Slice-Based Measurement of Function Coupling
Position Paper

Mark Harman, Margaret Okulawon, Bala Sivagurunathan and Sebastian Danicic
Project *Project* , School of Computing, University of North London,
Eden Grove, London, N7 8DB, UK.
tel: +44 71 753 5751
fax: +44 71 753 7009
e-mail: m.harman@unl.ac.uk

Keywords: Slicing, Measurement, Coupling and Cohesion

Abstract *In this paper we outline an approach to coupling measurement based upon program slicing. We claim that our coupling metric produces more precise measurements than related information-flow-based metrics.*

A prototype tool has been implemented to calculate these metrics. The next step is to evaluate the coupling (and related cohesion) metrics with respect to some hypotheses relating to the predication systems in which we claim they may be embedded.

1 Background

After initial interest in code-level metrics in the 70s and early 80s the focus of software measurement moved 'upstream' to the earlier stages of the product development life cycle. However, the definition and evaluation of code-level metrics has received renewed attention with the rise in popularity of the object-oriented paradigm [2, 3, 11]. This paper is concerned with code-level, slice-based metrics for assessing the level of information flow between functions and their coupling through this information flow.

Our approach is an extension of related work on slice-based cohesion measurement, which has been covered extensively in the literature [1, 17, 19, 20]. The new slice-based measurements we introduce refine the idea of information-flow [13, 12] which has traditionally been associated with coupling measurement.

2 Position

We believe that code-level metrics are not only useful to managers concerned with controlling and predicting the development process, but are also helpful to developers and maintainers, for which estimations of code properties act as potential predictors, which can be used in evaluating and directing the testing and reuse process.

We define code-level metrics for assessing coupling and outline the use of the metrics as components of predication systems which can be deployed during testing, comprehension and re-engineering.

We propose a set of hypotheses with respect to which we intend to attempt a validation of the metrics we propose and related cohesion metrics from the literature.

3 Slice-Based Measurement

A program slice captures the thread of computation associated with some set of variables at some point in a program [23, 15, 5, 7, 10]. Definition 1 below describes a static program slice, with respect to which slice-based measurements are defined.

Definition 1 (Program Slice)
A slice of a program p, at a node n in the Control Flow Graph (CFG) of p, with respect to a set of variables K, is a program p', constructed by deleting certain nodes from the CFG of p. The nodes which may be deleted are those which can have no effect upon the values residing in any of the variables in K, when execution reaches n. The pair (K, n) is known as the 'slicing criterion'. The notation $\mathcal{SS}(p, K, n)$ will be used to denote a slice of the program p with respect to the slicing criterion (K, n).

The slice described by definition 1 has come to be known as the backward slice after the introduction of forward slicing [15] by Horwitz et al. Whereas a backward slice contains those nodes of the CFG which affect the slicing criterion, a forward slice contains those nodes which are affected by the slicing criterion.

Figure 1 illustrates the way in which program slicing has been used to estimate the cohesiveness of a code fragment, using the approach introduced by Ott [19]. The code fragment in figure 1 has two variables 'of interest'[1], namely `Smallest` and `Largest`. The intersection (which we call the 'cohesive section') of the slices of these two variables indicates those statements whose computation contributes to the final values of both the variables. Thus, in some sense, the cohesive section captures the cohesive part of the program.

In the example fragment in figure 1, the level of cohesion is deemed to be relatively high as seven out of ten of the lines of the function find their way into the cohesive section. Bieman and Ott [1] provide a more formal treatment, showing how this informal observation leads to a rigorous definition of module cohesion.

3.1 More Precise Information Flow

Coupling is the measure of how one module depends upon or affects the behaviour of another. Traditionally, if coupling is measured at all, it is calculated in terms of Henry and Kafura's information flow metrics [13, 12].

Using slicing we may construct a slice for each of the principle variables of a function[2] f in the context of some program p. Some of each slice may lie inside the function body and some outside the function. The nodes of a slice which lie outside the function are those nodes of the program which affect the value computed by the function. They reflect how strongly the function depends upon the rest of the program.

Observe that this is a more precise measure of dependency than information flow, since the presence in the slice of a single node outside the body of f indicates an information flow from p to f. However, as well as detecting such a flow, we can assess the 'amount' of information flow (its 'bandwidth') by determining precisely how much of the slice lies outside the body of f. The slice-based measurement is therefore more precise in describing the *level* of information flow than measurements which merely detect its presence.

We now turn to a more formal definition of slice-based information flow metrics. In the following definitions $\mathbf{N}(f)$ denotes the number of nodes in the CFG of a function or program f, $\mathbf{E}(f)$ denotes the last line of a function f, $\mathbf{P}(f)$ denotes the principal variables of a function f, $\#$ and $|$ denote set cardinality and restriction operators respectively.

Definition 2 (Flow into Function)
We define the flow, $\mathcal{FI}^p_f$, into a function f from its context p (a program containing f) as follows

$$\frac{\#((\bigcup_{v \in \mathbf{P}(f)} \mathcal{SS}(p, \{v\}, \mathbf{E}(f))) - \mathbf{N}(f))}{\mathbf{N}(p) - \mathbf{N}(f)}$$

The flow into measure uses backward slicing to assess how much of the program containing a function affects the outcome of the function. Using forward slicing we could assess how much of the containing program *is affected by* a function. We would simply exchange the use of backward slicing, $\mathcal{SS}(p, \{v\}, \mathbf{E}(f))$ in definition 2, with corresponding forward slices.

Definition 3 below generalises definition 2 to the situation where the flow occurs between two functions rather than the flow to a function from the context in which a function is placed.

Definition 3 (Flow between functions)
We define the flow, $\mathcal{FF}^p_{(f,g)}$, from a function f to a function g in a program p (which contains both f and g) as follows

$$\frac{\#((\bigcup_{v \in \mathbf{P}(g)} \mathcal{SS}(p, \{v\}, \mathbf{E}(g))) \mid \mathbf{N}(f))}{\mathbf{N}(f)}$$

Definition 3 measures the relative amount of the body of f which contributes to the computation of the result of g. Once again, were a single node of f to be in any slice constructed at the end of g then this would indicate the *presence* of an information flow from f to g. The slice-based measure not only allows us to detect such an information flow[3], but also, it allows us to measure the bandwidth of information flow.

4 Coupling

Coupling is a measure of how information flows between functions [4]. To measure coupling it is there-

[1] This idea of variable of interest is made precise in [1].

[2] In this paper we shall assume, without loss of generality, that the only variables of interest are those output at the end of a function.

[3] That is, there is an information flow from f to g iff

$$\mathcal{FF}^p_{(f,g)} \neq 0$$

.

Original		Slice on `Smallest`	Slice on `Largest`	Cohesive Section Member?
1	`for (i=0;i<10;i=i+1) {`	S	L	√
2	`  input(num);`	S	L	√
3	`  NumArray[i] = num;`	S	L	√
	`}`			
4	`Smallest = NumArray[0];`	S	L	√
5	`Largest = Smallest;`		L	
6	`i = 1;`	S	L	√
7	`while (i<10) {`	S	L	√
8	`  if (Smallest > NumArray[i])`	S		
	`    Smallest = NumArray[i];`	S		
10	`  if (Largest < NumArray[i])`		L	
11	`    Largest = NumArray[i];`		L	
12	`  i = i + 1;`	S	L	√
	`}`			

Figure 1: Two Slices Representing High Cohesion

fore necessary to see how much of each of the functions f and g contribute to one another, normalised by the relative sizes of the bodies of the two functions. This can be calculated in terms of the flow into each of the functions from the other.

Definition 4 (Coupling)
We define the level of coupling, $\mathcal{C}^p_{(f,g)}$, between two functions f and g in a program p (which contains both f and g) as follows

$$\frac{\mathcal{FF}^p_{(f,g)}.\mathbf{N}(f) + \mathcal{FF}^p_{(g,f)}.\mathbf{N}(g)}{\mathbf{N}(f).\mathbf{N}(g)}$$

For example, consider the program in Figure 2. The function[4] `main` has a single principal variable, `a`. Slicing on the value of `a` at the end of the function `main` will produce the slice $\{1,3,4,6,7,8,9\}$. The portion of this slice in function `SumProd` is $\{1,3,4,6\}$, so, as there are 6 lines in `SumProd` the relative amount of `SumProd` that contributes to `main` is $\frac{4}{6}$. This is our measure of flow from `SumProd` to `main`.

Notice that using information flow, we would simply notice that the value of the four variables `Data`, `s`, `p` and `a` in `main` obtained their values from the function `SumProd` and that, there were therefore, 4 information flows from `SumProd` to `main`.

Of course, three of these four are irrelevant, as their value is discarded. However, even if we amend the definition of information flow to take account of this, we are left with the observation:

[4]There are some node labelling issues related to the graphical representation of function [14]. As we are interested in nodes which form part of the conventional CFG of the body of a function, unimportant nodes have been left unlabelled.

> "there is a flow from `SumProd` to `main` via the last parameter of `SumProd`".

Using slicing, we refine this observation to:

> "there is a flow from `SumProd` to `main` via the last parameter of `SumProd` with a bandwidth of $\frac{4}{6}$".

Had line 10 of `main` printed the value of `s`, we would have the slice $\{1,3,4,7,8,9\}$. In this case three nodes of `SumProd` would have contributed to the result of `main` and so the measure of flow from `SumProd` to `main` would have been $\frac{3}{6}$. Putting `s` in place of `a` in the print statement of `SumProd` therefore reduces the bandwidth of information flow from `SumProd` to `main`.

Coupling is simply measured as the normalised weighted flow between two functions. For the two functions in Figure 2 this produces a measurement of $\frac{4}{10}$ as there are ten nodes altogether of which four affect flows from `SumProd` to `main` and none affect flows from `main` to `SumProd`.

4.1 Prediction Systems

Our current implementation of these metrics is based upon the static backward slicing tool Unravel [18]. In order to fully test the hypotheses below, we shall need to extend the tool to allow construction of slices in a forward direction, together with the associated metrics.

Ideally, we should like to be able to evaluate whether our metrics satisfy the representation theorem [21, 6]. As coupling is measured on an ordinal

```
void SumProd(int a[100], int* sum,
             int* prod, int* ave)
{ int i;
  sum = 0;
  prod = 1;
  for(i=0;i<=99;i++)
  {  sum = sum+a[i];
     prod = prod*a[i]; }
  ave = sum/100; }

main()
{ int Data[100]; int i; int s,p,a;
  for(i=0;i<=99;i++)
    scanf("%d",&Data[i]);
  SumProd(Data,&s,&p,&a);
  printf("%d",a); }
```

Figure 2: Coupling and Information Flow Example

scale[5], testing our metrics with respect to the representation theorem will require knowledge about the 'true rank order' of a set of function pairs according to their 'real level of coupling'. As this information is not readily available, we aim to evaluate 'internal consistency' which appears to be a necessary prerequisite.

The first two hypotheses are concerned with checking this internal consistency of the metrics. If hypotheses 1 and 2 do not hold then there is something wrong with our view of cohesion and coupling respectively.

Hypothesis 1 *There is a strong correlation between cohesion measured at the end of a function using backward slicing on its defined variables and that measured at the beginning of a function using forward slicing on its referenced variables.*

That is, it should make little difference whether the direction of slicing is forward or backward as the justification for slice-based cohesion measurement rests upon the parts of the program which are bound together by the threads of the computation to which they contribute. Backward slicing will identify these threads by their target, while forward slicing will identify them from their source.

Hypothesis 2 *The is a strong correlation between the flow from a function f to a function g measured using backward slicing at the end of g on its defined variables and the flow to a function g from a function f measured using forward slicing at the end of f on its defined variables.*

[5]or so the present authors believe, but have yet to prove.

Hypothesis 2 is concerned with assessing the validity of slice-based information flow measurement in a similar manner to the way hypothesis 1 assesses the validity of slice-based cohesion measurement.

The next three hypotheses relate our metrics to the error predication systems in which they may be embedded.

Hypothesis 3 *There is a correlation between flow into f from a program p and the manifestation of errors in f caused by defects in the context of f (that is, in p but not in f).*

Hypothesis 4 *There is a correlation between flow out of f to a program p and the manifestation of errors in p (but not f) caused by defects in f.*

Hypothesis 5 *There is a correlation between coupling of functions f and g and the occurrence of errors in the two. That is, errors in f cause 'knock-on' errors in g and vice versa.*

It would also be interesting to see if these metrics can predict other, perhaps more significant, properties, such as reusability, but such properties may be harder to evaluate.

Hypothesis 6 *Functions with low coupling are easy to extract and reuse.*

5 Comparison

Approaches to cohesion measurement are based upon program slicing [23, 14, 22, 9]. Linda Ott and her colleagues were the first to point out the close correspondence between slicing and module cohesion [19]. The most complete and up-to-date treatment is due to Bieman and Ott [1] and the problem was also addressed by Lakhotia [16], who used dependence analysis (a prerequisite of slicing) to codify the seven levels of cohesion informally defined by Constantine and Yourdon [4].

Some of the present authors made a small contribution to this work on slicing as a basis for cohesion measurement [8], showing that a finer form of slice may be required for cohesion measurement and addressing some of the computability issues associated with cohesion measurement based upon program slicing. We also went a little way towards the extension of Ott and Bieman's work to the problem of object-oriented code-measurement.

The present paper is the first to address the problem of coupling measurement using slicing.

References

[1] Bieman, J. M., and Ott, L. M. Measuring functional cohesion. *IEEE Transactions on Software Engineering 20*, 8 (Aug. 1994), 644–657.

[2] Chidamber, S. R., and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering 20*, 6 (June 1994), 476–493.

[3] Churcher, N. I., and Shepperd, M. J. Comments on 'A metrics suite for object oriented design'. *IEEE Transactions on Software Engineering 21*, 3 (Mar. 1995), 263–265.

[4] Constantine, L. L., and Yourdon, E. *Structured Design.* Prentice Hall, 1979.

[5] Danicic, S., Harman, M., and Sivagurunathan, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[6] Fenton, N. E. *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1990.

[7] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability 5* (Sept. 1995), 143–162.

[8] Harman, M., Danicic, S., Sivagurunathan, B., Jones, B., and Sivagurunathan, Y. Cohesion metrics. In 8^{th} *International Quality Week* (San Francisco, May 29th – June 2nd. 1995), pp. Paper 3–T–2, pp 1–14.

[9] Harman, M., Danicic, S., Sivagurunathan, Y., and Simpson, D. The next 700 slicing criteria. In 2^{nd} *UK workshop on program comprehension* (Durham University, UK, July 1996), M. Munro, Ed.

[10] Harman, M., Simpson, D., and Danicic, S. Slicing programs in the presence of errors. *Formal Aspects of Computing 8* (1996), 490–497.

[11] Harrison, R., Samaraweera, L. G., Dobie, M. R., and Lewis, P. H. An evaluation of code metrics for object-oriented programs. *Information and Software Technology* (May 1996).

[12] Henry, S., and Kafura, D. Software quality metrics based on inter–connectivity. *Journal of Systems and Software 2*, 2 (1981), 121–131.

[13] Henry, S., and Kafura, D. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering 7*, 5 (1981), 510–517.

[14] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), pp. 25–46. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[15] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12* (1990), 26–61.

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

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

[18] Lyle, J. R., Wallace, D. R., Graham, J. R., Gallagher, K. B., Poole, J. P., and Binkley, D. W. Unravel project.

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

[20] Ott, L. M., and Thuss, J. J. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium* (1993), pp. 78–81.

[21] Shepperd, M. *Foundations of software measurement.* Prentice Hall, 1995.

[22] Tip, F. A survey of program slicing techniques. *Journal of Programming Languages 3*, 3 (Sept. 1995), 121–189.

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