Program Execution Based Module Cohesion Measurement

Neelam Gupta
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Praveen Rao
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721
rpraveen@cs.arizona.edu

Abstract

Module cohesion describes the degree to which different actions performed by a module contribute towards a unified function. High module cohesion is a desirable property of a program. The program modifications during successive maintenance interventions can have negative effect on the structure of the program resulting in less cohesive modules. Therefore, metrics that measure module cohesion are important for software restructuring during maintenance. The existing static slice based module cohesion metrics significantly overestimate cohesion due to the limitations of static slicing.

In this paper, we present a novel program execution based approach to measure module cohesion of legacy software. We define cohesion metrics based on definition-use *pairs in the dynamic slices of the outputs. Our approach significantly improves the accuracy of cohesion measurement. We implemented our technique and measured module cohesion for several programs. Cohesion measurements using our technique were found to be more insightful than static slice based measurements.*

Keywords - *software maintenance, software restructuring, module cohesion, software metrics.*

1 Introduction

Module cohesion measures the degree to which all its internal processing elements (statements) contribute towards computing a unified function. The payoff of highly cohesive modules is higher reliability and understandability of the code. In an empirical study of 148,000 source line system from a production environment, data binding and clustering analysis techniques [26] were applied to the error data collected from system design to system field test operation, in order to characterize the error prone system structure. It was found that routines with highest coupling to cohesion ratios had 7 times as many errors (per 1000 source statements) as those with the lowest coupling to cohesion ratios [26] and were 20 times as costly to fix.

As changes are made to software over time, the entropy of the system increases [10] and the software becomes more difficult to understand and maintain. Lehman and Belady [7, 19] have studied the history of successive releases in a large operating system. They find that the total number of modules increases linearly with the release number, but the number of modules affected by the release increases exponentially with release number. Even if the software was originally designed with cohesive modules, program changes made during its maintenance can introduce auxiliary functionality to existing program units resulting in less cohesive modules. Thus, periodic restructuring and refactoring of existing software is important to increase module cohesion during software maintenance [10, 24]. In order to restructure existing software to increase its module cohesion, we need techniques and metrics that accurately measure module cohesion.

There are two main components to functional cohesion measurements. First, we must decide on a model to identify the relatedness between the operations performed by the elements of the program. Second, we must define cohesion metrics that capture the above relatedness between the operations performed in the function. In [21], Ott and Thuss propose a static slice [31, 30, 14] based model to measure module cohesion. In [6, 23], Ott and Beimann develop cohesion metrics based on static slices for outputs computed by a function. They compute backward static slice from each final use node for each output variable and compute forward static slice of the output variable from the top of the backward slice. They refer to the union of the forward and backward slices of the output as its *data slice* [23] and define cohesion metrics by counting *data tokens in the common statements* in data slices of the

0-7695-1426-X/01 $17.00 © 2001 IEEE

outputs. Kang and Beiman [16] use these metrics in restructuring of software through a series of decomposition and composition operations.

```
function SUMorPROD(flag,n,A[]);
1:   if (flag==0)
2:        result = 0;
3:   else
4:        result = 1;
5:   endif
6:   for (i=0; i<n; i++)
7:        if (flag==0)
8:             result += A[i];
9:        else
10:            result *= A[i];
11:       endif
12:  endfor
13:  return(result);
endfunction
```

output	Static Slice
result	{1,2,4,6,7,8,10}

output	flag	Dynamic Slice
result	0	{1,2,6,7,8}
result	1	{1,4,6,7,10}

Figure 1. Treatment of individual outputs.

Although the above research illustrates that module cohesion can be measured quantitatively and used in code restructuring during software maintenance, there are some drawbacks of the static slice based approaches. In these approaches, procedures with a single output variable are always considered to have maximum cohesion. For example, the static slice based approaches will conclude that the program in Figure 1 has maximum cohesion since it has only one output variable called **result**. In fact, this program computes two distinct functions even though it returns a single output. Depending upon the value of *flag*, it either computes **sum** of n numbers or their **product**. Thus, a function with a single output variable need not always have maximum cohesion. As more and more flaws are fixed in successive releases of software, very often the output variables of the program are reused in mutually exclusive semantic contexts to fix bugs. In these situations, the existing static slice based approaches will incorrectly conclude that the module has maximum cohesion. In addition, the existing approaches [6, 23] significantly overestimate module cohesion due to limitations of static slicing. Also, there are some inadequacies associated with defining cohesion metrics by counting data tokens in the common statements in the data slices of outputs. Next we describe these limitations and motivate our approach for overcoming them.

Limitations of static slice based model for cohesion measurement. The cohesion measures based on static slices overestimate module cohesion because of common statements in the static slices that do not contribute to module cohesion, but are present in the slices due to inherent imprecision of static slices. As an example, consider the program in Figure 2. This program computes the sum of a series and returns the real and imaginary parts of the summation in outputs **res1** and **res2** respectively. The only statements common to computation of the two outputs are *while* and *if* control statements. However, as shown in Figure 2, the static data slices of **res1** and **res2** also have statements 9 and 13 in common due to overestimation in computation of static slices. Therefore, cohesion measurements using these static slices also overestimate module cohesion.

Cohesion metrics. There are some inadequacies in the cohesion measurement by the existing cohesion metrics. As mentioned before, the existing approaches [6, 23] construct the union of forward and backward slice, called *data slice*, for each output variable and count the data tokens in the common statements in the data slices for all the output variables. If a statement is in the forward slice of all the output variables but not in backward slice of any of the output variables, it should not contribute towards cohesion since none of the outputs depend on it. If a statement is in the forward slice of an output variable and in the backward slice of another output variable, then counting *all* the data tokens in this statement for measuring cohesion is not correct since there can be data tokens in this statement on which the value of only one of the output variables depends. Therefore, counting all the tokens in this statement gives an overestimation of cohesion.

Another inadequacy in the definition of existing metrics is the assumption that data tokens in only *common statements* in the data slices of outputs contribute to module cohesion. We illustrate this with the program in Figure 2. In this program, the statement 16 is in the static slice for output **res1** and statement 17 is in the static slice for output **res2**. They are different statements, so the existing approaches will assume that data tokens in these statements do not contribute to cohesion between outputs **res1** and **res2**. But, these different statements use the common definition of *ival* in statement 3. Therefore, computation of both the outputs are related by this *use* of common *definition* in statements 16 and 17. This should contribute to increased cohesion between the two outputs (i.e., the functional cohesion of the outputs should increase with the use of identical definitions). But in the existing approach, the use of a common definition in statements 16 and 17 will not have any impact on the cohesion measurement. In fact, existing metrics would obtain same cohesion measure irrespective of whether *ival* was used in statements 16 and 17 or not. Therefore, cohesion measures defined on the data tokens in the com-

```
function CompSeries(float val)
1:   res1=0;
2:   res2=0;
3:   ival=round(val);
4:   i=ival;
5    while (i ≠ 0) do
6:       if val > 0 then
7:           temp1=sqroot(i);
8:           res1=res1 + cube(temp1);
9:           i=i-1;
10:      else
11:          temp2=sqroot(-1.0*i);
12:          res2=res2 - cube(temp2);
13:          i=i+1;
14:      endif
15:  endwhile
16:  res1=ival*res1;
17:  res2=ival*res2;
18:  return(res1,res2);
```

```
function CompSeries(float val)
1:   res1=0;
3:   ival=round(val);
4:   i=ival;
5    while (i ≠ 0) do
6:       if val > 0 then
7:           temp1=sqroot(i);
8:           res1=res1 + cube(temp1);
9:           i=i-1;
10:      else
13:          i=i+1;
14:      endif
15:  endwhile
16:  res1=ival*res1;
18:  return(res1,res2);
```

Data Slice of res1

```
function CompSeries(float val)
2:   res2=0;
3:   ival=round(val);
4:   i=ival;
5    while (i ≠ 0) do
6:       if val > 0 then
9:           i=i-1;
10:      else
11:          temp2=sqroot(-1.0*i);
12:          res2=res2 - cube(temp2);
13:          i=i+1;
14:      endif
15:  endwhile
17:  res2=ival*res2;
18:  return(res1,res2);
```

Data Slice of res2

Figure 2. A program to compute Series Summation: $x * x^{3/2} + x * (x-1)^{3/2} + ... + x$, for $x > 0$, and $x * x^{3/2} + x * (x+1)^{3/2} + ... + x * (-1)^{3/2}$, for $x < 0$.

mon statements in the data slices are not adequate in measuring the module cohesion. The inadequacies of the existing static slice based metrics suggest that finer grained metrics are required for accurate measurement of module cohesion.

In this paper, we propose a novel approach to measure module cohesion of existing or legacy software. Our approach addresses the drawbacks in both the key components of cohesion measurement by considering the program's dynamic behavior and designing metrics based upon fine grained activities observed during program execution. We use dynamic slices of outputs for measuring cohesion. This overcomes the limitations of overestimation resulting from static slices. For example, the dynamic slices of the output variable **result** in the program in Figure 1, fall in two distinct categories, revealing that the single output variable is actually performing one of the two distinct functions in any given execution.

Our intuition for defining cohesion metrics is based on the fundamental observation that each output of a function is computed by executing a sequence of program statements. Each of these statements *uses* some data (*definitions*), generated by earlier statements (or constants), and performs some *operations* on these definitions to compute new definitions. Therefore, intuitively, the degree to which different outputs of a function are related to each other, is determined by the commonality between the *definitions* used and the *operations* performed for computing these outputs. This forms the basis for our approach to define module cohesion metrics. We consider common *definitions*, common *uses* and common *definition-use pairs*, belonging to the *dynamic slices* of different program outputs, to define module cohesion metrics. We have implemented our techniques and our experiments show that our metrics provide more precise measurement of module cohesion. The important contributions of this paper are:

- Our approach uses dynamic program behavior to identify situations in which a single program output variable is being used to perform multiple functions. Therefore, the information generated by our approach can be used in restructuring of programs that cannot be handled by existing approaches.
- We define new module cohesion metrics based upon common definitions, common uses, and common definition-use pairs in dynamic slices which enable more accurate cohesion measurement than existing metrics.
- By the judicious use of dynamic slices, our approach guarantees that the cohesion measures obtained by our technique are always at least as precise or more precise than the estimates obtained using static slicing.
- Finally, our experiments demonstrate that the cohesion measurements obtained using *definition-use* pairs on dynamic slices are more insightful than those obtained by counting tokens on common statements in static slices.

The remainder of this paper is organized as follows. In section 2, we describe our program execution based approach for measuring module cohesion and define the dynamic cohesion metrics. In section 3, we present results of our experiments and briefly illustrate how the

computed measures can be used to perform program restructuring that would not have been possible using cohesion measurements based upon static slices. We discuss the related work in section 4 and present the conclusions of our work in section 5.

2 Our Approach

As explained in the previous section, our program execution based approach uses the *definition-use* pairs on dynamic slices of outputs to measure functional cohesion. The use of dynamic slices overcomes the limitations of overestimation in static slices. The use of *definition-use* pairs enables us to measure functional cohesion more accurately by including all variable references that contribute towards functional cohesion. In this framework, we define module cohesion as follows:

Definition: **Module Cohesion** is the degree of overlap among the *definition-use* pairs exercised by the dynamic slices for different outputs of the module (function) .

Note that the above definition is general in its applicability. If the *outputs* in the above definition denote different output variables of the module then it measures cohesion between these output variables of the module. On the other hand, if the *outputs* in the above definition denote output values of a single output variable, then it measures module cohesion with respect to that output variable.

The set of *definition-use* pairs (called *def-use* pairs here onwards) exercised for each output can be recorded during various executions of the function and the extent of overlap among these sets can be measured quantitatively. Higher the overlap between the sets of *def-use* pairs used in computation of function outputs, more cohesive is the function. Our module cohesion metrics are defined to measure this degree of overlap between the function outputs. Now we explain our approach for cohesion measurement in detail.

Since our approach is based on program execution, we need a set of *representative inputs* for the program module (function) that provides *adequate coverage* for cohesion measurement. As our techniques are to be used during the maintenance of existing software, it is reasonable to assume that a test suite for some form of *def-use* coverage criterion is available or can be generated. At the end of this section, we show how we augment the available test suite of the function to obtain a set of representative inputs that provides adequate coverage for functional cohesion measurement using our approach. Our algorithm consists of three main steps.

- First we execute the function using representative inputs and obtain a set of *unique dynamic slices* for each of the function outputs. These dynamic slices are represented as an enumeration of the *def-use* pairs that are contained in each dynamic slice.
- Second we analyze the above information to identify commonality between *def-use* pairs in the dynamic slices.
- Third, using the information collected in the above analysis, we estimate cohesion using our newly developed metrics in two ways: (i) we compare the *def-use* pairs in dynamic slices of different output variables to measure module cohesion with respect to output variables; and (ii) we compare the *def-use* pairs in the dynamic slices of a given output variable to determine if the output variable performs a single function.

2.1 Construction of Unique Dynamic Slices

Since our cohesion measurements are based upon *def-use* pairs on dynamic slices of outputs, we execute the function on the representative inputs and collect information about *def-use* pairs exercised during each execution. While traditionally a dynamic slice [15, 1] consists of a set of statements, *our representation of a dynamic slice consists of the set of def-use pairs that resulted in inclusion of those statements in the traditional dynamic slice.* Note that we do not need to store the complete execution trace in order to generate the dynamic slices in terms of *def-use* pairs that are exercised during the execution. The programs are instrumented during the compilation phase (see [2, 3]), to collect the set of *def-use* pairs that are exercised. We represent a *def-use* pair by $v(dst, ust)$, where dst the statement number that generates the $definition$ of variable v and ust is the statement number that $uses$ this definition of variable v. The *def-use* pairs that are exercised repeatedly during multiple iterations of a loop are recorded only once, since for cohesion measurement, we are only interested in finding out whether a *def-use* pair is exercised during execution or not. Therefore, our dynamic slice representations consist of only unique *def-use* pairs exercised during execution. If there are at most two $uses$ per statement and there are n statements in the program, then the number of *def-use* pairs in a dynamic slice computed by our method are bounded by n^3. Therefore, our dynamic slice representation is compact. The cost of computing dynamic slices in our representation is bounded by a polynomial function of number of statements in the program.

```
function CompSeries(float val)
1:   res1=0;
3:   ival=round(val);
4:   i=ival;
5    while (i ≠ 0) do
6:       if val > 0 then
7:           temp1=sqroot(i);
8:           res1=res1 + cube(temp1);
9:           i=i-1;
10:      else
14:      endif
15:  endwhile
16:  res1=ival*res1;
18:  return(res1,res2);
```
Traditional Dynamic Slice (a)
(output=$res1$, input=9.5)

```
function CompSeries(float val)
2:   res2=0;
3:   ival=round(val);
17:  res2=ival*res2;
18:  return(res1,res2);
```
Traditional Dynamic Slice (b)
(output=$res2$, input=9.5)

```
function CompSeries(float val)
1:   res1=0;
3:   ival=round(val);
16:  res1=ival*res1;
18:  return(res1,res2);
```
Traditional Dynamic Slice (c)
(output=$res1$, input=-12.1)

```
function CompSeries(float val)
2:   res2=0;
3:   ival=round(val);
4:   i=ival;
5    while (i ≠ 0) do
6:       if val > 0 then
10:      else
11:          temp2=sqroot(-1.0*i);
12:          res2=res2 - cube(temp2);
13:          i=i+1;
14:      endif
15:  endwhile
17:  res2=ival*res2;
18:  return(res1,res2);
```
Traditional Dynamic Slice (d)
(output=$res2$, input=-12.1)

Figure 3. The set of traditional dynamic slices of outputs $res1$ and $res2$ of program in Figure 2.

By executing the given function for all the representative inputs, we construct a set of dynamic slices for each output variable. Some of these dynamic slices for an output variable may be identical. We eliminate such superfluous slices from consideration. More precisely, for each of the function's output variable v, we construct an ordered set of *Unique Dynamic Slices*, $UDS(v)$, such that each dynamic slice added to $UDS(v)$ contains at least one *def-use* pair that is not contained in dynamic slices already in $UDS(v)$. In order to compute the $UDS(v)$ set, every time a function is executed on a different input, the newly generated dynamic slice is compared with previously generated dynamic slices in the set to decide whether to retain it or discard it. Note that the set $UDS(v)$ is not unique and the resulting set depends upon the order in which the dynamic slices are generated and processed. We illustrate the above with the example program in Figure 2. The traditional dynamic slices for outputs $res1$ and $res2$, obtained by executing the program in Figure 2 on two inputs (val=9.5 and val=-12.1), are shown in Figure 3. We construct the sets $UDS(res1)$ and $UDS(res2)$ as explained above.

UDS(res1)={{def-use pairs of dynamic slice(a)}, {def-use pairs of dynamic slice(c)}}.
UDS(res2)={{def-use pairs of dynamic slice(b)}, {def-use pairs of dynamic slice(d)}}.

Note that although all the statements in traditional dynamic slice(c) are contained in traditional dynamic slice(a), the two slices are treated as distinct dynamic slices by our approach because dynamic slice(c) contains a def-use pair $res1(1, 16)$ which is not contained in dynamic slice(a). The def-use pair $res1(1, 16)$ is contained in dynamic slice(c) since definition of $res1$ in statement 1 is used in the statement 16 when the program is executed with input val= -12.1, which is the input used for dynamic slice(c). However def-use pair $res1(1, 16)$ is not contained in dynamic slice(a) since the definition of $res1$ in statement 1 is used in statement 8 (and not in statement 16) when the program is executed with input used for dynamic slice(c).

2.2 Analysis of Dynamic Slices to identify commonality among them.

As mentioned earlier, our cohesion metrics are based upon the commonality in the *def-use* pairs belonging to dynamic slices of different output variables. We observe that this commonality can arise in many ways. In particular, the cohesion between two outputs may result from the following:

TypeI *def-use* pair – a common *use* of a common *definition*: For example, let us consider the different types of *def-use* pairs on dynamic slice(a) and slice(d) in Figure 3. The *def-use* pair $ival(3, 4)$ (i.e., the *definition* of *ival* in statement 3 and *use* of *ival* in statement 4) forms a TypeI *def-use* pair because the *definition* as well as the *use* are common to the slices of both the outputs.

TypeII *def-use* pair set – different *uses* of a common *definition*: For dynamic slices (a) and (d) in Figure 3, the set of *def-use* pairs { $ival(3,16)$, $ival(3,17)$ } form a TypeII *def-use* pair set.

TypeIII *def-use* pair set – a common *use* of different *definitions*: For dynamic slices (a) and (d) in Figure 3, the set of *def-use* pairs {i(9,5), i(13,5)} form a TypeIII *def-use pair* set.

Two observations can be made regarding the commonality reflected in def-use pairs as characterized above. First, two outputs become increasingly cohesive as the commonality in their def-use pairs increases. Second the presence of TypeI def-use pairs results in a greater degree of cohesion than TypeII or TypeIII def-use pairs. This is because if there is a TypeI def-use pair on the dynamic slices of two outputs, then identical definition is used to perform an identical computation for the two outputs. On the other hand, in case of TypeII def-use pair set, two outputs only share a definition (i.e., use the data computed by same definition) but use it in different computations. Similarly, in case of TypeIII def-use pair set, the pairs perform identical computation but they use data from different definitions on which they perform the computation. Thus, the contribution of TypeI def-use pair towards cohesion is twice (common *definition* and common use) the contribution of TypeII (common *definition*) or TypeIII (common *use*) definition use pair set. Therefore, in defining our functional cohesion metrics, we assign a weight of 2 to a TypeI def-use pair and weight 1 to a TypeII or a TypeIII def-use pair set.

The use of *def-use* pairs on the dynamic slices provides the finer granularity required to accurately measure the functional cohesion as compared to cohesion measurement by counting the data tokens in common statements in the data slices of the outputs. In addition, the finer granularity provided by *def-use* pairs allows us to accurately measure functional cohesion by considering only the *backward dynamic slices* for the outputs. Next we define our functional cohesion metrics based on different types of *def-use* pairs in the backward dynamic slices of the module outputs.

2.3 Cohesion Metrics

Cohesion between different output variables of a module: The first set of metrics measure the cohesion between different output variables of the function. We define Strong Functional Cohesion (SFC) as that arising out of *def-use* pairs of each type common to the dynamic slices of all the outputs variables.

$$\mathbf{SFC} = \frac{2*\mid \mathbf{TypeI^{all}} \mid + \mid \mathbf{TypeII^{all}} \mid + \mid \mathbf{TypeIII^{all}} \mid}{2*(\textbf{total \# of def-use pairs in the module})}$$

where, $\mathbf{Typei^{all}}$ = set of all *def-use* pairs of *Typei* for i=I,II,III, such that the *def-use* pair is present on at least one dynamic slice of *each* of the output variables of the function.

Similarly, we define the weak functional cohesion to be that arising out of *def-use* pairs of each type that are found in dynamic slices of two or more output variables. As explained above, we use the weighted summation to take into account how strongly each *def-use* binds the output variables of the module.

$$\mathbf{WFC} = \frac{2*\mid \mathbf{TypeI^{\geq 2}} \mid + \mid \mathbf{TypeII^{\geq 2}} \mid + \mid \mathbf{TypeIII^{\geq 2}} \mid}{2*(\textbf{total \# of def-use pairs in the module})}$$

where, $\mathbf{Typei^{\geq 2}}$ = set of all *def-use* pairs of *Typei* for i=I,II,III, such that the *def-use* pair is present on at least one dynamic slice of *2 or more* output variables of the function.

The measurements of functional cohesion between the two output variables of the *CompSeries* program in Figure 2 are given in Table 1. The values in column labeled *DC* list the cohesion measures obtained using our metrics, whereas the values in the column labeled *SC* list the cohesion measures obtained using tokens in common statements in data slices of outputs. All the values are in the scale of 0-1, where 1 indicates maximum module cohesion and 0 indicates no cohesion. When input *val* is positive, real part of summation is computed and zero value is returned for the imaginary part of the summation. When input *val* is negative, imaginary part of summation is computed and the zero value is returned for the real part of summation. Cohesion between the two output variables results only from the common *while* control statement. That is why the dynamic cohesion is much lower than static cohesion. The static cohesion is higher due to overestimation of cohesion by false overlap of statements in the static slices of outputs. The values for SFC and WFC are same since there are only two output variables of the function.

Program	# of	SFC		WFC	
	outputs	SC.	DC.	SC	DC
CompSeries	2	0.394	0.2692	0.394	0.2692

Table 1. Cohesion between output variables of CompSeries program in Figure 2.

Module cohesion with respect to each output variable: Our approach also allows us to compute functional cohesion of a module with respect to each

individual output variable. Thus using our approach, functional cohesion of module that has only one output variable can also be computed. The cohesion measures similar to those described in the previous section can be computed for a function with a single output variable v by considering the *def-use* pairs of each type that belong to all the dynamic slices in the set $UDS(v)$. Thus, Strong Functional Cohesion for a function with a single output variable v is:

$$\mathbf{SFC(v)} = \frac{\mathbf{2 * \mid TypeI^{all} \mid + \mid TypeII^{all} \mid + \mid TypeIII^{all} \mid}}{\mathbf{2 * (total\ \#\ of\ def\text{-}use\ pairs\ in\ the\ module)}}$$

where, $\mathbf{Typei^{all}}$ = set of all *def-use* pairs of *Typei* for i=I,II,III, such that the *def-use* pair is present on *each* dynamic slice in the set $UDS(v)$ of *unique dynamic slices* of the output variable v.

Similarly, Weak Functional Cohesion can be defined by considering *def-use* pairs of each type that are common to two or more dynamic slices in the set of *unique dynamic slices* for the output variable.

The definitions of functional cohesion measures defined in this section can be used to measure the cohesion of a function with respect to each of the outputs variables of the function. In Table 2, we show the comparison between the static and dynamic module cohesion measurements with respect to each output variable of the *CompSeries* program in Figure 2. As the dynamic cohesion measures illustrate, the function has low cohesion with respect to each of the output variables. Hence, each of the output variable is computing multiple functions, contrary to the conclusion derived from static cohesion measures. Upon analyzing this program, we note that for positive values of val, the output variable $res1$ computes its respective series, but for other values of val, the module always assigns a constant zero value to its output variable res1. Same is true for the other output variable. This suggests that the program can be restructured using subroutines to improve its readability. In contrast, static slice based module cohesion measurement techniques cannot handle restructuring of this program. Note that module cohesion with respect to output variable $res2$ is slightly lower than that of $res1$ because multiplication with additional constant (-1) in statement number 11.

Program	Output	SFC		WFC	
		SC	DC	SC	DC
CompSeries	res1	1	.222	1	.222
	res2	1	.212	1	.212

Table 2. Module cohesion for each output of CompSeries in Figure 2.

2.4 Construction of a representative set of inputs

As mentioned before, we need to have a representative set of inputs that can give adequate coverage for dynamic functional cohesion measurement. We construct this set of inputs in the following manner. We assume that a test suite for some form of def-use structural coverage criteria exists or can be generated using existing techniques [25, 4, 18, 5, 32, 33]. We first initialize the set of representative inputs by this test suite and execute the function with the inputs in this test suite. We compute the dynamic slices for each output in terms of *def-use* pairs as explained before. If there is a *def-use* pair in the static slice of an output variable that is not exercised by any of the inputs, we try to generate an input for the *def-use* pair by considering a path that exercises the *def-use* pair. We use our iterative test data generation technique [12, 13] to generate test data for the selected path. If we are able to generate an input for the path, we add this input to the current set of representative inputs otherwise we pick next path through the *def-use* pair. If all the paths that exercise the *def-use* pair are detected infeasible, then we ignore this *def-use* pair, otherwise we add this *def-use* pair to the dynamic slice of the output variable. Since detection of infeasible paths and hence infeasible *def-use* pairs is undecidable in general, we add the *def-use* pair to the dynamic slice of the variable in the latter case to avoid underestimation of cohesion by dynamic slices. This guarantees that our approach never underestimates functional cohesion.

3 Experimental Results

Implementation. We have implemented our cohesion measurement techniques. Our implementation (in C++) supports only C programs as input with a limited set of constructs like scalars, arrays, expressions, while, if-then-else and function calls. We modified the SUIF 2 compiler system [28] on linux platform to generate instrumented source code of the given function [2]. This instrumented function generates the dynamic slices in the form of the *def-use* pairs exercised for computing each of the output variables during execution of the function. We execute the instrumented program for a set of representative inputs as explained in previous section and compute the set of *unique dynamic slices* in terms of *def-use* pairs exercised for each output. We then compute TypeI, TypeII and TypeIII def-use pairs using the dynamic slices and compute the cohesion metrics developed in earlier sections. We also implemented cohesion measurement using static slicing and counting data tokens in common statements as in [6] for comparison purposes. Both static and

dynamic slices are intraprocedural slices. We handle function calls as variable references while computing intraprocedural slices since we are interested in computing functional cohesion of a single program module (procedure). The following programs were used in our experiments.

VarStd - This program computes the *variance* and *standard deviation* of n *input* data values. The two outputs share only the computation of the average of n numbers. This program was taken from an undergraduate class project.

StatSig - Given a set of ordered pairs (x, y), the program establishes whether the change in y as x changes is statistically significant or not. It has 3 output variables: $sigma2$, t and p. $sigma2$ is used in the mathematical formula used to compute t. The absolute value of t decides whether variation in y is statistically significant or not, and the value of p is set to indicate this. This program is from the project on Landcover analysis at Dept of Atmospheric Physics at Univ. of Arizona.

Sin2Cos2 - This function takes *theta* (in radians) as the input and computes $sin^2(theta) - cos^2(theta)$ as the output. It is written to optimize the computation for certain inputs. When *theta* is zero it returns value -1. For small values of *theta*, $sin(theta)$ is approximated by *theta*. For all other cases, $sin(theta)$ is computed by using the series $x - x^3/3! + ...$ Therefore the output is computed differently for different classes of inputs. This program was written for numerical computation class.

SumSq - This program computes sum or sum of squares and product from 1 to n. It is taken from http://web.cacs.usl.edu/~arun/Wolf/demo/classic/module9/sum_sumsquares_product/LOGICAL.html

Results. We conducted experiments to measure the SFC and WFC of above programs using our dynamic cohesion metrics as well as existing [6] static slice based cohesion metrics as shown in Table 3. For the first program ($VarStd$), the dynamic cohesion using our technique is higher than static cohesion measurement because of some TypeII *def-use* pair sets that contribute to cohesion but are in different statements in the data slices. Hence they are not accounted for in the static cohesion. Thus in this case, the static cohesion metrics do not adequately measure cohesion.

The program *StatSig* has an infeasible path and therefore the *def-use* pairs on this path are never executed. The static cohesion measures are overestimates in this case because of common statements in data slices of all three outputs corresponding to this infeasible path. The program $Sin2Cos2$ has only one output. Therefore, the static cohesion measures conclude that the function has maximum cohesion, whereas the dynamic cohesion measures illustrate that the function indeed has very low cohesion with respect to its only output variable. This indicates that grouping computations for different cases into subroutines may increase the cohesion and readability of the code.The function $SumSq$ has comparable static and dynamic cohesion, although the dynamic cohesion measure is more accurate than the static cohesion measure.

Program	#of	SFC		WFC	
	outputs	SC	DC	SC	DC
VarStd	2	0.11	0.13	0.11	0.13
StatSig	3	0.77	0.60	0.86	0.69
Sin2Cos2	1	1	0.04	1	0.06
SumSq	3	0.11	0.092	0.30	0.31

Table 3. Functional cohesion measures with respect to all output variables.

Program	Outputs	SFC		WFC	
		SC	DC	SC	DC
VarStd	Variance	1	1	1	1
	StdDev	1	1	1	1
StatSig	sigma2	1	0.2	1	0.2
	t	1	0.19	1	0.19
	p	1	0.31	1	0.76
Sin2Cos2	result	1	0.04	1	0.06
SumSq	sum	1	1	1	1
	product	1	1	1	1
	sumproduct	1	1	1	1

Table 4. Functional cohesion measures with respect to individual output variables.

outputs	SFC		WFC	
	SC	DC	SC	DC
sum	1	1	1	1
product	1	1	1	1
sumsquares	1	1	1	1
sum, sumsquares	0.13	0	0.13	0
sum, product	0.13	0	0.13	0
product, sumsquares	0.47	.47	0.47	.47
all	0.11	0.09	0.30	0.31

Table 5. Cohesion Measures for SumSq

We also measured the module cohesion with respect to individual output variables shown in Table 4. These dynamic cohesion measures in Table 4 provide insight into how cohesive the function is with respect to each output variable which cannot be obtained from static cohesion measurements. We illustrate the use of these

cohesion measurements in code restructuring of program *SumSq* by computing its cohesion measurements for all outputs, individual outputs and also pairs of outputs shown in Table 5. Each output variable of this program has only one dynamic slice, so the dynamic functional cohesion as well as static functional cohesion with respect to each output is 1. But the dynamic cohesion between *sum* and *product* is zero and the dynamic cohesion between *sum* and *sumsquares* is zero, whereas the dynamic cohesion between *sumsquares* and *product* is high. So, we conclude that computation for *sum* should be separated from the computation of *sumsquares* and *product* to make the module more cohesive. In contrast, the static cohesion measurements for the same cases do not reveal the complete lack of cohesion in this module.

4 Related Work

The idea of cohesion was introduced in [27] by Stevens, Meyers, and Constantine. The concept of cohesion has evolved since then and [34] presents various levels of module cohesion. Emerson defined a cohesion metric [9] that measures module cohesion by seeing how many independent paths of the module go through different statements in a module. Longworth [20] was first to hypothesize that some of the static slice based metrics suggested by Weiser in [30, 31] may be used as indicators of cohesion. The uses of program slicing in software maintenance has been discussed in [11, 8, 17]. In [21, 22], Ott and Thuss defined *metric slices* to improve the static slice based metrics suggested in [20]. They proposed to use metric slices for module cohesion measurement. The metric slices were further refined in [6] to *data slices* by using data tokens rather than statements as the basic unit. In this paper, we have further refined slice based cohesion metrics by considering *def-use* pairs on the dynamic slices of outputs. As illustrated in the paper, counting data tokens in the *common statements* in the data slices of the outputs does not take into account cohesion between the outputs resulting from multiple references of same definition in different statements. Using common *def-use* pairs for measuring cohesion overcomes this limitation. Also, since our approach is based on program execution, it can handle efficiently handle arrays and pointers. Static slice based approaches cannot handle arrays and pointers efficiently. Using dynamic slices instead of static slices overcomes the overestimation of cohesion due to limitations of static slicing. Thus, our cohesion metrics based on *def-use* pairs on dynamic slices measure cohesion more accurately than existing cohesion measurement metrics.

Restructuring of existing software is a form of preventive maintenance that is often necessary when the system undergoes new releases. In [16], Kang and Beiman illustrate the use of cohesion measurement in software restructuring. Recently, a new approach based on concept analysis [29] has also been proposed for module restructuring. Our program execution based approach and cohesion metrics developed in this paper can make significant contributions in guiding restructuring of legacy software.

5 Conclusions

In this paper, we have presented a program execution based approach to define metrics for measuring module cohesion. Our metrics are based on common *def-use* pairs on the dynamic slices of outputs. Our approach also enables measurement of functional cohesion with respect to a single output variable. Using *def-use* pairs on dynamic slices, we obtain more accurate measurement of functional cohesion compared with existing techniques. It is also helpful in guiding code restructuring in some cases that cannot be handled by existing techniques.

Acknowledgments

We would like to acknowledge Kausik Sinnaswamy, Maqsood Mohammed and Mohammad Hossain for their contributions in implementation of our tool. We also thank Prof. Xubin Zeng, Department of Atmospheric Physics, Univ. of Arizona, for his help in one of the programs for our experiments.

References

[1] Agrawal H. and Joseph R. Horgan, "Dynamic Program Slicing", in the *Proceedings of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June, 1990.

[2] Ball T. and James R. Larus, "Efficient Path Profiling", *MICRO-29* , December 1996.

[3] Ball T. and James R. Larus, "Using Paths to Measure, Explain, and Enhance Program Behavior", *IEEE Computer*, July 2000.

[4] Beiman James M. and Janet L.Schultz, "An Empirical evaluation (and specification) of the all-du-paths testing criterion", in *Software Engineering Journal* 1992.

[5] Beiman James M. and Janet L.Schultz, "Estimating the number of test cases required to satisfy the all-du-paths testing criterion", in the *Proceedings of Software Testing, Analysis and Verification Symposium*, pages 179-186, Key West, Florida, December 1989.

[6] Beiman James M. and Linda M. Ott, "Measuring Functional Cohesion", in *IEEE Transactions of Software Engineering*, Vol. 20, No. 8, pages 644-657, August 1994.

[7] Belady L.A and M. M. Lehman, "A Model of Large Program Development", *IBM Systems Journal*, Vol 15, No. 3 1976, pp. 225-25

[8] Cimitile A, Lucia De. A and Munro M. "A Specification Driven Slicing Process for Identifying Reusable Function", *Software Maintenance: Research and Practice*, Vol. 8, No. 3 pages 145-178, 1996.

[9] Emerson Thomas J. "A Discriminant Metric for Module Cohesion", in *Proceedings of the 7th International Conference on Software Engineering*, pages 294-303, Los Alamitos CA, 1984.

[10] Fowler Martin, "Refactoring: Improving the Design of Existing Code" Addison Wesley Longman, Inc., 1999.

[11] Gallagher K. B and J. R. Lyle, "Using Program Slicing in Software Maintenance," *IEEE Transactions on Software Engineering*, vol. 17, no. 8, pages 751-761, 1991.

[12] Gupta Neelam, Aditya P. Mathur, and Mary Lou Soffa, "Automated Test Data Generation Using an Iterative Relaxation Method," in *Proceedings of ACM SIGSOFT Sixth International Symposium on Foundations of Software Engineering*, Orlando, FL, Nov. 1998.

[13] Gupta Neelam, Aditya P. Mathur, and Mary Lou Soffa, "UNA Based Iterative Test Data Generation and its Evaluation," in *Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, Cocoa Beech, FL, October 1999.

[14] Horwitz Susan, Thomas Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs", in *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 1, pages 26-60, January 1990.

[15] Korel B. and J. Laski, "Dynamic Program Slicing", *Information Processing Letters*, vol. 29, no. 3, pp, 155-163, 1988.

[16] Kang Byung-Kyoo and James M. Beiman, "A Quantitative Framework for Software Restructuring," in *Journal of Software Maintenance*, Vol. 11, pages 245-284, 1999.

[17] Kim, H. S, Y. R. Kwon and I. S. Chung, "Restructuring Programs through Program Slicing", *International Journal of Software Engineering and Knowledge Engineering*, Vol. 4, No. 3, pages 349-368, 1994.

[18] Laski, J. and B. Korel, "A data flow oriented program testing strategy", *IEEE Transactions of Software Engineering*, Vol. 9, No. 3, pages 347-354, 1983.

[19] Lehman M. M. and L. A. Belady, "Program Evolution-Processes of Software Change", *Academic Press*, New York, 1985.

[20] Longworth, H. "Slice based program metrics", Master's Thesis, Michigan Technological University, 1985.

[21] Ott Linda M. and Jeffrey J. Thuss, "The Relationship between Slices and Module Cohesion", in *Proceedings of 11th International Conference on Software Engineering*, IEEE Computer Society Press, Washington DC, pages 198-204, May 1989.

[22] Ott Linda M. and Jeffrey J. Thuss. "Slice based metrics for measuring cohesion", in *Proceedings of IEEE-CS International Symposium on Software Metrics*, 1993.

[23] Ott Linda M. and James M. Beiman. "Program Slices as an Abstraction for Cohesion Measurement" in *Information and Software Technology*, 40, pages 691-699, 1998.

[24] Parnas David L. "Software Aging", *Proceedings of 16th IEEE International Conference on Software Engineering*,pages 279-287, Sorrento, Italy, May, 1994

[25] Rapps, S. and Weyuker E.J., "Selecting software test data using data flow information", *IEEE Transactions of Software Engineering*, Vol. 11, No. 4, pages 367-375., 1985

[26] Selby, Richard W., and Victor R. Basili, "Analyzing Error-Prone System Structure", *IEEE Transactions of Software Engineering*, Vol. 17, No. 2, pages 141-152, February 1991.

[27] Stevens Wayne, Glenford Myers, and Larry L. Constantine. "Structured Design," *IBM Systems Journal*, Vol. 13, No. 2, pages 115-39, May 1974.

[28] *http://suif.stanford.edu/suif/suif2* The SUIF 2 Compiler System, Also Tutorial at *ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 4, 1999, Atlanta, GA.

[29] Tonella Paolo. "Concept Analysis for Module Restructuring", *IEEE Transactions of Software Engineering*, Vol. 27, No. 4, pages 351-363, 2001.

[30] Weiser, M. "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, pages 446-452, July 1982.

[31] Weiser, M. "Program Slicing", *IEEE Transactions of Software Engineering*, Vol. 10, No. 4, pages 352-357, 1984.

[32] Weyuker, E.J. "An Empirical study of the complexity of data flow testing", in the *Proceedings of Second Workshop on Software Testing, Verification and Analysis*, pages 188-195, Banff, Canada, 1988.

[33] Weyuker, E.J., "The cost of data flow testing: an empirical study", *IEEE Transactions of Software Engineering*, Vol. 16, No. 2, pages 121-128, 1990.

[34] Yourdon E. and Larry L. Constantine. *Structured Design*, Englewood Cliffs, N.J.: Prentice-Hall, 1979.