Available online at www.sciencedirect.com

ScienceDirect

Journal of Electrical Systems and Information Technology 5 (2018) 562–575

www.elsevier.com/locate/jesit

Aspect oriented programs: Issues and perspective

Sk. Riazur Raheman [a,*], Hima Bindu Maringanti [b], Amiya Kumar Rath [c]

[a] *Ph.D. Scholar, Computer Science & IT, North Orissa University, Baripada, Odisha, 757003, India*
[b] *Professor & Head, Dept. of Computer Application, North Orissa University, Baripada, Odisha 757003, India*
[c] *Professor, Dept. of Computer Science & Engineering, VSSUT, Burla, Odisha 768018, India*

Received 2 February 2017; accepted 11 June 2017
Available online 8 February 2018

Abstract

Aspect oriented programming (AOP) helps programmers for separating crosscutting concerns. All programming methodologies support split up and encapsulation of concerns. In object-oriented programming (OOP) crosscutting aspects are distributed among objects. It is hard to attain crosscutting in OOP as it is scattered in different objects. In AOP crosscutting concerns are addressed using one entity called aspect. This paper discusses varieties of existing slicing techniques of AOP. Also, we discuss a novel method to calculate dynamic slice of AOP. To represent AOP Aspect Oriented System Dependence Graph (AOSDG) is used. The complexity of this new approach is equal or improved as related to certain prevailing approaches.

Keywords: Aspect; AOP; Slicing; Crosscutting; AOSDG

1. Introduction

Program slicing is a decomposition procedure presented by Weiser in 1979 (Tip, 1995; Weiser, 1984; Horwitz et al., 1990; Sahu and Mohapatra, 2009; Raheman et al., 2013; Raheman et al., 2011). A slice is the extract from program statements that disturb the values calculated at certain point and that point is known as *slicing condition.* Slicing condition comprises of <S, V>, where S represents the statement in the program and V represents variable (Korel et al., 1988; Horwitz et al., 1990; Raheman et al., 2013; Mohapatra et al., 2006; Raheman et al., 2011). There are two techniques for calculating slice. The first method is Weiser's method and the second one is slicing through graph reachability. In the first approach, slice is calculated by figuring successive groups of related variables for every node in Control Flow Graph (CFG). In another method slicing is separated into two phases, initially dependency graph

* Corresponding author at: Ph.D. Scholar, Computer Science & IT, North Orissa University, Baripada, Odisha 757003, India.
E-mail addresses: skriazur79@gmail.com (Sk.R. Raheman), mhimabindu@yahoo.com (H.B. Maringanti), amiyaamiya@rediffmail.com (A.K. Rath).

Peer review under the responsibility of Electronics Research Institute (ERI).

Production and hosting by Elsevier

https://doi.org/10.1016/j.jesit.2017.06.003

of the program is created and then applying a procedure on CFG to get slices (Tip, 1995; Weiser, 1984; Agrawal and Horgan, 1990; Mohapatra et al., 2004; Korel and Laski, 1990; Raheman et al., 2011).

In OOP it is very difficult to achieve crosscutting concern, as crosscutting aspects are distributed among different objects. AOP is a novel knowledge for splitting crosscutting concerns. In AOP problems are disintegrated into a single section called aspect, that is easier to develop and maintain (Zhao, 2001; Ray et al., 2013; Raheman et al., 2014, 2011; Katti et al., 2012; Robinson, 2006). Features of AOP like, aspects, point-cut, advice and join points are different from procedural and OOP languages (Raheman et al., 2013, 2011; Ishio et al., 2003b; Sikka et al., 2013). Hence, it involves developing effective slicing procedures as well as appropriate intermediate representations for calculating slices in AOP (Raheman et al., 2013, 2011; Jain et al., 2013).

2. Features of AspectJ

AspectJ developed by Xerox Parc is used as an aspect weaver for Java. AspectJ adds certain features to Java as follows (Ray et al., 2013; Raheman et al., 2013, 2014, 2011; Ishio et al., 2003b; Kiczales et al., 2001).

- Aspects,
- Join points,
- Point-cut,
- Advice,
- Introduction,
- Point-cut designator.

2.1. Aspects

Aspects in AOP are same as like objects in OOP. Aspect contains all the real world problems and collects all the functionality within it.

```
aspect TestAspect
{

}
```

In the above example "TestAspect" is declared as a new aspect (Ishio et al., 2003a; Raheman et al., 2013, 2014, 2011; Evertsson, 2002; Kiczales et al., 2001).

2.2. Join points

Aspect and non-aspect codes in AOP are linked using join points. In AOP codes are separated as aspect and non-aspect. Non-aspect codes are simple java construct and aspect codes are the crosscutting concerns. In an AOP method execution, method call and method reception act as join points (Ray et al., 2013; Mohapatra et al., 2008; Raheman et al., 2013, 2014, 2011; Evertsson, 2002; Pollice, 2004; Sahu and Mohapatra, 2007). In Fig. 1 at statements 2, 3 and 4, we are calling the method Factnum(), Root() and Permutation() these methods call can be used as join points.

2.3. Point-cut

During execution join points are matched using point-cut. Advice is linked with point-cut and executed when join point is matched by the point-cut (Ray et al., 2013; Mohapatra et al., 2008; Raheman et al., 2013, 2014, 2011; Evertsson, 2002; Pollice, 2004). In Fig. 1 at statement 14 point-cut FactnumOp collects every call to Factnum().

2.4. Advice

Advice is associated with point-cuts. Advice contains certain task that is to be performed when a point-cut is found. It defines the crosscutting behavior at join points (Ray et al., 2013; Mohapatra et al., 2008; Raheman et al., 2013, 2014, 2011; Evertsson, 2002; Ishio et al., 2004). In AspectJ there are three forms of advices:

Non aspect code

```
Import java.util.*;
public class MathLib
{
    1.  public static void main(String args [])
        {
    2.      System.out.println("3!  is  "  +
            Factnum(3));
    3.      System.out.println("Root of 6 is " +
            Root(6));
    4.      System.out.println("P(3,2)  is  "  +
            Permutation(3,2));
        }

    5.  public static long Factnum(long  var)
        {
    6.      if (var==1)
    7.      return 1;
              else
    8.      return var*Factnum(var-1);
        }

    9.  public static long Root(long var)
        {
    10.     long root=Math.sqrt(var);
    11.     return root;
        }

    12. public static long Permutation(long var,
        long a)
        {

    13.     return Factnum(var)/Factnum(var-a);
        }

}
```

Aspect code

```
public aspect Test
{
14.   pointcut FactnumOp(long var) : (call(*
      *.Factnum(long)) && args(var);
15.   before (long var): FactnumOp(var)
      {
16.       System.out.println("Finding  the
          Factorial" +var);
      }
17.   after(long var) returning (long result):
      FactnumOp(var)
      {
18.       System.out.println("Showing  the
          result of factorial for" + var);
      }
19.   pointcut RootOp(long var) : (call(*
      *.Root(long))) && args(var);
20.   before (long var): RootOp(var)
      {
21.       System.out.println("Finding  the
          Root for " +var);
      }
22.   after(long var) returning (long result):
      RootOp(var)
      {
23.       System.out.println("Showing  the
          result of Root for" + var);
      }
24.   pointcut PermutationOp(long var,long a)
      : (call(* *.Permutation(long, long)) &&
      args(var , a);
25.   before  (long  var,  long  a):
      PermutationOp(var, a)
      {
26.       System.out.println("Finding  the
          Permutation for " + var +"  " +a);
      }
27.   after(long var, long a) returning (long
      result): PermutationOp(var, a)
      {
28.       System.out.println("Showing  the
          result of Permutation of" + var + "  "
          +a);
      }
}
```

Fig. 1. Aspect program to find the factorial, root and permutation of a number.

- *after*,
- *before*,
- *around*.

After advice execution starts after the join point. An after advice is defined in Fig. 1 at statement 17. *Before* advice execution starts before the join point. In the AOP given in - Fig. 1, statement 15 represents a before advice. *Around* advice runs as the join point reached (Ray et al., 2013; Mohapatra et al., 2008; Raheman et al., 2013, 2014, 2011; Evertsson, 2002).

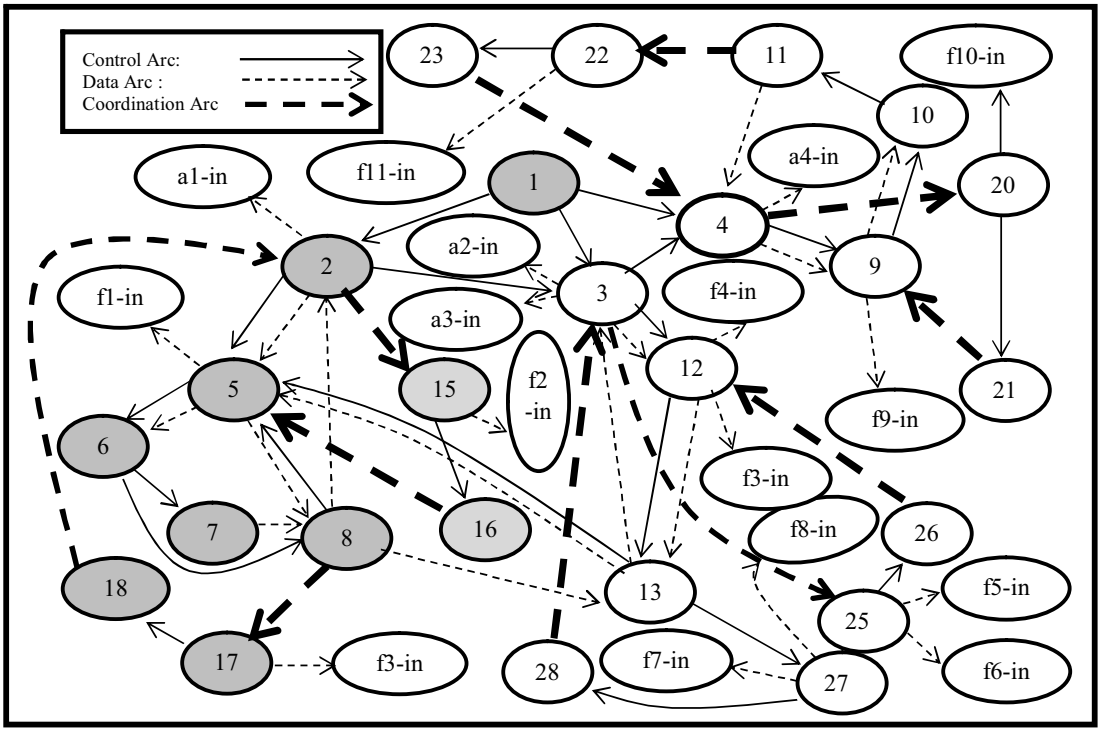

Fig. 2. ASDG of the program given in Fig. 1.

2.5. Introduction

Methods, fields and interfaces are added to an existing classes using introduction. It can be *private or public* (Mohapatra et al., 2008). A *private* introduction is accessible within the aspect that defines it, where as a *public* introduction is accessible by any code (Raheman et al., 2013, 2014; Sahu and Mohapatra, 2007; Kiczales et al., 2001).

2.6. Point-cut designator

Point-cut designator counterparts join points at runtime. All the join points are identified by point-cut designator. In Fig. 1 at statement 14 the *call (Factnum(long)) is a* point-cut designator (Raheman et al., 2014; Kiczales et al., 2001).

3. Aspect oriented program

An AspectJ program is separated into *Base Code* and *Aspect Code.* **Base code** contains class and additional Java constructs and **Aspect code** includes crosscutting concerns (Raheman et al., 2013, 2014). Fig. 1 represents an aspect oriented program to find the Factorial, Permutation and Root of any number.

3.1. Existing slicing techniques of AOP

Zhao (2002) was the first to develop aspect oriented system dependence graph (ASDG). ASDG is created by joining the non-aspect and aspect code and using certain dependence arcs (Xu et al., 2005; Zhao, 2001; Mohapatra et al., 2008). To calculate the static slice, Zhao implemented Larsen and Harrold's slicing approach on ASDG (Larsen and Harrold, 1996; Raheman et al., 2014). SDGs therefore provide a concrete base for the more study of AOP.

Let's analyze the static slice of the AOP given in Fig. 1 for statement 2 using Zhao approach. The ASDG of the AOP is given in Fig. 2. By applying the Larsen and Harrold's two phase slicing approach the static slice comes as: {1, 2, 5, 6, 7, 8, 15, 16, 17, 18}. The slices are marked as bold nodes in Fig. 2.

To calculate the dynamic slice of AOP, Mohapatra et al. (2008) has also suggested an approach. They represented the AOP using Dynamic Aspect Oriented Dependence Graph (DADG) as intermediated program representation. The AOP will be executed for a specific input and the implementation history is outlined by a trace file. Based on the execution history DADG is constructed. Then, the DADG is traversed by either BFS or DFS based on the given criteria to get the slice. The approach is implemented on the AOP given in Fig. 1 to get the dynamc slice.

The execution trace of the AOP given in Fig. 1: (with respect to statement 2)

1(1)	: public static void main (String args [])
2(1)	: System.out.println("3! is ¨ + Factnum(3))
14(1)	: pointcut FactnumOp(long var): (call(* *.Factnum(long)) && args(var)
15(1)	: before (long var): FactnumOp(var)
16(1)	: System.out.println("Finding the Factorial" + var)
5(1)	: public static long Factnum (long var)
6(1)	: if (var==1)
8(1)	: return var*Factnum (var-1)
5(2)	: public static long Factnum (long var)
6(2)	: if (var==1)
8(2)	: return var*Factnum (var-1)
5(3)	: public static long Factnum (long var)
6(3)	: if (var==1)
7(1)	: return 1
17(1)	: after (long var) returning (long result): FactnumOp(var)
18(1)	: System.out.println("Showing the result of factorial for" + var)
3(1)	: System.out.println("Root of 6 is ¨ + Root(6))
19(1)	: pointcut RootOp(long var): (call(* *.Root(long))) && args(var)
20(1)	: before (long var): RootOp(var)
21(1)	: System.out.println("Finding the Root for " + var)
9(1)	: public static long Root (long var)
10(1)	: long root = Math.sqrt (var)
11(1)	: return root
22(1)	: after (long var) returning (long result): RootOp(var)
23(1)	: System.out.println("Showing the result of Root for" + var)
4(1)	: System.out.println("P(3,2) is ¨ + Permutation(3,2))
24(1)	: pointcut PermuteOp(long var, long a): (call(* *.Permutation(long, long)) && args(var, a)
25(1)	: before (long var, long a): PermuteOp(var, a)
26(1)	: System.out.println("Finding the Permutation for " + var +" " + a)
12(1)	: public static long Permutation (long var, long a)
13(1)	: return Factnum (var)/Factnum-a)(var - a)
5(4)	: public static long Factnum (long var)
6(4)	: if (var==1)
8(3)	: return var*Factnum (var-1)
5(5)	: public static long Factnum (long var)
6(5)	: if (var==1)
8(4)	: return var*Factnum (var-1)
5(6)	: public static long Factnum (long var)
6(6)	: if (var==1)
8(5)	: return var*Factnum (var-1)
7(2)	: return 1
5(7)	: public static long Factnum (long var)
6(7)	: if (var==1)
7(3)	: return 1
27(1)	: after (long var, long a) returning (long result): PermuteOp(var, a)
28(1)	: System.out.println("Showing the result of Permutation of" + var + " " + a)

Based on this execution history DADG is constructed as shown in Fig. 3. To calculate the dynamic slice either BFS or DFS algorithm is applied on DADG. The dynamic slice for the criterion <2, n, n = 3> comes as: {1, 2, 14, 15, 16, 5, 6, 8, 5, 6, 8, 5, 6, 7, 17, 18}. The slices are marked as bold nodes in Fig. 3.

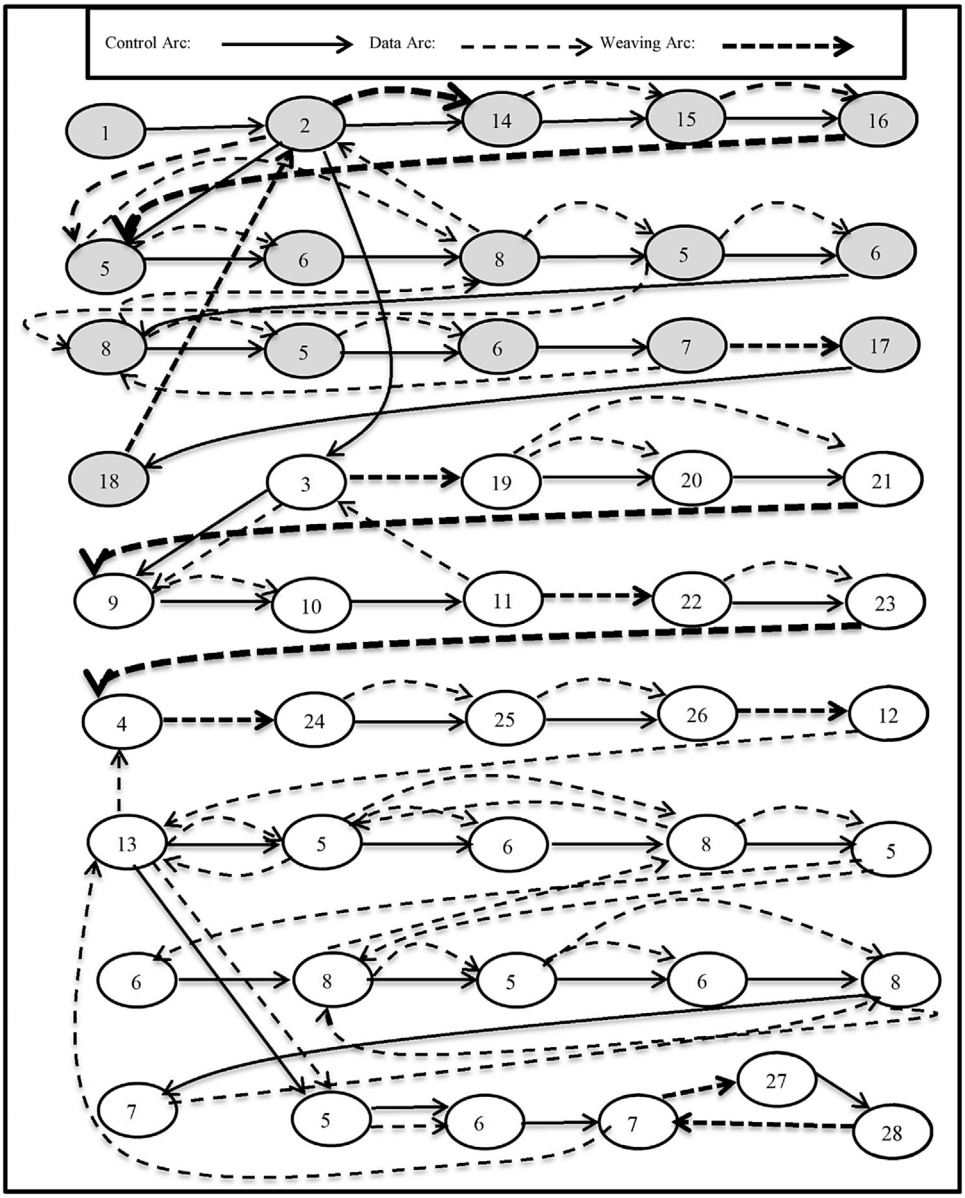

Fig. 3. DADG of the execution trace of the program given in Fig. 1.

In case of concurrent aspect oriented programs to calculate dynamic slice also one method was proposed by Ray et al. (2013). In this approach Concurrent Aspect Oriented System Dependence Graph (CASDG) used to represent AOP. The approach is centered on marking and unmarking of the executed nodes in CASDG during runtime.

Sahu and Mohapatra (2007) suggested a node marking method for dynamic slicing of AOP. They used Extended Aspect Oriented System Dependence Graph (EASDG) to represent AOP. The approach is centered on marking and unmarking of the executed nodes in EASDG suitably during runtime.

3.2. Motivation

Zhao (Zhao, 2002; Zhao and Rinard, 2003) has suggested ASDG as an intermediate representation for AOP. In this ASDG, the idea about handling the point-cuts is not given appropriately. Zhao and Rinard (2003) suggested an approach

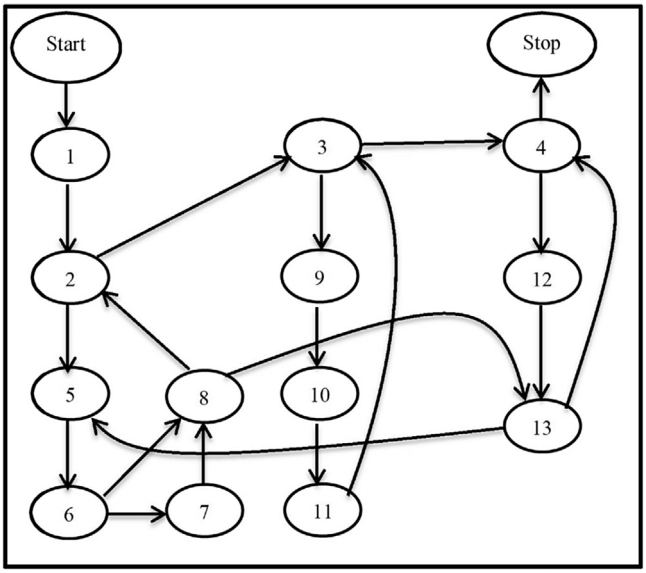

Fig. 4. CFG of non-aspect code of the program given in Fig. 1.

to create SDG for AOP. This approach provides an efficient way for addressing the concept point-cuts properly. The weakness is weaving procedure is not handled properly in this SDG.

Mohapatra et al. (2008) had suggested an efficient approach for dynamic slicing of AOP. The suggested work is centered on trace file. The drawback of this approach is, it store the every incidence of a statement during the execution. If a loop accomplishes N iterations then it will generate N nodes, hence it is a tedious work.

Mund and Mall (2002) had proposed an efficient algorithm to capture the dependences produced by procedure calls. The drawbacks of this algorithm is that the author is using only control dependency to address the dependency among statements. Also, this algorithm is exclusively designed for structured program.

Sahu and Mohapatra (2007) proposed an efficient node marking dynamic slicing (NMDS) algorithm. In this approach the author is not using the Trace file and no extra node is created during runtime.

This paper suggests *Aspect Oriented System Dependence Graph (AOSDG)* to represent AOP. Also, we suggest an approach for calculating dynamic slice of AOP. This AOSDG signifies all the features of an AOP. The AOSDG is created in two phases, creation of SDG's for non-aspect and aspect codes and linking the SDGs using call arc, weaving arc and C-Node.

4. Basic concepts

We present some elementary conceptions and terminologies related with AOP for intermediate program representations.

Definition 1 (control flow graph for non aspect code)

A CFG is a flow graph G = (N, E, Start, Stop). N represents the number of nodes, E represents the number of edges and start and stop nodes represent the unique entry and exit node. If there is a flow of control from node P to node Q then (P, Q) $\in$ E. The start and stop node do not have any ancestors and descendants respectively (Danicic et al., 2011; Horwitz et al., 1990; Mund et al., 2003; Mund and Mall, 2006). Fig. 4 represents the CFG of non-aspect code of the AOP given in Fig. 1.

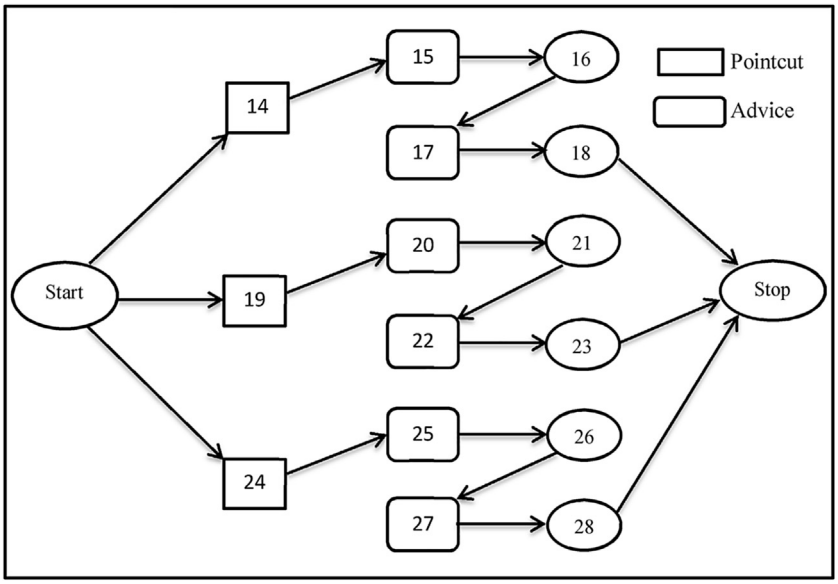

Fig. 5. CFG of aspect code of the program given in Fig. 1.

Definition 2 (control flow graph of aspect code)

A CFG of aspect code is the flow graph G = (V, E, Point-cut, Advice). V denotes the number of statements in AOP and E represents the number of edges. If there is a flow of control from node P to node Q then (P, Q) ∈ E. point-cut and advice represents the point-cut and advice in the program P (Raheman et al., 2014; Liang et al., 2006). Fig. 5 represents the CFG of aspect code of the AOP given in Fig. 1.

Definition 3 (C-Node)

The C-Node maintains the logical connectivity among different weaving points (Xu et al., 2005; Ray et al., 2012). It does not signify any particular statement in the program. C-Nodes capture the dependencies among the non-aspect and aspect codes. As it is not mapped to any particular statement of a program, we call C-Node as dummy or logical node.

Definition 4 (construction of aspect oriented system dependence graph (AOSDG))

The AOSDG of AOP is obtained by connecting the CFGs of non-aspect and aspect codes. The logical C-Node reveals the linking of non-aspect and aspect codes at proper join points. The AOSDG of an AOP contains data dependence, weaving arc, control dependence, control flow, parameter in or parameter out edges (Horwitz et al., 1990; Liang and Harrold, 1998; Zhao and Rinard, 2003; Raheman et al., 2014; Liang et al., 2006). Fig. 6 represents the AOSDG of the AOP given in Fig. 1.

Definition 5 (control dependence)

Control dependence represents the condition by which a statement executed (Danicic et al., 2011; Tip, 1995; Mund et al., 2003; Mund and Mall, 2006; Raheman et al., 2014; Ishio et al., 2003b). Suppose G is an AOSDG and M is a test node. The node N is control dependent on node M if:

1. Directed path S from M to N exists.
2. N post dominates every X! = M in S.
3. M is not post-dominated N.

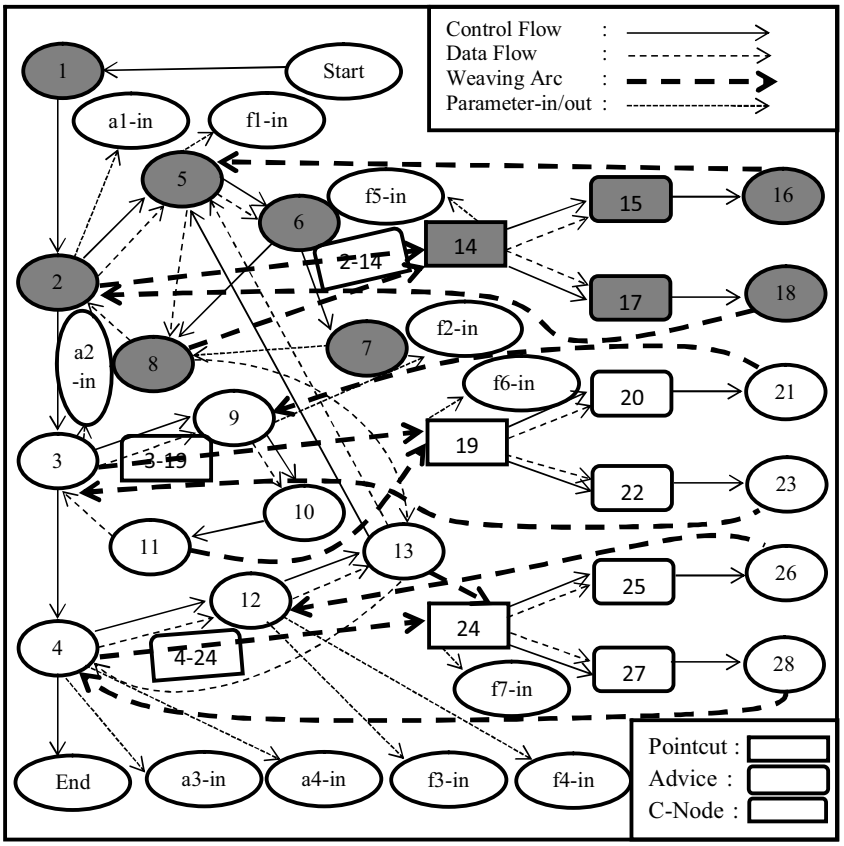

Fig. 6. AOSDG of the program given in Fig. 1.

Definition 6 (data dependence)

The flow of data between statements gives the data dependency (Tip, 1995; Mund et al., 2003; Raheman et al., 2014; Ishio et al., 2003b). The node P is data dependent on node Q if:

1. Q is a Def(var) node
2. P is a Use(var) node
3. There is a directed path S from P to Q and there is no redefinition of var in S.

Definition 7 (Def(var))

A node N in AOSDG is a Def(x) node if N is a definition statement that describes the variable x (Mund and Mall, 2002, 2006; Mohapatra et al., 2004; Ray et al., 2012).

Definition 8 (Use(var))

A node N in AOSDG is a Use(x) node if the statement N uses the value of the variable x (Mund and Mall, 2002, 2006; Ray et al., 2012).

Definition 9 (DefVarSet(u))

Let P is a program and U is a node in AOSDG of program P. DefVarSet(U) = {x: x is a variable in the program P, and U is a Def(x) node} (Mund and Mall, 2002, 2006; Ray et al., 2012).

Definition 10 (UseVarSet(u))

Let P is a program and U is a node in AOSDG of program P. UseVarSet(U) = {x: x is a variable in the program P, and U is a Use(x) node} (Mund and Mall, 2002, 2006; Ray et al., 2012).

Definition 11 (Active Control Slice(s))

Let in an AOP, N is a test node in AOSDG. UseVar Set(s) = $\{var_1,\ldots,var_k\}$. Initially ActiveControlSlice = ϕ. In an actual run of program P, ActiveControlSlice(s) = {s} U ActiveDataSlice(var_1) U…U ActiveDataSlice (var_k) U ActiveControlSlice(t), where t is the just executed successor node of N in AOSDG (Mund and Mall, 2006).

Definition 12 (Active Data Slice (var))

Let in an AOP, P is a variable and Q is a test node in AOSDG. Initially, ActiveDataSlice (P) = ϕ. UseVarSet(U) = $\{var_1,\ldots var_k\}$. At the end of execution of AOP, ActiveDataSlice(var) = {U} U ActiveDataSlice(var_1) U…U ActiveDataSlice (var_k) U ActiveControlSlice(t), where t is the just executed successor node of Q in AOSDG (Mund and Mall, 2006).

Definition 13 (DyanSlice(s, var))

Let in an AOP, P is a test node in AOSDG and Q is a variable in DefVarSet(n) U UseVarSet(s). Initially, DyanSlice(P,Q) = ϕ. At the end of the execution of AOP the dynamic slice, DyanSlice(P, Q) = ActiveDataSlice (P) U AspectCallSlice U AspectReturnSlice U ActiveControlSlice(t), where t is the most recently executed successor node of N in AOSDG (Mund and Mall, 2006).

Definition 14 (Aspect Call Slice)

Initially, AspectCallSlice = ϕ. During implementation U_{active} denotes active call node. AspectCallSlice = $\{U_{active}\}$ U AspectCallSlice U AspectBeforeAdvice U ActiveControlSlice(t), where t is the most recently executed successor node of U_{active} in AOSDG (Mund and Mall, 2002; Ray et al., 2012).

Definition 15 (Call Slice Stack)

Let G be the AOSDG of an AOP. During implementation the call node is calculated and pushed onto the CallSliceStack (Mund and Mall, 2006).

i. use AspectCallSlice to calculate appropriate runtime information confirming the implementation of N.
ii. pop the first element of CallSliceStack and apprise AspectCallSlice as the topmost element of the updated stack.

Definition 16 (Aspect Return Slice)

Let P is an AOP and G is the AOSDG of program P. Initially, AspectReturnSlice = ϕ. Let N is a RETURN node in G, and UseVarSet(n) = $\{var_1,\ldots, var_k\}$. During the implementation of the RETURN node N, AspectReturnSlice = {N} U AspectCallSlice U AspectAfterAdvice U ActiveDataSlice(var_1) U…U ActiveDataSlice (var_k) U ActiveControlSlice(t), where t is the just executed successor node of N in G.

i. Use AspectReturnSlice to calculate related runtime information confirming to the implementation of N.
ii. Update AspectCallSlice = ϕ.

5. Proposed algorithm

Smaller slices are more suitable for applications. So the objective of every algorithm is to calculate a slice which is very smaller in size.

Stage- 1: constructing static graph AOSDG

a. Start and stop will be two special nodes.
b. For every statement create a node X.
c. For every node X and Y in program P, if control flow from node X to node Y add edge(X,Y).
d. For every node X in program P, add data dependence edge (u,X) for every definition u of var.
e. For every test node u and node X in the range of u, add control dependence edge (u,X).

Stage-2: initialization

a. Set (ActiveControlSlice(u)) = ϕ for every test node u.
b. Set (DyanSlice(u,var)) = ϕ for every variable var ∈ DefVarSet(u) and UseVarSet(u).
c. Set ActiveDataSlice(var) = ϕ.
d. Set CallSliceStack = NULL.
e. Set AspectCallSlice = ϕ.
f. Set ActiveControlSlice = ϕ.
g. Set ActiveDataSlice(obj,var) = ϕ, if var is a data member of a corresponding class of an object.

Stage-3: runtime updations

Execute the program P using certain input values and repeat steps a, b and c till the program terminates depending upon the condition.

a Let Q is a procedure and u is a call node to Q, before implementation of every call node do the following:
  i. Update CallSliceStack and AspectCallSlice.
  ii. Update ActiveControlSlice, for test node u.
  iii. Update ActiveDataSlice.
  iv. Set ActiveDataSlice(Formal(u,var)) = ActiveDataSlice(Var) U AspectCallSlice.
b Update AspectReturnSlice before implementation of every return node u.
c Do this for every node u of P.

i Update ActiveDataSlice(Var).
ii Do ActiveDataSlice(Actual(u,var)) = ActiveDataSlice(var).

1. Update ActiveDataSlice(var) = AspectReturnSlice, where u is a defined variable.
2. Do ActiveDataSlice = ϕ, if var declared as automatic within the procedure Q.
3. Update ActiveControlSlice, for test node.
4. Update ActiveDataSlice, for each data variable.
5. Update CallSliceStack and AspectCallSlice.
6. Set AspectReturnSlice = ϕ.
7. Update DyanSlice(u,var), for all the variables ∈ DefVarSet(u) U UseVarSet(u).

d Exit

5.1. Analytical result

Let's implement the proposed algorithm to calculate the dynamic slice of AOP given in Fig. 1 with respect to criterion <2, n, n = 3>. Using the proposed algorithm the dynamic slice obtained are as follows: {1, 2, 5, 6, 7, 8, 14, 15, 16, 17, 18}. The nodes are also marked as bold in Fig. 6.

Before execution of node 2	
AspectCallSlice	= {2} U {14,15,16} U {1}
	= {1,2,14,15,16}
CallSliceStack	= {1,2,14,15,16}
ActiveDataSlice	= {2}
After execution of node 5	
ActiveDataSlice(var)	= {5} U ActiveDataSlice(var) U ActiveControlSlice(t)
	= {5} U {2} U {2}
	= {2,5}
After execution of node 6	
ActiveDataSlice(var)	= {6} U ActiveDataSlice(var) U ActiveControlSlice(t)
	= {6} U {2,5} U {5}
	= {2,5,6}
ActiveControlSlice	= {1,2,5,6}
AspectCallSlice	= {14,15,16}
DynamicSlice(6,var)	= ActiveDataSlice(var) U ActiveControlSlice(t) U AspectCallSlice U ActiveReturnSlice
	= {2,5,6} U {1,2,5,6} U {14,15,16} U {ϕ}
	= {1,2,5,6,14,15,16}
After execution of node 7	
ActiveReturnSlice	= {7} U DynamicSlice U AspectCallSlice
	= {7} U {1,2,5,6,14,15,16} U {14,17,18}
	= {1,2,5,6,7,14,15,16,17,18}
Before execution of node 8	
AspectCallSlice	= {8} U {14,15,16} U {6}
	= {6,8,14,15,16}
After execution of node 8	
AspectCallSlice	= {8} U {14,15,16} U {6}
	= {6,8,14,15,16}
CallSliceStack	= {1,2,5,6,7,8}
ActiveDataSlice	= {8} U ActiveDataSlice(var) U ActiveControlSlice(t)
	= {8} U {2,5,6} U {6}
	= {2,5,6,8}
ActiveReturnSlice	= {8} U DynamicSlice U AspectCallSlice
	= {8} U {1,2,5,6,7,14,15,16} U {14,17,18}
	= {1,2,5,6,7,8,14,15,16,17,18}
After execution of node 2	
ActiveDataSlice(var)	= {2} U ActiveDataSlice(var) U ActiveControlSlice(t)
	= {2} U {2,5,8} U {8}
	= {2,5,8}
ActiveControlSlice	= {1,2,5,6,7,8}
AspectCallSlice	= {14,15,16,17,18}
DynamicSlice(2,var)	= ActiveDataSlice(var) U ActiveControlSlice(t) U AspectCallSlice U ActiveReturnSlice
	= {2,5,8} U {1,2,5,6,7,8} U {14,15,16,17,18} U {1,2,5,6,7,8,14,15,16,17,18}
	= {1,2,5,6,7,8,14,15,16,17,18}

5.2. Space complexity

Suppose P is an AOP with S number of statements. So the AOSDG will consist of S number of nodes in matching to the statements in P. The algorithm also uses data structures as AspectCallSlice and AspectReturnSlice. These data structures need $O(S^2)$ space. Hence the space complexity of the algorithm is $O(S^2)$, where S is the number of statements in P.

5.3. Time complexity

Suppose P is an AOP with S number of statements. Every statement in P will be denoted by a single node in AOSDG. So there are S numbers of nodes in AOSDG. The time required to calculate the dynamic slice is linear with respect to the number of statements in P. Hence, the time complexity of the algorithm is O(S).

5.4. Comparison with existing work

Various algorithms have been proposed and implemented for slicing of AOPs. As we are calculating the dynamic slice of AOP so the weaving process is already done while the program is first compiled. To represent the links we are using the concept of C-Node. Different methods have been used to solve these problems. No such algorithm has been developed to capture the dependencies caused by different methods.

We have used the concept of Logical node or dummy node known as C-Node and the basic data structures like AspectCallSlice, AspectReturnSlice, CallSliceStack to find out an efficient dynamic slicing algorithm for AOPs which can capture all dependencies caused by different aspects.

6. Conclusion

As slicing AOP has thrown a lot of challenges, the proposed algorithm tried to solve one of them such as slicing AOP dynamically by using Aspect Oriented System Dependence Graph (AOSDG) as intermediate representation for AOP. The AOSDG uses dummy nodes to make the connection in between aspect and non-aspect codes. This algorithm uses the basic data structures like AspectCallSlice, AspectReturnSlice, CallSliceStack to find out dynamic slices of AOP.

References

Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation, SIGPLAN, Notices, Analysis and Verification vol. 25, 246–256.

Danicic, Sebastian, et al., 2011. A unifying theory of control dependence and its application to arbitrary program structures. Theor. Comput. Sci. 412, 6809–6842.

Aspect Oriented Programming, By Gustav Evertsson, 2002.

Horwitz, S., et al., 1990. Inter-procedural slicing using dependence graphs. ACM Trans. Progr. Lang. Syst. 12 (January (1)), 26–60.

Ishio, Takashi, et al., 2003a. Program slicing tool for effective software evolution using aspect-oriented technique. Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03), 3.

Application of Aspect-Oriented Programming to Calculation of Program Slice Takashi Ishio, Shinji Kusumoto, Katsuro Inoue. Technical Report, Submitted to ICSE 2003.

Ishio, Kusumoto, Shinji, Inoue, Katsuro, 2004. 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).

Jain, Sonam, et al., 2013. A new approach of program slicing: mixed S-D (static & dynamic) slicing. Int. J. Adv. Res. Comput. Commun. Eng. 2 (May (5)).

Katti, Amogh, et al., 2012. Application of program slicing for aspect mining and extraction—a discussion. Int. J. Comput. Appl. (0975–8887) 38 (January (4)).

Kiczales, Gregor, et al., 2001. An overview of AspectJ. Proceedings of the 15th European Conference on Object Oriented Programming, 327–353.

Korel, B., Laski, J., 1990. Dynamic slicing of computer programs. J. Syst. Softw. 13, 187–195.

Korel, B., et al., 1988. Dynamic program slicing. Inf. Process. Lett. 29 (3), 155–163.

Larsen, L., Harrold, M.J., 1996. Slicing object-oriented software. In: Proceedings of 18th International Conference on Software Engineering, March, pp. 495–505.

Liang, D., Harrold, M.J., 1998. Slicing objects using system dependence graph. In: Proceedings of the International Conference on Software Maintenance, IEEE, November, pp. 358–367.

Liang, S.H.I., et al., 2006. System dependence graph construction for aspect oriented C++. Wuhan Univ. J. Nat. Sci. 11 (3), 555–560.

Mohapatra, D.P., Mall, R., Kumar, R., 2004. An edge marking technique for dynamic slicing of object-oriented programs. September In: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04), vol. 1, pp. 60–65.

Mohapatra, Durga Prasad, Mall, Rajib, Kumar, Rajeev, 2006. An overview of slicing techniques for object-oriented programs. Informatica 30, 253–277.

Mohapatra, D.P., et al., 2008. Dynamic slicing of aspect-oriented programs. Informatica 32, 261–274.

Mund, G.B., Mall, R., 2002. An efficient dynamic program slicing technique. Inf. Softw. Technol. 44 (February (2)), 123–132.

Mund, G.B., Mall, Rajib, 2006. An efficient interprocedural dynamic slicing method. J. Syst. Softw. 79, 791–806.

Mund, G.B., et al., 2003. Computation of intraprocedural dynamic program slices. Inf. Softw. Technol. 45, 499–512.
Pollice, Gary, 2004. A look at aspect-oriented programming, Worcester Polytechnic Institute, from The Rational Edge, ibm.com/developer Work.
Raheman, Sk Riazur, et al., 2011. Dynamic slicing of aspect-oriented programs using AODG. IJCSIS) Int. J. Comput. Sci. Inf. Secur. 9 (April (4)).
Raheman, Sk Riazur, et al., 2013. An overview of program slicing and its different approaches. Int. J. Adv. Res. Comput. Sci. Softw. Eng. 3 (November (11)), ISSN: 2277 128X.
Raheman, Sk Riazur, et al., 2014. Dynamic slice of aspect oriented program: a comparative study. Int. J. Recent Innov. Trends Comput. Commun. 2 (February (2)).
Ray, A., Mishra, S., Mohapatra, D.P., 2012. A novel approach for computing dynamic slices of aspect oriented programs. Int. J. Comput. Inf. Syst. 5 (September (3)), 6–12.
Ray, Abhishek, et al., 2013. An approach for computing dynamic slice of concurrent aspect-oriented programs. Int. J. Softw. Eng. Appl. 7 (January (1)).
Robinson, David. An Introduction to Aspect Oriented Programming in e, Rel_1.0: 21-JUNE-2006.
Sahu, Madhusmita, Mohapatra, Durga Prasad, 2007. A node-marking technique for dynamic slicing of aspect oriented programs. The proceedings of 10th International Conference on Information Technology (ICIT), 155–160.
Sahu, M., Mohapatra, D.P., 2009. A node-marking technique for slicing concurrent object-oriented programs. Int. J. Recent Trends Eng. 1 (May (1)).
Sikka, P., et al., 2013. Program slicing techniques and their need in aspect oriented programming. Int. J. Comput. Appl. (0975–8887) 70 (May (3)).
Tip, Frank, 1995. A survey of program slicing techniques. J. Progr. Languages, 1–65.
Weiser, M., 1984. Program slices. IEEE Trans. Softw. Eng. SE-10 (July (4)), 352–357.
Xu, Baowen, Qian, Ju, Zhang, Xiaofang, Wu, Zhongqiang, Chen, Lin, 2005. A brief survey of program slicing. SIGSOFT Softw. Eng. Notes 30 (March (2)), 1–36.
Zhao, J., Rinard, M., 2003. System Dependence Graph Construction for Aspect-Oriented Programs. Technical Report. Laboratory for Computer Science, Massachusetts Institute of Technology, USA, March.
Zhao, J., 2001. Dynamic slicing of object-oriented programs. Uhan Univ. J. Nat. Sci. 6 (1–2), 391–397.
Zhao, J., 2002. Slicing aspect-oriented software. In: Proceedings of 10th International Workshop on Program Comprehension, June, pp. 251–260.