International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

AJcFgraph - AspectJ Control Flow Graph Builder for Aspect-Oriented Software

Reza Meimandi Parizi, Abdul Azim Abdul Ghani

Abstract**—**The ever-growing usage of aspect-oriented development methodology in the field of software engineering requires tool support for both research environments and industry. So far, tool support for many activities in aspect-oriented software development has been proposed, to automate and facilitate their development. For instance, the AJaTS provides a transformation system to support aspect-oriented development and refactoring. In particular, it is well established that the abstract interpretation of programs, in any paradigm, pursued in static analysis is best served by a high-level programs representation, such as Control Flow Graph (CFG). This is why such analysis can more easily locate common programmatic idioms for which helpful transformation are already known as well as, association between the input program and intermediate representation can be more closely maintained. However, although the current researches define the good concepts and foundations, to some extent, for control flow analysis of aspect-oriented programs but they do not provide a concrete tool that can solely construct the CFG of these programs. Furthermore, most of these works focus on addressing the other issues regarding Aspect-Oriented Software Development (AOSD) such as testing or data flow analysis rather than CFG itself. Therefore, this study is dedicated to build an aspect-oriented control flow graph construction tool called AJcFgraph Builder. The given tool can be applied in many software engineering tasks in the context of AOSD such as, software testing, software metrics, and so forth.

Keywords**—**Aspect-Oriented Software Development, AspectJ, Control Flow Graph, Data Flow Analysis

I. Introduction

ASPECT-Oriented Programming (AOP) [1] is a relatively new paradigm that can be used to achieve a higher level of modularization, by means of separation of core concerns from crosscutting concerns, in source code than traditional non aspect-oriented approaches. However, AspectJ [2] is a seamless aspect-oriented extension to Java, which is the most widely used aspect-oriented programming language. AspectJ adds some new concepts and associated constructs are called join points, pointcut, advice, inter-type declaration, and aspects into standard Java programming language.

Moreover, support for program analysis is crucial in modern programming languages. Especially, a control flow graph is one of the most basic information of a program to analyze various properties of a program, which in turn would be useful in many ways. However, due to the specific features and language constructs such as join points, advice, introduction, and aspects in Aspect-Oriented (AO) languages, the control-flow analysis for AO programs is more difficult and challenging than that for object-oriented programs, so that the current techniques in procedural or object-oriented paradigm are not able to address these features. Therefore, a good algorithm or technique to derive the control-flow graph for AO programs is valuable and important.

Furthermore, once control flow graphs are obtained they can be applied in many software engineering tasks in the context of AOSD such as software testing, software measurement or metrics, software maintenance, and concern interactions. For instance, It stands to reason that the behavior of an aspect-oriented system is the woven behavior of the aspects and the core; but this woven behavior may reveal conflicts in the goals of the system concerns, core or crosscutting, where such conflicts are called concern interactions (e.g. advice overlaps, where multiple advice applies to the same join point, which can be viewed as potential source of interaction) and may cause ripple effects on the overall AO system, therefore a CFG of the given system would be helpful as map to detect or reveal an easier identification of those point of impacts. Besides, imagine one wants to obtain the CFG of a given AO system manually, therefore a vast of energy probably with less accuracy will be accounted to do so.

On the other hand, although many CFG constructing approaches have been proposed for procedural and object-oriented programming, but there are only a few works in literatures including [3]–[7] are involved with generation of control flow graphs for aspect-oriented programs. These works even though define the concepts and foundations, to some extent, for control flow analysis of aspect-oriented programs but they do not provide a concrete tool that can solely construct the CFG of these programs. Furthermore, most of these works focus on addressing the other issues with respect to these programs such as testing or data flow analysis rather than CFG itself, in which they give a little attention to the detailed information of CFG construction process or its tool support. Therefore, an efficient technique and tool that are appropriate for constructing CFG of aspect-oriented programs are needed. This work presents an algorithm for constructing the control flow graph of AOPs and its tool support. The proposed approach and its given tool can be used to provide

R.M.Parizi is with the Department of Information System, University Putra Malaysia, Serdang, 43400, Selangor, Malaysia (phone: +60-173579565; fax: +60-389466576; e-mail: parizi@fsktm.upm.edu.my).

A.A.A. Ghani is with the Department of Information System, University Putra Malaysia, Serdang, 43400, Selangor, Malaysia (phone: +60-389466555; e-mail: azim@fsktm.upm.edu.my).

useful support to solve the aforementioned issues and other problems for many software engineering tasks in the case of aspect-oriented programming.

The rest of the paper is organized as follows: Section II discusses related work; Section III presents the tool architecture; Section IV gives the explanation of the code compilation unit; Section V describes the data structures used in the given tool; Section VI presents the proposed approach for analyzing the CFG of aspects as well as, the algorithm that takes care of constructing the control flow graph of aspect-oriented programs named BuildAOCFG; Section VII discusses the feasibility and applicability of the builder; Section VIII reports the conclusion and future work on this work.

II. Related work

Several and different forms of control flow graphs have been proposed along the years for procedural or object-oriented paradigm [8]–[11] to address the change in advance of programming technologies. However, as the adoption of AOP in software development is gaining ground [3], some code representation forms for AOP have been proposed in recent years researches based on CFG, which in turn provide information for analyzing AspectJ or similar AOP language programs. Moreover, most of these works are focused on data flow graphs and testing criteria for aspect-oriented programming including [5],[7],[12], of course generation of a dataflow graph usually requires a control flow graph in advance but they do not provide the detailed information of CFG constructing process especially its tool support.

A preliminary contribution is from J.Zhao. J.Zhao in [4] proposes a technique to construct control flow representations for aspect-oriented programs. Although his work defines good foundations, our work is inspired by it to some extent, for constructing CFG of aspect-oriented programs but it does not provide a tool which is capable of putting those theories and definitions into practice. In addition, the proposed technique cannot handle the around advice in the process of constructing control flow graphs, where this issue is taken into account in our work.

O.A.L.Lemos, A. M.R. Vincenzi,J.C. Maldonado and P. C. Masiero in [5] propose the derivation of a control and data flow model for aspect-oriented programs based upon the static analysis of the object code (the Java bytecode) than source code level which resulted from the compilation process. Using this model, called aspect-oriented def-use graph (AODU), traditional and also aspect-based testing criteria were defined. Furthermore, a prototype tool called JaBUTi/ AJ was proposed to support proposed criteria and given model. One of the issue associated with this approach, since it works on bytecode than source code, is difficulty in building and preserving a map that relates the data flow effects of each entity in the bytecode back to its corresponding entity in source code. Furthermore, the obtained map is also subject to change form one complier to another since different compiler can produce totally dissimilar mapping. Therefore, such issues can be addressed though a source code level approach like [6].

More recently, G. Xu and A. Rountev in [6] propose a source-code-level framework called AJANA (**A**spect**J** **ANA**lysis) for interprocedural dataflow analysis of AspectJ programs. The results associated with the evaluation of the framework shows that the given approach is superior to an approach based on the woven bytecode like in [5] which in turn enables analyses to be both faster and more precise as well as, being a promising candidate for systematic foundations of dataflow analysis in AspectJ programs. The proposed framework was implemented as extension to the `abc` AspectJ compiler.

Besides, AJDT (AspectJ Development Tools) [13] on Eclipse provides some information on aspect-oriented programs. With this, although it is possible to construct a simple CFG builder for AspectJ programs but it is not a straightforward process and still needs codification and integration of some other tools such as XRef, Visualizer, or Build path configuration in order to obtain those required information for constructing CFG of AspectJ programs.

Although these approaches provide good foundation for control flow analysis of AOPs and may able to construct the CFG of aspect-oriented programs as a built-in process, but they do not present a concrete tool that can do this, which consequently make them to be not applied directly to construct a complete CFG of aspects due to either remaining in theory than practical or aforementioned issues associated with them. The work presented in this paper is a source-code-level approach which can handle the problems of constructing the CFG of AspectJ software that are unique to aspect-oriented programs, by proposing a tool call *AJcFgraph*, with a high-level program analysis and program understanding.

III. Tool Architecture

The tool, *AJcFgraph Builder*, is developed to automate the construction of the AOCFG (section VI). The following, Fig. 1, outlines the high-level architecture of a system for constructing and managing the AOCFG representations in the given tool. In other words, it illustrates the conceptual view of *AJcFgraph Builder*'s process flow and interaction among its components.

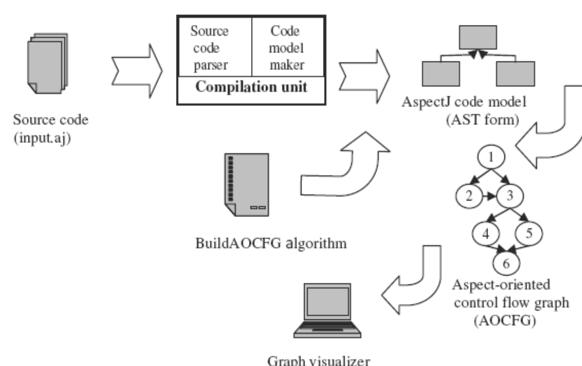

Fig. 1 AJcFgraph Builder architecture model

The given tool is object-orientedly designed and implemented in Java programming language. The main components of the tool, as shown in Fig. 1, are as follows:

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

- *Parser*, which is used to parse AOP code written in AspectJ on the base of modification of *abc* (AspectBench Compiler).
- *Algorithm*, which builds the MCFG, pAOCFG, and AOCFG based on AST of each compilation unit obtained in parsing process.
- *Graph visualizer*, which is used to portrait the obtained AOCFG graph on the screen.

In the following sections each of above component and their basic properties of implementation is described in more detail. Where each is studied in terms of input used, mechanism, library, or framework applied, and output that it produces in the flow of process.

IV. Code Compilation Unit

Generally, this unit is used to analyze the input program to get control flow information such as the predecessors and successors of each statement, caller/called information and finally identification of each module in aspects or classes, which are necessary for constructing the control flow graphs.

The compilation unit has two main components: the source code parser and code model maker. The unit uses the parser to parse input source code written in AspectJ and produces a parse tree, then uses the code model maker to build the code model for the input source from the parse tree produced by parser. Simply, it transforms the given parse tree into an Abstract Syntax Tree (AST).

A. Source Code Parser

The parsing itself is a process with two levels of grammar, as shown in Fig. 2, which are lexical and syntactic. In other words, a context-free grammar [17] is used to define the syntax of AspectJ language by using two sets of rules: lexical rules and syntactic rules.

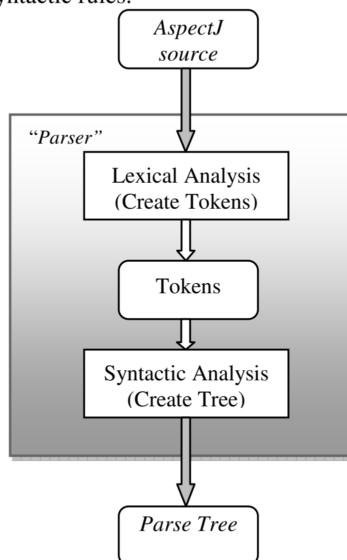

Fig. 2 Process of parsing AspectJ source code

In *AJcFgraph Builder* tool, the language parsing in given parser is implemented with the help and modification of abc[1] (AspectBench Compiler) which is an extensible of ajc (AspectJ Compiler). To the best of our knowledge, at this moment no parser generator can support the grammar of AspectJ as its input; because the conventional parser generators such as CUP, YACC, etc. only accept a restricted class of context-free grammars. However, the AspectJ grammar has already been defined and determined, which can be found in [17].

1) Lexical Structure: The lexical analysis of AspectJ is complicated by the fact that there are really three different languages being parsed: (1) normal Java code, (2) aspect declarations, and (3) pointcut definitions [17]. Therefore, to this end, each of these three sub-languages should have its own lexical structure and implementation, as addressed in *AJcFgraph* Builder.

2) Syntactic Structure: Once the grammars and tokens (using lexer) are determined, the implementation of the syntactic part of the given parser would be straightforward. To this end, the PPG [18] framework is used, in this work, to handle the LALR (1) grammars of AspectJ language as well as, tokens produced by lexical component. Besides, PPG allows changes to an existing grammar to be entered in separate file, overriding, inheriting and extending production from the base grammar. This result in modular extensions, which can easily be maintained should the base grammar change.

B. Code Model Maker

The given component simply traverses the parse tree and constructs code model objects from the parse tree, which is used as program's sole intermediate representation that seeds the algorithm. Then it performs all the static checks required by a number of passes which rewrite the tree. Moreover, semantic checks are implemented by considering appropriate methods on the relevant AST node, where every AST node implements a typeCheck (TypeChecker) method. The type checker is run after all variable references are resolved; all checks that do not require further data structures are typically put in the *typeCheck* method.

The Fig. 3 illustrates the overall view of code mode maker, where it gets the parse tree as input and produces the AspectJ model in the form of AST.

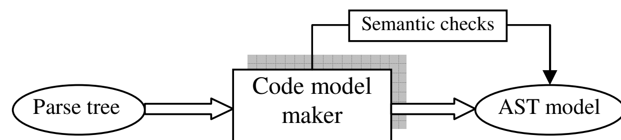

Fig. 3 Code mode maker macro view

The code model maker is actually a set of classes that are capable of simulating the AST of given parser specification

[1] The given compiler is based on Polyglot extensible compiler framework [14] and the Soot byte code analysis and transformation framework [15], which forms the frontend and backend of *abc* respectively.

(based on parse tree) [19]. The AST itself only contains AspectJ and Java constructs where the specific-language details are removed. Furthermore, the elements of the tree are implemented as instances of classes. Instance variables of the AST classes are used to represent the children of the node. Instance variables of leaf nodes are used to hold information about the node's value, e.g. literal values, references into a symbol table, etc. All of the AST node classes are directly or indirectly derived from an abstract base class. A common super/ base class of nodes can be as follows:

```
public abstract class Node implements
Switchable, Cloneable {
   public abstract Object clone();
   void parent() {…}
   void parent(Node parent) {…}
   abstract void removeChild(Node child);
   abstract void replaceChild(Node oldchild,
  Node newchild);
   public void replaceBy(Node node) {…}
   protected String toString(Node node){…}
   protected String toString(List list){…}
   protected Node cloneNode(Node node){…}
   protected List cloneList(List list){…}
   …}
```

In this tool two alternatives in designing AST nodes are used, "smart" objects and "dumb" objects. If there is only one client for the AST, the smart object approach is used. Where, the nodes of the AST provide a method for performing the work the client requires. If there are multiple clients, then dumb objects should be used. With dumb objects, there are few or no methods implemented by the AST nodes. Furthermore, to perform work with the ASTs, the `VISITOR` pattern is used. The Polyglot's visitor-based architecture makes implementing much easier. In this case, two new passes are added. The first stores all global pointcuts in a static variable, and the second applies that pointcut to the relevant code. For reason of code brevity, in given tool implementation, these two passes are implemented by the same class, named `GlobalAspect`; it uses a member variable called pass to distinguish which of the two functions it is performing. On the other hand, to instantiate the `VISITOR` pattern for the implement AST pattern, a `visit` method is defined in each AST. The `visit` method has a parameter for the Visitor object. The Visitor object might do type-checking, code-generation, etc as well (note, Visitor pattern is used to implement traversals when using dumb objects). Therefore, class definitions for the abstract syntax tree are built from the parser specification. The following is an example for the AST class discussed in the above:

```
abstract class AST {
 }
  public class Variable extends AST {
  public Identifier id;
  public Variable(Identifier id) {
    this.id = id;
    }
  }
  public class Constant extends AST {/* .. */}
 public class OrExp extends AST { /* .. */} }
```

V. DATA STRUCTURE

This section briefly describes the data structure used in *AJcFgraph Builder*. The data structures are what are common across all analysis modules in the given tool. One of the key techniques devised in our tool, in order to support AOP features and ease of the analysis development, is the presented data structure. What the data structures represent is explained in the following sub-sections.

A. Graphs, Nodes, Edges

Graphs represent program elements and the relationship between these elements. The program elements are represented as nodes, and the relationships are represented by edges. Different nodes can represent the various program elements in a Java application, and the different edges are used to represent the different relationships between the program elements.

All edges are directed and therefore have a source and destination. Adding new relationships involves adding a new class that *extends* the abstract `Edge` class. All edges can have their own properties. This is especially important for the visual representation of the graphs that can be generated. For example, in control flow graph, a *call edge* is a dashed arrow or an *intra-module control flow edge* is only arrow.

Nodes are more complicated. A node holds an object which represents a program element. This object can either be a Soot object [15], or a `NodeElement` object. Soot objects are only used by `GraphMaker` (section VI.D) while constructing a graph, so once a graph is made, all reference to Soot objects are replaced with references to `NodeElement` objects. This ensures serializability and also that there are no references to Soot from anywhere in *AJcFgraph Builder* once a graph is made. The `NodeElement` is an abstract class which contains two abstract methods, `getSignature` and `setSignature`. Each concrete class must implement these methods. This signature is the actual representation of some program element. The format of these signatures is the same as those used in Soot [16]. The reason the signatures are not stored directly into the nodes is because these signatures represent the most elementary program element in Java, whereas there are some NodeElements that are associated with more than one Node. For example, a class can be abstract, or it can be an interface, or it can be a regular class. Each of these is represented by a different node; however, all these different nodes contain an object of type Class, which is a `NodeElement`, and their signature formats will be the same.

B. Groups

Groups represent an ordered set of nodes from a graph. The order is based on the order in which nodes are added to the group. There are two types of Group, `JGroups` and `Vectored-Statistics`.

There are two main differences between JGroups and

VectoredStatistics. The first is that JGroups can only contain nodes, whereas VectoredStatistics can contain both nodes and JGroups. The second is that JGroups are supposed to represent sets of nodes generated by a graph traversal performed by the `GraphTraverser` (section VI.E) component. VectoredStatistics represent sets gnereated by the `GraphAnalyzer` (section VI.F) component.

C. *Marks*

Marks represent objects that can be attached to Node, Edge, or Graph. Mark can hold any type of object. Marks are used to attach information to other elements. This information can be weight, distance, a string representing color, or any other attribute that the other elements may require.

VI. The Proposed Approach and BuildAOCFG Algorithm

The control flow graph used in procedural systems, non-AOP languages, has been extented to be applicable for object-oriented system as ECFG [20]. ECFG is a collection of CFGs in a layered manner where nodes refer to module, e.g. methods, rather than statements (in contrast with procedural). Our approach, in the case of aspect-oriented, shares the same viewpoint with OO approach in the sense that a control flow graph for AOP is also a collection of CFGs in a hierarchical manner but further since an AO program (in contrast with OO program that is only base code e.g. classes) is divided into two parts: *base code* which normaly 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; the interaction between these two parts as well as unique features of AOP must be considered. To this end, three layers or hierarchies towards coming up with the final control flow graph of AOP are defined; where each layer denotes a special named CFG that captures some part of an AO code as well as, utilizing the lower layers as part of its construction process.

The Fig. 4 depicts the macro view of three layers and their name with respect to proposed approach, where lower layers are used in the upper layers in order to form a hierarchical structure of CFGs, which results in the final control flow graph called *Aspect-Oriented Control Flow Graph (AOCFG).*

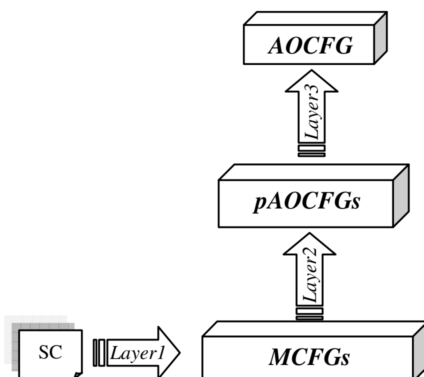

Fig. 4 The macro view of control flow graph construction process

Besides, a modified version of an AspectJ program taken from [21] is used as small AOP named *ShadowTraker*, throughout this section in order to introduce example with respect to approach. Fig. 5 gives the AspectJ code, which consists of two classes (`Point` and `Shadow`) and one aspect (`PointShadowProtocol`) that associates shadow points with every point object. For brevity, some implementation details such as import packages are omitted from the figure.

```
1   public class Point {

2      protected int x, y;
3      public Point(int _x, int _y) {
4         x = _x;
5         y = _y;
6        }
7      public int getX() {
8           return x;
9       }
10     public int getY() {
11          return y;
12        }
13     public void setX(int _x) {
14        x = _x;
15        }
16     public void setY(int _y) {
17        y = _y;
18        }
19     public void printPositionP() {
20        System.out.println("Point at( "+x+"
      ,"+y+")");
21        }
       // main method
22     public static void main(String[]  args) {
23        Point p = new Point(1,1);
24        p.setX(2);
25        p.setY(2);
26        }
27   }

28   class Shadow {

29     public static final int offset = 10;
30     public int x, y;

31     Shadow(int x, int y) {
32        this.x = x;
33        this.y = y;}
34     public void printPositionS() {
35        System.out.println("Shadow at ("+x+",
      "+y+")");
36       }
37   }

38   aspect PointShadowProtocol {

39     private int shadowCount = 0;
40     public static int getShadowCount() {
41       return PointShadowProtocol. aspectOf().
      shadowCount;
42        }
        // introduction of a shadow field into class
        Point
43     private Shadow Point.shadow;
       // its own methods
44     public static void associate(Point  p, Shadow
          s){
45          p.shadow = s;
46        }
47     public static Shadow getShadow(Point  p) {
48        return p.shadow;
49        }

       // pointcuts definitions
50      pointcut  setting(int x, int y, Point p):
          target(p)&& args(x,y)&&
          call(Point.new(int,int));
```

```
51       pointcut settingX(Point p):  target(p) &&
            call(void Point.setX(int));
52       pointcut settingY(Point p):  target(p) &&
          call(void  Point.setY(int));
         // advice body
53    after (int x, int y, Point p)   returning
            :setting(x, y, p) {
54       Shadow s = new Shadow(x,y);
55       associate (p,s);
56       shadowCount++;
57          }
58    after (Point p): settingX(p) {
59       Shadow s = new getShadow(p);
60       s.x = p.getX() + Shadow.offset;
61       p.printPositionP();
62       s.printPositionS();
63          }
64    after (Point p): settingY(p) {
65       Shadow s = getShadow(p);
66       s.y = p.getY() + Shadow.offset;
67       p.printPositionP();
68       s.printPositionS();
69          }
70 }
```

Fig. 5 ShadowTracker aspect-oriented program (AspectJ)

A. MCFG, pAOCFG, AOCFG Definitions

This section provides the definitions of the different layers (or CFGs) used in the proposed approach, Fig. 4, where detailed information regarding these layers and their formal definitions can be found in [22].

- *MCFG*: The first layer or hierarchy in constructing the final control flow graph is capturing the control flow graphs of individual modules (any block of code either in aspects or classes such as methods, constructors, advice, and so forth) that are the building blocks of classes and aspects, which are called *Module Control Flow Graphs*. Given module *m*, MCFG is defined as a directed graph, which is represented by a quadruple: $MCFG_m = (s_m, e_m, V_m, E_m)$, where s_m is the start node of the module that represents the unique entry node into *m* and e_m is the exit node representing the termination of MCFG of a module. V_m is the set of nodes or vertices in the *m,* where each represents one type of statement that can be characterized as three types, which is

$$V_m = V_s \cup V_{ca} \cup V_{jp} \,. \tag{1}$$

Statement nodes (V_s), representing normal code statements (common with OO), *Call nodes* (V_{ca}), representing a statement in the code that contains a call to a method or creating an object, and *Join point shadow nodes* (V_{jp}), representing the join points in the code where the flow of control can be delegated towards an advice. Note that, in order to handle around advice for a given join point, our approach differentiates join point node-before (*JPb*), and join point node-after (*JPa*). That is, when a join point is regarded to an around advice, in the MCFG two nodes, *JPb* and *JPa,* will be considered just before and after the given join point affected by the around advice (in implementing of the graph for simplicity, the line's number that is contacting the given join point is considered as *JP*, for instance, if the desired join point is a call to a method in line 23, of AspectJ example, and an around advice is associated with this join point, therefore two nodes, *23b* and *23a* will be inserted to the given MCFG exactly before and after node 23, to represent the impact of around advice). E_m is a set of directed edges to represent the flow of control between two nodes. More specifically, the MCFG nodes are linked by *intra-module control flow* edges in order to show the transfer of control between code statements.

- *pAOCFG*: The second layer or hierarchy in constructing the complete CFG of a given AO system is, capturing the control flow graph of an aspect, which is called *partial Aspect-Oriented Control Flow Graph*. The word "partial" is used because the given graph does not take into account the relationship between base code and aspect code, whereas it only represents the control flow relationships among the components (advice, pointcut, inter-type declaration, and method) within an aspect. Given *as* an aspect consists of *n* modules such that $MCFG_k = (s_k, e_k, V_k, E_k)$ for $k=1,2,\ldots.n$ are the control flow graphs of *n* modules . The partial Aspect-Oriented Control Flow Graph is defined as a directed graph that is a collection of MCFGk ($k = 1 \ldots n$) and edges linking them to make up the pAOCFG of the considered system. A quadruple: $pAOCFG_{as} = (s^{as}, e^{as}, V^{as}, E^{as})$ is used to denote the pAOCFG, where s^{as} is the aspect start node (or entry node) of *as* and e^{as} is the exit node of the given aspect; similar to MCFG these nodes are unique and have in-degree and out-degree of zero respectively. V^{as} is constructed by the set of all nodes in *n* modules which form the given aspect as well as, the pointcut nodes, which is

$$V^{as} = \bigcup_{k=1}^{n} V_k \cup V_{ptc} \,. \tag{2}$$

Pointcut nodes (V_{ptc}), representing a set of pointcuts in the aspect *as*. It means each pointcut is represented by a pointcut node, *ptc*, that is used to denote the entry to the given pointcut as well. Lastly, E^{as} is the set of all edges in *n* modules forming *as* aspect in addition some new types of edges that capture the interaction among the modules inside the aspect *as*, which is

$$E^{as} = \bigcup_{k=1}^{n} E_k \cup E_m^{as} \cup E_{ca}^{as} \cup E_{ptc}^{as} \,. \tag{3}$$

The E_m^{as} is the set of *aspect-membership edges*, which denotes the membership relationships between aspect *as* and its components. The E_{ca}^{as} is a set of *call* and *return edges*. These edges represent the caller/called relationships (or control flow) between two modules in *as.* The E_{ptc}^{as} represents a set of *pointcut edges,* which are used to link the *ptc* nodes to the start node of corresponding advice modules in *as* to represent the relationship, or control flow, between them.

- *AOCFG*: The third layer is the last hierarchy of the constructing process that results in the complete CFG of AOP, which is called *Aspect-Oriented Control Flow Graph.* The only difference between this and the pAOCFG is in consideration of the interaction between aspects with classes. Generally, in AspectJ, an aspect can interact with a class in several ways, i.e., by *object creation*, *method call*, and *advice weaving* [4]. Therefore, the aspect-oriented control flow graph of an AOP should be able to represent these interactions

between aspects and classes as well. Given *AO* an aspect-oriented program (like that one in Fig. 5) with *r* modules in base code or classes (that are affected by aspects) and *t* aspects so that $MCFG_k = (s_k , e_k , V_k , E_k)$ for $k=1,2,\ldots.r$ are the control flow graphs of *r* modules and likewise, $pAOCFG_p = (s^p, e^p, V^p, E^p)$ for $p=1,2,\ldots.t$ are the control flow graphs of *t* aspects within *AO*. Therefore, the Aspect-Oriented Control Flow Graph is defined as a directed graph that is a collection of $MCFG_k$s and $pAOCFG_p$s, as well as edges linking them to make up the AOCFG of the whole AO system. On the other hand, since pAOCFGs are constructed by a set of MCFGs within themselves, therefore it can be viewed that the $AOCFG_{AO}$ is a collection of all MCFGs no matter from base code or aspect code, where each represents a method (a method of a class, aspect, or `main ()` method), a piece of advice, or an inter-type member. A quadruple: $AOCFG_{AO}= (s^{AO}, e^{AO},V^{AO}, E^{AO})$ is used to represent the given graph, where s^{AO} denotes the start node or the point from which the execution of the given program is started and e^{AO} that represents the exit node of the aspect-oriented program, in which the execution of *AO* will be terminated; similar to MCFG and pAOCFG these nodes are unique and have in-degree and out-degree of zero respectively. Roughly speaking, the s^{AO} is the same with the start node of the MCFG of `main()` method in base code, because in AOCFG (in contrast with MCFG or pAOCFG), the dynamic or running view of whole aspect-oriented program is traced. V^{AO} is a set of nodes in *AO* that is defined as

$$V^{AO} = \bigcup_{k=1}^{r} V_k \cup \bigcup_{p=1}^{t} V^p . \qquad (4)$$

Where, the first element is the set of all nodes in *r* modules in *AO* aspect-oriented program; each represents a call node, statement node, or joint point shadow node. Likewise, the second element is the set of all nodes in *t* aspects where each represents a pointcut node, call node, or statement node of modules in aspects. Therefore, in the case of AOCFG, there is no new or additional type of node that is going to be added; whereas it only synthesizes the nodes obtained from the first and second layers, that is

$$E^{AO} = \bigcup_{k=1}^{r} E_k \cup \bigcup_{p=1}^{t} E^p \cup E_{ca}^{AO} \cup E_{w}^{AO} . \qquad (5)$$

The first two elements denote the set of all edges in the corresponding *r* modules and *t* aspects within *AO* aspect-oriented program (already obtained from the first and second layers). The E_{ca}^{AO} captures the set of the *inter-module control flow* or *call/return* edges in interaction of aspects with classes while there is call or object creation. More specifically, in the case of *method calls*, a call may occur between two modules m_1 and m_2 either in aspect or base code. In this case, an inter-module control flow edge is used to link the call node of m_1's MCFG to the start node, s_2, of m_2's MCFG and in return an inter-module control flow edge from the exit node, e_2, of m_2's MCFG towards the immediate statement of call node in m_1 is drawn as well. Besides, in the sense of *object creation*, a module *m* is capable of creating an object of a given class *C* either in an aspect or a class. The creation process can be done via class type declaration or by using an operator such as `new`. In this case, by definition, behind the scene an implicit call will be done from module *m* (at the place of object creation) to class *C*'s constructor. Therefore, to take into account this implicit constructor call and represent it in AOCFG, an *inter-module control flow* edge is also used to link the call node of module *m* to the start node, *s*, of the MCFG of *C*'s constructor as well as, one edge for returning the flow of control from the *C*'s constructor to the module *m* is drawn as well. Lastly, E_{w}^{AO} denotes the set of *aspect-weaving edges* that captures the advice weaving process in the *AO* aspect-oriented program. Generally, as mentioned earlier, in AOP new behaviors or functionalities defined by advice are woven into the join points in the base code. Therefore to represent such relationships (or control flow) between an advised module *m* (normally a class's method or constructor) and advice *a* (after, before, or around), *aspect-weaving* edges are used to link the join point shadow nodes within *m* to the start nodes, *s*, of the $MCFG_a$ corresponding to advice.

B. Example of AOCFG Construction

In this section the AOCFG of complete aspect-oriented program, *ShadowTracker* (Fig. 5) that is referred as *A*, is analyzed and also shown in the Fig. 7. The *ShadowTraker* program is composed of 9 modules in the side of base code (in both `Point` and `Shadow` classes) which are: `Point`'s constructor, `Shadow`'s constructor, `getX()`, `getY()`, `setX()`, `setY()`, `printPositionP()`, `printPositionS()`, and `main()` method as well as, one `PointShadowProtocol` aspect that is consisting of 6 modules. At this moment, assuming the MCFGs of 15 (all together) modules are done and are available. Therefore, based on the aforementioned definitions, the control flow analysis of the *ShadowTraker* program can be presented as follows:

$AOCFG_A = (s^A, e^A,V^A, E^A)$, where $s^A = 22$ and $e^A =26$. Next

by referring to (4), we have: $V^A = \bigcup_{k=1}^{9} V_k \cup \bigcup_{p=1}^{1} V^p$

The first element in V^A is a set of all nodes in nine modules in *A* program; each represents a call node, statement node, or joint point shadow node. The second element is a set of all nodes in `PointShadowProtocol` aspect where each represents a pointcut node, call node, or statement node of six modules in given aspect that have been reused. Lastly by referring to (5), we have

$$E^A = \bigcup_{k=1}^{9} E_k \cup \bigcup_{p=1}^{1} E^p \cup \{(23,3),\ (6,23a),\ (24,13),\ (15,24a),\ (25,16),\ (18,25a),\ (54,\ 31),\ (33,55),\ (57,24),\ (60,7),\ (9,61),\ (61,19),\ (21,62),\ (62,34),\ (36,63),\ (63,25),\ (66,10),\ (12,67),\ (67,19),\ (21,68),\ (68,34),\ (36,69),\ (69,26)\} \cup \{(23a,\ 53),(24a,\ 58),(25a,64)\}.$$

International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:9, 2008

The first two elements in E^A denote the set of all reused edges in the corresponding nine modules and `PointShadowProtocol` aspect within *A* program. The third set is consisting of the *inter-module control flow edges* from each there is either a call/object creation to a given method/constructor as well as, corresponding return edges (E^A_{ca}). For instance, (54, 31) represents a `Shadow` object creation on line 54 that causes a call to the given class's constructor placed on the line 31. In other words, in the AOCFG a directed edge from call node within the *after* advice module (line 54) is drawn towards the start node of the `Shadow`'s constructor MCFG (line 31). Lastly, the fourth set denotes the set of *aspect-weaving edges* (E^A_w), for example, (23a, 53) represents that there is a join point shadow node named 23a in `main ()` method's MCFG that is associated with `after ()` advice on line 53 in which `setting ()` pointcut captures any call to the `Point` class's constructor. More specifically, `Point` object creation on line 23 causes a call to the given class constructor that in turn triggers the `setting ()` pointcut, since the given pointcut is associated with the after advice (line 53) therefore the body of advice will be executed. In a nutshell, such flowing of the control between `Point`'s constructor and advice is represented by (23a, 53) edge.

C. The BuildAOCFG Algorithm

Finally, the basic algorithm that can construct the AOCFG for an aspect-oriented program is given below. Fig. 6 shows the pseudo-code of the algorithm, `BuildAOCFG`.

algorithm `BuildAOCFG` */* generate a control flow graph from the given AO program */*

input An aspect-oriented program written in AspectJ: $\mathcal{A}$
output The aspect-oriented control flow graph of $\mathcal{A}$: AOCFG$_{\mathcal{A}}$

begin // *BuildAOCFG*
1. */* preprocess (parse) the aspect-oriented program $\mathcal{A}$ */*
2. **for** each class C
3. Identify the methods
4. **endfor**
5. **for** each aspect A
6. Identify the advice, introduction, pointcut, and method
7. **endfor**

8. */* build MCFGs for methods in each class and advice, introduction, and methods in each aspect*/*
9. **for** each class C
10. GraphMaker(m) //Construct MCFG for each method *m* in C
11. **endfor**
12.
13. **for** each aspect A
14. **if** A is concrete then
15. **for** each advice, introduction, or method *m* declared in A
16. GraphMaker(m) // Construct MCFG for *m*
17. **endfor**
18. **else**
GraphTraverser(A)
19. **for** each advice, introduction, or method *m* in the extended aspects
20. **if** *m* is marked as "inherited" then
21. Reuse module control flow graph (MCFG) from the base aspect
22. Adjust call sites
23. **else**
24. GraphMaker(m) // Construct the given MCFG for *m*
25. **endif**
26. **endfor**
27. **endif**
28. **endfor**
29. */*connecting MCFGs at call sites in order to build pAOCFG */*
30. GraphCombiner()
31. */* weaving MCFGs at join point sites in order to build complete AOCFG*/*
32. GraphWeaver()
33. */* return the complete AOCFG of $\mathcal{A}$ */*
34. return AOCFG$_{\mathcal{A}}$

end // *BuildAOCFG*

Fig. 6 Basic algorithm for AOCFG construction

According to algorithm, as input `BuildAOCFG` gets the source code of the given aspect-oriented program, $\mathcal{A}$, and as output BuildAOCFG returns the $\mathcal{A}$'s control flow graph, which is AOCFG$_{\mathcal{A}}$. First, BuildAOCFG preprocesses each aspect and class in $\mathcal{A}$ to get those kinds of information that are necessary for constructing the AOCFG, in this step it identifies advice, introduction, pointcut, and methods (line 1-7). Second, algorithm builds the MCFG for each piece of advice, introduction, and method in aspect or class, using `GraphMaker()` module. Note that, it builds these MCFGs in a bottom-up fashion according to the aspect and class hierarchies (line 8-28) therefore, to traverse the graph of a given class or aspect a `GraphTraverser()` module is devised. After that, BuildAOCFG utilizes `GraphCombiner()` module to connect these MCFGs at the call sites to form the pAOCFG (line 30). Finally, BuildAOCFG builds the complete AOCFG for $\mathcal{A}$, AOCFG$_{\mathcal{A}}$, by calling `GraphWeaver()` module to weave the MCFG at the join points for each piece of advice into the MCFGs for its corresponding methods in the pAOCFG (line 32-34).

The algorithm that is implemented has a class "Controller", which is the brain of *AJcFgraph Builder* tool. It takes in options, and generates constraints (type parameter objects and value parameter objects are collectively known as constraints) for the analysis modules and makes the appropriate calls. The first step in this process is to parse the input source code, and based on this; generate objects representing the options (constraints). After this, the Controller calls the different modules or component in the order specified by the algorithm towards complete construction of AOCFG, and if there are any objects for the modules, it passes them in.

The Controller invokes all other modules in the order specified in algorithm. The following is the module or component invocation order:

1. `GraphMaker`
2. `GraphTraverser`
3. `GraphAnalyzer`
4. `GraphCombiner`
5. `GraphWeaver`
6. `GraphVisualizer`

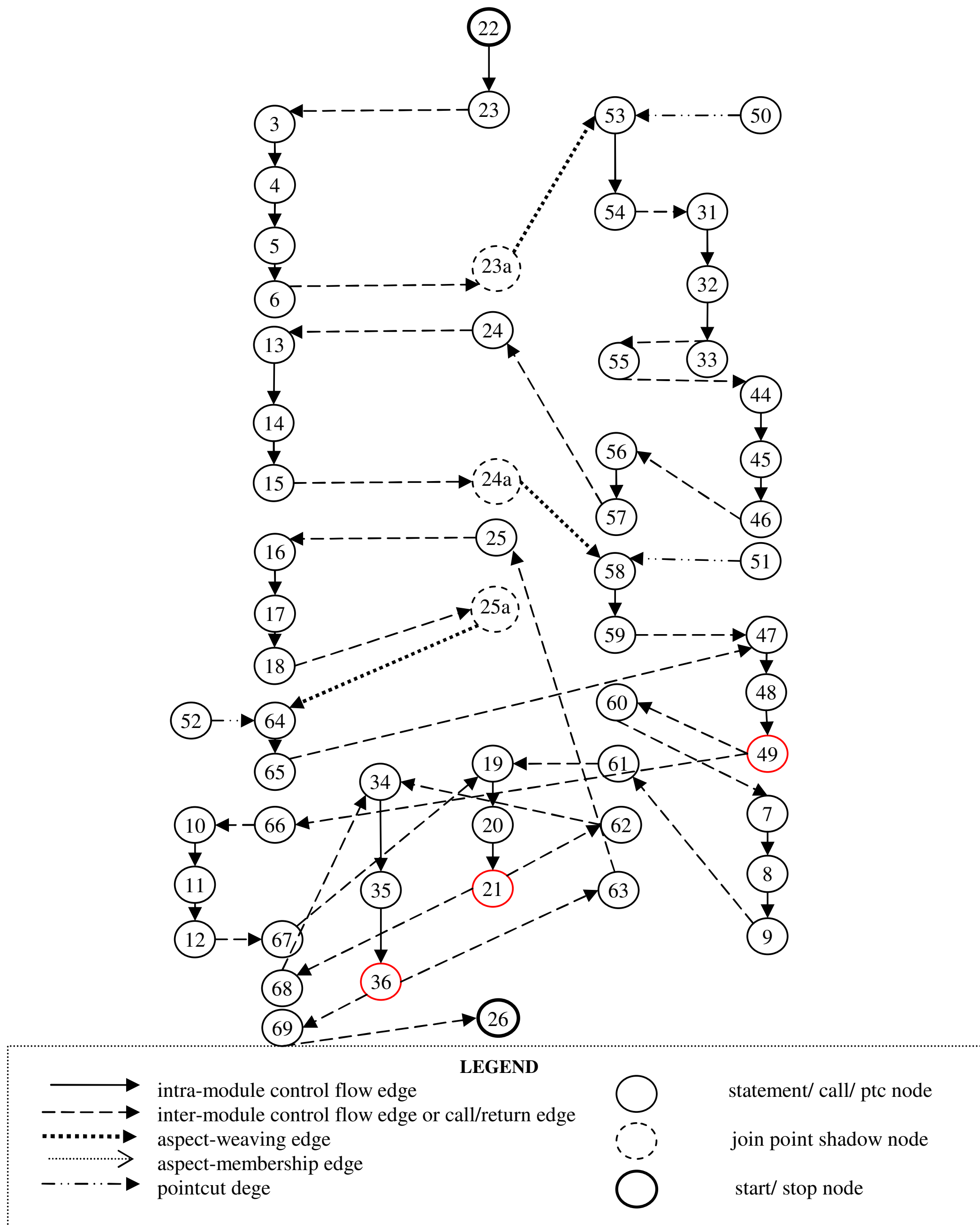

Fig. 7 The complete AOCFG for ShadowTraker aspect-oriented program

The details of the components or modules used in this algorithm and their interactions will be discussed in next sub-sections, whereas each module is responsible for a part of the BuildAOCFG construction process.

D. Graph Maker

`GraphMaker` has two main functions: building a graph from application code, and adding or removing edges or nodes from a graph (i.e. modifying a graph). This is called filtering a graph. To build a graph, GraphMaker uses the Soot Jimple API [15]. For filtering a graph, the user specifies filters to use on a graph. Filters are also value constraints.

1) Building a Graph: Building a graph is divided into two steps:

1. Creating nodes
2. Adding edges between nodes created in step 1

Each graph type first creates a set of nodes based on the type of graph it is, and these nodes are then passed through a set of value constraints for nodes (i.e. these value constraints

specifically apply to nodes). An example of a value constraints is to only pass nodes that have *x* number of parameters.

The next step is to generate edges between the nodes. Each edge, before being added to graph, is passed through a set of value constraints for edges. If an edge fails any of the constraints, it is deleted. It is important to note that, using the node and edge value constraints allows many different alternatives to be run on the same graph type.

2) Filtering a Graph: Filters allow modification of a graph from a holistic point of view. Whereas the constraints only act on a single node or edge at a time, a filter takes in an entire graph and can modify it. The modification can depend on the `Filter` class itself. For example, a filter can remove all nodes with no edges incident to it. These filter objects are the only way to add or remove new edges or nodes form a graph once it has been created. Currently *AJcFgraph Builder* has 4 different filters.

E. Graph Traverser

`GraphTraverser` takes in a graph, and generates a set of JGroup based on the types of traversal specified. The following are the three main steps performed by this component or module:

1. Generate a set of staring points for the traversal
2. Traverse the graph once for each starting point and generate a JGroup for that starting point
3. Filter the generated JGroups

1) Generating Starting Points: By default, every node in the graph is used as a starting point. This can be changed by specifying starting point value constraints. Each starting point is used as a node to begin the traversal, and the set of nodes visited by a traversal is added to a JGroup for the starting point. The starting point is always the first element added to the JGroup being generated.

2) Traversing the Ggraph: The type of traversal specifies how a graph is going to be traversed. This is the type constraint for this component. The starting points specify where the traversal begins. Traversing a graph from a starting point is broken into the following three steps:

1. Follow an edge
2. Visit a node and add the node to the JGroup being constructed
3. Add a subset of the edges incident to the node in step 2 to the list of edges to follow

For each traversal, the starting point node is added to a new JGroup. After this, based on the traversal type, a list of edges incident to the starting point is generated. Each of these edges is passed through the following edge value constraints, and if it fails any constraints, the edge is discarded. After an edge passes all the following edge value constraints, the edge is followed and the node at the other end is visited (because the edge can be followed forward or backward, the new node may be the source or destination of the edge). The new node is then passed through adding node value constraints, and if it passes all constraints, it is added to the JGroup, and the edges incident to it are analyzed. If it fails any of the adding nodes' value constraints, then the node is discarded, and none of the edges incident to it are analyzed.

3) Filtering Groups: Filtering groups is similar to filtering a graph, and represents additional value constraints. This step provides a holistic view of all the groups. This is the only place where groups can be deleted. One important thing to note is that whereas `GraphTraverser` can only create JGroups, this part of the module can be used to delete either JGroups or VectoredStatistics. In this case, all filters must extend the abstract class `FilterGroupConstraints`. Examples of filters include deleting groups that are smaller than a certain size, or merging similar groups into a single group.

F. Graph Analyzer

`GraphAnalyzer` generates VectoredStatistics data structure. VectoredStatistics are generated using the JGroup created by the *GraphTraverser*. The type constraints define what kind of analysis will be run.

Unlike the other modules, the value constraints for this module are specific to the type constraint. For example, this module can perform clustering, and the value constraint for this is the number of clusters. There are no generic value constraints.

G. Graph Combiner

The function of this component is to take in a number of `Graph` objects, and create a new graph based on these. This is the only module that allows graphs from different objects (MCFG or pAOCFG) to be analyzed together and combined into a single graph.

More specifically, using the graphs and groups, the algorithm performs an analysis (done by `GraphAnalyzer`) on the graph and groups and generates a new graph. An example of this is merging all nodes with the same signature (i.e. the signature held in the node element data structure of the given node) from different graphs. This is very useful when different graphs that represent the same code or program are being combined.

Creating a new graph here is very similar to the graph creating steps in `GraphMaker`. The following are the steps in this component:

1. Create all new nodes
2. Create new edges between nodes

Each new node is passed through a set of value constraints, and if it passes them all, then it is added to a new graph. After all nodes are added, new edges are created, and each new edge is passed through the value constraint for those edges. If the edge passes all of these constraints, then it is added to the graph. Both the creation of nodes and the addition of edges will be dependent on the type of analysis being done.

H. Graph Weaver

The algorithm to build the AOCFG is based on the `(static)` analysis of the aspect's pointcut expressions. However, for more complicated join points, it would be necessary to override the code that iterates through an entire method body looking for join point shadows. The overriding code can do

any required analysis of the method body to find instances of the new join points, for instance, one might want to inspect all control flow edges to find the back edges of loops. In particular the algorithm (using `GraphWeaver`) inserts join point shadow nodes into the MCFG and adds directed edges that link the join point shadow nodes, in all modules, to the advice whose pointcuts picks them up. For each advice, the associated pointcut expression AST tree (obtained from the previous component) is built in order to generate the resulting set of join point shadow nodes related to the statements selected by the pointcuts. Thus, the set of join point shadow nodes which are related to the control flow graph region are picked up by the pointcut declaration [3].

During the traversal of the pointcut expressions, shadow nodes and corresponding edges are added to each MCFG. Each join point shadow node is associated with additional information about the pointcut's expressions matching it.

More specifically, based on AspectJ code model (or AST) obtained, passes on tree use data flow information to check initialization of local variables and the existence of *return* statements. Again, each AST node implements methods to build the control flow graph for these purposes. The traversal of the AST is performed by the `ContextVisitor` Polyglot class. The new pass extends `ContextVisitor` with a method that performs the required action when it encounters a relevant AST node. The following code fragment illustrates the behavior of the new visitor upon entering an AST node:

```
public NodeVisitor enter(Node parent, Node n){
        if (pass == COLLECT)&& n instanceof
GlobalPointcutDecl{
            (( GlobalPointcutDecl)n).
                 registerGlobalPointcut(this,
context(), nodeFactory);
         }
         return super.enter(parent,n);
}
```

As mentioned above, both new passes are implemented by the same class, and hence the check that *pass = = COLLECT* makes sure the process is doing the right thing. If the current node is a *GlobalPointuctDecl* (it is an AST node) so it registers itself with the data structure storing global pointcuts. Then it delegates the rest of work (the actual traversal) to the super class.

I. Graph Visualizer

`GraphVisulizer` module is responsible for depicting the information obtained from algorithm component on the screen.

The `GraphVisulizer` takes in a text file in dot format (where the text file represent a graph, *G* (*V,E*) produced by the algorithm) and convert it to a visual representation of a graph. To this end, in *AJcFgraph Builder* implementation, a separate Java graph library called "*JGraph*" is used [23]. JGraph is the leading Open Source Java Graph Visualization Library. It follows Swing design patterns to provide an API familiar to Swing programmers and functionality that provides a range of features.

In addition, there are value constraints for nodes that can determine which nodes get written out and value constraints for edges that determine which edge get written. In the output, the nodes and edges all have different colors and style, where the different color or styles represent different type of nodes or edges.

VII. Discussion

In order to assess the feasibility and correctness of *AJcFgraph Builder* some AspectJ software were analyzed and represented by the AOCFG using the proposed tool. Our study used five AspectJ programs taken from AspectJ example package as shown in Table I. The rationale behind was, this collection of programs has also been used as benchmarks by other researchers including [6], [24], [25]. Moreover, for each program table gives the number of aspects, classes, and modules, where *module* denotes a class constructor, a piece of advice, an inter-type declaration, and a method in either aspects or classes. Note that, since a pointcut does not contain any body code therefore it is not considered as module and consequently does not need any MCFG to represent it.

We verified those AOCFGs generated by the tool against a manual inspection of the graph and the associated analyzed source code for each of aforementioned programs. Our small experiment showed that the AOCFG generated by the tool were correct so that representing AO software by the AOCFG provides a useful support for gaining a better knowledge of the internal structure of these complicated programs, by reducing the effort needed for obtaining them in a variety of software engineering tasks.

TABLE I
ANALYZED PROGRAMS

Program	#Modules	#Classes	#Aspects
`telecom`	53	10	3
`bean`	19	2	1
`observer`	25	6	2
`tjp`	8	1	1
`introduction`	19	1	3

Moreover, some user interfaces screenshots of the tool are presented below, which are captured based on the main functionality of *AJcFgraph Builder* tool.

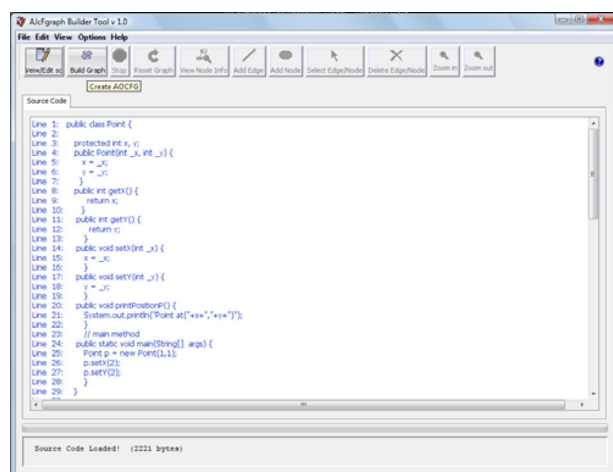

Fig. 8 Source code view (after loading)

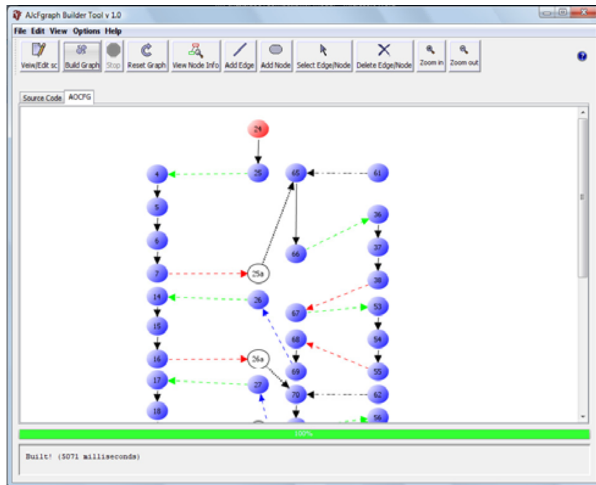

Fig. 9 AOCFG construing result view

VIII. Conclusion and Future Work

In this study the *AJcFgraph Builder* tool, in supporting of the automated construction of AOCFG approach, was proposed. The main features, definitions, and functionalities, as well as its architecture were discussed. The current implementation of tool allows defining, editing, storing and applying AOCFG to an AspectJ system. One of the useful aspects of *AJcFgraph Builder* is its *portability* and being run on multiple platforms. The reason is the Java codes are compiled into intermediate code called byte code. Besides, the *extensibility* is the other aspect of the given tool that allows easily extension of new type constraints or value constraints. It means, to this end, the related class has to be modified, and the new class must extend the appropriate abstract class, and implement the appropriate methods that need to be implemented.

As future research, the *AJcFgraph Builder* tool will be improved and evolved to a more mature and scalable tool in order to be used in more complex systems and multi-language aspect-oriented environment. In addition, future work also includes focusing on the given tool to be extended into a quantitative evaluator tool of AO software in which AO metrics are computed, but as part of it the proposed AOCFG and its tool support will be used as underlying basis to develop or adopt existing structural metrics (of OO or procedural) for aspect-oriented paradigm. Also, cost estimation of building representation and graph minimization still remain as open research questions for further research. Furthermore, we are currently working on the given tool to make it as Web-based tool to be online accessible from any corner of the world.

References

[1] A. Colyer,A.Clement, "Aspect-oriented programing with AspectJ", *IBM Systems Journal* , pp.301-308, 2005.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "An Overview of AspectJ", *In Proceedings 13th European Conference on Object-Oriented Programming*, LNCS, Vol.1241, Springer-Verlag, pp.220-242, 2000.

[3] M. L.Bernardi,G.A.Di Lucca, "An Interprocedural Aspect Control Flow Graph to Support the Maintenace of Aspect Oriented Sysesms", *ICSM'07*, IEEE computer society, pp. 435-444, 2007.

[4] J. Zhao, "Control-Flow Analysis and Representation for Aspect-Oriented Programs", *In Proceedings of the Sixth International Conference on Quality Software (QSIC'06)*, IEEE computer society, 2006.

[5] O. A.L.Lemos, A. M.R. Vincenzi,J.C. Maldonado and P. C. Masiero, "Control and data flow structural testing criteria for aspect-oriented programs", *Journal of Systems and Software* , pp.862-882, 2007.

[6] G. Xu and A. Rountev, "AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software", *AOSD*, ACM, 2008.

[7] J. Zhao,"Data-Flow-Based Unit Testing of Aspect- Oriented Programs", *In Proceedings 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, Dallas, Texas, pp. 188-197, 2003.

[8] M. J. Harrold, D. Liang, "Efficient Points-to Analysis for Whole-Program Analysis", *In Proceedings 7th European Software Engineering Conference*, Springer, Lecture Notes in Computer Science, Toulouse, France, 1999.

[9] R. G. Sargent, D. G. Fritz, "An overview of hierarchical control flow graph models", *In Proceedings 27th Conference on Winter simulation*, ACM Press New York, Arlington, Virginia, United States, 1995.

[10] N. Schwartz," *Steering Clear of Triples: Deriving the Control Flow Graph Directly from the Abstract Syntax Tree in C Programs"*, Technical Report: TR1998-766 , New York, NY, USA, 1998.

[11] M. J.Harrold, Gregg Rothermel, "A Coherent Family of Analyzable Graphical Representations for Object-Oriented Software", Technical Report OSU-CISRC-11/96- TR60, 1996.

[12] J. Zhao, "Tool support for Unit Testing of Aspect-Oriented Software", *OOPSLA Workshop on Tools for Aspect-Oriented Software Development*, Seattle, WA, USA, 2002.

[13] AJDT: AspectJ Development Tools http://www.eclipse.org/ajdt/.

[14] N. Nystrom,M.R.Clarkson, A.C.Myers, "Polyglot: An extensible compiler framework for Java", *In 12th International Conference on Compiler Construction,* Lecture Notes in Computer Science, pp.138-152, 2003.

[15] R. ValleeRai, E.Gagnon, L.J.Hendren, P.Lam, P.Pominville, V.Sundaresan, "Optimizing Java bytecode using the Soot framework:Is it feasible?", *In 9th International Conference on Compiler Constructio,*pp.18-34, 2000.

[16] R. ValleRai, "Soot: A Java Bytecode Optimization Framework", McGill University, School of Computer Science, 2000.

[17] L. Hendren, O.de Moor, A.S.Christensen, the abc team, "The abc sanner and parser, including an LALR(1) grammar for AspectJ", september 2004.

[18] M. Brukman,A.C.Myers, "PPG: a parser generator for extensible grammars", 2003 : Available at www.cs.cornell.edu/projects/polyglot/ppg.html.

[19] J. Jones, "Abstract Syntax Tree Implementation Idioms", University of Alabama, 2004.

[20] A. Kanjilal, G.Kanjilal, S.Bhattacharya, "Extended Control Flow Graph: An Engineering Approach", *CIT'03*, India, pp.22-25, 2003.

[21] The AspectJ Team, "The AspectJ Programming Guide", 2001.

[22] R. M. Parizi, "Control flow structure and graph embodiment of aspect-oriented programs: definitions, algorithm, and tool support", Master Thesis, Dept. of Information System, University Putra Malaysia, unpublished.

[23] JGraph: Java Open Source Graph Drawing Component http://www.jgraph.com/jgraph.html

[24] B. Dufour, C. Goard, L.Hendren, O.de Moor, G.Sittampalam, and C. Verbrugge, " Measuring the dynamic behavior of AspectJ programs", *OOPSLA*, pp.150-169, 2004.

[25] G. Xu, A. Rountev, " Regression test selection for AspectJ software", *ICSE*, pp.65-74, 2007.