Cohesion Metrics

M. Harman, S. Danicic, B. Sivagurunathan,
B. Jones and Y. Sivagurunathan

Project *Project*
School of Computing
University of North London
Eden Grove, London, N7 8DB
tel: +44 71 607 2789
fax: +44 71 753 7009

e-mail: `m.harman@uk.ac.unl.clstr`

Abstract

We consider ways of measuring the cohesion of program fragments based upon techniques for program slicing, following the work of Ott et al [12, 2, 13, 11, 10]. The approach is based on the idea that the intersection of a program's slices represents that part of the fragment which is cohesive.

We produce cohesion metrics that are structurally identical to those of Ott and Thuss [13], the difference is that we consider different ways of measuring the significance of the intersection of a program's slices. We introduce expression metrics, to calculate the significance of the code in the intersection of slices, arguing that this approach may provide better answers than the 'Lines of Code' approach implicit in much of the literature [12, 10, 13].

The substitution of the 'Lines of Code' metric with alternative metrics, within the metrics defined by Ott and Thuss, motivates the consideration of metrics as higher–order functions, expressed in a pure functional programming language. This provides us with many convenient notational conventions with which to construct new metrics by partial application and instantiation of function–valued parameters, within such higher–order metrics.

We also raise some new issues concerning the computability of a cohesion metric, based upon program slices, indicating that the approach may, in some cases give an over–large value for cohesion.

1 Introduction

One of the problems facing an organisation attempting to use cohesion to measure the quality of their software, is that of finding a suitable *metric* which provides an adequate measure of the cohesion of a system or component. Unfortunately, the original formulation of cohesion [3] provides no well defined metric for measuring the level of cohesion exhibited by a system. Ott and Thuss [12] suggest a remedy, using the amount of 'overlap' between a program's slices, arguing that this might provide a good basis for a 'cohesion metric'.

A natural language definition of cohesion is [15]:

> "Cohesion is the measure of the strength of functional relatedness of elements within a module."

where a module is a "bounded, contiguous group of statements having a single name by which it can be referred to as a unit" [3] and an element is described [15] as being "any piece of code that accomplishes some work or defines some data".

The functional relatedness of a module is how tightly bound together a modules internal elements are to one another. Constantine and Yourdon [3] identify debugging as the major contributor to the cost of developing a computer system. By breaking the problem of developing a large system into smaller relatively independent pieces, the complexity of the system can be decreased, thus decreasing the errors and cost in development.

We shall use the notion of 'output variables' [13, 6] as a way of capturing the 'elements' of a piece of code, and we shall consider arbitrary code fragments for the cohesion that they exhibit. Following the work of Ott, Thuss, Bieman, Weiser and Longworth [12, 16, 2, 20, 7] we shall use program slices, and the overlap between slices constructed for output variables, as a basis for a measurement of the level of cohesion within a system. Our approach differs from these previous approaches in the way we choose to measure the significance of the sections of the program which are considered to be cohesive.

The rest of this paper is organised as follows:-

In section2 we introduce the concept of a program slice, and in sections 3 and 4 we show how the intersection of a program fragment's output slices can be used as a crude measure of cohesion. In section 5, we introduce expression metrics, which give a 'fine–grained' measure of the quality of the code residing in the 'cohesive section' of a program fragment, and in section 6 we use these expression metrics as a way of modifying the metrics of tightness, overlap and coverage introduced by Weiser, Longworth and Ott and Thuss. In section 7 we consider some problems with the approach. Section 8 describes the way in which our approach relates to previous work in this area, and section 9 contains directions for future work (most notably, this consist of exploring further the possibility of exploiting functional programming notational conventions in the definition of program metrics).

2 Slicing

A program slice [19, 20] is constructed with respect to a slicing criterion; a pair, $< V, n >$, where V is a set of variables and n is a line number. The slice is constructed by deleting any commands from the original program which can have no effect upon the values of variables in V at line number n.

For example consider the program (example 1) below:

Example 1

```
1    main()
2    {
3    int a,b,c,d,e,f;
4      c=4;
5      b=c;
6      a=b+c;
7      d=a+c;
8      f=d+b;
9      e=d+8;
10     b=30+f;
11     a=b+c;
12   }
```

A conventional slice of this program with respect to $(\{b\}, 12)$ simplified program below [5].

```
1    main()
2    {
3    int a,b,c,d,e,f;
4      c=4;
5      b=c;
6      a=b+c;
7      d=a+c;
8      f=d+b;
10     b=30+f;
12   }
```

Slicing has removed lines 9 and 11 because they have no effect upon the slicing criterion. In this paper we shall only be interested in slices constructed for output variables (defined below), and for criteria of the form (V, n), where V contains only output variables and for which the line number, n, is the last line of the program or program fragment.

Definition 1 (Output Variable)
An output variable is a modified global variable, a modified reference parameter or a variable which occurs in any expression used as a parameter to an output statement.

We define an 'output variable' with respect to a program component, c, so that output variables are those which contribute to the *overall* computation of c. Notice that we only consider those global variables and reference parameters whose values are *modified* by the program component[1], in contrast to the approach taken by Ott and Thuss [12] and Bieman and Ott [2], where *any* global variable or reference parameter counts as an 'output variable'. Lakhotia [6] also defines output variables (the 'processing elements' of [3]) to be only those which are modified.

Definition 2 (Output Slice Set)
Let p be a program fragment,
V^p be the set of output variables for the program fragment p and
n be the last line of p.

A slice q is a member of the output slice set of p if and only if q is constructed with respect to the slicing criterion $(\{x\}, n)$, where $x \in V^p$.

The output slice set is the set of slices obtained from a component by selecting all the slicing criteria which contain one of the output variables and for which the 'point of interest' is the end of the program[2].

3 The Cohesive Section

A crude measure of cohesion is given by the number of statements which occur in each and every output slice of the program. This is the starting point for measuring cohesion [12, 13]. We use the term 'cohesive section' for the distributed intersection of the set of output slices.

Ott and Thuss define the cohesive section as follows[3]:

Definition 3 (cohesive section)

$$SL^p_{int} = \bigcap_{v \in V^p} SL_v$$

That is, SL^p_{int} is the cohesive section of program p, the distributed intersection of the output slices of p. The 'crude measure of cohesion', is simply the cardinality of SL^p_{int}, written $\#(SL^p_{int})$. Better would be the ratio of statements in the cohesive section to statements in the program as a whole, called the 'tightness' metric [13], which we investigate in section 6.

4 Examples

Consider example 2 below:

Example 2

```
for (i=0;i<10;i=i+1) {
  input(num);
  NumArray[i] = num;
```

[1] In fact, calculating the set of variables whose value is modified in a program is not generally computable. We can, however, approximate this set by calculating the defined variables [1]. Such an approximation is *safe* in the sense, that a variable whose value *is* altered by a program will definitely be in the set of defined variables.

[2] For simplicity we shall assume that all output occurs at the end of the program fragment. Without loss of generality, we use only output variables which occur in output statements in our example (as opposed to call by reference parameters and global variables). These restrictions can be removed with relative ease, but this would complicate the exposition somewhat.

[3] This is essentially the definition proposed by Ott and Thuss ([12] and the definition of tightness [13]), with some minor technical differences arising from our definition of output variable.

```
1   for (i=0;i<10;i=i+1) {
2     input(num);
3     NumArray[i] = num;
    }
4   Smallest = NumArray[0];
5   Largest = Smallest;
6   i = 1;
7   while (i<10) {


10    if (Largest < NumArray[i])
11      Largest = NumArray[i];
12    i = i + 1;
    }
```

Slice A

```
1   for (i=0;i<10;i=i+1) {
2     input(num);
3     NumArray[i] = num;
    }
4   Smallest = NumArray[0];

6   i = 1;
7   while (i<10) {
8     if (Smallest > NumArray[i])
9       Smallest = NumArray[i];


12    i = i + 1;
    }
```

Slice B

Figure 1: Two Slices Representing High Cohesion

```
    }
4   Smallest = NumArray[0];
5   Largest = Smallest;
6   i = 1;
7   while (i<10) {
8     if (Smallest > NumArray[i])
9       Smallest = NumArray[i];
10    if (Largest < NumArray[i])
11      Largest = NumArray[i];
12    i = i + 1;
    }
13  output(Smallest);
14  output(Largest);
```

Slice A and Slice B of figure 1 are constructed with respect to the criteria $(\{Largest\}, 14)$ and $(\{Smallest\}, 14)$ respectively, forming the two output slices of example 2.

Clearly, this is a highly cohesive example, and slicing reveals this, because there is a great deal of overlap between Slice A and Slice B, that is, the cardinality of $SL_{int}^{example\ 2}$ is high, relative to the number of statements in the whole program. By contrast, example 3 below, is a rather less cohesive program fragment, as indicated by the way in which relatively few statements find their way into the cohesive section (see figure 2).

Example 3

```
1    Pass = 0 ;
2    Fail = 0 ;
3    Count = 0 ;
4    while (!eof()) {
5      input(Marks);
6      if (Marks >= 40)
7        Pass = Pass + 1;
8      if (Marks < 40)
9        Fail = Fail + 1;
10     Count = Count + 1;
```

```
3  Count = 0 ;
4  while (!eof()) {
5    input(Marks);
10   Count = Count + 1;
   }
```

Slice A

```
1  Pass = 0 ;
4  while (!eof()) {
5    input(Marks);
6    if (Marks >= 40)
7      Pass = Pass + 1;
   }
```

Slice B

```
2  Fail = 0 ;
4  while (!eof()) {
5    input(Marks);
8    if (Marks < 40)
9      Fail = Fail + 1;
   }
```

Slice C

Figure 2: Three Slices which Represent Low Cohesion

```
     }
11   output(Count) ;
12   output(Pass) ;
13   output(Fail) ;
```

4.1 Two Indistinguishable Programs

For examples 2 and 3 the use of the cohesive section provides a rough guide to the cohesion of the programs. However, the number of statements in the cohesive section may be too coarse a measure to be acceptable in all cases. Consider the examples 4 and 5 below:

Example 4

```
p = a+b ;
q = b-a;
x = sin(q)*cos(q)/(sin(p)+cos(p)) ;
y = cos(p)*sin(p)/(sin(q)+cos(q)) - 2*sin(p)*sin(q) ;
output(x,y) ;
```

Example 5

```
p = sin(a)*cos(a)/(sin(b)+cos(b)) ;
q = cos(b)*sin(b)/(sin(a)+cos(a)) - 2*sin(b)*sin(a) ;
x = p+q ;
y = p-q ;
output(x,y) ;
```

For both examples 4 and 5 the cohesive section contains two statements (the assignments to `p` and `q`). Using the quantitive measure of cohesion - number of statements in the cohesive section, we would conclude that the two programs are equally cohesive. However, our intuition indicates that the two assignments to `p` and to `q` in example 4 are of less significance than those to `p` and `q` in example 5. We might, therefore, consider example 5 to be more cohesive than example 4. In order to distinguish between these two program fragments we need a 'finer grained' measure than the 'number of statements'.

5 Fine Grain Metrics

Clearly, we could measure the McCabe complexity [9] of the cohesive section, with the view that high cohesion is characterized by a complex cohesive section. Sadly this will not be suitable because the metrics are too 'coarse grained'. For example, they fail to distinguish between complex assignment statements and simple ones, all would receive a McCabe measure of 1.

For the purpose of measuring the cohesion of individual functions and procedures in a program, we require a method of measuring the complexity of a program at a lower level than that of programming language constructs.

In this section, we introduce ways in which program *expressions* can be measured for complexity, thus allowing us to measure, with a high degree of precision, the complexity of 'fine grain' system components. This allows us to make metrics based upon measuring the cohesive section more sensitive to the quality of the code found in the cohesive section.

5.1 Expression Complexity

Complexity metrics are traditionally aimed at calculating properties of programs in terms of their statement constructs. The natural level to which to descend from the statement level is the expression level, since we are concerned with the complexity of the *contents* of individual expressions.

We consider four ways of measuring the complexity of an expression, based upon the number of nodes and/or leaves in its abstract syntax tree, the number of tokens in the concrete syntax and the numbers of variable names which occur.

There are many ways to compare the complexity of two expressions. Consider the following two expressions:

$$e_1 = x + x + x \quad \text{and} \quad e_2 = 3 * x$$

Intuitively e_2 is simpler than e_1 because it contains fewer characters, fewer nodes in its abstract syntax tree, fewer variable references and so on. We shall consider four measures of expression complexity, in an attempt to capture these intuitions about expression complexity (although many other alternatives are possible):

V : $< exp > \rightarrow \mathbb{R}$
V(e) is the number of distinct variables in the expression e. For example V(`x+x-5*y`) = 2
N : $< exp > \rightarrow \mathbb{R}$
N(e) is the number of internal nodes in the abstract syntax of the expression e. For example N(`x+x-5*y`) = 3
L : $< exp > \rightarrow \mathbb{R}$
L(e) is the number of leaves in the abstract syntax of the expression e. For example L(`x+x-5*y`) = 4
T : $< exp > \rightarrow \mathbb{R}$
T(e) is the number of tokens in the concrete syntax of the expression e. For example T(`x+x-5*y`) = 7

6 Five Metrics Introduced by Ott and Thuss

In this section we describe five metrics introduced by Ott and Thuss [13], all of which are concerned with measuring cohesion based upon slices[4]. each of the metrics is a codification of the informally defined slice–based metrics proposed by Weiser in his seminal paper on program slicing [20].

First we present the metrics as suggested by Ott and Thuss, then we consider replacing the 'Lines of Code' metric, which is used implicitly in the way each calculates the significance of portions of code, such an output slice, the cohesive section and the program fragment as a whole. The replacements we consider measure the significance of these portions of code using the expression metrics we introduced in section 5.1.

We require a few auxiliary definitions. Let V^p be the set of output variables of a program p. Let SL_i^p be a slice constructed from program p with the slicing criterion $(\{i\}, L)$, where L is the last line of p. Let $\#(x)$ denote the number of elements in a set x, $\mathcal{P}(x)$ denote the powerset of a set x and let S be the set of statements in the language under consideration.

Recall definition 3, which defines the intersection of the output slices of p to be SL_{int}^p.

Definition 4 (Coverage)

$$\text{Coverage} : \mathcal{P}(S) \rightarrow \mathbb{R}$$

[4]We have made slight alterations to the exposition, which we believe, make the metrics easier to read, but which in now way affect the function computed by any of them.

$$\text{Coverage}(p) = \frac{\sum_{v \in V^p} \#(SL_v)}{\#(V^p) \times \#(p)}$$

The coverage is the average ratio of the number of statements in an output slice to number of statements in the fragment as a whole.

Definition 5 (Overlap)

$$\text{Overlap} : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{Overlap}(p) = \frac{\sum_{v \in V^p} \frac{\#(SL^p_{int})}{\#(SL_v)}}{\#(V^p)}$$

The overlap is the average amount of an output slice which is in the cohesive section.

Definition 6 (Tightness)

$$\text{Tightness} : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{Tightness}(p) = \frac{\#(SL^p_{int})}{\#(p)}$$

The tightness is the size of the cohesive section relative to the size of the program fragment as a whole. This is the 'crude metric' which allowed us to decide that example 2 exhibited high cohesion, whilst example 3 exhibited low cohesion. It also, however, leads us to believe that examples 4 and 5 are equally cohesive.

Definition 7 (MinCoverage)

$$\text{MinCoverage} : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{MinCoverage}(p) = \frac{\min_{v \in V^p} \#(SL_v)}{\#(p)}$$

The MinCoverage is the ratio of the amount of statements in the smallest output slice to the size of the program fragment as a whole.

Definition 8 (MaxCoverage)

$$\text{MaxCoverage} : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{MaxCoverage}(p) = \frac{\max_{v \in V^p} \#(SL_v)}{\#(p)}$$

The MaxCoverage is the ratio of the amount of statements in the largest output slice to the size of the program fragment as a whole.

We construct five tables (figures 3,4,5,6,7), which give the results of applying twenty five metrics. These are the five versions of the Ott and Thuss metrics, and the four alternative versions, each of which replaces the 'statement count' component of the original with one of the four expression metrics introduced in section 5.1. We count the total number of leaves, nodes, tokens and distinct variables in place of the statement count. Thus each of Ott and Thuss' original five metrics are replaced by the alternative versions below, where M is one of the four expression metrics:

Definition 9 (Expression Coverage)

$$\text{Coverage}' : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{Coverage}'(p) = \frac{\sum_{v \in V^p} M(SL_v)}{\#(V^p) \times M(p)}$$

Definition 10 (Expression Overlap)

$$\text{Overlap}' : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{Overlap}'(p) = \frac{\sum_{v \in V^p} \frac{M(SL^p_{int})}{M(SL_v)}}{\#(V^p)}$$

Definition 11 (Expression Tightness)

$$\text{Tightness}' : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{Tightness}'(p) = \frac{M(SL^p_{int})}{M(p)}$$

Definition 12 (Expression MinCoverage)

$$\text{MinCoverage}' : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{MinCoverage}'(p) = \frac{\min_{v \in V^p} M(SL_v)}{M(p)}$$

Definition 13 (Expression MaxCoverage)

$$\text{MaxCoverage}' : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{MaxCoverage}'(p) = \frac{\max_{v \in V^p} M(SL_v)}{M(p)}$$

In section 9, we briefly suggest view of program metrics based upon the functional programming paradigm [21], which will enable us to reason about the replacement of subordinate metrics within larger metrics in a convenient notational style.

Each table shows that whilst the original versions of the metrics fail to distinguish between the two examples 4 and 5, the other versions of the metrics, based upon expression metrics, *do* distinguish between these two programs.

From the tables we can see that the distinct variable appears to be inappropriate as a basis for calculating the significance of the cohesive section, due to the fact that it gives the same value in many cases, failing to distinguish between the two example program fragments. This is to be expected, since the number of distinct variables in a program is nearly as 'coarse' a metric as the number of statements in a program.

We can also see that the metrics based on the 'Lines of Code' metric are also inappropriate, because they fail to distinguish between the two programs. The metric values for example 5 are identical across the different metrics because both output slices contain the same number of tokens. This does not happen in the case of example 4 because the two output slices contain structurally different expressions.

Finally, we observe that our intuition concerning the two simple programs in reflected by the fact that example 5 is more cohesive than example 4 according to our versions of the Ott and Thuss metrics for leaf, node and token counts.

7 Some Other Issues

In this section we informally consider some remaining issues associated with using slices as a basis for measuring cohesion, and, more generally, with the very notion of automated 'measurement' of cohesion. The most significant result is that we prove (albeit informally) that cohesion metrics based upon slicing will produce over–large values for cohesion in some cases, due to the non–computability of statement–minimal slices. This leads us to conjecture that cohesion may not, in general, be a computable property of programs.

Tightness Metrics	Example 4	Example 5
Original (using lines of code count)	0.4	0.4
Using leaf count	0.24	0.65
Using variable count	0.5	0.5
Using internal node count	0.1	0.9
Using token count	0.1	0.87

Figure 3: Tightness

Overlap Metrics	Example 4	Example 5
Original (using lines of code count)	0.67	0.67
Using leaf count	0.43	0.85
Using distinct variable count	0.5	0.5
Using internal node count	0.18	0.95
Using token count	0.19	0.95

Figure 4: Overlap

Coverage Metrics	Example 4	Example 5
Original (using lines of code count)	0.6	0.6
Using leaf count	0.56	0.76
Using distinct variable count	1.0	1.0
Using internal node count	0.55	0.95
Using token count	0.53	0.92

Figure 5: Coverage

Min Coverage Metrics	Example 4	Example 5
Original (using lines of code count)	0.6	0.6
Using leaf count	0.47	0.76
Using distinct variable count	1.0	1.0
Using internal node count	0.43	0.95
Using token count	0.44	0.92

Figure 6: Minimum Coverage

Max Coverage Metrics	Example 4	Example 5
Original (using lines of code count)	0.6	0.6
Using leaf count	0.65	0.76
Using distinct variable count	1.0	1.0
Using internal node count	0.67	0.95
Using token count	0.63	0.92

Figure 7: Maximum Coverage

7.1 Is Cohesion Computable?

Of necessity metrics are constructed with respect to a program's syntax. It may be that cohesion should be regarded as a semantic property, which can sometimes be evaluated accurately from a program's syntax, but which may in some cases be non–computable. This seems reasonable, after all similar program properties such as the referenced variables and defined variables [1] are not computable, but we can form conservative approximations to these sets. The approximations will be safe, because they will always include a variable if it is defined (respectively 'referenced'), but may include *too many* variables.

For example, in the expression `x-x`, there are *no* referenced variables, but conventional (syntax–directed) algorithms for calculating defined variable sets will decide that `x` is referenced by the expression `x-x`. Notice that we cannot simply 'fix' this problem by adding a special 'check' for the expression `x-x`, because the situation applies to the general expression schema $E - F$, where E and F are semantically equivalent expressions, which happen to have different syntax. Detecting that E is equivalent to F is not computable, because the problem of deciding whether two arbitrary functions are equivalent is, in general, undecidable.

Now, clearly, if we had defined a metric "number of referenced variables", this would not be computable, because the referenced variable set is not computable. However, as we can always construct a conservative approximation to the referenced variable set, we can also construct a conservative value for the "referenced variable metric".

Turning to cohesion, even though it may be that cohesion is computable[5], we can readily prove that measures based upon program slices *must* yield conservative approximations to the true measure of cohesion. This is because it is known that constructing statement–minimal slices is not computable [20]. To circumvent this problem, slicing algorithms are forced to produce potentially over–large slices. That is a slice constructed by an algorithm will always be *at least* as big as the set of statements which are *genuinely* affected the slicing criterion.

In turn, this means that metrics based on slices for calculating the portion of code which is 'cohesive', may give over–large values for each of the cohesive metrics described by Ott et al (and by us).

7.2 Is Cohesion a Syntactic or a Semantic Property?

A further problem with measuring cohesion could lie in the very nature of cohesion itself. Can we be sure that cohesion is a property solely dependent upon the syntax of a program, or are we relying upon some semantic intuition about the *purpose* of a program, when we ascribe an intuitive measure for cohesion?

Perhaps, when we say

> "Program x is more cohesive than program y"

we really mean that we know about the programmers' intention was in writing programs x and y, and specifically, we know that it is possible to coalesce the disparate computations of program y.

Consider the simple example below:

```
x=1;
y=2;
output(x,y);
```

which contains two processing elements, x and y, and couldn't be less cohesive (according to the intersection of its slices, which is empty). However, if we re–write the program as the semantically equivalent version below:

```
x=1;
y=x+1;
output(x,y);
```

then we shall find that the intersection of the slices contains the assignment to `x`. Now, are we to conclude that the second version is more cohesive than the first? Perhaps we can. After all, cohesion *is* a property of *how* a program is written and not a property of *what* the program computes. We should, therefore be able to re–write a non–cohesive program in a more cohesive style.

[5] Intuitively this seems unlikely, as it is surely at least as 'deep' as calculating referenced variable sets

7.3 Redundant Code

In the metrics for Coverage, Tightness and Overlap, we have used the number of statements in the program as a whole, as a 'benchmark' against which we have compare the numbers of statements in various slices and intersections of slices. Perhaps, instead, we should use the union of the output slices for our 'benchmark', because the program may contain code that does not contribute to the computation of *any* output slices, thus artificially reducing the value obtained when the cohesion metrics are calculated.

8 Related Work

Mark Weiser introduced Program slicing [18] and postulated the notion of metrics based upon calculation involving slices [20].

Longworth codified Weiser's informal suggestion of a Coverage, Overlap and Tightness metric in his master's thesis [7], and his work has been considerably developed by Ott and Thuss [12, 8, 16] who introduce the notion of the cohesive section and of 'metric slice'. A metric slice, is the union of the conventional (backward) slice, with the corresponding forward slice [17] constructed for the same criterion. A forward slice is the set of statements which are affected *by* the slicing criterion. Ott and Thuss claim that the use of metric slices leads to cohesion metrics which are less sensitive to minor program changes than those based upon the backward slice alone. The use of end–slices [6], circumvents this problem by choosing the end of the program as the point at which to construct the slice, thus the forward slice would (in the case of end slices) be empty. In this paper we have also used end–slices as do Ott et al in [12, 2].

Later, Ott and Ott and Thuss [10, 13] elaborated upon the notion of metric slice and the measures of tightness etc., focusing on the impact of the metrics on software maintenance [10] and looking at the effects of changes to the programs upon the values obtained from the metrics [13].

Recently, [2], Bieman and Ott extended the work of Ott and Thuss, by defining metrics for measuring cohesion based upon data slices. A data slice contains all the lexical tokens for variable identifiers and for constants, and so is strongly related to our notion of expression complexity based upon the number of leaves and the number of tokens in an expression.

Also, in [2] Bieman and Ott calculate a metric for measuring the 'average adhesiveness' of a procedure p. A token is adhesive if it is used in more than one output slice (it 'glues' these slices together). The adhesiveness of a token which glues together more than one slice, is the ratio of the number of slices containing the token to the total number of slices. The adhesiveness of an entire function is the average adhesiveness of its tokens.

The use of data slices, gives a fine grained measure of cohesion, so that examples 4 and 5 will receive different values for cohesion, overcoming the difficulties associated with the implicit use of the 'Lines of Code' metric in the measures of tightness, overlap and coverage.

The essential difference between our approach and that of Bieman and Ott is in the choice of a data slice in the case of Bieman and Ott, compared to the choice of expression metrics in our work. We shall need to investigate further the differences between these two approaches, before we feel able to make a definitive statement about the relative merits of each approach. It appears that the choice of a data slice, corresponds roughly to the use of the 'leaves' metric for expressions. The difference between the two is that the leaves count applies only to expression, whilst the data slice will contain leaves *and* instances of variable names occurring on the LHS of assignment statements and in declarations.

We should also point out, that Bieman and Ott have augmented their work with a more rigorous investigation of the scale properties [4, 22] of their measures. Our work, by comparison, is at an earlier stage of development.

Arun Lakhotia [6] takes a different approach. His work is more closely related to the original seven levels of cohesion identified by Constantine and Yourdon [3]. Effectively, Lakhotia codifies the natural language definitions of the seven levels of cohesion using logical expressions whose terms are expressions concerning the properties of the Variable Dependence Graph, a variant of the Program Dependence Graph [14].

The Variable Dependence Graph has variable names as nodes and has two kinds of edge: data and control, of which the control edges are labelled and the data edges are unlabelled. A data edge from x to y indicates that the variable x is used in defining the value of the variable y. A control edge from x to y, labelled with the pair (n, k), indicates that a predicate node n references a variable x, and n controls the execution of an assignment to the variable y down its 'k branch', where k is either $true$ or $false$. Like Ott et al, Lakhotia chooses output variables to be the 'processing elements' concerned in the natural language definition of cohesion. He then 'translates' each of the seven levels of of cohesion (as defined using natural language) into logic. For example if we have $x \rightarrow y \vee y \rightarrow x$ then the value of x is used to defined y or vice versa. This corresponds closely to the natural language description of sequential cohesion:

> "The output of one processing element serves as the input to the other."

Lakhotia proposes the logical expression, $x \rightarrow y \vee y \rightarrow x$, as a replacement for the natural language version, and gives similar logical expressions, capturing the other six levels of cohesion, demonstrating the correspondence between his formal definitions, and the natural language versions with examples from the literature on cohesion.

9 Future Work

We are currently investigating the way in which the notational conventions of functional programming languages such as ML [21] can be used to reason about program metrics as higher–order functions. This seems natural as a metric is a function, and there is no reason not to exploit the benefits of the pure functional style. The possibility of defining higher–order metrics, which take as their parameters, metrics, as well as the programs which they are to measure, offers a way of avoiding the exponential increase in notation required to handle the many possible substitutes for 'Line of Code' in the five metrics introduced by Ott and Thuss (see section 6).

For example, the notation used to combine the first order expression metrics defined in section 5.1, could have been written as a higher–order function, whose two parameters are the metric used to calculate the complexity of an expression, and the set of statements to be measured. In order to calculate the sum of the individual metric value for each expression in a set of statements, we would define this higher–order summation metric as follows (where $\mathcal{E}[\![s]\!]$ denotes the expressions of a set of statements s):

Definition 14 (Summation)

$$\mathbf{E}^{sum} : (< exp > \rightarrow \mathbb{R}) \rightarrow \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\mathbf{E}^{sum}\ M\ p = \sum_{e \in \mathcal{E}[\![p]\!]} M(e)$$

The metrics for the total number of leaves, internal nodes, distinct variables and tokens of a set of statements are thus given by partially applying $\mathbf{E}^{sum}$ to each of the metrics for leaves, internal nodes, distinct variables and tokens as follows:

$$TotalLeaves = \mathbf{E}^{sum} L$$

$$TotalInternalNodes = \mathbf{E}^{sum} N$$

$$TotalDistinctVariables = \mathbf{E}^{sum} V$$

and

$$TotalTokens = \mathbf{E}^{sum} T$$

In a similar manner, we could have raised the order of the five metrics introduced by Ott and Thuss [13], by abstracting the 'lines of code' metric used in their definition. This gives us a higher–order version of the metrics. For instance, performing the abstraction on the 'tightness' metric, gives us the higher–order version Tightness$'$, defined below:

$$\text{Tightness}' : (\mathcal{P}(S) \rightarrow \mathbb{R}) \rightarrow \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{Tightness}'\ M\ p = \frac{M(SL^{p}_{int})}{M(p)}$$

We can use this higher–order version to define, for example, the version of tightness based upon the total number of expression leaves, given in figure 3, as follows:

$$\text{LeafTightness} : \mathcal{P}(S) \rightarrow \mathbb{R}$$

$$\text{LeafTightness} = ottightness(\mathbf{E}^{sum}\, L)$$

Alternatively, if LoC denotes the 'lines of code metric, then we have:

$$\text{Tightness} = \text{Tightness}'\ LoC$$

demonstrating that the abstraction of these metrics from first order functions to higher–order functions does not prevent us from returning to the original first order versions, if we so choose.

More work is required to investigate the notational potential of this approach.

10 Conclusion

We have introduced some simple metrics for evaluating the 'complexity' of an expressions and have used these in some standard metrics introduced by Ott and her colleagues. Specifically, we have replaced the use of the 'Lines of Code' metric with these expression metrics because we believe they give a more accurate measure of cohesion (a belief which we have yet to prove rigorously, but which intuition and the examples we have considered appear to verify).

We have also informally proved that slice–based cohesion metrics are liable to give measures of cohesion which are larger than they should be, due to the non–computability of statement–minimal slices, leading us to postulate that cohesion may not be a computable property of programs.

More work is required to establish the scale properties of expression metrics and to develop a notation which will allow us to consider metrics as more general, higher–order functions, written in a pure functional notation. We believe that the functional style of programming will be extremely useful in this respect.

Acknowledgements

We would also like to thank Dan Simpson of Brighton University for his advice and support.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, techniques and tools.* Addison Wesley, 1986.

[2] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[3] Larry L. Constantine and Edward Yourdon. *Structured Design.* Prentice Hall, 1979.

[4] Norman E. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[5] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[6] Arun Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[7] H. D. Longworth. Slice–based program metrics. Master's thesis, Michigan Technological University, 1985.

[8] H. D. Longworth, L. M. Ott, and M. R. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)*, pages 383–389, 1986.

[9] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[10] Linda M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium*, pages 16–23, 1992.

[11] Linda M. Ott and James M. Bieman. Effects of software changes on module cohesion. In *IEEE Conference on Software Maintenance*, pages 345–353, November 1992.

[12] Linda M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 11^{th} ACM conference on Software Engineering*, pages 198–204, May 1989.

[13] Linda M. Ott and Jeff J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 71–81, Baltimore, Maryland, USA, May 1993. IEEE Computer Society Press, Los Alamitos, California, USA.

[14] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[15] Meilir Page-Jones. *Practical Guide to Structured System Design.* Prentice Hall, 1980.

[16] J. J. Thuss. An investigation into slice–based cohesion metrics. Master's thesis, Michigan Technological University, 1988.

[17] Guda A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[18] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[19] Mark Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[20] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[21] Åke Wikstrom. *Functional Programming Using Standard ML.* Prentice Hall, 1987.

[22] H. Zuse. Support of validation of software measures by measurement theory. In *15th International Conference of Software Engineering*, Baltimore, MD, May 1993.