Journal of Computer Science

Original Research Paper

Effectiveness Analysis of Aspect-Oriented Dependence Flow Graph as an Intermediate Representation Tool

[1]Syarbaini Ahmad and [2]Abdul Azim A. Ghani

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

Article history
Received: 29-10-2017
Revised: 08-01-2018
Accepted: 16-02-2018

Corresponding Author:
Syarbaini Ahmad
Faculty of Science and Information Technology,
International Islamic University College Selangor (KUIS),
43000 Kajang Selangor
Malaysia
Email: syarbaini@kuis.edu.my

Abstract: Graph-based representations of programs such as control flow graph or dependence graph have been used to support program analysis tasks such as program comprehension and software maintenance. However, in the case of Aspect-Oriented Programming (AOP), such graph representations individually is not enough to represent the features of aspect-oriented programs because it could need to identify the flow of control and the relationship of the data. AOP is a technique for improving modularity by separating crosscutting concerns in software development. In this article, a graph model known as Aspect-Oriented Dependence Flow Graph (AODFG) is proposed to represent the structure of aspect-oriented programs. The graph is formed by combining control flow graph and dependence graph into a single graph representation. As a consequence, more information about dependencies involving the features of AOP, such as join point, advice, aspects, their related constructs and the flow of control are able to be analysed. Effectiveness analysis of AODFG has been conducted in an experiment involving twenty software experts applying the graph on the AspectJ benchmark programs. The findings show that they were very satisfied with AODFG as an effective tools for analysing code.

Keywords: Aspect-Oriented Program, Control Flow Graph, Dependence Graph, Program Analysis, Du-Chains, Ud-Chains, Code Analysis

Introduction

Graph-based representations for programs are useful in supporting program analysis tasks such as program comprehension and software maintenance. Traditionally, control flow graph and data dependence graph are used to model the flow of control and flow of data in programs respectively. With the advance of Aspect-Oriented Programming (AOP) as a means for handling modularization of software systems by reducing the tangling and scattering of crosscutting concerns, the traditional source code representations models are inadequate to model features of AOP such as join point, pointcut, advice, introduction and aspect. To ameliorate this inadequacy, varieties of code representations for AOP have been proposed in the literature, such as Aspect-Oriented System Dependence Graph (ASDG) (Zhao, 2002), Inter-procedural Aspect Control Flow Graph (IACFG) (Bernardi and di Lucca, 2007) and Aspect-Oriented Control Flow Graph (AOCFG) (Parizi and Ghani, 2008).

In this study, we propose an intermediate code representation called Aspect-Oriented Dependence Flow Graph (AODFG) to support program analysis of aspect-oriented programs. This graph has been formed by combining aspect-oriented control flow graph with aspect-oriented dependence graph. In order to get the benefit from this graph, we performed an effectiveness analysis in using AODFG with its tool support. The experts involved in the analysis show satisfactory results.

The rest of the paper is organized as follows. In section 2, we describe the conceptual design of AODFG. In section 3, we present the concept of dependence flow graph. In section 4, we present the construction of AODFG. In section 5, we present the validation and its findings. In section 6, related work is discussed. Finally in section 7, we present the conclusion.

Aspect-Oriented Dependence Flow Graph Conceptual Design

Aspect-oriented Dependence Flow Graph (AODFG) is a code representation tool to represent the graph of

control and dependency simultaneously. It is a technique to show the relationship of control and data dependency in a single graph (Ahmad *et al*., 2014). The initiatives of AODFG are coming from literature of control flow graph and data dependence graph of aspect-oriented programs.

The conceptual design of AODFG is shown in Fig. 1. It shows that there are two different targets that can be extracted from a programming code. One is the flow of control and another one is the dependencies of the data. The flow of control can be represented by using control flow graph which is good to show the statements or variables flow of event from one to another node in the program. Another target is to know the dependencies among the nodes in the program. This is good to show the data dependencies relationship. The edge in the CFG and DG are the transition of the data and flow of control.

It is more about '*du* and *ud* chain' as a relationship among the nodes and edge in the program structure. The '*du* and *ud* chain' will be useful for analysis control flow graph and data dependence graph.

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* (Pingali *et al*., 2003). The standard definition of *du-chains* and *ud-chains* are as in definition 1 and 2.

Definition 1

A definition of node 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 (n_1, n_2) *such that* n_1 *defines x,* n_2 *uses x and the defines of x at* n_1 *reaches the uses of x at* n_2.

Definition 2

A definition of variable x is said to reach a 'define' 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(n_1, n_2) such that n_1 uses x, n_2 define x and the uses of x at n_1 reaches the defines of x at n_2.

Figure 2 shows the examples of implementation the two definitions. Figure 2a is example of C code. Beside the program is the defined *def-use* in Fig. 2b for representation of control flow graph and (c) is representation for dependence graph. Figure 2c is combination of the CFG and DG that produces DFG. In Fig. 2b, nodes are representing either assignment statements or conditional expressions that affect flow of control and edges represent possible transfer of control between nodes. An assignment node has a single successor, while a conditional node has two successors representing the possible branching of control.

Def-use chains for dependence graph are graphs that have the same nodes as control flow graphs (Pingali *et al*., 2003), but the edges connect each definition of a variable to all uses reached by that definition. In Fig. 2c, edges in the graph represent dependencies that are classified as flow (*def-use*), anti (*use-def*), or output (*def-def*) dependences. Note that the data dependence graph is not an executable representation and does not incorporate information about flow of control.

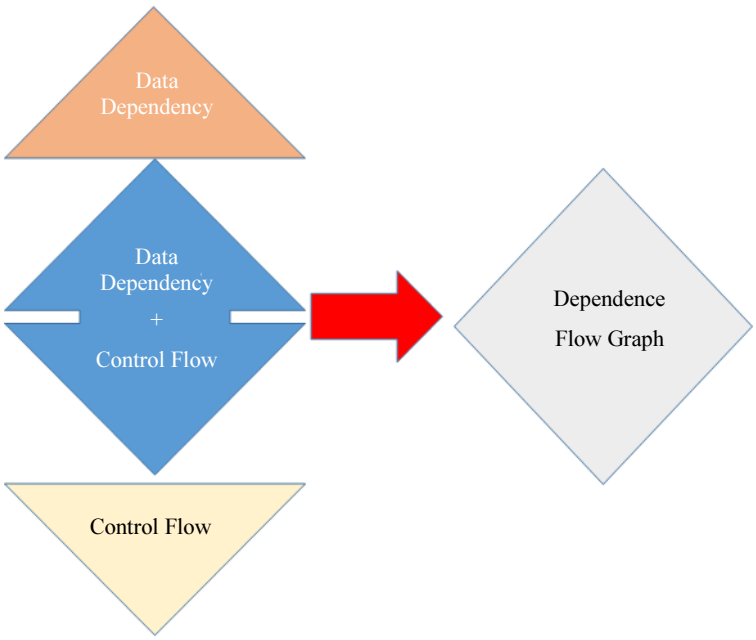

Fig. 1: Overview of AODFG original concept study

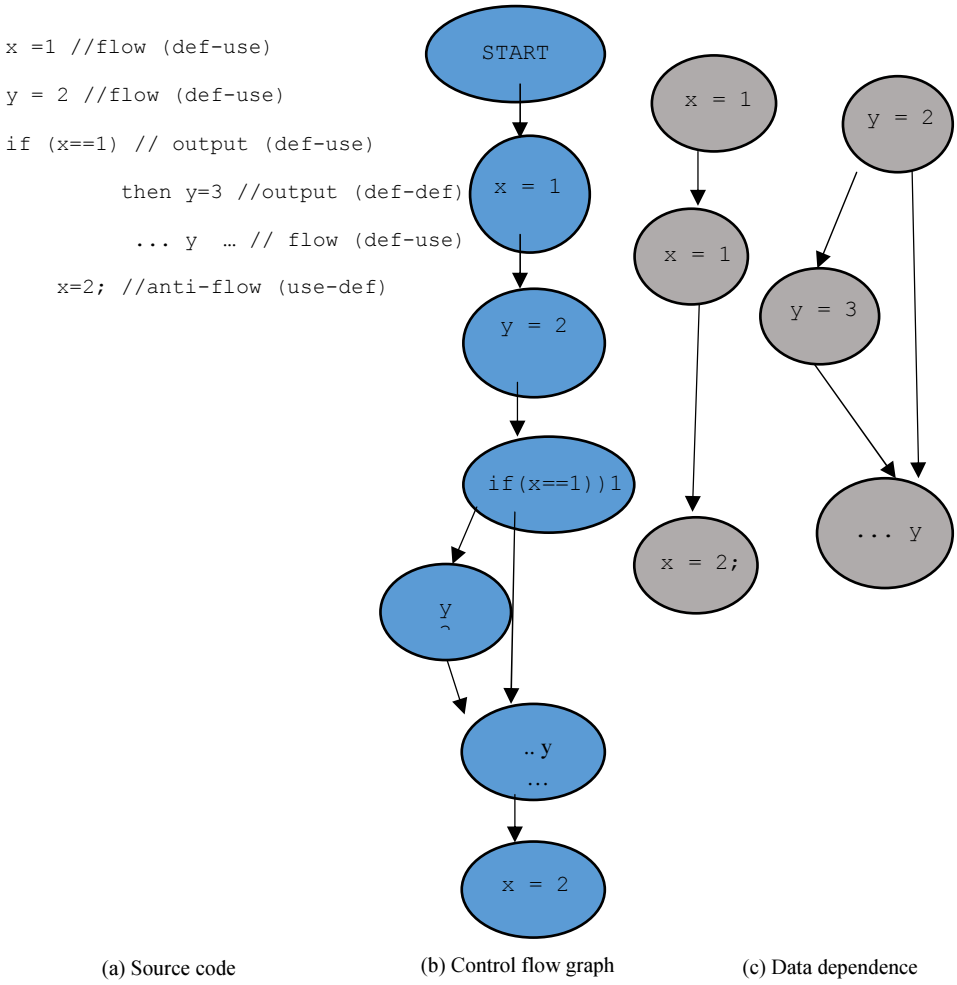

Fig. 2: Example program and its representation

DFG and AODFG Construct

This section presents the conceptual study of DFG as a whole. It depicts the construction of DFG as it is coming from the bringing of CFG and DG into one single graph. This means that DFG is a consecutive of hybrid relations that consist of flow of control and data dependencies between class, method or statement into a single graph representation.

A control flow graph is a directed graph where a node represents a basic block and edge is a flow of control between one to another block. Control flow is the sequential of instructions that are executed in a program (Ahmad *et al*., 2014). The structure reflects the iterative and looping data in the nature of programs (Bernardi and di Lucca, 2007). It is a sequence of consecutive of statements starting from the early stage of statement definition until it completes the process. In CFG, if there is an edge between the node x and y, y is defined as a successor of x and x is the predecessor of y.

Data dependencies are program statements that have a dependency with other program statements. The DG graph is the statements and predicate expressions that can be characterized by the nodes (Arora *et al*., 2012). A flow of dependences are representing in graph called dependence graph contains nodes and edges. Nodes represent either, method, or statements in the program. Edges represent data dependencies among method and statements. The relational between one to another nodes are using two types either using *du* or *ud* chains.

Most researchers (Parizi, 2008; Weiser, 1979; Jia *et al*., 2008; Lallchandani and Mall, 2009) generate a control flow graph as a first step towards computing dependence graph. In software maintenance, both control flow and dependence of the data are useful and recommended. This is the idea of the development of hybrid algorithm that uses the data structures together. The flows of control and data are not independent. They are following and relating sequentially from one to another Line of Code (LOC). But, there are few analyses needed to

understand the flow in *def-use* and *use-def* definition as discussed previously.

The *def-use* chain can identify which nodes may execute the control statement in CFG. From the chain in Fig. 2, there are two work list keep track on this execution as flow work list and *def* work list. The flow work list is used to propagate the executable flag through the control flow graph. If a non-conditional node *N* may be executed, then its successors may be executed. Once the predicate of a conditional has been assigned a value, the executable flag can be propagated down one or both sides as appropriate. The *def* work list is used to propagate values along *def-use* edges.

The problem of maintaining two data structures to represent the program's execution semantics and its dependencies is addressed in part by the program dependence graph. This graph consists of the data dependence graph augmented with control dependence arcs. A more elegant algorithm can be developed using the program dependence graph. However, program dependence graphs inherit another problem to the data dependence graph such as constant propagation which needs an execution to perform the program transformation. This problem will counter by Dependence Flow Graph (DFG). DFG has a capability to represent an execution of semantic and its dependencies. It also can view the data structure and easy to verify the dependence arc. Another point is DFG is executable and the semantic is generalization of the data driven execution.

To understand dependence flow graphs, it is useful to execute the graph depicted in Fig. 3 as a simple example of execution process. Execution begins by pushing token when the START operator sends a token to the store operations $x = 1$ and $y = 2$. Depending on whether the token received on *arc b* is true or false, the switch operator outputs the token it receives on *d*4 onto either *arc d*5 or *d*6. In the example, the switch routes the token to *d6* and the definition strictly merge is executed. The merge operator receives a token on either one (but not both) of its inputs and simply outputs this token. The reader can verify that a token carrying the value 3 will be generated on *arc v*1.

From Fig. 3, it can be seen that DFG simplifies the two different kind of graph representations and compresses them into one single graph representation without dropping the information gathered by the graphs (CFG and DG). Hence, it is a hybrid process that carries two different graphs into a single graph representation.

Creating a Control Flow for Aspect-Oriented

Control flow analysis is one of the phases to represent the AODFG graph for aspect-oriented program. Control flow is the sequential of instructions that are executed in a program. AOCFG is a standard CFG that model the control flow within Java classes (including AspectJ), within aspects and across boundaries between aspects and classes through non-advice method calls and iterative data flow that model the interactions between methods and advices at join points (Xu and Rountev, 2007). Iterative data flow is a key point to work with AOCFG. Iterative data flow analysis has a capability to discover the loop process in the code structure. Then, control flow analysis can characterize the flow of programs. So that, any unused generality can be removed and the related and important code will be classified into related group. The rules of classification are reffered to the following definitions.

Definition 3

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

- The source of e_1 defines x
- The destination of e_2 uses x
- There is a control flow path from e_1 *to* e_2 with no Assignment to x

Furthermore, edges ($n_{a'}$, n_{out}) were added for each node $n_{a'}$ 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 $N = \{n_{in},n_{out}\}$ and $E = \{(n_{in},n_{out})\}$.The node n_{in} is the only entry node and the node n_{out} is the only exit node of the control flow graph. Note that the control flow graph G_f is a graph where each node (except n_{in} and n_{out}) corresponds to one statement in the function *f*.

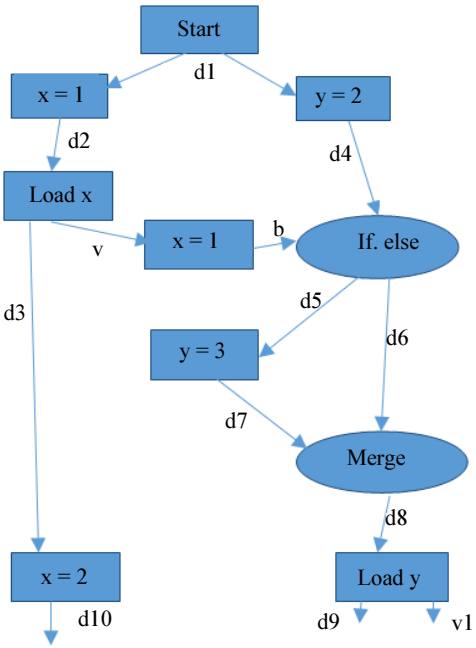

Fig. 3: Example of DFG for a small program m

When analysing the source code statically, the technique used in getting AOCFG starts by discovering its data structure such as *if*, *else* and *if-then-else* with a *loop* in the intermediate code. Then, the code is extracted line by line and represents it visually in flowchart to make it clearer to the eye. Next, basic block is identified to show a straight line sequence of code which has an entry and exit point. The characteristics of producing basic block are as in Definition 4.

Definition 4

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 the 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.

AODFGs model use AOCFG as a part of the analysis phase. Specifically, for selections, guards, synchronisations and input event nodes, the path where the condition is unsatisfied is not explicitly represented in AODFG. In AOCFG, all such paths would be represented as a *false* branch from the node. For selections, when the condition is unsatisfied, the thread terminates, so an end node must be introduced and the false branch must link to it. For guards, synchronisations and input event nodes, the control flow waits until the condition becomes satisfied, so the false branch must revert back to the node itself.

A control flow graph consists of a sequence of nodes of method $< m_0, m_1 ... m_k >$, where, for every mi, such that $0 \leq i < k$, $(m_i, m_i+1) \in E$. The statement s consists of condition in the method, where $m(s_1, s_2, \ldots s_n)$. But, not all methods have its statement. Due to these differences, instead of using an AOCFG as a part of AODFG, it must be transformed into a new structure since there are some features in aspects are included into the analysis. The features are *join point, pointcut, aspect, advice* (*before, after and around*) *and introduction*. All the features above should be defined as aspect node (Ahmad *et al*., 2014).

Figure 4 is an example of AOCFGc(n_a, n_a') to get more understanding on the architecture. If the statement *a*' is executed immediately after the statement *a*. For the first statement a_1 in the function, AOCFGe (n_{in}, n_{a1}) is introduced. Furthermore, AOCFGs (n_a', n_{out}) were added for each node n_a' 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 AOCFG of an empty function, i.e., a function without any statements consists of $N = \{n_{in}, n_{out}\}$ and $E = \{(n_{in}, n_{out})\}$.

The node n_{in} is the only entry node and the node n_{out} is the only exit node of the control flow graph. Note that the control flow graph G_f is a graph where each node (except n_{in} and n_{out}) corresponds to one statement in the function *f* (Gold, 2015).

Each node in AODFG is represented by a node in the corresponding to the original control flow graph. Control flow graphs additionally have *end* nodes which do not correspond to AODFG or DFG nodes. A part from *end* nodes, the nodes in control flow graphs retain all the information of the corresponding AODFG nodes, such as their component names and types. The term *node* will be used to refer to AOCFG, AODG and AODFG. The following steps are used to construct an AOCFG:

- Create a node in the control flow graph to represent the root node of the AODFG
- For each node *n* in the AODFG which has a corresponding node *m* in the control flow graph, locate each of the children of *n* in the AODFG. For each child, place a node *c* into the control flow graph, with an AOCFG from *m* to *c*. In this manner, a control flow graph node will be created for every AODFG node, with edge representing the arrows in the AODFG
- For a single sequential node *n* in the AODFG, locate its corresponding node in the control flow graph *m*. Then, label all of the outgoing AOCFG of *m* as *true*. Insert an additional outgoing AOCFG from *m* to a new *end* node. The AOCFG represents the semantics of selection nodes. If the condition of the selection is satisfied, the control flow may proceed to all subsequent nodes; otherwise the control flow for this thread terminates. Figure 5 is an example of sequence representation of nodes in the program structure
- For each guard, synchronisation node or input event node (both external and internal event types) in the AOCFG, locate its corresponding node in the control flow graph *m*. Label all of the outgoing AOCFG of *m* is *true*. Insert an additional outgoing AOCFG from *m* back to itself, labelled *false*. See the following diagram as an example (Fig. 6). If a synchronisation node is also a conditional node, it will have two *false* of AOCFG in the control flow graph: one representing the *false* case of the condition and one for when the synchronising partners have not yet been reached

```
Void search(int arr[], int key,
            int *found, int *index)
{
  int i = 0;
  int b;
  *found = 0;

while (i < N)

 {
  if (b = isabsequl (arr[i], key))
  {
     *found = b;
     * index = i;
     Return;
   }
   i++;
  }
}
```

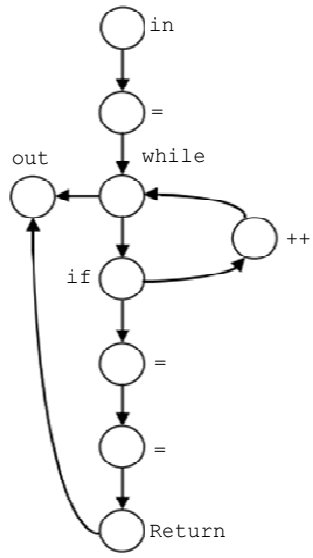

Fig. 4: Example of CFG definition node

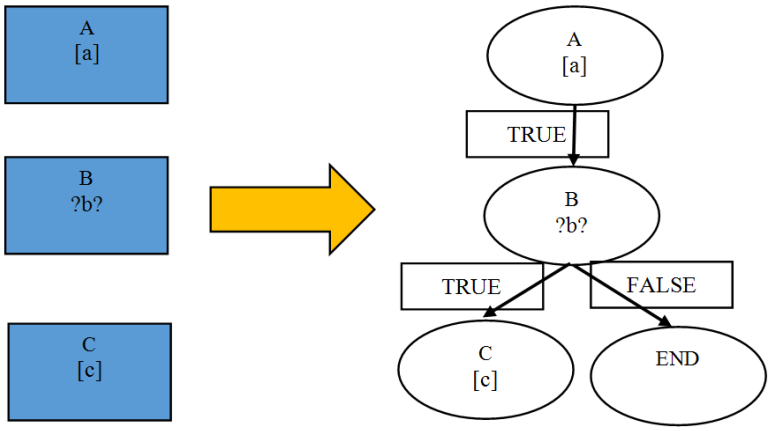

Fig. 5: Example of representing a selection node

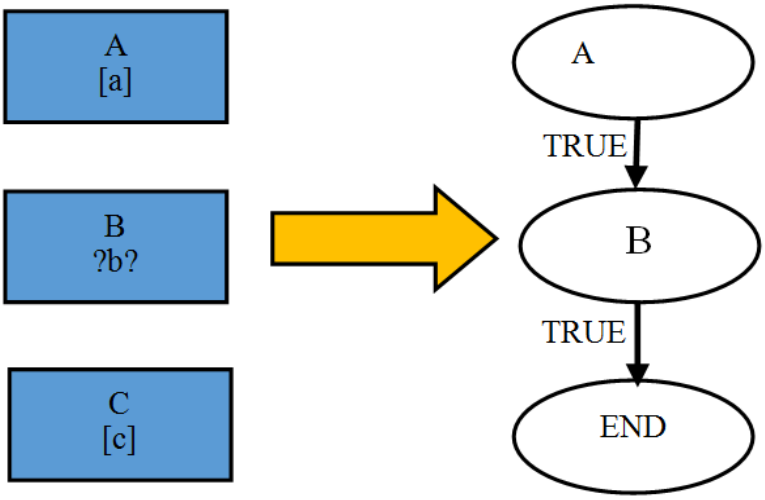

Fig. 6: Example of representing a guard node

■■

Dependence Graph in Aspect-Oriented

The dependence graph for aspect-oriented is a digraph that consists of a number of aspect code dependency such as advice, an introduction, or a method in the aspect and some special kinds of dependence arcs to represent direct or indirect dependencies between a call and the called advice, introduction, or method and transitive interprocedural data dependencies in the aspect (Zhao, 2002). Dependence analysis in AODFG is a construct of dependence graph that represents the dependences in AODFG for aspect-oriented programs. The difference with common dependence graph does not present the dependencies on itself but the information from the dependence analysis of another perspective which is AODFG. The purpose of dependence analysis is to determine the ordering relationships between instructions that must be satisfied for the code to execute correctly.

A dependence graph is a directed graph between statements S_1 and S_2 where S_1 is definition of variable v and S_2 is the uses of variable v. There is a path from S_1 to S_2 and v is not redefined. The definition of v in S_1 reaches the use of v in S_2. If statement S_2 is flow dependent on statement S_1, then S_1, S_2 is known as *def-use*.

Compared to the 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 S execution. If S_1 precedes S_2 ($S_1 \prec S_2$) in their execution order, means S_2 is dependence of S_1. There are 4 types of data dependences (Yatapanage *et al*., 2010).

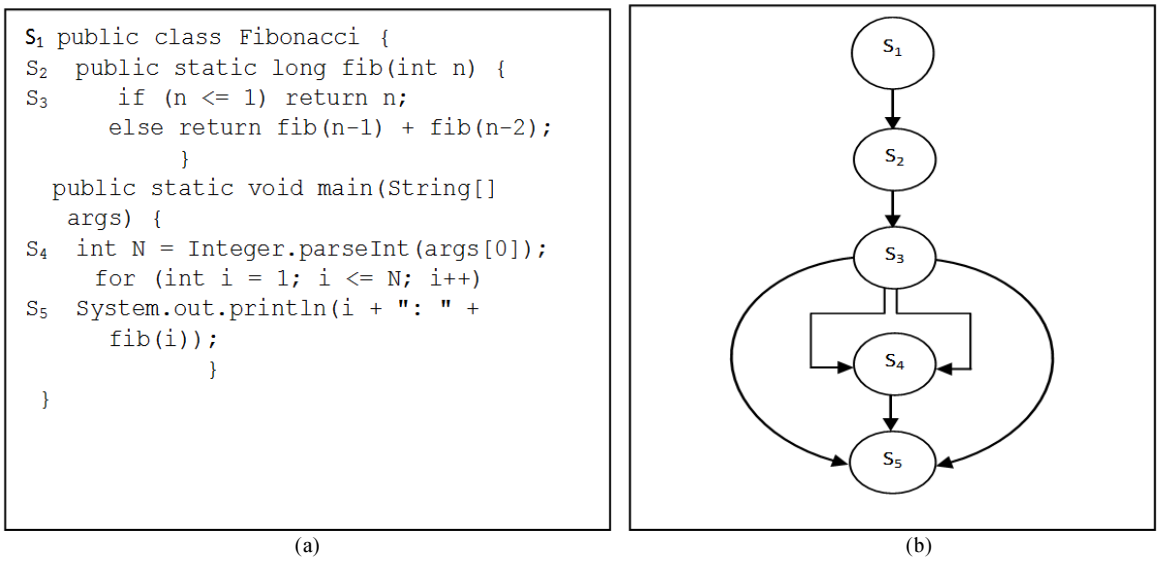

Fig. 7: Example of dependence graph

Definition 5

The character type of dependencies:

Type1: *Flow dependence/true dependence; If* $S_1 \prec S_2$ *and the former sets of value that the later uses*

Type2: *Anti dependence; If* $S_1 \prec S_2$, S_1 *uses some variable's value and* S_2 *sets it*

Type3: *Output dependence; If* $S_1 \prec S_2$ *and both statements set the value of some variable*

Type4: *Input dependence; If* $S_1 \prec S_2$ *and both statements read the value of some variable*

Figure 7 is an example consists of the four types of dependence as explained. Figure 7 a is simplified of the analysed code and Fig. 7b is dependence graph of Fig. 7a. The flow dependence between S_3 and S_4 is *Type1* when the former sets a value that the latter uses. In the reverse order, S_3 uses some variable's value (e) and S_4 sets it as *Type2*. S_3 and S_5 are set the value of some variable which is mentioned in *Type 3*. *Type 4* are dependence between S_3 and S_5 since both read the value of e.

Data Dependence

Data dependence is defined as a node that represents the program statements and edges that represent data dependencies between statements. A node is data dependent on another one if it refers to the state of a variable (component or attribute) that the other node defines or updates. For example, a selection node button pushed would be data-dependent on a state realisation node Button (pushed) or even a node Button (released).

Definition 6. Data Dependence

For two nodes p and q in a control flow graph, node q is data-dependent on node p, ($p \rightarrow q$), if:

- $\exists\ c \in DEF(p)$ such that $c \in REF(q)$
- $\exists\ \pi = trace(p, q)$, where $\forall\ k \in \pi$, $c \notin DEF(k)$ and • $!(conc(p, q))$
- $\neg(conc(p,q))$

Implementation of AODFG

This section enhances the extension of the construction of both control flow and data dependency to come up with AODFG. Our approach, in the case of aspect-oriented, shares the same viewpoint with procedural (Pigoski, 1997) and Object-Oriented (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 (Capilla *et al*., 2010).

The construction of AODFG is the arrangement from the original study of DFG as explain in previous section It was applied on traditional programming using *du* and *ud* chain. In order to tailor it with aspect-oriented programming, some additional features need to be injected to the steps of construction. The followings are the steps for the creation of AODFG:

- Analyze the control flow structure of the program as the technique used in CFG. The control flow graph $G_f = (N, E)$ of a function f has one node $n_a \in N$ for each statement a in f and two additional nodes n_{in}, n_{out}. Adding an edge (n_a, $n_{a'}$) if the statement a' is executed immediately after the statement a. For the first statement a_1 in the function, an edge (n_{in}, n_{a1}) is introduced. Furthermore, adding edges ($n_{a'}$, n_{out}) for each node $n_{a'}$ 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 $N = \{n_{in}, n_{out}\}$ and $E = \{(n_{in}, n_{out})\}$.The node n_{in} is the only entry node and the node n_{out} is the only exit node of the control flow graph. Note that the control flow graph G_f is a graph where each node (except n_{in} and n_{out}) corresponds to one statement in the function f
- Analyze the dependencies among the statements in the program as a technique used in dependence graph. The character type of dependencies such as flow dependence/true dependence; If $S_1 \prec S_2$ and the former sets of value that the later uses. Anti-dependence; If $S_1 \prec S_2$, S_1 uses some variable's value and S_2 sets it. Output dependence; If $S_1 \prec S_2$ and both statements set the value of some variables. Input dependence; If $S_1 \prec S_2$ and both statements read the value of some variables
- Using AspectJ as a target language and advice execution as a method call. The features of AOP introduced are the followings:

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, a dependency edge has been connected from an advice to pointcut

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

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

Findings and Validation

This section presents an experiment for validating the proposed theory for aspect oriented programs using dependence flow graph (Gallagher and Lyle, 1991). We used an improvement-oriented software maintenance model that was used by Gallagher to validate his approach using program slicing in software maintenance. The reasons were that his model has a capability to illustrates a comprehensive approach attempting to integrate the software maintenance process in a single software life cycle framework.

The validation process is started by stating the improvement goals of the representation process. In this case, the goal is AODFG effectively useful for representation of the aspect-oriented programs. Validation goal is specified with the object, to propose an AODFG implementable as a representation technique. The purpose is to learn if the representation of AODFG can be implemented in the aspect-oriented programs.

To identify whether this research has achieved the goal or not, the issues were focused on the effectiveness of making changes to aspect-oriented given program. The subjects for the experiment were twenty software developers. All of them had wide experiences as practitioners in the software development. We wanted to look at the change occurs after the treatment. Thus, the subjects were randomly given one sample of the AO program with average 100 to 500 LOC with less than ten

classes or *aspects*. They had to describe or represent the relationship among the code in the program with some limited time, based on their own experiences and their own maintenance tool.

The subject were given a short description on the aspect-orientation methodology. They also could view the related graph and perform a tutored practice to become familiar with our analysis technique and aspect-oriented nature. During the observation, the subjects represented the aspectJ program by using their own technique. Then, we were given the treatment by consult about OADFG and demonstrate the proposed technique as an alternative of code representation. When the subjects passed through the experiment's treatments, the subjects were repeat the first observed by AODFG representation. The AODFG graph presented some information about the code dependencies and region. Since it is difficult for editing and writing activities, merely report was generated from the beginning until compilation. This can be loosely constructed as the time to design and implement the change.

Our study used ten benchmarks ((eclipse.org//) of AspectJ examples as shown in Table 1-3 from the collections of AspectJ Development Tools (AJDT) plug-in with some modification code to suite with our analysis technique. Our concern is to look at the consistency of output between CFG, DG and DFG. We also looking at the extraction from AODFG compare to CFG and 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.

Table 1: CFG analysis data

Package/program	Aspect	Ctr node	Ctrl edge
Event pooling	1	16	14
Bean example	1	15	13
Introduction	3	19	22
Aspect.GUI	2	40	36
Hashable point	1	16	13
Coordination	1	12	9
Spacewar	4	197	183
Observer	2	16	27
Telecom	3	30	22
DCM	1	8	8

Table 2: DG analysis

Package/program	Aspect	Dep. node	Dep. edge
Event pooling	1	91	88
Bean example	1	15	14
Introduction	3	19	25
Aspect.GUI	2	42	41
Hashable point	1	15	14
Coordination	1	9	8
Spacewar	4	261	251
Observer	2	16	53
Telecom	3	42	39
DCM	1	18	18

Table 3: DFG analysis

			Method					
			---	---				
Package/program	Aspect	LOC	OO	AO	DFG Edge	DFG Node	Formula	Spread
Event pooling	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
Hashable point	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

First, we extract the output data from CFG. CFG 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 Fig. 8 is graph represent the output of analysis.

Then, the same program were used to extract the output from DG. DG will show 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 showed the representation of the same program in a DG representation view. Figure 9 represent the output from DG that were getting from Table 2.

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 by assuming the more LOC in the program the more complexity to the relationship will be work on. 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 one AO features in the program compare to *Coordination* with 448 LOC and three methods and three AO features. Figure 10 is the output represented in graph to look in the statistical view.

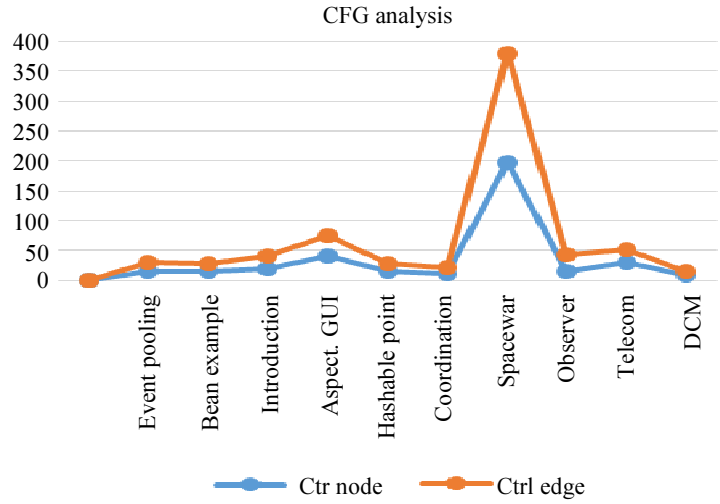

Fig. 8: CFG output graph

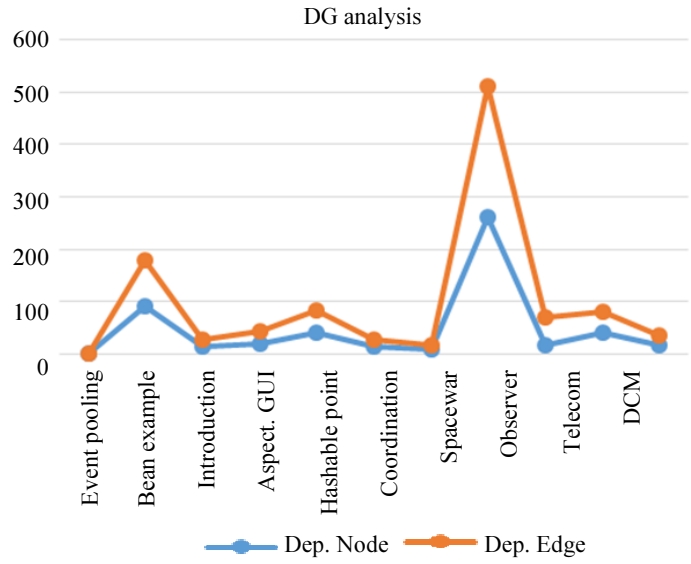

Fig. 9: DG output graph

■■

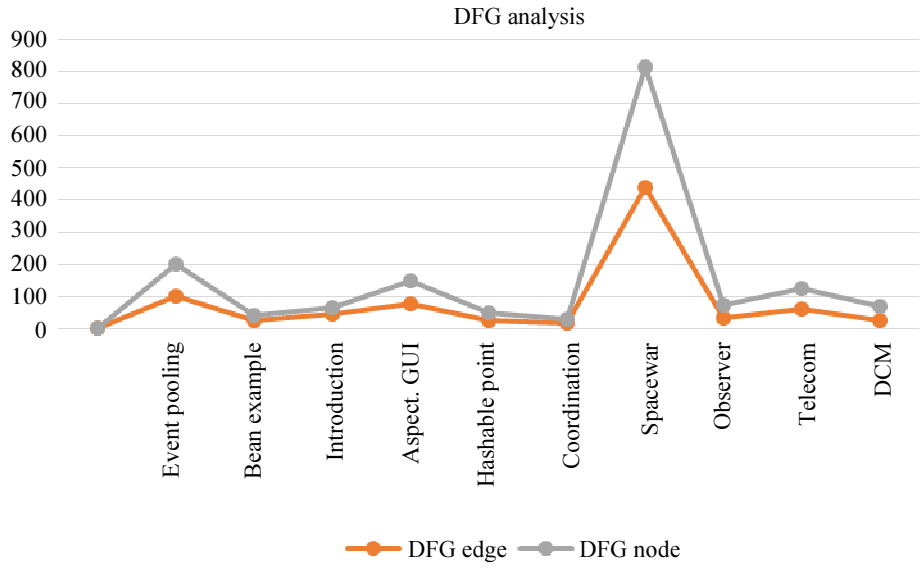

Fig. 10: Output of DFG analysis

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 is 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.

The subjects were given a questionnaire as a proven of the goal of this study. Twenty five questions were asked to the same sample regarding the effectiveness of AODFG. The set of questions related to effectiveness were: Does AOST effectively in representing the code? Does AOST help you in slicing the code? Does AOST help you in maintaining the software? Does AOST help you save your time in modifying the program? and Do you think that AOST can help software maintainer to solve complexity problem in software maintenance?

Descriptive statistical analysis was used to summarize and describe our collected data. Descriptive statistics are very important to this research because it can enable us to present the data in a more meaningful way, which allows a very simple interpretation of the data but very easy to understand. The survey was given to twenty respondents among the experienced practitioners as mentioned before. They were given the questionnaire after they had completely followed the training on aspect-oriented and felt the usability of AOST. The results from the questionnaire are given in Table 4.

Table 4 shows a set of questions to survey the effectiveness of AODFG as a presentation of aspect-oriented programs. Since effectiveness is related to the functionality of the system, question 1 until 5 is to know the respond from our subjects. The outputs are 80% agree that AODFG effectively represent the code architecture and 85% agree that AODFG can help software maintainer slice the code safely. 85% agree that AODFG help them to maintain the software and 90% said AODFG help them save their time in modifying the program. Lastly, as the end of the effectiveness point, question 5 asks either AOST can help software maintainer to solve complexity problem in software maintenance or not, where 85% agreed that it was helpful.

The twenty subjects agreed that they were very satisfied with AODFG in order to use it as one of the software development support tools. It is effective in aspect of information provider, time saving and complexity problem solving in tracking the relations of the code, especially in a big and medium size of LOC. The AODFG can help software maintainer not only for identifing the program in graph, but also can help them effectively analyse the aspect-oriented program with a very minimum effort.

Related Work

Gold (2015) redefined the control flow graph by giving a uniform definition by subsumes the reduction of abstraction of segment graph, directed graph and program graph. It is used to define statement coverage and branch coverage such that coverage notions correspond to node coverage and edge coverage. It can help software engineers to analyse the control flow of the program analysis in the design of test cases in software engineering. It then improves representation by focusing to the paths of control flow since executions of programs are represented by paths (Gold, 2014). The composition of reductions makes a stepwise analysis approach to the program. It is possibly not restricted to control flow graphs.

Bernardi and di Lucca (2007) introduced an inter-procedural aspect control basic system which focusing on non-weaving aspect code. It can make the maintenance of a software flow graph representing the flow and relationship of the program. The proposed research discussed on representing an AOP system by using Inter-procedural Aspect Control Flow Graph (IACFG), reporting the way aspects interact among the components of the program. They are referring to Zhao (2002) as their anchor study. The idea proposed was to allow an easier identification of the impact between aspects and the base code structure. They showed their findings with a tested code and figured out the customization of the CFG graph known as IACFG. Nevertheless, there still have to improve the accuracy of the graph with respect to both polymorphic calls and interceptions.

Singh *et al*. (2016) proposed a parallel dynamic slicing algorithm for distributed aspect-oriented programs by introducing parallelism into a slicing algorithm to make the slice computation process much faster. DDG generator is a tool that used to generate the required intermediate graphs for distributed of aspect-oriented programs. This proposed slicing technique is compared with one related existing technique using three case studies. The experiment is to look at the time consuming on generates precise slices compared to other three existing algorithm.

Ohmann and Liblit (2013) provided an extended core-dump information for debugging by minimizing overhead effort in analysis debugged program. The goal is to aid during post-deployment debugging by giving programmers additional information about program activity shortly before failure. Latent information in post-failure memory dumps, augmented by low-overhead, tuneable run-time tracing were used in the experiment activity.

The complexity of the program is very related to the algorithm that use control flow graphs and dependence graph. Arora *et al*. (2012) also compare the features of control flow and dependence flow of representation. They show that, dependence graph supports features like control, data and transitive dependence, single and multiple procedure, inter and intra procedure calls, multiple types of edges, slicing, context sensitivity, inheritance and polymorphism, test case generation and parameter passing. Whereas flow graph be deficient in representing data and transitive dependence, multiple procedures, inter and intra procedure calls, multiple types of edges, slicing, context sensitivity, inheritance and polymorphism etc.

Mohanty *et al*. (2015) applied aspect-oriented reverse hierarchical dynamic slicing algorithm on the intermediate program representation to compute the dynamic slices. This approach constructs the graph and computes the dynamic slices level wise. However, the complexity of the program is related to the algorithm that not only concerns about dependence of the data.

Conclusion and Future Work

We have applied the AODFG on a benchmark aspect-oriented program and let the right person to test the prototype. Then, the collections of data from the subjects were analysed to look for the effectiveness as a representation tool. From the experiment that has been done, it shows that AODFG definitely provides the same value of nodes, data edge and control edge compared to the dependence graph and the control flow graph for the same class or aspect file. The AODFG provides two different kind of information in one single graph. The advantage of AODFG is using a single graph representation to get information such as the flow of data and the flow control in the program.

Looking further at the subjects, can conclude that aspect-oriented is relatively new for the developers. Not many people know about it, although it was introduced and proposed more than a decade ago. Maybe developers especially programmers are very satisfied and compatible with object-oriented. However, aspect-oriented is still depending on object-oriented as a based technique and this can make software development technology keep growing and research activity in this area are become more interesting.

Aspect-oriented although is not as popular as object-oriented, but AODFG can be one of the alternatives for program analysis. Positive responds from our subjects whom are the twenty experienced software practitioners from different companies almost agree that AOST can effectively help in analysing aspect-oriented programs.

This study was proposed for DFG that works for CFG and DG implemented in aspect-oriented programs based on the AspectJ. But there are many other techniques that can be introduced to work with aspect-oriented programming and future programming trends. Some of them are control dependence graph, program dependence graph, mapping information, symbol table information, local *def-use*, dominator tree and so on.

Author's Contributions

Syarbaini Ahmad: Writing the manuscript, testing and analyze the output.

Abdul Azim A. Ghani: Conceptual study and contents. Critical review of each version, correction and approval.

Ethics

The authors confirm that they abide to all ethical protocols and procedures while preparing this manuscript.

References

Ahmad, S., A.A.A. Ghani and F.M. Sani, 2014. Dependence flow graph for analysis of aspect-oriented programs. Int. J. Software Eng. Applic., 5: 125-144.

Arora, V., R. Bhatia and M. Singh, 2012. Evaluation of flow graph and dependence graphs for program representation. Int. J. Comput. Applic., 56: 18-23.

Bernardi, M.L. and G.A. di Lucca, 2007. An interprocedural aspect control flow graph to support the maintenance of aspect oriented systems. Proceedings of the IEEE International Conference on Software Maintenance, (CSM' 07), pp: 435-444. DOI: 10.1109/ICSM.2007.4362656

Capilla, R., J.C. Duenas and R. Ferenc, 2010. A retrospective view of software maintenance and reengineering research-a selection of papers from European conference on software maintenance and reengineering. J. Software Maintenance Evolut. Res. Pract. DOI: 10.1002/smr.548

Gallagher, K.B. and J.R. Lyle, 1991. Using program slicing in software maintenance. IEEE Trans. Software Eng., 17: 751-761.

Gold, R., 2014. Reductions of control flow graphs. Int. J. Comput. Electrical Automat., Control Inform. Eng., 8: 427-434.

Gold, R., 2015. A uniform approach to control flow graphs of program. Far East J. Applied Math., 93: 27-49.

Jia, L., Y. Jing, W. Ming and J.C. Hong, 2008. Crosscutting invariant and an efficient checking algorithm using program slicing. ACM Sigplan Notices, 43: 12-20. DOI: 10.1145/1361213.1361215

Lallchandani, J.T. and R. Mall, 2009. Static slicing of UML architectural models. J. Object Technol., 8: 159-159. DOI: 10.5381/jot.2009.8.1.a2

Mohanty, S.R., P.K. Behera and D.P. Mohapatra, 2015. Slicing aspect-oriented program hierarchically. Int. J. Comput. Sci. Inform. Technol., 6: 5004-5013.

Ohmann, P. and B. Liblit, 2013. Lightweight control-flow instrumentation and postmortem analysis in support of debugging. Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering, Nov. 11-15, IEEE Xplore Press, Silicon Valley. DOI: 10.1109/ASE.2013.6693096

Parizi, R.M. and A.A.A. Ghani, 2008. AJcFgraph - AspectJ control flow graph builder for aspect-oriented software. Int. J. Comput. Sci., 00: 170-181.

Parizi, R.M., 2008. Control flow structure and graph embodiment of Aspect-Oriented Programs (AOPs): Definitions, algorithm and tool support. PhD Thesis, University Putra Malaysia.

Pigoski, T.M., 1997. Practical Software Maintenance: Best Practices for Managing your Software Investment. 1st Edn., Wiley, New York, ISBN-10: 0471170011, pp: 384.

Pingali, K., M. Beck, R., Johnson, P. Stodghill and M. Moudgill, 2003. Dependence flow graphs: An algebraic approach to program dependencies keshav pingali. Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (PPL' 03) pp: 67-78.

Singh, J., S. Panda, P.M. Khilar and D.P. Mohapatra, 2016. A graph-based dynamic slicing of distributed aspect-oriented software. ACM Sigsoft Software Eng. Notes, 41: 1-8.

Weiser, M., 1979. Program slices: Formal, psychological and practical investigations of an automatic program abstraction method. PhD Thesis, University of Michigan, USA.

Xu, G. and A. Rountev, 2007. Data-flow and control-flow analysis of aspectj software for program slicing. Program, Ohio State University.

Yatapanage, N., K. Winter and S. Zafar, 2010. Slicing behavior tree models for verification of A large systems. Theoretical Comput. Sci., 323: 125-139.

Zhao, J., 2002. Slicing aspect-oriented software. Proceedings 10th International Workshop on Program Comprehension, (WPC' 02), pp: 251-260. DOI: 10.1109/WPC.2002.1021346