A New Cohesion Metric and Restructuring Technique for Object Oriented Paradigm

Mehmet Kaya
Department of Computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
mkaya@syr.edu

James W. Fawcett
Department of Computer Science and Electrical Engineering
Syracuse University
Syracuse, NY, USA
jfawcett@twcny.rr.com

Abstract— When software systems grow large during development and maintenance, they may lose their quality and become complex to read, understand and maintain. A software component should be of good quality for the readers of the code to find its intents clear and the code behavior obvious. When this is the case it will be less costly to maintain the code and when its intent is clear, the code will be reusable, which is one of the key features of object oriented programming. Several software quality metrics have been proposed to measure overall or partial quality of software units such as classes or procedures. Cohesion is one of the most widely used metrics to measure quality of a software unit in terms of the relatedness of its components. This work presents a new cohesion metric based on program slicing and graph theory for units using object oriented paradigm. One can make a judgment on clarity of intent of the code using the metric we propose here. We aim to find out if a class is cohesive, handling one specific operation. We identify all program statements which constitute operations in the same abstraction domain. When a class has more than one abstraction, this technique suggests a restructuring for generating more cohesive units based on this new cohesion metric.

Keywords— Object Oriented Cohesion Metric; Code Restructuring; Extract Class; Program Slicing; Graph Theory;

I. INTRODUCTION

Software development is a continuing process which requires maintenance after release of the software product. During this maintenance phase, developers may need to add some new functionality to the product or may need to change some of the existing functionalities based on changing needs of the users and transient working environments of the product. During this phase, changes made on the source code may reduce the quality of the software and make future changes more costly. Restructuring techniques help source code regain its quality after maintenance operations or sometimes even before the first release of the product making its components more readable, reusable and more cohesive. Therefore they reduce the overall maintenance cost required for the entire system increasing overall quality of the code.

A. Basics of Program Slicing

Program slicing is one of the preferred techniques to measure the cohesion level of software units. The concept of program slicing was first proposed by Mark Weiser [9, 12, and 13]. Weiser describes program slicing as the method of automatically decomposing programs by analyzing their relationships between statements based on data and control flow. Given the criterion \( C = (s, V) \), where \( s \) is a program statement and \( V \) is a subset of variables in a program \( P \), program slicing is the process of finding all the program statements that affect value of a variable \( v \) in statement \( s \). Figure 2 shows the program slice on the example program fragment \( P \) given in Figure 1, with respect to the criterion \( C = (9, \text{sum}) \). Here the number 9 represents the statement in line 9 of program \( P \), i.e. \( \text{cout} << \text{sum} \);

```
1  int i;
2  int sum = 0;
3  int product = 1;
4  for(i = 0; i < N; ++i)
5    {
6      sum = sum + i;
7      product = product *i;
8    }
9  cout << sum;
10  cout << product;
```

Figure 1. Example Program Fragment P

B. Related Works

Program slicing has been used in procedural programming extensively since it was first proposed by Weiser. An empirical study of some slice-based cohesion metrics can be found in [5]. Slice based cohesion metrics have been proposed for measuring cohesion level of a procedure and used in various studies [1-13]. In [6], reviews on slice-based cohesion measures for procedural paradigm and object-oriented paradigm are discussed. Slice-based cohesion measures for object oriented software discussed in [6] are either an extension of functional cohesion measures [11] or use data member-method interactions for measurement [18], and they are quite different from the approach described in this paper.

Some studies using program slicing, suggest restructuring for procedures by defining low cohesive parts to be extracted from the procedure [10, 21]. In [10], this technique is used to measure the cohesiveness of each statement in a procedure and to identify parts of the procedure that cause low cohesion for restructuring purposes. In [21], cohesion level between output variables of a function is determined and a graph (pair-wise cohesion graph) is generated to visualize this relationship.

In [2], the notion of data slices is defined and used for measuring functional cohesion. Cohesion is measured based
on the percentage of data tokens that appear in more than one data slice and the ones that appear in all the data slices. In this version of slicing, data tokens are the basic units rather than statements in the program.

| 1 | int i; |
| 2 | int sum = 0; |
| 3 | for(i = 0; i < N; ++i) |
| 4 | { |
| 5 | sum = sum + i; |
| 6 | } |
| 7 | cout<< sum; |

Figure 2. Slice of P with resp. to C=(9,sum)

Some slice based data cohesion measures for object oriented designs are defined in [11], as a modification of slice based functional cohesion measures defined by Bieman and Ott [2]. In [11], private and protected member variables of a class are considered as the data tokens and data slices are determined based on them. They do not alter the definition of measuring cohesion and use exactly the same measurement technique proposed by [2].

There are some other techniques used to indicate the cohesion level of a class. The majority of them use interactions between data members and methods of the class such as data member usage or sharing of data members [14, 15, 16, 17, 18, and 19]. These existing cohesion evaluation techniques do not help us with defining program statements that constitute the same operation in the class for restructuring purposes.

C. Paper Organization

The rest of the paper is structured as follows. In section 2 we explain how we apply program slicing to object oriented classes with some new definitions of dependencies between statements. The section highlights the process of determination of slicing criteria and program slices based on the criteria. Section 3 introduces the Data-Slice-Graph (DSG) and the cohesion metrics we develop for it. A restructuring process based on the cohesion and DSG is given in section 4. Section 5 provides an application of the approach on a small explanatory source code and on a well-designed reusable class. Concluding remarks are provided in Section 6.

II. DETERMINATION OF SLICING CRITERIA AND CONSTRUCTION OF PROGRAM SLICES

In this section we describe our slicing criteria and how we approach the identification process of program slices based on those criteria.

A. Slicing Criteria in a Class

The first step in our approach is to determine the slicing criteria in the class. To identify slicing criteria and slices in a class C, we have defined the following sets:

- DM_C is the union of all private data members defined in class C.
- ST_{d,C} is the set of all program statements which use data member d in C where d \in DM_C.

Therefore each element of the set ST_{d,C} represents a slicing criterion for data member d.

It is a common practice to make all data members of a class private or protected to support encapsulation principles in Object Oriented programming. Because of this reason there should never be a public data member in a class and we ignore them while constructing our slicing criteria.

In this work, we aim to improve the structure of an existing class without changing its external behavior. To protect the behavior of the whole system and keep the client codes in the system unaffected from any changes that we make on the class to be restructured, there are two options to be considered during restructuring regarding protected data members. By default, we include in our slicing criteria only those protected data members that are never used in a class derived from the class in which they are defined. If this is the case, those protected data members can be treated just like private data members. The second option is to include all protected data members as slicing criteria. In this case, if a protected data member is moved into a newly created class, our restructuring process should support multiple-inheritance for the derived classes that use the moved data members. In this work, the classes that we have to restructure in [20] do not exhibit any inheritance; therefore we do not consider protected data members in this version of our study.

Since a private data member in a class can never be used outside of the class it is defined in, we include all the private data members in our slicing criteria.

B. Defining Relationships between Statements

In this study, we analyze the relationships between statements to construct the slices. Our primary focus is to find program statements which pertain to the operations in the same abstraction domain. For this reason we define a set of conditions for the statements to be evaluated in this manner. We say that two statements, S1 and S2, are related when one of the following conditions is true:

1. Execution of statement S1 is controlled by statement S2, or vice versa. An “if” control statement and a “for” loop statement are good examples for this case.
2. A variable defined in S1 is being used in S2
3. A variable, defined in statement S’ which uses a variable defined in S1, is being used in S2.
4. A variable defined in statement S’ is being used in both S1 and S2.
5. Invocation of a function f() which includes the statement S1 is controlled by statement S2.
6. Execution of both S1 and S2 is controlled by the statement S’.
7. A variable defined in S1 is passed to a function f as an argument and the argument is being used in statement S2 of function f.

After finding a specific slicing criterion, we take all the statements, which we think are semantically related to the criterion, into the slice rather than only the statements which affect value of a specific variable in the criterion. Figure 3
shows the slice we get from program fragment given in Figure 1 with respect to the criterion C= (9, sum) considering the dependency conditions stated above. Notice that from Figure 3 one can infer that all the statements are related in that program fragment with respect to the given criteria based on our conditions.

```
1 int i;
2 int sum = 0;
3 int product = 1;
4 for(i = 0; i < N; ++i)
5 {
6   sum = sum + i;
7   product = product *i;
8   cout<< sum;
9   cout<< product;
10 }
```

Figure 3. Slice of P with resp. to C= (9, sum)

**Condition 1, 5 and 6**

```
16 void ErrorInSize()
17 {
18   cout<<"Index out of range!\n";
19   cout<<"The Array has "<top<< "elements.\n";
20   return;
21 }
```

**Conditions 2 and 3**

```
129 if (top > 0)
130 {
131   top--;  
132   int temp_int=stk[top];
133   return temp_int;
134 }
135 else
136   {  
137     ErrorInSize();
138     return -1;
139 }
```

**Condition 4**

```
64 int w=x2.x1;
65 int b=y2.y1;
66 int a=w*h;
67 return a;
```

**Condition 7**

```
29 string temp="Class1 invoked: \"C\";"
30 temp+=t;
31 PushFunInvok(temp);
```

```
97 void PushFunInvok(std::string str)
98 {
99   if (toplevel < 100)
100   {
101     funinvoke[topInvok]=str;
102     toplevel+=1;
103   }
104   else
105     ErrorInSizeFunInvok();
106 }
```

Figure 4. Dependencies between Statements

Figure 4 demonstrates the dependency conditions listed above using some fragments of the source code of the original version of Class1 given in [20]. In condition 1, 5 and 6 of the figure, let the statements at line numbers 129, 131, 132 and 19 be represented by S’, S1, S2 and S3 respectively. S’, S1 and S2 are dependent since execution of S1 and S2 is controlled by S’ and moreover S’ and S3 are also dependent as invocation of function ErrorInSize(), which includes S3, depends on S’. In condition 2 and 3, let the statements at line numbers 64, 65, 66 and 67 be represented by S’, S1, S2 and S3 respectively, S’, S1, S2 and S3 are dependent because of their variable usage. In condition 4, let the statements at line numbers 111, 112 and 113 be represented by S’, S1 and S2 respectively. S’, S1 and S2 are dependent because a variable defined in S’ is used in both S1 and S2.

In condition 7, let the statements at line numbers 29 and 101 be represented by S1 and S2 respectively. S1 and S2 are dependent because a variable defined in S1 is being passed to function PushFunInvok(std::string str) as an argument and the argument is being used in statement S2 of the function.

C. Determination of Program Slices

To identify slices for data members defined in a class C, we have defined the following sets in addition to the sets defined in section 2.A:

- SL_{st} is the set of all program statements which directly or indirectly depend on the statement st based on the conditions listed in section 2.B. In other words, SL_{st} is the union of backward and forward slices based on the criterion of statement st.
- SL_{d} is the union of all SL_{st} where st \in ST_{d} and d \in DM_{C}.

\[ SL_{d} = U \in ST_{d} \subseteq SL_{st} \]

Therefore SL_{d} is the slice in our class C which includes all statements that directly or indirectly related to at least one of the statements which reference data member d in C.

III. DATA-SLICE-GRAPH AND A NEW COHESION METRIC

Class structure is the key unit of object oriented programming. Therefore, developers aim to design classes with high quality so that they can be reused, maintained, and tested easily. To reduce maintenance cost, these key units are expected to be simple, understandable, and readable as well.

In object-oriented programming, a class generally is designed to handle one certain operation (one abstraction). To achieve this, most classes have some data members and functions to handle some part of the operation based on the clients’ requests using some of the data members defined in the class. From this point of view we think that there is likely to be a relationship between the data members which are used to perform the intended operation of the class. If the class has more than one abstraction, there must be a group of data members involving in each abstraction domain. In other words, if there is more than one independent group of data members in the class definition, we can say that there is more than one abstraction in the class and so it is not cohesive.

In this study, we aim to formalize the idea described in the paragraph above, and suggest a restructuring to partition the class into two or more cohesive target classes. For doing so, we generate data-slice-graphs (DSG) to visualize the relationship between the data members.

In DSG, each node represents a data member of the class which may possibly need to be restructured. We have the following definitions for DSG:
DSG= (V, E) is an undirected graph such that V is the finite set of data members representing vertices in the graph and E is the finite set of relationships between data members representing edges in the graph.

|V| is the number of data members of the class.

Let v1v2 represent an edge between two nodes v1 and v2;

\[ v1v2 \in E \text{ iff } SL_{v1\in C} \cap SL_{v2\in C} \neq \emptyset \]

The description of DSG indicates that two data members, d1 and d2, are related if there is at least one program statement in the class that affects at least one occurrence of both data member d1 and data member d2 based on the dependency conditions given in section 2.B. Therefore the vertices, v1 and v2, representing data member d1 and data member d2 respectively, have an edge between them in DSG, i.e. v1v2 is in E.

We define the cohesion level of the class as the number of connected components, NC, in its DSG. The bigger NC the less cohesive our class is. Each connected component in DSG refers to one abstraction that the class holds.

IV. RESTRUCTURING THROUGH DSG

To restructure the class at hand, we use DSG and the number of connected components (NC) in DSG. Before discussing restructuring we shall explain what various values of NC mean:

- NC = 0 means there is not any data members defined in the class. That is a class that has no state - a utility class may be an example for this. Note that we do not apply this restructuring idea on this type of classes as DSG does not reveal any relationship for this kind of classes.

- NC = 1 occurs when the class has only one abstraction and when it is most cohesive. We do not restructure this kind of classes as this is the best situation a class may be in.

- NC > 1 occurs when the class has more than one abstraction. DSG reveals this by having more than one connected component and each connected component in this case represent one different abstraction the class is designed to handle. We restructure the code in this case and generate one cohesive class out of each connected component in DSG.

In DSG each connected component is a candidate to be extracted as a new smaller yet more cohesive class. In the example DSG given in Figure 5, C1 and C2 represent two different abstractions and our approach suggests that they should be extracted as new classes. Therefore data members represented by v1-v5 together with their slices are to be one class and data members represented by v6-v8 together with their slices are to be another class. We propose to generate a method in the class with each consecutive set of statements in the slice of any data member that construct the connected component of the class.

In this study, we do not want to have any dependency from the classes we generate to the original class as this will affect the reusability of the classes by having a mutual dependency with the original class. This scenario is possible if a slice is including a function call. We have defined the following cases regarding possible problems with function calls during restructuring:

- Case 1: Function call in a control block

**Definition:** Our technique always guarantees that the function definition and the function call in this case are in the same slice. The 5th dependency condition listed in section 2.B assures this. Code fragment given for this condition in Figure 4 demonstrates this case. In that example code, statements at lines 18, 19, 20 and 137 are always guaranteed to be in the same slice.

**Action:** We suggest changing that function call in the control block with a function call to the corresponding function created in the new class. That will eliminate a call-back to the original class.

- Case 2: Function call without an argument

**Definition:** Our technique always guarantees that the function call in this case will not reside in any of the slices defined by our technique as there is not any data being used in this function call.

**Action:** We do not need to do anything special for this case. This case does not cause any call back to the original class.

- Case 3: Function call with an argument

**Definition:** Our technique does not guarantee that all of the statements in the definition of the function will always be in the same slice as function call, although we think that this is a case unlikely to happen. Yet, at least some parts of the function will be in the same slice with the function call, but the action in case 1 would not solve this problem in this case. The 7th dependency condition listed in section 2.B is related to this case and code fragment given for this condition in Figure 4 demonstrates it.

**Action:** One should verify if the definition of the function consists of statements which are fully dependent based on some previously defined criteria or based on criterion of the statement which uses the argument passed to the function. If the function definition consists of fully dependent statements then this case is same as case 1. Otherwise this function call should be removed from the slice and the argument in the
function call should be replaced with a return value from the function created in the new class corresponding to this consecutive set of statements where the function call resides.

Considering all of these cases will eliminate any possible undesirable dependencies that might arise from call-backs to the original class.

To better explain the restructuring process at the statement level, some example code fragments from the Class1 shown in [20] are given in Figure 6. In Figure 6, statements between line number 124 and 128 in the code fragment before restructuring are in the same slice with respect to the criterion of C=(125, rawtime). On the other hand, statements between line number 129 and 139 in the code fragment before restructuring are in the same slice with respect to the criterion of C=(129, top). As we see in Figure 7, data member rawtime and data member top are in different connected components in the DSG of Class1, therefore those two slices mentioned above should be in different classes created for the corresponding connected components in DSG.

As we see in Figure 6, function fun2_10() in class New2 is created corresponding to the slice with respect to the criterion of C=(125, rawtime) in the code fragment before restructuring and function fun1_4() in class New1 is created corresponding to the slice with respect to the criterion of C=(129, top) in the code fragment before restructuring. Statements that construct those slices are then replaced with appropriate function calls as shown in Figure 6.

The example code fragment given in Figure 6 exhibits two of the edge cases described above: Case 1 and Case 3. At line number 137 we see a function call to the function ErrorInSize() in a control block. According to the explanation of Case 1, all the statements in the implementation of ErrorInSize() are automatically bounded to the statements in the slice where the function call resides. That means a function will be created for the statements in the implementation of ErrorInSize() and that function call can be replaced with a function call to the function created corresponding to it. For example in Figure 6, function fun1_1() in class New1 is created corresponding to ErrorInSize() and the function call to ErrorInSize() is replaced with a function call to fun1_1() in the restructured version of the code.

The other edge case, Case 3, is observed at line 128 in the code fragment before restructuring with a function call to Pushfuninvok(temp). According to the explanation of Case 3, when we analyze that function we will see that all the statements are fully dependent based on our dependency conditions and that allows us to create a new function for the statements in the implementation of PushFunInvok(temp) and replace that function call with a function call to the corresponding function created in the restructured version of the code. As we see in Figure 6, function call to PushFunInvok(temp) is replaced with a function call to fun2_8(temp), which is the corresponding new function to PushFunInvok(temp) in the restructured version of the code.

V. CASE STUDY

We now present two examples for demonstration of this new cohesion and restructuring approach. We have our initial classes as shown in [20]. Our first class is named Class1 and its corresponding DSG is given in Figure 7. This class has 9 data members and from its DSG we can see that it is not...
cohesive having three connected components. Our restructuring proposes to generate three new classes for each one of those connected components.

The second class we have analyzed is called Token and its source code is given [20]. We have used this class without any single change on its implementation in many of our projects for source code analysis. It is in fact well designed and has one certain task to accomplish: reading words from an attached file (usually source code files) or string based on some predefined rules on the word boundaries. Token class has 16 private data members and its DSG representation is given in Figure 8. From the DSG for the class Token in Figure 8, we see that there is only one connected component. When the number of connected components is 1, we say that the class is in its most cohesive form and we do not propose any restructuring for a class holding this property.

VI. CONCLUSION

In this study we have proposed a new cohesion metrics and an extract class restructuring technique for classes in object oriented environments using program slicing and graph theory. Our approach is different from other related works in a way that we try to find statements that constitutes the same abstraction in a class rather than regrouping existing components of a system. Methods of a class may have low cohesive groups of statements in their implementations [10]. Considering this fact, for our extract class restructuring technique, using a cohesion metrics defined based only on data member references in methods would not be sufficient. We define our cohesion metrics considering also the control scopes inside a method utilizing the conditions we listed in section 2.B. Rather than defining related methods in a class, we take extract class restructuring as the process of defining all the statements in a class which pertain to the same abstraction domain. We aim to create new classes with such statements. A tool support is needed for this approach to be applied to large software system and that remains as a future work of our study.

REFERENCES

[1] M. Harman, S. Daničić, B. Sivagurunathan, B. Jones, and Y. Sivagurunathan. “Cohesion metrics,” In Proceedings of the 8th International Quality Week, San Francisco CA, May 1995.
[2] J. M. Bieman, and L. M. Ott, “Measuring functional cohesion,” IEEE Trans. Softw. Eng., vol. 20, no. 8, pp. 644–657, Aug 1994.
[3] S. Karstu, “An examination of the behavior of slice based cohesion measures,” Master’s thesis, Department of Computer Science, Michigan Technological University, 1994.
[4] H. D. Longworth, “Slice based program metrics,” Master’s thesis, Michigan Technological University, 1985.
[5] T. Meyers, and D. W. Binkley, “Slice-based cohesion metrics and software intervention,” In 11th IEEE Working Conference on Reverse Engineering (WCRE 2004), pp. 256–265, 8-12 Nov 2004
[6] L. M. Ott, and J. M. Bieman, “Program slices as an abstraction for cohesion measurement,” Information and Software Technology, vol. 40, no. 11/12, pp. 691–700, 1998.
[7] L. M. Ott, and J. J. Thuss, “The relationship between slices and module cohesion,” In Proceedings of the 11th ACM conference on Software Engineering, pp.198–204, 1989.
[8] L. M. Ott, and J. J. Thuss, “Slice based metrics for estimating cohesion,” In Proceedings of the IEEE-CS International Software Metrics Symposium, pp. 71–81, 21-22 May 1993.
[9] M. Weiser, “Program slices: formal, psychological, and practical investigations of an automatic program abstraction method,” PhD thesis, University of Michigan, Ann Arbor, 1979.
[10] J. Krinke, “Statement-Level Cohesion Metrics and their Visualization,” Seventh IEEE Int. Working conference on Source Code Analysis and Manipulation, pp. 37-48, Sept. 30 2007-Oct. 1 2007
[11] B. Gupta, “A critique of cohesion measures in the object oriented paradigm,” Master’s thesis, Department of Computer Science, Michigan Technological University, 1997.
[12] M. Weiser, “Program Slicing,” IEEE Transactions on Software Engineering, vol. 10, no. 4, pp. 352-357, July 1984
[13] M. Weiser, “Program Slicing,” Proceedings of the 5th International Conference on Software Engineering, pp. 439-449, 1981
[14] S. R. Chidamber, and C. F. Kemerer, “Towards a Metrics Suite for Object-Oriented Design,” Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices, Vol. 26, No. 10, pp. 197-211, October 1991.
[15] S. R. Chidamber, and C. F. Kemerer, “A Metrics suite for object Oriented Design,” IEEE Transactions on Software Engineering, Vol. 20, No. 6, pp. 476-493, June 1994.
[16] W. Li, and S. Henry, “Object oriented metrics that predict maintainability,” Journal of Systems and Software, vol. 23, no. 2, pp. 111-122, February 1993.
[17] M. Hitz, and B. Montazeri, “Measuring coupling and cohesion in object oriented systems,” Proceedings of the Int. Symposium on Applied Corporate Computing, October 1995.
[18] J. M. Bieman and B.-K. Kang, “Cohesion and Reuse in an Object-Oriented System,” Proc. Symp. Software Reusability, pp. 259-262, 1995.
[19] H. S. Chae, Y. R. Kwon, and D. H. Bae, “Improving Cohesion Metrics for Classes by Considering Dependent Instance Variables,” IEEE Transaction on Software Engineering, vol. 30, no. 11, pp. 826-832, November 2004.
[20] All the example codes and detailed restructuring process is posted on “http://www.ecs.syr.edu/faculty/fawcett/handouts/research/kaya/files/appendix.pdf”
[21] A. Lakhota, and J-C. Deprez, “Restructuring Functions with Low Cohesion,” Proc. Sixth Working Conf. Reverse Eng. (WCRE), pp. 36-46, 6-8 Oct 1999.