www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 6 Issue 2 Feb. 2017, Page No. 20176-20182
Index Copernicus Value (2015): 58.10, DOI: 10.18535/ijecs/v6i2.02

Identification and Removal of Interfernce in Aspectj Programs

G. Barani[1], V. Suganya[2], S. Rajesh[3]
[1,2,3]Assistant Professor, Department of Computer Science and Engineering,
Sri Ramakrishna Institute of Technology, Coimbatore, Tamilnadu, India.

Abstract

The motivation behind introducing Aspect Oriented Programming (AOP) has been to increase the modularity of software by allowing a clear separation of core and cross-cutting concerns in software. AspectJ is a common AO programming technique used by programmers with excellent support from the Eclipse community. In AspectJ, complex interactions between the base code and aspects can make the code very difficult to understand and maintain. Added to this, there is also a possibility for the occurrence of interference between cross-cutting functionalities offered through advices and woven at the join points in AspectJ software. These interferences cannot be identified by the developer without a proper analysis on its existence. In order to address the problems arising out of interferences in AspectJ programs, this paper summarizes the work done to provide capabilities for the definition and identification of the rules of violation. A tool has been developed to define and identify interferences and also to provide possible solutions for the removal of interferences in a given AspectJ code.

Keywords: *Aspect Oriented Programming (AOP), Interference Analysis, Control Flow Interference, Data Flow Interferecne.*

1. Introduction

Aspect Oriented Programming *(AOP)* is a programming paradigm focusing on improving the modularity of software by encapsulating the cross-cutting concerns into independent units of functionalities named as *Aspects*. *AOP* [6] is able to increase the modularity by enabling the clear separation of core and cross-cutting concerns. A cross-cutting concern is a tangled or scattered functional code that can possibly affect other functionalities programmed in software. Persistence, logging, transaction and caching are some of the non-functional cross-cutting concerns easily visible right from the design and implementation of software. The implementation of cross-cutting concerns which are usually found as tangled and scattered code segments, may lead to

reduction in modularity. *AOP* includes programming constructs and tools that support the modularization of cross-cutting concerns. *AspectJ* is an extension of *Java* programming language that provides new constructs such as aspect, pointcut, advice and introduction that enables the software developer in defining cross-cutting code segments as independent units.

Even though *AOP* is a very useful and powerful technique, it introduces new type of risks involving interferences between cross-cutting functionalities. In *AspectJ*, program flow is modified by defining advices for encapsulating cross-cutting code. It is also possible to encapsulate more than one advice inside an aspect. In such cases, interference between functionalities defined in advices can possibly occur. Hence the designer has to define the order in which the advices of an aspect need to be executed. Undesirable interferences may occur when several aspects are woven at the same join point of the base code. For example, one aspect can prevent the execution of another aspect, or can even update a shared variable that the other aspect is reading to view its current state. Since multiple aspects independently encapsulate different cross-cutting concerns, their executions in the base code are usually uncoordinated. The interferences caused due to this design in *AspectJ* programs cannot be manually identified and removed. Hence, an automated testing tool is needed to analyze the existence of interference in a given *AspectJ* code.

This paper introduces two types of interference that can possibly occur in an *AspectJ* program and provides a mechanism to identify and remove the interferences. The types of interferences identified are Data flow interference and Control flow interference. Interference that occurs due to actions affecting the passing of control to the next advice or to the base code is called as control flow interference. Interference that occurs due to read/write access by two or more modules on the shared data is called as data flow interference. Interference will not necessarily stop the execution of the program. But, it can possibly change the intended behavior of aspects during the program execution. Interference analysis and removal will help in removing interference based errors that may occur in a given *AspectJ* code.

Manual identification and removal of data and control flow interferences usually requires adept skill and effort. Hence, an automated tool to identify the existence of interference and suggest alternative removal methods have been developed and applied on a given *AspectJ* code. A *Java* based tool named Aspect Oriented Software Interference Analysis *(AOSIA)* tool has been developed that identifies the existence of interference in the given *AspectJ* code. The tool is hard coded with rules to identify the data and control flow interferences and generates a violation report consisting of types of interferences found in the given *AspectJ* code. This report also contains removal methods for the identified interferences.

Section II expands the work done on the identification of interferences in *AspectJ* code. Section III explains the existing tools available for the analysis of a given *Java* code. Section IV brings out the motivation behind the need for interference analysis. Section VI describes the types of interference within the scope of interference analysis done in this work. Section VII explains the architecture of AOSIA tool that was developed to identify and remove data and control flow interferences. The application of the AOSIA tool to sample *AspectJ* programs is explained in Section VIII. Section IX concludes and provides pointers for future work to be done on interference analysis.

2. RELATED WORK

In *AspectJ* programming, it is possible to define more than one advice inside an aspect. In such a case, interference may occur between the functionalities defined in the advices. Hence, the designers have to necessarily define the order of advice execution. In a work done by Storzer [12], in order to change the runtime behavior of a program, more than one advice was defined in an aspect. The first advice encrypts password obtained from the user and the second advice sends the encrypted password to the server. The order of execution of the two advices defines the final outcome of the program. Hence, this may lead to interference between the functions defined in the advices. Further analysis of advices is a major challenge in order to develop an analytical framework for *AOP* software.

Zhang [14] studied the complicated interactions between the aspects and base code in *AspectJ* programs. The author also proposed a concise classification of impacts based on state and computation changes and caused by advice and inter-type declarations.

Interferences between the aspects were identified using formalization and proof methods by Katz [7]. Modular interference detection methods have been used to identify the possible interferences, i.e., the library of aspects is checked independently of any base system. Consider a situation in which a user would like to weave multiple aspects from the library into the base system. In this case, the only check that should be performed is that the base system satisfies the assumptions of all the aspects. Only after this satisfaction the aspects will be woven to the base system. By using this proof of satisfaction the user has to manually check the interferences and no automated verification procedure have been included by the author.

Lauret [9] avoids undesirable interference by mandatory control of order of execution of conflicting advices. In this work, executable assertions were used to model the code by attaching non-interference requirements to the composition of advices. Avoiding of interference has been done manually. And if more number of advices is present in the software, then the time consumed to identify and remove interferences will be increased. The manual checking can be automated using a tool that can check and analyze the existence of interference in a given *AspectJ* program.

3. AVAILABLE TOOLS FOR SOURCE CODE ANALYSIS

In the literature, a few tools are available for analysis of the static part of a given code. A source code analyzer tool provides mechanism for the automated testing of source code with due purpose of identifying interferences in the given software. The source code is the most permanent form of a program, even though the program may later be modified, improved or upgraded.

Find bugs [9] is an open source program created by Bill Pugh and David Hovemeyer which looks for bugs in a given *Java* code. It uses static analysis to identify hundreds of potentially different types of errors in *Java* programs. Additional rule sets can be plugged in Find bugs to increase the checks performed only for *Java*. This tool cannot be extended to identify interferences in *AspectJ* code.

A tool named *AJATO* [5] provides support to compute *AO* metrics for software implemented in *Java* and *AspectJ* programming languages. The metrics available in *AJATO* are Concern Diffusion over Components, Number of Attributes per Concern, Number of Operations per Concern, Vocabulary Size, Number of Attributes, and Number of Operations. In addition to the assessment of metrics the tool also implements some heuristic rules in order to automate modularity analysis. Currently, *AJATO* does not provide features that can be used to extend and write interference rules for *AspectJ* programs.

PMD [3] is a static rule set based *Java* source code analyzer that verifies for the existence of interferences using pre-defined set of rules. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. *PMD* errors are not true errors, but rather inefficient code, i.e. the application could still function properly even if they were not corrected. But, the same *PMD* tool has not been extended to define interference rules for *AspectJ* programs.

Based on the explanation given above about the available tools, it is evident that a tool to identify the existence of interference in *AspectJ* programs is currently not available. This necessitates the need for a customizable tool to define and identify interference rules for *AspectJ* programs.

4. MOTIVATION

During the development of an *AspectJ* application there is a possibility that more than one advice need to be woven at the same join point. Whenever two advices are woven at the same join point, there are possibilities that behavior defined in one advice can interfere with the behavior defined in the other advice. In order to identify this possibility, the developer has to manually check for the existence of interference. For a large *AspectJ* application we cannot manually find the existence of interferences and if so it becomes a cumbersome task. Based on the explanation given in the previous section, the existing tools also do not provide constructs and facilities to define interference rules for *AspectJ* programs. Hence, there is a need for an environment to define and analyze the existence of interferences in *AspectJ* programs.

5. PROPOSED WORK

The objective of this research work has been to develop an environment to define, identify and remove interferences in given *AspectJ* programs. To summarize the following are the list of contributions of this paper.

- Definition of data and control flow interference rules for *AspectJ* programs.
- Identification of the existence of interferences in the given *AspectJ* programs.

- Suggestion of alternate methods to remove the identified interference.

6. INTERFERENCE ANALYSIS IN *AOP*

In programming, modules are designed to implement functionalities of the application. This leads to possibilities that the functionalities of two modules can interfere with each other during its execution. Due to this effect, the program might not generate the expected result leading to flaw in its design. An analysis on the design and the implementation of software is needed to identify the causes of interference. In *AspectJ* program interferences between aspects is possible due to shared join points, order based advices and shared variable between advices. Hence, there is a need to analysis the existence and consequence of interferences between related aspects and advices.

During the sequence of execution of *AspectJ* program, two types of interference, control flow and data flow interference are possible in the given *AspectJ* program.

6.1 Control Flow Interference

Interference that occurs due to actions affecting the passing of control to the next advice or to the base code is called as control flow interference. Consider the case study of an on-line shopping system whose main functionalities are security, persistence, transaction and logging. All these four have been defined as *before()* advices in aspects and woven at a common join point (call to *purchase()* method) shown in Fig. 1.

The functionalities defined in the four *before()* advices are security, transaction, persistence and logging. A brief statement on the purpose of functionalities of the advices is given below:

- **Security**: Checking the validity of the user login and credit card information.
- **Transaction**: To debit the cost of the purchased products.
- **Persistence**: Updating the database once the goods have been sold.
- **Logging**: Maintaining the details of the users and products purchased by them.

In this scenario, the order of weaving the four advices in the shared join point is important because, the order decides the final outcome. Based on the list given above, if the order of execution is (1) (2) (3) and (4) then interference between the advices will not occur. If the order of execution is changed to (1) (3) (2) and (4) then control flow interference can occur which leads to wrong output. Hence, control flow interference occurs when more than one inter dependent and ordered functionality is applied at a single join point.

Interference analysis needs to identify the possible occurrence of this type of interference in the given *AspectJ* code.

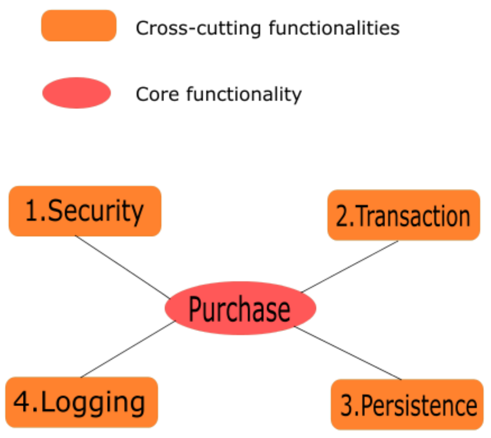


Fig -1: Scenario for Control Flow Interference

6.2 Data flow interference

Interference that occurs due to read/write access by two or more modules on the shared data is called as data flow interference. Consider the case study of complaint registry system whose main functionalities are defined as: (1) complaintChecker() (2) statusChecker() and (3) complaintProgress(). Initial value for the property #OfComplaint is set to zero. A diagrammatic representation of the operations on the #OfComplaint property of complaints functionality shared by the two operations is shown in Fig. 2.

Consider a scenario, with the following sequence of operations executed in order.

- A complaint is registered by a customer, and the property #OfComplaint is incremented by 1.
- Another complaint is registered by the next customer, and the value of #OfComplaint is now 2 (incremented by 1).
- The complaintChecker() removes the first complaint from the list of complaints, processes it to resolve the complaint and the property #OfComplaint is decremented by 1 (#OfComplaint is 1).
- One more complaint is registered by another customer, and the value of #OfComplaint is now 2.
- Now statusChecker() queries the number of unresolved complaints, i.e., value of #OfComplaint is 2.
- If complaintChecker() could not resolve the complaint that is being processed, then #OfComplaint is incremented by 1 and now the value of #OfComplaint is 3.

In step 5 the statusChecker() has reported that the number of unresolved complaints is 2 without considering the complaint being processed by complaintChecker(). Similarly, in step 6 the number of complaints is incremented by 1, since the complaintChecker() could not resolve the complaint that it is processing to resolve. But, the statusChecker() has already reported that the number of unresolved complaints is 2 without counting the complaint that is being processed. Such a scenario is called data flow interference because; the correct value of a property is not available for a module. When a property is shared by two functionalities, there are possibilities for using incorrect values for it. Such a kind of interference between the two functionalities is called as data flow interference. Interference analysis can help in identifying such scenarios in order to remove such interferences from the code.

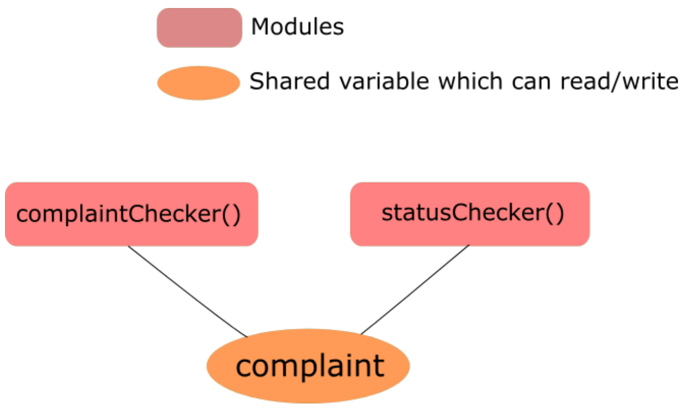


Fig -2: Scenario for Data Flow Interference

1. PROCESS FLOW OF AOSIA TOOL

Based on the explanation given in the previous sections, there is a need to detect occurrence of interference between the constructs in a given *AspectJ* program. The detections can be done manually or using an automated tool. An automated AOSIA tool to detect the interferences has been developed using *Java* programming language. The overall flow of processes designed for the tool is shown in Fig. 3. The first process of *AOSIA* takes an *AspectJ* program as input and frames the corresponding Abstract Syntax Tree *(AST)*. Next, the generated AST will be queried to identify the occurrence of data and control flow interferences. Finally, a report is generated with the list of identified interferences that are present in the given *AspectJ* code and possible removal methods for the interferences.

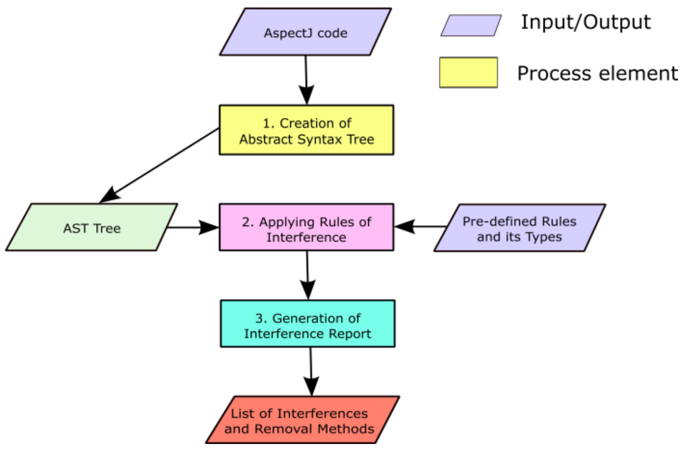

AspectJ code
Input/Output
Process element
1. Creation of Abstract Syntax Tree
AST Tree
2. Applying Rules of Interference
Pre-defined Rules and its Types
3. Generation of Interference Report
List of Interferences and Removal Methods

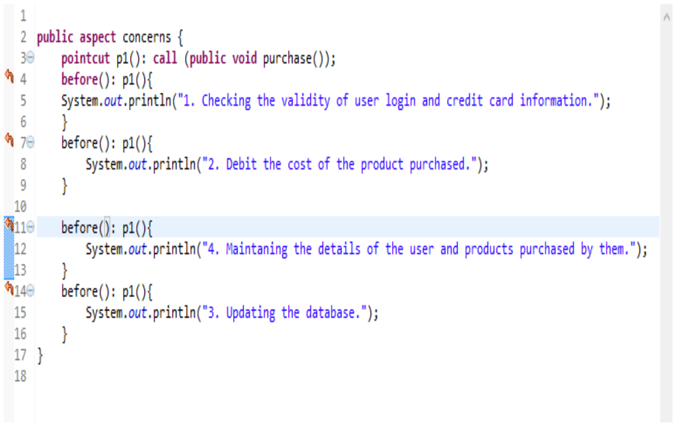
Fig -3: Process Flow of AOSIA Tool

As explained the tool takes *AspectJ* program as input and need to analyze it in order to detect the existence of interferences. This process requires the analysis of the code in the given *AspectJ* program. Some source code analyzers are available in order to analyze different types of errors in a given code. The commonly available source code analyzer tools are *Findbugs* [9], *Check style* [2], *PMD* [2] and *AJATO* [4]. These source code analyzer tools takes code written in different programming languages but there is no extension available for identifying interferences in *AspectJ* software. Even though *AJATO* tool provides facilities to write heuristic rules for *AspectJ* programs, it cannot be extended to define interference rules and to find the existence of interference in a given *AspectJ* program.

Based on this need a tool named Aspect Oriented Software Interference Analysis *(AOSIA)* tool has been developed to identify the occurrence of interferences in the given *AspectJ* programs**.** In the identification and removal of interferences, there are three sub processes namely, (1) Creation of Abstract Syntax Tree (AST), (2) Applying Rules of Interference, (3) Generation of Interference Report.

7.1 Creation of Abstract Syntax Tree (AST)

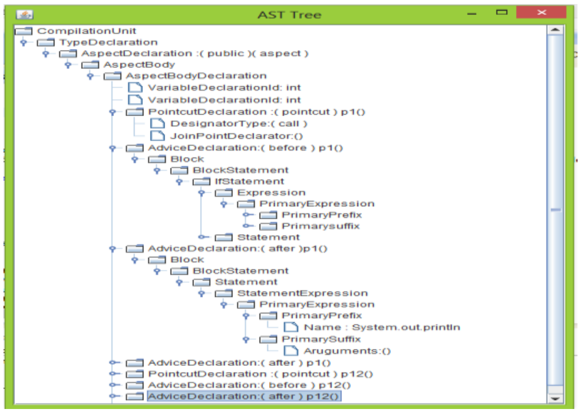
The first process of *AOSIA* tool will allow selecting files containing *AspectJ* code as input using a dialog box. An Abstract Syntax Tree *(AST)* for the selected *AspectJ* program will be generated. AST is a tree representation of the source code which contains nodes representing the constructs of the aspects. The constructs represent statements, conditions, signatures of join points, pointcuts, advices, variables and operators. The expressed syntax is in the AST is "abstract" and does not represent every detail found in the given *AspectJ* program. The primary intension for the creation of *AST* is to use extensively during semantic analysis of the program. During semantic analysis the compiler checks for correct usage of the elements of the program. The *concern* aspect code for the scenario explained to introduce control flow interference in the previous section is shown in Fig. 4.



```

public aspect concerns {
    pointcut p1(): call (public void purchase());
    before(): p1(){
    System.out.println("1. Checking the validity of user login and credit card information.");
    }
    before(): p1(){
        System.out.println("2. Debit the cost of the product purchased.");
    }

    before(): p1(){
        System.out.println("4. Maintaning the details of the user and products purchased by them.");
    }
    before(): p1(){
        System.out.println("3. Updating the database.");
    }
}

```

Fig -4: Code Snippet for Control Flow Interference

The AOSIA tool generates AST tree for a given *AspectJ* program consisting of nodes representing its constructs. The Abstract Syntax Tree generated by the AOSIA tool for the *concern* aspect is shown in Fig. 5.



```
CompilationUnit
  TypeDeclaration
    AspectDeclaration :( public )( aspect )
      AspectBody
        AspectBodyDeclaration
          VariableDeclarationId: int
          VariableDeclarationId: int
          PointcutDeclaration :( pointcut ) p1()
            DesignatorType:( call )
            JoinPointDeclarator:()
          AdviceDeclaration:( before ) p1()
            Block
              BlockStatement
                IfStatement
                  Expression
                    PrimaryExpression
                      PrimaryPrefix
                      Primarysuffix
                  Statement
          AdviceDeclaration:( after )p1()
            Block
              BlockStatement
                Statement
                  StatementExpression
                    PrimaryExpression
                      PrimaryPrefix
                        Name : System.out.println
                      PrimarySuffix
                        Aruguments:()
          AdviceDeclaration:( after ) p1()
          PointcutDeclaration :( pointcut ) p12()
          AdviceDeclaration:( before ) p12()
          AdviceDeclaration:( after ) p12()
```

Fig -5: Abstract Syntax Tree (AST) for *concern* aspect

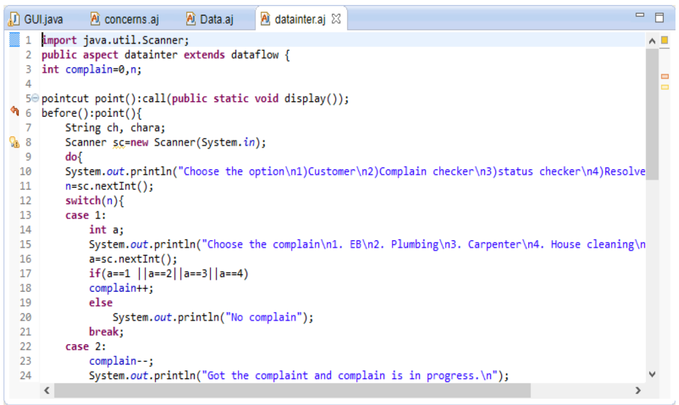
7.2 Applying Rules of Interference

Typically weaving of advice at a common join point may possibly lead to the occurrence of control flow interference between advices of aspects. In the given *AspectJ* code, if more than one advice is woven at a single join point without the definition of order of precedence for advices, then such a scenario will be identified as interference by the AOSIA tool.

The tool will also identify the same type of interference that can possibly occur between aspects in the given *AspectJ* code. *AspectJ* programming language permits the definition of the order in which aspects should execute using dominates construct when more number of aspects are present in *AspectJ* software.

The *concern* aspect given in Fig. 4 is a classical example that can be used to illustrate the occurrence of control flow interference. The aspect contains more than one *before()* advice for the pointcut *p1()*. All the *before()* advices are woven at the same join point. Based on this, it is clearly evident that the order of execution of the advices plays a significant role on the outcome of the program. The order of execution of the four *before()* advices has not been specified in the given code. Hence, there is a possibility for the occurrence of control flow interference between the advices. Since the AOSIA tool has been designed to identify the existence of such a kind of scenario, the tool identifies this scenario as control flow interference.

Based on the scenario provided for the explanation of data flow interference an *AspectJ* code segment has been developed as shown in Fig. 6.



```
import java.util.Scanner;
public aspect datainter extends dataflow {
int complain=0,n;

pointcut point():call(public static void display());
before():point(){
    String ch, chara;
    Scanner sc=new Scanner(System.in);
    do{
    System.out.println("Choose the option\n1)Customer\n2)Complain checker\n3)status checker\n4)Resolve
    n=sc.nextInt();
    switch(n){
    case 1:
        int a;
        System.out.println("Choose the complain\n1. EB\n2. Plumbing\n3. Carpenter\n4. House cleaning\n
        a=sc.nextInt();
        if(a==1 ||a==2||a==3||a==4)
        complain++;
        else
            System.out.println("No complain");
        break;
    case 2:
        complain--;
        System.out.println("Got the complaint and complain is in progress.\n");
```

Fig -6: Code Snippet for Data Flow Interference

Since, statusChecker() and complainChecker() methods access the same property which is #OfComplaint, the outcome of the execution of both the methods depends upon the value of #OfComplaint. As explained previously, data flow interference is possible between the methods. The AOSI|A tool will identify such situations existing in the given *AspectJ* program. Once the tool identifies the existence of such instances, it will be added to the interference report.

7.3 Generation of Interference Report

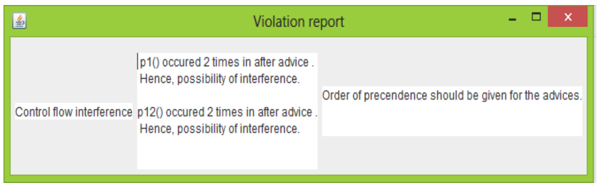
The tool generates a report that includes the name of interference, number of interferences and the corresponding removal methods. The types of interference in the given *AspectJ* program are identified and possible solutions to remove the identified interferences are also added to the report. Two types of removal methods for the identified interferences will be included in the report, dominated language constructs and order of precedence. For example, *AspectJ* permits definition of order in which aspects should execute using "*dominated language constructs*". If two aspects namely, *encryption* and *compression* are defined in an application, then by using the keyword *dominates* it is possible to define the order of execution of the two aspects. If more number of aspects is defined for a single join point and the order of weaving of advices at the common join point is not clear, then the order can be specified by using the "*precedence*" keyword in the aspect. Similar to the previous example, if two aspects containing *encrypt()* and *compress()* functionalities as their respective advices and have the same join point, then the *precedence* keyword can be used to define the order of execution of the two advices.

2. RESULTS AND DISCUSSION

The AOSIA tool has been designed to identify the existence of interferences in the given *AspectJ* program. The tool has been executed by taking the two sample programs as input one at a time. Reports are generated for each sample program. The report clearly indicates the type of interferences identified in the two sample *AspectJ* programs. The report also includes possible removal methods for each of the identified interferences. The interference report generated for the first and second sample programs are shown in Fig. 7 and Fig. 8.

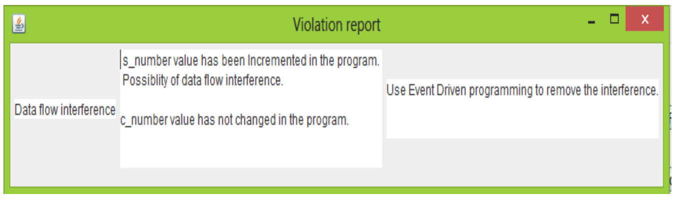
The name of interferences, number of interferences and the corresponding removal methods for the first sample program which is generated by the AOSIA tool is shown in Fig. 7. Similarly, Fig. 8 shows the same characteristics for the second sample program. In the first sample program control flow interference between the four *before()* advices have been identified and the corresponding removal method are generated in the report. For the second sample program occurrence of data flow interference is identified and the needed remedial measure is included in the report.

Now, the interference has to be manually removed by modifying the *AspectJ* programs using the removal methods specified in the report. Further, similar *AspectJ* programs can be given as input to the AOSIA tool to check for the existence of the two types of interference.


Violation report

Control flow interference	p1() occured 2 times in after advice . Hence, possibility of interference. p12() occured 2 times in after advice . Hence, possibility of interference.	Order of precendence should be given for the advices.

Fig -7: Generated Interference Report and Removal methods for Control Flow Interference


Violation report

Data flow interference	s_number value has been Incremented in the program. Possiblity of data flow interference. c_number value has not changed in the program.	Use Event Driven programming to remove the interference.

Fig -8: Generated Interference Report and Removal methods for Data Flow Interference

3. CONCLUSION AND FUTURE WORK

Interference analysis in *AspectJ* programs leads to the identification of code segments that interfere with each other. In this paper, definition and identification of control flow and data flow interference have been clearly explained with suitable examples. Existing tools to identify interferences in other programming languages have been explained and reasons behind the inability to extend them to *AspectJ* are clearly analyzed. Further, a methodology to identify the existence of these two interferences in a given *AspectJ* program has been developed. A *Java* based AOSIA tool has been developed for testing the existence of interferences in a given *AspectJ* program. The tool has been successfully used to identify the interferences found in the given *AspectJ* programs. The two types of interferences identified by this methodology are control flow and data flow interference found in the sample *AspectJ* programs. A report generated by the AOSIA tool includes name of the interferences, number of interferences removal methods for the interferences.

AOSIA tool was used to identify the interferences that can possibly occur in the given *AspectJ* program. Since, source code cannot be queried directly to identify the existence of interference; the concept of *AST* has been introduced to enable modification of source code for the easy identification of interference. In this paper, we have also attempted framing of *AST* for the given *AspectJ* program and the identification of two major types of interferences which may be present in the *AspectJ* program. As an extension it is also possible to extend this methodology to identify other types of interferences that might be introduced in a given *AspectJ* program. The AOSIA tool can also be extended by adding a module to automatically remove the identified interferences. Further, other types of *AOP* languages can also be analyzed for the identification of occurrence of similar types of interferences.

REFERENCES

1. Babu C. and Krishnan H.R., “Fault Model and Test-Case Generation for the Composition of Aspects,” In proceedings of ACM SIGSOFT Software Engineering Notes, pp. 1-6, 2009.
2. http://checkstyle.sourceforge.net/
3. http://www.pmd.sourceforge.net/
4. Disenfeld C. and Katz S, “A Closer Look at Aspect Interference and Cooperation,” In Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, pp. 107-118, 2012.
5. Figueiredo E., Garcia A., and Lucena C., “AJATO: An AspectJ Assessment Tool,” In proceedings of European Conference on Object Oriented Programming (ECOOP Demo), France, 2006.
6. Gradecki J.D. and Lesiecki N, “Mastering AspectJ: Aspect- Oriented Programming in Java,” John Wiley & Sons, 2003.
7. Katz E. and Katz S, “Incremental Analysis of Interference Among Aspects,” In Proceedings of the 7th Workshop on Foundations of Aspect-oriented Languages, pp. 29-38, 2008.
8. Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C.V., Loingtier J.M., and Irwin J., “Aspect-Oriented Programming,” In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, pp. 1-16, 1997.
9. Lauret J., Waeselynck H., and Fabre J.C., “Detection of Interferences in Aspect-Oriented Programs using Executable Assertions.,” In Proceedings of 23rd IEEE International Symposium Software Reliability Engineering Workshop (ISSREW), pp. 165-170, 2012.
10. http://www.findbugs.sourceforge.net/manual/eclipse.html
11. St¨orzer M., “Analysis of AspectJ Programs,” In proceedings of 3rd German Workshop on Aspect-Oriented Software Development, 2003.
12. St¨orzer M. and Jens K, “Interference Analysis for AspectJ,” In Proceedings of the Foundations of Aspect-Oriented Languages, pp. 33- 44, 2003.
13. http://www. checkstyle.sourceforge.net/.
14. Zhang D., “Aspect Impact Analysis,” Ph.D. Thesis, McGill University, 2008.

BIOGRAPHIES

Ms. G. Barani is currently working as a assistant professor in the Department of Computer Science and

DOI: 10.18535/ijecs/v6i2.02

Engineering at Sri Ramakrishna Institute of Technology, Coimbatore. She is interested in the field of Aspect oriented programming, Ontology and Software testing.

Ms. V. Suganya is currently working as a assistant professor in the Department of Computer Science and Engineering at Sri Ramakrishna Institute of Technology, Coimbatore. She is interested in the field of Networks, Aspects programming and Data Structures.

Mr. S. Rajesh is currently working as a assistant professor in the Department of Computer Science and Engineering at Sri Ramakrishna Institute of Technology, Coimbatore. He is having more than 5 years of experience in teaching field. He is pursuing his Ph.D in the field of Mobile Adhoc Networks (MANET).