DEPENDENCE FLOW GRAPH FOR ANALYSIS OF ASPECT- ORIENTED PROGRAMS

Syarbaini Ahmad[1] , Abdul Azim A. Ghani[2] ,Fazlida Mohd Sani[3]

[1]Faculty of Science & Info Technology, International Islamic University College Selangor (KUIS), 43000 Kajang Selangor Malaysia
[2,3]Faculty of Computer Science and Information, Technology, University Putra Malaysia (UPM), 43400, Serdang Selangor, Malaysia

Abstract

Program analysis is useful for debugging, testing and maintenance of software systems due to information about the structure and relationship of the program's modules . In general, program analysis is performed either based on control flow graph or dependence graph. However, in the case of aspect-oriented programming (AOP), control flow graph (CFG) or dependence graph (DG) are not enough to model the properties of Aspect-oriented (AO) programs. With respect to AO programs, although AOP is good for modular representation and crosscutting concern, suitable model for program analysis is required to gather information on its structure for the purpose of minimizing maintenance effort. In this paper Aspect Oriented Dependence Flow Graph (AODFG) as an intermediate representation model is proposed to represent the structure of aspect-oriented programs. AODFG is formed by merging the CFG and DG, thus more information about dependencies between the join points, advice, aspects and their associated construct with the flow of control from one statement to another are gathered. We discussthe performance of AODFG by analysing some examples of AspectJ program taken from AspectJ Development Tools (AJDT).

Keywords

dependence flow graph, control flow graph, dependence graph, aspect-oriented, program analysis, intermediate representation, maintenance.

1. Introduction

Program analysis is a key activity in debugging [13], testing [9] and maintenance [16] to optimize functionality. In software engineering, debugging is activity of allocation and solution or avoidence from syntax error. Maintenance is the process of enhancing or optimizing the functionality and usability of the system. The main purpose of analysis in software engineering is to get information about the program by tracing the dynamic or static program. It is very helpful to improve the accuracy of array, subscript and variable aliasing analysis, to test legality of loop transformations, to detect potential error during run-time program, and to provide user respond during human and computer interactive environments [26]. Analysis activity can be provided in every single phases of software development such as in the requirement, design, testing, and maintenance. Ideally, program analysis will allow us to seek in depth information into the structure of the program without modifying the general framework. Program analyses through intermediate representation also pave the way for more intensive approaches to optimize and provide a better feedback to maintainer to enhance the usability and functionality of the program. It is very important to depict relationship among the code components in the program. So far, research in this area has focused on the establishment of intermediate representation tools and its functionality for AO programs such as call graph [10, 26, 37] control flow and data flow graph [11, 14, 27, 28]

Aspect-oriented is an improvement of Object- Oriented (OO) paradigm proposed by Kiczales et. al. [33] with the goal to enhance software maintainability through new modularization mechanisms for encapsulation of crosscutting concerns. It is very useful in software engineering to reduce the complexity in program development especially for reverse engineering and maintenance activities such as program analysis, slicing and refactoring. Separation of concerns [32] are able to identify, encapsulate and manipulate in isolated way only those parts of software that are relevant to a concept, object or intention given.

AOP presents unique opportunities and problems for program representation schemes. The crosscutting concerns produce complex structures to the program. To enable the development of effective analysis for AOP, the program representation scheme must appropriately preserve the structure associated with the use of features such as pointcut, joint points, advice and introduction. It is therefore of interest to analyse the structure of AOP program and present with proper and informative view.

The proposed approach in this paper is an intermediate representation as an analysis tool which is based on the dependence flow graph (DFG) to get the details of code relationships. DFG originally is used for procedural programming [13] to show the detailed information about the code structure. It combines two different kind of representation which is CFG and DG. CFG is used to produce the information about flow work list that is used to propagate the executable flag. It is a model of node (or point) that corresponds to a program statement, and each arc (or directed edge) indicates the flow of control from one statement to another [35].

DG produces data structure to be traversed to obtain information about dependencies among the node. DG is a directed graph normally used to represent dependencies of several objects towards each other. In OO [36], It is a collection of method dependence graph representing a main() method or a method in a class of the program, or some additional arcs to represent direct or indirect dependencies between a call and the called method and transitive interprocedural data dependencies. DG in AO is used to represent the dependencies between the concept of join points, advice, aspects and their associated constructs. In our study, we regenerate the use of DFG by improving the usability of the graph from procedural programming only into aspect-oriented programming.

In this paper, we propose an analysis tool for aspect- oriented programs. The tool is able to identify which part of an aspect or non-aspect program is affected by a specific aspect or non-aspect program. Tool can provide a better understanding of how the analysis output can help software engineer to analyse the dependencies among the AO codes in the program. Maintainer will not need to spend more time to understand the structure and relationship among the variables in order to modify the program. In particular, we use DFG which first known introduced by

Pingali et.al. [13] implemented in procedural programming. This research focus on the program analysis activity and proposed new intermediate representation called Aspect Oriented Dependence Flow Graph (AODFG). The AODFGs of some small AO systems have been generated to verify and validate the feasibility and correctness. It could be useful for maintenance purpose especially for corrective and adaptive maintenance.

This paper is organized as follows; Section 2 represents the related research in program representation by focusing on aspect oriented. Section 3 is a description about characterization of CFG and DG analysis in producing DFG. Section 4 presents a construction of AODFG. Section 5 discusses performance evaluation of analysis result and finally, section 6 contains the conclusion and future works.

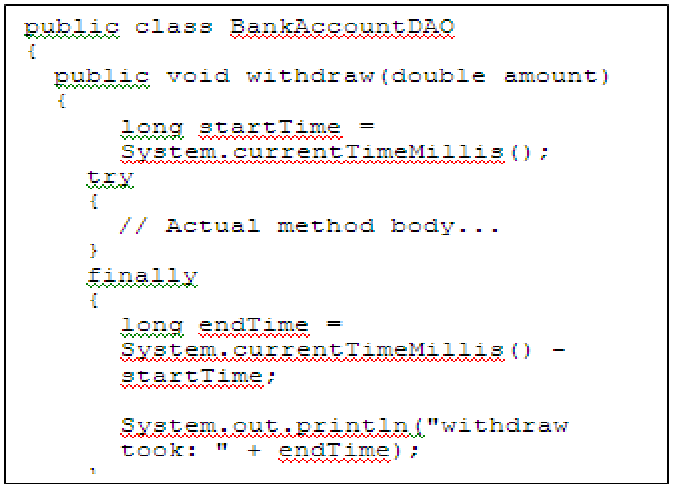
2. Aspect-oriented Programming Paradigm

AOP introduces a modularization unit which is aspect, which means a focus on a specific crosscutting functionality in the program. By weaving crosscutting concern into the core classes it creates applications that are easy for maintenance purpose. AOP does not replace object-oriented but it is complementary to object-oriented. AOP behaviour looks like it should have structure, but it is difficult to get the structure in code with traditional object-oriented techniques since complexity with the relationship.

A crosscutting concern as a main behaviour of AOP is important for "cuts" across multiple points in the object models. As a development methodology, AOP recommends the abstract and encapsulate crosscutting concerns.

Figure 1[39] is the example of OO code which is important to implement AOP to measure the amount of time to invoke a particular method in.

While this code works, there are a few problems with the traditional approach



```
public class BankAccountDAO
{
  public void withdraw(double amount)
  {
      long startTime =
      System.currentTimeMillis();
    try
    {
      // Actual method body...
    }
    finally
    {
      long endTime =
      System.currentTimeMillis() -
      startTime;

      System.out.println("withdraw
      took: " + endTime);
    }
```

Figure 1 Class Bank Account DAQ

1. It's extremely difficult to turn metrics on and off, as we have to manually add the code in the try>/finally block to each and every method or constructor we want to benchmark.

2. The profiling code really doesn't belong sprinkled throughout application code. It makes code bloated and harder to read

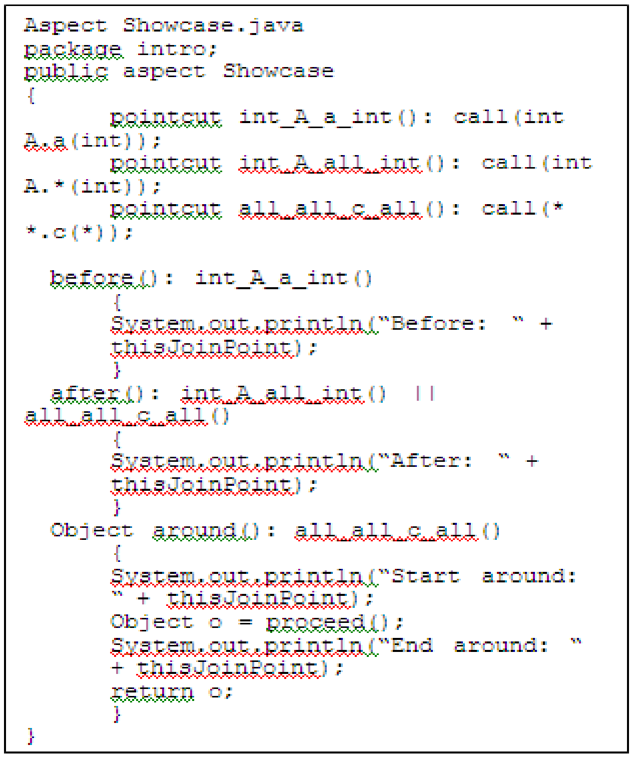
3. If we want to expand this functionality to include a method or failure count, or even to register these statistics to a more sophisticated reporting mechanism, we have to modify a lot of different files (again). This approach to metrics is very difficult to maintain, expand, and extend, because it's dispersed throughout the entire code base. Aspect-oriented programming gives us a way to encapsulate this type of behaviour functionality. It allows us to add behaviour such as metrics "around" our code. For example, AOP provides us with programmatic control to specify that we want calls to BankAccountDAO to go through a metrics aspect before executing the actual body of that code.

The most popular AOP language is AspectJ. AspectJ is extensions for JAVA that enable developers to work with crosscutting concern effectively [38].

AOP is important to work with abstractions that correspond more directly to all the different aspects of concern in software [2]. It is used to encapsulate the crosscutting behaviours into modular units called aspects. These units are composed of advices that realize the crosscutting behaviour, and pointcut descriptors, which designate the points in the program where the advices are inserted. Join point is a special well-defined point in the program flow that is needed where horizontal bundling occurs in the code. Example of join points are method and constructor calls or executions, attribute accesses, and exception handling.

Pointcut is used to pick a join point when a method is made (see figure 2 for example [39]). It will select certain join points along with some context values specific to the points. All join points are call methods for pointcut in the program flow. For example int_A_a_int() picks a join point when a method call to a method int a(int) of a class A is made.

Advice is an executable code, which will be an actual implementation of a concern. It gets executed when a given pointcut is reached. The advice's code can be run before, after, or instead of join points are reached. Figure 2 shows implementation of before, after and around advice in an example program [39].



```
Aspect Showcase.java
package intro;
public aspect Showcase
{
      pointcut int_A_a_int(): call(int
A.a(int));
      pointcut int_A_all_int(): call(int
A.*(int));
      pointcut all_all_c_all(): call(*
*.c(*));

  before(): int_A_a_int()
      {
      System.out.println("Before:  " +
      thisJoinPoint);
      }
  after(): int_A_all_int() ||
all_all_c_all()
      {
      System.out.println("After:  " +
      thisJoinPoint);
      }
  Object around(): all_all_c_all()
      {
      System.out.println("Start around:
      " + thisJoinPoint);
      Object o = proceed();
      System.out.println("End around: "
      + thisJoinPoint);
      return o;
      }
}
```

Figure 2 Aspect Showcase.java

The before advice of the Showcase aspect will run before the int_A_a_int() pointcut's join points, in this particular case, before all calls to int A.a(int) method.

• The after advice will run after one of the join points of either int_A_all_int() or all_all_c_all() pointcuts, in this example, after methods void intro.B.c(double), String intro.A.c(String), int intro.A.b(int), or int intro.A.a(int) will finish.

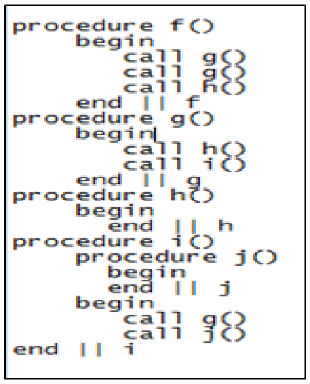
• The around advice will run instead of all join points picked by the all_all_c_all() pointcut. It means that when such a join point is reached, the control will be passed to the advice, and then it can decide what to do.

2.1 Modelling of AOP Program Structure

Recently, AOP is making its way through in almost all stages of software development life cycle. There has been much research in the area of requirements engineering [4, 5], architectural modelling [6, 19], design [3, 7, 12] and testing [8]. In this section, we briefly explain the idea of analysis aspect oriented program as a complicated relationship in the structure by representation using DFG. It is much related with CFG and DG.

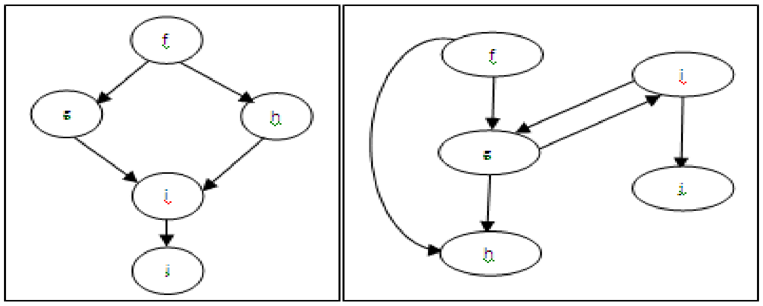
CFGs are showing relationship as nodes represent either assignment statements or conditional expressions that affect flow of control of the program and the edges represent the possible transfer of control between the statements. The approaches differ with respect to the handling of branching and the merging of branches, and the representation of statements that are always executed together [9].

The example of CFG is shown in Figure 4(a) from the skeleton presented in Figure 3. In aspect oriented analysis for CFG, Xu, G. [27] proposes an approach for program slicing of AspectJ software, based on a novel data flow and control flow program representation. He describes a general approach for constructing a static data flow and control flow representation for AspectJ, for the purposes of dependence analysis, program slicing, and similar inter-procedural analyses. He further conducted an experimental study on 37 AspectJ programs on AJANA, as analysis framework based on abc AspectJ compiler. The results were compared with analysis of the woven byte code in which they supported that is proposed model was suitable for static analysis.Cacho, N. et.al. [28] using CFG for AOP presenting the novel exception handling mechanism to promote improved separation between normal and error handling code. They promote better maintainability of normal and error handling code by separating the handlers and eliminating annoying exception interface declarations. They develop EJFlow with small syntactic additions to AspectJ in their study. CFG will allow us to formulate a simple algorithm to analyse the code based on abstract interpretation on the control structure. However, the program statement is not only about the structure. It is more about discovering data dependencies among statements and its relationships.



```
procedure f()
    begin
       call g()
       call g()
       call h()
    end || f
procedure g()
    begin
       call h()
       call i()
    end || g
procedure h()
    begin
      end || h
procedure i()
    procedure j()
      begin
      end || j
    begin
       call g()
       call j()
end || i
```

Figure 3: A program skeleton



(a)Control Flow Graph (b) D Figure 4: Graph

Figure 4: Graph Representation

DG is arises from data flow of the program. It shows the relationship among the statements as in Figure 3(b). The oval is representing the statement (S) and the arrow represents the dependency of data. Sg is arises from Sf. Sh and Si is execution order from Sg. In aspect oriented, Rinard and Zhao [1] proposed an AO code representation based on an adaptation of the dependence graph to support the slicing of AO code. They extend their previous study on system dependence graphs (SDG) [35] by providing a solid foundation for further analysis of AOP program by using dependence graph. Mohapatra [11] then proposed a dependence based representation to represent dynamic dependences between the statements. They improved from [1] by producing Dynamic Aspect Oriented Dependence Graph (DADG) as intermediate representation. DADG was used to represent various dynamic dependences between the statements of AOP

DFG are proposed by Pingali et. al.[13] with procedural programming. The original study is based on two crucial reasons. The first reason is representation must be a data structure that can be rapidly traversed to determine dependence information. Second, this representation must be a program in its own right, with a parallel, local model of execution. They propose DFG which have the two of the properties mentioned above. An interesting feature of this representation is that it naturally incorporates the best aspects of many other representations, including call graph, dependence graphs and control flow graphs by speed up the analysis and optimization [17] since it provides hybrid information which comes from CFG and DG.

3. Theoretical Characterization of DFG

In maintenance activity, constructing a global understanding of the program and how the program abstraction, control flow graph is very useful to allows us to formulate a simple algorithm to visualize the structure of the program. It is based on interpretation of find possible paths that need to be extracting [16]. Dependence analysis is very useful tool for instruction scheduling by determining the ordering relationships between the code executions [22]. Algorithms that use the various dependence graphs are more complex, and none of them found any possible paths without some program transformation. However, the asymptotic complexity of such algorithms would be better than the algorithms that use control flow graphs.

DFG is an ideal program representation to get a region of dependencies and flows of the program. It can be viewed as a data structure that can be traversed efficiently for dependence information, and at the same time it can also be viewed as a precisely defined language with a local operational semantics. It can help in reverse engineering of the program with perspectives of reuse, refactoring, reengineering or slicing. Dependence flow graphs are a synthesis of ideas from data dependence graphs and control flow computation. As in the data dependence graph, the dependence flow graph can be viewed as a data structure in which arcs represent dependencies between operations.

However, unlike data dependence graphs, dependence flow graphs are executable, and the execution semantics, called dependence-driven execution, is a generalization of the data-driven execution semantics of data flow graphs. In data flow graphs, nodes represent functional operators that communicate with each other by exchanging value-carrying tokens along arcs in the graph. Abstractly, this concept represents the linked lists in general by using du or ud chain as a function from a variable and a basic-block position pair. In this section, we will look at the computation process of generating AODFG which uses data flow analysis and dependence analysis for aspect- oriented programs.

3.1 Du and Ud Chains

It is often convenient to directly link labels of statements that produce values to the labels of statements that use them. For each use of variable, associate all assignments that reach that use are called use-definition chains or ud-chains. For each assignment, associate all uses are called Definition-use chains or du-chains. The standard definition of du-chains and ud-chains are as in definition 1 and 2.

Definition 1. A definition of variable x is said to reach a use of x if there is a control flow path from the defines to the uses that does not pass through any other definition of x.
A du- chain for variable x is a node pair (n1, n2) such that n1 defines x, n2 uses x, and the defines of x at n1 reaches the uses of x at n2.

Definition 2. A definition of variable x is said to reach a defines of x if there is a control flow path from the uses to the defines that does not pass through any other defines of x.

A ud-chain for variable x is a node pair(n1, n2) such that n1 uses x, n2 define x, and the uses of x at n1 reaches the defines of x at n2.

3.2 Control Flow Graphs

In this section, we present a control flow analysis which is one of the analysis phases to represent the DFG graph. We use iterative data flow analyzers users dominators to discover loops. Control flow analysis is very important to characterize the flow of programs, so that any unused generality can be removed and optimizing the manipulations of the data.

Definition 3. A du-chain for variable x is an edge pair (e1, e2) such that

1. The source of e1 defines x,
2. The destination of e2 uses x
3. There is a control flow path from e1 to e2 with no assignment to x.

Definition 4. The control flow graph Gf = (N, E) of a function f has one node na ∈ N for each statement a in f and two additional nodes nin, nout. We add an edge (na, na') if the statement a' is executed immediately after the statement a. For the first statement a1 in the function, we introduce an edge (nin, na1). Furthermore, we add edges (na', nout) for each node na' that is associated to a statement a', after which the control flow leaves the function because of a return-statement or the right brace that terminates the function. The control flow graph of an empty function, i.e., a function without any statements consists of N = {nin,nout} and E = {(nin,nout)}.The node nin is the only entry node and the node nout the only exit node of the control flow graph. Note that the control flow graph Gf is a graph where each node (except nin and nout) corresponds to one statement in the function f.

We present a simple JAVA program as shown in Figure 5 that shows the computes Fibonacci numbers program [15].

When we analyse the source code statically, the technique used in getting CFG start by discovering its data structure such as if, else and if-then-else with a loop in the intermediate code. Then, we extract the code line by line and represent it visually in flowchart to make it clearer to the eye.

Next, we identify basic block to show a straight line sequence of code which have an entry and exit point. The characteristic of producing basic block are as in Definition 5.

Definition 5. The characters of analysis policy;

Char1: The entry point of the routine x.
Char2: a target branch y, or

Char3: instruction following a branch y or a return to x.

Such instructions are defined as a leader. Each leader is flowing to another until exit in sequence. The flow will become clear to analyze backward dataflow by add entry block as a successor and exit block at the end of the branches.

By merging the code in Figure 5(a), the region of data as in Figure 5(b) can be performed. Nodes in line 1 through 4 as a basic block B1. Line 6 through 11 forms another block B6 which has a back edge to line 4 in block B4. The others is a basic block unto itself; which are node for line 7 into B2, node for line 8 into B3 and node for line 5 into B5.

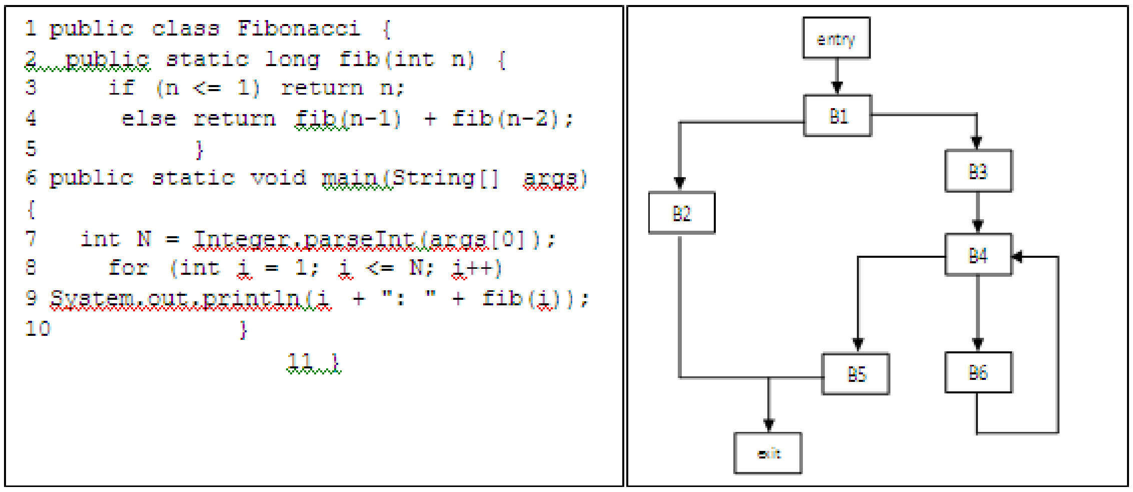

```
1 public class Fibonacci {
2 public static long fib(int n) {
3     if (n <= 1) return n;
4      else return fib(n-1) + fib(n-2);
5           }
6 public static void main(String[]  args)
{
7   int N = Integer.parseInt(args[0]);
8     for (int i = 1; i <= N; i++)
9 System.out.println(i + ": " + fib(i));
10                }
                     11 }
```

Figure 5: Fibonacci computing program and control flow graph.

3.3 Dependence Flow Graphs

Dependence analysis will construct a graph called dependence graph that represents the dependences in the program. But in our study, we not present the dependence graph on itself but the information from the dependence analysis.

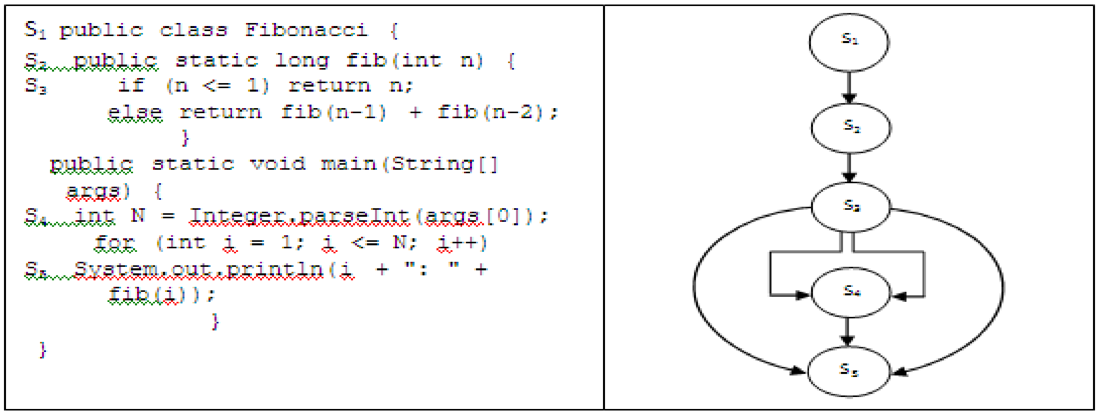
The purpose of dependence analysis is to determine the ordering relationships between instructions that must be satisfied for the code to execute correctly.

Compare to control flow analysis, dependence analysis can be applied at any level in the program. This is because; the source of dependence analysis will perform based on statement (S) execution. If S1 precedes S2 (S1 □S2) in their execution order, we know that S2 are depending to S1. There are 4 types of data dependences [15]:

Definition 6. The character type of dependencies

Type1: Flow dependence/true dependence; If S1 □ S2 and the former sets of value that the later uses.
Type2: Anti dependence; If S1 □ S2 , S1 uses some variable's value, and S2 sets it.
Type3: Output dependence; If S1 □ S2 and both statements set the value of some variable.
Type4: Input dependence; If S1 □ S2 and both statements read the value of some variable.

Figure 6 is an example consists of the four types of dependence as explained. Figure 6(a) is simplified of the analysed code and Figure 6(b) is dependence graph of Figure 6(a). The flow dependence between S3 and S4 is Type1 when the former sets a value that the latter uses. In the reverse order, S3 uses some variable's value (e) and S4 sets it as Type2. S3 and S5 are set the value of some variable which is mentioned in Type3. And finally Type4 are dependence between S3 and S5 since both read the value of e.



```
S1 public class Fibonacci {
S2  public static long fib(int n) {
S3      if (n <= 1) return n;
        else return fib(n-1) + fib(n-2);
            }
  public static void main(String[]
   args) {
S4  int N = Integer.parseInt(args[0]);
      for (int i = 1; i <= N; i++)
S5  System.out.println(i + ": " +
        fib(i));
              }
 }
```

(a) (b)

Figure 6 Example of dependence graph.

4. AODFG Construction

This section discuss on the extension of the construction of both control flow and data dependency to come up with AODFG. We assume that AODFG can help maintainer to manage the maintenance activity with aspect-oriented program. Therefore our approach focuses on maintainer view to understand the architecture of the program in a short time.

Our approach, in the case of aspect-oriented, shares the same viewpoint with procedural [20] and OO approach in the sense that it is also a collection of information about the dependencies of the data and the flow of control represent in a hierarchical manner. But the different between AO and others is only the AO features that exist in aspect code). As a concept, AO survival is depending on base code which is OO. Base code which normally includes classes, interfaces, and standard Java features or constructs, and aspect code which put into practice the crosscutting concerns in the program by using aspect, advice, etc. [23].

Figure 7 depicts the macro view of AODFG construct. It shows that the AODFG diagram are generating from analysis of control flow and dependence flow. It just brings the information from control flow and dependence flow and represents hybrid information in one single graph.

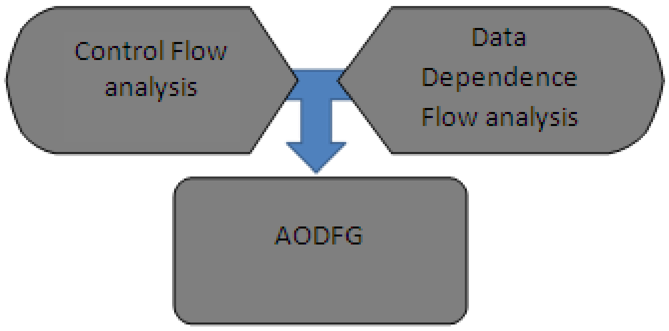


Figure 7: The macro view of AODFG

4.1 Aspect-oriented Construct in AODFG

We implement the DFG algorithm to suite with aspect-oriented features. The AODFG algorithm consists of the following phases:

i. Analyze the control flow structure of the program as the technique used in CFG. The control flow graph Gf = (N, E) of a function f has one node na ∈ N for each statement a in f and two additional nodes nin, nout. We add an edge (na, na') if the statement a' is executed immediately after the statement a. For the first statement a1 in the function, we introduce an edge (nin, na1). Furthermore, we add edges (na', nout) for each node na' that is associated to a statement a', after which the control flow leaves the function because of a return-statement or the right brace that terminates the function. The control flow graph of an empty function, i.e., a function without any statements consists of N = {nin,nout} and E = {(nin,nout)}.The node nin is the only entry node and the node nout the only exit node of the control flow graph. Note that the control flow graph Gf is a graph where each node (except nin and nout) corresponds to one statement in the function f.

ii. Analyze the dependencies among the statement in program as a technique used in dependence graph. The character type of dependencies such as Flow dependence/true dependence; If S1 □ S2 and the former sets of value that the later uses. Anti-dependence; If S1 □ S2 , S1 uses some variable's value, and S2 sets it. Output dependence; If S1 □ S2 and both statements set the value of some variable. Input dependence; If S1 □ S2 and both statements read the value of some variable.

iii. We used AspectJ as a target language and advice execution as a method call. The features of AOP introduced are following:

Join point: AspectJ provides join point object in order to access context information. The method join point is prepared for accessing parameters. Since the parameter of the method call is determined in runtime, the caller of the method call is handled as references to all parameters of the method of the join point.

Pointcut: An advice depends on a pointcut definition. Since a pointcut determines an advice execution, we connect a dependency edge from a pointcut to an advice.

Advice call: consists of an advice type (before, after and around). A vertex corresponding to a join point shadow is regarded as a caller vertex of the advice.

iv. Construct a graph that contains information about the control flow and data dependencies in the program.

From the algorithm of AODFG, we develop a tool to visualize the output of analyzing AspectJ program as shown in Figure 7.

4.2 AODFG Representation Tool - AOST

In order to test the functionality of our study, we develop Aspect Oriented Slicing Tool (AOST). AOST is a tool to visualize control view and dependence view of JAVA aspectJ program. We use C# as a programming tool and Graphviz as a graph representation tool. We can use AOST to analyze the aspectJ program line by line. Then it can identify the method, aspect and any related statement. Then visualize the relationship as an AODFG graph.

AOST are divided into three compartment which are original code, generated code and DFG. Figure 8 are AOST interface. The detail operational are as below:-

i. Original Code

As shown in figure 8, Original Code will show the a complete JAVA program. We can import java class from location. After select the destination by using menu file the complete class will appear on this view. If we want to look for a part or a full program which include with more than one class, we can copy the code from each class or aspect manually and paste in the location continues after each other.

ii. Generated Code

The next step after original code viewed will react after we press on “Generate” button. This button is very important to sort the code and classify by following the rules of JAVA syntax. For example we look at the following rule that describe the structure of a ‘while’ statement.

“<while_stmt> □ while (<logic_expr>) <stmt>”

A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of terminal and nonterminal symbols. A grammar is a finite nonempty set of rules. An abstraction (or nonterminal symbol) can have more than one RHS as in the statement as follows.

<stmt> □ <single_stmt>
| begin <stmt_list> end

Then, we identify the program as describe in chapter 4.1. this is very important to differentiate between

- object oriented and aspect oriented,
- data dependency and it relationship and
- flow of control and it relationship.

The output from this level is an analyzed code derived from original code. The different in this view is the code are represented with sort of method and aspect (aspect, pointcut, join points and advice) and statement.

iii. DFG view

In this level, we will represent the construct of graph based on the information from previous steps. Graphviz will work in this level. Graphviz is open source graph visualization software. It is a way of representing structural information as diagrams of abstract graphs for AODFG. It various functionality is important to show the dependencies of data and flow relationship of control.

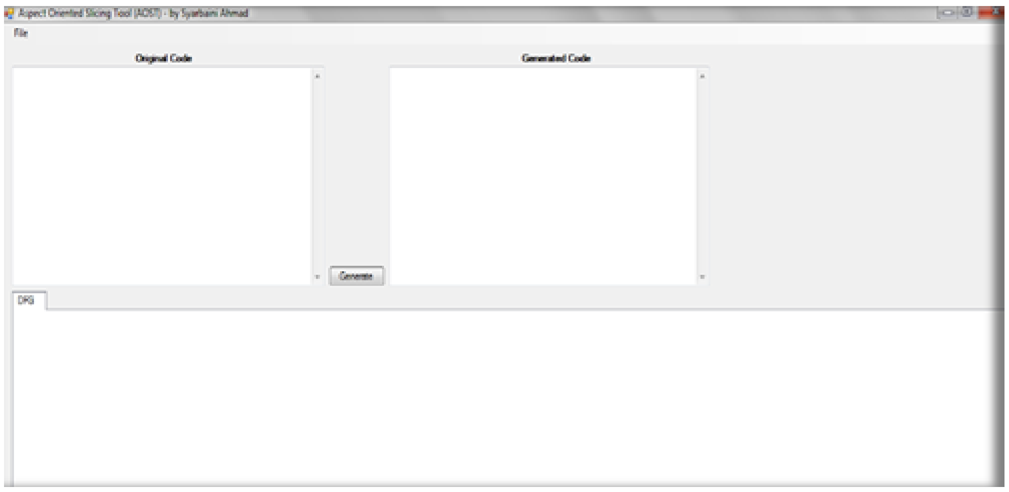


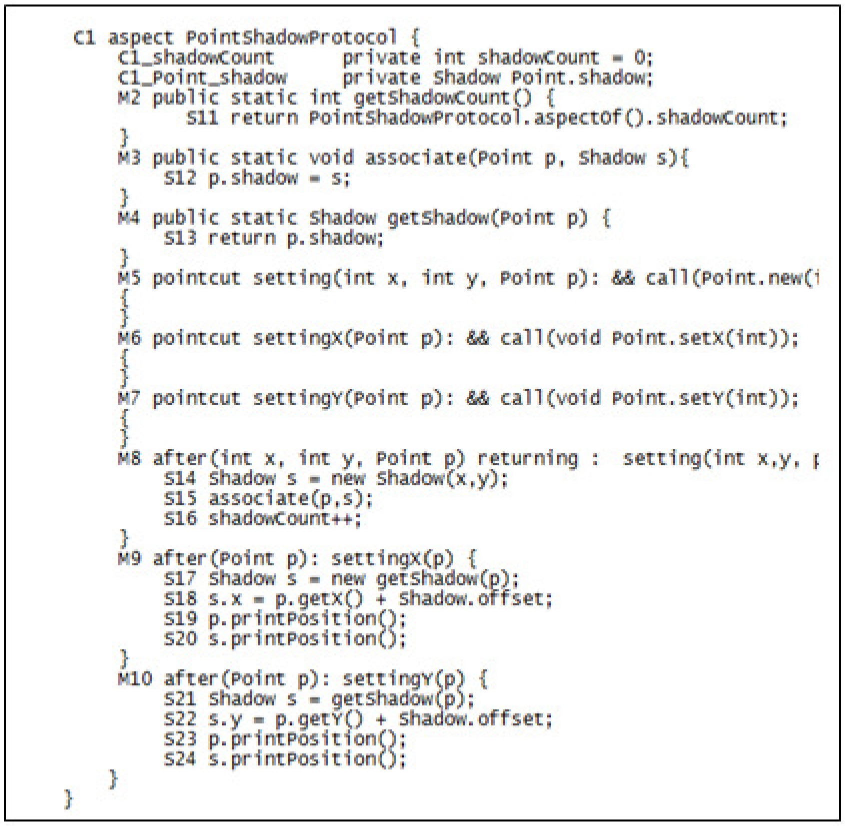
Figure 8: AOST Interface

In order to introduce example of AODFG, we used a modified version of an AspectJ program taken from [24]. Figure 9 gives the analysed AspectJ code from the program named ShadowTraker.We bring one aspect from the whole program named PointShadow-Protocol.

The analysed code is generated with the label C represent the class or aspect, M is for method and S is for statement. The program output is shown in figure 10. The AODFG contain control flow and data dependences edges between the AO nodes.

Figure 9 illustrates an AODFG for the running example, focus on after advice (M9). Since relevant control is associated with placeholder decision nodes, the algorithm can take into account the data dependencies between such nodes and the nodes that define the data. For example, the selection decision node S17 is data dependent on node M9. At the same time S17 have a control flow M9_p a reference to the receiver object of the crosscut call site, because this node represents the target pointcut that guards the execution of dynamic advice after (M9).

The graph represents a pointcut node (M5), call node M9), and the statement which is differentiate by the type of shape. The bull eye represents an aspect node. The control flow and data dependences are differentiate by type of edge. The detail type of shape can be referring to the legend in Figure 10.



```
C1 aspect PointShadowProtocol {
    C1_shadowCount      private int shadowCount = 0;
    C1_Point_shadow     private Shadow Point.shadow;
    M2 public static int getShadowCount() {
          S11 return PointShadowProtocol.aspectOf().shadowCount;
    }
    M3 public static void associate(Point p, Shadow s){
        S12 p.shadow = s;
    }
    M4 public static Shadow getShadow(Point p) {
        S13 return p.shadow;
    }
    M5 pointcut setting(int x, int y, Point p): && call(Point.new(i
    {
    }
    M6 pointcut settingX(Point p): && call(void Point.setX(int));
    {
    }
    M7 pointcut settingY(Point p): && call(void Point.setY(int));
    {
    }
    M8 after(int x, int y, Point p) returning :  setting(int x,y, p
        S14 Shadow s = new Shadow(x,y);
        S15 associate(p,s);
        S16 shadowCount++;
    }
    M9 after(Point p): settingX(p) {
        S17 Shadow s = new getShadow(p);
        S18 s.x = p.getX() + Shadow.offset;
        S19 p.printPosition();
        S20 s.printPosition();
    }
    M10 after(Point p): settingY(p) {
        S21 Shadow s = getShadow(p);
        S22 s.y = p.getY() + Shadow.offset;
        S23 p.printPosition();
        S24 s.printPosition();
    }
}
```

Figure 9 : Point Shadow Protocol Aspect Code

5. Evaluation of AODFG

In order to assess the effectiveness of AODFG analysis tool, some AspectJ program were analysed. Our study used ten benchmarks of AspectJ examples as shown in Table 1, 2 and 3[37] from the collections of AspectJ Development Tools (AJDT) plug-in with some modification code. Our concern is to look at the consistency of output between CFG, DG and DFG. We also will look at the extraction that we can get from AODFG compare to CFG or DG. For each program, table gives the numbers of aspect, LOC, methods, statement and AO denotes as aspect modules separately. LOC represents the value of lines of code included class and aspect files. We define pointcut as AO module even it did not contain any body code since the style of structure is same with module. We verified those AODFGs generated by the tool against a manual inspection of the graph and the associated analysed source code for each of aforementioned programs.

First, we extract the output data from CFG. As we describe in previous chapter, CFGs shows the relationship between the nodes represent either an assignment statement or a conditional expression that affect the control flow and the edges represent the possibility to transfer the control between statements. So the output that we need from execution is to identify any possibility transition between the edges and the flows of control. Table 1 show the output of AODFG execution and the output graph as shown in figure 11.

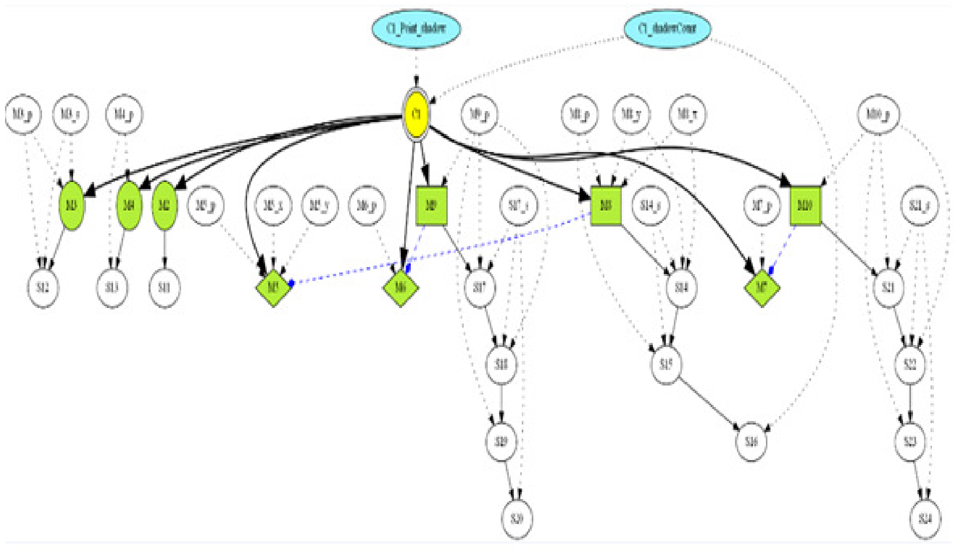


Table 1: CFG Analysis data

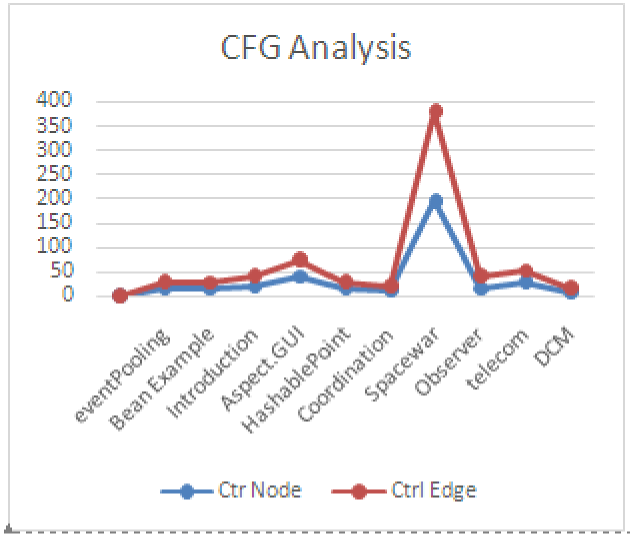


Figure 11: CFG Output Graph

Then, we use the same program to extract the output from DG. DG will shows the relationship among the statements in the program. So the output that we need from the execution of DG is relationship among data in the program. Table 2 show the representation of the same program in a DG representation view. Figure 12 represent the output from DG that we get from table 2.

Table 2: DG Analysis

Package/ Program	Aspect	Dep. Node	Dep. Edge
eventPooling	1	91	88
Bean Example	1	15	14
Introduction	3	19	25
Aspect.GUI	2	42	41
HashablePoint	1	15	14
Coordination	1	9	8
Spacewar	4	261	251
Observer	2	16	53
telecom	3	42	39
DCM	1	18	18

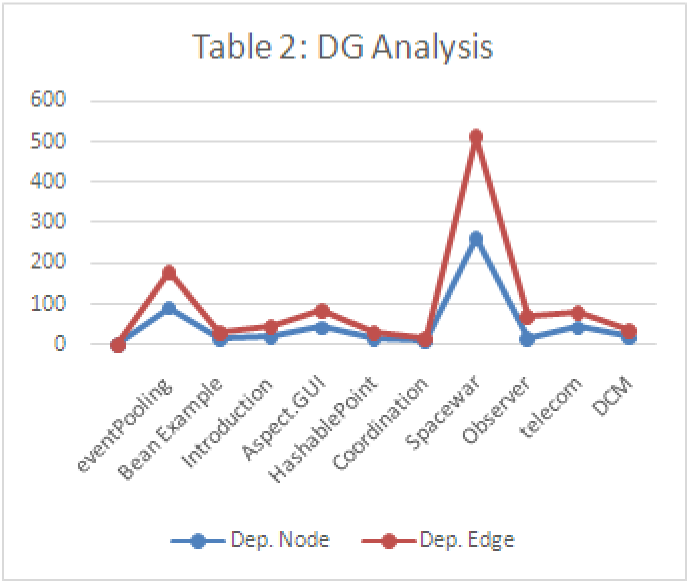


Figure 12: DG Output Graph

We repeatedly modify the source code with a minimal customization to help AODFG representation tool in their debugging process. For example, if we found any incorrect value of a variable that related with AO, we try to change to any suitable. We assume that the more LOC will make more complexity to the relationship of the program. From Table 3, The quantity of AO is not related with the value of neither methods nor statement. AODFG identify the AO features in the program, based on existing AO source code in the program. For example, Introduction with 234 LOC and 18 methods have 1 AO features in the program compare to Coordination with 448 LOC and three methods and three AO features.

Table 3: DFG Analysis

Package/ Program	Aspect	LOC	Method		DFG Edge	DFG Node	Formula	Spread
			OO	AO				
eventPooling	1	108	3	1	102	99	Edge > Node	3
Bean Example	1	159	14	1	27	15		12
Introduction	3	234	18	1	47	19		28
Aspect.GUI	2	101	0	8	77	72		5
HashablePoint	1	48	2	3	27	21		6
Coordination	1	448	3	3	17	13		4
Spacewar	4	636	0	40	438	377		61
Observer	2	243	16	2	35	40	Edge < Node	64
telecom	3	119	7	8	61	63		2
DCM	1	211	0	3	26	42		16

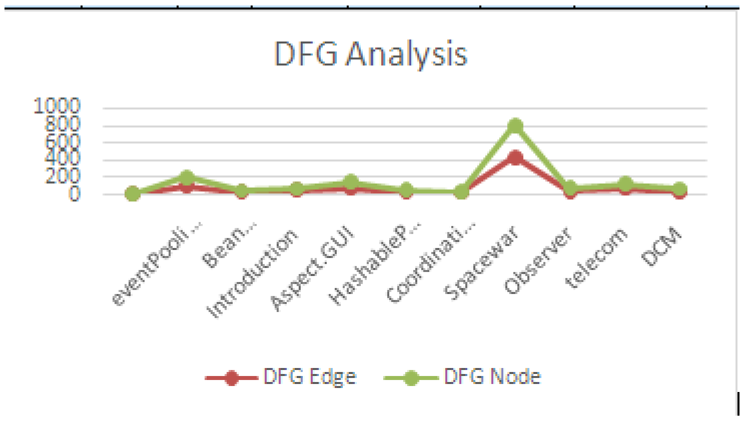


Figure 13: Output of DFG Analysis

From the experiment, we can see that program generated by the tool were correct and consistently show the same output as CFG and DG. The advantage AODFG are proposed all together DG and CFG in a single graph representation. In other words, we can get information about flow work list that can help us to get the executable flag and we also understand the dependencies among the object and aspect methods in the program.

It shows that, representing AO software by using AODFG provides a useful support for gaining a better knowledge of the internal structure even in the complicated programs, by reducing the effort needed to understrand the detail structure of the program. It just another way to represent the code structure that obtain more useful information which are dependencies among the program and its flow of control.

5. Conclusion and Future Work

An ideal program representation for aspect-oriented would have a local execution semantics from which an abstract interpretation can be easily derived. It would also be a sparse representation of program dependencies, in order to yield an efficient algorithm. Like a Necker cube, this representation will permit two points of view: It can be viewed as a data structure that provide dependence information, at the same time it can also be viewed as a precisely defined language with a local operational semantics. The dependence flow graph is just such a representation.

As future research, this kind of representation can be improved to a more mature and compatible tool for maintenance task in order to be applied in many kind of complex program with multiple language and environment. It is also good to plug-in this representation into Java IDE. Thus, developers can easily reverse and forward development activity to understand the program structure concurrently in the case they use AOP in their projects.

Reference

[1] Zhao, J., & Rinard, M. (2003). System Dependence Graph Construction for Aspect-Oriented Programs. Laboratory for Computer Science.
[2] Masuhara, H., Kiczales, G., & Dutchyn, C. (2002). Compilation Semantics of Aspect-Oriented Programs, 17-26.
[3] E. Baniassad and S. Clarke. Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley, 2005.
[4] I. Jacobson and P.-W. Ng. Aspect-Oriented Software Development with Use Cases (Addison-Wesley Object Technology Series). Addison-Wesley Professional, 2004.
[5] A. Rashid and R. Chitchyan. Aspect-oriented requirements engineering: a roadmap. In Proceedings of the 13th International Workshop on Software Architectures and Mobility, pages 35–41. ACM, 2008.
[6] M. Katara and S. Katz. Architectural views of aspects. In Proceedings of the 2nd International Conference on AOSD, pages 1–10. ACM, 2003.
[7] J.-M. J´Equel. Model driven design and aspect weaving. Software and Systems Modeling, 2008.
[8] Xu, D., & Xu, W. (2006). State-based incremental testing of aspect-oriented programs. Proceedings of the 5th international conference on Aspect-oriented software development - AOSD '06, 180. New York, New York, USA: ACM Press.
[9] Gold, R. (2010). Control flow graphs and code coverage. International Journal of Applied Mathematics and Computer Science, 20(4), 739-749.
[10] Ishio, T., Kusumoto, S., & Inoue, K. (2004). Debugging support for aspect-oriented program based on program slicing and call graph. 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., 178-187. Ieee.
[11] Mohapatra, D. P. (2008). Dynamic Slicing of Aspect-Oriented. Informatica, 32, 261-274.

[12] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J. M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. Transactions on Aspect-Oriented Software Development, pages 75–105, 2006.

[13] Pingali, K., Beck, M., Johnson, R., Moudgill, M., & Stodghill, P. (1991). Dependence flow graphs: an algebraic approach to program dependencies. Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '91, 67-78. New York, New York, USA: ACM Press.

[14] Yin, W., Jiang, L., Yin, Q., Zhou, L., & Li, J. (2009). A Control Flow Graph Reconstruction Method from Binaries Based on XML. 2009 International Forum on Computer Science-Technology and Applications, 226-229. IEEE.

[15] Muchnick S.S. (1997), Advanced Compiler Design Implementation. Morgan Kaufmann Publishers, USA.

[16] Beck, M., Johnson, R., & Pingali, K. (1991). From Control Flow to Dataflow. Journal of Parallel and Distributed Computing, 129(12), 118-129.

[17] Johnson, R., & Pingali, K. (1993). Dependence-based program analysis. ACM SIGPLAN Notices, 28(6), 78-89.

[18 Hovsepyan, A., Scandariato, R., Baelen, S. V., Berbers, Y., & Joosen, W. (2010). From Aspect-Oriented Models to Aspect-Oriented Code ? The Maintenance Perspective, 85-96.

[19] Munoz, F., Baudry, B., Delamare, R., & Le Traon, Y. (2009). Inquiring the usage of aspect-oriented programming: An empirical study. 2009 IEEE International Conference on Software Maintenance, 137-146.

[20] Johnson, R. C. (1994). Efficient Program Analysis Using Dependence Flow Graphs. Cornell University.

[21] Fuentes, L., & Sánchez, P. (2007). Towards executable aspect-oriented UML models. Proceedings of the 10th international workshop on Aspect-oriented modeling - AOM '07, 28-34, New York, USA: ACM Press.

[22] F. Würthinger, T. (2007). Visualization of Program Dependence Graphs. Johannes Kepler University Linz.

[23] Parizi, R. M., Azim, A., & Ghani, A. (2008). AJcFgraph - AspectJ Control Flow Graph Builder for Aspect-Oriented Software. Journal of Computer Science, vol. 3, pp. 170-181. Zhao, J. (2006). Control-Flow Analysis and Representation for Aspect-Oriented Programs. 2006 Sixth International Conference on Quality Software (QSIC'06), 38-48. IEEE.

[24] Johnson, R. C. (1994). Efficient Program Analysis Using Dependence Flow Graphs. Cornell University N.Y USA.

[25] Lin, Y., Zhang, S., & Zhao, J. (2009). Incremental Call Graph Reanalysis for AspectJ software. 2009 IEEE International Conference on Software Maintenance, 306-315. IEEE.

[26] Xu, G. (2007). Data-flow and Control-flow Analysis of AspectJ Software for Program Slicing. Program. Ohio State University.

[27] Cacho, N., Filho, F. C., Garcia, A., & Figueiredo, E. (2008). EJFlow : Taming Exceptional Control Flows in Aspect-Oriented Programming. AOSD'08 (pp. 72-83).

[28] Cao, W. (2008). A Software Function Testing Method Based on Data Flow Graph. 2008 International Symposium on Information Science and Engineering, 28-31. IEEE.

[29] Würthinger, T. (2007). Visualization of Program Dependence Graphs. Johannes Kepler University Linz.

[30] G. Carlos, J.M. Murillo, Pablo A. Amaya, Aspect Oriented Analysis: a MDA Based Approach, (2004), Quercus Software Engineering Group, University of Extremadura Spain.

[31] G. Kiczales, et al., Aspect Oriented Programming, (1997) Proc. Of ECOOP'97, LNCS 1241, Finland, 220-242.

[32] Elisa.Baniassad, Siobhan.Clarke, Theme: An Approach for Aspect-Oriented Analysis and Design,(2004) Trinity College, Dublin 'ICSE 2004'.

[33] Norman E. Fenton, Software Metrics: A Rigorous & Practical Approach 2nd Edition, 1997, page 280, PWS publishing Company, Boston UK.

[34]Jianjun Zhao, Slicing aspect-oriented software, In IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension, page 251, Washington, DC, USA, 2002. IEEE Computer Society.
[35]AspectJ Team: AspectJ Development Environment. http://www.eclipse.org/ajdt/
[36]D.Sereni, O.D.M., Static Analysis of Aspects, in AOSD'03. 2003, ACM: Boston, Massachusetts. p. 30-39.
[37]Hohenstein, U. D., & Gleim, Using aspect-orientation to simplify concurrent programming. Proceedings of the tenth international conference on Aspect-oriented software development companion - AOSD '11, . 2011.
[38]I. Kiselev, Aspect-Oriented Programming with AspectJ, SAMS, Indianapolis, p. 19.
[39][1] Chapter 1. What Is Aspect-Oriented Programming?, http://docs.jboss.org/aop/1.1/aspect-framework/userguide/en/html/what.html

Authors

Syarbaini Ahmad is a software engineering Phd student from University Putra Malaysia. His research study is about aspect-oriented program analysis and slicing specifically for maintenance purpose. He doing his M.S degree in real time software engineering at University Technology of Malaysia.

Dr. Abdul Azim Abdul Ghani (Prof.) received his M.S. degrees in Computer Science from University of Miami, Florida, U.S.A in 1984 and Ph.D. in Computer Science from University of Strathclyde, Scotland, U.K in 1993. His research areas include software engineering, software metric and software quality. He is now a professor in Department of Software Engineering and Information System, Faculty of Computer Science and Information Technology, University Putra of Malaysia.

Dr. Nor Fazlida Mohd Sani (Assoc. Prof.) is a lecturer in the Department of Computer Science at the Universiti Putra Malaysia. She received her Ph.D. from Universiti Kebangsaan Malaysia in 2007. Her current research interests are authentication security, information security, program understanding, and program analysis. Dr. Nor Fazlida's research has been published in various international journals and conferences, and frequently serves as a technical editor for journal and program committee for international conferences.