RESEARCH ARTICLE OPEN ACCESS

Crosscutting Specification Interference Detection at Aspect Oriented UML-Based Models: A Database Approach

Ahmed Sharaf Eldin*, Maha Hana**, Shady Mohammed Elsaid***
*(Department of Information Systems, Faculty of Computers and Information, Helwan University, Egypt)
**(Department of Information Systems, Faculty of Computers and Information, Helwan University, Egypt, Department of Business Technology Information, Canadian International College, Egypt)
***(Department of Information Systems, Faculty of Computers and Information, Helwan University, Egypt, Department of Computer Science, Faculty of Computers and Information Systems, Um Al-Qura University, Makkah Al-Mukaramah, Saudi Arabia)

ABSTRACT
In aspect oriented development, obliviousness is one of its pillars as it helps developers to implement crosscutting concerns via aspects, which increases the overall software modularity. Despite of its merits, obliviousness brings the problem of interferences among aspects as several aspects pointcuts may address the same joinpoint for the same advice. Existing approaches deals with conflicts at design level use graphs structures, which increase in size as project size increases. In this work, a relational database model is used to map aspect oriented design models and then conflicts are extracted by an algorithm runs over this database. This approach is simpler than other approaches and enables large project sizes while the other approaches get complicated due to increment in graph size. The proposed approach can be extended to the distributed team development, dependent on the database engine used.
Keywords – Aspect Oriented Development, Crosscutting Concerns, Databases, Interference Detection.

I. INTRODUCTION

In conventional software development paradigms like object oriented development; a requirement may be needed crosswise some modules. This is called a crosscutting concern. To improve modularity; the concept of aspect orientation is introduced as an extension to object oriented development (1).

Aspects in aspect oriented programming – AOP – implement the crosscutting concerns as separate modules. Aspects are then woven into a certain point in code called joinpoints and implement the crosscutting concerns required in this place. Thus, the overall system modularity is increased (2).

Developers use AOP are not required to know where their aspects are going to be woven into, or what other joinpoints are supposed to be targeted by aspects. This is called obliviousness (3), which is source of AOP strength and conflicts as well (4) (5).

Crosscutting concerns are implemented in aspect via means of pointcuts. A pointcut includes the task required to be done at a specific point in the code called joinpoint in a specific action like method call or execution. A pointcut has to be advised when to run with regard to the joinpoint, either before, after, or around. Aspect weaver is then required to weave the aspect into the point matches the joinpoint signature and advice (6).

A simple example written in AspectJ enclosed in listing 1 illustrates aspectual behavior. It includes a class with an overloaded method, which represents joinpoints. An aspect is defined with only one pointcut matches only one signature of the overloaded method on it call. When a method is called, aspect weaver examines its signature against all joinpoints signatures. If a match occurs, its advice will be woven and run as a part of the running code, otherwise nothing occurs.

```
public class Check
    publics static int add(int x, int y){       //joinpoint
        return x+y;
    }
    public static double add(double x, double y){
                                                //joinpoint
        return x+y;
    }
public aspect Printer{
    pointcut pc(): call (int Check.add(int, int));
    before() : pc(){
        System.out.print("\n Integer Addition: ");   }
 }

public class Main{
    public static void main(String[] args) {
      int x = Check.add(4,5);
     double y = check (4.3,5.3);
     System.out.print(y);
     System.out.print(x);
    }
}
```

```
Output:
9.6
Integer Addition: 9
```

Listing 1 Simple Aspect Oriented Program

Conflicts may occur if two or more pointcuts address the same joinpoint signature. In (2) (4) (5) researches were conducted toward conflicts among aspects. The work presented here proposes aspect conflict detection algorithm – ACDA – that detects conflicts occur among crosscutting specifications in aspect oriented design models. Detecting interferences at design stage gives developers space to resolve it in abstraction level rather than resolving it after coding or having it at runtime.

The rest of the paper is organized as: the second section demonstrates AOP interference problem subjected in this work. The third section shows the related work that addresses AOP interference detection problem. The fourth one explains the proposed technique that uses relational database schema and pseudo code. The fifth section includes a test case and its results run over the proposed solution. Finally, conclusions and expected future work.

II. Crosscutting Interference

Obliviousness may cause aspect developers to write two or more pointcuts that address the same joinpoint at the same advice which results in a conflict. This conflict could be caused by exact method signature matching, or by usage of wildcards that causes a single pointcut to match with several joinpoints with different signatures. A wildcard operator (*) replaces a return type and any character(s) in module or method names, or replaces the entire module or method names. A wildcard operator (..) replaces any number of parameters or none (7). Listing 2 includes a definition to an aspect that causes interferences to the program in listing 1.

```
public aspect InterferenceShow{
    pointcut pcI(): call (int Check.add(int, int));
    before() : pcI(){
        System.out.print("\n TesterI: ");   }

    pointcut pcII(): call (* Check.add(int, int));
    before() : pcII(){
        System.out.print("\n TesterII : ");   }

    pointcut pcIII(): call (* *.ad*(..));
    before() : pcIII(){
        System.out.print("\n TestIII: ");   }

 }
```

Listing 2 Crosscutting Specifications Interference

The first three pointcuts defined in listing 2 causes interference with the joinpoints in listing 1. They all have the same advice, and they match with the joinpoint with definition int Check.add(int, int). The pointcut pcIII matches any method starts with ad that returns any value and declared at any type, class or aspect, with any number of parameters with any type. When considering the obliviousness concept, there is no rule to set the execution order via code. In other words any of these pointcuts can be executed first or last.

III. Related Work

Conflicts among aspects are captured at runtime as unexpected executions or sometimes as runtime errors. Detecting conflicts at design level have several advantages as abstraction in models enables fixing errors in lower cost than in code or maintenance phases. Fixing conflicts at design level removes this potential of deviating from model to actual program. If an aspect oriented CASE tool has code generation feature, then the code generated is free from this conflict types.

In (8) a technique represented that analyze AOP program and then produces a graph that represents each shared joinpoint. The graph has a runtime state representation for this joinpoint and the program elements belong to it such as class and method signature that is matched by the pointcut. Graph transformation rules are then applied to this primary graph. Thus, a meta-graph called labeled transition system – LTS – is generated. LTS helps in recognizing the joinpoint execution. Aspects target this joinpoint are then examined against interference to ensure that the final execution order is not changed due to them. This technique is complicated as it generates a graph for each joinpoint and processes each generated graph before runtime. Also, it captures errors after coding that means high cost of interference resolution. Researches gathered in (5) represent several code level detections for interference among aspect.

Work done in (9) has graph-based model checker named GROOVE (10) as a back end for their work. Initially it transforms the aspect oriented UML-based model into a graph representation. Graph transformations are then produced to simulate the runtime behavior of the aspect UML extended model. This simulation is verified against invariants using computational tree logic expressions to detect conflicts among aspects. Despite of this technique distinction it gets complicated as project size increases as each program element is represented in a graph node and edges represent the relationships among these nodes. It assumes that an aspect oriented model should contain little number of conflicting aspects, otherwise it's a poorly designed model or out of the produced tool capability.

Figure 1 shows a new approach was introduced in (11) to detect conflicts related to intertype declarations based on relational database model. It maps relationships among aspect oriented UML-based model into a database model. Then through a set of relational algebraic expressions, conflicts due to intertype declarations are extracted. This approach differs from the other graph approaches as it simplifies the detection mechanism.

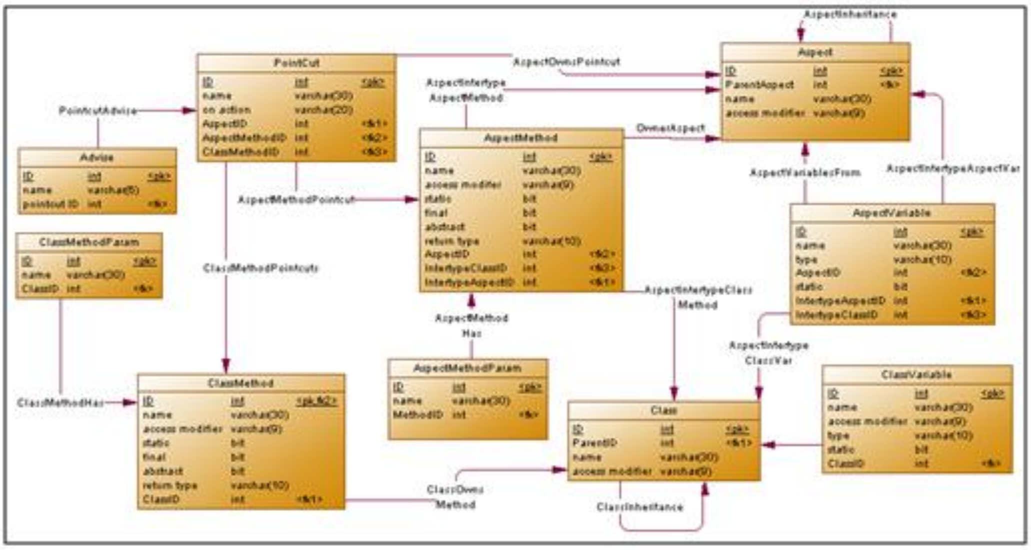

Fig. 1 Detecting Intertype Declaration Conflicts Database Model (11)

IV. Aspect Conflict Detection Algorithm: ACDA

Work presented here relies on (11) model with little modifications to bring obliviousness into practice. Figure 1 shows a pointcut is set to be active on one and only one method defined in a class or an aspect. This is not quite correct as a pointcut may be defined in one and only one method in case of not using wildcards, or may match many methods at several types if the wildcards are used. In figure 2 there is a new database schema focuses on crosscutting specification interferences only not with intertype declarations issue. It overcomes the mentioned limitation and enables obliviousness practice.

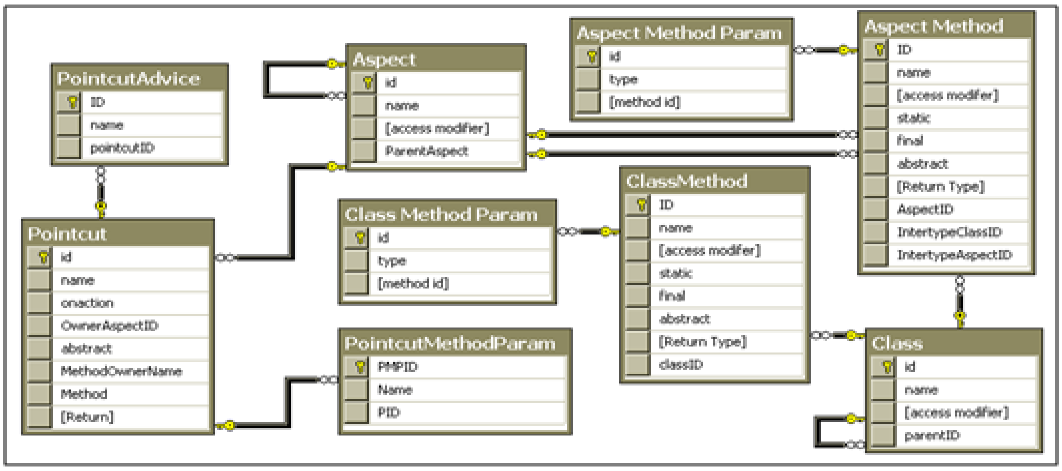

Fig. 2 Enhanced Relational Database Schema Represents Aspect Oriented UML-based Model

```
public aspect InterferenceShow2{
        pointcut pcIV(): call (* Test.addition(..));
        before() : pcIV(){
         System.out.print("\n TestIV: ");   }
  }
```

Listing 3 Obliviousness Example

Listing 3 shows an example for obliviousness where a pointcut – pcIV – is defined over a method called addition with vague parameters declared in a type, class or aspect, named Test. Neither the method exists nor does the type. Despite of this inexistence, aspect oriented development allows such definitions as aspect developer shouldn't have a prior knowledge of the entire system being developed. In figure 2, this concern has been addressed by letting a pointcut defines its method and owner type freely independent from what is already exists.

In the following listings a line numbered pseudo code and SQL statements are used to represent ACDA used to detect crosscutting specification conflicts at aspect oriented UML-based model. Each listing demonstrates a logic unit and a brief illustration is narrated to clear the idea behind. The main objective of this algorithm is to determine pointcuts that match in advice and method signature with regard to wildcard usage. If two or more passed the two tests then they conflict with each other.

ACDA can be viewed as a series of steps starts from extracting aspect oriented design model elements, usually an extended UML class diagram, and store it in ACDA database. Through programming logic represented in listings 4:11 matched pointcuts methods, advices, and parameters are extracted as interfering pointcuts. Figure 3 shows a block diagram represents ACDA.

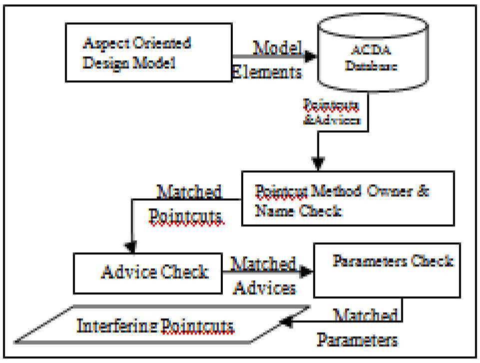

Fig. 3 ACDA Block Diagram

```
ACDA()
1 bParamMatch = false
2 bAdviceMatch = false
3 FOREACH OuterPointcutRecord IN Pointcut
4    OuterPointcutParams ←
              SELECT name as OuterParam
              FROM PointcutMethodParam
              WHERE PID = OuterPointcutRecord.ID
5    OuterPointcutAdvices ←
         SELECT name as OuterAdvice
         FROM PointcutAdvice
         WHERE PointcutID= OuterPointcutRecord.ID
6    InnerPointcuts ←
              SELECT *
              FROM Pointcut
              WHERE ID <> OuterPointcutRecord.ID
         And Method like OuterPointcutRecord.Method
         And MethodOwnerName like
              OuterPointcutRecord. MethodOwnerName
         And [Return] like OuterPointcutRecord.Return
         And OnAction=OuterPointcutRecord.OnAction
```

Listing 4 ACDA: Initiation

Listing 4 includes the initiation phase, a loop start in line 3 is considered as outer loop holds all pointcuts in the system and extracts them one by one. For each extracted pointcut record, its parameters and advices are extracted as well for further comparisons. To lighten the processing on the database engine used, exact values at "where" clause are passed, instead of inner joins. InnerPointcuts record set includes those pointcuts with the same method name, type defined in class or aspect, return value, and action such as call or execute. In this step string values passed after "like" operator is modified by replacing all "*" to the database engine used wildcard such as "%" in Microsoft SQL server.

```
7   FOREACH InnerPointcutRecord IN InnerPointcut
8     InnerPointcutParam←
              SELECT name as InnerParam
              FROM PointcutMethodParam
              WHERE PID = InnerPointcutRecord.ID
9     InnerPointcutAdvices ←
          SELECT Name as InnerAdvice
          FROM PointcutAdvice
          WHERE PointcutID = InnerPointcutRecord.ID
```

Listing 5 ACDA: InnerPoincuts

Listing 5 starts an inner loop deals with the pointcuts found matching with the outer loop current pointcut. For each single record from those inner loop pointcuts, its method parameter(s) and advice(s) are extracted for next step comparisons.

```
10  IF OuterPointcutAdvices.RowsCount <> 0
11    bAdviceMatch= false
12    FOREACH OuterPointcutAdviceRecord IN
      OuterPointcutAdvices
13      FOREACH InnerPointcutAdviceRecord IN
        InnerPointcutAdvices
14        IF OuterPointcutAdviceRecord.OuterAdvice
          == InnerPointcutAdviceRecord.InnerAdvice
15          bAdviseMatch = true
16          BREAK //loop at 13
17        ENDIF
18      ENDFOR
19      IF bAdviceMatch == true
20        BREAK //loop at 12
21      ENDIF
22    ENDFOR
23  ENDIF
```

Listing 6 ACDA: Advice Check

As shown in listing 6, ACDA takes into consideration that a single pointcut may have more than one advice. The check is done as if any advice at the outer loop matched with the one in the inner loop then it shouldn't continue looping and turns bAdviceMatch into true to proceed to the next step. This is a key for performance improvement, not to go to parameter check if no advice matched.

```
24  IF bAdviceMatch == true
25    IF OuterPointcutParams.RowsCount ==
      InnerPointcutParams.RowsCount
26      bParamMatch = false
27      iCursor = 0
28      FOREACH OuterPointcutParamRecord
        IN OuterPointcutParams
29        IF
          OuterPointcutParamRecord.OuterParam
          == ".." OR
          InnerPointcutParam.Records[iCursor].In
          nerParam == ".."
30          iCursor++
31          CONTINUE
32        ENDIF
33        IF
          OuterPointcutParamRecord.OuterParam
          <>
          InnerPointcutParam.Records[iCursor].In
          nerParam
34          BREAK
35        ELSE
36          iCursor++
37        ENDIF
38      ENDFOR
39      IF
        iCursor==OuterPointcutParam.RowsCount
40        bParamMatch = true
41      ELSE IF OuterPointcutParam.RowsCount
        == 0 AND
        InnerPointcutParam.RowsCount == 0
42        bParamMatch = true
43      ENDIF
```

Listing 7 ACDA: Parameter Check – Case I

Listing 7 checks whether two pointcuts are matched. In parameters there are several cases due to wildcard (..) usage that can replace any number of parameters even none. First, ACDA starts with the exact matching case, where no wildcards used and only data types and their order are matched in both outer loop pointcut parameters and inner loop pointcut parameters.

```
44    ELSE IF OuterPointcutParams.RowsCount
      == 1 AND
      OuterPointcutRecord.OuterParam == ".."
45      bParamMatch = true
46    ELSE IF InnerPointcutParams.RowsCount
      == 1 AND InnerPointcutRecord.InnerParam
      == ".."
47      bParamMatch = true
48    ELSE
49      2DotsCount = SELECT count(*)
                  FROM OuterPointcutParam
                  WHERE OuterParam = ".."
50      NormCount = SELECT count(*)
                  FROM OuterPointcutParam
                  WHERE OuterParam <> ".."
51      IF 2DotsCount > 1 AND NormCount == 0
52        bParamMatch=true
53      ENDIF
54    ENDIF
```

Listing 8 ACDA: Parameter Check – Case II

The second case in parameter check comes when the wildcard (..) is used without any real parameters. It has several forms, such as using it only at any of the two pointcuts parameters under check, lines 44-47, or using it multiple times but without any real parameter as well, lines 49-52 in listing 8.

```
2DotsCount = SELECT count(*)
             FROM OuterPointcutParam
             WHERE OuterParam = ".."
IF 2DotsCount >1
  OuterParamWithout2Dots ←
             SELECT OuterParam
             FROM OuterPointcutParams
             WHERE OuterParam <> ".."
  IF OuterParamWithout2Dots.RowsCount
  == InnerPointcutParams.RowsCount
    iCursor=0
    bParamMatch=false
    IF OuterParamWithout2Dots.RowsCount
    ==0
      bParamMatch=true
    ELSE
      FOREACH
      OuterPointcutParamRecord IN
      OuterParamWithout2Dots
        IF
        OuterPointcutParamRecord.OuterPara
        m <>
        InnerPointcutParam.Records[iCursor]
        .InnerParam
          bParamMatch = false
          BREAK
        ELSE
          bParamMatch = true
          iCursor++
        ENDIF
      ENDFOR
    ENDIF
ELSE
  OuterRealParamsCount ←
             SELECT count (*)
             FROM OuterPointcutParams
             WHERE OuterParam <> ".."
  IF OuterRealParamsCount <
  InnerPointcutParams.RowsCount
    iCursorInner =0
    bGap = false
    stOuter =
    OuterPointcutParams.Records[0].OuterP
    aram
    tInner =
    InnerPointcutParams.Records[0].InnerPa
    ram
    IF stOuter == stInner OR stOuter == ".."
      endOuter =
      OuterPointcutParams.Records[last].Out
      erParam
      endInner =
      InnerPointcutParams.Records[last].Inner
      Param
      IF endOuter == endInner OR
      endOuter == ".."
        FOREACH OuterPointcutRecord
        IN OuterPointcutParams
          IF
          OuterPointcutRecord.OuterParam
          == ".."
            bGap = true
          ELSE
            IF bGap == true
              WHILE iCursorInner <
              InnerPointcutParams.RowsC
              ount
                IF
                OuterPointcutRecord.OuterP
                aram ==
                InnerPointcutParams.Records
                [iCursorInner].InnerParam
                  iCursorInner++
                  bParamMatch = true
                  BREAK
                ELSE
                  iCursorInner++
                  bParamMatch = false
                ENDIF
              ENDWHILE
            ELSE
              IF iCursorInner <
              InnerPointcutParams.RowsCou
              nt
                IF
                OuterPointcutRecord.OuterP
                aram ==
                InnerPointcutParams.Records
                [iCursorInner].InnerParam
                  iCursorInner++
                  bParamMatch=true
                ELSE
                  bParamMatch = false
                  BREAK
                ENDIF
              ENDIF
            ENDIF
            bGap=false
          ENDIF
        ENDFOR
      ENDIF
    ENDIF
  ENDIF
ENDIF
ENDIF
```

Listing 9 ACDA: Parameter Check – Case IIIa

As the parameter wildcard (..) can replace any number of parameters including zero, this is the first case addressed in Listing 9. It omits the parameters from the outer loop pointcut and checks if the remaining parameters types match the inner loop one. Case of having this wildcard replaces one and only one parameter type is resolved already within listing 7.

Listing 10 ACDA: Parameter Check – Case IIIb

Listing 10 includes the second case of parameter types matching logic where the parameter type wildcard used several times to replace any number of parameter types. First it has to ensure that the start and the end of the outer loop pointcut parameter types are identical like those for the inner loop or a wildcard parameter type. Second the number of non-wildcard parameter types at outer pointcut parameter types must be less than or equal to those in the inner one. Then, start comparing the inner parameters from the beginning with those at the outer side. If two real parameters are met, then go the next one at both sides, if a wildcard is met then proceed to the next inner parameter type till the end, if found then proceed to the next otherwise if the outer parameter type is not found it means no matching. Finally, if all parameters types in the inner pointcut side are found in the outer one or a wildcard replaces the missed one, the parameters are matched, otherwise no matching.

```
IF bParamMatch
   WRITE OuterPointcutRecord.OuterParam + "
   Interfere With " +
   InnerPointcutParamRecord.InnerParam
ENDIF
bParamMatch = false
bAdviceMatch = false
ENDFOR
ENDFOR
```

Listing 11 ACDA: End

The last step in ACDA is shown in listing 11, as if the parameter types are matched, it means that the advices are also matched because checking parameter types is dependent on the advice. Flags bParamMatch and bAdviceMatch are then reset to false for next iteration.

V. Experiment

In order to test ACDA, extensive test cases are generated including all possible conflict causes. In figure 4, an aspect oriented UML-based model is created with one class named MyClass and two extended classes to represent aspects, aspectA and aspectB.

Pointcuts may target already existing joinpoints or due to obliviousness may address joinpoints not created yet. If a joinpoint already exists, then an extended dependency link, crosscut, will be from aspect defines the pointcut to type owns the joinpoint either class or another aspect. Pointcuts themselves are considered to be an extended type of operations inside aspect type. Extending UML is done by stereotyping a UML model element to the specific domain required. (12) (13).

MyClass has overloaded methods: add and addition. Some pointcuts like pcA1 and pcB2 targets already existing joinpoints at MyClass. Some other pointcuts address joinpoints that do not exist yet like pcB3. Finally, some methods address generic joinpoints like pcGn1 that matches any joinpoint in the system. Table 1 shows data stored in the database that ACDA works on.

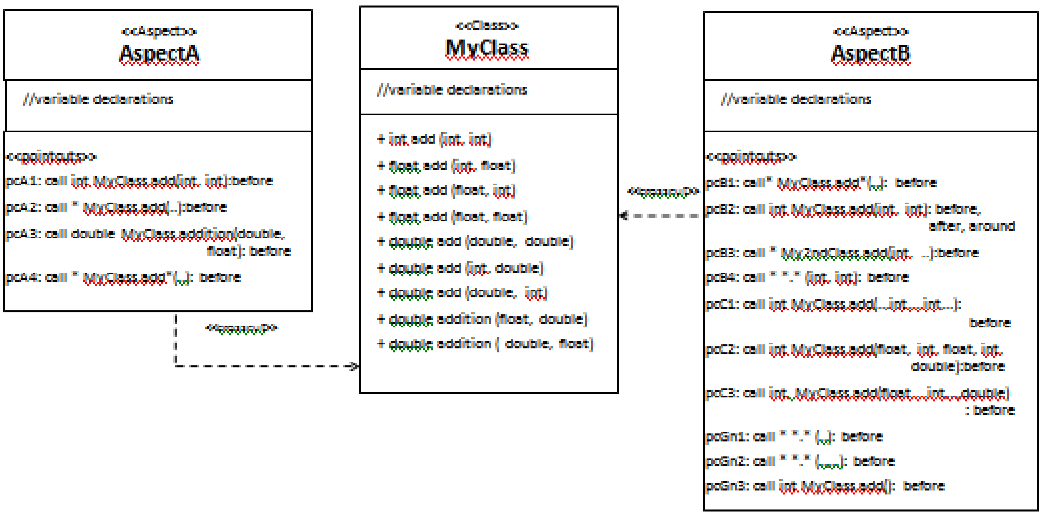

Fig. 4 Aspect Oriented UML-based Model: ACDA Test Cases

Class

ID	NAME	ACCESS MODIFIER	PARENTID
15	MyClas s	public	NULL

ClassMethod

ID	NAME	ACCESS MODIFIER	STATI C	FINAL	ABSTRA CT	RETURN TYPE	CLASSI D
26	add	public	0	0	0	int	15
27	add	public	0	0	0	float	15
28	add	public	0	0	0	float	15
29	add	pubic	0	0	0	float	15
30	add	public	0	0	0	double	15
31	add	public	0	0	0	double	15
32	add	public	0	0	0	double	15
33	addition	public	0	0	0	double	15
34	addition	public	0	0	0	double	15

Class Method Param

ID	TYPE	METHOD ID
30	int	26
31	int	26
32	int	27
33	float	27
34	float	28
35	int	28
36	float	29
37	float	29
38	double	30
39	double	30
40	int	31
41	double	31
42	double	32
43	int	32
44	float	33
45	double	33
46	double	34
47	float	34

PointcutMethodParam

PMPID	NAME	PID
3	int	16
4	int	16
5	..	18
6	double	19
7	float	19
8	..	20
9	..	21
10	int	22
11	int	22
12	int	23
13	..	23
14	int	24
15	int	24
16	..	25
17	int	25
18	..	25
19	int	25
20	..	25

21	float	26
22	int	26
23	float	26
24	int	26
25	double	26
26	float	27
27	..	27
28	int	27
29	..	27
30	double	27
31	..	28
32	..	29
33	..	29

Aspect

ID	NAME	ACCESS MODIFIER	PARENTASPECT
12	aspectA	public	NULL
13	aspectB	public	NULL

Pointcut

ID	NAME	ON ACTION	OWNERASPECTID	ABSTRACT	METHOD OWNER NAME	METHOD	RETURN
16	pcA1	call	12	0	MyClass	add	int
18	pcA2	call	12	0	MyClass	add	*
19	pcA3	call	12	0	MyClass	addition	double
20	pcA4	call	12	0	MyClass	add*	*
21	pcB1	call	13	0	MyClass	add*	*
22	pcB2	call	13	0	MyClass	add	int
23	pcB3	call	13	0	My2ndClass	add	*
24	pcB4	call	13	0	*	*	*
25	pcC1	call	13	0	MyClass	add	int
26	pcC2	call	13	0	MyClass	add	int
27	pcC3	call	13	0	MyClass	add	int
28	pcGn1	call	13	0	*	*	*
29	pcGn2	call	13	0	*	*	*
30	pcGn3	call	13	0	MyClass	add	int

PointcutAdvice

ID	NAME	POINTCUT ID
15	before	16
16	before	19
17	before	20
18	before	22
19	before	23
20	before	24
21	before	18
22	before	21
23	after	22
24	around	22
25	before	25
26	before	26
27	before	27
28	before	28
29	before	29
30	before	30

Table 1 ACDA Test Cases Equivalent Data

VI. Results

After running ACDA, the following results in table 2 come out. Each pointcut is examined against the rest pointcuts, and the pointcuts interfere with it only will appear as a conflict points, denoted by (♦) in the intersection between the row and the column represent each pointcut.

It is not always a mutual exclusive task, meaning that a certain pointcut may interfere with another one and vice versa, or may not. If two or more pointcuts address a certain joinpoint signature, they are conflicting mutually exclusive, such as pcA1 and pcB2. If one or more of them address the joinpoint via wildcard, it means that the wildcard holders are conflicting with other pointcuts but not necessarily the others do, such as pcA2 and pcA1.

Table 2 shows diagonal in shaded form as ACDA can recognize that a pointcut cannot interfere with itself although matching occurs. Other empty cells also indicated there is no conflict between the two pointcuts at the row and column headers and they are different.

		Conflicted Pointcuts													
		pcA1	pcA2	pcA3	pcA4	pcB1	pcB2	pcB3	pcB4	pcC1	pcC2	pcC3	pcGn1	pcGn2	pcGn3
Conflicting Pointcuts	pcA1						♦								
	pcA2	♦					♦			♦	♦	♦			♦
	pcA3														
	pcA4	♦	♦	♦		♦	♦			♦	♦	♦			♦
	pcB1	♦	♦	♦	♦		♦			♦	♦	♦			♦
	pcB2	♦													
	pcB3														
	pcB4	♦	♦		♦	♦	♦	♦					♦	♦	
	pcC1	♦					♦				♦				
	pcC2									♦					
	pcC3									♦	♦				
	pcGn1	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦		♦	♦
	pcGn2	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦	♦		♦
	pcGn3														

Table 2 ACDA Experiment Results

VII. Conclusion and Future Work

Although AOP takes modularity to its extreme, it introduces problem of conflicts among its modules. Approaches discussing this problem from graph perspective resolved this problem within limit due to its complexity.

The approach discussed in this paper is believed to provide an automated, modular, and simple solution to a complicated problem in aspect oriented design models. Automation comes as there is no manual user interactions required for the conflicts extraction. Modularity comes as the detection is done isolated from the design model and won't affect it. Simplicity comes as to implement ACDA there is no need for sophisticated techniques or expertise.

ACDA relies on the UML-based ones, but it can be extended to any design model takes into consideration that aspect oriented development is an extended form of object oriented development. The solution provided in (11) can be augmented to the solution proposed here to resolve both conflict types in intertype declarations and crosscutting specifications.

In this approach, queries are done over pointcut, pointcut method param, and advice tables. Thus, it isn't affected by number of aspects, or classes and therefore it reduces the overall cost of detection process.

ACDA avoids self-join queries by passing parameters to a new query for extracting data. This increases the efficiency of ACDA as database engines uses indexers over its key attributes. For those non-key attributes indices can be created to enhance ACDA performance as well.

CASE tools supports aspect oriented modelling can be supported by ACDA either with a local database file or a server database in case of multiuser environment. If a local file solution is selected, XML format and X-Queries can be used to implement ACDA. Standardizing aspect modelling either by UML-based extensions or as a new modelling technique is now useful to support aspect oriented development after detection crosscutting specification and intertype declaration interferences easily. Thus, aspect oriented development can be refreshed up again.

References

[1] Berg, Klaas van den, Conejero, Jose Maria and Chitchyan, Ruzanna. *AOSD ontology 1.0 public ontology of aspect orientation*. s.l. : Common Foundation for AOSD, 2005. p. 90, Report.

[2] Aspects: Conflicts and Interferences (A Survey). André Restivo, Ademar Aguiar. 2007. *Proceedings of the 2nd Conference on*

Methodologies for Scientific Research. pp. 145-153.

[3] Robert E. Filman, Daniel P. Friedman. *Aspect-Oriented Programming is Quantification and Obliviousness*. Research Inistitution for Advanced Computer Science. s.l. : Workshop on Advanced Separation of Concerns, 2001.

[4] Durr, Pascal, Bergmans, Lodewijik and Aksit, Mehmet. *Reasoning about Behavioral Conflicts between Aspects*. Enschede : University of Twente, 2007. Technical Report . TR-CTIT-07-15.

[5] Katz, Emillia, et al. *Detecting Interference Among Aspects.* Computing Department, Lancaster University. Lancaster : European Network of Excellence on Aspect-Oriented Software Development, 2007. p. 38, Report.

[6] Aspect-Oriented Programming. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes,. Finland: Springer-Verlag, 1997. *European Conference on Object-Oriented Programming (ECOOP).* pp. 220-242.

[7] Kiselev, Ivan. *Aspect_Oriented Programming with AspectJ.* 2nd Edition. s.l. : SAMS, 2003. 0-672-32410-5.

[8] Staijen, Tom and Rensink, Arend. *A graph-transformation-based semantics for analysing aspect. Natal, Brazil : Workshop on Graph Computation Models,*, 2006.

[9] Ciraci, Selim, et al. *A graph-based aspect interference detection approach for UML-based aspect-oriented models*. [ed.] Shmuel Katz and Mira Mezini. Transactions on aspect-oriented software development. Heidelberg : Springer-Verlag, 2010, Vol. VII, pp. 321- 374.

[10] Model Checking Dynamic States in GROOVE. *Kastenberg, Harmen and Rensink, Arend.* Berlin : Springer-Verlag, 2006, Vol. 3925.

[11] *Detecting Aspect Intertype Declaration Interference at Aspect Oriented Design Models: A Database Approach*. Sharaf Elin, Ahmed, Hana, Maha and Mohammed Elsaid, Shady. 7, India : IJERA, July 2014, International Journal of Engineering Research and Applications, Vol. 4, pp. 164-171. 2248-9622.

[12] *UML Extensions for Aspect Oriented Software Development*. Losavio, Francisca, Matteo, Alfredo and Morantes, Patricia. 5, July 2009, Journal of Object Technology, Vol. 8, pp. 85-103.

[13] *Extending UML Using Enterprise Architect. UML tools for software development and modelling - Enterprise Architect UML modeling tool.* [Online] 5 2010. [Cited: 1 24, 2012.] http://www.sparxsystems.com/bin/EAUserGuide.pdf.