Aspect Interference Analysis Using Component Filter Model Semantics and Slicing

Rishabh Shukla, Subramanhyam Kuntamukkala
Infosys Research Labs, Infosys Ltd.

Abstract

Aspect Oriented language aims to make cross-cutting concerns clearly identifiable with special linguistic construct called aspects. In order to analyze the properties of an aspect one should consider the aspect itself and the part of the system it affects. This part is just a slice of the entire system and can be extracted by exploiting program slicing algorithms. However they will behave correctly in isolation, but when interaction changes an aspect's behavior or disables and aspect, we will term it as aspect interference. We will propose an approach to detect aspect interference, Aspect composition are modeled by using graph production system for modelling aspect-language semantics. This graph is transformed into runtime-state representation. Combined with the production system (also with proper tool) the execution of the aspect is simulated. This simulation results in LTS(labelled transition system) that can be used to analyze verify different behavior at join points..

1. Introduction

Aspect-oriented programming(AOP) is a widely accepted language concept to improve separation of concerns on the implementation level. Before or during the execution of the program the behavior of the aspects is imposed on to the base .One of the major advantage of this is that is allow separate development of base program and the aspects. In Section 2 We will discuss the method of slice extraction,In this we have taken a sample code and generated its corresponding Control Dependence Graph and Flow Dependence Graph. Finally we have extracted backward slice of the sample code. In Section 3 We have discussed issues of analyzing interaction of aspect. In Section 4 we have discussed Conclusion where we discuss interaction of slice and aspect and a way to avoid any interference. In Section 5 We suggested the tool that will be used in out proposed scheme. We propose the usage of GROOVE for implementation of this approach.

2. Study of Slice Extraction

Program Slicing[1] is a technique aimed at extracting program elements related to particular computation. A slice of program is a set of statements which affect a given point in a executable program. There are basically two types of slicing in which one can compute statically the set of statements that potentially affect the slicing criterion for every possible program execution. The other technique consider the information about a particular execution of program and derive a dynamic slice[2] of a program.There are three type of slice, Thefirst one is Backward Slice which is at point p is the program point p is the program subset that may affect p. The second one Forward Slice at point p is the program subset that may be affected by p. The program subset between program points p and q that may be affected by p and that may affect q is called chop.

Slicing can be done with the help of Program Dependence Graph(PDG) in which Nodes are statements and Edge represent either Control Dependence or Data dependence. Backward slice can be computed from point p, by computing backward reachability in the PDG from node p. Forward slice can be computed from point p by computing forward reachability. To compute chop between p and q identify all paths between p and q.

We will explain slice extraction with a example code. Firstly we will develop Control Dependence Graph for that sample code as shown in Fig 1. The edge from one node to another node will be there if edge from first node branches one way, another node will be eventually reached and if edge from first node branches another way than second node may not be reached.

The second is the Flow Dependence Graph as shown in Fig 2 which will together form Program Dependence Graph. For Flow dependence graph edge from one node to another node will be there if values of variable assigned at first node may be used at second node. For our sample code the Flow Dependence graph is shown below.

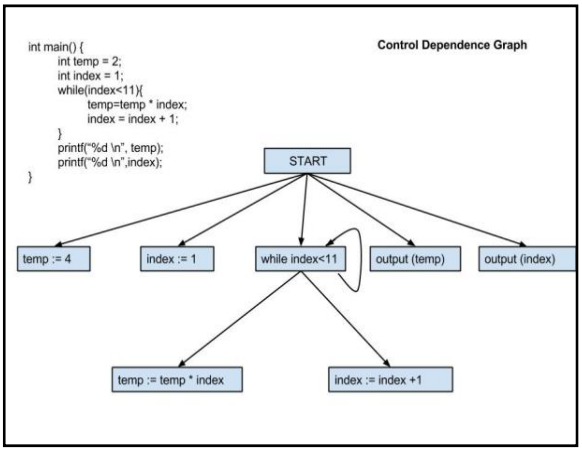

Figure 1.

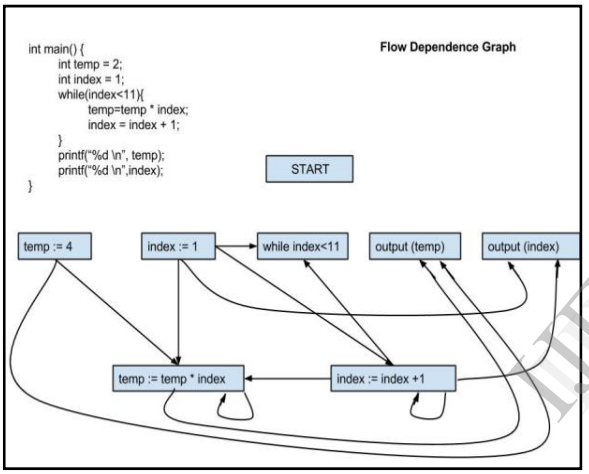

Figure 2.

To find Backward Slice we will find the backward reachability. The node "output(index)" has incoming edge from three nodes and don't have any outgoing edge this corresponds to line 1 , 3 ,6 and last line. Similarly we will continue and finally reach to node involving the loop condition this will correspond to 4 line of the sample program . The final extracted slice shown in Fig 3. In the figure the bold line corresponds to that of flow dependence graph and simple line is of control dependence graph.

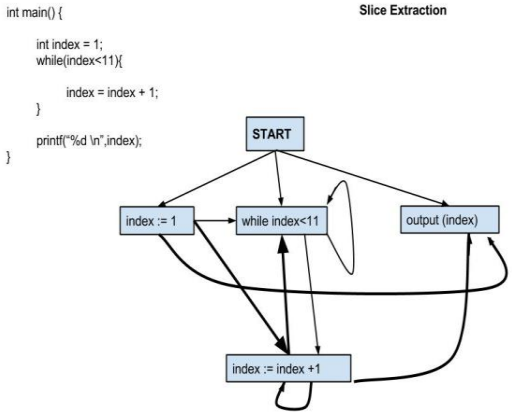

Figure 3.

3. Analysis of Aspect Interaction

This section deals with analysis of interaction among aspects. An aspect oriented program is composed by weaving aspect and class together. The newly formed aspect is weaved with and it add it as new cross-cutting concern functionality without breaking the rules.Let a code unit be an aspect or a class of a system. We say that an aspect SampleAspect does not interfere with code unit SampleClass if and only if every interesting predicate on the state manipulated by SampleClass is not changed by the application of SampleAspect. For instance if an object sampleObject manipulated by SampleClass exist such that the predicate sampleObject<= 0 must hold for the correctness of the system, SampleAspect does not interfere with SampleClass only if SampleClass woven with SampleAspect preserves sampleObject<= 0.

Let SampleAspect1 and SampleAspect2 be two aspect and SampleSlice1 and SampleSlice2 the corresponding backward and forward slices obtained by using pointcuts declarations defined in SampleAspect1 and SampleAspect2 as slicing criteria. Now we need to identify interference between SampleAspect1 and SampleAspect2.

3.1. Composition Filter Model

It is extension of conventional object-based model, where objects are enhanced with filters for the manipulation of incoming and outgoing messages.Filters are grouped into components called filter models shown in Fig 4. These units of reuse provide execution context for the filters.

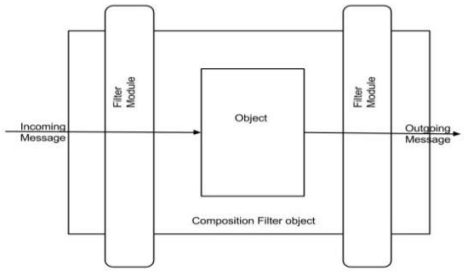

Figure 4.

Composition Filters concept can be mapped to those of regular AOP-language. Superimposition specification pointcut designator. We present a sample code of SampleAspect in Fig 5, it consist of filter module named SampleModule, which contain one input-filter. This filter is evaluated when a message is received by an object enhanced with this filter-module. The input-filter declaration contains the name of the Sample filter and a matching pattern which matches the selector send. A substitution part(*.*) will pass the matched target and selector to the action performed by the filter. The superimposition selects class Server using query on the static structure of the base program, and superimposes the SampleModule filter module on this class, Thus, whenever a method named send is called on an instance of class Server.

```
concern SampleAspect {

    filtermodule SampleModule {
        inuptfilters :
        Sample : Sample = { [*.send] *.* }
    }

    superimposition {
    filtermodule

     classes = { x | ClassByName(x , 'Server')};
    superimposition

    classes <- SampleModule;
    }
}
```

Figure 5.

Now we have defined composition filter model we have to check the interference for the following condition. We should ensure that.

SampleAspect1 ∩SampleSlice2 = NULL

AND

SampleAspect2 ∩SampleSlice1 = NULL

Now with aspects and slice we will generate a transition system of execution using graph transformation based operational semantics. We will then identify the occurrence of above two cases from this transition system. For a Composition Filters program we will generate a graph of Abstract Syntax Tree.

3.2. Production Rules

In order to carry out transformation and generate state spaces we propose to use GROOVE as a tool. GROOVE notation shown in Fig 6 contain nodes and edges, the labels in nodes are in fact self-edges connected to those nodes Different line style have different significance.

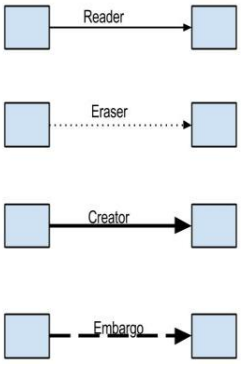

Figure 6.

The first figure shows a normal line is called as Reader element and used for matching, the second with dashed elements are eraser elements which will be removed and thus also are required for matching the rule, the third figure which represent thick lines represent creator elements which will be added to the graph when the rule applied. The fourth is thick dashed line represent embargoes, it is negative application conditions which when matched prevent the rule from being applicable.

From the AST,we will generate Abstract Syntax Graph, By the time the graph is generated the complier has already resolved the superimposition part and the filtertype(which is replaced by the accept and reject action.)

3.3. Generation of Control Flow Graph

Then next step is to add control flow information and we will get Control Flow Graph. It consists of flow

and branch edges; the latter lead to dedicated Branch nodes, which in turn identify the value under which a particular control flow is taken. Then we use production system for simulation of execution where ever rule specified the runtime semantics of a single flow element.

4. Conclusion

In our proposed approach allows to abstractly specify the behavior of advice actions, such that only relevant behavior is in-corporated. Even though it doesn't guarantee that a composition of aspect is free of interference, there will be a warning for interference in case of non-confluent result. We propose that when advices are commutative for every combination of condition value the shared join points is highly likely free of interference. In Labeled Transition System the visual nature helped in getting the knowledge of composition of advices, even as simply as seeing different shapes under difficult condition values. This will help in decision for debugging purpose.

5. Tool Support

In our proposed method the graph generator will be implemented as s Compose complier module which will be compile time and run time implementation of Composition Filter language. Compose is available both in Java and .Net platform.

After graph have been generated run-time simulation is started. The final Labeled Transition System can be opened in GROOVE viewer. Analysis of the state space to give understandable feedback to the user can only be obtained by visual aid, automatic capability is still not there.

10. References

[1] Weiser, M., "Program Slicing", IEEE Transactions on software engineering, Vol. 10, Issue 4, 1984, 352-357.

[2] Korel, B. and Laski, J., "Dynamic Program Slicing", Information Processing Letters, Vol. 29, Issue 3, doi>10.1016/0020-0190(88)90054-3, 26 October 1988,155-163.

[3] Mehmet Aksit ,ArendRensink, and Tom Stajien "A graph-transformation-based simulation approach for analysing aspect interference on shared join points"AOSD'09 March 2-6, 2009, Charlottesville, Virginia, USA.

[4] DavideBalzarotti, MattiaMonga "Using Program Slicing to Analyze Aspect Oriented Composition" Foundation of Aspect Oriented Language2004.

[5] Tom Staijen, ArendRensink "A Graph Transformation-Based Semantics for Analysing Aspect Interference" .

[6] Kim Mens, Tom Tourwe "Evolution Issues in Aspect-Oriented Programming".