Clamp: Automated Joinpoint Clustering and Pointcut Mining in Aspect-Oriented Refactoring

Prasanth Anbalagan Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
panbala@ncsu.edu, xie@csc.ncsu.edu

ABSTRACT

Software refactoring consists of a set of techniques to reorganize code while preserving the external behavior. Aspect-Oriented Programming (AOP) provides new modularization of software systems by encapsulating crosscutting concerns. Based on these two techniques, Aspect-Oriented (AO) refactoring restructures crosscutting elements in code. AO refactoring includes two steps: aspect mining (identification of aspect candidates in code) and aspect refactoring (semantic-preserving transformation to migrate the aspect-candidate code to AO code). Aspect refactoring clusters the join points for the aspect candidates and encapsulates each cluster with an effective pointcut definition. With the increase in size of the code and crosscutting concerns, it is tedious to manually identify aspects and their corresponding join points, cluster the join points, and infer a pointcut expression. This paper proposes an automated framework that clusters join point candidates and infers a pointcut expression for each cluster.

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [3] provides constructs for modularizing crosscutting concerns. Existing software applications often contain instances of such crosscutting concerns. Refactoring such applications towards AOP helps modularize these crosscutting functionalities. Aspect-Oriented (AO) refactoring [4] provides additional means to conventional refactoring techniques. While steps in conventional refactoring modularize code into a clean OO implementation, the use of AOP squeezes out code that cannot be further refactored. We next illustrate the aspect-oriented refactoring process through an example.

Figure 1 shows the implementation of an `Account` class that performs a permission check at the beginning of two methods. Here a method call to `checkPermission` is spread into these two methods. Here we create a pointcut that captures all the join points where we would like to add the refactored functionality. To minimize unwanted effects, the pointcut simply enumerates each of the required methods. The pointcut definition is given below:

```
private pointcut permissionCheckedExecution() :
    ( execution(public void Account.creditAccount(float)) ||
      execution(public void Account.debitAccount(float));
```


FSE '06 Portland, Oregon


```
public class Account {
  public Account(int accountNumber) {
    _accountNumber = accountNumber;
  }

  public void creditAccount(float amount) {
    AccessController.checkPermission("accountOperation");
    _balance = _balance + amount;
  }

  public void debitAccount(float amount) {
    AccessController.checkPermission("accountOperation");
    ....
  }
```

Figure 1: Partial Implementation of an Account Class

The preceding example shows refactoring for a simple case of repeated method invocations in each method. Here no *clustering* has been done on the join points to group similar join points together so that each cluster can be characterized by a simple pointcut. Instead, all the join points have been combined using the logical operator "||". In real-world applications, the source code would span thousands of lines and it would be tedious to manually identify aspects and their corresponding join points, cluster the join points, and infer a pointcut expression. In this paper, we propose an automated framework, called Clamp, to address the problem of clustering the join point candidates (identified manually or automatically by existing aspect mining techniques [2]) and inferring a pointcut expression for each cluster; our framework complements the existing aspect mining techniques [2], serving as a post-processor of the results produced by these aspect mining techniques.

2. APPROACH

The input to our proposed Clamp framework includes aspect candidates and join point candidates. The inputs are provided either manually or automatically by an aspect mining tool [2], which identifies aspect candidates and and their respective join point candidates in the original code. Our framework uses existing aspect mining tools like Aspect Mining Tool (AMT) [1] to identify aspects and existing data mining tools like WEKA[1] to cluster identified join point candidates. Then the clustered join points are fed as input to an inference engine. The inference engine, based on a string alignment algorithm, is used to infer pointcuts for the clustered join points.

Figure 2 provides an overview of our Clamp framework. Our framework consists of two components: the clustering engine and inference engine. The clustering engine receives the aspect candidates and join point candidates (identified before using our framework) as inputs. In order to perform clustering, we process the join

[1] `http://www.cs.waikato.ac.nz/ml/weka/`

Figure 2: Overview of the Clamp framework

point candidates to form a data set, which is fed as input to the data mining tool. The data set includes two sections: attributes and data. The attributes are usually different naming parts of a join point candidate for an aspect and are the prime factors based on which we perform the grouping of join points. For example, if a join point is that of a method execution, then the attributes would be naming parts of the method, i.e., its modifiers, return type, method name, and arguments. The data contains distinct values, i.e., the join point candidates arranged in the order described by the attributes. A sample data set has been shown in the preliminary results section.

We provide the user with an option to select the attributes based on which the clustering is performed. By default, our approach automatically chooses the prime factors based on which the grouping of join points is performed. In our current implementation, the prime factor has been chosen as the name field of a join point candidate. The name field of a join point candidate is split into different parts. Splitting the name field gives a higher probability of detecting a common pattern among the name fields of different join points. For example, consider the method names of the join points `creditAccount` and `debitAccount`. The names are split into "`credit`", "`Account`", and "`debit`", "`Account`". After the names are split, we find that there is a common pattern "`Account`" and the method names are grouped into one cluster. Similarly the procedure of splitting the name field is repeated for all join point candidates. The data input to be fed to the clustering tool is formed by including the split name fields and the individual join points.

In the case of join points for the class `Account`, one cluster is possible where the methods `public void creditAccount(float)` and `public void debitAccount(float)` have a common pattern `Account` in their names. The cluster assignment is shown in Figure 3. This file is in a format similar to the data input but each join point is assigned a cluster number in the end of data lines. The framework automatically performs the clustering and provides the output as a textual file. This file is in a format similar to the dataset but each data value is assigned a cluster number. The output file is then processed to group the elements belonging to a cluster together. The join points belonging to the same cluster are identified by the cluster number assigned to each join point. This clustered data is then fed as input to the inference engine.

We use the inference engine to form a pointcut given a set of clustered join point candidates. This component is based on a string alignment algorithm such as the similarity measure algorithms in Simmetrics[2], an open source similarity measure library. The clustered data input has the join points with its naming parts separated by commas. The inference engine parses each item in the data input and identifies the naming parts. Then the inference engine forms expressions for each naming part and combines the individual expressions of the naming parts to form a complete pointcut. If more than one expression can be inferred for a cluster, an optimal one is selected. The optimality is determined as the expression that suits best for all strings in the cluster as well has the maximal similarity measure based on a string distance measure. The formed expression is combined together with the designator corresponding to the join point candidates.

[2]`http://sourceforge.net/projects/simmetrics/`

```
Join points
  public void Account.creditAccount(float)
  public void Account.debitAccount(float)

Clustered Data
  @relation  Method_cluster
  @attribute Modifier       {public}
  @attribute return_type    {void  }
  @attribute Method_name    {creditAccount,debitAccount}
  @attribute Class_name     {Account}
  @attribute arguments      {(float)}
  @data
    0,public,void,Account,creditAccount,(float),cluster0
    1,public,void,Account,debitAccount,(float),cluster0

Poincut expression for clustered data
  execution(public void Account.*Account(float));
```

Figure 3: Preliminary results

3. PRELIMINARY RESULTS

We have implemented the approach and performed preliminary experiments on a few sample sets. Figure 3 shows a sample result for the example described in Section 1. It also shows the intermediate results produced by the cluster engine. Clustering can be done based on a single attribute or a set of attributes. The results show a sample clustering based on the class name, i.e., join points belonging to the class `Account` have been grouped together to belong to a single cluster. This output is similar in format to the data set except that the individual data values are assigned to a cluster. Since there is only one possible grouping of join points here, the join points are grouped to a cluster named as `cluster0`. After the data set is processed to group elements belonging to the same cluster together, the data set is fed as input to the inference engine. The expression "`public void Account.*Account(float)`" denotes the expression inferred from the join points in `cluster0`. Because the modifier, return type, class name, and argument of the methods identified as join points are similar, the expression is inferred for the strings "`creditAccount`" and "`debitAccount`", which are the method names. After the expression is formed, the respective designator is assigned to the pointcut expression. The final result gives the complete pointcut expression based on the cluster data.

4. REFERENCES

[1] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering*, pages 220–242, 2001.

[2] A. Kellens and K. Mens. A survey of aspect mining tools and techniques. Technical Report 2005-08, INGI, UCL, Belgium, 2005.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[4] R. Laddad. *Aspect Oriented Refactoring*. Addison-Wesley, September 2006.