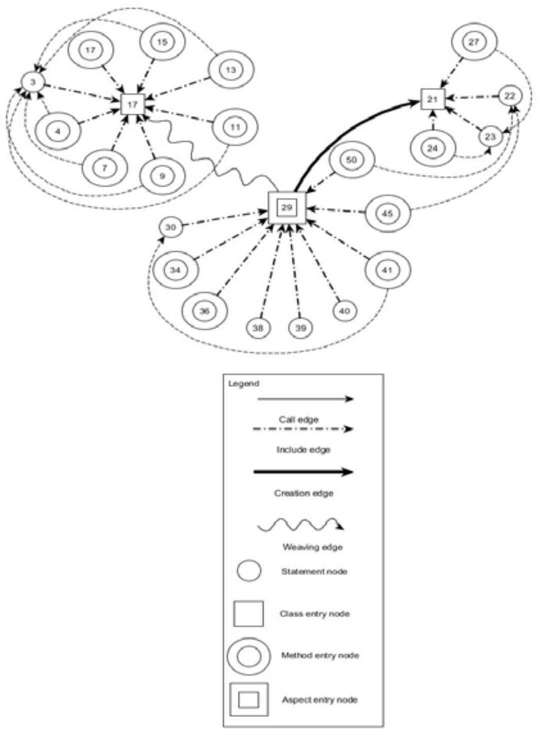


Fig. 5. Aspect-oriented AC weaving level dependence graph of example program given in Fig. 2

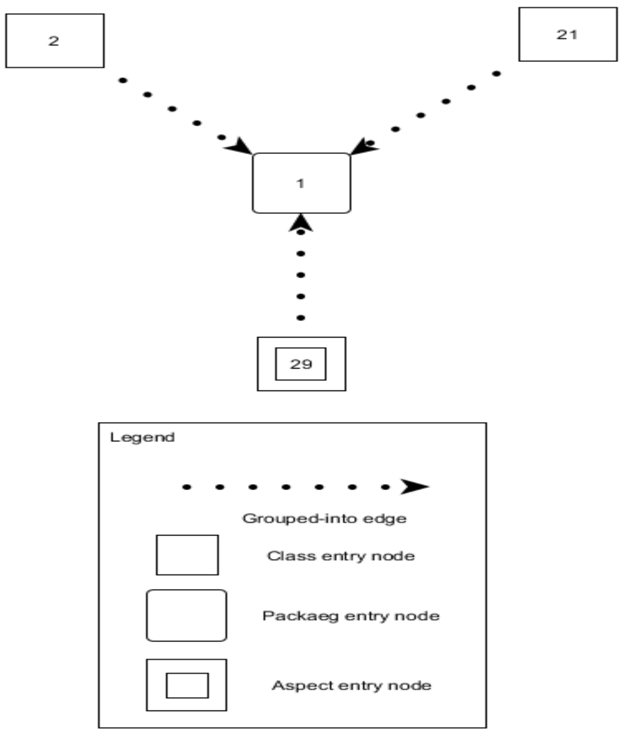


Fig. 6. Aspect-oriented package level dependence graph of example program given in Fig. 2

of dependencies between the nodes. The above two sets can be defined as follows:
$N3_d = \{n$ / n is the statement of the program $P_{ro}\}$.
$E3_d = \{e$ / e represents the dependency between n_1 and n_2, such that $n_1, n_2 \in N3_d\}$. AOACWLDG contains the class entry nodes, aspect entry nodes, nodes representing the data members, nodes representing the member functions. It also reflects the class to class and aspect to class inter-dependencies.

AOACWLDG of the example program in Fig.2 is given in Fig.5.

6) Aspect-oriented package level dependence graph: Aspect-oriented package level dependence graph is an arc classified graph and can be defined as follows:
$Gh_{AOPL} = \{N4_d, E4_d\}$
Where $N4_d$ is the set of nodes in the aspect-oriented package level graph and $E4_d$ is the set of edges depicting various types of dependencies between the nodes. The above two sets can be defined as follows:
$N4_d = \{n$ / n is the statement of the program $P_{ro}\}$.
$E4_d = \{e$ / e represents the dependency between n_1 and n_2, such that $n_1, n_2 \in N4_d\}$.

AOPLDG contains aspect entry nodes, class entry nodes, package entry nodes and their interdependencies.

AOPLDG of the example program in Fig.2 is given in Fig.5.

III. Proposed Algorithms

This section presents our proposed algorithms. We have divided this section into two subsections. The first subsection presents the proposed algorithm to construct the different aspect-oriented level dependence graphs. The second subsection presents the algorithm, which will be applied on the constructed graphs constructed to compute the dynamic slice of an aspect-oriented program. We named our algorithm *aspect-oriented reverse hierarchical slicing algorithm*.

After presenting the algorithms, we have given the overview of the aspect-oriented reverse hierarchical slicing algorithm. Then, we explain the working of our algorithm followed by the correctness and complexity analysis.

A. Algorithm for constructing different aspect-oriented level graphs

Before computing the dynamic slice of an aspect-oriented program, we represent it using different level graphs. We have identified four different levels of constructing the graphs and here we present the algorithms of constructing the aspect-oriented level graphs.

Let us consider that there are *n* number of statements, *l* number of methods, *q* number of classes/aspects. Among these, there are *q1* number of classes and *q2* number of aspects present. Such that *q1+q2=q*. And *r* numbers of packages are there.

1) Algorithm for constructing aspect-oriented statement level dependence graph: Considering AOS_G is a two-dimensional array storing the information of the aspect-oriented statement level dependence graph.

```
1) Begin
2) For each statement draw a circle
3) For each method entry draw double circle
4) For i = 1 to n
   For j = 1 to l
      a) if $AOS_G[i]$ control dependent on $AOS_G[j]$
         Add control dependent edge from $AOS_G[i]$ to $AOS_G[j]$
      b) if $AOS_G[i]$ calls $AOS_G[j]$
         Add call edge from $AOS_G[i]$ to $AOS_G[j]$
5) For i = 1 to n
   For j = 1 to n
      a) if $AOS_G[i]$ data dependent on $AOS_G[j]$
         Add data dependent edge from $AOS_G[i]$ to $AOS_G[j]$
6) For i = 1 to l
   For j = 1 to l
      a) if $AOS_G[i]$ control dependent on $AOS_G[j]$
         Add control dependent edge from $AOS_G[i]$ to $AOS_G[j]$
7) End
```

2) Algorithm for constructing aspect-oriented method level dependence graph: Considering AOM_G is a two-dimensional array storing the information of the aspect-oriented method level dependence graph.

```
1) Begin
2) For each class draw a square
3) For each aspect draw double square
4) For each method entry draw double circle
5) For i = 1 to l
   For j = 1 to q1
      a) if $AOM_G[i]$ is member function of $AOM_G[j]$
         Add include edge from $AOM_G[i]$ to $AOM_G[j]$
6) For i = 1 to l
   For j = 1 to q2
      a) if $AOM_G[i]$ is a member function or advice of $AOM_G[j]$
         Add include edge edge from $AOM_G[i]$ to $AOM_G[j]$
7) For i = 1 to l
   For j = 1 to l
      a) if $AOM_G[i]$ calls $AOM_G[j]$
         Add call edge from $AOM_G[i]$ to $AOM_G[j]$
      b) if $AOM_G[i]$ weaved with $AOM_G[j]$
         Add weaving edge from $AOM_G[i]$ to $AOM_G[j]$
8) End
```

3) Algorithm for constructing aspect-oriented AC weaving level dependence graph: Considering $AOAC_G$ is a two-dimensional array storing the information of the aspect-oriented AC weaving level dependence graph.

```
1) Begin
2) For each class draw a square
3) For each aspect draw double square
4) For each method entry draw double circle
5) For each statement draw a circle
6) For i = 1 to n
   For j = 1 to q1
      a) if $AOAC_G[i]$ is a data member of $AOAC_G[j]$
         Add include edge from $AOM_G[i]$ to $AOM_G[j]$
7) For i = 1 to l
   For j = 1 to q1
      a) if $AOAC_G[i]$ is a member function of $AOAC_G[j]$
         Add include edge edge from $AOAC_G[i]$ to $AOAC_G[j]$
8) For i = 1 to n
   For j = 1 to q2
      a) if $AOAC_G[i]$ is a member function of $AOAC_G[j]$
         Add include edge from $AOAC_G[i]$ to $AOAC_G[j]$
9) For i = 1 to l
   For j = 1 to q2
      a) if $AOAC_G[i]$ is a member function of $AOAC_G[j]$
         Add include edge from $AOAC_G[i]$ to $AOAC_G[j]$
10) For i = 1 to q1
    For j = 1 to q1
      a) if $AOAC_G[i]$ instantiate object of $AOAC_G[j]$
         Add creation edge from $AOAC_G[i]$ to $AOAC_G[j]$
11) End
```

4) Algorithm for constructing aspect-oriented package level dependence graph: Considering AOP_G is a two-dimensional array storing the information of the aspect-oriented package level dependence graph.

```
1) Begin
2) For each class draw a square
3) For each aspect draw double square
4) For each package draw a rounded square
5) For i = 1 to q1
   For j = 1 to r
      a) if $AOP_G[i]$ belongs to $AOP_G[j]$
         Add grouped-into edge from $AOP_G[i]$ to $AOP_G[j]$
6) For i = 1 to q2
   For j = 1 to r
      a) if $AOP_G[i]$ belongs to $AOP_G[j]$
         Add grouped-into edge from $AOP_G[i]$ to $AOP_G[j]$
7) End
```

5) Aspect-oriented reverse hierarchical dynamic slicing algorithm: In this section, we present our proposed algorithm to compute the dynamic slices of aspect-oriented programs. We have used the term like I_1, I_2, I_3, I_4 to hold the intermediate dynamic slice at different levels respectively.

```
1) MainAlgo()
2) Begin
3) Input an aspect-oriented program $P_{ro}$
4) Run the program with the required input values
5) Get the execution trace
6) Based on the execution trace construct the AOSLDG
7) Fetch the slicing node
8) From the slicing node if outgoing control dependence edge exist
   Traverse through the outgoing control dependence edge and list the reached node in $\boldsymbol{l}_1$
9) For each outgoing call edge, call ForOutgoingCallEdge()
10) For each outgoing weaving edge, call ForoutgoingWeavingEdge()
11) For each incoming weaving edge, call ForIncomingWeavingEdge()
12) Do the following steps to find set $\boldsymbol{l}_2$
    a) Construct the AOMLDG
    b) For i=1 to $|\boldsymbol{l}_1|$
       i) if $\boldsymbol{l}_1$ is a call node or method entry node
          Traverse through the outgoing all edge or outgoing weaving edge or incoming weaving edge and list the reached node in set $\boldsymbol{l}_2$
       ii) Update $\boldsymbol{l}_2 = \boldsymbol{l}_1[\boldsymbol{i}] \cup \boldsymbol{l}_2$
       iii) EndIf
       iv) EndFor
13) Do the following steps to find set $\boldsymbol{l}_3$
    a) Construct the AOACELDG
    b) For i=1 to $|\boldsymbol{l}_2|$
       i) From $\boldsymbol{l}_2[\boldsymbol{i}]$ traverse through the include edge and reach the node $n_5$
       ii) If !(search($\boldsymbol{l}_3$, $n_5$)==true)
           Update $\boldsymbol{l}_3 = \boldsymbol{l}_3 \cup \boldsymbol{l}_6$
       iii) EndIf
       iv) EndFor
14) Do the following steps to find set $\boldsymbol{l}_4$
    a) Construct the AOPLDG
    b) For i=1 to $|\boldsymbol{l}_3|$
       i) $\boldsymbol{l}_3[\boldsymbol{i}]$ is a class entry node or aspect entry node then
          Traverse through the outgoing grouped-into edge and list the reached node $n_7$
       ii) If !(search($\boldsymbol{l}_4$, $n_7$)==true)
           Update $\boldsymbol{l}_4 = \boldsymbol{l}_4 \cup \boldsymbol{l}_7$
       iii) EndIf
       iv) EndFor
15) $AO\_Dslice = \boldsymbol{l}_1 \cup \boldsymbol{l}_2 \cup \boldsymbol{l}_3 \cup \boldsymbol{l}_4$
16) For i=1 to $|AO\_Dslice|$
    a) Output $AO\_Dslice[\boldsymbol{i}]$
    b) EndFor
17) End
```

ForOutGoingCallEdge()

```
1) Begin
2) If from the node outgoing all edge exist
   Traverse and list the reached node in set $\boldsymbol{l}_1$
3) If incoming control dependence edge exist
   Traverse and list the reached node in set $\boldsymbol{l}_1$
4) For each reached node if outgoing data dependence edge exist
   Traverse and list the reached node in set $\boldsymbol{l}_1$
5) For each node repeat ForOutGoingCallEdge()
6) End
```

ForOutGoingWeavingEdge()

```
1) Begin
2) Traverse through the outgoing weaving edge and list the reached node in set $\boldsymbol{l}_1$
3) If incoming control dependence edge exist to reached node
   Traverse and list the reached node in set $\boldsymbol{l}_1$
4) For each node list node call ForOutGoingCallEdge()
5) End
```

ForIncomingWeavingEdge()

```
1) Begin
2) If incoming weaving edge exist
   Traverse through the incoming weaving edge and list the reached node in set $\boldsymbol{l}_1$
3) If incoming control dependence edge exist
   Traverse and list the reached node in set $\boldsymbol{l}_1$
4) For each node repeat ForOutGoingCallEdge()
5) End
```

B. Overview of aspect-oriented reverse hierarchical slicing algorithm

Aspect-oriented reverse hierarchical dynamic slicing algorithm computes the dynamic slices of an aspect-oriented program in four different levels. In this section, we present the overview of our proposed algorithm. Our algorithm first inputs the program. Then the program is being executed with required input values. After the execution, we get the *execution trace* of the program under consideration. Based on the execution trace, we construct the AOSLDG (Aspect-oriented statement level dependence graph). In this level, we fetch the *slicing node*, on which the dynamic slice has to be computed. We named the dynamic slice computed at this level *AOSL_Dslice* (Aspect-oriented statement level dynamic slice).This dynamic slice is stored in an array named as $\boldsymbol{l}_1$.

To compute the AOSL_Dslice, our algorithm defines three different subroutines named as *ForIncomingWeavingEdge()*, *ForOutGoingWeavingEdge()*, *ForOutGoingCallEdge()*. Which are called in three different context.

After the slicing node is being fetched, our algorithm traverses from the slicing node through the *outgoing control dependence edge* and list the reached nodes. Then our algorithm traverses if any outgoing *data dependence edge* exist and list the reached nodes in the array $\boldsymbol{l}_1$.

Then our algorithm verifies for different condition like if any *outgoing call edge* exist or if any *incoming weaving edge* exist or if any *outgoing weaving edge* exist from the slicing node and recursively call the three specified subroutine to list the dependent nodes in the array $\boldsymbol{l}_1$.

Then our algorithm proceeds to compute the second level dynamic slice named as *AOML_Dslice*. For this, our algorithm first constructs the AOMLDG (Aspect-oriented method level dependence graph). Already array $\boldsymbol{l}_1$ contains the nodes belongs to the aspect-oriented statement level dynamic slice. Now our algorithm starts traversing from the method entry nodes of array $\boldsymbol{l}_1$ through the *call edges* or *weaving edge* and reached other dependent method entry nodes and list those nodes in the array $\boldsymbol{l}_2$. Which constitute the aspect-oriented method level dynamic slice.

To compute the *AOCL_Dslice* (aspect-oriented class level dynamic slice), our algorithm first constructs the AOACWLDG (Aspect-oriented aspect-class weaving level dependence graph). From array $\boldsymbol{l}_2$ our algorithm starts traversing from the method entry nodes through the *include edge* to get its respective class or aspect node. Then it also traverses through the *data dependence edges* to get some of the nodes on which method entry nodes are data dependent.Basically in this level our algorithm find out the classes or aspect and their data members, those contribute towards the aspect-oriented class level dynamic slice and save those nodes in the array $\boldsymbol{l}_3$.

Finally our algorithm computes the *AOSL_Dslice* (aspect-oriented package level dynamic slice). To compute the slice at package level, our algorithm starts traversing from the class entry nodes or the aspect entry node from the array $\boldsymbol{l}_3$ through the *outgoing grouped-into edge* and list the package entry node in array $\boldsymbol{l}_4$.

After getting the distinct four set $\boldsymbol{l}_1$, $\boldsymbol{l}_2$, $\boldsymbol{l}_3$, $\boldsymbol{l}_4$, our algorithm does the union of those sets to compute the *AO_Dslice* (aspect-oriented dynamic slice) of an aspect-oriented program.

C. Working of the proposed algorithm

We illustrate the working of our algorithm with the help an example. Let us consider an aspect-oriented program given in Fig.2. At first we run the program and get the execution trace. Based on it the aspect-oriented statement level dependence graph is being constructed, which is shown in Fig.3. Now, we fetch the slicing node say node 18, on which the dynamic slice has to be computed. Our algorithm traverses from the node 18 through the *outgoing control dependence edge* and reach the reached node 17. Node 17 is stored in $\boldsymbol{l}_1$.

From node 18, we traverse through the *outgoing call edge* and reach the node 4. Here for addressing a *outgoing call edge* our algorithm calls the subroutine ForOutGoingCallEdge(). It collect all the nodes which control dependent on node 4. Node 4, 5, 6 are now saved in $\boldsymbol{l}_1$. Then our algorithm calls the subroutine ForOutGoingWeavingEdge() and reaches the node 41. Then our algorithm traverses the incoming control dependent edges and lists the reached nodes 42, 43, 44. From node 43 there are *data dependence edges* to nodes 42 and 1, which have to be considered, but already $\boldsymbol{l}_1$ contains nodes 42 and 18. Again from each reached node, we verify for any *outgoing call edge, outgoing weaving edge* or *incoming weaving edge*. From node 42 and 43 there are two *outgoing call edges* for which subroutine ForOutGoingCallEdge() is being called and we get the nodeS 24, 34 and node 25, 26, 35, which are control dependent on nodes 24, 34.

Finally staring with node 18 as slicing node, we have nodes 17, 18, 4, 5, 6, 41, 42, 43, 44, 24, 34, 25, 26, 35 in $\boldsymbol{l}_1$.

Then our algorithm constructs the aspect-oriented method level dependence graph. From $\boldsymbol{l}_1$ we get the method entry nodes such as node 17, node 4, node 41, node 24, node 34 and traverse through the *call edge or weaving edge* to get the dependent nodes in the aspect-oriented method level dependence graph, and save them in $\boldsymbol{l}_2$. Now $\boldsymbol{l}_1$ contains nodes 17, 4, 41, 24, 34.

To get $\boldsymbol{l}_3$, our algorithm constructs the aspect-oriented aspect-class weaving level dependence graph. From the method entry node collected in $\boldsymbol{l}_2$, our algorithm traverses through the *include edge* to get the class/aspect entry nodes. From nodes 17 and 4 our algorithm moves through the *include edge* and reach the node 2. From node 41 and 34 through *include edge* our algorithm reach node 29. From node 24 our algorithm traverses through the *include edge* and reach node 21.

Now, from the nodes listed in $\boldsymbol{l}_2$, our algorithm traverses through the *outgoing data dependence edge* to list the nodes 3, 30, 23. Our algorithm listed node 21, 2, 29, 3, 30, 23 in $\boldsymbol{l}_3$.

To get the aspect-oriented package level dependence graph, our algorithm constructs the aspect-oriented package level dependence graph. From the class entry nodes or aspect entry nodes i. e. from 21, 2, 29 our algorithm moves through the *outgoing grouped-into edge* and lists the reached node 1 in $\boldsymbol{l}_4$, which is a package entry node.

Finally our algorithm performs the union of $\boldsymbol{l}_1$, $\boldsymbol{l}_2$, $\boldsymbol{l}_3$, $\boldsymbol{l}_4$ and save the result in *AO_Dslice*. Now *AO_Dslice* contains nodes 17, 18, 4, 5, 6, 41, 42, 43, 44, 24, 34, 25, 26, 35, 21, 2, 29, 3, 30, 23, 1, which constitutes the dynamic slice of our example program under consideration with respect to statement number 18.

D. Correctness of the algorithm

This section presents the correctness of the proposed algorithm.

Theorem: Aspect-oriented reverse hierarchical dynamic slicing algorithm always computes correct dynamic slices with respect to a given slicing criterion.

Proof: We will prove the correctness of aspect-oriented reverse hierarchical dynamic slicing algorithm using the mathematical induction. Our algorithm computes the dynamic slices of an aspect-oriented program by analyzing the dependence relation between the statements of the program under consideration. Aspect-oriented dynamic slice of a statement S_i will certainly contain statements, that are having dependence relation with statement S_i. Let us consider our aspect-oriented program contains only one statement S_1. So *AO_Dslice* of S_1 will contain S_1, because S_1 is dependent on S_1. The computed dynamic slice for first statement S_1 is correct.

Suppose there are two statements S_1, S_2 presents in our aspect-oriented program. So dynamic slice at S_2 will contain S_1, if S_2 is dependent on S_1. Otherwise, it will not be included. In this context, we can argue that dynamic slice of 2nd statement is also correct.

Like this, we can prove that dynamic slices of statements prior to S_u is correct. According to the definition aspect-oriented dynamic slice of statement S_u, only contains those statements, on which statement S_u is dependent. For all statements prior to statement S_u the computed dynamics are correct. Hence, the dynamic slice of statement S_u is correct. This establishes the correctness of our algorithm.

E. Complexity Analysis of Aspect-oriented reverse hierarchical dynamic slicing algorithm

We analyze the complexity of our algorithm in terms of space and time.

Space Complexity: Our proposed algorithm works on four different level graphs to compute the dynamic slice of aspect-oriented programs. The space requirement for those level graphs is :$O(n^2)$ + $O(l^2)$ + $O(q^2)$ +$O(r^2)$.

Here *n, l, q, r* are the number of nodes in aspect-oriented statement level, aspect-oriented method level, aspect-oriented aspect-class weaving level and aspect-oriented package level graph respectively. Where *n* suppose to be greater than *l, q, r*. Hence, we consider the space requirement of these graphs is $O(n^2)$.

Time Complexity: The time complexity of our algorithm is equal to the sum of the time required to construct the different level graphs and the time required to traverse the graph. We have computed the time complexity at a different level. For an aspect-oriented statement level graph having *n* numbers of the node. The time required to construct the graph is $O(n^2)$. And the time requires traversing the graph is $O(E_1 n)$. Where E is the total numbers of edges in the graph. Substituting the value of E, we will get the time complexity at aspect-oriented statement level is: $O[(n-1)n]$ which is equal to $O(n^2)$. Similarly we compute the time complexity for other levels and get the total time complexity: $O(n^2)$ + $O(l^2)$ + $O(q^2)$ +$O(r^2)$. Assuming *n* is greater than the *l, q, r*, the time complexity of our algorithm is $O(n^2)$.

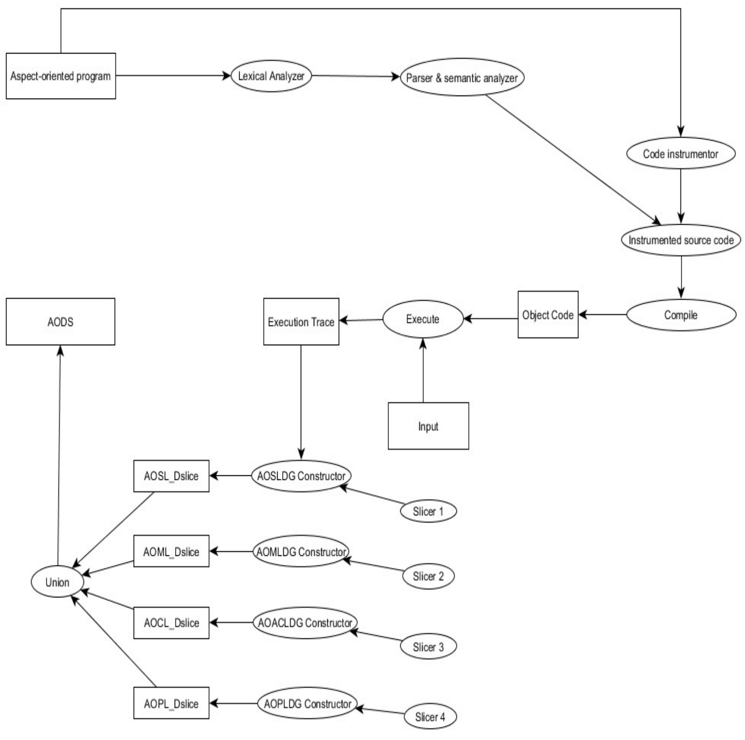


Fig. 7. Aspect-oriented dynamic slicer

IV. TOOL ARCHITECTURE OF ASPECT-ORIENTED DYNAMIC SLICER

In this section, we briefly explain the tool architecture of *AODS*. AODS is an aspect-oriented dynamic slicer, which may be used for implementing our proposed algorithm.
Aspect-oriented dynamic slicer accepts an aspect-oriented program as input and generates aspect-oriented dynamic slice of the inputted program as the output.
Once the aspect-oriented program is accepted by our proposed tool, lexical analyzer starts identifying the tokens, in the program. We have parser and semantic analyzer as a component, which will verify the grammar of the inputted program.
Then the aspect-oriented program is instrumented by a code instrument. After that, the instrumented code is being submitted to the compiler for generating the object code. Now the inputted program is executed with the input given by the user and we get the execution trace. Based on the execution trace different level dependence graphs are being constructed. Once a level dependence graph is being constructed, the slicer is applied to it to get the dynamic slice. After the dynamic slice is computed at different levels, all are integrated to compute the dynamic slice of an aspect-oriented program.

We presented the architecture of our tool in Fig.7. Which accept an aspect-oriented program as input, computes the dynamic slice by constructing different level graphs and applying the aspect-oriented reverse hierarchical dynamic slicing algorithm.

V. Comparison with related work

We have compared our approach with some other related approach. In this section, we present few of those comparative study done by us. Mohapatra et al. [5] in their approach proposed DADG (Dynamic Aspect-oriented Dependence Graph), which represents all the statements of an aspect-oriented program in a single graph. The proposed graph may become unmanageable if the program under consideration is large.

Zhao et al. [6] proposed their approach to computing the static slice of aspect-oriented programs. They introduced ASDG, which consists of three parts: System Dependence graph for non-aspect code, a group of dependence graphs for aspect code and some additional arcs to make connections between the graph for aspect and nonaspect code. The ASDG may complicate for large programs. Hence, their proposed algorithm will take more time to traverse and compute the dynamic slice.

Ahmed et al. [7] proposed Dependence Flow Graph (DFG) to represent the aspect-oriented programs. They introduced Static Dependence Slicing Tool to compute the static slice of the aspect-oriented program. As they compute the static slice, the slice may be not useful in some of the interactive software engineering applications.

Ray et al. [8] proposed AOSG (Aspect System Dependence Graph) as the intermediate program representation and computes the dynamic slice of an aspect-oriented program by marking and unmarking the edges of the graph during the run time. As this approach considered the complete program to construct the AOSG. The AOSG may complicate for large programs, hence the proposed algorithm will take more time in marking and unmarking the edges of the graph.

But in our approach, we introduced level graphs, which represents the program level wise. Hence, it seems to be an efficient intermediate representation as compared to above-mentioned approaches. Our proposed AORHDS algorithm computes dynamic slice of an aspect-oriented program starting from the statement level to package level. It can generate the intermediate dynamic slice at different levels. It will take less time in traversing the graphs and computing the dynamic slices.

VI. Conclusion and Future work

This paper proposed an algorithm which can work on different level graphs to compute the *dynamic slices* of an aspect-oriented program. This intermediate representation is efficient because it minimizes our area of focus. We will be able to compute the dynamic slice at different levels like statement level, method level, class level and package level. This can be a simplified approach to computing the dynamic slices of aspect-oriented programs.The *aspect-oriented reverse hierarchical dynamic slicing* algorithm proposed in this paper accepted a single slicing criterion (slicing node) to compute the slice. The resulting slice can be used in various software engineering applications such as debugging and testing etc.
In future, we will focus on finding the *dynamic hierarchical slices* of feature-oriented programs and web-based programs.

References

[1] Beszedes, Arpad, Gergely, "Graph-less dynamic dependence-based dynamic slicing algorithm," *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pp. 21-30, 2006.

[2] Wang, T., Roychoudhury, A., "Dynamic slicing on Java bytecode traces", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2008.

[3] Zhang, X., Gupta, R., Zhang, Y.,"Precise dynamic slicing algorithms", *Software Engineering, 2003. Proceedings. 25th International Conference on Software Enginering*, pp. 319–329, 2003.

[4] Mohapatra, D. P., Mall, R., Kumar, R., "An Overview of Slicing Techniques for Object-Oriented Programs", *Informatica*, pp. 253–277, Vol. 30, 2006.

[5] Mohapatra, D. P., Sahu, M., Kumar, R., and Mall, Rajib,"Dynamic slicing of aspect-oriented programs", *Informatica*, vol. 32, number. 3, 2008.

[6] Zhao, J., "Slicing aspect-oriented software", *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pp. 251–260, 2002.

[7] Ahmad, S., Ghani, A. Sani, F. Atan, R.,"Slicing aspect oriented program using dependence flow graph for maintenance purpose", *Proceedings of Regional Conference on Knowledge Integration in ICT*, pp. 236–241, 2010.

[8] Ray, A., Mishra, S., Mohapatra, D., P., "A Novel Approach for Computing Dynamic Slices of Aspect-Oriented Programs", *arXiv preprint arXiv:1403.0100*, 2014.

[9] Li, B., Fan, X., "JATO: Slicing Java program hierarchically", *TUCS Techinical Reports*,number. 416, 2001.

[10] Hammer, C., Snelting, G., "An improved slicer for Java", *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 17–22, 2004.

[11] Panda, S., and Mohapatra, D. P., "Application of Hierarchical Slicing to Regression Test Selection of Java Programs", 2013.

[12] Singh, J., Mohapatra, D. P., "A unique aspect-oriented program slicing technique", *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*, pp. 159–164, 2013.