

## ASPECTUAL CONCEPTS

by John J. Sung

Advisor: Karl Lieberherr

Reader: Mitch Wand

Aspect Oriented Programming (AOP) is becoming prominent in the field of Computer Science. Much work has been put into creating tools that allow programmers to use AOP to improve their productivity. The purpose of this research is to understand the fundamental concepts behind aspect oriented programming, how these concepts are applied in different Aspect Oriented Programming tools, and create a coherent model of AOP concepts that would allow us to make informed decisions about how these concepts could be applied in novel ways.

Let's take a look at some fundamental concepts that are evident in AspectJ, DJ, and Branch Oriented Programming. These tools use the concepts of predicated execution, advice and traversal in different ways. AspectJ uses predicated execution and advice concepts on a program's control flow graph. Branch Oriented Programming attempts to define the program's control flow graph using the predicated execution and around concepts. DJ uses traversals and advice for data structures. Some of these tools have been around for years, but we still do not understand how these tools are related. We also do not know how other fundamental concepts are related and how we should view these tools during tool analysis.

One way to explore these issues is to attempt to implement concepts from one tool to another. The simple implementation of factorial function written in Fred and AspectJ with Branches shown below illustrate how one can apply the concepts of branches using AspectJ.

Factorial example in Fred:

```
(define-msg fact)

(define-branch (and (eq? (dp-msg dp) fact)
  (= (car (dp-args dp)) 1))
  1)

(define-branch (eq? (dp-msg dp) fact)
  (let ((x (car (dp-args dp))))
    (* x (fact (- x 1)))))
```

Factorial Example in AspectJ with Branches:

```
class FactClass {}
aspect FactExample {
```

```

static int FactClass.fact(int x) {
    return 1;
}

pointcut factBase(int x):
    args(x) && call(static int FactClass.fact(int))
    && if(x==1);

int around(int x) : factBase(x) {
    return 1;
}

pointcut factRecursion(int x):
    args(x) && call(static int FactClass.fact(int))
    && if (x > 1);

int around(int x) : factRecursion(x) {
    return x * FactClass.fact(x-1);
}
}

```

In this implementation the FactClass is empty and the factorial method is defined within the FactExample aspect. The two pointcuts and the corresponding around advices are the two possible branches from the method fact().

The close correspondence between the Fred implementation and the branches with AspectJ implementation of the factorial function is very clear in this example. There is almost one to one correspondence between the two implementations. This illustrates the possibility of implementing the Branch Oriented Programming concept using AspectJ and that the process of adapting Branch Oriented Programming concepts into AspectJ was relatively easy.

In order to find more of these relationships, one can see that a consistent view that one can take to make all of the AOP approaches more clear would be useful. One way to accomplish this is to view these ideas as abstracting the different parts of a program into graphs. The nodes and edges have different meaning, but the basic themes are the same. This allows us to think of all of the different things that one can do with graphs, traversals, mapping, grouping of nodes, sub-graphs, etc. If we apply these concepts to the different graphs evident within programs and consistent with the different domain of Aspect Oriented Programming tools, we can find possibilities for future development of Aspect Oriented Programming.

The idea of abstracting these concepts into generic graphs will be explored by attempting to implement an idea from a graph operation to different tools. In the example before, we're looking at the case where one wants to have access to some node or context that the entity doing the traversal has seen before.

In AspectJ you would use the pointcut to specify the specific join point that you want to expose to the join point encountered later in the traversal of the dynamic call graph. In the example, we want to expose the Caller c to all of the Workers w.

```

pointcut perCallerWork(Caller c, Worker w):
    cflow(invocations(c)) && workPoints(w);

```

With traversal strategies specified in [demeter strategy paper]

<http://www.ccs.neu.edu/research/demeter/biblio/strategies.html>  
we would specify a strategy and a NodesOf operator:

NodesOf(from Caller to \*)

and we would intersect this set of classes with the set of classes  
in set Worker:

NodesOf(from Caller to \*) intersect Worker

The NodesOf operator, when applied to a graph, returns all the nodes  
in the graph and it would be useful to extend AspectJ with  
strategies and NodesOf to define AspectJ type patterns more flexibly.

In DJ, one would use the ContextVisitor to expose the stack of objects that the visitor has  
encountered thus far in the object graph during a traversal of the object graph.

In all of these cases, the program can expose parts of the graph that was traversed earlier  
during the current traversal. This is a fundamental concept that are evident in AspectJ,  
DemeterJ and DJ. This thesis will attempt to find and categorize these types of commonalities  
between AOP tools and integrate them into a consistent view of Aspect Oriented  
Programming concepts.

The relevant experiments for this thesis include using AspectJ to implement the concepts from  
other AOP tools, integrating AspectJ and AP Library and implementing a real-world  
application using the tool from the integration work. The purpose of the first experiments are  
to find out the relationship between AspectJ and other tools. Some of these experiments were  
shown earlier. The main focus of these experiments, integrating AspectJ and AP Library, will  
find out how well these tools interact with each other. The last experiment will measure the  
effectiveness of the integration experiment. All of these experiments have only one purpose, to  
direct the shaping of the consistent view of Aspect Oriented Programming that integrate  
concepts from AspectJ and Demeter.

The consistent view of Aspect Oriented Programming that will be shaped by the proposed  
experiments should be based on abstracting Aspect Oriented Programming concepts into  
graphs and the manipulation of these graphs as mentioned previously. This new view should  
be a starting point for a new way of analyzing Aspect Oriented Tools, i.e. Aspect Oriented  
Tool Analysis. This new type of analysis should expand the current understanding of Aspect  
Oriented Programming tools and concepts that will lead to future development of those tools  
and concepts.